Cloud Chamber: A Self-Organizing Facility to Create, Exercise, and Examine Software as a Service Tenants

M. Brent Reynolds  
Naval Surface Warfare Center  
Crane Division  
michael.reynolds@navy.mil

Don R. Hulce  
H2 Support Services, LLC  
don.hulce@h2supportservices.com

Kenneth M. Hopkinson  
Air Force Institute of Technology  
kenneth.hopkinson@afit.edu

Mark E. Oxley  
Air Force Institute of Technology  
mark.oxley@afit.edu

Barry E. Mullins  
Air Force Institute of Technology  
barry.mullins@afit.edu

Abstract

Cloud Chamber is a testbed for understanding how web services behave as tenants in a Software as a Service (SaaS) environment. This work describes the Cloud Chamber testbed to investigate autonomic resource management of web services in a cloud environment. Cloud Chamber is a virtualized environment which provides web servers as services, facilities to apply loads to the tenant services, algorithms for autonomic organization and reconfiguration of service assignments as demand changes, and sensors to capture resource consumption and performance metrics. The testbed inserts sensors into web servers to collect the resource utilization of CPU cycles, memory consumption, and bandwidth consumption of the individual web services, the web server, and the operating system. This high resolution performance data generates profiles of the resource usage of each web service and the resource availability of each server. The testbed, as described in this work, utilizes these profiles to efficiently place services on servers, thus balancing resource consumption, service performance, and service availability. Once services have been placed, the testbed monitors changes such as traffic levels, server churn, and the introduction of new services. The information gathered is used to calculate configurations of service placement which better meet the changing requirements of the environment.

1. Introduction

Software as a Service (SaaS) is the name given to the cloud computing model in which a vendor provides to the consumer access to software running on the vendor’s hardware. This relationship allows the consumer to utilize all the benefits of the software without the cost and challenge of administering the hardware and software themselves. This also provides the vendor with an economy of scale.

This work specifically discusses web services as tenants of a web server. The work can be extrapolated to include database services and other middleware services where the customer executes software (services) as request and response transactions. These tenants are scripts, executables, or objects requested from the outside which execute in a server side process and return data. The server side process is a web server, or even a database server.

As the technology improves, as the economies of scale are realized, and as the administrative freedoms increase, cloud based computing models, such as SaaS, have the potential to become the dominant mechanism of providing services to end users. Given this growth the providers of SaaS need increased autonomicity of these systems. The large scale environments of the future will out pace the abilities of individual administrators to monitor their performance and implement system level policies.

The Cloud Chamber, introduced in this work, is a testing environment where services are modeled, executed, and autonomic policies and methods are tested. We believe emulating services in real life web server environments is key to understanding how to model and measure the behavior of services in the wild.

Specifically our requirements for the Cloud Chamber included but were not limited to the following. The ability to test heterogeneous services on heterogeneous servers is critical. Services of...
different shapes and sizes execute differently on servers of different shapes and sizes. The environment should be limited to a virtualized environment such as VMWare, Xen, etc. Performance data will be collected in real-time to a SQL database. The traffic of requests to the services can be generated based on a pre-recorded load of arbitrary length. The assignment of services to servers will be autonomic based on interchangeable algorithms and numeric methods which react to changes in the environment such as increased traffic load, loss of a server, or policy changes.

Given the above requirements, this work discusses the foundations of the Cloud Chamber, its architecture, and some of the pitfalls discovered along the way. This concludes with an experiment demonstrating its application and exploring its possibilities. The next section presents related and foundational works by others regarding testbed environments and the mechanisms employed in Cloud Chamber. The third section discusses the virtual environment, how services are created, and the components of the Cloud Chamber. The fourth section presents how the services and servers are mathematically modeled, measured, and finally organized into profiles. The fifth section outlines the self-organizing, autonomic aspects of the Cloud Chamber such as its gossip network and heuristic search methods. The work finishes with an experiment demonstrating the autonomic features of the Cloud Chamber.

2. Related Works

Our research focuses on the Tenant Placement Problem and is most generally defined as an optimization constraint problem, a multi-dimensional, multi-choice knapsack problem [1], and a specialization of the Application Placement Problem (APP) [2]. In [3], Zhang et al. describe the On-Line Tenant Placement Problem (OTPP). It differs from the APP and the Online APP [2] because applications do not require containers, e.g. these are executable code. Tenants in SaaS reside in a service container such as a web server, database server, or middleware server. Zhang et al. consider a multi-resource environment where each tenant requires multiple types of resources such as processor utilization, memory consumption, and bandwidth. Unlike our work, they assume that tenants (services) can not be moved once assigned. They further assume that more servers (nodes) can be added as needed to meet demand, i.e. no under-provisioning is allowed.

Urgaonkar et al. [2] describe applications as having multi-tier capsules. They demonstrate that APP is NP-Hard. In the offline condition/ environment/scenario APP is addressed as a matching problem on a bipartite graph with LP-relaxation. Online APP is shown to reduce the placement problem to the minimum-weight perfect matching problem. The application capsules and servers are described using only a single resource value.

This article focuses on the Cloud Chamber, a service testbed created to specifically investigate the self-organizing solution to the on-line web service tenant placement problem. Other works mentioned below are web service or service oriented architecture (SOA) testbeds, each with subtle differences.

Juszczynk et al. [4], outlines the Genesis2, the most similar work to ours. Genesis2 is a framework in which varieties of SOA testbeds can be created. It allows researchers to model web services from which it creates an executable testbed. Genesis2 provides generation of traffic, monitors performance, allows for plug-ins, and modification on the fly. As of this writing, it is not yet available for download. Its primary difference with our work lies in its generalization. Our testbed is specifically interested in testing on-line self-organizing methods.

Bianculli et al. [5] describe SOABench as a framework for the automatic generation and execution of testbeds for benchmarking middleware in SOA architecture. In the work SOABench tests three middleware servers, finding their weak points under heavy loads. This framework differs from ours, but not significantly, due to its focus on middleware. They are not focused on placing or moving services in reacting to performance but are more focused on investigating static performance in order to find weaknesses.

Bertolino et al. [6] offer PUPPET as a means to test web services under development that rely on unavailable external web services. The external services are unavailable for testing due to various reasons such as implications to production, cost of usage, or other side effects. PUPPET creates a stub for the otherwise absent services that perform per its prescribed functional and service level agreements.

Ricci et al. [7] describe the network testbed mapping problem. The network testbed consists of compute nodes, routers, switches, links, etc. A test scenario needs a set of these resources. Ricci et al. focus on a solution to quickly select a mapping of requirements to resources in an NP-Hard problem. This is similar in regards to the mapping but outside the area of services.

Several other works implementing a real world web service testbed deploy only a single service. Although these are multi-tiered or composite services, they differ from ours in that ours is a collection of heterogeneous services on heterogeneous servers. Two quality,

In other autonomic service oriented architecture works, [12], [13], [14], the focus is based on maximizing profits while maintaining multiple tier service level agreements. Almeida et al. in [12] break the resource allocation problem into a short term arrangement problem and a long term allocation problem. They utilize queuing and optimization techniques in their model and performance measurements. Ardagna et al. in [13] similarly model and measure with queuing and optimization techniques to assign virtual machines to CPUs in a physical server. A self-adaptive capacity management framework described in [14] uses queuing and optimization in a multi-tier virtualized environment to demonstrate significant profit gains relative to a static model.

As discussed in the self-organization section, the testbed currently utilizes a combinatorial heuristic [15] similar to [16] to quickly find quality configurations measured by the Provisioning Norm [17]. Resende [16] describes a multi-start, hybrid heuristic. The heuristic using a tunable parameter is defined as multi-start by its use of several points in the search space as starting points. Each start point creates inputs into each phase of the heuristic. The hybrid nature of the heuristic is the combination of several methods, each is a phase in the process. These include local search, path-relinking, elite solutions, intensification, and post-optimization.

Kempe et al. [18] provide an analysis of simple gossip-based protocols. In particular they describe the information dynamics of these protocols and the effective fault-tolerance and self-stabilization of the population as a whole. They investigate mathematically the probability of diffusion using Uniform Gossip, which can be described as a simple push gossip protocol as implemented in this work.

In [19], Jenkins et al. describe a gossip-based multicast protocol implemented in an environment of multiple process groups. Their modified push gossip protocol, referred to as Gravitational Gossip, provides a mechanism for trading off the quality of information updates to reduce the amount of overhead required to deliver messages. Whereas in their work the participating nodes are characterized by values of infectivity and susceptibility to determine the required quality of information updates, in our work we characterize the messages themselves to determine the required quality of information updates and from that the mechanisms by which they are reconciled into views, viz. instantaneous node messages are all equally infectious, but the best quality configuration is required to be highly infectious.

3. Servers and Tenants

The Cloud Chamber provides an environment for the placement of tenants (services) on node resources. Clients execute services based on pre-defined loads. These loads produce performance data for both the services and the nodes. This data, using on-line analysis, determines a new service-to-node configuration. A distributed configuration manager implements a configuration by reassigning services to nodes.

The architecture of the testbed is comprised of commercial off the shelf (COTS) hardware and software as well as custom applications. The test bed’s physical hardware includes two servers. The primary server, HP Proliant ML330G6 with dual Intel Xeon E5620 processors and 36GB of memory, runs VMWare ESXi 4.0. The other server houses a web server for administrative interaction, such as starting and stopping tests, and a SQL database engine for data collection and storage.

The VMWare ESXi 4.0 [20] server houses virtual machines (nodes with resources). These virtual machines have TinyCore Linux 3.0 [21] installed as the operating system. TinyCore is a minimal Linux installation without all non-essential services. In addition to this minimal core of Linux, the nodes have a small web server, lighttpd. In VMWare ESXi, each virtual machine has limits on its virtual hardware resources such as CPU speed, memory capacity, hard drive capacity, and network bandwidth. By varying these resource limitations across machines, a heterogeneous set of compute resources is created.

Each service is executed as a fast common gateway interface (FastCGI) executable written in C. FastCGI [22][23][24] is a protocol defining the method to pass a web request from a web server such as Apache, IIS, and lighttpd to an application server and vice versa, such as Perl, PHP, and others. The FastCGI application, serviceman, emulates a web service by consuming a prescribed amount of system resources.

Currently, each service consumes CPU cycles, memory, and bandwidth based on scalar values assigned to each service. For example, a service described respectively by 100, 2000, 40 executes a for loop 100 microseconds, uses 2000KB of memory, and returns 40KB of data over the network. These values define the service and represent the service’s code behavior. This FastCGI application could include additional resource consumption functions such as
reads and writes to permanent storage. These values are stored in a service file like `service1.svc`. An http request to the web server arrives in the form `http://10.10.10.10/service1.svc`. The service file is passed to serviceman by the FastCGI application. Serviceman proceeds to consume the CPU and memory and finally returns a payload of prescribed size to the web server. The web server finally returns the payload to the requester consuming the network bandwidth.

In order to allow services of various shapes and sizes, the Cloud Chamber currently supports three distributions from which the consumption values are derived. The values are stored in the service (.svc) file as parameters to distribution functions. One distribution is fixed in which the value is the same for every request for the service. The second supports a distribution which is uniform and random and in which a lower bound and upper bound prescribe a range over which the value will be uniformly derived. The third is the standard distribution in which a mean and a standard deviation determine the random values. Other options such as Poisson, exponential, etc. will be added. If the tester wishes to override these values, the alternative values are passed to `serviceman` via parameters in the URL.

In developing this consumption emulator, `serviceman`, a handful of notable observations were encountered. Primarily, these derived from compiler and operating system optimizations. For memory, a simple `malloc()` call to acquire memory is insufficient to consume memory in such a fashion for the system methods of measuring memory consumption to notice. First if the requested memory is never accessed, the operating system (in this case a recent flavor of Linux, TinyCore) does not necessarily completely allocate the memory. Secondly, the service execution times, e.g. time between `malloc()`, memory accesses, and `free()`, were sufficiently small that utilities for system monitoring, like top, did not sample the system fast enough to see substantial memory utilization. The current work around is that `serviceman`, upon the first invocation of a service, allocates memory which persists between invocations of the service. This can be thought of as the service’s `stateful` memory requirements. The service’s `stateless` memory requirement are also allocated and deallocated during each invocation.

CPU cycles are consumed using a for loop. The `clock_gettime()` function provides the number of nanoseconds for which a process or thread has executed on the CPU so far. This function is called at the beginning of the consumption, before the memory has been allocated and accessed and data sent to the network. This allows for the capturing of the CPU time consumed by memory consumption and data sent to the network. After these two tasks, the for loop is executed a specified amount of microseconds emulating additional CPU consumption. Compiler optimization issues arose when attempting to do busy work in order to burn CPU cycles. In one example, writing the same value to the same place multiple times did not take as long as expected. In order to avoid any IO blocking and other opportunities for the service to be preempted, the for loop is a loop performing only an increment of a counter.

The services are invoked using a custom application. The `loadrunner`, an application based on `http_load[25]` and similar to `httperf[26]`, executes the requests on each service. The load of requests for each service is the number of hits per second for each time step, over n time steps. Each `loadrunner` is capable of simultaneously handling the requests for multiple services. As determined in the initial testing and sandboxing, the `loadrunner` application is able to deliver up to a combined total of 500 http requests per second. In order to achieve more than 500 http requests per second, multiple nodes are deployed. Each

![Figure 1. High Level Architecture](image1.png)

![Figure 2. Inside the TinyCore Node](image2.png)
executes its own instance of the loadrunner application.

The loadrunners expect http responses as defined by the RFC 2616 [27]. Successful executions of the service return an http status code of 200. Once a service is moved, the next request by the loadrunner in charge of the moved service will receive a “301 Moved Permanently” http status code with the new IP address as the content of the response. The loadrunner updates its information and all new requests are directed to the new IP.

Data is collected about the node performance and the web server performance. Serviceman reports data every second about the node’s performance such as CPU utilization, memory consumption and bandwidth. Serviceman also reports every second the number of responses it has handled and other supporting data. Loadrunner reports every second about its requests such as response times, open connections, http codes, and other supporting data. This data is sent to the SQL Server via a UDP datagram.

4. Profiles, Models, and Measures

Services require specific computational resources to function. These resources include, but are not limited to, CPU cycles, short term memory, long term storage, and bandwidth. As discussed in [2], these requirements can be seen as a type of workload profile for the service. The profile is a vector of numeric values indicating the service’s resource requirements. Each element of the vector represents the required amount of a type of resource. Each entry is normalized to the interval [0,1]. In the Cloud Chamber services have three categories of resource requirements: CPU, memory, and bandwidth; a service is described using a three valued vector, <0.02 0.10 0.04>. These vectors are referred to as profiles through out this article.

In the Cloud Chamber, a service is an executable program on an individual virtual web server (a node), and one node can service many services. A more complex, hierarchical service (N-tier) requiring multiple computational nodes is decomposed into its separate (albeit dependent) constituent services. Each treated as an individual.

The services’ resource requirements are fulfilled by the physical or virtual computational nodes upon which the services are executed. Similar to a service, a node is modeled as a vector of resources. The node’s resource vector indicates the resources provided by the node to the system. The number of elements in the node vector is equal to the number of elements in the service vector; each corresponding to CPU, memory, and bandwidth. Note: the computational models described here are not limited to three resource values.

To reiterate, the profile is a numeric representation of resources. A node’s profile is based on its resource capacities. A service’s profile is based on its resource consumption. The profile results from the quantification of resource measurements into a range of normalized values. The profiling performed in the paper assumes that the different resources are to be weighted equally. In order to accomplish this, the normalized value of each resource belongs to [0,1].

For the nodes, the CPU MHz ranges up to 2000 MHz, the memory ranges up to 512 MB, and bandwidth ranges up to 100 Mbs. A node with resources 2000 MHz, 512 MB, and 100 Mbs is profiled as <1.0, 1.0, 1.0>. A node with resources 500 MHz, 128 MB, and 1 Mbs is profiled as <0.25, 0.25, 0.01>.

Each virtual machine’s virtual hardware constraints are implemented in VMWARE. Memory size is defined when the virtual machine is created. CPU and network card speeds are not yet definable in the virtual machine definitions. CPU speed is defined in Resource Allocation. In VMWare ESXi 4.0, individual port groups are assigned to each virtual machine. A port group has traffic shaping settings that allow for throttling bandwidth. Three levels of each resource type are defined for the testbed. CPU levels are 2000MHz, 1000MHz, and 500MHz. Memory levels are 512MB, 256MB, and 128MB. Bandwidth levels are 100Mbs, 10Mbs, and 1Mbs. This allows for 27 different sized nodes. A 1000MHz, 256MB, 10Mbs node is profiled as <0.5 0.5 0.1>. As the technology exists, the node’s resources can change on the fly. The bandwidth limiting values can change without rebooting the node because this is a function of VMWARE. Similarly the limits on CPU speed can be changed on the fly. Memory changes can only take effect once the node has been rebooted. For this paper node resources are considered fixed.

Services are similarly profiled on the same relative scale. The Cloud Chamber dynamically profiles services during execution. This is a four step process: data collection, data aggregation, best fit linear model, and finally normalization to the aforementioned normalized scale. Per second consumption of each of the three resource types is logged into a revolving Resource Usage Window. The windows are an array with an element for each second. The window is 20 minutes long and wraps around as needed. Similarly, the services’ requests (hits) per second are recorded into a Hits Window. Periodically, the Resource Histogram and Hits Histogram are built from the data in their respective windows. The Hits Histogram indicates the various rates of traffic experienced by the service. The Resource Histogram is the per hit mean
of resource consumption at various rates of traffic experienced by the service. For example, in the nth position in the Hits Window, the rate is 14 hits per second. The corresponding values in the nth Resource Usage Window position are accumulated in the Resource Histogram at the 14th position. Once built, the Resource Histogram is a scatter plot of the service’s per hit resource consumption at various levels of hits per second. Using Box-Muller [28], a linear model is developed for each resource type. This linear model takes hits per second as an input and yields the corresponding expected resource consumption. The method of determining hits per second is left for later inspection, and as of this writing it is an average of the last 60 seconds. This mean hits per second is the input to the linear function yielding a value approximating the total amount of expected resource usage. This linearly approximated value is normalized to the interval [0,1] by dividing it by the maximum value represented by the normalized interval, such as 512MB or 100Mbs. These three values, CPU, memory, and bandwidth are the service’s profile. The profile is fed to the nodeman process for self-organization, described in the next section.

5. Self-Organization

The nodeman process implements a gossip messaging protocol between nodes to (1) maintain the view of nodes and services within the population and (2) generate a configuration based on these views. The purpose of the gossip network is to develop an environment of nodes which autonomically discovers the population of both nodes and services and autonomically calculates a configuration placing services on nodes.

Nodes send messages in rounds to propagate information contained within their views. A view is defined as a set of data on which the population of nodes needs to agree. Nodes maintain separate views for nodes, services, epochs, and configurations of service placement. During each messaging round, a source node selects an element from a view and sends that element to a target node selected at random. Every node participates in sending messages during every messaging round, but target nodes are selected at random, and thus not every node is guaranteed to receive a message during each messaging round. This method of selecting an element from a view and sending it to a randomly selected population member is best characterized as a simple push gossip protocol [18].

During node initialization, each node joins the population and becomes self-aware through a UDP broadcast message. A node continues to broadcast its existence in each messaging round until it is both self-aware and not alone. Once the node view has become aware of a population, services are introduced through the interface with the serviceman process. An initial population of nodes with an awareness of services enters into the first of three population time periods. The behavior of nodes cycles through these three distinct periods which are referred to collectively as an epoch and individually as search, reconciliation, and settle.

The search period is a process in which nodes and services are both discovered and updated. Each node infects other nodes with elements from its node and service views, new nodes and services are discovered by the population, and nodes and services expire from the population. This allows the population and service placement to autonomically adapt to changing network environments. The node and service views are instantaneous views which are persistent across all period transitions, i.e. they are not dependent upon a specific epoch. If the nodes have a reconciled view of nodes and services, i.e. a view which is unique to a given epoch, each node independently generates a configuration using the heuristic defined in [15] and calculates the quality measure of each using the Provisioning Norm defined in [15].

One or more nodes initiate a transition into the reconciliation period. During this period, the population reconciles the instantaneous node and service views into static epoch views which are used in the calculation of configurations. If configurations were generated during the previous search period, each node also sends messages regarding the best configuration it has calculated, or the best configuration that it has discovered from another node. These configurations are validated and compared by quality measure, resulting in a best configuration which is highly infectious to the population through iterative messaging rounds [19].

A third period, the settle period, follows reconciliation. This settle period accommodates variations in node awareness of the timing of the period transitions. The relative end time of the reconciliation period is synchronized through a timing message which is pushed throughout the reconciliation period. Nodes in the settle period are immune from infection by messages initiating period transitions. This ensures that all nodes have completed the reconciliation process and prevents epoch churn initiated by errant nodes which have joined the population between time synchronizations.

Several tunable parameters were externalized to allow administrative adjustment of the population behavior. The frequency of messaging rounds, the
duration of the search period, the duration of the reconciliation period, the duration of the settle period, and a Time-To-Live (TTL) value for individual nodes are set to satisfy the requirements of timely message propagation while limiting network traffic and computational overhead.

The need for parallel computation of service to node configurations required consensus of node and service views across all nodes in the population. Consensus ensures the validation of a calculated configuration and permits comparison and selection of a best configuration. As the implemented gossip protocol is a probabilistic messaging system [18], an adaptation was required to satisfy the need for deterministic population consensus. The solution was to implement a reconciliation phase that reduces the node and service views to the minimum set of elements shared by all nodes within the population.

Note that the set of nodes and services populating the epoch view is the set which was reconciled from the previous search period. The persistent node and service views may be different than the set of nodes and services in an epoch view. Each epoch retains its own view of nodes and services upon which the configuration generated during that epoch is dependent. A node may have up to three other distinct views of nodes and services in addition to its persistent views. It may also have the view that will participate during the next epoch and which is currently being reconciled, the view that corresponds with the configuration calculated during the previous epoch, and the view which is associated with the configuration that was implemented during the previous epoch.

Table I illustrates the relationship between the view of nodes and services and their presence during epoch periods.

<table>
<thead>
<tr>
<th>Set of Nodes &amp; Services</th>
<th>E0</th>
<th>ER0</th>
<th>E1</th>
<th>ER1</th>
<th>E2</th>
<th>ER2</th>
<th>E3</th>
<th>ER3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current View</td>
<td>E0</td>
<td>ER0</td>
<td>E1</td>
<td>ER1</td>
<td>E2</td>
<td>ER2</td>
<td>E3</td>
<td>ER3</td>
</tr>
<tr>
<td>Configuration In Process</td>
<td>-</td>
<td>-</td>
<td>ER0</td>
<td>-</td>
<td>ER1</td>
<td>-</td>
<td>ER2</td>
<td>-</td>
</tr>
<tr>
<td>Configuration Implemented</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>ER0</td>
<td>ER0</td>
<td>ER1</td>
<td>ER1</td>
<td></td>
</tr>
</tbody>
</table>

E – Epoch, ER – Epoch Reconciliation

6. Experiment

The experiment presented here demonstrates the self-organizational properties of the Cloud Chamber. The experiment compares the performance of two scenarios measured in mean response time. The first scenario tests services in the Cloud Chamber without autonomic reconfigurations. The second scenario tests services in the Cloud Chamber with autonomic reconfigurations. Ten nodes as previously are available for providing resources to services. The nodes’ specific profiles are described in Table II. Each scenario started with the same thirty services, described in Table III. Each test was run for 25 minutes and repeated 20 times.

Each trial of the test used the same service load values. This load is plotted over time in Figure 3. Five of the thirty services (services number 1-5 in Table 3) start with a high amount of traffic for the first ten minutes, decrease over the next five minutes, and finally reach a low traffic rate for the final ten minutes. Five other services (numbered 6-10) behave in an opposite fashion going from low to high over the same 25 minutes. Ten other services (numbered 11-20) of medium size are called at a steady, medium rate for the entire 25 minutes. The last ten services (numbered 21-30) are relatively large in consumption but are set at a steady, low level of traffic for the entire execution.

Initially the services’ profile values are all zeros because there has been no traffic on which to calculate the profile. Therefore as the services are placed into the Cloud Chamber, the first configuration, before the loads have been applied are based on profiles of zero. The time required to generate sufficient traffic to calculate an accurate profile and for that profile to propagate throughout the population and be included in the calculation of a configuration is approximately four
minutes. During this bootstrapping timeframe the services do not perform well because they are not of good quality. In the first scenario (without continuous autonomic reconfiguration), reconfigurations are allowed only for the first four minutes of the first trial. This allows for a good configuration to be found prior to the inhibition of reconfigurations throughout subsequent trials of this scenario.

The hypothesized results are as follows. As the traffic load of some services decreases while the traffic load of other services increases, service profiles will respectively change. As these calculated profiles shrink and grow, the scenario with reconfigurations will outperform the scenario without reconfigurations.

### 7. Results

The results, as expected, demonstrate that reconfiguring service placement provides better performance as conditions change. Figure 4 shows the mean response times of both scenarios over the 1500 seconds. Per Figure 3, the traffic changes at t=600, t=750, and t=900. The center of the chart shows that at time t=750 and t=900 significant disruption of response time occurs. From the scenario without reconfigurations, the response time increases from less than 10ms to 500ms, and never returns. This is in contrast to the scenario with reconfigurations where disruptions occur, but are not as high and return to near previous overall performance.

Also of interest is the exceptional performance of the scenario without reconfigurations during the first 600 seconds. This is due to the fixed configuration that was calculated in the bootstrap timeframe of the initial trial for the level of traffic occurring during the first 600 seconds. The initial 240 second hump of the scenario with reconfigurations is the bootstrapping timeframe of each trial where a quality profile is not found until t=240. At t=750 and t=900, the system with reconfigurations shows increased response times but then recovers quickly.

One conclusion and one conjecture are drawn from this relatively simple experiment. First the conclusion, tenant services must be rearranged as loads on those services vary significantly over time. This is clearly demonstrated by Figure 4 where t=900. Second, the
conjecture, if a configuration is found for a particular set of loads, e.g. \( t < 600 \) that performs well, caching it for later use will decrease the amount of time it takes to find a configuration (no search algorithm is needed as it can be selected from the cache) and therefore should reduce response time. This is demonstrated by the left side, \( t < 600 \), of Figure 4. The consistently higher values show that well performing configurations are not always found and switching to a new (and unproven) configuration every minute does not work as well as using one that has been proven to perform well.

8. Future Work

Our future work using the Cloud Chamber as it functions now includes investigating on-line methods of implementing configurations to improve performance and availability of services. This involves how and when configurations are found and implemented. These on-line methods will include caching previously utilized configurations as conjectured above. The Cloud Chamber is built so that different algorithms and heuristics can be employed in the calculation of configurations and in the calculation of their quality measure. Additionally, integrating a more robust gossip structure will allow for greater scalability and reduce network utilization and processor overhead. Future versions of the Cloud Chamber should become a conduit, wrapper, or module for any FastCGI server so that application engines such as Perl, PHP, and Java can be investigated.

9. Conclusion

In conclusion the Cloud Chamber meets all the requirements described in the introduction. The Cloud Chamber creates a facility to create, exercise, and examine the behavior of tenants in a Software as a Service environment. Services of various shapes and sizes can be deployed onto a heterogeneous set of nodes providing different amounts of resources. These services can be executed with any size of prescribed load for any length of time. The nodes self-organize autonomously finding and implementing tenant assignments in response to changes in the environment. The authors are unaware of any such facility. The Cloud Chamber will provide an environment for rich research in the area of the tenant placement problem and general tenant behavior.

12. References


