by Pablo Santos and Francisco J. Garcia

Distributed Unit Testing

Supporting multiple platforms accurately and efficiently

At Codice Software we design and develop software configuration management tools that run on various combinations of operating systems and hardware platforms. For instance, both server and clients run on Windows XP/2000/2003/Vista, Linux, and Mac OS X. And since our software uses .NET, it runs on native Microsoft implementation for Windows and Mono for UNIX-like operating systems.

Pablo *is a Software Engineer at Codice Software. He can be reached at psantosl@codicesoftware.com.*
Francisco *is a Professor of Computer Science at the University of Salamanca. You can reach him at fgarcia@usal.es*

There's only one way to accurately and efficiently support platform combinations such as these—testing, and lots of it. However, running a software package on 36 platform combinations, release after release, is a Herculean effort. The solution we adopted was to distribute testing tasks by running them in parallel. In addition to uncovering bugs, this approach significantly speeded up the testing process. The key to our distributed testing solution is PNUnit, short for "Parallel NUnit." PNUnit is a modified version of the familiar NUnit (www.nunit.org) testing framework that was originally ported from JUnit to .NET. The source code and related files for PNUnit are available electronically; see "Resource Center," page 5.

We were already using NUnit to develop unit tests because our development is .NET based. NUnit lets you write unit tests with all .NET languages, and even adhere to test-driven development principles. Still, our primary concern was testing on multiple platforms using distributed test scenarios. Unfortunately, stock NUnit doesn't support this, hence our decision to extend NUnit to support distributed testing.

One of the reasons we wanted to stick with the NUnit framework is that we were familiar with its environment. Usually when you move to a new testing platform, the first thing you have to do is learn a new scripting language. By extending NUnit, we could use the same programming language and constructions (test suites, fixtures, and the like) that we were used to with regular unit testing.

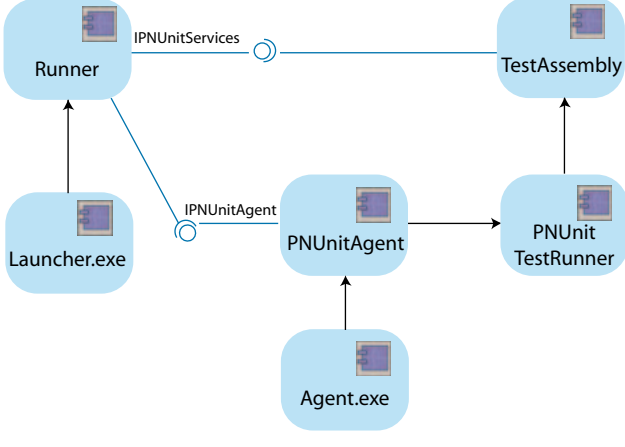


Figure 1: ***PNUnit high-level structure.***

Developing Automated Tests with PNUnit

Figure 1 shows the basic structure and main components of the PNUnit system. Launcher is the program responsible for launching test suites on test machines. It is called by a test configuration file as an argument, reads the file, and sends instructions to the testing machines. It then gathers test results and prints them on the screen. The test configuration (testconf) file is written in XML and contains a definition of the test scenario being created. It defines

```
<TestGroup>
     <ParallelTests>
          <ParallelTest>
               <Name>BasicLabelling</Name>
               <Tests>
                    <TestConf>
                         <Name>Server</Name>
                         <Assembly>cmtest.dll</Assembly>
                         <TestToRun>cmtest.server.Run</TestToRun>
                         <Machine>linuxbox00:8080</Machine>
                         <TestParams>
                              <string>../server</string> <!-- server path -->
                         </TestParams>
                    </TestConf>

                    <TestConf>
                       <Name>Client00</Name>
                       <Assembly>cmtest.dll</Assembly>
                       <TestToRun>cmtest.LoadTest.SimpleFileGet</TestToRun>
                       <Machine>winbox00:8080</Machine>
                       <TestParams>
                            <string>.\wkspaces</string> <!-- client path -->
                            <string>linuxbox00:8084</string> <!-- server name -->
                       </TestParams>
                    </TestConf>

                    <TestConf>
                         <Name>Client01</Name>
                         <Assembly>cmtest.dll</Assembly>
                         <TestToRun>cmtest.LoadTest.SimpleFileGet</TestToRun>
                         <Machine>winbox01:8080</Machine>
                         <TestParams>
                              <string>.\wkspaces</string> <!-- workspace dir -->
                              <string>linuxbox00:8084</string> <!-- server name -->
                         </TestParams>
                    </TestConf>

               </Tests>
          </ParallelTest>
     </ParallelTests>
</TestGroup>
```

Each ParallelTest defines tests to run in parallel. Different ParallelTests groups run sequentially.

To define a test we specify the assembly where the test is defined, the test fixture and the machine in which it will be run. Then each test can have params, specific of each kind of test.

Figure 2: ***Test script.***

the appropriate assemblies, specifies where they have to run, and identifies the data to execute. Figure 2, a typical testconf file, contains at least one parallel test section (test group). Each test group runs sequentially. Several tests are defined inside the test group and identified with TestConf labels. Each test inside the group runs in parallel, so each test group defines a parallel scenario.

The TestConf section tells Launcher in which assembly the test resides, which method to invoke, and where the machine is to launch it. Launcher uses this information to communicate with the Agent. To handle communications, Launcher creates a Runner for each test defined in a test group. The Runner is the component that calls the Agent methods. More importantly, Runner handles synchronization via the *IPNUnitServices* interface that it exports to the tests. Runner also gathers test results and informs Launcher about them.

Agents are located on test machines. Each test machine needs to have at least one Agent waiting to run tests. Agent is a small .NET application that, once started, registers a remote interface called *IPNUnitAgent*. This is the interface that Runner uses to launch the tests. *PNUnitAgent* is the actual component inside agents that handles the various processes.

Each time PNUnit receives a call in the *IPNUnitAgent::RunTest* method (meaning that a new test needs to be launched), it creates a new instance of *PNUnitTestRunner*. In NUnit parlance, *PNUnitTestRunner* is a real test runner. It creates a *TestDomain* instance, runs the test specified by the remote Launcher, collects the results, and notifies the remote Runner. Each test resides on regular assemblies, just as if they were normal NUnit tests.

Where does the NUnit framework fit in? It is the key layer in which *PNUnitTestRunner* resides. We take advantage of all of

TestConf Section Field	Meaning
Name	Test name you want displayed on the results screen.
Assembly	.NET Assembly in which the test resides.
TestToRun	Method name that defines the test to be run.
Machine	Machine where you want the test executed.
TestParams	Test-specific parameters.

Table 1: ***TestConf members.***

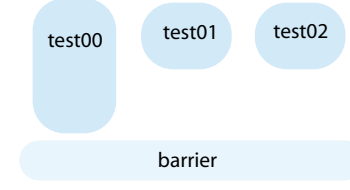
NUnit's functionality to load/run tests and collect results.

Synchronization Facilities

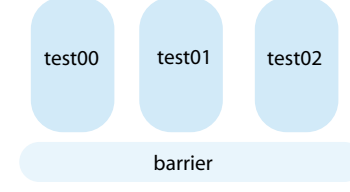
Runner is responsible for providing synchronization mechanisms for the tests. Example 1 presents the methods in the *IPNUnitServices* interface. For now, it simply provides a barrier-based synchronization mechanism.

You can initialize a barrier using Test by defining the maximum number of elements that pass through the barrier, or Test can just let Runner handle it. In the latter case, Runner assumes that all tests pass through the barrier, so it only depends on the definition made in the XML file. However, the former approach is useful when you need to define synchronization between a few participants in the test.

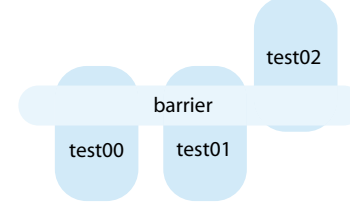
As Figure 3 illustrates, the barrier mechanism is a basic synchronization primitive. All tests must pass through a barrier before any of them are allowed to pass.



Step 1: Only test00 is hitting the barrier



Step 2: All clients hit the barrier, they can pass



Step 3: Tests passing through the barrier

***Figure 3:* Barrier mechanism in action.**

Listing One, the barrier implementation, shows that there are two important methods plus a constructor for the *Barrier* class. The constructor initializes the number of elements that must pass through the barrier to release it. The *Enter()* method is invoked when a test enters a barrier. If it doesn't enter a barrier and all of the other involved tests have gone through it, then the thread goes to sleep using a Monitor. The second method, *Abandon()*, is used to release barriers under fail conditions. For instance, if a test fails before hitting a barrier, *Abandon()* is called, so that an erroneous test won't freeze the rest of the tests.

With the *Barrier* method, we have been able to implement large test suites, covering all the basic core functionality of our system. Still, it would be easy to include some other primitives—semaphores, for instance.

Writing the Tests

At first glance, a PNUnit test doesn't appear that different from standard NUnit tests. All the facilities available for a normal NUnit test (assertions and the like) are also available.

Listing Two presents a couple of PNUnit tests.

The differences between a PNUnit and regular NUnit test involve the functionalities present in the *PNUnit.Framework* package. It includes a class called *PNUnitServices* through which the extended functionality can be accessed.

The *PNUnitServices* class provides methods for initializing a barrier,

```
public interface IPNUnitServices
{
    void NotifyResult(string TestName,PNUnitTestResult result);
    void InitBarrier(string TestName, string barrier);
    void InitBarrier(string TestName, string barrier, int Max);
    void EnterBarrier(string barrier);
}
```

Example 1: **IPNUnitServices *interface.***

Listing One

```
public class Barrier
{
     private int mCount;
     private int mMaxCount;
     private Object mLock = new Object();
     public Barrier(int maxCount)
     {
          mCount = 0;
          mMaxCount = maxCount;
     }
     public void Enter()
     {
          lock( mLock )
          {
               ++mCount;
               if( mCount >= mMaxCount )
               {
                    mCount = 0;
                    Monitor.PulseAll(mLock);
               }
               else
                    Monitor.Wait(mLock);
          }
     }
     public void Abandon()
     {
          lock( mLock )
          {
               --mMaxCount;
               if( mCount >= mMaxCount )
               {
                    mCount = 0;
                    Monitor.PulseAll(mLock);
               }

          }
     }
}
```

entering it as previously described, and retrieving the test name and test parameters. Why doesn't the test access the *IPNUnitServices* interface directly? Because of an implementation problem: Since the test runs on a different application domain than the *PNUnitTestRunner*, the interface received from the remote Runner must be made available to the running test. This is done with the *CreateInstanceAndUnwrap* API.

So the tests listed in Listing Two simply create a barrier, then use it to synchronize each other. The first barrier waits after initialization, making the second one wait for it in the *EnterBarrier* statement. With only these primitives, you can implement complex scenarios and distribute the tests over the network.

Listing Three is a typical configuration file. The script specifies both the methods where tests are implemented and the machines to execute them. Of course, the test machines must have an Agent started and listening in the correct port number—8080 by default—and the assemblies must be deployed, too.

PNUnit is a modified version of the familiar NUnit testing framework

Listing Two

```
using System;

using NUnit.Framework;
using PNUnit.Framework;

namespace SimpleTest
{
     [TestFixture]
     public class Test
     {
          [Test]
          public void FirstTest()
          {
               string[] testParams =
                    PNUnitServices.Get().GetTestParams();
               PNUnitServices.Get().InitBarrier("BARRIER");
               // wait two seconds
               System.Threading.Thread.Sleep(2000);
               PNUnitServices.Get().WriteLine(
                    string.Format(
                         "FirstTest started with param {0}",
                         testParams[0]));
               PNUnitServices.Get().EnterBarrier("BARRIER");
          }
          public void SecondTest()
          {
               PNUnitServices.Get().WriteLine(
                    "Second test will wait for first");
               PNUnitServices.Get().InitBarrier("BARRIER");
               // will wait for the first test
               PNUnitServices.Get().EnterBarrier("BARRIER");
               PNUnitServices.Get().WriteLine(
                    "First test should be started now");
          }

     }
}
```

Listing Three

```
<TestGroup>
     <ParallelTests>
          <ParallelTest>
               <Name>SimpleTest</Name>
               <Tests>
                    <TestConf>
                         <Name>FirstTest</Name>
                         <Assembly>test00.dll</Assembly>
                        <TestToRun>SimpleTest.Test.FirstTest</TestToRun>
                         <Machine>localhost:8080</Machine>
                         <TestParams>
                              <string>Option1</string>
                         </TestParams>
                    </TestConf>
                    <TestConf>
                         <Name>SecondTest</Name>
                         <Assembly>test00.dll</Assembly>
                         <TestToRun>SimpleTest.Test.SecondTest</TestToRun>
                         <Machine>testbox:8080</Machine>
                    </TestConf>
               </Tests>
          </ParallelTest>
     </ParallelTests>
</TestGroup>
```

Conclusion

Over the past few months, we've been using PNUnit for three types of tests—smoke, load, and merging tests. Smoke tests are run every time a developer finishes a task, together with regular NUnit test suites. Smoke tests basically cover core product functionality, and have been extended to include more and more test cases.

Each time we create a new release (typically once a week), the "integrator" (one of our team members) merges all the tasks to create a new baseline, then executes all the smoke tests. However, he does so by launching them on different platform combinations, covering all our supported platforms. Merge tests (a large test suite covering lots of branch-merging scenarios) are also executed during integration on different platforms. Finally, PNUnit tests are specifically designed to be scalable. Each test implements simple operations so they can be combined on the testconf script, and easily define load scenarios through the available test machines.

We've also started to use PNUnit on a cluster (about 50 Xeon machines), letting us test software under heavy load conditions. The framework has proved to be very useful on our project, having detected many bugs over the months.

There are still several aspects of the framework that need improvement. For instance, tests are currently only run from the command line, and results are gathered this way. A huge step forward would be to include GUI testing support in the framework. Also, it would be useful to be able to record user events and play them back as with other tools, but with the added power of multiplatform support. DDJ