Tenant Level Checkpointing of Meta-Data for Multi-Tenancy SaaS

Basel Yousef, Hong Zhu and Muhammad Younas
Department of Computing and Communication Technologies
Oxford Brookes University, Oxford, OX33 1HX, UK
Email:{basel.yousef-2011, hzhu, m.younas}@brookes.ac.uk

Abstract

Traditional checkpointing techniques are facing a grave challenge when applied to multi-tenancy software-as-a-service (SaaS) systems due to the huge scale of the system state and the diversity of users' requirements on the quality of services. This paper proposes the notion of tenant level checkpointing and an algorithm that exploits Big Data techniques to checkpoint tenant's meta-data, which are widely used in configuring SaaS for tenant-specific features. The paper presents a prototype implementation of the proposed technique using NoSQL database Couchbase and reports the experiments that compare it with traditional implementation of checkpointing using file systems. Experiments show that the Big Data approach has a significantly lower latency in comparison with the traditional approach.

Keywords -- Cloud Computing; Software-as-a-Service; Multi-tenancy; Meta-data; Checkpointing; Fault Tolerance; NoSQL database; Big Data.

1 Introduction

In two recent incidents of outage, the Salesforce's multi-tenancy software-as-a-service (SaaS) system took more than 10 hours to recover [1, 2]. Salesforce, one of the world leading SaaS providers, was widely criticized for the lost of services to its tens of thousands of tenants during the outages. This flags a signal that traditional checkpointing techniques [3,4,5,6,7,8] are facing a grave challenge. SaaS does not only require checkpointing to be capable of dealing with a huge volume of data with minimal disturbances of the services to a huge number of tenants, but also capable of recovering swiftly from failures. These issues motivate the need for a new solution.

In this paper, we propose a novel checkpointing technique called a tenant level checkpointing. The basic idea is to extract and save the data that belong to a specific tenant for each invocation of the checkpointing operation. Its main advantage is that each checkpointing operation can be done at an appropriate time that has minimal effect on the whole system's performance. The whole system's state can be saved through a number of invocations of checkpointing operations, for example, in a tenant-by-tenant way. Another advantage of the proposed approach is that frequency of checkpointing can be tailored (and varied) according to the tenant's requirements on reliability and quality of services. Moreover, after a system outage, the rollback can also be preformed gradually in a tenant-by-tenant way so that the total time of system outage can be shortened and critical tenants be restarted sooner with minimal loss of services. In the case of system's partial failure, rollback can also be performed via restarting only the affected tenants.

The second basic idea of the proposed approach is to employ Big Data technology in the implementation of checkpointing operations. This further reduces the time latency and other overheads by taking the advantages of parallel processing power of cloud infrastructure and the distribution of checkpoint data over a cluster of servers.

The remainder of the paper is organized as follows. Section 2 briefly reviews the architectures of multi-tenant SaaS. Section 3 outlines our proposal. Section 4 reports the implementation of a prototype and the results of some preliminary experiments. Section 5 concludes the paper with a comparison with related work and a discussion of future work.

2 SaaS Architectures

Multiple tenant SaaS can generally be provided in one of two types of architectures [9]: *multi-instance architecture* and *single-instance architecture*. The former employs virtual machine to install multiple instances of the software so that each tenant is served by one instance. In contrast, the later only runs one instance of the software on the servers and deliver the services to all tenants. Thus, it is called *multi-tenancy architecture*.

It is widely accepted in industry that the multi-tenancy architecture has advantages in terms of cost reduction, scalability, maintainability, and usability over multi-instance approach [10]. A number of variants of multi-tenancy SaaS architectures have been developed; see [9] for a survey.

In general, as shown in Figure 1, the multi-tenancy architecture consists of two data storage/processing

components [11]: one uses traditional relational database for structured data, and the other uses NoSQL database for semi-structured or unstructured data. There is often an ETL engine that extracts, transforms and loads data between them [12].

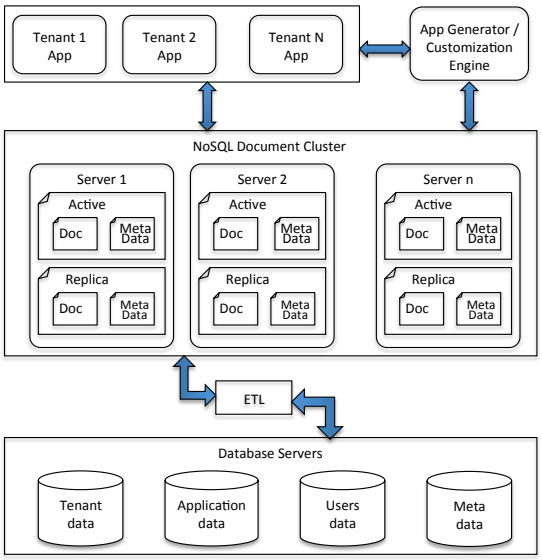

Figure 1. A General Model of SaaS Architectures

A common characteristic of multi-tenancy architectures is that tenant-specific features and customer configurations of the software are specified via meta-data, which are interpreted by a runtime engine [13, 14, 15]. Meta-data are mostly unstructured, thus, usually stored in a NoSQL database. They have the following features.

- *Large volume*. For example, it is estimated that there are tens of GBs in one of Google file system clusters [18]. Salesforce uses terabytes of storage space to store its metadata in thousands of caching servers distributed around the world [16, 17].
- *Small proportion.* Meta-data is usually only a small proportion of all data on a cloud. For example, meta-data occupy only about 1% of all data in Google file system [18].
- *High demand.* Meta-data is usually highly accessed by the users. For example, an analysis of Unix file system has reported that 50% - 80% of all file system accesses are to meta-data [19]. Therefore, in practice, meta-data are often cached in memory to deliver sub-millisecond random reads, with high-throughput writes.
- *Separable ownership*. Meta-data mostly belong to different tenants. The existing cloud systems and SaaS applications, such as Salesforce, support the identification of the ownership for each piece of meta-data for security reasons [17].

Meta-data plays a crucial role in the operation of SaaS applications. It is desirable to ensure the integrity of meta-data by employing a reliable and efficient fault tolerant technique. Currently, cloud service providers do take meta-data fault tolerance into serious consideration. However, the current practice is unsatisfactory.

3 The Proposed Approach

3.1 Architecture

Figure 2 shows the architecture of the proposed approach. It extends the generalized SaaS architecture shown in Figure 1 and is comprised of the following components.

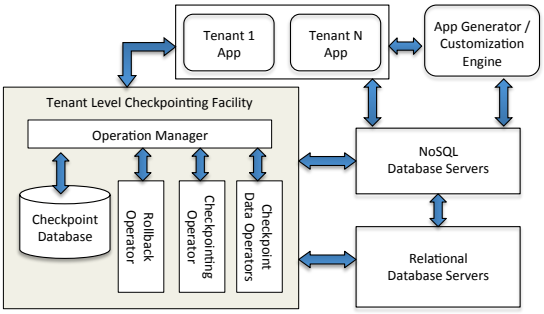

Figure 2. Multi-Tenancy Checkpointing System Structure

- *Operation Manager.* The manager is responsible for the invocation of checkpointing and recovery operations. It manages the partial state checkpoint data and communicates with the SaaS application to obtain system operation state; e.g., whether a tenant is active or inactive, whether the system is busy or lightly loaded, etc. It uses such information to decide when to checkpoint a particular tenant according to a timing strategy and a tenant selection policy. It then invokes the checkpointing operator to create checkpoints and to store them in the system. It also updates the records of Checkpoint Database. It will also decide when to remove a checkpoint from the system according to the duplication strategy. When required, it invokes the rollback operator and uses the checkpoint data to rollback the system's state to the previous state saved in the checkpoints. It will use the Checkpoint Database to retrieve the information about checkpoint data and to decide on the sequence of rollback operations using a Rollback strategy.
- *Checkpointing Operator.* It creates various types of partial state checkpoints. The input parameters of the operator include:

(a) the owner of the checkpoint data, such as the tenant ID or the system;
(b) the type of the checkpoint data, such as structured data, unstructured data, meta-data, etc.

- *Rollback Operator.* It recovers the system by reloading a set of checkpoint data back to the system.

• *Checkpoint Data Operator*. It provides a set of operations on checkpoint data, such as:

(a) deleting a checkpoint data from the system when it is out of date because new checkpoint data becomes available;
(b) relocating a checkpointing data from one place to another.

• *Checkpointing Database*. It is a database that stores information about checkpoint data. Typically, it keeps the following information about checkpoints:

(a) the timestamp of the creation of the checkpoint;
(b) the type of checkpoint, such as meta-data checkpoint, or user data checkpoint, or system data checkpoint, etc.;
(c) the owner of the checkpoint, i.e. the tenant ID;
(d) the location where the checkpoint data is stored;
(e) the size of the checkpoint data,
(f) the health state of the checkpoint data, etc.

In the next subsections, we give further details of the checkpointing operator. Due to the space limitation, details of other components are omitted.

3.2 The Checkpointing Algorithm

The checkpointing process consists of two stages.

• *Collecting Data*

During this stage, a collection of relevant data that represents a partial view of the system/application's state is collected from the cloud cluster. Such a partial view is characterized by a set of data selection criteria, which include the identifiers of checkpointed tenant(s), the type of data (such as meta-data), the time interval(s), etc.

• *Saving Data*

In this stage, the checkpoint data retrieved in the collecting stage are divided into a number of units and then stored. There are a number of different ways that these checkpoint data can be stored. In particular, the following two alternative ways are applicable.

– A checkpoint can be stored in a specific file system on a dedicated server in the same way as in the traditional disk-based checkpointing.
– A checkpoint can be saved back to the cloud cluster, but on different machines. This is similar to the diskless checkpointing techniques.

In both cases, the location information of the saved data will be stored together with the view description and maintained by the manager. It is used in the rollback operation.

Figure 3 describes the main steps of the algorithm. CPID is the checkpoint identifier, which consists of the tenant ID and the timestamp of the checkpointing operation to provide a unique ID for the checkpoint data. The *Map* function is executed on all servers to collect the kind of checkpoint data for the tenant whose identifier is TenantID. Here, the kind can be either meta-data or normal data. The collected checkpoint data CPData are in Result, which is split into a number of blocks. The block size is predetermined. Each block has a unique sequence number SeqNum so that it can be reassembled when needed for rollback. Each block is saved through the *Save* function either to a file on a dedicated server or distributed back to the cloud through a NoSQL database. The *Save* function also inserts to the checkpoint database CPDB a record that consists of the checkpoint identifier CPID, the block sequence number SeqNum and the location where the block is saved in. These records are used when checkpoint data is used in rollback or deleted when it becomes outdated. After the checkpointing operation is completed, a record of this operation is also added into the checkpoint database CPDB by invoking the *InsertCPDB* function.

```
Algorithm Checkpointing
Input TenantID: Int;
      Kind: {Meta-Data, Data};
Begin
   Setup connection;
   CPID = <TimeStamp, TenantID>;
   Result = Map(TenantID, Kind);
   SeqNum = 0; j = 0; CPData = nil;
   For i=0 to Length(Result)-1 do
      {  CPData = CPData+Result[i];
         j++;
         If j == BlockSize then
            {Save(CPID, SeqNum, CPData);
             CPData=nil; j=0; SeqNum++;
      };
   If CPData <> nil then
      {Save(CPID, SeqNum, CPData); SeqNum++};
   Close connection;
   InsertCPDB(CPID, SeqNum);
End algorithm
```

Figure 3. Checkpointing algorithm

3.3 Prototype Implementation

We have implemented the checkpointing algorithm in a prototype system called *Tench*, which stands for *Ten*ant level *Ch*eckpointing. It is developed using Eclipse 3.7.2 with Couchbase Java SDK library.

Tench checkpoints tenant's meta-data stored in the NoSQL database Couchbase Server 2.1.1[(1)]. It is configured to offer a caching layer to store the documents contents as keys and values by using Membase as a cache storage layer. Queries in Couchbase are stored in views that can be called to get the required results. While Couchbase can be configured to store a number of replicas for every bucket, here we choose to have only one replica per bucket in the experiment.

The prototype system is deployed and executed on a small cluster of PC computers running on Linux Ubuntu 12.4 that consists of 8 machines connected via

[(1)] http://www.couchbase.com/

Ethernet. Two of the compute nodes have Intel i5 2.67GHz processors, and the other ones have Intel Core 2 Duo 3.0GHz processors. All of the computers have 4GB memory.

4 Experiments

We have conducted preliminary experiments with the prototype system *Tench*. Given the importance of meta-data in multi-tenancy SaaS applications, the experiments focused on checkpointing meta-data. This section reports the results of the experiments.

4.1 Design of the Experiments

The experiments are designed to demonstrate the feasibility of tenant level checkpointing and to find out whether using NoSQL database for saving checkpoint data is more efficient than traditional disk-based checkpointing (with dedicated checkpoint file server). We conducted experiments on the following research questions.

1) How does checkpoint latency vary with the size of checkpoint data?
2) How does the number of tenants affect checkpoint latency?

Here, the latency of a checkpointing operation is the total time elapsed from the start of collecting checkpoint data to the finish of saving the data.

To answer the above research questions, the following two experiments were conducted.

– *Experiment 1*. In this experiment, the checkpointing operation was applied to system states where the size of the checkpoint data of a tenant varies from 10,000 records to 1,000,000 records, where each record is a JSON document. Each step increases the size by 10,000 records. Each record is of 128 bytes.
– *Experiment 2*. In this experiment, the size of data to be checkpointed is fixed, but the number of tenants in the system varies.

For each of these two experiments, a NoSQL database (Couchbase) is populated with random data of JSON documents of the following class.

```
Class Tenant {
   int Tenant_id;  -- Tenant Identifier.
   String DataFormat;
   String DateFormat;
   String phone_number;}
```

The following is a typical example of randomly generated instance of the class *Tenant*.

```
{"Tenant_id": 0,
 "DataFormat": "lTTCUBXFDz0tG23MYI53",
 "DateFormat": "0nupP5ivm6kDECfU4Eos",
 "phone_number": "+49-85-176-771-15"}
```

Collecting checkpoint data for a tenant is performed by applying a map function to all records in the database. A map function is called a "*view*" in Couchbase's terminology. The following is the view for checkpointing the meta-data for the tenant whose Tenant_id is *x*.

```
View_Tenant_id (x) {
   if(doc.Tenant_id == x) {emit(doc)}
}
```

where the meat-data are instances of the class *Tenant* and they are contained in *doc*.

Note that, for different parameters of the checkpointing operator, different views are used to collect the data or meta-data.

4.2 Experimental Results

- *Latency of different checkpoint sizes*

As shown in Figure 4, for disk-based approach, the checkpointing latency increases linearly with the increase of the size of checkpoint data.

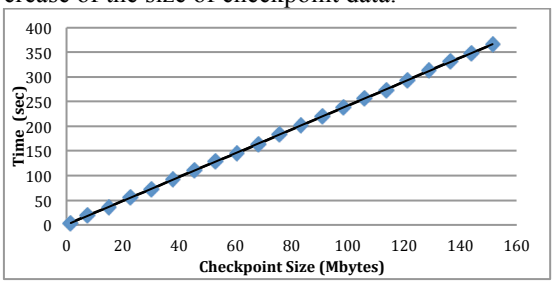

Figure 4. Latency of Disk-Based Checkpointing

A similar observation is made in the experiments with NoSQL database based checkpointing (see Figure 5). In both approaches, the latency increases in a linear way with the increase in number of records.

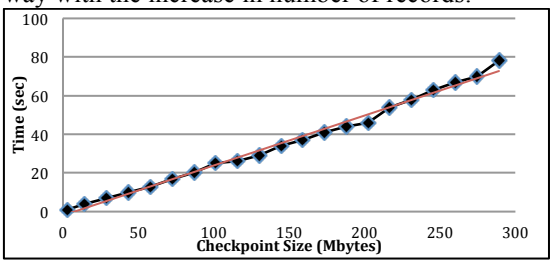

Figure 5. Latency of NoSQL Based Checkpointing

However, the latency of saving checkpoint data to the Couchbase NoSQL database is much smaller than saving checkpoint data to a file as shown in Figure 8. The main reason for this is that NoSQL databases (such as Couchbase), use a multiple-level memory cache system to store data, as shown in Figure 6. In this structure, most recently accessed or created data are saved in memory space. It is then written to hard disks. Therefore, checkpoint data is first saved into memories of the computers in the cluster, thus the checkpointing operation finishes much faster than directly saving them to a file. In such a situation, using NoSQL database to save checkpoint data is similar to diskless checkpointing. Thus, we say that NoSQL database based approach is *diskless-like*.

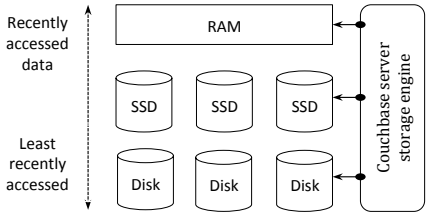

Figure 6. Multiple level cache structure of NoSQL databases

The above explanation is supported by the experiment data. Figure 7 and Figure 8 show the time spent on collecting data and saving data in each checkpointing for two approaches, respectively. It shows that saving checkpointing data to file takes much longer than saving to Couchbase.

- *Latency of different numbers of tenants*

Since a SaaS application may have a large number of tenants, it is important to understand how the scale of SaaS in terms of number of tenants will affect the latency of checkpointing.

The experiment data shows that the number of tenants has a minimal effect on latency for both disk-based and diskless-like approaches.

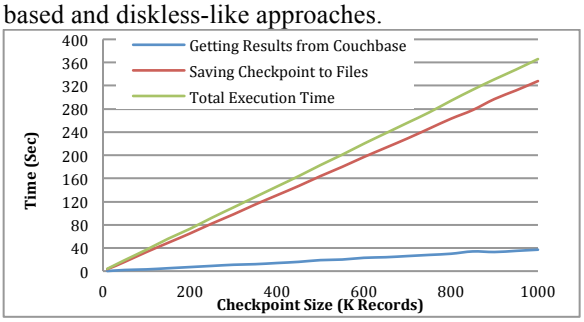

Figure 7 Time Overhead of Disk-Based Checkpointing

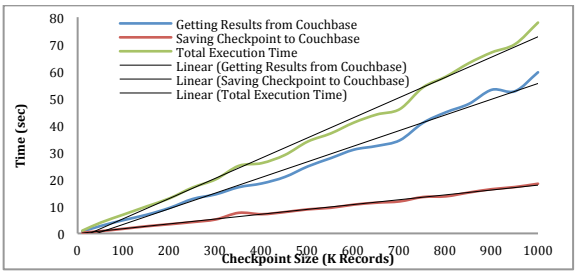

Figure 8. Time Overhead of NoSQL Checkpointing

As shown in Figure 9 andFigure 10, the latency of checkpointing varies little with the increase of the number of tenants in the system if the total amount of data in the system and the amount of data to be checkpointed remain the same.

However, in NoSQL database based checkpointing, the variation of latency is much larger than file-based checkpointing. This is again caused by the multiple-level cache structure of NoSQL databases.

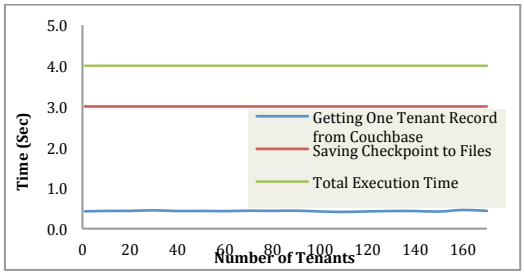

Figure 9. Latency of Disk-Based Checkpointing

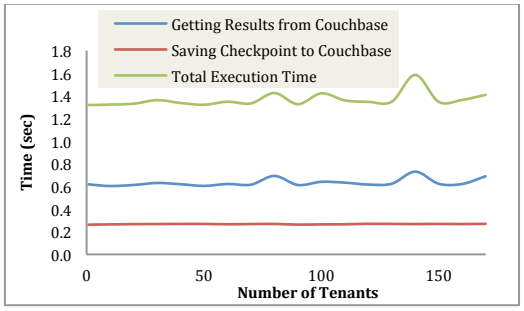

Figure 10. Latency of NoSQL Checkpointing

5 Conclusion

This paper proposes the notion of tenant level checkpointing for SaaS applications. It differs from existing levels of checkpointing operations, i.e., at system level and at application level. Because each checkpoint data is only a partial state of the SaaS application, it promotes a new type of checkpointing techniques, i.e. partial state checkpointing, in contrast to full state checkpoint. The traditional incremental checkpointing techniques can then be considered as a special type of partial state checkpointing.

Tenant level checkpointing has a number of advantages. First, while a SaaS application is run continuously, tenant level checkpointing can be performed targeting a specific tenant when the users of the tenant are less active. Thus, checkpoint may cause less disruption (or blocking) to the normal operation of the system.

Second, tenants with different requirements of the quality of service (e.g., varying level of reliability), can be treated accordingly by changing the frequency of tenant level checkpointing.

Moreover, cloud applications are often of huge scale. It is unrealistic to save the state of the whole system within one checkpointing operation. Tenant level checkpointing decomposes the checkpointing operation and the checkpoint data into a number of partial states of much smaller scale. It makes checkpointing more manageable and practical.

In the case of system crash, recovery can be performed through tenant-by-tenant rollback so that the most important tenants are recovered first. The total

outage time of the system can be significantly shortened.

Finally, through partial checkpointing, different types of data can be treated differently. The more important the data is, the more frequent it can be checkpointed. It is particularly useful for checkpointing the meta-data, which plays a crucial role in SaaS applications in the multi-tenancy architecture.

In this paper, we also proposed an architecture for managing the operations and data of partial checkpointing and rollback in the context of multi-tenancy SaaS architecture. A checkpointing algorithm and a prototype implementation are presented. They use a NoSQL database to collect and save checkpoint data. Experiments are conducted to compare this approach with the traditional approach that saves checkpoint data in a file system on a dedicated checkpointing server. Experimental results show that for both approaches, the latency increases linearly with the size of checkpoint data, while the number of tenants in the system has little impact on latency. However, the NoSQL database approach was proved to have a noticeably lower latency in comparison to the traditional disk-based approach. It is due to the multiple level cache storage structure used by NoSQL databases. Thus, the NoSQL database approach demonstrates a diskless-like dynamic behavior and performance. Moreover, NoSQL databases like Couchbase have utilized parallel processing power of cloud cluster by employing MapReduce in processing database queries and updates. They are capable of dealing with large-scale checkpoint data, thus more suitable for cloud applications, especially for checkpointing meta-data.

We are further improving the performance of the checkpointing operations by (1) combining partial checkpointing with other techniques, such as compression and incremental checkpointing; (2) utilizing the parallel processing power of MapReduce to collect and save checkpoint data.

We are also investigating the policies and techniques for deciding when to checkpoint the data and meta-data of a tenant with minimal disruption to system performance. Another future work is to study the policy and technique for efficient rollback so that the system can be recovered from failure rapidly.

References

[1] C. Kanaracus. "*Salesforce.com hit with second major outage in two weeks*". URL: http://www.infoworld.com/d/cloud-computing/salesforcecom-hit-second-major-outage-in-two-weeks-197383. Jul., 2012. Last Access: 14 Dec., 2013.

[2] R. Miller. "*Major outage for salesforce.com*". URL: http://www.datacenterknowledge.com/archives/2012/07/10/major-outage-salesforce-com/. Jul., 2012. Last Access: 14 Dec., 2013.

[3] A. Agbaria and R. Friedman, "Virtual-machine-based heterogeneous checkpointing." *Softw. Pract. Exper.*, Vol. 32, No. 12, pp. 1175–1192, Oct., 2002.

[4] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance." *ACM Trans. Comput. Syst.*, Vol. 14, No. 1, pp. 80–107, Feb., 1996.

[5] C. Chen, y. Ting, and J. Hen, "Low overhead incremental checkpointing and rollback recovery scheme on windows operating system," in *Proc. of WKDD'10*, IEEE CS, Jan., 2010, pp. 268–271.

[6] B. Nicolae and F. Cappello, "Blobcr: efficient checkpoint-restart for HPC applications on IAAS clouds using virtual disk image snapshots," in *Proc. of SC'11,* ACM, Nov., 2011, pp. 34:1–34:12.

[7] J. S. Plank, K. Li, and M. M. Puening, "Diskless checkpointing." *IEEE Trans. Parallel Distrib. Syst.*, Vol. 9, No. 10, pp. 972–986, Oct., 1998.

[8] Z. Chen, et al., "Fault tolerant high performance computing by a coding approach," in *Proc. of PPoPP'05*. ACM, Jun., 2005, pp. 213–223.

[9] W.T. Tsai, X. Bai and Y. Huang, "Software-as-a-Service (SaaS): Perspectives and Challenges", *Science In China*, Vol. 53, No. 1, pp1–18. May, 2012.

[10] H. Koziolek, "The sposad architectural style for multi-tenant software applications," in *Proc. of WICSA'11*. Jun., 2011, pp. 320–327.

[11] C. Weissman and S. Bobrowski, "The design of the force.com multitenant internet application development platform," in *Proc. of SIGMOD'09*. ACM, Jun., 2009, pp. 889–896.

[12] D. Abadi, et al., "Aurora: a new model and architecture for data stream management." *The VLDB Journal*, Vol. 12, No.2, pp. 120–139, Aug., 2003.

[13] T. Kwok, T. Nguyen, and L. Lam, "A software as a service with multi-tenancy support for an electronic contract management application," in *Proc. of SCC'08*. IEEE CS, Jul., 2008. pp. 179–186.

[14] C. J. Gue, et al., "A framework for native multi-tenancy application development and management," in *Proc. of CEC/EEE'07*, Jul., 2007, pp. 551– 558.

[15] C. Bezemer, et al., "Enabling multi-tenancy:an industrial experience report," in *Proc. of ICSM'10*, Sept., 2010, pp1–8.

[16] DeveloperForce, "Force.com sites best practices". URL: http://wiki.developerforce.com/page/Force.com_Sites_Best_Practices. Last Access: 14 Dec. 2013.

[17] R. Woollen, "The internal design of salesforce.com's multi-tenant architecture," in *Proc. of SOCC'10*. ACM, Jun., 2010, pp. 161–161.

[18] S. Ghemawat, H. Gobioff, and S. Leung, "The Google file system", in *Proc. of SOSP'03*.ACM,Oct.,2003, pp.29–43.

[19] J. K. Ousterhout, et al., "A trace-driven analysis of the Unix 4.2 BSD file system," in *Proc. of SOSP'85*. ACM, Dec., 1985, pp. 15–24.