Web services are at the heart of distributed computing, and interaction between them is often difficult to test. Distributed development, large teams of developers, and a desire for code to become more componentized, means that development of Web services is becoming increasingly more susceptible to obscure bugs. These types of bugs can be extremely difficult to detect. Stress testing is an efficient method of detecting such code defects, but only if the stress systems are designed effectively. This article will give some insights into the fundamental requirements of such stress systems.

Testing methods

Traditional testing methods include some form of simple Unit Testing, often performed by developers. These tests are designed with a knowledge of the software internals, and are nearly always aimed at testing a very small and specific part of the product. These kinds of tests are well-suited to simple Web services which have little or no interaction with other code components.

Functional Verification is a testing process in which designers, who have limited knowledge of the product source code, identify the core functionality of a product or service. Tests are designed to prove this core function conforms to the specification. For example, does my online auction display the correct bid entered? Does my insurance broker system find the cheapest quote? If these tests fail, a fundamental problem with the product has usually been detected (and usually a problem that is straightforward to fix). Again these tests are suited to simple Web services, allowing you to check whether a service performs its individual function correctly.

System Test usually occurs after the functional verification stage is complete, which is after the core function has been verified. It is intended to find problems with the entire system as a whole -- to see how Web services behave as part of a system and how they interact with each other. Since the system test phase occurs near the end of a development life cycle, there is often a lack of time allocated for its completion. Due to tight release schedules and slipping of development milestones, the stages of system test are often overlooked, and the unique bugs that each
uncovers too often go undetected. Even if such bugs are found, it is often too late to determine their cause and attempt to fix them. It is therefore imperative that system test applications are designed to be as efficient as possible in finding code defects. System test usually comprises of three areas. These are:

1. **Performance:** It involves the process of determining the relevant product statistics. For example: How many messages per second? How many simultaneous users of a service are acceptable?

2. **Scenario:** It is the process of recreating an exact configuration that a customer requires. Any problems found in the scenario can therefore be detected before the customer uses the product.

3. **Stress (or workload balancing):** It is different from the other two areas in that it is designed to strain the software by applying a large workload effort. If carried out effectively, by maintaining a highly strenuous usage of the product (but not beyond the limits determined by the performance statistics), stress testing often uncovers many obscure bugs that any of the other techniques mentioned above will not find (it is also often the case that they will be the most difficult to fix).

Arguably the most efficient of the three system test components, in terms of detecting code defects, is the area of stress testing. However, too often the process is confused with other elements of system or functional testing, and the methods involved in the process are not approached or implemented correctly.

**Stress bugs**

There are many varieties of bugs that you can expect to find with stress testing that are more difficult to uncover with other testing methods. Two types are:

1. **Memory leaks:** An extremely difficult phenomenon to detect. Memory leaks often find their way into released products simply because it is extremely hard to design test cases to detect them. With simple, functional tests, memory leaks are very rarely uncovered since the test does not generate enough usage of the product before it completes. Memory leaks often require operations to be repeated many, many times in order for the memory consumption to become significant enough to be noticed. Although it is more difficult to introduce memory leaks into Java programs than other languages, such as C/C++, it is still possible for objects to be instantiated and never de-allocated, as long as the program still holds references to the objects.

2. **Concurrency and Synchronization:** Stress tests excel in finding concurrency issues due to the many different code paths and timing conditions they exercise during any single test life-cycle. As a general rule, the longer a stress test runs, the more code-path combinations and timing conditions can be covered, and thus exercised. Of course, this does make it very difficult for these problems to be recreated (a defect that could occur after 5 minutes or 5 days). Deadlocks, thread leaks, and any general synchronization problems are often detected only at the stress testing stage. It is very difficult to find these types of problems by performing unit testing. A developer will not always consider how his or her code will interact with other areas of code (which may not have even been written at the time of the unit test).
Existing stress tools

There are a host of tools available that claim to be able to stress test products under development. An area with fairly widespread coverage are those tools aimed at Web services. However, many of these tools are simple HTML/SOAP generators, which simulate many client connections and therefore generate a high load on the Web server (which is useful for finding problems with the Web server, but not so good for finding problems with the Web services). These tools are useful for basic stressing, but often they merely extend the functional verification phase to repeatedly perform the same functional task. If enough time and resources are available, more effective testing can be achieved by creating custom-built stress testing systems. Since the designers of the stressing system will usually have more knowledge of the product and the Web services being tested, they will be able to ensure that the stress system is able to target specific areas of the code.

Designing stress applications

Test systems that attempt to stress a Web service need to be designed to exercise code in particular ways. These styles go beyond functional verification to see if a Web service not only does what it is supposed to, but also continues to perform as it should when certain stressful conditions are applied. There are four basic conditions which stress tests must apply to a Web service. Many established stress systems apply these conditions. Effective stress testing systems apply these key conditions:

1. **Repetition:** Probably the most obvious and easy to understand stress condition is test repetition. In other words, test repetition means performing a particular operation or function over and over, such as repeatedly calling a Web service. A functional verification test can be designed to see if an operation either works or does not work. A stress test will determine if an operation works and continues to work every time it is carried out. This is essential in concluding whether a product is fit to be used in a production situation. A customer will typically use a product repeatedly, and therefore stress testing should find the code defects before a customer finds them. Many naive stress systems implement only this condition, but simply extending a functional verification test to be repeated many times does not constitute an effective stress test. When used in combination with the following principles, repetition can uncover obscure code defects.

2. **Concurrency:** Concurrency is the act of performing several operations simultaneously. In other words, performing several tests at the same time, for example calling a number of Web services on the same server simultaneously. This principle may not apply to all products (such as stateless services), but the majority of software has some element of concurrent or multi-threaded behavior, which can only be tested by executing multiple instances of the code. A functional or unit test will rarely incorporate any concurrent design. A stress system must go beyond the functional tests to exercise multiple code paths at the same time. How this is done depends on the specific product. For example, a Web service stress test would need to simulate multiple clients at once. A Web service (or any multi-threaded code) will typically access some shared data amongst the thread instances. The complication that is added by this extra dimension of programming often means that code has many defects attributed to the concurrency. Since introducing concurrency means that code in a thread...
might be interrupted with code from other threads, defects are uncovered that are only found when a set of instructions are executed in a particular order (such as with a particular timing condition). By combining with the principle of repetition, you can cover many code paths and timing conditions.

3. **Magnitude:** Another condition that stress systems should apply to their products concerns the amount of load they apply in any single operation. A stress test can repeatedly carry out an operation, but that operation should also strain the product by itself. For example, if you have a Web service that allows a client to enter a message, you could incorporate high usage into a single operation by simulating a client that enters a huge message. In other words, you increase the magnitude of the operation. This magnitude is always application specific, but can be identified by looking for values in the product that can be measured and modified by a user - for example, size of data, length of delay, amount of funds transferred, speed of input, variety of input, etc. On its own, a single strenuous operation might not find a code defect (or might only find a functional defect), but when combined with the other stress principles, you increase the chances of finding a problem.

4. **Random Variation:** Finally, no stress system would be complete without an element of randomness. If you randomly use the countless number of variables that the previous stress principles introduce, you are able to cover many different code paths each time a test is run. The following are just a few examples of how you can vary a test during its lifetime. With repetition you could vary the time between repetitions, the number of repetitions before you restart or reconnect to service, or the order of Web services that are repeated. For concurrency you can vary the Web services that are carried out together, the number of Web services running at any one time, or whether to run a number of different services or a number of the same instance. Magnitude is perhaps the easiest to modify -- each time a test is repeated you can modify the variables that are present in the application (for example, sending assorted sizes of message or values of numerical input). Since it is difficult to consistently recreate a stress defect if the test is entirely random, some systems use random variation based on a constant random seed. In this way the defect has a higher chance of being recreated with the same seed.

A single stress test will typically combine all of the above, and run for as long a period of time as is allowed. The longer the test is allowed to execute, the more code paths are covered and the more defects can be found. Of course, once a defect is found it must be diagnosed and fixed. Since a code defect in a stress test could show itself after days of running, the system must ensure that all available debug information is generated when something goes wrong -- otherwise the same amount of time might have to be taken to recreate it.

**Conclusion**

Testing is an essential part of the software development process, and an important area that is often misunderstood or overlooked is that of stress testing. By following the principles detailed above, you can design and implement effective stress testing systems that aim to find some of the more devious problems associated with your code. Whether a pre-written tool is utilized or a fully specialized stress system is created, stress testing is an essential method for finding problems with Web services (or any other programs) and ultimately improving the quality of your software products.