PERFORMANCE TESTING CONCURRENT ACCESS ISSUE AND POSSIBLE SOLUTIONS – A CLASSIC CASE OF PRODUCER-CONSUMER

Arpit Christi

Visiting Faculty
Department of Information Science
New Horizon College of Engineering, Bangalore, India
Christi.arpit@gmail.com

Abstract
Concurrent access issue is one of the most frequently encountered issues in performance testing. The issue is a classic case of producer consumer problem where multiple clients try to access same code on the server at the same time, resulting in collision in accessing the code/data. Readily available solutions to producer consumer problem are applied with proper implementation to solve the above mentioned issue. Relative pros and cons of each solution are also discussed.
***Keywords*:** performance testing, concurrent access issue, producer-consumer problem.

1. Performance Testing

Today most of the real world applications are client-server applications. Multiple clients send requests to server and server responds to these requests. Server can responds to limited number of client requests at a time satisfactorily depending upon multiple factors. These factors include server side code, client side code, physical specification of server (RAM, Cache, Processor speed etc), network bandwidth between client and server etc [4]. Here it is important to note that server just doesn't only mean an isolated single server. It can be a web farm consisting of large number of servers. Still, there is an upper limit on how many client requests it can serve depending upon above mentioned factors. Before any application is deployed and is started being utilized, it is important to find out how many simultaneous client requests it will be able to serve. Load Testing or Performance testing exactly does that. First an environment is created that imitates real world environment of server and multiple clients/users. Then certain requests are sent to server repetitively and response time, CPU utilization, memory utilization etc are measured and averaged to have better approximation of how the application is going to behave in real world environment where multiple users will send similar requests [5]. It is important to note that in most of client server application, server does all the heavy lifting so measuring performance of server is all what matters.

2. Performance Testing Environment – PERFLAB:

Performance testing environment imitates real world environment. We have here created a generic environment that we call PERFLAB and it envelops most of the real world scenario in a generic way. It consists of bunch of machines with specific tasks assigned to them. We don't want to have single machine to act as server, clients and data collectors. We have multiple machines (virtual machines) that we are using to carry out different tasks.

They are:
PerfServer – single server machine that acts as server of your application
PerfClient(s) – a client machine where multiple users will be created and then requests will be sent to client
Controller – a machine that monitors and collects all the parameters that are under considerations(server CPU utilization, RAM utilization, number of simultaneous users)
TestAuthor – A machine where tests are written and from where testing will be initiated
We are using Visual studio 2010 ultimate edition that allows us to do load testing [2]. This version is installed on TestAuthor machine.

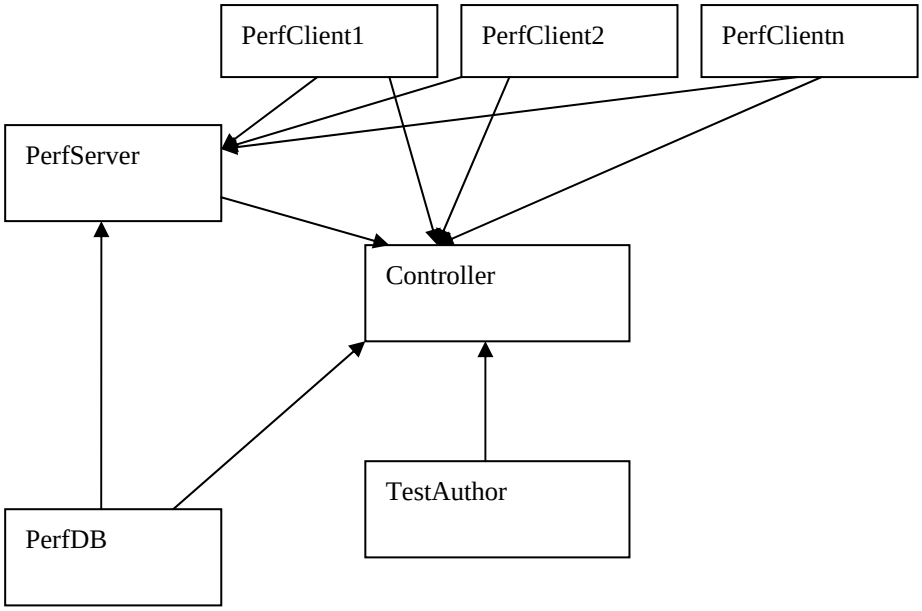


Fig. 1. Performance testing run for Pseudo Code 1

As shown in the fig 1,
TestAuthor triggers the process by activating controller.
Controller will in turn activate DB, Application server and clients.
Client will start sending requests to PerfServer.
As required PerfServer will query PerfDB(database) to get the data.
While the process is going on, controller will keep on collecting data from all the machines under consideration and create statistics out of it.

3. Automated Performance Tests (Unit Tests and Web Tests)

Performance of an application can be measured in different ways. Unit Tests and Web Tests are most common of them. Here unit tests differ from normal unit tests by the fact that instead of testing a single unit or module, it tests a scenario that represents some user action(s). For example, for an asset management system, a single unit test may represent following actions. Log in to system, search for certain kind of assets, pick one asset from these assets, update some fields for this asset and save the asset. Web tests does essentially same thing but in a different way. If web application is being used then web tests records user actions in form of series of HTTP requests and response and then plays it back to give you similar action(s). For our further discussion, we will just consider unit tests and not web tests, but very similar ideas can be applied to web tests without much difficulty. Also, to avoid any confusion with traditional unit tests, we will call them scenario tests [3]. Once this scenario tests are written, we will load them in our PERFLAB and run them against multiple users. So, client will create n number of users and run scenario tests allowing all n users to perform operations specified in scenario tests simultaneously. Test Initialization and Test finalization are important aspects of automated tests. Test Initialization lets us create environment/data for our test while Test finalization lets us delete data that was created as part of our initialization or tear down the environment [6].

4. Hypothetical Assest Management System and its operations

To make our case of “Performance Testing Concurrent Access Issues and Possible Solution” we will define our hypothetical asset management system and then we will write our scenario tests against this system. For simplicity we are allowing five operations for this Asset Management System. The 5 operations are
Create an asset – create an asset with all specification and then store it in persistent storage device.
Read an asset – Depending upon assetId, it will bring all fields of asset from persistent storage and displays it to user.
Update an asset – Allows user to update specific asset and then update the asset in persistent storage when user saves updates

Delete an asset – delete an asset.
Search for asset/assets – depending upon criteria provided by user, it searches asset/assets and display a list of assets.
We are not very much concerned on how this operations are performed but concerned about how they will behave when n number of users will run these operations simultaneously.
Persistent storage is a Microsoft SQL server database. We have single table in this database called Asset with following fields.

Table 1 Asset Table

AssetTable
AssetID – string
AssetDescription – string
AssetInstallationDate - Date

Initially we start with create operation only and as create operation is sufficient to demonstrate the problem. We have at our disposal an asset class that looks like following in class diagram.

Table 2 Asset Class Specification

Asset(Class)
AssetID – string AssetDescription – string AssetInstallationDate – Date
Bool CreateAsset();

Following is pseudo code for create operation

```
Public bool CreateAsset()
{
Asset myAsset = new Asset();
myAsset.AssetID = Assign Data Collected from user form/page
myAsset.AssetDescrpition = Assign Data Collected from user form/page
myAsset.AssetInstallationDate = Assign Data Collected From user form/page
bool success = myAsset.StoreAsset();
return success;
}
```

Here StoreAsset() function stores asset in persistent storage but before that it validates if there exists an asset with same ID and returns false if it exists. If not so then it will store this asset in persistent storage and returns true. Now, we write a scenario test against CreateAsset function and we call it CreateAssetTest.

```
TestInitialize()
{
}

Public void CreateAssetTest()
{
        Asset myAsset = new Asset();
        myAsset.AssetID = "ABCDEFGHIJKLMNOPQRST";
        myAsset.AssetDescription = myAsset.AssetID + " "  + "My Descprtion";
        myAsset.AssetInstallationDate = "11/11/2011";

        Assert.IsTrue(myAsset.StoreAsset(), " Asset already exists");
}

TestFinalize()
{
```

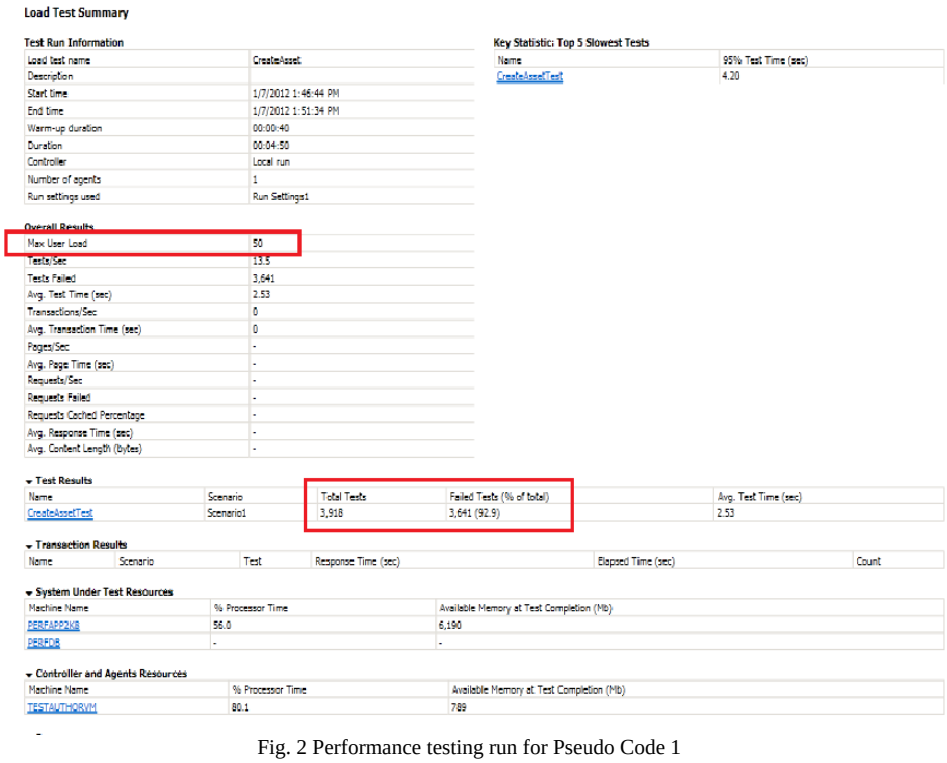
```
        Delete the asset so that it can be used fresh for next unit test run.
}
```

Pseudo Code 1

As TestFinalize() deletes the asset from database, next run of this scenario test will pass as the asset is deleted from database. One can run this unit test as many times, it will pass.

5. Concurrent Access Issue with performance testing:

Let's run the above mentioned scenario test in our PERFLAB. Now the code that creates Asset will become part of server side code and client will call this code to create assets. In our case, being precise, the PerfClient machine will create clients and run this scenario test and operations will be performed on server side. Using visual studio ultimate 2010 on Testauthor machine we will define how many users will be created [2]. We are starting with 50 users. So, basically PerfClient machine will have 50 constant users and they will simultaneously send request to create an asset to our PerfServer. Here is an example of our run generated at PERFLAB.



Load Test Summary

Test Run Information

Load test name	CreateAsset
Description	
Start time	1/7/2012 1:46:44 PM
End time	1/7/2012 1:51:34 PM
Warm-up duration	00:00:40
Duration	00:04:50
Controller	Local run
Number of agents	1
Run settings used	Run Settings1

Key Statistics Top 5 Slowest Tests

Name	95% Test Time (sec)
CreateAssetTest	4.20

Overall Results

Max User Load	50
Tests/Sec	13.5
Tests Failed	3,641
Avg. Test Time (sec)	2.53
Transactions/Sec	0
Avg. Transaction Time (sec)	0
Pages/Sec	-
Avg. Page Time (sec)	-
Requests/Sec	-
Requests Failed	-
Requests Cached Percentage	-
Avg. Response Time (sec)	-
Avg. Content Length (Bytes)	-

Test Results

Name	Scenario	Total Tests	Failed Tests (% of total)	Avg. Test Time (sec)
CreateAssetTest	Scenario1	3,918	3,641 (92.9)	2.53

Transaction Results

Name	Scenario	Test	Response Time (sec)	Elapsed Time (sec)	Count

System Under Test Resources

Machine Name	% Processor Time	Available Memory at Test Completion (Mb)
PERFAPP2K8	56.0	6,190
PERFDB	-	-

Controller and Agents Resources

Machine Name	% Processor Time	Available Memory at Test Completion (Mb)
TESTAUTHORVM	80.1	789

Fig. 2 Performance testing run for Pseudo Code 1

As you can see in Fig 2, failure rate is very high (92.9% for CreateAsset with 50 users measured at PERFLAB). Let's try to investigate the failure. As we have mentioned before when our test is finished, asset is removed from database. So, same asset ID can be used for coming tests. But we see here that for all of the failing tests, the error is, "Asset already exists". All 50 users are trying to create the asset at the same time by calling CreateAssetTest(). So, all 50 users are entering into test code at the same time. So, even before the first user can finish running the test, can go to TestFinalize() method and remove the asset, most of the users have entered into test method and have assigned their asset id to ABCDEFGHIJKLMNOPQRST. So when they will hit their StoreAsset() method, the validation for uniqueness will fail and StoreAsset() will return false and hence the test will fail. One possible case of above scenario is depicted below.

```
TestInitialize()
{
}

Public void CreateAssetTest()
{
        Asset myAsset = new Asset();
```

```
        User 31,32….50 are somewhere here. They
        will soon assign their asset ID to
        ABCDEFGHIJKLMNOPQRST

        myAsset.AssetID = “ABCDEFGHIJKLMNOPQRST”;
        myAsset.AssetDescription = myAsset.AssetID + “ “  + “My Descprtion”;
        myAsset.AssetInstallationDate = “11/11/2011”;

        User 2,3….30 are here. About to create new
        asset with asset ID
        ABCDEFGHIJKLMNOPQRST

Assert.IsTrue(myAsset.StoreAsset(), “ Asset already exists”);
}

TestFinalize()
{

        User 1 is here. About to delete the asset with
        Asset ID ABCDEFGHIJKLMNOPQRST

Delete the asset so that it can be used fresh for next unit test run.
}
```

The scenario is exactly producer consumer problem [7]. Users are consumers of our method that look for unique Asset ID, if they want to successfully create an asset, while the ID generation system is our producer. Currently there is no ID generation system in place or we can say that our ID generation system returns a single value (ABCDEFGHIJKLMNOPQRST). This kind of scenario is most common when scenario tests are used as performance tests and then loaded to performance labs. Surprisingly, without applying any smartness, we can get solution to producer consumer problem from textbook and implement it for this scenario and it will solve this problem also. Let’s try to apply three most basic solutions to producer consumer problem to the problem at our hand. The solutions are single buffer, bounded buffer and unbounded buffer [7]. We implement them for problem at our hands and discuss their pros and cons.

6. Single Buffer Solution

The problem that we have discussed already has a single buffer with entry of “ABCDEFGHIJKLMNOPQRST” [7]. The only way we can control the access of this single buffer is to lock it when a user is entered so that no other user can use the buffer until this user is finished using it and unlocks it [1]. We will modify our scenario tests accordingly and run our tests in PERFLAB again to see the test results.

```
Object _lockObject = new Object();
Public void CreateAssetTest()
{
        Lock(_lockObject)
{
        Asset myAsset = new Asset();
        myAsset.AssetID = “ABCDEFGHIJKLMNOPQRST”;
        myAsset.AssetDescription = myAsset.AssetID + “ “  + “My Descprtion”;
        myAsset.AssetInstallationDate = “11/11/2011”;
        DelayOf100MS();
```

```
            Assert.IsTrue(myAsset.StoreAsset(), " Asset already exists");
}
}
```

Pseudo Code 2 Single Buffer solution

Now test results are as follows.

Load Test Summary

Test Run Information

Load test name	CreateAssetSingleBuffer
Description	
Start time	1/10/2012 12:51:05 PM
End time	1/10/2012 12:55:55 PM
Warm-up duration	00:00:40
Duration	00:04:50
Controller	Local run
Number of agents	1
Run settings used	Run Settings1

Key Statistic: Top 5 Slowest Tests

Name	95% Test Time (sec)
CreateAssetSingleBufferTest	17.8

Overall Results

Max User Load	50
Tests Failed	1
Avg. Test Time (sec)	16.2
Transactions/Sec	0
Avg. Transaction Time (sec)	0
Pages/Sec	-
Avg. Page Time (sec)	-
Requests/Sec	-
Requests Failed	-
Requests Cached Percentage	-
Avg. Response Time (sec)	-
Avg. Content Length (bytes)	-

Test Results

Name	Scenario	Total Tests	Failed Tests (% of total)	Avg. Test Time (sec)
CreateAssetSingleBufferTest	Scenario1	837	1 (0.12)	16.2

Transaction Results

Name	Scenario	Test	Response Time (sec)	Elapsed Time (sec)	Count

System Under Test Resources

Machine Name	% Processor Time	Available Memory at Test Completion (Mb)
PERFAPP2K8	12.3	6,097
PERFDB	-	-

Fig. 3. Performance testing run for single buffer solution

As we can see in the figure, out of 837 only one test failed and that too for some random reasons. Failure rate is almost 0%. So, Single buffer solution solves the problem of concurrent access but it does so at the cost of concurrent access itself. With this solution, all other users are made to wait until current user enters into code and finishes execution of the code. That's why, though this test was run exactly same amount of time as previous run (4 minutes 50 seconds for both cases) number of tests came down from 3918 from previous case to 837 with single buffer solution. Though all 50 users are created almost together and they are trying to access the code together, with single buffer solution only one user is allowed to succeed with access of code. All 50 users are not running the code simultaneously that kills the purpose of load testing, loading code with multiple users at the same time. So, though single buffer solves the problem of concurrent access, it is not practically useful except in some cases where indeed certain portion of code is prevented from concurrent access and load testing is done for that portion of the code.

7. Bounded Buffer Solution

Bounded buffer solution [7] creates a buffer of some fix size and uses it to store producer's items and supplies one out of these items to consumer when requested until the item is available in bounded buffer. Let's try to implement this solution for our case. To understand bounded buffer solution better, we vary the size of buffer and run the load test in our PERFLAB. Bounded buffer solution algorithm is mentioned below. In this solution, producer is the process that adds asset ids to the buffer. Once a client/consumer has used this id, it is assumed to be removed from the buffer, so, if the same id is requested again, it's not available in the buffer and hence another client will not get the asset id and test for that client will fail. The same thing is achieved through our solution. TestFinalize() method removes the asset id from persistent storage, so by the time a client uses an id, but the test hasn't gone through finalization process, if other client requests the same id and try to create asset,

its test will fail as id still exists in persistent storage. Once an asset id is used by one client test and until the client is done with this test and finish its finalization process, that particular asset id is unusable by other clients, as bad as not being available in buffer. Once finalization is done for the a particular id used by a client, that id can be used by other client, as good as producer producing same id and putting it back to buffer.

```
Create a Static buffer of possible Asset ID with buffer size of n
        Array assetIDs = new Array(100)
        For I = 1 to 100
                assetID[i] = Some valid asset id

        public void CreateAssetBoundedBufferTest()
        {
        Asset myAsset = new Asset();
                myAsset.AssetID = Pick one asset randomly from assetIDs buffer
                myAsset.AssetDescription = myAsset.AssetID + " "  + "My Descprtion";
                myAsset.AssetInstallationDate = "11/11/2011";
                Assert.IsTrue(myAsset.StoreAsset(), " Asset already exists");
        }
```

Pseudo Code 3 Bounded Buffer solution

The scenario test is then loaded to our PERFLAB and results are generated using 50 constant users and buffer size of 25 and results are shown in fig 4(failure rate 21.4%). The same scenario test is then run using 50 constant users and buffer size of 50 and results are shown in Table 3 (failure rate 2.14%). Table 3 depicts failure rate of scenario test run using 50 users and buffer size of 10, 25, 50 and 100. It's very clear from this table that as buffer size increases with respect to number of users, failure rate decreases. Bounded buffer solution is bound to give better results when buffer size is equal to or greater then number of users, the later is preferred. Thus, problem of concurrent access is resolved using bounded buffer. Bounded buffer solution is not suited when number of clients is going to change drastically during load testing. With every change in number of users, buffer size need to be adjusted to make it bigger then number of users. Once your code is deployed in PERFLAB, this kind of adjustments may need a series of retesting and redeployment and is tedious and error prone. One can always keep buffer size very large, larger then the biggest anticipated number of users to solve the problem but in that case we have larger buffer occupying space is memory and selecting one item from buffer will become bit slower operation. Also, filling the buffer with larger number of items will be time consuming, increasing the preprocessing time of test. One more advantage this solution have over single buffer solution is that lock is applied nowhere and so all users are allowed complete freedom to act on code concurrently.

Table 3 Bounded buffer solution failure rate for varying user

No of Concurrent users	**Failure rate**
25	39.2
50	21.4
100	8.7
200	2.9

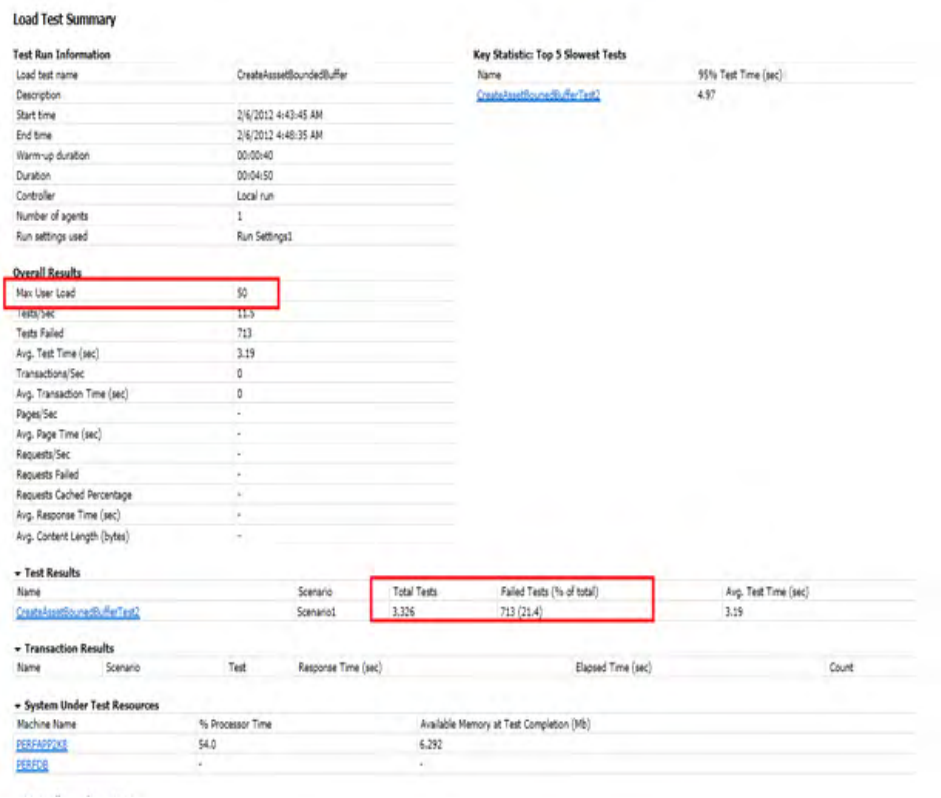


Fig. 4. Performance testing run for bounded buffer solution

8. Unbounded Buffer Solution

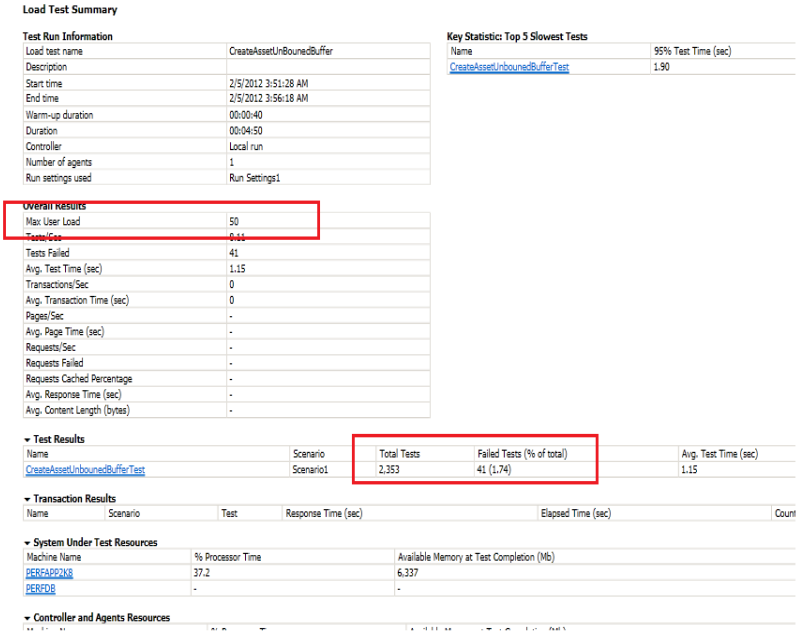
As we have noticed in bounded buffer solution, good results are produced if buffer is as big as number of concurrent clients or much bigger then that. Having a bounded buffer of certain size will guarantee good results in case of concurrent access against certain number of clients. In a practical scenario, clients can be varying and it can range in any number. Best solution that will solve concurrent access issue against any number of concurrently accessing consumers is to have unbounded buffer, a buffer of theoretically infinite size [7]. Following is the pseudo code of implementation of unbounded buffer solution to our load testing scenario. We have a randomizer class called Random that is capable of generating random numbers and strings of given size and limits. Class has two methods namely NextDigit that generates a random digit and NextString that generates a random string.

```
Public void CreateAssetUnboundedBufferTest()
{
        Asset myAsset = new Asset()
        myAsset.AssetID = Random.NextString(20); //Random string of 20 character
        myAsset.AssetDescription = myAsset.AssetID + " Description";
        myAsset.InstallationDate = "11/11/2011";

}
```

Pseudo Code 4 Unbounded Buffer solution

The scenario test is then loaded to our PERFLAB and results are generated using 50 constant users and results are given below.



Load Test Summary

Test Run Information

Load test name	CreateAssetUnBounedBuffer
Description	
Start time	2/5/2012 3:51:28 AM
End time	2/5/2012 3:56:18 AM
Warm-up duration	00:00:40
Duration	00:04:50
Controller	Local run
Number of agents	1
Run settings used	Run Settings1

Key Statistic: Top 5 Slowest Tests

Name	95% Test Time (sec)
CreateAssetUnbounedBufferTest	1.90

Overall Results

Max User Load	50
Tests/Sec	0.11
Tests Failed	41
Avg. Test Time (sec)	1.15
Transactions/Sec	0
Avg. Transaction Time (sec)	0
Pages/Sec	-
Avg. Page Time (sec)	-
Requests/Sec	-
Requests Failed	-
Requests Cached Percentage	-
Avg. Response Time (sec)	-
Avg. Content Length (bytes)	-

Test Results

Name	Scenario	Total Tests	Failed Tests (% of total)	Avg. Test Time (sec)
CreateAssetUnbounedBufferTest	Scenario1	2,353	41 (1.74)	1.15

Transaction Results

Name	Scenario	Test	Response Time (sec)	Elapsed Time (sec)	Count

System Under Test Resources

Machine Name	% Processor Time	Available Memory at Test Completion (Mb)
PERFAPP2K8	37.2	6.337
PERFDB	-	-

Controller and Agents Resources

Fig. 5. Performance testing run for Unbounded buffer solution

As one can see, failure rate is very less (< 2%) and most of the errors have to do with how PERFLAB is setup then actual failure because of concurrent access. Tests that actually fail because of same asset id are very less and negligible. Also, numbers of tests that run are quite large compared to that run for single buffer solution. In single buffer we stifled the full parallelism by locking certain parts of our scenario tests. Now things are running in full parallel with all 50 users allowed to create assets at the same time without affecting their parallel behavior at all. As an Asset ID, random id of length 20 is being used. If random generator function only uses character a to z then also the probability of collision of asset id is 26^{-20} that is $5.01e^{-29}$, almost none. Unbounded buffer solution with random string of 20 and character a to z is as good as having a buffer of 26^{20} and picking one item from it randomly. The best thing is that the buffer is never stored in memory at the cost of generating the string randomly every time when requested. Unbounded buffer scenario is most suited for most of the load testing cases as it allows full parallelism. One needs to have proper randomizer that will give random data as per need of the scenario test. Only overhead with this solution is the time taken to generate random data of n length by random function as that time adds to the time taken by the scenario test. One can safely at the end subtract the time taken by random function to generate random data from actual average test time to get accurate results.

9. Conclusion

A commonly encountered problem of concurrent access issue during load testing is discussed here. The problem is shown to be a case of producer consumer problem. Readily available solutions to producer consumer problem are applied to the problem at hand to solve it. Implementation of single buffer, bounded buffer and unbounded buffer solution is provided, results are presented and pros and cons of each solution are discussed. Single buffer solution is not much useful as it kills the purpose of letting users access code simultaneously. Depending upon one's need a load tester can decide between bounded buffer and unbounded buffer solution. If users are going to stay constant or very small change in user range is going to happen over a long period of time during load testing as well as the scenario suits best for preprocessing over doing some work in load testing method itself, bounded buffer solution is preferred. If users are going to be varied a lot in range to get results for varying user load or scenario suits best for generating some random data in load test method over preprocessing, unbounded buffer solution is preferred.

Acknowledgement

Author is thankful to Meridium INC (Roanoke, VA, USA) and Meridium services and labs Pvt. Ltd (Bangalore, India) for allowing him to use their existing performance lab to carry out experiments and producing results of performance tests.

References

[1] Birrell A. (2003). An Introduction to Programming with C# Threads (http://birrell.org/andrew/papers/ThreadsCSharp.pdf), 2003.

[2] Kamkolkar N. (2010). Getting Started With VS2010 Ultimate (http://blogs.msdn.com/b/nkamkolkar/archive/2010/11/02/getting-started-with-visual-studio-2010-ultimate-load-and-performance-testing.aspx), 2010.

[3] Kaner, C. (2003). The power of 'What If…' and nine ways to fuel your imagination: Cem Kaner on scenario testing, Software Testing and Quality Engineering Magazine, Vol. 5, Issue 5 (Sep/Oct), pp. 16-22, 2003.

[4] Liu, H. (2009). Software performance and scalability a quantitative approach, Wiley series on Quantitative Software Engineering.

[5] Meier J., Vasireddy S., Babbar A., Mackman A. (2004). Microsft® patterns and practices, Improving .NET application performance and scalability (http://msdn.microsoft.com/en-us/library/ff647788.aspx), 2004.

[6] Michaelis, M. (2005). A Unit Testing Walkthrough with Visual Studio Team Test, Microsoft® MSDN (http://msdn.microsoft.com/en-us/library/ms379625(v=vs.80).aspx), 2005.

[7] Silberschatz A., Galivn P., Gagne G. (2008). Process Synchronization, Operating System Concepts, Eightth edition, pp. 225-267, 2008.