International Journal of Grid Distribution Computing
Vol.7, No.6 (2014), pp.91-110
http://dx.doi.org/10.14257/ijgdc.2014.7.6.08

RUNES II: A Distributed Rule Engine Based on Rete Network in Cloud Computing

Rui Zhou, Guowei Wang, Jinghan Wang and Jing Li

School of Computer Science and Technology,
University of Science and Technology of China, Hefei, 230026, China
{rayzhou, weiking, heiswjh }@mail.ustc.edu.cn, lj@ustc.edu.cn

Abstract

In recent years, cloud computing has drawn more and more attention. Increasingly amount of systems and applications have been constructed in cloud environments, yet few researches of rule engine has been done. Rule engine technologies have been widely used in the development of enterprise information systems. These rule-based systems may encounter the problem of low performance, when a large amount of fact data are matched with these rules. Deploying rule engines in cloud environments can increase the capability and efficiency of these systems. In this paper, we propose an approach to implement rule engine based on a message-passing concurrency model in cloud computing. The approach can be extended conveniently and it can deal with extensive rules and facts efficiently. To improve the performance of the rule engine, an algorithm of allocation is proposed. A resource cost model is explored to make high efficient use of resources in cloud. In addition, we implement the rule engine system RUNES II in cloud platform and conduct experiments to show its performance.

Keywords: *Rule engine, Cloud computing, Rete algorithm, Allocation algorithm*

1. Introduction

In recent years, cloud computing has attracted increasing attention in industry and academia [15, 23]. In this area, the resources can be provided by services such as IaaS, PaaS, and SaaS. The appearance of cloud makes utilization of flexible computing sources for small companies and individuals possible. More and more systems and applications have been attempted to construct in cloud environment, such as e-commerce platform, customer relationship management (CRM), enterprise resource planning (ERP), supply chain management (SCM), and other systems. In this way, the cost of these systems can be reduced, and scales of these systems can be extended on demands. However, there are few researches of rule engine in cloud environments. Rule engine occupy a significant part in artificial intelligence since it is a good way of knowledge representation and reasoning. It has been widely used to build expert systems with many applications such as business management systems [16]. In the Big Data era, rule engine is facing a new challenge of increasing rules and data. Deploying rule engines in cloud environments can increase the flexibility and efficiency of these systems. Therefore, the aim of our work is to try to construct an efficient rule engine system in cloud environments.

The cloud computing has become a very important area in computing science. These resources can be used on demands dynamically. The cloud computing model is changing the ways how consumers can access to the technology solutions. Therefore, the cloud providers provide the infrastructure resources and computing capabilities as a service to the users. The

ISSN: 2005-4262 IJGDC

users can leverage a range of attractive features, such as resource elasticity, cost efficiency, and ease of management. The cloud computing model also compels the rethinking of economic relationships between the providers and the users based on the cost and the performance of those services.

A lot of researches on rule engine and expert system have been done in the past 20 to 30 years. A rule engine consists of three parts which are a knowledge base containing a set of facts, a rule base containing a set of rules (or productions) and an inference engine. When the rule engine starts, according to a certain match algorithm, the inference engine repeatedly tries to find rules in the rule base that could match facts that are accepted by the knowledge base. As a result we could get and execute desirable actions, which are specified in the matched rules. However, traditional rule engines are computationally expensive and slow. When the problems continue growing, the efficiency becomes much lower. In the age of cloud computing, the development of cloud offers efficient, stable and flexible computing resources to speed up rule engines. For the cloud environments, traditional models and architectures of rule engines are not appropriate. So, systems with these models are hard to transplant in cloud platform or these transplanted systems cannot take advantage of cloud computing. Then, the resource prediction of the rule engine is a significant issue in cloud computing. With the resource prediction, the rule engine can apply the appropriate resources and achieve better resource utilization in cloud. Furthermore, the allocation in cloud should be also considered to keep the load balancing. Meanwhile, the low degree of coupling among virtual machines is another target of allocation.

To resolve these problems, we propose an approach to enact a rule engine based on the message-passing concurrency model in cloud computing environments in this paper. The core algorithm is an improved Rete algorithm whose main idea is the Rete network, and we transplant the network into a cluster of virtual machines. It can extend conveniently and deal with extensive rules and facts efficiently. To improve the performance of the rule engine, a resource cost model is explored to make high efficient use of resources in cloud. Then, an algorithm of allocation is proposed. Finally, we implement the rule engine system RUNES II in cloud platform and conduct experiments to show its performance.

The contributions of this paper are as follows:

1. An innovative rule engine model based on concurrency and message-passing is explored to deal with large-scale rules and facts with cloud computing. In this model, nodes in Rete network are mapped to processes which can be distributed in different virtual machines. The rule engine is both feasible and efficient. Its performance is better than the existing rule engine products in large scale of rules. Moreover, the concurrency rule engine model can be deployed in elastic cloud environment easily.

2. Some models of resource cost are proposed to evaluate the virtual machine usage in cloud platform. The memory consumption and response time are discussed after a series of experiments. Cost models of them are concluded in some expressions. They provide the foundation of the determination of the quantity of virtual machines.

3. A process allocation algorithm is proposed to improve the efficiency of rule match. The algorithm can reduce passing messages and keep the balance among different computer nodes. Various experiments are conducted to evaluate the allocation algorithm. The results illustrate that our algorithm can optimize the distribution of the concurrency rule engine.

4. A system of the concurrency rule engine named RUNES II is implemented and deployed in cloud environment. A number of performance tests have been done to evaluate the advantage of the model and system. These results show the superiority of the model in cloud

environment. The ability of dealing with large-scale rules and facts is more powerful than current rule engine systems. On the other hand, the stronger expansion capability and higher resources utilization of the system are additional advantages compared to these rule systems implemented in traditional parallel way.

The rest of this paper is organized as follows. In Section 2, some related work in distributed rule engines is reviewed. Section 3 provides background of cloud rule engine, Rete algorithm and the message-passing concurrent model we use. Section 4 discusses the model and architecture of the rule engine in cloud platform. The algorithms of the allocation and node quantity are characterized in section 5. Section 6 describes the implementation of RUNES II in cloud. Section 7 outlines some experiments we have conducted. Finally, we conclude this paper in Section 8.

2. Related Work

Lots of research works have been done in past few decades. And one of the most important contribution was the creation of Rete by Forgy in 1982 [1], which hereafter inspired lots of its improvements and modifications including Treat [2], Rete/UL [3], Rete* [4] and [5]. But these could still not deal well with the situation when number of rules and facts become too large under the limitation of a single computer's capability. Some other works were about enhancing rule engines' ability of solving problems in a parallel or concurrent way by using multi-cores or distributed systems. A parallel asynchronous distributed production system was proposed in [6]. Another production system was implemented with message-passing computers in [7]. In [8], a concurrent architecture for production system was introduced. A parallel version of the Rete algorithm was proposed in [9]. In the past four years, a parallelizing CLIPS-based expert system was implemented in [10]. An Improved Rete Algorithm for distributed rule engine was put forward in [12]. Another enhanced design of a rule engine with structured query language is described in [13]. However, these researches either targeted on special hardware architectures thus lacked flexibility or only offered limited speedup. Therefore, these architectures do not suit the cloud environment.

More and more researchers pay attention to improve rule engine in emerging areas like web of things. The work in [19] improved Rete with Least-Recently-Used algorithm in context reasoning for web of things environments. The authors in [20] implemented a distributed expert system for classification of hadith based on cloud and SOA. A rule engine was also used in smart home in [21]. In that system, Rete algorithm was improved to implement an event engine. Another improvement to Rete was proposed for composite context-aware service in [22].

Existing researches of the rule engine in cloud computing are relatively limited. One recent work [11] implemented a rule engine on map-reduce based architecture. Although map-reduce technology is a popular way to deal with big data through batch processing in cloud computing, it could not match well with the rule engine involving lots of loops and iterations. As far as I know, except this work, little attention has been drawn to the rule engine in cloud environments so far.

3. Background

3.1. Rule Engine and Rete Algorithm

In a rule engine, one major problem is specified as a set of tuples containing facts that are some assertions, a set of rules related to the facts and a desired or final state. Facts are held in

the working memory and named working memory elements (WMEs), while rules are saved in the production memory (or rule base) and named production elements (PEs). Each rule is of the form IF-condition-THEN-action. The IF part of the rule (left-hand side or LHS) consists of a conjunction of condition elements. A condition element comes with a set of tests for attribute-value pairs of a WME, such as constant tests and consistency tests. The THEN part (right-hand side or RHS) specifies the actions that will be performed if the LHS is true.

During the run time, the rule engine executes three-phase cycle named match-resolve-fire repeatedly until either the goal is attained or no more rules can be executed. The matching phase matches LHS's of all rules against current WME's to find the set of satisfied rules named conflict set. The resolution phase selects one rule from the conflict set for execution in the next phase using criteria including chronological order, specificity and priority. In the fire phase, actions specified in the RHS of the selected rules are performed. If these actions lead to changes in the working memory such as addition, deletion or modification of WMEs, the match phase starts again.

The Rete algorithm is a highly efficient algorithm for the match phase. The elementary idea of the algorithm is to compile left hand side of rules into the Rete network and then perform match with creating and passing tokens along directed edges of the network. Tokens are actually tuples of working memory elements as partial instantiations.

There are basically four types of nodes forming the Rete network.

1. Constant-test nodes: These appear in the top layer of the network and perform intra-condition constant tests of condition elements. A constant test means checking whether a certain attribute has a given constant value.

2. Memory nodes: These include α-memory nodes and β-memory nodes. The former store intermediate results from constant test nodes as state while the latter store intermediate results from join-test nodes as state.

3. Two-input nodes: These appear in the low layer. Join-test node is a typical kind of two-input nodes, of which left-hand input is one β-memory node and right-hand input is one α-memory node. They perform tests for consistency satisfaction of distinct condition elements in two conditions. In more details, that whether the identical variable appearing in the two conditions is bound to the same value is tested.

4. Terminal nodes (or p-nodes): Each one appears at the bottom of the network and means the end of a rule. Actions in the RHS of the rule are stored in it, and are waiting for being triggered.

Sometimes one root node is introduced in the network as a start node. An example of the Rete network with rules and facts is showed in Figure 1.

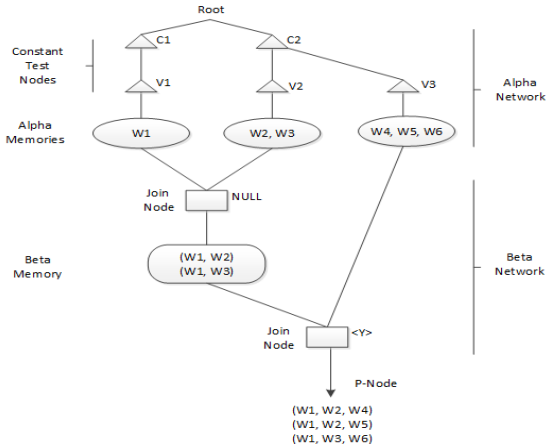


Figure 1. Rete Network

When the input of the Rete network makes changes to the working memory, they are introduced as tokens activating root nodes. They flow in constant-test nodes and generate tokens storing in following α-memory nodes if passing constant tests. Those new tokens then flow into two-input nodes and perform consistency tests. If passing, new tokens are generated and stored in β-memory nodes. This kind of flow goes on similarly. Finally tokens could flow into terminal nodes if all relevant tests are passed. Then corresponding rules are fire-able. Therefore the output of the network is the conflict set getting new fire-able rules. In brief, tokens flow through the Rete network from the top to the bottom and activate nodes along the route. The state of the algorithm is remained in memory nodes. The Rete network is a kind of data flow graph so that it has the potential to be parallelized in an easy and natural way.

3.2. Mapping the Rete Algorithm on the Message-Passing Concurrency Model

Message-passing concurrency model here refers to a parallel programming model. Its fundamental conception is the isolated lightweight process, which could be created thousands even millions easily and quickly on a single machine. Processes communicate with each other through massages. Upon receiving a message, the process executes pre-defined corresponding behaviours such as sending messages to other processes. The communication is mostly asynchronous and non-blocking. This implies that the sender could immediately continue its execution after sending a message without waiting for the message to be received. But the receiver is blocked since a message arriving earlier should be received and handled earlier than ones arriving later. Each process runs concurrently together with other processes to solve a concrete problem.

In our previous research, a message-passing model of Rete algorithm was proposed [24]. The message-passing model was designed in concurrent program architecture. In the architecture, a number of processes were maintained. In general, since the Rete algorithm is mainly about the Rete network consisting of different kinds of nodes, it is an intuitive way to map each node onto a single process in the message-passing model and view tokens passing through Rete network as messages. Then tokens' activating nodes could be viewed as sending and receiving of messages.

At first, when compiling rules into Rete network, our work is to create corresponding processes according to types of nodes. Each process has a unique identity and keeps a private state. The identity acts as an address to which other processes can send messages. The state describes necessary relevant information about the process. The only way to get the state for

other processes is through communication with messages so that each process can keep independent and isolated.

When new facts come, the rule engine starts to run. Then messages are created and transferred among processes. Processes behave according to receiving messages. Considering efficiency, messages of activation are designed to be asynchronous and non-blocking. Since each different process could work concurrently, we could make use of the power of multi-cores on a single machine if we could load these processes on cores evenly.

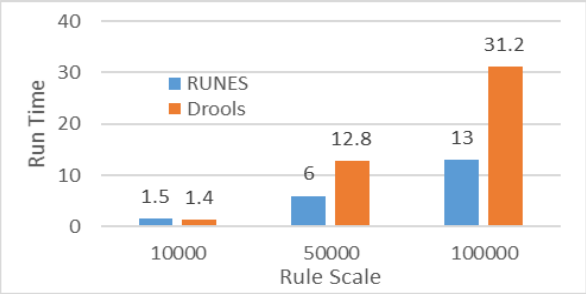


Figure 2. Run Time of RUNES and Drools for 1000 Facts

In this former work, the RUNES rule engine has been implemented. We benchmarked RUNES and Drools in an eight-core server with a focus on the time of matching all 1000 facts with different numbers of rules. Drools is one of the most popular business rule engine implementation and is implemented in Java [14]. The result is showed in Figure 2. We can see that the efficiency of RUNES is improved more significantly when the number of rules becomes bigger. This is mainly because that on a single machine the message-passing model can make well use of the power of multi-cores while the communication latency between cores can be neglected.

4. Model and Architecture of RUNES II in Cloud Environments

4.1. Distributing the Rule Engine in Cloud Environments

In RUNES II, the message-passing concurrency model of Rete is transplanted into cloud environment, and the processes in the Rete network are distributed in a cluster of virtual machines. Each kind of these processes in the Rete network is allocated evenly in every virtual machine as shown in Figure 3. A decentralized architecture was designed to distribute the rule engine among various virtual machines in cloud environment more easily. In more detail, in each virtual machine, we build an independent agent to maintain a full image of the rule engine. The image contains programs of all kinds of processes in the concurrency model. All the processes, which are allocated in the virtual machine, are controlled by the agent. As mentioned before, these processes corresponds the nodes in Rete network. They are constant test processes, α-memory processes, join processes, β-memory processes and P-node processes. In each virtual machine, these processes constitute a local subnet of Rete network for the rule engine. The subnet is constructed from a root-node process. These subnets in all of virtual machines constitute the whole Rete network of the rule engine. The Figure 4 shows an example of Rete network in cloud environments.

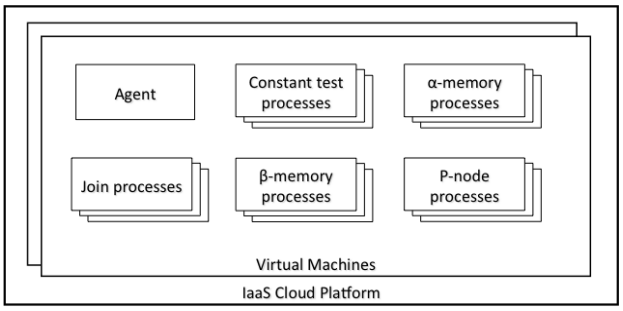


Figure 3. Processes in Virtual Machines

The Agent process is responsible for supervising processes in each virtual machine. Firstly, users can interact with each agent to add or delete rules and facts, start or pause the rule engine and so on. Secondly, when agents get new rules, they should compile the rules into Rete-node processes. Some of them belong to the local subnet of Rete, which is a part of the whole Rete network, and others do not. Processes in different subnets could be linked with each other naturally. From the perspective of linked processes, locations are transparent when they are communicating with each other. We achieve this by developing a global identity composed by the process local identity and its host agent identity, which could act as a global communication address in the function of sending messages. Thirdly, when one agent gets new facts, it could either make them enter the Rete network through local root process, send them to other virtual machines' root processes or both. But no matter through which agent the facts enter the Rete network, the matching could involve processes in other subnets if necessary. It means the match process is executed in the whole Rete network. Meanwhile activation and other kinds of messages are transferred between linked processed directly. Finally, a right agent should be chosen to reflect results of match-resolution-fire procedures to users.

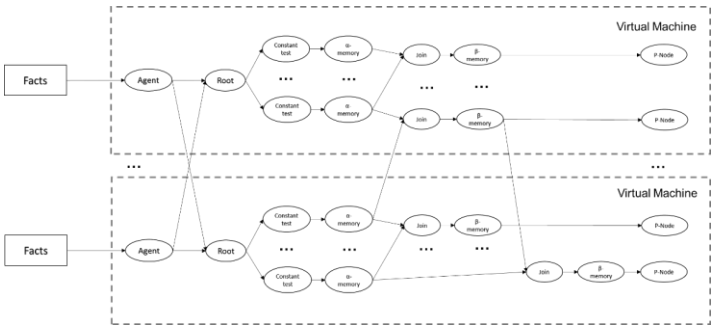


Figure 4. Rete Network in Cloud Environments

Overall, we introduce the agent to manage the distribution of the rule engine in a decentralized and easily scalable way. But the algorithm keeps running on the level of processes rather than agents because of the global identity, which hides the location. Since these processes are locations transparent when communicating, during the execution there is no great difference between a single-machine edition and a distributed edition from the perspective of algorithm if efficiency is not considered. It is worth to notice that the agent is the basic unit in our distributed system from the perspective of software. When agents are mapped onto the cluster of computing hardware, one computing node could load more than one agent if necessary.

4.2. Construction of Rules and Facts

In our rule engine, rules are compiled into Rete network consisting of linked processes. Each process represents a Rete node. These processes could be evenly allocated to different agents of the distributed engine. What is more, processes from one identical rule might be in different agents. More specifically, a rule has several conditions. Each condition means several constant-test nodes with α-memory nodes. Relationships between conditions bring β-memory nodes and join nodes. The allocation algorithm of these nodes will be described in next section.

As we discussed previously, all available agents in the system could get facts, and during the run time, facts exist in the form of working memories and tokens. Considering cost of communication, we definitely cannot transfer them between processes which might be distributed in different machines directly due to their sizes. As a result, we transfer fixed-length references instead. Each reference is globally unique for acting as a key to access a corresponding working memory or token. Moreover, working memories and tokens need to be accessed by processes in agents, which might be in different machines. This means that these data should have several copies stored in several agents to be accessed immediately and we should ensure the data consistency and isolation. Hence, we need an available distributed memory database.

In addition, memory nodes also store working memories and tokens as intermediate results. Considering convenience of management, these data are stored in the memory database instead of processes' states in a similar way.

At last, we introduced classification of facts. Each fact is assigned to a category. Then each rule also gets corresponding categories according to facts that match it. As a result, when we compile rules and allocate processes of one rule to different agents, we could save the allocation information as <category, agent list>. When we get new facts, we could simply check the saved records and allocate facts to related agents that are tagged with the right categories. If no relevant records are found, facts can be allocated to agents with less load or randomly. In this way, facts with the same category are allocated to the same agents. In consequence, not only can it avoid allocating facts to all available agents that then could lead to many unnecessary tests and related messages, but also it will reduce communication between different agents.

5. Optimization for Cloud Computing

5.1. Resource Cost Model

The resource prediction of the rule engine is a significant issue in cloud computing. With the resource prediction, the rule engine can apply the appropriate resources and achieve better resource utilization in cloud. The concurrency model of rule engine in cloud can deal with large-scale rules and facts in the rule engine. With the scale increasing, the resources of the system need to be increased in order to respond to these new coming facts effectively.

When the resource is not enough, it will cause the Rete network running out of memory and messages congest in computer nodes. This will lead to a sharp efficiency drop. In the other aspect, when the system occupies excess nodes, the load of each node will be too low. It will cause a decline in virtual machine utilization, which is unfavourable for the efficient usage of resources in cloud environments. Moreover, the excessive nodes will increase the message communication among them, which will be illustrated by the results of experiments. It will cause the decline of performance. Therefore, we need to consider the resource cost model for different loads.

To determine the node quantity, the main factor of the system is memory consumption, because of features of Rete algorithm. A lot of memory space is consumed to cache copies of previous matching results in order to cut down the amount of computation. Therefore, the memory consumption *M* is modelled with the following express.

$$M(R,F) = A(r) \cdot R + B(r,f) \cdot RF$$

In the model, the rule scale is denoted as *R*. The number of facts is estimated as *F*. Finally, *A* and *B* are coefficients between the rule system's scale and memory consumption. Values of *A* and *B* depend on the complexity of rules *r* and facts *f*. These can be evaluated with experiments. Some groups of experiments have been conducted to verify the model and evaluate these coefficients in our test rule set. In our experiments, *A* is 0.05(MB) and *B* is 5×10^{-5}(MB), while there are 4 conditions for each rule in average. For example, 1000MB of memory is required for 4000 rules and 4000 facts. In the actual experiment, this amount of rules and facts consumed 978MB of memory. Details of these experiments will be described in Section 7.3.

Based on the memory consumption model, the node quantity can be calculated in the form as following:

$$N = \left\lceil \frac{M}{S} \right\rceil$$

In the above expression, *S* is the free memory space of each virtual machine. In order to determine the node quantity, the scale of rule set and that of fact set are required. These values can be provided or estimated by users of the rule engine. Moreover, as the sizes of the rule and fact grow, the system can start new virtual machines to add memory space.

In our rule engine, the time complexity is irrelevant to the quantity of nodes. Therefore, if the memory is sufficient, increasing the quantity of virtual machines will not improve the performance. Instead, the addition of communication will degrade performance. But when the memory is not sufficient for the current data scale (rules or facts), the performance will decline significantly. It also can be illustrated in the results of experiments.

To evaluate performance of a rule engine, the matching time is a significant indicator. In our rule engine based on cloud computing, the matching time for facts is closely related to the number of virtual machines and the scale of rules. The total matching time is composed of process time and communication time. The model of matching time is described in the following expression.

$$T(F) = \left(c(n,R) + \frac{pR}{n} \right) \cdot F$$

In the expression, *F* is the number of facts set. $T(F)$ is the matching time for fact set. *n* is the number of virtual machines. *R* is the scale of rule set. $c(n,R)$ is the communication cost during the matching procedure. With the LWDG allocation method, *c* will become bigger with the increasing of *n* and *R*, which can be seen in the result of Section 7.1. *p* is a coefficient and it is related to the performance of a virtual machine. Since fact matching is related to the conditions in rules and it can be processed in parallel in different virtual machines, $\frac{pR}{n}$ is the average process time for a fact. The matching time model can be verified with experiments described in Section 7.4.

5.2. Algorithm of the Rete Process Allocation

In the proposed message-passing model, there are a large number of messages passed between processes. In our distribute Rete network, the message passing can be classified into

two types: within a virtual machine and across virtual machines. The latencies of the two types of message passing are different. As we know, the time of communication between machines is longer than that between processes in a same machine. In Cloud environment, the difference also exists. Therefore, the latency of messages passing between different virtual machines is longer than that in a same virtual machine. Reducing the messages between different virtual machines is an effective way to improve the performance in rule matching.

On the other hand, the load balance is also an important aspect in the process allocation. The load balance ensures that loads in different computing nodes are approximate as far as possible. In our Rete process allocation, the load is in direct proportion to the number of processes. If the loads of virtual machines are jagged or inclined, the virtual machine with more processes will reduce the performance of whole system. Therefore, the number of processes in different virtual machines is also a factor affecting performance. To maintain a balanced load in every virtual machine is considered in our algorithm.

In order to archive aforementioned two aims when allocating processes into several virtual machines, the linear weighted deterministic greedy (LWDG) algorithm of Rete process allocation is proposed. The algorithm of process allocation is based on a graph partition algorithm. The process allocation is a part of construction of Rete network from rules. As mentioned before, the Rete network construction can be divided into several types of nodes. In these types, constant test nodes and α-memory nodes are corresponding in Rete network. Join nodes and β-memory nodes (or terminal nodes) are also corresponding. In other words, there is a one-to-one correspondence between constant test nodes and its α-memory nodes. It is the same between Join nodes and β-memory nodes (or terminal nodes). In the proposed model, every process represents a node in Rete network. Therefore, the allocation of α-memory processes can be considered the same as the allocation of the corresponding constant test processes. Likewise, the allocation of β-memory processes is also the same as the allocation of the corresponding join processes. Hence, in the algorithm of allocation, only the constant test processes and the join processes need to be considered. The allocation algorithm is divided into two phases to deal with them respectively.

The first phase is constant test process allocation. At the beginning of this phase, there is no Rete process in every computing node. For rules in the rule engine, these conditions that have the same attribute and value in the left side of rules are mapped to an identical constant test process in our model. These rules are in set R, and these constant test processes are expressed in a set C.

Before the allocation of constant test processes, all of rules is traversed to create a matrix M, which is the adjacent matrix of constant processes.

The values of the adjacent matrix are the correlation of constant test processes. For the constant test processes c_a and c_b, the correlation in M is the frequency of the rules where they appear together. As follows:

$$m_{c_a c_b} = \sum_{r_i \in R} A^{r_i}(c_a, c_b)$$

in which,

$$A^{r_i}(c_a, c_b) = \begin{cases} 1 & c_a \in r_i \text{ and } c_b \in r_i \\ 0 & c_a \notin r_i \text{ or } c_b \notin r_i \end{cases}$$

Then, we can regard the adjacent matrix as a weighted graph of constant test processes. The allocation of these processes could be adapted and improved from a graph partition algorithm [17]. The allocation of processes is in the procedure of Rete construction. Rules are loaded one by one into the Rete network. In every rule, we consider every condition. If the

mapped constant test process has been created and allocated, it will be referred to the rest of construction in this rule directly. Otherwise, a new constant test process will be created and allocated. For the new constant test process c_i, the following allocation value AV of every partition p_j is computed. In the expression, P_j^t is the set of partition p_j at the time *t*.

$$AV_{c_i}^{p_j} = \left(\sum_{c_k \in P_j^t} m_{c_i c_k} \right) \cdot w(t, p_j)$$

where $w(t, p_j)$ is the weighted penalty function:

$$w(t, p_j) = 1 - \frac{|P_j^t|}{Cap}$$

Cap is the capacity of a partition. In order to get balanced m-partition in initialization, Cap is defined as:

$$Cap = \left\lceil \frac{|C|}{m} \right\rceil$$

where C is the constant test processes set.

After the $AV_{c_i}^{p_j}$ of each partition is computed for constant test process c_i, the c_i should be allocated into the partition which has the max allocation value.

With this algorithm, constant test processes of initial rules can be evenly allocated into different virtual machines in cloud platform, and correlations of constant test processes among different virtual machines can be reduced. It will be shown in the experiments in next section. The Figure 5 is the LWDG constant test process allocation algorithm.

Algorithm: Constant test process allocation (LWDG)

```
foreach r in R do
    foreach c in r do
        if c is not allocated then
            allocation ← 0;
            maxAV ← 0;
            foreach p_j in P do
                theAV ← 0;
                v ← 1;
                foreach c_k in p_j do
                    v ← m_{c_i c_k} + v;
                end foreach
                w ← 1 − sizeof(p_j) / capacity ;
                theAV ← v · w ;
                if theAV > maxAV then
                    maxAV ← theAV;
                    allocation ← j;
                end if
            end foreach
            allocation(c, p_allocation );
        end if
    end foreach
end foreach
```

Figure 5. LWDG Algorithm

For allocations of join processes in Rete network, a greedy algorithm is also involved. In the construction of each rule, the allocation of join process is only related to two memory processes, which will be linked. These two processes have been allocated as same as previous constant test processes or join processes. Therefore, in this step, there are two situations:

1. In the first situation, these two processes that the join process will link to are in the same virtual machine. Naturally, the join process should be also allocated in the same virtual machine.

2. In the other situation, these related two processes are in different virtual machines. The join process allocation will consider the balance of these two virtual machines. The loads of these virtual machines are compared. The virtual machine with less load will be chosen to allocate the join process.

For new added rules, the correlation of their conditions cannot be foreknow before these rules appear. Therefore, the allocation of new constant test process only requires considering the current situation. The new constant test process only relates with these processes in the same rule in this situation. These related processes of new constant test process c_l is defined in a set Γ_{c_l}. Similar to the algorithm above, the allocation value of every partition p_j is computed:

$$AV_{c_l}^{p_j} = \left|P_j^t \cap \Gamma_{c_l}\right| \cdot w(t, p_j)$$

where $w(t, p_j)$ is also the weighted penalty function. However, Cap is changed to the max constant test process capacity of a virtual machine. Then, c_l should be allocated into the partition that has the max allocation value.

6. Implementation

In order to verify the distributed rule engine model in cloud computing environment, the RUNES II was implemented and deployed in cloud platform.

6.1. Implementation of the Rule Engine

The implementation of RUNES II is based on our previous RUNES rule engine system. The whole system is implemented in Erlang, a programming language naturally supporting concurrent and distributed computing in a message-passing way. It could easily create millions of processes, load them on multi-cores in balance and make these processes location transparent during the communication. Moreover, Erlang runs its program in a virtual machine as Java does, thus could get platform independence that makes the deployment of our engine easily. The details of the implementation of RUNES can be found in [24].

The main architecture and modules of RUNES II are like those of RUNES. But, we improved the previous system in order to adapt to the cloud computing. As discussed in previous sections, the process allocation algorithm was added in the system and the construction of Rete network was optimized.

6.2. Deployment in Cloud

We deployed the system in the USTC Cloud platform [18]. The USTC Cloud platform is an on-demand virtualization infrastructure environment based on the elastic cloud computing. Besides the RUNES II Erlang program, a shell script program is implemented to deploy the system in virtual machines. At the beginning of deployment, the script program is responsible

for starting new virtual machines, initializing the Erlang environment, and constructing control processes of RUNES II. With the increasing of rules in the cluster, the performance of the cluster may decline. To cope with this problem, the script program can start new virtual machines on demand to deal with new processes of the rule engine.

All the programs of RUNES II were packed into a CentOS virtual machine image. In the USCT Cloud, we can apply for specific virtual machines and start them with our RUNES II image. Before deploying, the quantity of the virtual machines is evaluated with the scale of rules and the number of potential facts. Then, a appropriate number of virtual machines will be started and a subnet of these virtual machines can be set up to build a cluster. The agent in every virtual machine will construct Rete-note processes according to the rule set. In the procedure of construction, the allocation algorithm is considered. Finally, facts can be sent to match by agents of any virtual machine in the cluster.

7. Experiments and Discussion

In order to evaluate the performance of the model and the allocation algorithm, several groups of experiments have been designed and conducted. These experiments verify above algorithms and evaluate the performance of our system.

7.1. Simulation Experiments of the Allocation Algorithm

To verify the validity of the allocation algorithm, some simulation experiments have been conducted with Matlab before being implemented in the system. Rules in these experiments were generated randomly. In Matlab, the LWDG allocation algorithm is compared with the random allocation in a variety of scales of rules and constant test processes.

Firstly, we test the performance of two allocation strategies for allocating 12000 constant test processes into 4 partitions with different scales of rules. In each of rule, there are 4 correlating constant test processes. The results of these tests are shown in Figure 6. With the random allocation strategy, fractions of correlations between different partitions are always around 0.75 in different scales of rules. With the proposed LWDG allocation algorithm, the fraction of crossing correlations is lower than that with random allocation. It also can be found that the advantage of LWDG allocation reduces with the increasing of scale. To explain this trend, we explored the data in the experiment. As the number of rules increases, the correlations among them increase too. In the same scale of constant test processes, the connection density among these processes is more and more close to saturation. Therefore, the result will be close to the random allocation when the scale is big enough.

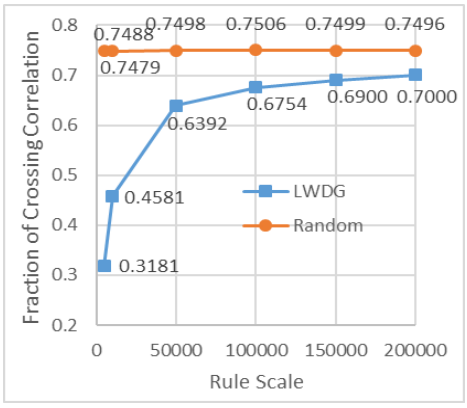


Figure 6. Simulation of Allocation for Rule Scale

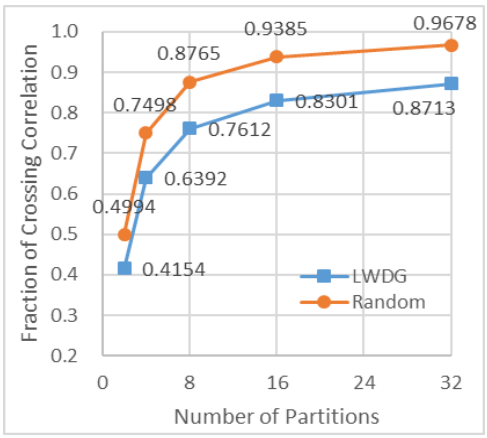


Figure 7. Simulation of Allocation for Partitions

Then, we evaluated effects of the allocation in different numbers of partitions. The numbers of partitions in these tests are from 2 to 32. The scale of rules is 50000, and the number of constant test processes is 12000 in each test. The result is shown in Figure 7. For different numbers of partitions, the consequence of LWDG algorithm is always better than that of random allocation. However, the effect of allocation may also decline with the increasing of number of partitions. Therefore, to define the number of virtual machines in the implemented system, the results of these tests will be considered.

The time consumption of allocation algorithm is much less than that of construction of Rete. However, the time complexity is still discussed. For every constant test process, the algorithm computes the allocation value for every partition. For each allocation value, it sums the correlations of processes which are already in the partition. So, the calculation amount is the quantity of allocated processes for each new constant test process. The time complexity is *O(n2)* for allocating all the processes. Based on the construction of Rete mentioned before, the time complexity of construction is higher than process allocation. Therefore, the allocation algorithm will not increase the time complexity in the whole construction of Rete. In addition, it can reduce the message passing among different virtual machines.

Because of the LWDG allocation algorithm is based on greedy method, the greedy algorithm may not be the best allocation algorithm from the view of global. Some other global optimization algorithms like Kernighan-Lin can provide better partition effect. But in our model where the rule engine compiles rules one by one, the global allocation algorithm needs to reconsider the existing allocation when new rule is added. It will bring in huge time consumption. Moreover, the LWDG can be blended in the procedure of Rete construction and can ensure a better allocation for every new process. As a result, the LWDG algorithm is a suitable allocation algorithm for our model and system.

7.2. Performance of the Allocation Algorithm in System

A series of experiments were designed to evaluate the performance of allocation algorithm. In these experiments, the compiling time and matching time of systems with LWDG allocation and random were selected as the metrics of the evaluation. Four virtual machines with 16GB of memory and eight cores of CPU were booted to construct a RUNES II rule engine. The rules and facts in our experiments were produced randomly. In every rule, there are 4 conditions, and each condition has 4 attribute-value pairs. In every fact, there are also 4 attribute-value pairs trying to match with the conditions in rules.

The results of compiling time with two allocation strategies are shown in Figure 8 by increasing the number of rules from 50000 to 200000. It can be seen that the LWDG allocation algorithm will consume a very little extend time compared with random allocation in the compiling stage. But in the matching stage, the system with LWDG allocation algorithm can provide better response time. From the Figure 9, we can see that the reduction rate of time consumption in matching is bigger than that of compiling. Therefore, when the rule engine is continuously running systems, the advantage of the LWDG allocation algorithm is remarkable.

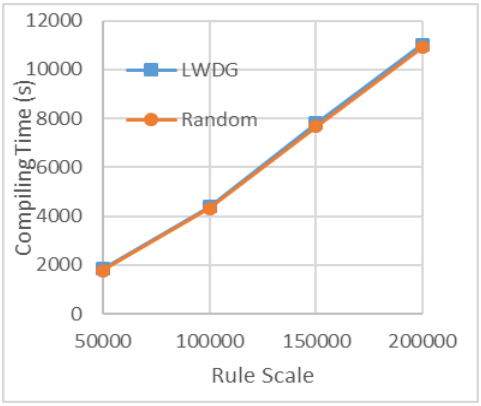


Figure 8. Compiling Time in 4 VMs

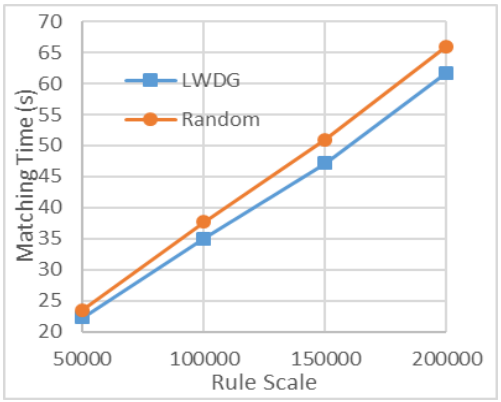


Figure. 9 1000 Facts Matching Time in 4 VMs

7.3. Experiments for Resource Cost

In order to evaluate the cost of the model in cloud environments and determine the configuration and scale of virtual machines, a group of experiments has been done.

Firstly, we evaluate the memory consumption in different rule scales and fact quantities. The rule scales in these experiments are 1000, 2000, and 4000. The number of facts is from 0 to 4000. Each combination of rule scales and fact quantities was tested in RUNES II to measure the memory consumption. The memory consumption is net usage, which excludes the basic consumptions of operation system and Erlang runtime environment. The Figure 10 and Figure 11 illustrate the trends of memory consumption in different rule scales and fact scales. These results show that the memory usage increases linearly, as both rules and facts increase. They verify our memory consumption model and determine the coefficient in the model.

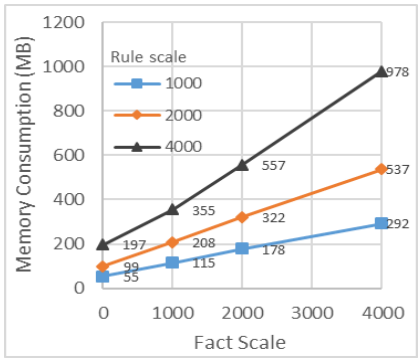


Figure 10. Memory Consumption for Fact Scale

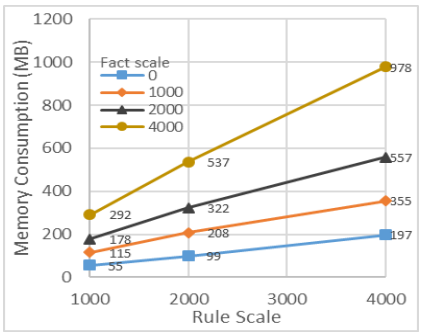


Figure. 11 Memory Consumption for Rule Scale

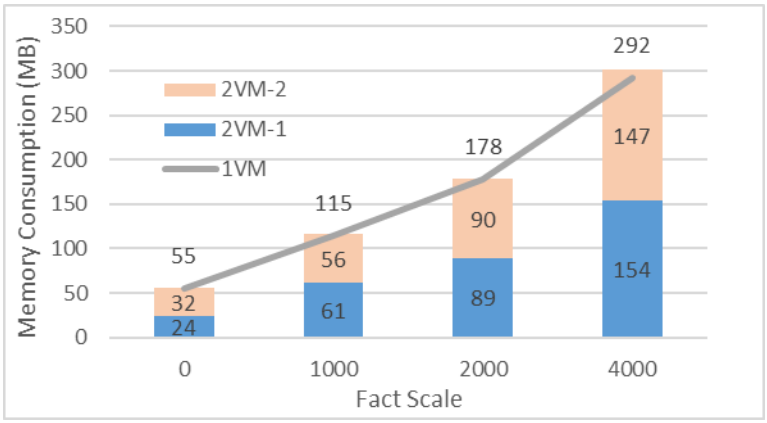


Figure 12. Memory Consumption in Different Numbers of VM for 1000 Rules

Then, another group of experiments were carried out to verify the total memory consumptions in different numbers of virtual machines are nearly equal for same scale of data. We tested the memory usages in 1000 rules for different numbers of facts, and compared these results in 2 virtual machines and in 1 virtual machine. As shown in Figure 12, the total memory consumption in 2 virtual machines almost equals the memory consumption in only one virtual machine. By the way, it can be found that the memory consumptions in two virtual machines are close when the rule engine is distributed in 2 virtual machines. It shows the validity of the load balance in the allocation algorithm.

7.4. Experiments in Cloud Environment

In Cloud Environment, the flexibility of the RUNES is increased, but what about the performance? In order to verify the system and test the performance in Cloud Architecture. A series of experiments were done.

In these experiments, each virtual machine has 10GB of memory, 40GB of disk space and eight cores of CPU. 1, 2, and 4 virtual machines were used to run with different numbers of rules from 50000 to 200000. The compiling time and matching time of these experiments are shown in Figure 13 and Figure 14 respectively. In general, the more virtual machines consume a little more time in compiling stage, because of the more cost of communications. Yet, when the memory consumption is close to the total memory with the increasing of rule scale, the compiling time is much longer than in normal. As we can see in Figure 13, the compiling time in 1 virtual machine is longer than it in 2 and 4 virtual machines when the

number of rules is 200000. Referring the model of memory consumption, the rule engine takes up more than 10GB of memory in this scale of rules. Adding the system memory usage, it is beyond the memory capacity in one virtual machine. Therefore, the performance of the system will decline sharply in one virtual machine.

Figure 13. Compiling Time of the Rule Engine in Cloud Environments

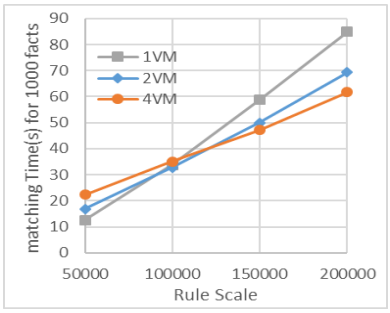


Figure 14. Matching Time of the Rule Engine in Cloud Environments

For the matching time of the rule engine, 1000 facts were added in the abovementioned rule engines. The result of matching time is illustrated in Figure 14. For 50000 rules, the matching time in 4 virtual machines is longer than it in only one virtual machine for additional communications among virtual machines. With the increasing of rules, the matching times also increase in all cases. But the rate of increasing in the rule engine with more virtual machines is lower than that in one virtual machine rule engine. Thus, when the rules increase to 100000, the matching times in all cases are similar. Moreover, for the 200000 rules, the rule engine in 4 virtual machines matches faster than in 1 or 2 virtual machines.

The results agree with the matching time model. The rule engine in more virtual machines shows a steady speedup with more computing resources when the rule scale is large enough. Therefore, in the cloud environments, RUNES II can better display its advantages with the bigger data of rules and facts.

8. Conclusions

In this paper, we describe a rule engine named RUNES II in cloud environments. It is based on the previous message-passing rule system RUNES. The system is improved and deployed in cloud platform. Details of design and implementation are explained. A model of resource cost is studied to estimate the suited virtual machine resources. An algorithm of process allocation is proposed to optimal the communication in matching procedure. A lot of

experiments have been conducted to verify the allocation algorithm, resource cost model, and performance of the system. From these experiments, it can be shown that the RUNES II rule engine is flexible and efficient. Especially when the number of rules and facts is in very large scale, the performance in the rule engine with multiple virtual machines will be better. Our rule engine based on cloud computing has better utilization and more practicability than those traditional parallel rule engines.

Acknowledgements

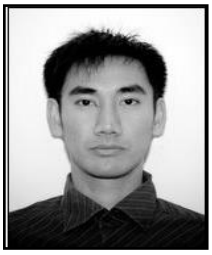
We thank the Supercomputing Center at University of Science and Technology of China (USTC) for their platform and technical support. This research was supported by the National Key Technology R&D Program under Grant No. 2012BAH17B03 and the Cloud Computing Joint Laboratory of USTC and Lenovo.

References

[1] C. L. Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem", Artificial intelligence, vol. 19, no. 1, **(1982)**, pp. 17-37.
[2] D. P. Miranker, "TREAT: A New and Efficient Match Algorithm for AI Production Systems", Morgan Kaufmann Publishers, San Francisco, USA, **(1990)**.
[3] B. R. Doorenbos, "Production Matching for Large Learning Systems", University of Southern California, **(1995)**.
[4] I. Wright and J. A. R. Marshall, "The Execution Kernel of RC++: RETE*, a Faster RETE with TREAT as a Special Case", International Journal of Intelligent Games and Simulation, vol. 2, no. 1, **(2003)**, pp. 36-48.
[5] J. A. Kang and A. M. K. Cheng, "Shortening Matching Time in OPS5 Production Systems", IEEE Transactions on Software Engineering, vol. 30, no. 7, **(2004)**, pp. 448-457.
[6] J. G. Schmolze and S. Goel, "A Parallel Asynchronous Distributed Production System", National Conference on Artificial Intelligence (AAAI), Boston, Massachusetts, USA, **(1990)** July 29–August 3.
[7] A. Anurag, M. Tambe and A. Gupta. "Implementation of Production Systems on Message-passing Computers", IEEE Transactions on Parallel and Distributed Systems, vol. 3, no. 4, **(1992)**, pp. 477-487.
[8] J. N. Amaral and J. Ghosh, "A Concurrent Architecture for Serializable Production Systems", IEEE Transactions on Parallel and Distributed Systems, vol. 7, no. 12, **(1996)**, pp. 1265-1280.
[9] M. M. Aref and M. A. Tayyib, "Lana–Match Algorithm: a Parallel Version of the Rete–Match Algorithm", Parallel Computing, vol. 24, no. 5-6, **(1998)**, pp. 763-775.
[10] C. Wu, L. Lai and Y. Chang, "Parallelizing CLIPS-based Expert Systems by the Permutation Feature of Pattern Matching", 2010 Second International Conference on Computer Engineering and Applications (ICCEA), Bali Island, Indonesia, vol. 1, **(2010)** March 19-21.
[11] B. Cao, J. Yin, Q. Zhang and Y. Ye, "A MapReduce-based Architecture for Rule Matching in Production System", IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom), Indianapolis, USA, **(2010)** November 30-December 3.
[12] T. Dong, J. Shi, J. Fan and L. Zhang, "An Improved Rete Algorithm Based on Double Hash Filter and Node Indexing for Distributed Rule Engine", IEICE Transactions on Information and Systems, vol. 96, no. 12, **(2013)**, pp. 2635-2644.
[13] M. J. Sawar, U. Abdullah and A. Ahmed, "Enhanced Design of a Rule Based Engine Implemented using Structured Query Language", International Conference of Computational Intelligence and Intelligent Systems, London, UK, **(2010)** June 30-July 2.
[14] Drools http:// www.jboss.org/drools/.
[15] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph and R. Katz, "A View of Cloud Computing", Communications of the ACM, vol. 53, no. 4, **(2010)**, pp. 50-58.
[16] F. Rosenberg and S. Dustdar, "Business Rules Integration in BPEL-a Service-Oriented Approach" IEEE International Conference on E-Commerce Technology, Munich, Germany, **(2005)** July 19-22, pp. 476-479.
[17] I. Stanton and G. Kliot, "Streaming Graph Partitioning for Large Distributed Graphs", Proceedings of the 18th ACM SIGKDD International Conference On Knowledge Discovery And Data Mining, Beijing, China, **(2012)** August 12-16, pp. 1222-1230.
[18] USTC Cloud http://cloud.ustc.edu.cn/.

[19] T. Gao, X. Qiu and L. He, "Improved RETE Algorithm in Context Reasoning for Web of Things Environments", IEEE International Conference on Internet of Things, Beijing, China, **(2013)** August 20-13, pp. 1044-1049.
[20] K. Bilal and S. Mohsin, "Muhadith: A Cloud Based Distributed Expert System for Classification of Ahadith", 2012 10th International Conference on Frontiers of Information Technology (FIT), Islamabad, Pakistan, **(2012)** December 17-19, pp. 73-78.
[21] Q. Li, Y. Jin, T. He and H. Xu, "Smart Home Services Based on Event Matching", 10th International Conference on Fuzzy Systems and Knowledge Discovery, Shenyang, China, **(2013)** July 23-25, pp. 762-766.
[22] M. Kim, K. Lee, Y. Kim, T. Kim, Y. Lee, S. Cho and C.G. Lee, "RETE-ADH: An Improvement to RETE for Composite Context-Aware Service", International Journal of Distributed Sensor Networks, **(2014)**.
[23] W. Voorsluys, J. Broberg and R. Buyya. "Introduction to Cloud Computing", Cloud Computing, **(2011)**, pp. 1-41.
[24] J. Wang, R. Zhou, J. Li and G. Wang, "A Distributed Rule Engine Based on Message-Passing Model to Deal with Big Data", Lecture Notes on Software Engineering, vol. 2, no. 3, **(2014)**, pp. 275-281.

Authors



Rui Zhou is a PhD student in Computer Science and Technology at the University of Science and Technology of China. He received his bachelor degree in Xidian University in 2009. His research interests include Cloud computing, service-oriented computing, mobile computing and context-aware. He is author of some research papers published at conference proceedings.

Guowei Wang is a PhD student in Computer Science and Technology at the University of Science and Technology of China. He received his bachelor degree in the University of Science and Technology of China in 2012. His research interests include Cloud computing, rule-based computing, mobile computing and distribute computing.

Jinghan Wang is a master student in Computer Science and Technology at the University of Science and Technology of China. He received his bachelor degree in Hefei University of Technology in 2011. His research interests include Cloud computing, rule-based computing, mobile computing and distribute computing.

Jing Li received his B.E. in Computer Science from University of Science and Technology of China (USTC) in 1987, and Ph.D. in Computer Science from USTC in 1993. Now he is a Professor in the School of Computer Science and Technology at USTC. His research interests include Distributed Systems, Cloud Computing and Mobile Computing. He is author of a great deal of research studies published at national and international journals, conference proceedings as well as book chapters.