Self-Organizing Autonomic Computing Systems for Cloud Resource Management

Bogdan Solomon*, Dan Ionescu†, Cristian Gadea‡
School of Electrical Engineering and Computer Science
University of Ottawa
Ottawa, ON, Canada
Email: *bsolomon@ncct.uottawa.ca, †dan@ncct.uottawa.ca, ‡cgadea@ncct.uottawa.ca

***Abstract*—In recent years, a great deal of research has been undertaken in the area of automating the enterprise IT Infrastructure. For enterprises with a large number of computers, the IT Infrastructure represents a considerable amount of the enterprise budget assigned to its operation. Autonomic Computing Systems are systems which were created to correct and optimize the IT infrastructure's own self-functioning by executing corrective operations without any need for human intervention. Applications of autonomic systems range from usage optimization for clusters of servers or virtual machines, to error prevention and error recovery for web applications. In most cases where autonomic computing systems have been developed, this was achieved by the addition of external global controllers monitoring the subsystems of the enterprise IT Infrastructure, determining where changes should be made and applying appropriate commands to implement these changes. At the same time, the software industry has seen an explosion in the usage of cloud based services. By moving services in the cloud, companies can decrease the IT budget such that they only use the resources which are needed, when they are needed. Furthermore, services deployed in the cloud can be distributed geographically across multiple datacenters such that users obtain a higher quality of service by connecting to a closer datacenter. Self-Organizing systems are systems which reach a global desired state without the use of any central authority. Such systems are better suited to manage a cloud of servers than a centralized controller as they scale better when the controlled system is scaled also. In this paper, an autonomic system based on a self-organizing architecture is introduced. The system is applied to the self-optimization of a web-based real-time collaborative application running on a geographically distributed cloud. Experiments and simulations with the self-organizing, self-optimizing autonomic computing methodology are given to support the design and the implementation described.**

I. Introduction

Cloud computing has become an integral technology as more and more services make use of the capability to both use hardware resources as a service and to scale a system from a small number of users to millions of users easily. Companies can now make use of outside resources and simply rent the hardware as needed from cloud providers like Amazon, Google, and Microsoft. In the past, scaling an application after release would take weeks as new servers would need to be purchased, configured, tested and finally set to production. With the advent of cloud computing, a new image of a server can be created and be ready for use in minutes or even seconds.

At the same time, the world has become more connected with companies having offices all over the world and with people wishing to communicate with people in far away countries. Due to these reasons, collaboration tools have become more important allowing people to communicate not only by text but also audio/video chat while at the same time share documents, images, and videos and collaborate on them. Previous work in [1] presented such a web based collaboration tool. The collaboration tool is cloud based and makes use of a geographically distributed server architecture, in which multiple clouds at different locations host instances of the media server for the collaboration application. Clients connect to one of the available clouds and can communicate via the collaboration application with clients connected to any other server in any cloud location. Within a cloud, servers use a Group Membership Service (GMS) to communicate with each other such that servers can be dynamically created/destroyed and the servers would join/leave the group as needed. A second level of communication is used through the addition of gateways which enable the communication between clouds. Through the use of these two levels of communication the system can keep the proper state across all the servers, no matter the location of the server.

The geographically distributed cloud based collaboration system requires an autonomic system which would allow the clouds to scale dynamically based on demand. As more users connect to one of the cloud datacenters, that datacenter must be able to scale up. When users disconnect, the number of servers must decrease as well to avoid paying for unused resources. However, new servers must be added before the performance of the service has degraded due to too many users being connected to the service. Similarly, it is desired that the extra servers are removed once the system detects that they are unneeded with as little delay as possible, as every extra minute of resources being used can incur extra payments. Autonomic systems are usually modeled using a MAPE-K loop. The system which is being automated is monitored to determine its behaviour. The monitored data is analyzed such that possible breaches of the system's Service Level Agreement can be detected. Once such a breach is detected the plan function decides on a course of action to fix the problem. Finally the execute function applies the desired plan to bring the resource back into compliance. The K in the MAPE-K loop stands for

knowledge and represents the various models, policies and information the control loop has about the resource being controlled.

In this paper, a **Self-Organizing autonomic system** which self-manages a cloud of servers is introduced. The system architecture, system design and results obtained via simulations and experiments with a live system are presented. The self-managing function ensures that all the servers in the same datacenter maintain the same response time and CPU usage. At the same time, the self-organizing system can add or remove servers as needed when load for the service increases or decreases.

The rest of the paper is structured as follows. Section II introduces the self-organizing model for the self-optimizing system. Section IV introduces a simulation environment used to test the self-organizing algorithm. Section V presents some performance data from the test bed and simulation. Finally, Section VI reflects on the contributions of this paper and proposes topics for future research.

II. Self-organizing Model

The collaboration server which is managed by the SLA breach detection system presented in this paper is described in [2] and [1] and uses a self-organizing approach in order to manage the distribution of clients across the cloud as well as to scale the cloud up and down. The self-organizing system can not be based on a model which considers only the number of clients as a perturbation because load for a collaboration server is a combination of the number of clients connected to the server, the number of sessions running on the server and most importantly the number of streams received by the server and multiplexed towards receiving clients. Based on these perturbations, each server can decide if it should receive more clients or not, independent of any decision made by another server. On top of the server self-organization with relation to accepting new clients, the system also needs an approach to determine when the cloud's SLA will be breached and take proactive action by adding or removing servers from the cloud.

The reason to use a self-organizing systems for the adaptation is due to the intrinsic properties that self-organizing systems poses:

1) Adaptable - the ability to deal with changes in the environment which were not predicted at design time. This is important for the system developed for the paper as we do not know the bounds of how many servers could be in a cluster at one time or the maximum peak demand for the service and as such we want a system which is able to adapt the control law dynamically.
2) Resilient - parts of the system can die or be lost but the remaining still perform their goal. In a cloud environment this is desired, as servers can come up an down at any time, and even network connectivity could be lost between servers or between data centers.
3) Emergent - the complex behaviour arises from the properties and behaviour of the simple parts. As a cloud of servers increases in size it becomes more complicated for a single controller to manage the entire system. As such a system where the control emerges from the interactions of a lot of small parts is desired as it decreases complexity and can scale much more, at the cost of overall resource utilization.
4) Anticipation - the system can anticipate problems and solve them before they impact the whole system. This is obviously desired, as we wish to detect SLA breaches before they happen, such that corrective actions can be taken and the breach can be avoided.

III. SLA breach prediction: Ant Colony Optimization

The ant colony optimization (ACO) algorithm [3], [4] is best known for load balancing clouds or finding the best route in a network. The algorithm has received a lot of attention and work in academia and was presented in chapter **??**. At a high level the algorithm uses a number of simple agents called ants which traverse the network and leave a trail of pheromones which other ants can then follow. Good paths or solutions are reinforced by having more ants traverse them and leaving more pheromones. Once a path or solution is no longer viable, less ants travel it and another path is reinforced as more ants travel that path. In networking optimization ants behave as normal packets and traverse the network from router to router. When an ant reaches a router it can look at how long the arrival router's buffer queue is for the router the ant came from and then reinforce that route based on latency, buffer sizes, etc.

Take as an example a network that looks like the one in Figure 1 in its initial non-traversed state, where the load of the links is the value shown on the edges of the graph, and in which ants start at node *S* and finish at node *E*. If there are 6 ants in the system, each ant deposits pheromone at a rate of $\frac{1}{edgeLoad}$, and in the initial state ants have an equal chance of taking an edge as there are no pheromones deposited, then the initial probability of an ant taking any given edge is shown in figure 2. Each edge in figure 2 has a probability of being traversed by an ant of $\frac{1}{numberEdgesFromNode}$.

After the first run of the ants the pheromone in the network will look as in figure 3. In figure 3 the edges show the number of ants taking each edge and the total amount of pheromone deposited in the form of $\frac{numberAnts}{totalPheomoneLevel}$. For example, the edge from *S* to *1* has 2 ants traversing it and total pheromone being deposited with a value of 2, while the edge from *5* to *6* has 2 ants traversing it and total pheromone being deposited with a value of 4 because the link has less load. While this example assumes that the ants split exactly as the probabilities shown in figure 2, note that it possible for more ants to take an edge with lower probability since ants roll a random value when deciding the edge to take.

When going back to the source, more ants will prefer edges with higher pheromone values thus resulting in that path being reinforced as shown in figure 4 where the probability of ants taking an edge is shown. This example does not include the decay of the pheromone level, but in an ACO implementation

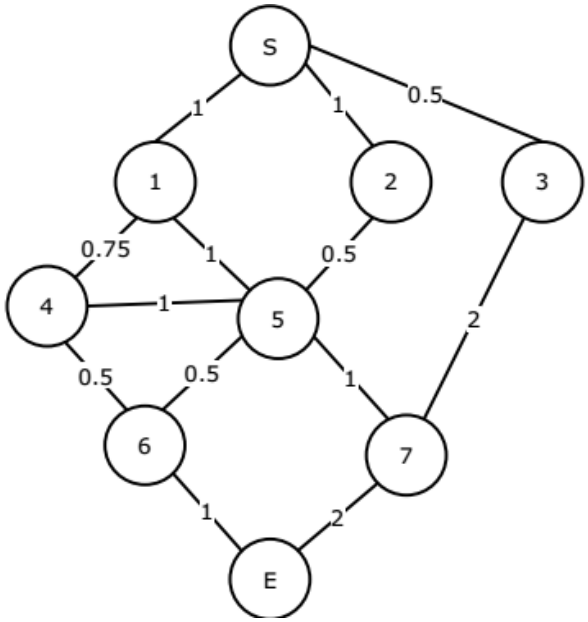

Fig. 1: Network Ant Colony Optimization - Initial state

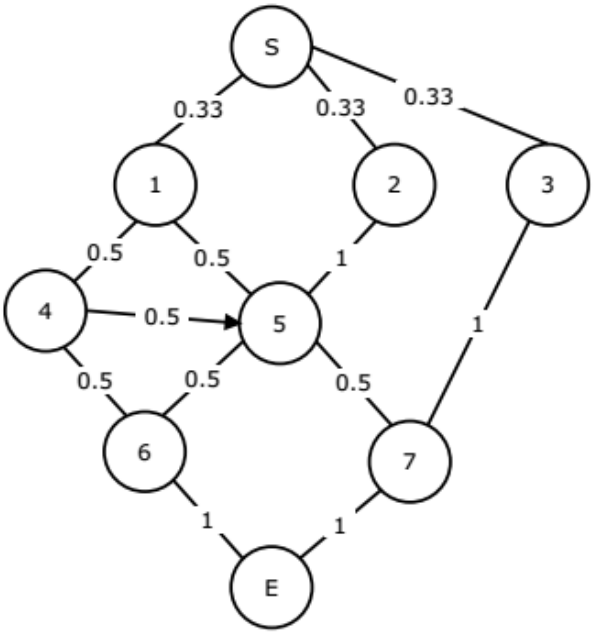

Fig. 2: Network Ant Colony Optimization - Path probability initial

the pheromone level always decays at a given rate such that edges that are no longer traveled have their pheromone levels slowly decrease. As more time passes, ants will continue to add more pheromone on good paths in the end resulting in the path with the lowest load being reinforced. If the load in the network changes, ants will be able to start reinforcing the path with the better performance and forget about the previous path.

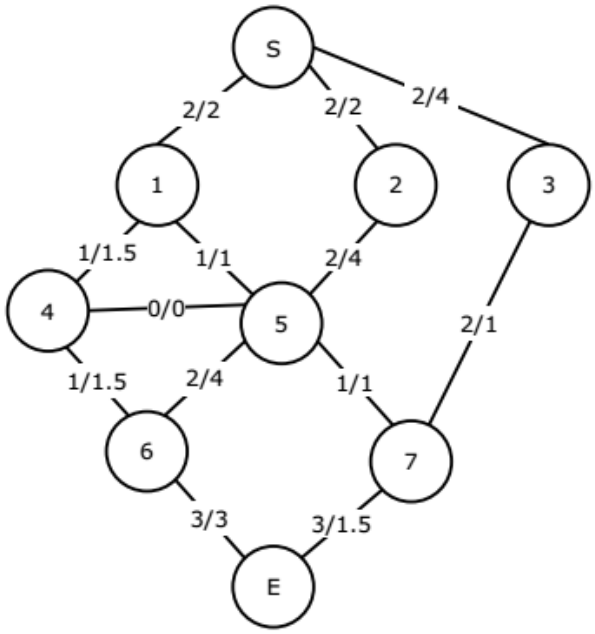

Fig. 3: Network Ant Colony Optimization - Pheromone update

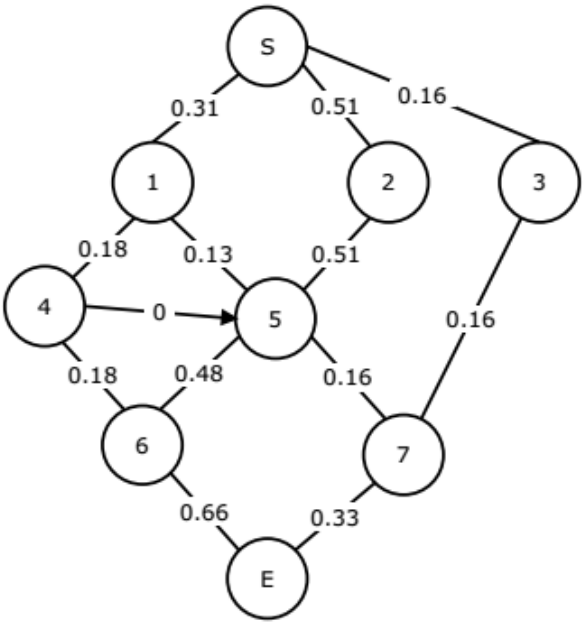

Fig. 4: Network Ant Colony Optimization - Path probability update

A. ACO Model for SLA breach prediction

The self-organizing algorithm used to predict SLA breaches in cloud systems is similar in nature to how the ACO algorithm behaves when finding the best route in a network. However, there are a number of significant differences due to the nature of the problem that is being resolved. The goal of the ACO algorithm while performing path optimization is to discover how viable different routes are and reinforce the good routes while at the same time diminishing the use of bad routes. For this problem the ants deposit pheromones on the routes between nodes such that paths with higher pheromones are more desired. In the case of the SLA breach prediction problem, the goal of the ACO algorithm is to determine based on the knowledge about the visited servers if the whole cloud is about to breach it's SLA requirements. For this problem, the ants deposit pheromones at the servers and use the amount of pheromone as a proxy of how loaded the whole cloud.

The ACO model for SLA breach prediction is composed of three components:

1) Ants - ants are simple agents which traverse the network of servers and deposit pheromone at the servers based on a mathematical model
2) Pheromone levels - each server has it's own pheromone level which is increased by ants visiting the server and decays periodically. Pheromones are a proxy of how overloaded a single server is.
3) Mathematical model - this is the model which converts the fuzzy model result of each server into a pheromone value. The mathematical model also represents when an SLA breach is predicted.

B. Self-organizing agent: The ant

Ants are simple agents that traverse the network of servers. They have no knowledge about the global state of the cloud and how overloaded all the servers in the cloud are. Ants only know the state at the currently visited server and also have a short memory about the last x servers which the ant has visited and what the pheromone level was at those servers.

Whenever a server starts, the server's control system creates a new ant and sends it through the network of servers. Thus the total number of ants is equal to N where N is the number of live servers in the cloud. When an ant is first created, because it has no prior knowledge, it is sent to another server randomly. If only one server exists in the cloud, then the ant keeps visiting the same server.

As ants reach other servers they deposit pheromone at the server they arrive at, at a rate inversely proportional to the load of the server as represented by the fuzzy control system. At the same time ants wait at the server a time proportional to the load of the server. Thus overloaded servers will have less pheromone deposited when compared to an under-loaded server. By having ants wait a longer time at overloaded servers, the overall amount of pheromone in the network will further decrease. With this approach, a lower average amount of pheromone in the network of servers means that the cloud has higher load due to the fact that a larger number of servers have low pheromone levels because of being overloaded.

Server	Time since last visit (s)	Pheromone Level
Server 1	15	10
Server 2	20	5
Server 3	5	8
Server 4	35	5.5
Server 5	0	10

TABLE I: Ant routing knowledge prior

Server	Probability (%)
Server 1	35.10
Server 2	26.48
Server 4	38.42
Server 5	0

TABLE II: Ant routing probability

Ants also store information about the last x servers the ant has visited and the time passed since the last visit to each of these servers. When an ant decides which server to go to, it uses a random function which is proportional to the time since it has visited a server combined with the pheromone level of the destination server. Thus the ant will give preference to the servers it has not visited in a long time, and especially the servers it has never visited or which are not in it's memory. Because the server structure is not stable and servers can join and leave at any time, ants decide the next server to visit based on the servers known by the current server the ant is at. Assume a cloud with 5 servers and an ant which has the information in table I in it's history table and which reaches Server 5.

Based on the table, the ant computes the probability of visiting each server as:

$$P_s = (t_s / \sum_{i=1}^{N} t_i + p_s / \sum_{i=1}^{N} p_i)/2 \quad (1)$$

where

$$i \in \{\text{servers known by current server the ant is at}\} \quad (2)$$

where P_s is the probability of visiting server s, t_s is the time since it has visited server s last time and p_s is the pheromone at server s. These probabilities are computed only on the servers known as being up by the server the ant is at. The server the ant is currently at has to be excluded from the calculations, and it's visit probability will be 0. Assuming also that Server 5, which is the server the ant is at currently does not yet know about server 3. As such, the visit probability table looks as in Table II.

As such, the ant will roll a random value between 0 and 1, and choose which server to go to. A value between 0 and 0.3842 means that the next server will be *Server 4*, between 0.3842 and 0.7352 it means that the next server will be *Server 1* and a value between 0.7352 and 1 means *Server 2*. Assuming the random value the ant rolls is 0.7, and that the ant waits at Server 5 for 5s and deposits 1 pheromone, the routing table

Server	Time since last visit (s)	Pheromone Level
Server 1	0	10
Server 2	25	5
Server 3	10	8
Server 4	40	5.5
Server 5	5	11

TABLE III: Ant routing knowledge posterior

after the ant moves to the next server will be as the one in Table III.

In order to avoid having all ants visit a new node at the same time, whenever an ant discovers a new server it initializes the time since it visited the server with a random value. Assume that after the ant reaches Server 1, it discovers a new server which was unknown before - Server 6. The time since last visiting Server 6 will be initialized with a random value between 0 and the maximum time since last visiting a server which is known to be alive as in Equation 3. Furthermore the known pheromone level of the new server will be 0. If Server 1 only knows Server 2, 3 and 5 then the random value will be between 0 and 25s.

$$t_{new} = random(0, max(t_{known})) \tag{3}$$

An ant's behaviour can be described by the algorithm below:

Algorithm 1 Ant Colony Optimization Pseudocode

```
Calculate pheromone at current node
Update ant's server history tables
if Ant should morph then
    Morph ant
else
    Calculate next server for ant
    Send ant to next server
end if
```

For example, for a network of five servers Figure 5 shows the behaviour of the ant algorithm.

C. Server and cloud load: Pheromone levels

For the purpose of controlling the number of servers in the cloud, the pheromone level in the network is used as a proxy for how loaded the entire cloud of servers is and is used to decide when to add or remove servers. As ants move through the network of servers they deposit pheromones - whenever an ant reaches a server it computes how much pheromone to deposit based on the result of the control system for that server. If the server's control system has a result which shows that the server is underloaded then more pheromone is deposited. If the server is overloaded then less pheromone is deposited. At the same time the pheromone amount at each server decays periodically at a constant rate. As such, a large amount of pheromone shows that servers are underloaded and servers should be removed, while a small amount of pheromone shows that servers are overloaded and that servers should be added.

On top of updating the pheromone at each server, each ant is also responsible for measuring the pheromone level at each

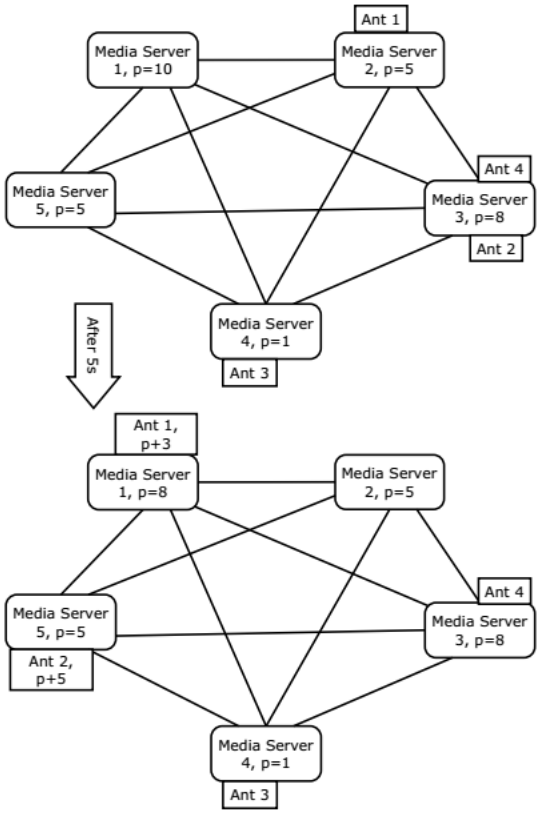

Fig. 5: Ant Colony Optimization Network

node it visits. Ants store a history of the last x nodes that they visited and the pheromone level at each of these nodes. Every time an ant moves to a new server, it removes the oldest pheromone value from this list and adds the pheromone at the current node. It then computes an aggregate metric of the pheromone at the last x nodes visited, which is a simple average:

$$p_{ant} = \sum_{i=1}^{N} p_N / N \tag{4}$$

where p_N is the pheromone at server N. Once the pheromone level of an ant goes under a predefined value, the ant triggers the algorithm to determine the new count of servers which should be added to the cloud. Similarly, if the pheromone level of an ant becomes too high, the ant triggers the algorithm to determine the number of servers which should be removed from the cloud. The algorithm used to increase or decrease the size of the cloud triggers when more than half of the ants in the system agree that the algorithm should start. Until enough ants agree to start the cloud size optimization the ants continue moving through the network of servers and update the pheromone levels. Once more than half of the ants agree all ants stop and the optimization algorithm starts.

D. Mathematical model

The pheromone level at a single node is defined base don the previous section as:

$$p_n^t = p_n^{t-1} + \sum_{i=1}^{K}(\tau_i * (1-p)) - \rho \tag{5}$$

where p_n^t is the amount of pheromone at node n at time t where t can be considered discrete in some time increment and K is the amount of ants arriving at the node in the time frame between $t-1$ and t.

Based on this equation, there are a number of parameters that can be tuned for the ACO algorithm:

1) Decay amount - the amount that the pheromone level decreases at each server: ρ
2) Decay rate - how often does the pheromone level decay at each server: T_{decay}
3) Ant minimum wait time - the minimum wait time for an ant at a server: $T_{minwait}$
4) Ant pheromone level - the maximum amount of pheromone deposited by an ant at a server: τ_{max}
5) Ant history size - how many servers an ant should store in it's history
6) Minimum/maximum trigger level - the minimum/maximum average pheromone level across the last x servers visited by the ant required for the ant to trigger the optimization algorithm: Pt_{max}/Pt_{min}

Based on these parameters the following three cases can be considered for how the system will behave.

1) Under-loaded server: If an under-loaded server is considered where the ant waits the minimum time and deposits the maximum amount of pheromone and we set $T_{decay} = T_{minwait}$ then in each time period the amount of pheromone at the server can be defined as:

$$\begin{aligned} p_n^t &= p_n^{t-1} + (\tau_{max} - \rho) \\ p_n^t &= (n-1) * (\tau_{max} - \rho) \end{aligned} \tag{6}$$

At the same time the ant will make a decision when $p_n^t > Pt_{max}$ or $p_n^t < Pt_{min}$. Because the server is under-loaded we expect that the amount of pheromone will continuously increase since the ant's goal should be to remove servers due to over-provisioning. The ant will make a decision when:

$$\begin{aligned} p_n^t &= Pt_{max} \\ (n-1) * (\tau_{max} - \rho) &= Pt_{max} \\ (n-1) &= \frac{Pt_{max}}{(\tau_{max} - \rho)} \end{aligned} \tag{7}$$

As such it can be determined that:

1) τ_{max} must be greater than ρ
2) The time to wait before the ant makes a decision can be defined by Pt_{max} and $(\tau_{max} - \rho)$. A smaller difference between τ_{max} and ρ will lead to slower decisions, while a smaller Pt_{max} will lead to quicker decisions.

2) Balanced server: If a balanced server is considered - that is a server where the control function shows that the server is well balanced and neither under-loaded, nor overloaded then both the amount of pheromone and the wait time of the ant change. Define T_b as the time for the ant to wait at a balanced server and τ_b as the amount of pheromone deposited at a balanced server. Both T_b and τ_b are defined in terms of the control function, where $T_b = T_{minwait}/(1-p)$ and $\tau_b = \tau_{max} * (1-p)$

$$\begin{aligned} p_n^t &= \frac{t * \tau_b}{T_b} - \frac{t * \rho}{T_{decay}} \\ p_n^t &= \frac{t * \tau_{max}(1-p)}{\frac{T_{minwait}}{1-p}} - \frac{t * \rho}{T_{decay}} \end{aligned} \tag{8}$$

The goal of the ant system in such a case is to maintain the level of the pheromone such that servers are not added or removed. As such it can be determined that:

$$\frac{t * \tau_{max}(1-p)}{\frac{T_{minwait}}{1-p}} = \frac{t * \rho}{T_{decay}} \tag{9}$$

which means that the amount of decay equals the amount of pheromone deposited over long periods of time.

$$\begin{aligned} t * \tau_{max} * (1-p) * T_{decay} &= t * \rho * \frac{T_{minwait}}{1-p} \\ \tau_{max} * (1-p)^2 * T_{decay} &= \rho * T_{minwait} \\ \frac{\tau_{max} * (1-p)^2}{\rho} &= \frac{T_{minwait}}{T_{decay}} \end{aligned} \tag{10}$$

If as in the previous case for an under-loaded server $T_{minwait} = T_{decay}$, then the equation becomes:

$$\tau_{max} * (1-p)^2 = \rho \tag{11}$$

This equation allows the user to set the relation between τ_{max} and ρ for a given p where the cluster size should be stable. For example, if a cluster size should be stable when $p = 50\%$ then

$$\begin{aligned} \tau_{max} * (0.5)^2 &= \rho \\ \tau_{max} * 0.25 &= \rho \end{aligned} \tag{12}$$

3) Over-loaded server: The previous equation holds for an over-loaded server as well. If we take the same example as before where the system is set to be balanced for $p = 50\%$, if p goes up to 90% then the previous equation becomes:

$$\begin{aligned} p_n^t &= \frac{t * \frac{\rho}{0.25} * 0.1}{\frac{T_{decay}}{0.1}} - \frac{t * \rho}{T_{decay}} \\ p_n^t &= \frac{t * \frac{\rho}{0.25} * 0.1}{\frac{T_{decay}}{0.1}} - \frac{t * \rho}{T_{decay}} \\ p_n^t &= \frac{t * \rho}{T_{decay}} * (0.04 - 1) \\ p_n^t &= \frac{t * \rho}{T_{decay}} * -0.96 \end{aligned} \tag{13}$$

This equation means that the pheromone rate at the node will decrease by $\rho * -0.96$ every decay period.

Once enough of the ants discover that either the minimum or maximum threshold is breached, the cloud size optimization self-organizing algorithm triggers.

E. Cloud size optimization: Ant house hunting

The ant house hunting algorithm is inspired by the behaviour ant colonies use in order to reach consensus decisions in the case of relocating the nest as in [5] and [6]. Ants use a complex, distributed approach to find a new nest which involves some ants searching for a new nest, scout ants communicating with each other about the suitability of a discovered nest, recruitment of other scout ants to a discovered nest, and finally a majority of ants agreeing on a new nest and moving the ant colony to the new nest.

Initially, when the nest is in danger of being destroyed ants start searching for a new nest where the colony can relocate. In this phase, scouting ants leave the nest and search for a new location for the colony to relocate to. Once a scout finds a suitable location, it evaluates the location based on various factors and then returns to the home nest. Once a scout ant is satisfied with the new nest it goes back to the home nest and starts recruiting other ants to its chosen nest. Ants then start doing tandem runs where one ant leads another. Upon arriving at a new nest, a recruited ant evaluates the new nest and then starts doing tandem runs if the new nest is acceptable. At this point there are many possible nests and the ants must reach a consensus as to which nest will be the new home. One approach that ants are assumed to use is that of quorum threshold [7]. When ants reach the home nest they check if a quorum has been reached and if yes, they start to move the entire ant colony to the new nest.

F. Model for Cloud size optimization

Based on the above natural process, a self-organizing system can be developed in order to optimize the number of servers in the cloud by having an ant colony search for the optimal number of servers which should be up in the cloud at a given time. This algorithm works by having each ant in the ant colony search for a new nest, where a nest is represented by a new optimum number of servers in the cloud. The ants are the same ants for the ACO algorithm. When the SLA breach is predicted the ants morph to house hunting ants and start to look for a new home. The algorithm assumes that there is a home nest where the ants can meet after searching for a new nest and exchange information. Once all the ants have agreed on the same optimum value, the servers are added or removed as desired and all ants are morphed back into food foraging ants for the ACO algorithm.

The model for optimizing the size of the cloud is composed of three components:

1) Nest candidates - each nest candidate represents a possible solution in term of the number of servers that should be up in the server
2) House hunting ants - ants are simple agents which search for a new home nest
3) Mathematical model - this is the model which represents the viability of a nest from the point of view of an ant.

G. Nest candidates

For each ant that morphs into a house hunting ant a new nest is created. The nest is initialized with a value representing the number of servers which should be up in the cloud if the nest is selected as the new home. These possible solutions are computed as permutations of the current size of the cloud as known by ant which will have the nest as its first candidate. If the ant was morphed because of lack of pheromone, then a random value is generated which represents the number of servers to be added. Each random value is proportional to the size of the cloud and the pheromone level across the last N servers known by the ant. For example, if the cloud contains 100 servers, then the ant would be initialized with a random values, which is defined as:

$$ServerCount_{add} = \frac{N}{2} * rand + \frac{N}{2} * \frac{Pt_{max} - p_{ant}}{Pt_{max}} \quad (14)$$

where $rand$ is a random value between 0 and 1 and p_{ant} is the average pheromone level seen by the ant across the last n servers it went to. The pheromone is used as a scaling factor such that the lower the pheromone value, the higher the probability of more servers being added. The random value has the effect of generating multiple possible solutions across different ants.

If however the ant was morphed because of too much pheromone, then the nest does a similar initialization where it computes the number of servers to remove as:

$$ServerCount_{remove} = \frac{N}{2} * rand + \frac{N}{2} * \frac{p_{ant} - Pt_{max}}{Pt_{max}} \quad (15)$$

Similar to the add case, the higher the pheromone level compared to the maximum threshold the more servers will be removed from the cloud.

H. Self-organizing agent: House hunting ant

In order for an ant to determine the viability of a nest, the ant will check the number of servers to add or remove and simulate what effect that will have on the ants pheromone levels. If the cloud should increase in size, then the ant replaces a proportional number of servers in its history with empty servers and then recalculate the average pheromone across those servers. If the cloud should decrease in size, then the ant removes a proportional number of servers in its history and then redistributes the pheromone across the remaining servers.

For example, if the ant has in its history 10 servers and the nest has a desired server value of 15, then the ant will hide 3 of its servers and replace them with 3 servers with medium pheromone level and recalculate the average pheromone across the servers. If however the ant has in its history 10 servers and the nest has a desired server value of 7, then the ant will hide

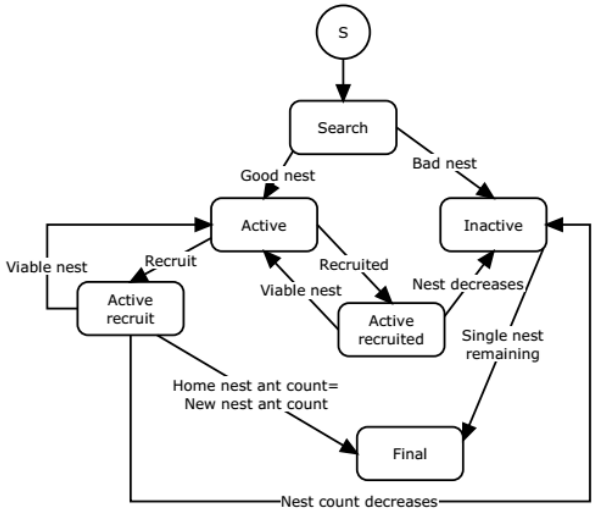

Fig. 6: Ant House Hunting State Diagram

3 of its servers and redistribute the pheromone from those 3 servers on the remaining 7 levels.

The house hunting algorithm is performed in rounds and ants can be in one of four states:

1) Search - This is the initial state of ants which search for a new nest
2) Active - The ant is committed to a good nest and tries to recruit other ants to it
3) Passive - The ant is committed to a bad nest and is waiting to be recruited
4) Final - A single nest remains so all ants go to it and that is the solution

The state diagram for the ants can be described as in Figure 6

In the first round all house hunting ants initialize a nest and go to it. The ants then simulate the performance of the solution as described previously. If the simulated result's performance is under a given threshold, then the ant goes into the passive state, otherwise it goes into the active state.

In the second round, all ants return to the home nest and active ants try to recruit other ants at the home nest. Passive ants can not be recruited until the final round when all ants go to the single remaining nest. Active ants recruit randomly from the other active ants at the home nest by choosing another ant to recruit and bring it to it's committed nest. In order to ensure no conflicts between recruiting ants, the ants recruit iteratively and an ant which was already recruited does not recruit someone else. This recruitment is biased by the perceived suitability of the nest by the ants, such that better nests are given priority. The bias is achieved by sorting the ants by the perceived viability and then randomly choosing a recruiter ant such that ants with better suitability have a higher chance of being recruiters and ants with lower viability have a higher chance of being recruited. This is achieved using the following formula, where the probability of an ant to recruit is defined as the viability of the ant's chosen nest divided by the sum of all the ant's chosen nests.

$$p_{recruit} = ant_{viability} / \sum_{i=1}^{n} ant_{viability} \tag{16}$$

Once an ants recruits another ant the two ants go together to the nest of the recruiting ant. When reaching a new nest, active ants count the number of ants at the nest and check if the nest they reached is the same as the previous nest they went to. Based on the number of ants and the nest there are three cases:

1) If the nest is the same nest that the ant went to before and the number of ants has increased or remained the same then the nest is still a possible solution so the ant updates the count and waits an extra round at the nest. After waiting a round, the ant checks if the number of ants at the home nest is the same as that at its current nest. If they are the same then the ant goes to the final state, new servers are added or removed and all ants morph back to ACO ants.
2) If the nest is the same nest that the ant went to before and the number of ants has decreased then the ant becomes passive because the nest is in the process of being dropped out. The ant returns home in the same round that active ants wait at the new nest.
3) If the nest is different then the nest the ant was committed to then the ant waits a round and then it checks to see if the number of ants at its new nest has decreased or not. If the number has decreased, then this nest is dropping out as the ants already committed to it have gone to the home nest and the ant becomes passive. Otherwise, the ant commits to its new nest and goes home to recruit other ants.

The above algorithm can be described as follows in pseudo-code in Figure 2. R_x represents round x.

I. Mathematical model: nest viability

In order to determine if a nest should be considered as a solution ants must determine the viability of the nest. Because it is impossible for ants to predict how the user requests would be processed by more/less servers the ants use a heuristic to approximate the viability of each solution. It is important to note that due to each ant having a different memory of servers it has recently visited, different ants will see the same nest as having a different viability score.

There are two cases to consider - one when a cloud is over-loaded and servers should be added, and one where the cloud is under-loaded and servers should be removed.

1) Over-loaded cloud: In the case of an over-loaded cloud, each of the nests will have its solution contain more servers than the count of servers known by each ant. As such in order to determine the viability of the nest, the ant needs to simulate how a cloud with the extra servers would behave. The ant only has a short memory of its last x visited servers and the amount of pheromone at these servers when it visited them. Because

Algorithm 2 Ant House Hunting Pseudocode

R_1: Go to new nest
Compute suitability of new nest
if Suitability $<$ threshold **then** Switch to passive
end if
R_2: Go to home nest
if Ant is active && Ant is not recruited **then**
    Recruit another ant
    Go to new nest
else if Ant is recruited **then**
    Go to new nest
end if
R_3: Count number of ants at new nest = $count_{new}$
if Nest is same and $count_{new} >= count_{old}$ **then**
    Wait round
else if Nest is same and $count_{new} < count_{old}$ **then**
    Switch to passive
    Go to home nest
else if Nest is different **then**
    Wait round
end if
R_4: Count number of ants at new nest $count_{new}$
if $count_{new} == count_{home}$ **then**
    Switch to final state
else if $count_{new} < count_{old}$ **then**
    Switch to passive
    Go to R_2
end if
Return final state
Switch ants to ACO ants

of this, the ant assumes that it would have visited the new servers with equal probability, so it replaces y servers in its history with servers with optimal pheromone level, where the optimal pheromone level is defined as:

$$p_{opt} = \frac{Pt_{max} + Pt_{min}}{2} \tag{17}$$

and

$$y = x * \frac{newServerCount - cloudSize}{newServerCount} \tag{18}$$

As such, the ant computes the average pheromone level across the last x servers it would have visited as:

$$p_{averageScaled} = \frac{y * p_{opt} + \sum_{i=y+1}^{x} p_i}{x} \tag{19}$$

Finally, the viability of the nest is defined as:

$$viability = 1 - \frac{|p_{averageScaled} - p_{opt}|}{p_{opt}} \tag{20}$$

With this approach, the ant simulates how the system would behave if it had y extra servers taking some of the load from the existing servers. It then computes the viability based on how close the simulated pheromone is to the optimal pheromone.

2) Under-loaded cloud: In the case of an under-loaded cloud, each of the nests will have its solution contain less servers than the count of servers known by each ant. As such in order to determine the viability of the nest, the ant needs to simulate how a cloud with the lower count of servers would behave. The ant computes a number of servers y in its history which should be removed as:

$$y = x * \frac{cloudSize - newServerCount}{cloudSize} \tag{21}$$

and then simulates the average pheromone for the cloud as:

$$p_{averageScaled} = \frac{\sum_{i=y+1}^{x} p_i}{x} \tag{22}$$

The viability of the nest is calculated as in the case of the overloaded cloud. With this approach, the ant simulates how the system would behave if it had y less servers and the remaining servers are taking the load from the removed servers. The viability is computed based on how close the simulated pheromone is to the optimal pheromone.

IV. Simulation Environment

In order to test the performance of the self-organizing system previously described, a simulation was created on top of the CloudSim and CloudSimEx systems. CloudSim is a framework for modelling and simulating cloud systems in which various tests can be run - for example the placement of tasks in a cloud or the performance of a cloud system. CloudSim allows users to define datacenters which are composed of hosts which can run virtual machines and virtual machines which run on top of the hosts. Both hosts and virtual machines are defined in terms of million instructions per second (MIPS) and memory available. Once a datacenter is defined, a workload can be defined as well to be executed in the datacenter. CloudSim workloads are called cloudlets - a cloudlet represents a user task and defines how many million instructions (MI) the task takes as well as how much memory the task needs. As such a cloudlet's execution time will depend on its' lifetime in MIs and how many MIPS the VM the cloudlet is allocated to has. The usage of MIPS and MI to represent loads and available resources make it complicated to define workloads that would represent real world behaviours.

CloudSimEx extends CloudSim by adding the concept of web sessions. Internally a web session is composed of a number of web cloudlets, where each web cloudlet has a small RAM and MIPS requirements and later web cloudlets in a web session can not be completed until all previous web cloudlets have completed. Web sessions also have the property of targeting both application server VMs and database VMs, with the web session alternating cloudlets running on either application server or database server. Furthermore, CloudSimEx also adds the concept of scaling policies which can be applied on a cloud. A scaling policy adds or removes servers to/from

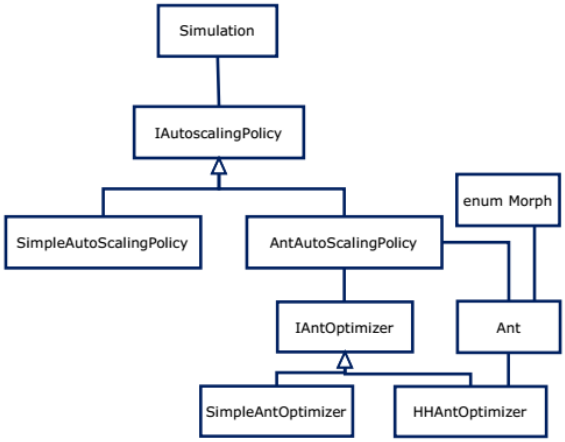

Fig. 7: Simulation Framework System

the cloud when the policy determines that some conditions are breached. A basic policy is implemented in CloudSimEx which scales the cloud when the CPU usage passes certain thresholds and also includes a quiet time after a scaling action has been performed. The basic policy implements an interface which makes it easy to add other scaling policies.

Because CloudSim and CloudSimEx only include CPU usage and memory usage as measurable metrics, the simulation of the self-organizing system uses the CPU usage as an input for the ACO algorithm to determine how the pheromone should be updated. In order to run the self-organizing system in this paper and compare it to existing results, a number of workloads which are predefined in CloudSimEx are used.

The architecture of the simulation framework is presented in Figure 7.

The simulation runs three different scaling policies for comparison. The first scaling policy used is the simple scaling policy provided by CloudSimEx. This scaling policy measures the average CPU usage of the cloud and when predefined thresholds are breached it either adds one server if the upper threshold is breached or removes one server if the lower threshold is breached. The policy also includes a quiet time, which is a period of time after an action was taken during which no other action will be taken. For the purpose of these tests the thresholds for the simple scaling policy are set to:

- Upper threshold is set to 80% average CPU utilization across the cloud
- Lower threshold is set to 10% average CPU utilization across the cloud
- Quiet time is set to 150s during which no other action will be taken

The values chosen match the default values for the Amazon EC2 auto scaling policy.

The second policy is based on the predictive ACO algorithm presented in this paper but without the house hunting algorithm to determine the number of instances to scale up or down. The policy uses the ACO algorithm to determine when instances should be added or removed. When a decision is taken that servers need to be added or removed a single server instance is added or removed at one time. The ACO policy does not need quiet time defined as it inherently includes quiet time due to the fact that it takes time for the pheromone to build up or decrease. For the purpose of these simulations the ACO algorithm runs with the following parameters:

- Decay is set to 1
- Decay period is set to 15s
- Minimum wait time is set to 1 at an under-loaded node
- Maximum ant pheromone is set to 4 added at an under-loaded node
- Ant memory size is set to 5
- Pheromone level for triggering server removal is set to 100
- Pheromone level for triggering server addition is set to 25
- CPU is considered to be balanced if it is between 45% and 55% utilization

The third tested policy contains both the ACO algorithm for prediction of SLA breaches and the house hunting algorithm for the optimization of the cloud size when a breach is detected. The tests are run with the same parameters as the second policy.

V. Results

The performance of the self-organizing system presented in this paper was run on top of the simulation environment presented in the previous section. A number of tests were run, with different workloads in order to determine the performance of the system and also compare how the developed ant based system performs compare to other auto scaling systems. Due to the fact that the workloads are generated stochastically, each of the tests was run 5 times and the results were aggregated. The performance tests only focus on the auto-scaling algorithm for the application tier and assume that the cloud has enough hosts and also that the cloud has enough database VMs to take care of the load. For this purpose all simulations are run with 60 host computers which can run VMs and 30 database VMs which are always up. Each of the hosts is defined as having 2 CPU cores, each of which provides 1000 MIPS. All of the application VMs are equal in power and they provide 250 MIPS to the simulated web application running on top of the VM. Each of the simulations is run for 48 hours, with a workload defined for 24 hours and which repeats after the first 24 hours. Workloads are generated using a Gaussian distribution.

A. Low workload, low number of VMs

The first simulation run is a simulation where the workload is low and as such not a large number of VMs are required to run at the same time in order to meet the desired SLAs for the application.

The workload for this test starts with a mean of 60 sessions per hour and a standard deviation of 1 and gradually increases towards a mean of 1000 sessions per hour with a standard deviation of 5, at 10 - 12 hours. This is followed by a small drop at 13 hours to 500 sessions after which the load goes back to 1000 sessions for hour until 17 hours. After this, the workload decreases gradually until it reaches 30 sessions per hour with a standard deviation of 1 at 24 hours. The

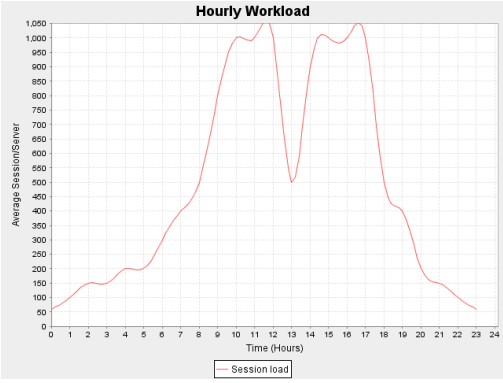

Fig. 8: Simulation - low workload

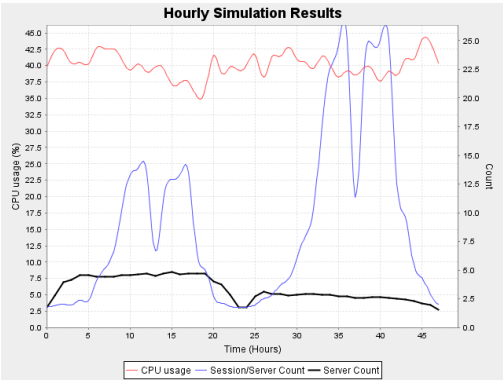

Fig. 9: Simulation - Low workload: Simple autoscaling

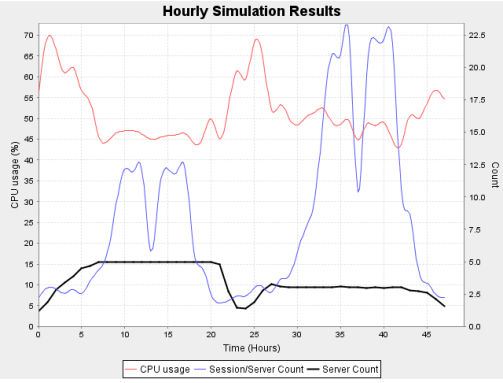

Fig. 10: Simulation - Low workload: Simple ant autoscaling

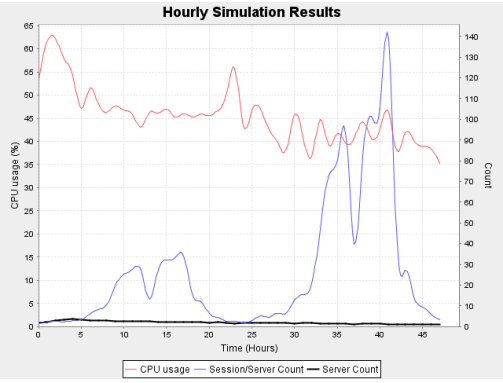

Fig. 11: Simulation - Low workload: House hunting ant autoscaling

distribution of the sessions per hour can be seen in Figure 8. This test simulates a usual workload seen in online services where the demand peaks at one point during the day.

Figures 9, 10 and 11 show the statistical results for the simulation runs. The results are averaged per hour and across the 5 runs for each auto-scaling algorithm. The graphs include average CPU usage as well as average number of sessions per server and average provisioned servers. Looking at the three graphs, it can be seen that the simple ant auto scaling algorithm obtains better CPU utilization than the simple auto scaling approach, and better than the house hunting algorithm also. At the same time, the simple ant auto scaling algorithm achieves this, while maintaining the server count more stable than the simple auto scaling algorithm.

Similar results can be seen in Table IV which summarizes the tests. Scale up and scale down count represents the number of times the optimizer decided to add or remove servers. A decision to add N servers, counts as a single scale up action. $Overprovisioned - x\%$ represents the time in seconds that the average utilization of the cloud was under $x\%$, suggesting low utilization and over provisioning of the cloud. Similarly $Highusage - x\%$ represents the time in seconds that the utilization of the cloud was over $x\%$. This could be either due to under provisioning if response times are delayed at the same time, or can be a sign of good provisioning if sessions still end fine. $Averagesessiondelay$ represents the average delay across all of the sessions, while $Maxsessiondelay$ represents the maximum delay. CloudSim defines a desired end time for a session, which represents when the session should end if there are no delays due to scheduling. Delay is defined as the time it took for the session to complete after this desired end time.

The simple scaling algorithm takes a lot more scale up and scale down actions than the two ant based algorithms. This is because as soon as the average utilization breaches the thresholds an action must be taken, while for the ant algorithms the detection is not triggered by a single drop or spike in utilization. At the same time, the simple ant scaling algorithm has a much shorter time when the cloud is over provisioned (too many servers for the load) compared to the simple scaling algorithm, and longer periods when the cloud is under high usage with over 70% and 80% CPU usage. The house hunting algorithm performs worse than the simple ant algorithm for this test, as it tends to over provision the cloud when a breach is predicted by adding too many servers. This can be seen, as it has higher scale up/down counts and also higher over provisioned times. The delay results in table V show similar behaviour with the simple ant algorithm behaving better than the simple scaling algorithm and the house hunting algorithm behaving worse.

TABLE IV: Low workload simulation results - utilization

Scaling Algo-rithm	Scale Up Count	Scale Down Count	Over Provi-sioned - 20% (s)	Over Provi-sioned - 40% (s)	High usage - 70% (s)	High usage - 80% (s)
Simple scaling	755	762	7140	88423	3048	1080
Simple ant scaling	128	132	2796	40104	16008	7260
House hunt-ing scaling	223	339	16656	53352	11184	5904

TABLE V: Low workload simulation results - delays

Scaling Algorithm	Average Delay (s)	Maximum delay	Average Servers
Simple scaling	28.8	118.6	3.35
Simple ant scaling	20.5	75.3	3.49
House hunting scaling	115.4	574.3	1.95

B. High workload, high number of VMs

The second test uses the same workload as the first test but scaled 5 times. With this scaling the simulation ends up requiring a large number of servers. The distribution of the sessions per hour can be seen in Figure 12.

Figures 13, 14 and 15 show the statistical results for the simulation runs. The results are averaged per hour and across the 5 runs for each auto-scaling algorithm. Looking at the three graphs, it can be seen that the simple ant auto scaling algorithm and the simple auto scaling approach behave very similar. Both increase the number of servers as demand increases, and then once demand decreases they both decrease the number of servers in the cloud. Due to the fact that the two algorithms add servers one by one, they end up under provisioning the cloud when demand is high and over provisioning when demand goes low. The house hunting algorithm behaves much better than the two simple scaling algorithms as it is able to keep up with the demand by adding more servers at one time and removing more servers at the same time. The same results can be seen in Table VI. The house hunting algorithm is able to achieve better server utilization - only 768 seconds of average CPU utilization under 20% compared to more than 20000s for the simple scaling algorithms, while performing less scale up/scale down actions. The delay results in table VII are very similar for all 3 scaling algorithms with average delay at, or near 0 and very small maximum delays. In such a case, the house hunting algorithm behaves better as it takes less actions and uses less servers than the simple scaling algorithm.

C. Very high workload, very high number of VMs

The final test uses a very high workload with an initial peak of 5000 sessions/hour after 5 hours, followed by a decrease in requests and a second much higher peak of 10000 sessions/hour after 12 hours. The distribution of the sessions per hour can be seen in Figure 16.

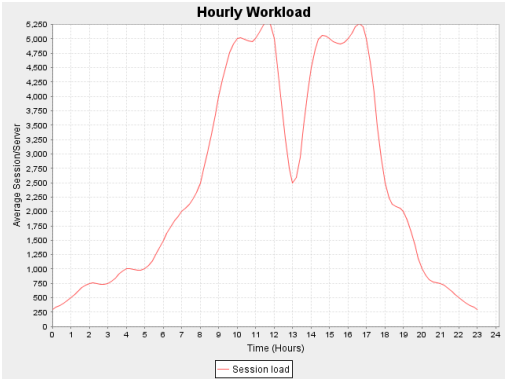

Fig. 12: Simulation - High workload

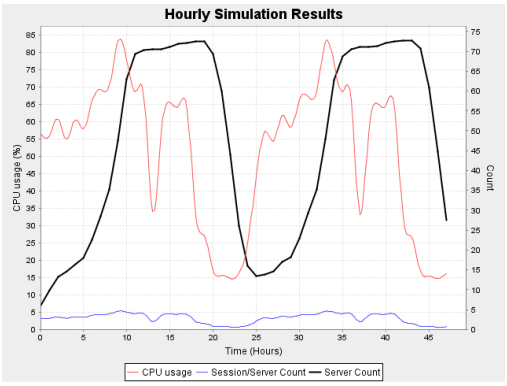

Fig. 13: Simulation - High workload: Simple autoscaling

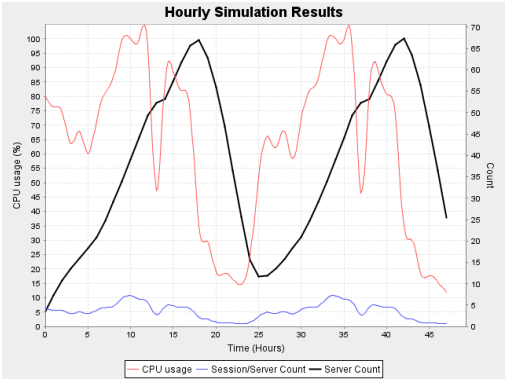

Fig. 14: Simulation - High workload: Simple ant autoscaling

TABLE VI: High workload simulation results

Scaling Algo-rithm	Scale Up Count	Scale Down Count	Over Provi-sioned - 20% (s)	Over Provi-sioned - 40% (s)	High usage - 70% (s)	High usage - 80% (s)
Simple scaling	717	693	26255	50503	8952	231
Simple ant scaling	661	656	22241	42750	85873	62593
House hunt-ing scaling	332	253	768	32643	35419	18631

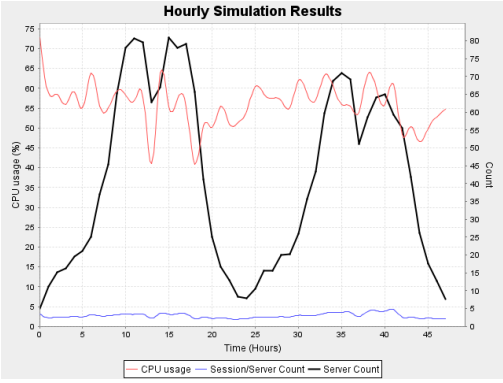

Fig. 15: Simulation - High workload: House hunting ant autoscaling

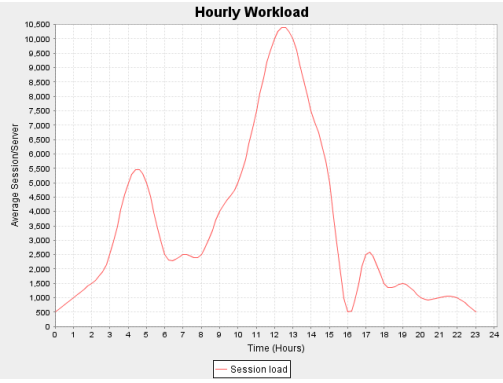

Fig. 16: Simulation - very high workload

TABLE VII: High workload simulation results - delays

Scaling Algorithm	Average Delay (s)	Maximum delay	Average Servers
Simple scaling	0	14.1	46.9
Simple ant scaling	1.1	15.3	37.7
House hunting scaling	1.1	28.4	40.9

Figures 17, 18 and 19 show the statistical results for the simulation runs. The results are averaged per hour and across the 5 runs for each auto-scaling algorithm. Looking at the three graphs, the house hunting algorithm again performs the best, however the simple ant autoscaling algorithm performs worse than the simple autoscaling algorithm. Both simple autoscaling algorithms have trouble as the demand increases quickly, but the simple ant autoscaling can not answer quickly enough. This is due to the fact that there's a certain time needed for the pheromones to pass the trigger level, and after the trigger is passed only a single server can be added. This results in long periods when all the servers are overloaded and average CPU usage is 100%. Compared to the two simple autoscaling algorithms, the house hunting algorithm behaves much better reaching the optimal number of servers quicker and maintaining this number of servers.

The same results can be seen in Table VIII. The house hunting algorithm is able to achieve better server utilization again - only 2000 seconds of average CPU utilization under 20% compared to more than 15000 and 46000s for the two simple scaling algorithms, while performing less scale up/scale down actions. The delay results in table IX show the house hunting algorithm with slightly higher average delays than the other two algorithms, and much higher maximum delay. This can be explained by the fact that the other two algorithms overprovision - seen by the over provision metrics, while the house hunting algorithm underprovisions in a worst case.

The performance tests show that the system behaves as desired and that the ACO algorithm properly detects a breach of SLA when the cloud is under loaded or over loaded while the house hunting algorithm can determine the number of servers added or removed. Due to the randomness which exists

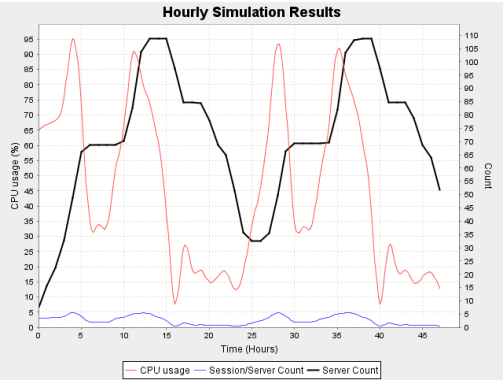

Fig. 17: Simulation - Very high workload: Simple autoscaling

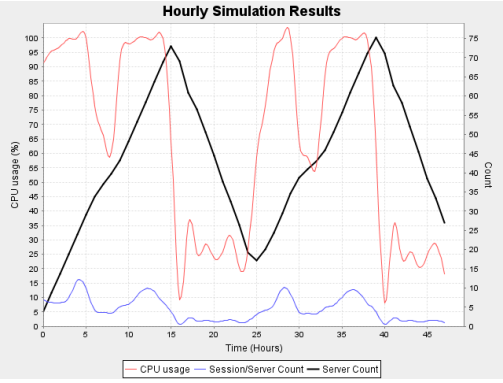

Fig. 18: Simulation - Very high workload: Simple ant autoscaling

TABLE VIII: Very high workload simulation results

Scaling Algorithm	Scale Up Count	Scale Down Count	Over Provisioned - 20% (s)	Over Provisioned - 40% (s)	High usage - 70% (s)	High usage - 80% (s)
Simple scaling	1004	833	46339	86035	42768	20604
Simple ant scaling	726	679	16412	54641	83245	76837
House hunting scaling	354	287	1956	33864	32151	22080

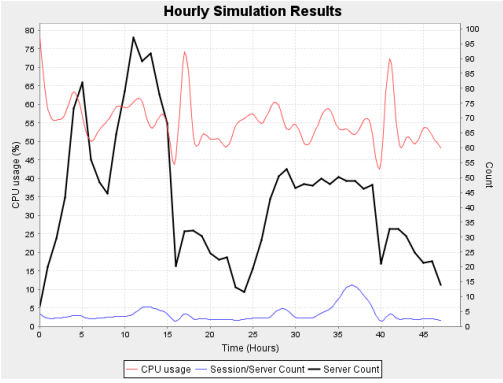

Fig. 19: Simulation - Very high workload: House hunting ant autoscaling

TABLE IX: High workload simulation results - delays

Scaling Algorithm	Average Delay (s)	Maximum delay	Average Servers
Simple scaling	1	27.6	70.6
Simple ant scaling	3.9	22.5	43.2
House hunting scaling	10.7	192	42.8

inside the house hunting algorithm it is possible to overshoot or undershoot the correct number of servers, however this is corrected quickly after another SLA breach is detected.

VI. Conclusion And Future work

This paper has presented a self-organizing SLA breach prediction algorithm based on the ACO algorithm. This algorithm is capable of predicting a future SLA breach by having small agents - ants - move from server to server based on a pseudo random chance and use the information at the servers to lay down pheromones for other ants to use in their calculations. The pheromone level across the servers then acts as a proxy for how overloaded is the server cloud.

In this paper the amount of how much pheromone to add and how long an ant waits at a server is based on a single value which represents the stable cloud. It would be better if the calculation was using a range of values for when the cloud is stable, otherwise the system can lead to oscillations. Future work will be done to improve the algorithm and also add a robust self-organizing plan function to decide how many servers to add or remove from the server when an SLA breach is detected.

References

[1] B. Solomon, D. Ionescu, C. Gadea, S. Veres, M. Litoiu, and J. Ng, "Distributed clouds for collaborative applications," in *Collaboration Technologies and Systems (CTS), 2012 International Conference on*, may 2012, pp. 218 –225.

[2] B. Solomon, D. Ionescu, C. Gadea, and M. Litoiu, *Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environments*. IGI Global, 2013, ch. Geographically Distributed Cloud-Based Collaborative Application.

[3] C.-H. Chu, J. Gu, and Q. Gu, "A heuristic ant algorithm for solving qos multicast routing problem," in *CEC '02: Proceedings of the Evolutionary Computation on 2002. CEC '02. Proceedings of the 2002 Congress*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 1630–1635.

[4] K. M. Sim and W. H. Sun, "Ant colony optimization for routing and load-balancing: survey and new directions," *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, vol. 33, no. 5, pp. 560–572, Sept 2003.

[5] A. L. Cronin, "Consensus decision making in the ant myrmecina nipponica: house-hunters combine pheromone trails with quorum responses," *Animal Behaviour*, vol. 84, no. 5, pp. 1243 – 1251, 2012.

[6] M. Ghaffari, C. Musco, T. Radeva, and N. Lynch, "Distributed house-hunting in ant colonies," in *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, ser. PODC '15. New York, NY, USA: ACM, 2015, pp. 57–66. [Online]. Available: http://doi.acm.org/10.1145/2767386.2767426

[7] S. C. Pratt, E. B. Mallon, D. J. Sumpter, and N. R. Franks, "Quorum sensing, recruitment, and collective decision-making during colony emigration by the ant leptothorax albipennis," *Behavioral Ecology and Sociobiology*, vol. 52, no. 2, pp. 117–127, 2002. [Online]. Available: http://dx.doi.org/10.1007/s00265-002-0487-x