Abstract

New versions of existing large-scale web services such as Passport.com have to go through rigorous performance evaluations in order to ensure a high degree of availability. Performance testing (such as benchmarking, scalability, and capacity tests) of large-scale stateful systems in managed test environments has many different challenges, mainly related to the reproducibility of production conditions in live data centers. One of these challenges is creating a dataset in a test environment that mimics the actual dataset in production. Other challenges involve the characterization of load patterns in production based on log analysis and proper load simulation via re-utilization of data from the existing dataset. The intent of this paper is to describe practical approaches to address some of the aforementioned challenges through the use of various novel techniques. For example, this paper discusses data sanitization, which is the alteration of large datasets in a controlled manner to obfuscate sensitive information, preserving data integrity, relationships, and data equivalence classes. This paper also provides techniques for load pattern characterization via the application of Markov Chains to custom and generic logs, as well as general guidelines for the development of cache-based load simulation tools tailored for the performance evaluation of stateful systems.

1. Introduction

Large-scale online web services are subject to very different loads and conditions when they are released on the internet. Services such as Passport.com can receive up to 300,000 user-driven transactions per second, and may contain a dataset of over 500,000,000 users. For this reason, new versions of such services have to undergo rigorous performance testing before going public. Irrespective of the tests being executed (benchmarking, load, scalability, capacity, etc), the high-level process consists of: environment (clusters, data, tools) preparation, execution, and analysis. This paper describes practical approaches for creating accurate environments for the execution of performance tests for stateful web services.

Web services are considered stateful if they contain hard-state instead of soft-state data [3]. Hard-state data is data that cannot be lost due to the unfeasibility of reconstructing it. An example is a user profile and user transactions for a bank account. Soft-state data can be reconstructed from hard-state data. An example would be aggregated financial reports. Many web services available today are stateful.

In performance test environments, pre-population of test data is a crucial step towards replicating production conditions. Traditional approaches for test data generation consist of synthesizing the data based on the application code, random and probabilistic techniques, or custom applications [4]. Other approaches make use of limited data sanitization processes based on predetermined heuristics [5]. However it is impractical to synthesize the same hard-states observed in production environments due to the unpredictability of the many ways in which the data may have been transformed based on users’ activities. Performance tests are particularly sensitive to the dataset since slight differences in the test data may result in significant discrepancies in the test results.

The ideal dataset would be constructed from the same set of production data for the performance tests, but many of the items in production have restricted access. Therefore there is a need for a sustainable process of obfuscating restricted data items so that the dataset can be safely used in test laboratories. This is accomplished by the use of the Data Sanitization process [6], which aims to retrieve a set of databases and deterministically obfuscate restricted data, while preserving data integrity, relationships, and data equivalence classes. This process is described in section 2.

Once the right data is in place, there is still a need to simulate production behavior by duplicating the right mix of various types of transactions processed by the system and simulating the dependency between related sequences of transactions as observed in production. Many live environments contain a set of logs which can be mined to provide up-to-date statistics describing
the transaction mix. By using data mining techniques, one can determine not only how APIs (Application Programming Interfaces) are being invoked, but also the relationship between invocations of the different APIs. We discuss one of these techniques based on an application of Markov Chains to generic/custom log data. This process is fully described in section 3.

Section 4 discusses approaches for writing cache-based performance test tools which can leverage the previously retrieved and sanitized production data, imported into test environments. Finally, we discuss practical results from the use of these techniques, as well as future improvements in section 5.

2. Data Sanitization

Using production data for testing new features which will operate on existing data (i.e., new search functionality that allows more fine-grained search criteria) is crucial not only for functional validation of correctness, but also for performance evaluation, since different amounts of data, dependencies, and data characteristics may have significant affects in the overall system’s performance. Production data, however, contains large amounts of information regarding individuals which should be kept confidential, even if only used in restricted test laboratories. The general term to classify such confidential data is called Personally Identifiable Information (PII) [1]. The data sanitization process consists of a set of tools and methodologies to take production data and obfuscate all the pre-determined PII, preserving three key characteristics:

i. Data Integrity: Constraints applied to relational database tables, such as Primary Keys and Uniqueness are carried over after sanitization.

ii. Data Relationships: Relationships between tables in a relational database persist after the sanitization process.

iii. Data Equivalence Classes: Subsets of the domain input data are preserved, such that all elements in the subsets are assumed to be the same from the specification of the subsets [2].

The current process is tailored for obfuscation of data stored in relational databases. The process consists of a sequence of steps listed as:

2.1 PII Identification: the first step consists of identifying the data that needs to be obfuscated. Databases for large-scale systems may contain thousands of different tables and columns. A set of tools is provided along with the data sanitizer framework to assist the user with the identification of PII, although the identification process requires manual intervention and cannot be fully automated due to the subjectivity of the data. The database schemas are modeled as XML files [7] containing all of the metadata pertinent to the tables in all of the databases.

2.2 Sanitization Method Assignment: The method used to sanitize a certain PII field can now be chosen in such a way to preserve (i), (ii) and (iii). Figure 1 below illustrates the PII identification and the sanitization method assignment:

A sanitization method may have the following generic signature:

\[
\text{object SanitizationMethod( object OriginalValue, object[] Metadata)}
\]

The metadata may consist of the details of the particular field in question, such as data type and length. The method should perform a one-way transformation in order to avoid reverse engineering of the sanitized data. The data sanitizer framework comes with several sanitization methods, including the following ones (table 1).

<table>
<thead>
<tr>
<th>Standard Sanitization Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Erase</td>
<td>Erases any non-binary field</td>
</tr>
<tr>
<td>EraseBinary</td>
<td>Erases binary fields</td>
</tr>
<tr>
<td>FillWithChar</td>
<td>Replaces the entire field with a random string of same length</td>
</tr>
<tr>
<td>FillWithDigit</td>
<td>Replaces the entire field with a random number of same length</td>
</tr>
<tr>
<td>HashString</td>
<td>Applies a one-way SHA1 salt-based (password) hash function</td>
</tr>
<tr>
<td>HashDigits</td>
<td>Applies a one-way SHA1 salt-based (password) hash function, but the result is numeric</td>
</tr>
<tr>
<td>NewGUID</td>
<td>Replace a GUID with a different (new) GUID</td>
</tr>
</tbody>
</table>

Table 1. Standard sanitization methods

New methods can be added to the framework when deemed necessary. The use of one-way SHA1 hash functions as a sanitization method is essential to ensure
data integrity and relationship preservation post-sanitization: correlated data across tables/DBs can be sanitized by using one-way, salt-based SHA1 hash, ensuring the same output, thus consistency.

A crucial step is to review PII sets as well as sanitization methods and assignments with security experts and legal personnel to validate the correctness of the procedures.

2.3 Test and Sanitization Execution: after the proper identification of PII and sanitization methods, the overall sanitization process is tested in restricted laboratories on non-production data. Upon verification, the process is carried out in production environments. Since the sanitization is an in-place procedure, a copy of production data is made, all within secure production environments. The main sanitizer tool is multithreaded for optimal speed, thus multiple databases and tables are processed concurrently. Because of this, database constraints must be removed prior to the sanitization execution, since during the execution phase data relationships might be temporarily violated. All these constraints are saved prior to the sanitization, and are recreated upon completion of the sanitization process (figure 2).

Table 2 below shows some results obtained running the data sanitizer on HP DL385 G1, 4xAMD 2.4GHz processors, 4GB RAM servers, against large data sets. The average percentage of PII fields identified in these data sets was ~13%, with one-way hash functions accounting for ~10% of the sanitization methods:

<table>
<thead>
<tr>
<th>Databases</th>
<th>Data</th>
<th>Size</th>
<th>Time (h)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subscription Service</td>
<td>33MM users</td>
<td>850GB</td>
<td>22</td>
</tr>
<tr>
<td>Partner Reporting Service</td>
<td>30MM users</td>
<td>600GB</td>
<td>16</td>
</tr>
<tr>
<td>Financial Reporting Service</td>
<td>32MM users</td>
<td>800GB</td>
<td>21</td>
</tr>
<tr>
<td>Customer Assistance Service</td>
<td>1.4MM tickets</td>
<td>50GB</td>
<td>3.5</td>
</tr>
<tr>
<td>Authentication Service</td>
<td>400MM users</td>
<td>3.5TB</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 2. Sanitization runs experiments

3. Markov Chain Stress Model

The Markov Chain Stress Model includes two major components: the knowledge retriever component and the knowledge exerciser component. Both components are based on the concept of the Markov Chain dynamic stochastic process, which describes the state of systems at successive times [8]. The knowledge retriever component applies data mining on production activity logs and discovers the parameter load patterns of each API. The knowledge exerciser component uses Markov Chain Monte Carlo methods [9], which are a class of algorithms for sampling from probability distributions, based on constructing a Markov chain with the desired distribution as its stationary distribution, to manifest the former statistical knowledge to the stress test environment, and generates dynamic scenarios consistent with production load patterns.

During the knowledge retriever step, we assume that:

- A distributed application has a set with countable number of APIs, represented as:
  \[ X = \{X_0, X_1, \ldots, X_n\} \]
- Any API \( X_j \) is logically connected to API \( X_i \) with a probability weight, where \( j = i \) is possible.
- Each API \( X_i \) has a known set of domain data as input parameters:
  \[ X_i = f(p_0, p_1, \ldots, p_m) \]
  Where \( p_j \) belongs to a domain set \( S_j \), \( p_j \in S_j, j = 0, 1, \ldots, m_i \)
  where \( S_j \) is the set of all possible values of \( p_j \) for this API \( X_i \)
- A client application makes API calls against the web service according to the Markov process. To simplify our model, we use the first-order Markov process [9] to implement the program, i.e. current API call \( X_t \) at time \( t \) depends only on the previous API call \( X_{t-1} \):
  \[ P(X_{i,t} | X_{t-1}) = P(X_{i,t} | X_{i,t-1}) \]
- Client is homogenous over time, meaning that its behavior is consistent, the transition matrix can be re-built any time.

With the above assumptions, we use real production trace data as samples to estimate the Markov Transition Matrix.
\[ X_0 \quad X_1 \quad X_2 \ldots \quad X_n \]

\[ X_i \quad p_{i0} \quad p_{i1} \quad p_{i2} \ldots \quad p_{in} \]

\[ X_{i+1} \quad p_{i+11} \quad p_{i+12} \quad p_{i+13} \ldots \quad p_{i+1n} \]

\[ X_n \quad p_{n0} \quad p_{n1} \quad p_{n2} \ldots \quad p_{nn} \]

where:

- \( p_{ij} \) represents the transition probability
- And we have:
  For \( \forall i, j \mid i = 0,1,\ldots,n; \ j = 0,1,\ldots,n \)
    - \( 0 \leq p_{ij} \leq 1 \)
    - \( \sum_{j=0}^{n} p_{ij} = 1 \)

During the knowledge exerciser step our objective is to produce load patterns that are as close as possible to those in production, using the Markov Transition Matrix we created during the Knowledge Retriever step.

An important aspect of this methodology is that the transition matrix extracted only reflects the average behavior over a certain period of time. Therefore, in order to reproduce the exact same pattern observed in production (in terms of different variables, such as CPU utilization, memory utilization, disk utilization, etc.), the matrix has to be periodically updated.

The following diagram (figure 3) describes the entire workflow:

![Markov Chain workflow diagram](image)

In the above workflow, we first aggregate the production activity logs by session, and order them by timestamps. Through data mining over the aggregated activity logs, we can get: (1) the Markov Transaction Matrix over the APIs, and (2) the interval between each of the APIs transacting (some authors refer to it as “Thinking Time” [10]). Through data mining over each API’s parameters with aggregated statistics, we can get the parameter calling patterns of each API. At this point, we are done with the knowledge retriever step.

Our next step is to exercise the knowledge collected in previous steps in the stress testing environment. In practice, we found that it is more challenging to recover and mimic the parameter patterns of each API call than to produce the Markov transition matrix. This occurs due to the fact that the majority of input parameters are user-specific information with a high degree of randomness (i.e., user’s first name). For user-specific information, we created tools or methods that would generate valid parameter values by creating them or retrieving them from our data store, and then deposit them in parameter pools.

We then use a thread to represent a client. This thread will call APIs one by one based on the Markov Transaction Matrix. We also introduce a sleep interval between two contingent API calls to make the simulation more realistic. The model will fetch needed parameters from the parameter pool for each API call.

Our model is scalable such that generation of stress load is possible. The parameter generator and threads can be distributed across machines via configurable parameters.

Figure 4 is an example of our stress load simulation during a 3 hour run. In this example, we generated load on the primary database servers in a test environment using knowledge learned from SQL profiler traces on database servers with the same role in production environments. The load profile in our test clusters was very similar to that observed in production.

![CPU utilization in Production and Test Environment simulation](image)

The Markov chain model not only simulates production-like stress load on test environments, but also provides important insights about system behavior, especially the correlation among APIs and among the parameters of each API. For example, highly correlated APIs might get a performance boost by improving locality, while unexpected correlations
might be an indication of a potential area in need of further inspection.

4. Cache-based Load Simulation Tools

Methods described in previous sections provide a large dataset resembling production as well as a practical way to analyze and simulate real-user behaviors (i.e. scenarios). The performance tuning of a stateful system, however, often requires a particular API to be executed numerous times in isolation in order to determine the bottlenecks of the system. Web services are generally modeled after finite state machines in that the majority of APIs expect the invoking entity to be in a certain state prior to an operation [11]. The conventional test approach is to generate an entity and induce it into the required state before the actual API invocation (sometimes the state-inducement makes use of other APIs). This on-the-fly data generation and entity state inducement can hide actual system bottlenecks, since the preparation work may be the bottleneck itself. In addition, the conventional approach does not allow for taking advantage of sanitized production data.

Our solution is cache-based load simulation. This process requires pre-determination of the individual API calls to the state(s) in which the entity it operates against must be. Each defined state can be represented as a bucket. The bucket definitions are collections of Boolean conditions. When a particular entity matches the pre-defined set of conditions, the entity is said to belong to the corresponding bucket. The states are not mutually exclusive, so a given entity can potentially satisfy more than one state condition and therefore be present in more than one bucket in the cache.

Once the API state mapping and the evaluation criteria for each state have been established, the process becomes trivial. Given any API, we execute the following:

1. Determine the matching bucket for the API
2. Extract an entity from the bucket
3. Invoke the API with the selected entity
4. Re-evaluate state post API invocation
5. Re-insert the entity in the new bucket(s). If entity is no longer usable, it is discarded

Our implementation of the entity cache uses a SQL database to prevent data loss in the event of a crash, as well as allowing easy data sharing across multiple instances of the tool.

To simplify access to the database, a layer of abstraction was introduced to wrap the database calls. This layer exposes methods to modify the entities as well as the buckets. This layer exposes other methods in addition to the normal “add”, “get”, and “remove” calls (table 3):

<table>
<thead>
<tr>
<th>Method Type</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>InitializeCache</td>
</tr>
<tr>
<td></td>
<td>LoadBuckets</td>
</tr>
<tr>
<td></td>
<td>Clear</td>
</tr>
<tr>
<td>Bucket Access</td>
<td>AddBucket</td>
</tr>
<tr>
<td></td>
<td>GetBucketList</td>
</tr>
<tr>
<td></td>
<td>GetCountPerBucket</td>
</tr>
<tr>
<td>Entity Access</td>
<td>AddEntity</td>
</tr>
<tr>
<td></td>
<td>GetEntityFromBucket</td>
</tr>
<tr>
<td></td>
<td>RemoveEntity</td>
</tr>
<tr>
<td></td>
<td>GetPreExistingEntity</td>
</tr>
</tbody>
</table>

Table 3. DB access methods for cache-based simulation

Since the operations performed on the entities will most likely change the entity state, each entity is similar to a critical section – only one thread may act on an entity at a time (however, multiple threads can still operate on different entities). To avoid corruption of entity state, we remove the entity from its matching buckets just prior to the API invocation, and after the API execution, we re-evaluate the entity state and place it back into the appropriate bucket(s).

The aforementioned process depends on a well-populated cache to work. During the cache-population stage, we utilize the database access method “GetPreExistingEntity” mentioned above to pick random entities from the existing dataset and use the state evaluation step to place the entities in the appropriate buckets in the cache. Population of specific buckets may only be accomplished by executing a set of pre-defined steps, which may or may not leverage existing data. Therefore, the resulting cached data would then be a combination of sanitized data and synthetic data.

In practice, we observed that the entity state can usually be determined by parsing the results from “state-retrieving” API calls (e.g. Get calls). In cases where the state-retrieving API calls do not provide sufficient information, we construct custom data access methods to query the data store.

This cache-based approach can be implemented in any of the existing commercial applications for load generation since the model is focused on interactions with the underlying system and not the load volume.

5. Results and Future Work

We have described in this paper techniques for building proper environments and performance test tools which can be used to accurately simulate the same conditions observed in production (live)
environments, targeting large-scale stateful web services. The use of Data Sanitization, Markov Chain Stress Model, and Cache-based Load Simulation Tools have been successfully used for benchmark, capacity planning, and scalability tests of three major distributed web services: Subscription and Commerce Web Services, Identity Services Web Services, and Customer Assistance Web Services, all part of the Microsoft Member Platform Group. Accuracy of performance numbers collected in test laboratories have increased to a deviation of less than 5% from performance numbers observed in production environments (compared to ~9% with synthesized data). The number of real performance and functional issues found during the quality assurance process has increased by 15% with the introduction of the techniques described in this paper as part of the testing methodology.

Future work involves the enhancement and generalization of the techniques described in this paper, including:

- Extending the application of the data sanitization process to other data sources in addition to relational databases
- Real-time data sanitization
- Generalization of the application of Markov Chain Stress Model to different log sources
- Generalization of the Cache-based load simulation tools to automatically identify potential matching buckets based on the Finite State Machine for the system being tested.

6. References


