Economical Efficient for High Scalable Applications

Sreedevi Pogula [α] & M. Ganesh Kumar [σ]

Abstract - **Service-oriented architecture (SOA) paradigm for the purpose of large-scale applications offers meaningful cost savings by rework existing services. However, the high oddity of client appeal and the allocated character of the access may depreciate service response time and chance. Static cloning of components in database for placing load spikes need efficient resource planning and also uses the cloud infrastructure. Moreover, no service chance gives trust is provided in situations like datacenter crashes. In this paper, we explore a cost-efficient usage for dynamic and geographically-diverse cloning of elements in a cloud computing infrastructure that perfectly adapts to load differences and provides service chance guarantees. When comes to economic level, components hire server opportunities and clones or trashes themselves based to self optimizing situations. We proved in real time applications that such an access better in response time even full cloning of the components in all servers, while providing service chance guarantees under failures.**

I. Introduction

Cloud computing is deemed to replace high capital expenses for infrastructure with lower operational ones for renting cloud resources on demand by the application providers. However, with static resource allocation, a cluster system would be likely to leave 50% of the hardware resources (i.e. CPU, memory, disk) idle, thus baring unnecessary operational expenses without any profit (i.e. negative value flows). Moreover, as clouds scale up, hardware failures of any type are unavoidable.

A efficient online application is of capable to resist traffic spikes and huge crowds efficiently. And also the service offered by the application have to be volatile to different types of failures. A perfect solution opponent to load differences would be static over-provisioning of resources, at last it result into resource underutilization for most of the time. Resource redundancy should be employed to increase service reliability and chance, yet in a cost-effective way. Most importantly, as the size of the cloud increases its administrative overhead becomes unmanageable. The cloud resources for an application should be self managed and adaptive to load variations or failures. In this paper, we propose a middleware ("Scattered Autonomic Resources", referred to as Scarce) for supple sharing to avoid stranded and underutilized computational resources that dynamically adapts to changing conditions, such as failures or load variations. Our middleware simplifies the development of online appliances composed by multiple independent components (e.g. web services) following the Service Oriented Architecture (SOA) principles. We consider a virtual economy, where components are treated as individually rational entities that rent computational resources from servers, and migrate, replicate or exit according to their economic fitness. This fitness expresses the difference between the utility offered by a specific application component and the cost for retaining it in the cloud. The server rent price is an increasing function of the utilization of server resources. Moreover, components of a certain applications are dynamically replicated to geographically-diverse servers according to the chance requirements of the application. Our access combines the following unique characteristics:

- Adaptive component replication for accommodating load variations.
- Geographically-diverse placement of clone component instances.
- Cost-effective placement of service components for supple load balancing.
- Decentralized self-management of the cloud resources for the application.

Having implemented a full prototype of our access, we experimentally prove that it effectively accommodates load spikes; it provides a dynamic geographical replica placement without thrashing and cost-effectively utilizes the cloud resources. Specifically, we found that our access offers lower response time even than full replication of the service components to all servers.

II. Motivation - Running Example

Building an application that both provides robust guarantees against failures (hardware, network, etc.) and handles dynamically load spikes is a non-trivial task. As a running example, we have developed a simple web application for selling e-tickets (print@ home) composed by 4 independent components:

- A web front-end, which is the entry point of the application and serves the HTML pages to the end user.

Author α : M.Tech, CSE Dept, HITS, Hyderabad.
E-mail : sreedevigsky.net@gmail.com
Author σ : M.Tech, Asst. Prof. MREC, Hyderabad.
E-mail : ganeshkumar.programs@gmail.com

© 2013 Global Journals Inc. (US)

- A user manager for managing the profiles of the customers.
- The profiles are stored in a highly scalable, eventually consistent, allocated, structured key-value store.
- A ticket manager for managing the amount of available tickets of an event. This component uses a relational database management system (MySQL).
- An e-ticket generator that produces e-tickets in PDF format (print@home).

Each component can be regarded as a stateless, standalone and self-contained web service. Figure 1 depicts the application architecture. A token (or a session ID) is assigned to each customer's browser by the web front-end and is passed to each component along with the appeal. This token is used as a key in the key-value database to store the details of the client's shopping cart, such as the number of tickets ordered. Note that even if the application uses the concept of sessions, the components themselves is stateless (i.e. they do not need to keep an internal state between two appeals).

This application is highly sensitive to traffic spikes, when, for example, tickets for a concert of a famous band are sold. If the spike is foreseeable, one wants to be able to add spare servers that will be used transparently by the application for a short period of time, without having to reconfigure the application. After this period, the servers have to be removed transparently to the end users. As this application is business-critical, it needs to be deployed on different geographical regions, hence on different datacenters.

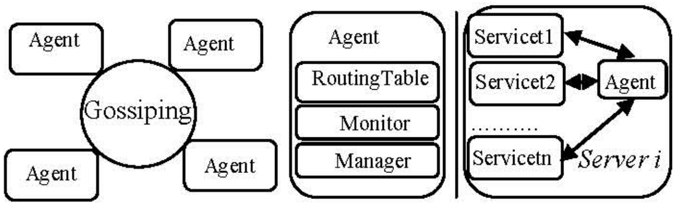
III. Scarce: The Quest of Autonomic Appliances

a) The Access

We consider appliances formed by many independent and stateless components that interact together to provide a service to the end user, as in Service Oriented Architecture (SOA). A component is self-managing, self-healing and is hosted by a server, which is in turn allowed to host many different components. A component can stop, migrate or replicate to a new server according to its load.

b) Server Agent

The server agent is a special component that resides at each server and is responsible for managing the resources of the server according to our economic-based access, as shown in Figure 2. Specifically, this agent is responsible for starting and stopping the components of the various appliances at the local server, as well as checking the "health" of the services (e.g. by verifying if the service process is still running, or by firing a test request and checking that the corresponding reply is correct). The agent knows the properties of every service that composes the application, such as the path of the service executable, its minimum and maximum replication factor. This knowledge is acquired when the agent starts, by contacting another agent (referred to as "bootstrap agent"). Any running agent participating in the application cluster can act as a bootstrap agent.



c) Routing Table

Instead of using a centralized repository for locating services, each server keeps locally a mapping between components and servers. It is maintained by a gossiping algorithm (see Figure 2), where each agent contacts a random subset (log (N) where N is the total number of servers) of remote agents and exchanges information about the services running on their respective server. Contrary to usual web services architectures, there is no central repository [such as a UDDI registry (uddi.xml.org)] for locating a service, but each agent maintains its own local registry (i.e. the routing table), as shown in Table I.

The Local Routing Table

Component	*Server*
Component1	*Server A, Server B*
Component2	*Server B, Server C*
Component3	*Server A*

A component may be hosted by several servers; therefore we consider 4 different policies that a server s may use for choosing the replica of a component:

1. A proximity-based policy: thanks to the labels attached to each server, the geographically nearest replica is chosen.
2. A rent-based policy: the least loaded server is chosen; this decision is based on the rent price of the servers.
3. A random-based policy: a random replica is chosen.
4. A net benefit-based policy: the geographically closest and least loaded replica. For every replica of the component residing at server j, we compute a weight:

© 2013 Global Journals Inc. (US)

d) Economic Model

Service replication should be highly adaptive to the processing load and to failures of any kind in order to maintain high service chance. To this end, each component is treated by the server agent as an individual optimizer that acts autonomously so as to ascertain the pre-specified chance guarantees and to balance its economic fitness. Time is assumed to be split into epochs. At every epoch, the server agent verifies from the local routing table that the minimum number of replicas for every component is satisfied; thus, no global or remote knowledge is required. If the required chance level is not satisfied and if the service is not already running locally, the agent starts the service. When the service has started, the server agent informs all others by using a hierarchical broadcast to update their respective routing tables. At each epoch, a service pays a virtual rent r to the servers where it is running. The virtual rent corresponds to the usage of the server resources, such as CPU, memory, network, disk (I/O, space).

To avoid oscillations of a replica among servers, the migration is only allowed if the following migration conditions apply:

1. The minimum chance is still satisfied using the new server,
2. The absolute price difference between the current and the new server is greater than a threshold,
3. The usages of the current server s are above a soft limit.
4. Replicate: if it has positive balance for the last f epochs, it may replicate. For replication, a component has also to verify that it can afford the replication by having a positive balance b0 for consecutive f epochs:

IV. Evaluation

a) Customer Registration

Customer has to enclose their details into the server. In this page several fields are mentioned name, e-mail id, phone number etc.., and also to provide card details are available and visa card, master card, one more advantage is expiry date and cvc no standard for card verification number in this cvc number is checking their card details.

The main advantage login as customer they can select their ticket details, but each ticket details are identified by one secrete key based on that secrete key it will process.

b) User Manager

The user manager for managing the profile of the customers, the profiles are stored in highly scalable, distributed, structured key value User manager login they will monitor how many number of users are requested the ticket. We consider many independent components that interact together to provide you service to the end user as SOA.

- ✓ In this page session is set for 60 Seconds. It beyond 60 seconds session will be expired.
- ✓ Distributed Optimization Algorithm is used. It maintains the all users' profile, also gossiping.
- ✓ The user manager Components are there: from, to, date, quota, type ticket, class, train number, etc…..
- ✓ The user manager including Sub modules: train details, passenger details, card details finally it will generate ticket format.

Designing : The four modules are using asp.net and coding c#.

Database : Sql Server.

c) Ticket Manager

The ticket manager for managing the amount of available tickets of an event. In this application to maintain how many members is ticket booking. Store user profile and all profiles are maintained using one of the controls is grid view control to visible in all booking details.

Grid view is store multiple records and retrieve through database in this control one more advantage is paging and modification, update, delete operation are applied.

d) E-Ticket Generator

An e-ticket generator that produces e-tickets in PDF format, it will generate automatically all details in report format. And how many members ticket sanctioned to main all ticket details in this module using grid view control.

Grid view is store multiple records and retrieve through database in this control one more advantage is paging and modification, update, delete operation are applied.

i. *Experimental Setup we employ two different test bed settings*

A single application setup consisting of 7 servers and a multi-application setup consisting of 15 servers. In the former setup, the cloud resources serve 1 application and in the latter one 3 appliances. We assume that the components of the application may require up to all servers in the cloud. We simulate the behavior of a typical user of the e-ticket application of Section II by performing the following actions:

1) Request the main page that contains the list of entertainment events;
2) Request the details of an event A;
3) Request the details of an event B;
4) Request again the details of the event A;
5) Login into the application and view user account;
6) Update some personal information;
7) Buy a ticket for the event A;
8) Download the corresponding ticket in PDF.

A client continuously performs this list of actions over a period of 1 minute. An epoch is set to 15 seconds and an agent sends gossip messages every 5 seconds. Moreover, the default routing policy is the random-based policy. We consider two different placements of the components:

- A static access where each component is assigned to a server by the system administrator.
- A dynamic access where all components are started on a single server and dynamically migrate/ replicate/stop according to the load or the hardware failures.

ii. *Results Dynamic vs static replica placement*

First, we employ the single-application experimental setup to compare our access with static placements of the components, where we consider two cases: i) each different component is hosted at a different dedicated server; ii) full replication, where every component is hosted at every server. The response time of the 95% percentile of the appeal is depicted in Figure 3. In the static placement (i), where a component runs on its own server, the response time is lower bounded by that of the slowest component (in our case, the service for generating PDF tickets). Thus, the response time increases exponentially when the server hosting this component is overloaded. In the case of full replication [static placement (ii)], the appeal are balanced among all servers, keeping the latency relatively low, even when the amount of concurrent users is meaningful. In the dynamic placement access, all components are hosted at a single server at startup: then, when the load increases, a busy component is allowed to replicate, and unpopular components may replicate to a less busy server. Our economic access achieves better performance than full replication, because the total amount of CPU available in the cloud is used in an adaptive manner by the components: processing intensive (or "heavy") components migrate to the least loaded servers and heavily used components are assigned more resources than others. Therefore, the cloud resources are shared according to the processing needs of components and no cloud resources are wasted by over-provisioning.

V. Future Enhancement

We will continue to research on security mechanisms that support: to maintain data securable and highly allocated data are the most important step in software development process. Before developing the tool it is necessary to determine the time factor, economy n company strength. Once these things r satisfied, ten next steps is to determine which operating system and language can be used for developing the tool. Once the programmers start building the tool the programmers need lot of external support. This support can be obtained from senior programmers, from book or from websites. Before building the system the above consideration r taken into account for developing the proposed system. Cloud computing providing unlimited infrastructure to store and execute customer data and program. As customers you do not need to own the infrastructure, they are merely accessing or renting; they can forego capital expenditure and consume resources as a service, paying instead for what they use.

a) Security a Major Concern

1. Security concerns arising because both customer data and program are residing Provider Premises.
2. Security is always a major concern in Open System Architectures.

b) Data Centre Security

1. Professional Security staff utilizing video surveillance, state of the art intrusion detection systems, and other electronic means.
2. When an employee no longer has a business need to access datacenter his privileges to access datacenter should be immediately revoked.
3. All physical and electronic access to data centers by employees should be logged and audited routinely.
4. Audit tools so that users can easily determine how their data is stored, protected, used, and verify policy enforcement.

c) Data Location

1. When user uses the cloud, user probably won't know exactly where your data is hosted, what country it will be stored in?
2. Data should be stored and processed only in specific jurisdictions as define by user.
3. Provider should also make a contractual commitment to obey local privacy requirements on behalf of their customers,
4. Data-centered policies that are generated when a user provides personal or sensitive information that travels with that information throughout its lifetime to ensure that the information is used only in accordance with the policy.

In this application simulation of e-ticket it is used to provide service to end user and data is most securable, allocated and reduce cost, large-scale allocated appliances offers meaningful cost savings by rework existing service. Over come to centralize sever and Increase response time and no chance of server hang, Network bandwidth is increases.

VI. Related Work

There is meaningful related work in the area of economic accesses for allocated computing. In [4], an access is proposed for the utilization of idle computational resources in a heterogeneous cluster. Agents assign computational tasks to servers, given the budget constrain for each task, and compete for CPU time in sealed-bid second-price auction held by the latter. In a similar setting, Popcorn access [5] employs a first-price sealed-bid auction model. Cougaar allocated multi-agent system [6] has an adaptivity engine which monitors load by employing periodic “health-check” messages. An elected agent operates as load balancer and determines the appropriate node for each agent that must be relocated based on runtime performance metrics, e.g. message traffic and memory consumption. Also, a coordinator component determines potential failure of agents and restarts them. However, cost-effectiveness among the objectives of Cougaar, and moreover our access is more lightweight in terms of communication overhead. In [7], a virtual currency (called Egg) is used for expressing a user’s willingness to pay as well as a provider’s bid for a accepting the job, and finally is given to the winning provider as compensation for job execution. Providers estimate their opportunity cost for accepting a job and regularly announce a unit price table to a central entity for a specific period. The central Egg entity informs all candidate providers about the new job and acquires responses (cost estimations). However, the access in [7] is centralized and it does not provide chance guarantees. In [8], appliances trade computing capacity in a free market, which is centrally hosted, and are then automatically activated in virtual machines on the traded nodes on-call of traffic spikes. The appliances are responsible for declaring their required number of nodes at each round based on usage statistics and allocate their statically guaranteed resources or more based on their willingness to pay and the equilibrium price; this is the highest price at which the demand saturates the cluster capacity. However, [8] does not deal with chance guarantees, as opposed to our access. Also, our access accommodates traffic spikes in a prioritized way per application without requiring the determination of the equilibrium price. Pautasso et al. ropose in [9] an autonomic controller for the JOpera allocated service composition engine over a cluster. The autonomic controller starts and stops navigation (i.e. scheduler) and dispatcher (i.e. execution and composition) threads based on several load-balancing policies that depend on the size of their respective processing queues. The autonomic component also has self-healing capabilities. However, proper thread placement in the cluster and communication overhead among threads are not considered in [9]. Also, in [10], SLA agreements for a specific QoS level for web services are established. However, monitoring of SLA compliance may require the involvement of third-parties or centralized services. A bio-networking access was proposed in [11], where services are provided by autonomous agents that implement basic biological behaviors of swarms of bees and ant colonies such as replication, migration, or death. To survive in the network environment, an agent obtains “energy” by providing a service to the users. Moreover, several implementation frameworks exist to build reliable SOA-based appliances: [12] is a mechanism for specifying fault tolerant web Service compositions, [13] is a virtual communication layer for transparent service replication, and [14] is a framework for the active replication of services across sites. These frameworks do not consider dynamic adaptation to changing conditions, such as load spikes, or do not provide guarantees about geographical diversity of replicas.

VII. Conclusion

A web front-end, which is the entry point of the application and serves the HTML pages to the end user. A user manager for managing the profiles of the customers. The profiles are stored in a highly scalable, eventually consistent, allocated, structured key-value store. A ticket manager for managing the amount of available tickets of an event. This component uses a relational database management system. An e-ticket generator that produces e-tickets in PDF format (print home). The main advantage login as customer they can select their ticket details. but each ticket details are identified by one secrete key. Based on that secrete key it will process. This project is used to manage efficiently with less cost by using secretary keys. Every transaction is identified based on this key only.

The complete communication between customer, user manager, ticket manager and e-ticket generator with centralized server. Each component can be regarded as a stateless, standalone and self-contained web service. Figure 1 depicts the application architecture. A token (or a session ID) is assigned to each customer’s browser by the web front-end and is passed to each component along with the appeal.

References Références Referencias

1. “The apache cassandra project,” http://cassandra.apache.org/.
2. L. Lamport, “The part-time parliament,” ACM Transactions on Computer Systems, vol. 16, pp. 133–169, 1998.
3. N. Bonvin, T. G. Papaioannou and K. Aberer, “Cost-efficient and differentiated data chance guarantees in data clouds,” in Proc. of the ICDE, Long Beach, CA, USA, 2010.
4. C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Stornetta, “Spawn: A allocated computational economy,” IEEE Transactions on Software Engineering, vol. 18, pp. 103–117, 1992.
5. O. Regev and N. Nisan, “The popcorn market. online markets for computational resources,” Decision Support Systems, vol. 28, no. 1-2, pp. 177 – 189, 2000.

© 2013 Global Journals Inc. (US)

6. A. Helsinger and T. Wright, “Cougaar: A robust configurable multi agent platform,” in Proc. of the IEEE Aerospace Conference, 2005.
7. J. Brunelle, P. Hurst, J. Huth, L. Kang, C. Ng, D. C. Parkes, M. Seltzer, J. Shank, and S. Youssef, “Egg: an extensible and Economics-inspired open grid computing platform,” in Proc. of the GECON, Singapore, May 2006.
8. J. Norris, K. Coleman, A. Fox, and G. Candea, “Oncall: Defeating spikes with a free-market application cluster,” in Proc. of the International Conference on Autonomic Computing, New York, NY, USA, May 2004.
9. C. Pautasso, T. Heinis, and G. Alonso, “Autonomic resource provisioning for software business processes,” Information and Software Technology, vol. 49, pp. 65–80, 2007.
10. A. Dan, D. Davis, R. Kearney, A. Keller, R. King, D. Kuebler, H. Ludwig, M. Polan, M. Spreitzer, and A. Youssef, “Web services on demand: Wsla-driven automated management,” IBM Syst. J., vol. 43, no. 1, pp. 136–158, 2004.
11. M. Wang and T. Suda, “The bio-networking architecture: a biologically inspired access to the design of scalable, adaptive, and survivable/ available network appliances,” in Proc. of the IEEE Symposium on Appliances and the Internet, 2001.
12. N. Laranjeiro and M. Vieira, “Towards fault tolerance in web services compositions,” in Proc. of the workshop on engineering fault tolerant systems, New York, NY, USA, 2007.
13. C. Engelmann, S. L. Scott, C. Leangsuksun, and X. He, “Transparent symmetric active/active replication for service level high chance,” in Proc. of the CCGrid, 2007.
14. J. Salas, F. Perez-Sorrosal, n.-M. M. Pati and R. Jim´enez- Peris, “Ws-replication: a framework for highly available web services,” in Proc. of the WWW, New York, NY, USA, 2006, pp. 357–366.
15. M. Wang and T. Suda, “The bio-networking architecture: a biologically inspired access to the design of scalable, adaptive.

© 2013 Global Journals Inc. (US)

Global Journals Inc. (US) Guidelines Handbook 2013

www.GlobalJournals.org