"QA in the Era of 'Always On' systems" by Daniel Bigelow, Quality Assurance Lead, HPC-Portal Team – MHPCC

Not long ago, "24 by 7" was a techy term used amongst service providers to describe their requirement to keep computer servers and networks available all of the time. In today's fast moving and inter-connected world, it's expected that our utilities, phones, and network-based services are simply expected to 'be there' any time we access them – 24x7. This presents an interesting challenge for contemporary testing and quality assurance teams, who have typically been able to validate and verify system changes offline, before the software product or service was released for general access. Today, rather than being replaced, the systems being discussed almost morph from version to version, showing little, if any downtime.

Take any of the new DSRC HPC-Portals. They are each comprised of a general framework, where users can launch a variety of web-based applications. Not only does the framework stand on its own as a customized application, but each of its sponsored applications (e.g., Kestrel, MATLAB, WebShell, FileManager, etc.) interact with that framework, as well as supporting their own software functions. Any changes and upgrades to a portal's operating system and or framework that might cascade into the various applications, require that those applications be (re)tested and verified, to insure that any unexpected side effects can be quickly identified and resolved.

While the timeless goals of quality assurance teams (availability, responsiveness, reliability, etc.) still remain, the traditional tools and models no longer apply. In a web-based world, tracking user authentication, browser variations, network congestion, and security mechanisms all result in the need for an upgraded approach to quickly, efficiently, and deterministically validate these dynamic systems. Having five portals, with four or more applications on each, all of which need to be tested across eight browser families can no longer be efficiently tested using manual interactive methods (that is, many humans clicking on many web page buttons).

Fortunately, the tools and techniques used to identify system issues have matured right along with the systems they validate. The practical solution is twofold: extending classic unit-testing techniques to pre-test small code segments 'closer' to the user interface, and the use of automated interface testing tools.

In order to keep up with these rapidly changing systems, we're relegated to using programs to test our programs. In addition to using well-established Unit Testing systems like J-Unit, we now have automated testing toolkits like the Selenium Framework (www.seleniumhq.org) , and PhantomJS (www.phantomjs.org – a simulated web-browser), that allow us to pre-record user actions, convert those actions into test codes, and run those codes on servers that don't need (or even have) monitors, keyboards, or mice. In this way, we can quickly and repetitively run complete sets of test simulations.

Because humans still have to record, refine, and maintain the many test sequences that insure coverage of the key elements of our ever changing framework and its applications, it's not quite as simple as we'd like it to be. However, it's much more efficient, interesting, and productive to design and refine these automated testing systems, verses clicking on the same application repeatedly, on a variety of browsers, on multiple sites, and checking to see if it all still works properly.