CAOPLE: A Programming Language for Microservices SaaS

Chengzhi Xu[(1)], Hong Zhu[(2)], Ian Bayley[(2)], David Lightfoot[(2)], Mark Green[(2)] and Peter Marshall[(2)]

[(1)] *School of Computer Science, Hubei University of Technology, Wuhan, China. Email: xcz911@gmail.com*
[(2)] *Dept. of Comp. & Comm. Tech., Oxford Brookes Univ., Oxford, UK. E-mail: hzhu@brookes.ac.uk*

Abstract

The microservices architecture is widely regarded as a promising approach to service-oriented systems. However, developing applications in the microservices architecture presents three main challenges: (a) how to program systems that consists of a large number of services running in parallel and distributed over a cluster of computers; (b) how to reduce the communication overhead caused by executing a large number of small services; (c) how to support the flexible deployment of services to a network to achieve system load balance. This paper presents a programming language called CAOPLE and reports the implementation of the language on a virtual machine called CAVM-2. The paper demonstrates how this approach meets these challenges.

Keywords *-- Service-oriented software; Microservices architecture; Virtual machine; Programming languages, Cloud computing, Parallel programming model; Agent orientation.*

1 Introduction

Microservices (MS), as a software architecture style, has sprung up over the last few years, and attracted more and more attention from researchers and companies [1]. It is widely considered as the best way to structure SaaS systems [2].

Since the early years of service-oriented architectures, the monolithic architectural style has been the dominant approach to structuring web-based applications [3]. Such an application runs in a few processes of coarse granularity. Each process implements a large block of functionality, such as receiving service requests, executing some business logic, retrieving and updating data from a database, and sending out response messages. To startup companies, the monolithic style may be a good choice. However, as user numbers increase, the domain logic grows in complexity and the databases expand. The monolithic style becomes less and less suitable for large-scale applications.

One of the main barriers to the development of service oriented applications is the scalability problem. Among many dimensions of scalability, horizontal scaling plays a crucial role in cloud computing, which means replicating multiple identical copies of the processes of the application behind a load balancer [4]. This kind of scaling can be too expensive for a monolithic architecture because if one of the components overloads the whole process has to be duplicated. Another barrier is system updating. When a system grows in scale and complexity, making changes to an application in a monolithic architecture become problematic for programmers and customers, because redeploying a new version means to restart the server, which will take a long time. This is particularly problematic when the application is located in the cloud, where a large customer base cannot tolerate being off line frequently.

Facing these challenges, Lewis and Fowler pointed out that the MS architecture is a promising solution to meet the requirement of cloud computing and big data [1]. Instead of building a single monstrous, monolithic service, the idea is to split a SaaS application into a set of smaller, interconnected services [5]. Lewis and Fowler defined MS as an architectural style in which a single application consists of "*a suite of small services, each running in its own process and communicating with lightweight mechanisms*". These services are "*independently deployable by fully automated deployment machinery*" [1].

MS address the above barriers by decomposing a system into a large number of fine-grained services that are connected together through a communication mechanism and supported by a deployment mechanism for replicating and relocating MS in a cluster of servers. One MS's collapse or going off line will be less likely to have devastating effect on the whole system. A change that takes place in one MS only needs the MS to be rebuilt, tested and redeployed. It often has less disruption to the whole system. The update process could be much faster such that customers do not even recognize that the service is off line, because a MS is very simple and lightweight.

However, developing service-oriented applications in the MS architecture faces three challenges. The first is how to program a large set of fine-grained services running in parallel. The second is the need for a lightweight facility to enable MS to communicate with each other. And, finally, it needs a deployment mechanism and facility that enable services to be deployed flexibly and uninterruptively. This paper proposes a programming language solution to all these problems.

The remainder of the paper is organized as follows. Section 2 reviews related work about MS. Section 3 introduces a novel programming language called CAOPLE. Section 4 presents a virtual machine called CAVM-2 on which the CAOPLE language is implemented. Section 5 gives a few examples of CAOPLE programs to illustrate its program-

ming style. Section 6 concludes the paper with a discussion of future work.

2 Related Work

Chong and Carraro proposed a maturity model of SaaS applications and classified them into four levels according to their architectures [6]. At Level 1, which is the lowest in maturity, each customer has their own version of the application customized and tailored to meet their requirements, and their application runs on a dedicated server. At the second level, the vendor provides the same copy of a configurable application to all customers. Each customer runs one instance of the software on a dedicated server and configures their application instance to meet their requirements. At the third level, all tenants share one instance of the software, and the vendor provides a virtual separation between the tenants. In this case, the configuration and customization of the software is achieved through using metadata. At the highest level, multiple instances of the software serve the tenants through a load balancer. As Chong and Carraro pointed out [6], the evolution of SaaS applications from lower level to higher level brings with it improvements in scalability, efficiency and configurability. The MS architecture can be regarded as a maturity level on top of Chong and Carraro's level 4. In the MS architecture, instead of duplicating the identical instances of the whole SaaS application, only the components of a SaaS application, i.e. the fine-grained small-scale services called *MS*, are duplicated and distributed to different servers. This further improves the scalability and efficiency, but requires ever more flexibility for configuration and customization.

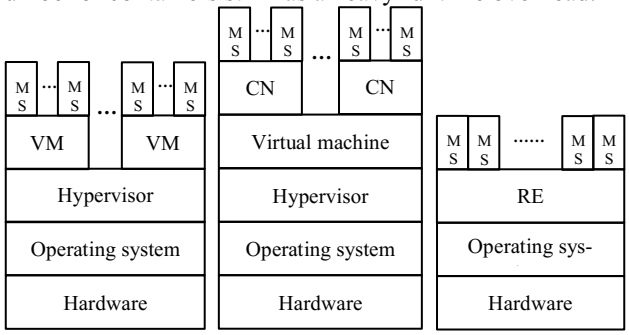
As moving to cloud computing and big data having become the main tendency in recent years, the IT industry urgently needs a new approach to meet the challenges of Big SaaS [7]. Many companies and organizations have adopted MS, such as Amazon, eBay, and Netflix. However, problems remain in how to develop applications in the MS architecture and how to improve the efficiency of running a large number of MS in a cluster of servers.

For a long time, the virtual machine (VM) has been the main protagonist of cloud computing. A VM is a heavyweight solution. It virtualizes hardware, such as disk storage, memory, CPUs and networks, and gives the customer physical separation [8]. Its main advantage is flexibility. It enables services to be executed on a VM regardless of the operating system and hardware platform beneath it. However, a VM consumes system resources, so that it becomes inefficient to deploy many VMs on one server. In other words, the VM has become a bottleneck for MS.

Container is a new technology, which overcomes the shortage of VMs. Compared with VM, container is a kind of lightweight virtualization, which shares system memory, system processes and the file system, but provides separation to customers [9]. Thousands of containers can be deployed on one host easily and can restart quickly.

In the last two years, many kinds of containers have emerged. Docker is an open-source project that automates the deployment of applications inside containers by providing an additional layer of abstraction and automation of operating-system-level virtualization on Linux [10]. Many companies have adopted Docker. Amazon has published Container Service (Amazon ECS), which is a highly scalable and fast container management service that makes it easy to run, stop and manage Docker containers on a cluster of Amazon EC2 instances [11]. Google has published an open-source platform, Kubernetes, for automating deployment, scaling and operations of applications on Linux containers across clusters of hosts [12]. Oracle provides Solaris Zones as Container for Oracle Solaris 11 OS [13]. Microsoft has also started to work on the container technology.

In summary, existing work on supporting MS has been focused on the deployment mechanism that enables MS to be easily duplicated and relocated on different servers to achieve system efficiency. These container technologies are more efficient than virtual machines because running thousands of containers on one virtual machine has less runtime overhead than running thousands of virtual machines, as shown in Figure 1(a) and (b). However, running a large number of containers still has a heavy runtime overhead.



(a) VM (b) Container (c) Proposed solution

VM: Virtual Machine, CN: Container, RE: Runtime Environment

Figure 1. Architectures of MS Technology

Our proposed solution is to develop a programming language that constructs service-oriented systems with MS as the basic building blocks that can be easily deployed to different servers with a light-weight runtime environment similar to the Java Virtual Machine. In this approach, the MS can be run across different platforms without the need for virtual machines and containers, as shown in Figure 1(c). Therefore, the runtime overhead can be further reduced.

In this paper, we present a programming language called CAOPLE and its runtime environment in the form of a language virtual machine called CAVM-2. We demonstrate how the language supports programming SaaS applications that consist of a large number of services executing in parallel and communicating with each other, and how to deploy the MS seamlessly in a cluster of servers.

3 The CAOPLE Programming Language

This section presents a novel programming language and

discusses how it supports programming MS.

3.1 The Conceptual Model

CAOPLE stands for Caste-centric Agent-Oriented Programming Language and Environment [14]. Here, agents means service providers just like in the real world where estate agents provide services in buying and selling properties, and travel agents provide services in buying and selling air-tickets. It is worth noting that agents are not just service providers, they can also be service requesters. They are the basic building blocks of service-oriented systems in our agent-oriented approach. Our approach is a purely agent-oriented one because the agent is the only building block of a software system.

In the literature, the word "service" in service-oriented architectures, and similarly its corresponding notion of "microservice" in the MS architecture, has two meanings. First, a service is the functionality provided by a computer system and delivered to the users [15]. Second, the word service also refers to the computational entities that provide the services in the first sense. Here, we separate these two concepts by using the word service only to refer to the functionalities that a computational system provides, while the computational entities that provide such functionality are called "agents". By doing so, an analogy between service-orientation and object-orientation can be made clearly. Consequently, our programming language bears a similarity to OO programming languages.

In particular, agents can be classified in a similar way that objects in OO programming languages are classified. In our CAOPLE language, the classifier of agents is called *caste*. Therefore, agents are runtime instances of castes just like objects are runtime instances of classes, while a caste is a template for agents just as a class is a template for objects.

Moreover, by separating these two meanings of the word service, it facilitates the study of service-oriented software architectures as concerned with the composition of entities into a certain structure. This is particularly important in the study of MS architecture.

In the context of the MS architecture, the notion of "MS" also bears two further meanings: first, MS are identical copies of a service where each copy is a runtime computational entity. Second, a MS is a template from which instances can be generated and deployed to different servers. As we will see below, our agents are autonomous, encapsulating data, operations and behavior rules, executing in parallel and cooperating with each other via asynchronous communications through a set of well-defined communication channels. These characteristics are exactly what MS services are when the word bears the meaning of computational entities. Castes, on the other hand, capture the meaning of a template from which runtime instance are instantiated. Each caste can have a number of instances. These instances can run on the same machine, but, more often are spread over a computer network. Therefore, the notion of agents and their classifier caste provides a perfect conceptual model of MS.

The following is the Hello world example of caste declaration in the CAOPLE language.

```
caste Peer() {
   action speak(words: string){print words;}
   init {speak("Hello world");}
   body {}
}
```

The above caste declaration defines the structure of agents called `Peers`, which are capable of performing an action called `speak`. When an agent of the caste `Peer` is created, it will take an action of `speak("Hello world")`. The affect of such an action is to print out the string "`Hello World`" on the console and to generate an event of `speak` with parameter "`Hello world`".

Like OO programming languages, CAOPLE also provides an inheritance mechanism to enable "polymorphism", i.e. agents with a number of variant functions, internal structures and behaviors. The following are two sub-castes of `Peer`, which reply to a `Peer` agent's `speak("Hello world")` action with different responses: one says "`Welcome`", the other says "`Good morning`".

```
caste Peer_WC() extend Peer {
   observe all x in Peer;
   var str: string;
   init {super();}
   body{
      when exist x : speak("Hello world"){
         speak("Welcome");}}
}
caste Peer_GM() extend Peer {
   observe all x in Peer;
   init { super();}
   body{
      when exist x: speak("Hello world"){
         speak("Good morning");}}
}
```

The *observe*-causes in the above castes declare communication channels so that an agent of `Peer_WC` or `Peer_GM` listens to the events of all agents of the `Peer` caste, including agents of `Peer_WC` and `Peer_GM`, due to the inheritance relationships.

Similar to OO, an agent may have a number of other agents (i.e. MS) as its components. The following is a caste declaration that defines a system called `Community` that creates and instantiates `100` agents of `Peer_WC` caste and `100` of agents of `Peer_GM` caste.

```
caste Community() {
   observe all x in Peer;
   var cnt: int;
   init{
      cnt:=0;
      for (i:=0 to 99){ create Peer_WC();};
      for (i:=0 to 99){ create Peer_GM();}}
   body{
      when exist x: speak("Hello world"){
         cnt:= cnt+ 1;}
   }
}
```

An important difference between caste and class is that an agent can join into a caste and quit from a caste dynamically, even suspending and resuming its caste membership at runtime. Moreover, an agent can be a member of multiple castes and a caste can extend multiple castes.

3.2 Overall Structure of CAOPLE Programs

The overall structure of a CAOPLE program consists of a set of caste declarations plus a set of data-type declarations, which serve the purpose of defining standards for the format of data exchanged between agents. Our approach is caste-centric in the sense that every agent must be an instance of one or more caste.

In general, a caste declaration has the following syntax in EBNF, where terminal symbols are in bold font.

```
CasteDec ::=
   caste CasteName[(Parameters)][Inheritances] {
     {EnvDec | StateDec | ActionDec}
     InitPart
     BodyPart
   }
Parameters ::= { ParamID : TypeName,}
Inheritances ::= extend {CasteName,};
EnvDec ::=
   observe (all | const | some ) AgentVar
     in CasteName ;
StateDec ::=
   var (public|internal) StateVar: TypeName ;
ActionDec ::=
   action (public|internal) ActionId ([Params])
     {Statements};
InitPart ::= init { Statements }
BodyPart ::= body { Statements }
```

An action declaration starts with the keyword `action` followed by an optional list of parameters and the body, which is a sequence of statements. An action can be modified by a visibility keyword `public` or `internal`. When an agent performs the action, the associated body statements will be executed and when it finishes the action, an event will be generated with the action name as the event name and the value of the parameters as the event's parameter. If the action is public, the event will be issued to the environment and delivered to those agents who observe the behavior of this agent. However, when the visibility of the action is `internal`, the event will not be issued to the environment outside the agent. When the visibility modifier is missing in an action declaration, the default is `public`.

A state declaration starts with the keyword `var` followed by a variable identifier and its data type. It can also be modified by a visibility keyword `public` or `internal`. The former declares a state variable that other agents can observe; while the latter declares an internal state variable that other agents cannot observe. Note that for both internal and public state variables, only the agent itself can change their values. As other agents cannot modify an agent's variables, the write-write type of data racing is eliminated. When the visibility modifier is omitted, the default is `public`.

The *observe*-clauses in a caste declaration define the ports of the communication channels that the agent listens to by defining the set of agents it observes. These can be established flexibly and updated at runtime; see Table 1.

For example, the communication channels between different types of `Peers` in the above examples are statically declared in the caste declaration that an agent of `Peer_WC` can listen to events from all visible actions of all agents of the caste `Peer` and its subcastes.

Table 1 Various Formats of Environment Declarations

Format	Meanings
observe all x in C;	The agent observes all the agents in the caste C except itself if the agent is also a member of C.
observe some x in C;	When an agent A in caste C is bound to the variable x, the agent observes agent A.
observe const A in C;	The agent will observe the agent A in caste C. Here, A is a constant agent ID.

The *init*-clause consists of a sequence of statements, which are executed once when the agent is created or when an existing agent joins the caste. It serves the purpose of initialization of the agent's state variables.

The body of an agent is a sequence of statements. It is executed repeatedly after the initialization until the agent finishes its casteship, i.e. when it is destroyed or it quits the caste. For example, in the `Peer_WC` caste, the *when* statement is executed as if it is inside an indefinite loop so that it can reply to every `Peer` agent's action of `speak("Hello world")` with an action `speak("Welcome")`, rather than just reply to one `Peer` agent. This again gives the event-driven feeling of the code.

3.3 Communication Mechanism

As pointed out in Section 1, the communication mechanism plays a crucial role in the support of service-oriented applications in the MS architecture. CAOPLE provides a set of language facilities that support flexible and secure, but lightweight, communications for event-driven parallel and distributed programming that are transparent to the network structure.

CAOPLE's language facility supports the following communication and concurrent programming mechanisms.

- *Subscribe-and-Publish*

The observe-clauses in a caste declaration actually define the communication ports that an agent listens to. It is similar to the *subscribing* part of the widely used subscribe-and-publish communication mechanism. Such a port can be a one-to-one port that is dedicated to observing a particular agent if it is in the form of "*observe const A in C*". It can also be a port that is configurable to listen to an agent of a given caste, if it is in the form of "*observe var x in C*". The agent can assign the variable to different agents at runtime, thus changing the subscription. Moreover, it can be a port to listen to all agents of a caste, if it is in the form of "*observe all in C*". Because the agents can change their caste membership dynamically, such a definition of the environment that an agent observes is not static, not closed, but also not completely open. It allows type checking of the uses of such ports, for example, when they are used in the *when*-statement and *till*-statement.

- *Event-Driven Computation*

An event is generated and a "message" is sent out by an agent when it performs a public action. The basic syntax of performing an action is the same as a procedure call as shown in the `Peer` example. This promotes a programming style in which the developers simulate how people collaborate with each other through taking actions. Such an event

can be broadcast to a large number of agents in the system to trigger a variety of different reactions. For example, in the `Peers` example, when one agent takes the action `speak("Hello world")`, a public event will be generated and delivered to all the agents of `Peer`'s subcastes who observes this agent's behavior. Consequently, they will respond by saying either "`Welcome`" or "`Good morning`". The `Community` agent will also react to this event by increasing its state variable `cnt` by 1. Such a mechanism is at a high level of abstraction.

CAOPLE provides two statements to enable event-driven computation. The first statement is the *when*-statement, which has the following syntax structure.

```
WhenStatement ::=
  when { scenario { statements } ;
     [else { statements }] }
scenario ::=
   (AgentExp | exist AgentVar) in CasteName :
     ActionID([Params])
```

The execution of a *when*-statement checks whether the scenarios are true. When a scenario is true, the corresponding statements of the scenario are executed. If no scenario is true, the body statements of the *else* branch are executed. If the *else*-clause is omitted, the statement is skipped.

Here, CAOPLE allows two forms of scenarios. In the first, a scenario is specified in the form of "*A in C: Act(params)*" meaning the particular agent *A* in caste *C* takes the action *Act(params*). In the second, the scenario is specified in the form of "*exist x in C: Act(params)*", where *x* is a variable ranging over the caste *C*, means that if there is an agent in the caste *C* that takes the action then the variable *x* will be bound to the agent that took the action in the execution of its body statements.

The second statement that CAOPLE provides for event-driven computation is the *till*-statement, which has the following syntax structure.

```
TillStatement ::=
  till { scenario { Statements } ; }
```

The *till*-statement is very similar to the *when*-statement, but when no scenario is true, instead of skipping the body statements, the till-statement will wait until one of the scenarios becomes true, then execute the corresponding body.

- *Prevention of Data Race*

As mentioned in Section 3.2, an agent's state variables can only be modified by the agent itself. Because each agent is one thread, this prevents write-write type of data race.

To help prevent write-read type data racing, we redefined the semantics of the *with*-statement in traditional programming languages such as Pascal.

```
WithStatement::= with expr { statements }
```

where the expression *expr* evaluates to a value *d* of a structured data type. Although the body statements of a with-statement apparently modify the elements of *d,* the actual update of the value of *d* will only take place when the execution of the body-statements finishes. Therefore, the changes to the various elements of a structured data *d* inside the *with*-statement will not be interrupted by reading its value. Thus, the atomic property of updating structured data can be ensured.

- *Control of Communication Security*

In the introduction to the CAOPLE language in the previous sub-section, we have already seen that each action and state variable can be declared to be either public or internal. Only the message/event associated with performing a public action or the value of a public state variable are observable by other agents in the environment. To support the control of communication security, an action statement can specify a restriction on the target agents that the event is to be delivered to. The general syntax structure is as follows.

```
actionStatement::=
   actionID([params]) [to target]
target ::=
  {(all in casteName | AgentExps in casteName),}
```

This not only limits the range over which events are delivered, but also reduces the communication cost.

3.4 Deployment Mechanism

An automated deployment mechanism is one of the key features of container technology. It aims at enabling services to be placed on the servers dynamically. Our agents can be created on any computer in the network. Figure 2 below illustrates the process of how castes are compiled, released and instantiated.

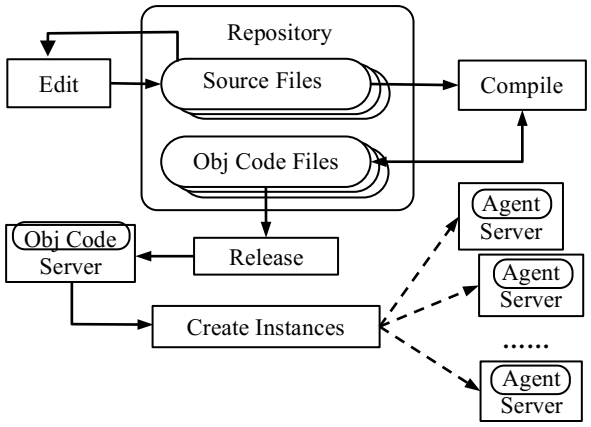


Figure 2 CAOPLE's Compilation, Release & Instantiation Process

Here, castes can be gradually added into the system and agents created on various machines at different times. This makes CAOPLE differ from traditional programming languages such as Java that follow a linear process of compilation, link and installation.

Figure 3 is a screen snapshot of a simple development, deployment, and debugging tool that enables CAOPLE source code to be edited and compiled into object code, then the object code loaded to servers, and its instances, i.e. agents, created on various machines in a cluster of computers and integrated into the system while it is already up and running. Thus, the agents are executed, tested and debugged. It is not just a deployment tool, but supports the whole lifecycle of programming MS in a continuous integration and continuous evolution agile process.

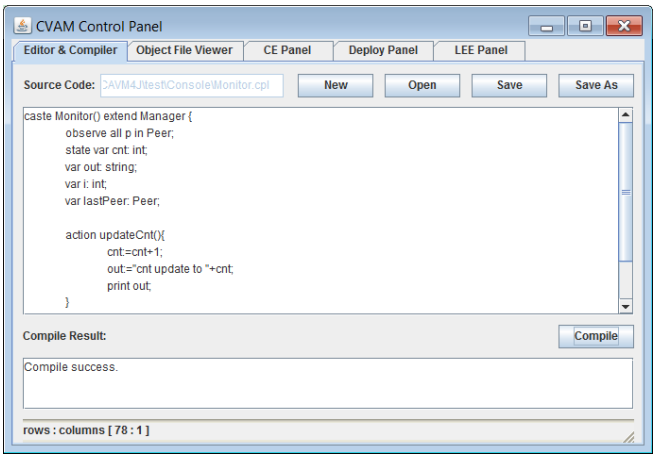


Figure 3 Screen Snapshot of CAOPLE's Simple Deployment Tool

Moreover, not only can deployment be done through such a tool, or by using a script language or command line commands, but also through the execution of code written in the same programming language CAOPLE, as shown in the example of `Community`. The general form of agent creation statement is in the form of

```
AgentCreationStatement ::=
  create [AgentVar of] casteName ([params])
    [@ locationExp]
```

where the *locationExp* evaluates to a string representing a location in the computer network such as an *ip* address. The parameters of an agent-creation statement are used to initialize the agent. The execution of an agent-creation statement will create a new agent on the machine at the location and assign the agent ID to the agent variable, which is used to refer to the agent, for example, to establish communication channels.

In addition to the agent-creation statement, CAOPLE also provides the following set of agent-operation statements that operate on the status of the agents.

```
JoinStatement ::= join casteName ([params])
QuitStatement ::= quit [ casteName ]
SuspendStatement ::= suspend casteName
ResumeStatement ::= resume casteName
EvolveToStatement::=
  evolve [casteName] to casteName([params])
```

These statements change the agent's caste membership state. The *suspend*-statement suspends the agent's casteship to a caste. The agent will still hold the values of the state variables of the caste, but not execute the body statements until it resumes the casteship. Executing a quit-statement, means the agent quits from the caste. Consequently, it will lose all components of the caste, including the state variables, actions, environment, and the body. The evolve-statement is logically equivalent to first quit the caste then move to another caste, but it will preserve the internal states declared in the common super-castes.

4 The Virtual Machine CAVM-2

CAVM-2 stands for CAOPLE Virtual Machine version 2. It is an improved version of CAVM reported in [16]. This section describes the architecture and function of CAVM-2. It has been implemented in Java. A substantial subset of the CAOPLE programming language has been implemented through compilation of source code into object code interpreted by the CAVM-2 virtual machine. This section gives a brief description of the architecture and operation of the virtual machine.

4.1 The Overall Architecture of CAVM-2

CAVM-2 is the runtime environment of CAOPLE programs just as the JVM is the runtime environment for Java programs. Its overall structure is shown in Figure 4.

Like its previous version [16], CAVM-2 consists of two parts: a *Communication Engine* (CE) and a *Local Execution Engine* (LEE). As their names indicate, the CE is responsible for the communications and the LEE is responsible for execution of object code instructions. In a cluster of computers, a complete CAOPLE runtime environment must have at least one CE, but may have many CEs running on different machines. Similarly it must have at least one LEE, but may have many LEEs running on different machines. The numbers of CEs and LEEs depend on the needs of the application. A CE and an LEE can also run on the same machine, thus a single machine can also form a complete runtime environment to execute CAOPLE programs. For example, in Figure 4 CE_1 and LEE_1 are within the same dotted rectangle. This means they are located in computer C_1. However, CE_2, LEE_2 and LEE_k are located in different computers.

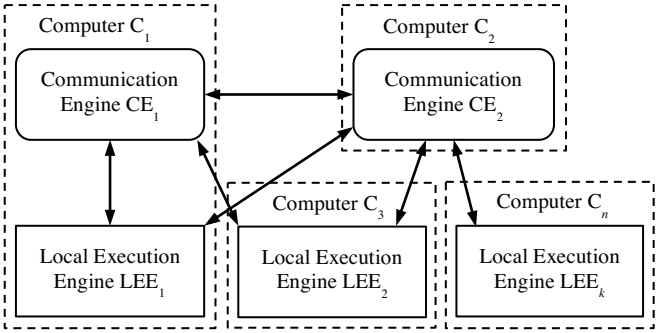


Figure 4. Overall Architecture of CAVM-2

Messages generated by LEEs are transmitted to a CE first, and then transmitted to other LEEs or CEs. A caste's object-code file is deployed on one of the CEs in the cluster. The LEEs download the object file from the CE when the first agent of the caste is created on the machine. Each CE serves as a message distribution center and a caste repository. A LEE uses the object code of a caste as a template to create agents, and executes the object code. It maintains a list of agents running on the LEE. In this sense, LEEs are agent drivers and containers.

4.2 Message Format and Semantics

The communication facility of the CAOPLE programming language is supported by CAVM-2. It implements a lightweight communication mechanism.

Each message generated and processed by CAVM-2 consists of four parts, as shown in Figure 5.

SRC_IP	DEST_IP	INSTR	CONTENT

Figure 5. Message format

SRC_IP is the source-machine IP where the message is generated. DEST_IP is the destination-machine IP, where the message will be sent to. If DEST_IP is a broadcast IP, the message will be broadcast in network accordingly. INSTR is an instruction code, which defines how the message should be processed. CONTENT is the contents of the message. Table 2 lists all the instruction codes and gives their meanings. All messages are represented as JSON objects in the form of name-value pairs. Table 3 lists the name and the meanings of the message contents.

4.3 The Communication Engine

Shown as Figure 6, a CE has three main functional modules, namely, *Receiver*, *Sender* and *CE Message Processor*. It operates on four main data structures, namely, *Receive Queue*, *Send Queue*, *Caste List* and *Agent Info List*.

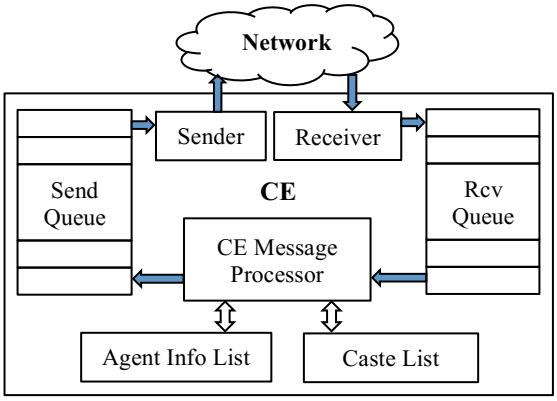


Figure 6. The structure of CE

When the *Receiver* collects a message from the network, the message will be added into the *Receive Queue*. The *CE Message Processor* reads messages from the Receive Queue one by one and removes each one after processing it. Each message is processed according to the instruction code of the message, and sometimes the message contents as well. The following describes how the CE processes some key messages.

(1) *Store A Caste into Caste List*

When the instruction code is *DEPLOYMENT_CASTE*, the message content contains the object code for a caste. The CE will store the object code into the caste list. If there already exists an object code for the caste, the old object code will be replaced by the new one.

(2) *Download A Caste from Caste List*

If a LEE wants to download a caste from a CE, it will send a *REQUEST_CASTE* message to the CE where the object code is stored. After receiving the request, the CE will retrieve the object code of the caste from its Caste List and wraps it in a reply message. Then, the message will be sent back to the LEE.

Table 2. Instruction Code of Messages

Name	Meaning
DEPLOYMENT_CASTE	Deploy caste to CE or LEE
DEPLOYMENT_CASTE_REPLY	Reply to message sender after deployment
DETECT_CE	Detect CE in network
DETECT_CE_REPLY	Reply to message source after detecting
REQUEST_CASTE	Request caste from CE
REQUEST_CASTE_REPLY	Reply the request to message source
CREATE_AGENT	Telling CE an agent is created on LEE
CREATE_AGENT_REPLY	Reply to LEE that CE has known the creating action
OBSERVE_STATE	Want to visit one agent state value
OBSERVE_STATE_REPLY	Reply state value to message source
INVOKE_ACTION	Agent telling network that it has invoked an action
INVOKE_ACTION_RELAY	*Relay INVOKE_ACTION* message via CE
DELETE_AGENT	Telling CE an agent is deleted on LEE
DELETE_AGENT_REPLY	Reply to LEE that CE has deleted the agent
CREATE_AGENT_REMOTE	Create agent on remote LEE
CREATE_AGENT_REMOTE_REPLY	Reply to LEE that the agent has been created remotely

Table 3. Elements of Message Contents

Name	Meaning
MSG_UUID	Message ID
VARIABLE_NAME	*Variable name*
STATE_NAME	State name
STATE_DATA_TYPE	State data type
STATE_VALUE	State value
CASTE_NAME	Caste name
AGENT_UUID	Agent ID
ACTION_NAME	Action name
ACTION_PARAM_VALUES	Action parameters values
OBSERVE_AGENT_UUID	Agent ID who wants to observe others
OBSERVED_AGENT_UUID	Agent ID who is observed by others
OBSERVE_LEE_IP	IP of LEE where observing agent locates in
OBSERVE_CE_IP	IP of CE which observing agent communicates to
SOURCE_CE_IP	CE IP where the message comes from
SOURCE_LEE_IP	LEE IP where the message comes from
SOURCE_AGENT_UUID	Agent ID who create a new agent remotely
REMOTE_CE_IP	CE IP where is destination of remote_create message
REMOTE_LEE_IP	LEE IP where is destination of remote_create message
REMOTE_AGENT_UUID	Agent ID who is created remotely
AGENT_PARAM_LIST	Initial parameter list of creating an agent
AGENT_PARAM_NAME	Initial parameter name
AGENT_PARAM_VALUE	Initial parameter value
RETRANSMIT	The mark indicates the message is from LEE to CE or from CE to CE

(3) *Store an Agent's Information into The Agent Info List*

When an agent of a caste is created on an LEE, the agent's basic information will be wrapped up into a *CREATE_AGENT* message and sent to the CE where the caste is stored. When the CE receives such a message, the agent's information will be stored into its Agent Info List.

(4) *Delete an Agent's Information from Agent Info List*

When an agent of a caste C is deleted from a LEE, the agent's data must be deleted from the CE, too. The LEE sends a *DELETE_AGENT* message to the CE where the agents of caste C are maintained. When it receives such a message, the CE searches for the agent in its Agent Info List and deletes the agent from the list.

After finishing a task, the CE Message Processor will add a corresponding reply message into the Send Queue.

The sender component reads a message in the Send Queue one by one, sends it to the network and then removes the message from the queue.

4.4 The Local Execution Engine

A LEE consists of two components: The first is the local communication element (LCE) which processes the messages received from CEs and further distributes them to the agents running on the LEE. It also sends messages generated by its agents to CEs. The second, the logic-processing element (LPE) executes the object code instructions.

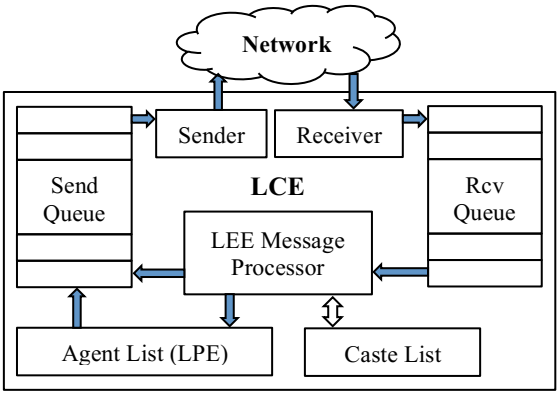


Figure 7. LEE's Message Processing Structure

A LCE has a similar structure to that of a CE, as shown in Figure 7. It also has three main functional modules, namely, *Receiver*, *Sender*, and *LEE Message Processor*. A LCE also operates on four main data structures, which are also called *Receive Queue*, *Send Queue*, *Caste List* and *Agent List*. When the Receiver collects a message from the network, it will be added into the Receive Queue. Similarly to a CE, the LEE Message Processor processes the messages in the Receive Queue one by one and removes them after processing. However, it processes messages differently as shown below:

(1) *Deploy A Caste's Object Code into Caste List*

When receiving a DEPLOY_CASTE message, the LEE will store the object code contained in the message to its Caste List.

(2) *Request A Caste's Object Code*

When an agent of a caste C is to be created, the object code of the caste is required. The LEE first searches for the object code in its Caste List. If it is not found, a message is generated and sent to a CE to request the object code.

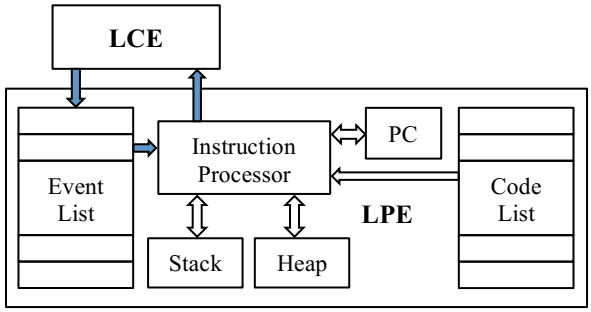


Figure 8. Agent message loop

The structure of a LPE is shown in Figure 8. Each agent is associated with an instance of a LPE, which executes as a thread. According to the value of PC, the Instruction Processor reads instructions from a caste's object code stored in the Code List and performs the corresponding processing of the data, such as retrieving data stored in the Stack or Heap at a location, or getting an event in the Event Queue, etc. Note that, the events are placed into the Event Queue by the LCE, and removed by the LPE. The LPE also generates new events/messages according to the instructions executed. In such a case, the message is placed in the Send Queue in the LCE for processing. There are two sets of instructions corresponding to taking a public action and visiting other agent's public states that generate events/messages and trigger communication. For the sake of space, the details of the instructions are omitted.

5 Examples

In this section, we give a few examples to demonstrate CAOPLE's programming style and its expressiveness in developing distributed applications.

5.1 Example 1: Service to Generate A Random Number

The following is a pair of castes where the first defines a service that generates a random number when requested, while the second defines the interface for making such a request.

```
caste RanIntGenerator(req : RanIntRequestor) {
  observe y in RanIntRequestor;
  var randomInt : int;
  action RanIntGenerated(rd:int){
    (* Omitted code for generating a random num-
ber and assigning it to rd. *)}
  init { y:= req;}
  body {
    when y: RequestRanInt()  {
      RanIntGenerated(randomInt);}
} }
caste RanIntRequestor(Loc: string) {
  observe x in RanIntGenerator;
  var myGenerator : RanIntGenerator;
  action RequestRanInt(){ }
```

```
  init{
    create myGenerator of RanIntGenerator(self);
    x :=  myGenerator;
  }
  body{}
}
```

The caste `RanIntGenerator` can be regarded as a server-side program, while the caste `RanIntRequestor` is similar to an API, thus the body is empty. The former implements the function of the service, while that latter defines the interface between the service provider and the service requestor.

`RanIntGenerator` in the above example only serves one particular service requestor, which is the parameter of the caste and initialized when it is created. The body of the caste contains a when-statement, which states that when receiving a request from its dedicated requestor, the agent generates a random number and send it back. The following is an example CAOPLE caste that uses the random number generator.

```
caste RanIntUser() extend RanIntRequestor {
  var str: string;
  var ranInt: int;
  init{
    super("192.168.0.9");
    ranInt:=0;
    RequestRanInt();
  }
  body{
    till x: RanIntGenerated(rcv ranInt);
    str := "Random number is: " + ranInt;
    create MessageBox("My Message", str);
  }
}
```

It extends the `RanIntRequestor`. Its initialization statements set the location where the generator is to be located and sends out a message to request a random number. In its body statements, the random-number user waits for the random number to be generated using the till-statement, and then displays it in a message box, which is an interface agent that displays a message box on the screen.

5.2 Example 2: Map-Reduce Style of Parallelism

We now demonstrate how to program a map-reduce kind of parallel computation in CAOPLE. We will use the same caste of `RanIntGenerator`, but write a new caste `RanIntMap` as follows.

```
  caste RanIntMap(Locs: string[]) {
    observe all x in RanIntGenerator;
    action RequestRanInt(){ }
    init{
      for (i:=0 to length(Locs)-1){
        create RanIntGenerator(self)@Locs[i];
      }
    }
    body{}
  }
```

The RanIntMap creates a number of instances of `RanIntGenerator` on a collection of servers given in the parameter `Locs`. The following code in CAOPLE sends a broadcast message requesting random numbers to all these generators, collects the results using a *when*-statement, sums up the values of these random numbers, and finally displays the sum in a message box.

```
  caste RanIntReduce() extend RanIntMap {
    var str: string;
    var myServers: string[];
    var total: int;
    var cnt:int;
    var ranInt: int;
    init{
      total := 0;
      cnt :=0;
      myServers:= … (* omitted*);
      super(myServers);
      RequestRanInt();
    }
    Body {
      when
        exist x: RanIntGenerated(rcv ranInt)
        { total := total + ranInt;
          cnt := cnt+1;
          if (cnt = length(myServers)){
            str:= "The sum is: " + total;
            create MessageBox("Message", str);
            quit; };};
    }
  }
```

5.3 Example 3: Chat Room

Chat room is a typical Internet-based application. Here, we write a simple chat room in CAOPLE to demonstrate its programming style and its expressiveness in writing distributed systems. Our chat-room consists of two castes: the `Chatter` defines the function of chatting, while `ChatGUI` implements a simple graphic interface as shown in Figure 9.

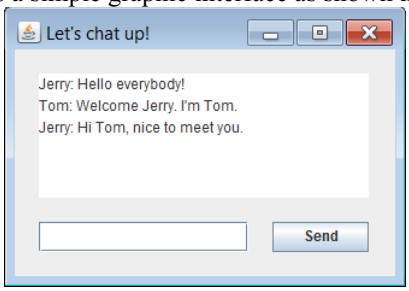


Figure 9 Chat Room GUI

```
caste Chatter(name: string) extend Peer {
  observe all x in Chatter;
  observe some myGUI in ChatGUI;
  var sentence: string;
  var chatterName: string;
  action showSentenceInGUI(x: string){}
  init{
    chatterName:=name;
    create myGUI of ChatGUI(self, chatterName);
  }
  body{
    when myGUI: enter(rcv sentence){
      sentence:=chatterName+": "+sentence;
      speak(sentence);};
    when exist x: speak(rcv sentence){
      showSentenceInGUI(sentence);
    }
  }
}
caste ChatGUI (cht:Chatter,nm:string) extend GUI {
  observe var ct in Chatter;
  observe var bt in Button;
  var internal content: string;
  var internal myFrame: Frame;
```

```
  var internal myTextArea: TextArea;
  var internal myTextField: TextField;
  var internal myButton: Button;
  action enter(x: string){}
  action addToTextArea(x: string){}
  action clearField(){}
  init{ /* create and initialize gui elements */
    ...
    bt := myButton;
    ct := chatter; }
  body{
    when bt:click(){
      content:=myTextField.text;
      enter(content);
      clearField();}
    when ct:showSentenceInGUI(rcv x){
      addToTextArea(x);
    }
  }
}
```

A user can start a chat with all the other users over the Internet by creating an agent of the caste `BootChatter`, which will create a chatter agent and a graphic user interface.

```
caste BootChatter(){
  observe x in InputBox;
  var IB: InputBox;
  var chatter: Chatter;
  var inputText: string;
  init{
    create IB of InputBox(
      "Please Enter Chatter's Name: ");
    till IB: click(rcv inputText) {
      create chatter of Chatter(inputText);
  }
  body{}
}
```

Note that, with a slight change of the caste `BootChatter`, we can generate a `Chatter` agent and a `ChatterGUI` agent on different machines, for example, the chatter on a server and the GUI on a notebook or other mobile device.

6 Conclusion

In this paper, we presented a novel programming language for programming MS. It is based on an agent-oriented conceptual model of software systems. Its language facilities are designed for creating, operating and managing large numbers of agents, which are instances of MS, and executing them in a distributed computer network. It provides strong support to SaaS in MS architecture by programming at a high level of abstraction. We have also designed a virtual machine called CAVM-2 for CAOPLE language and implemented it with Java. The architecture of our approach enables a large number of MS executing in a distributed environment with a low runtime overhead of communication, and high flexibility of agent deployment to the network environment.

The key features of the language demonstrated in this paper have been implemented. We are currently working on advanced programming-language features, thus as a complete library of GUI package and structured datatypes. We will also conduct experiments to evaluate the runtime overhead and optimize the efficiency of the virtual machine. Future research directions include how to support big-data applications by providing a library package that links to various NoSQL databases, etc.

Acknowledgement

The work reported in this paper is partially supported by EU FP7 project MONICA on Mobile Cloud Computing (Grant No.: PIRSES-GA-2011-295222), National Natural Science Foundation of China (Grant No. 61170025), and Natural Science Foundation of Hubei Province, China (Grant No. 2013CFB021).

References

[1] Lewis, J., and Fowler, M., *Microservices*, URL:http://martinfowler.com/articles/microservices.html#footnote-monolith, 25 Mar. 2014. (*Last access on 2 Nov. 2015*)

[2] High Scalability, *The Great Microservices Vs Monolithic Apps Twitter Melee*. URL: http://highscalability.com/blog/2014/7/28/the-great-microservices-vs-monolithic-apps-twitter-melee.html. Jul.28,2014.(*Last access on 2 Nov. 2015*)

[3] SSA Research, *Service Oriented Architectures*, Part 1 and 2, SSA Research Note SPA-401-068, 12 April 1996.

[4] Abbott, M. L., and Fisher, M. T., *The Art of Scalability: Scalable Web Architecture, Processes, And Organizations for The Modern Enterprise*. Pearson Education, 2009.

[5] Richardson C., *Introduction to Microservices*. URL: https://www.nginx.com/blog/introduction-to-microservices/ May 19, 2015. (*Last access on 2 Nov. 2015*)

[6] Chong, F. and Carraro, G., *Architecture Strategies for Catching the Long Tail*, Microsoft Corporation, April 2006. URL: https://msdn.microsoft.com/en-us/library/aa479069.aspx. (*Last access on 2 Nov. 2015*)

[7] Zhu, H., Bayley, I., Younas, M., Lightfoot, D., Yousef, B., Liu, D., Big SaaS: The Next Step Beyond Big Data, in *Proc. of IEEE CLOUD 2015*, Jun. 2015, p1131-1140.

[8] Letaifa, A. B., Haji, A., Jebalia, M., Tabbane, S., State of the Art and Research Challenges of New Services Architecture Technologies: Virtualization, SOA and Cloud Computing. *International Journal of Grid and Distributed Computing*. Vol. 3, No. 4, p69-88, Dec., 2010.

[9] Pahl, C., Containerization and the PaaS Cloud. *IEEE Cloud Computing*, Vol.2, No. 3, pp. 24-31, May-Jun. 2015.

[10] Merkel, D., Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, Vol. 2014, No. 239, p2, 2014.

[11] Amazon Products & Services, *Amazon EC2 Container Service*. URL: https://aws.amazon.com/ecs/. (Last access on 2 Nov. 2015)

[12] Brewer, E. A., Kubernetes And The Path To Cloud Native. *Proc. of SoCC'15*, 2015, pp167-167.

[13] Van Surksum, K., *Release: Oracle Solaris 11*. URL: http://virtualization.info/en/news/2011/11/release-oracle-solaris-11.html. (*Last access on 2 Nov. 2015*)

[14] 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), Chapter XLIV, pp679–691, 2006.

[15] Singh, P. M., and Huhns, N. M., *Service-Oriented Computing: Semantics, Processes, Agents*. Wiley, 2005.

[16] Zhou, B., Zhu, H., A Virtual Machine for Distributed Agent-Oriented Programming. *Proc.of SEKE'08*, pp.729-734, 2008.