Javier Palomares, Patrick Sigourney
Professor Kurchid
Software Testing
15 August 2018

Testing A Multithreaded Application

Abstract

We present a testing criteria for multi threaded applications which serve multiple requests in parallel. We call this testing criteria "parallel node coverage" and define it as a test requirement where the test artifacts fulfilling traditional node coverage are executed concurrently in order to assert correctness of the application during execution of parallel threads. We apply this criteria to a simple order management application and present our results.

Introduction

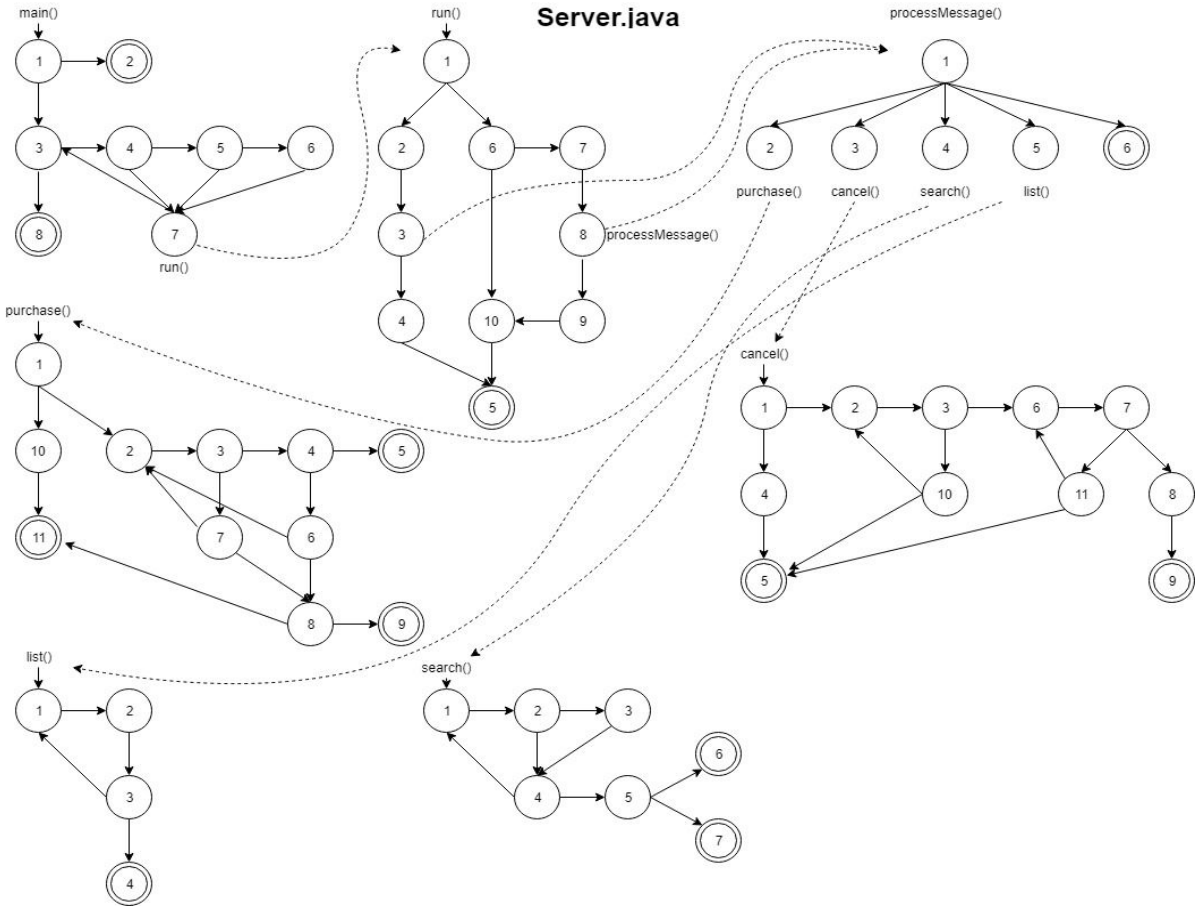
Client/Server is a commonly used architecture style across the software industry. Because of its popularity and wide use, we formally tested a Server application which had previously been developed as an assignment for the UT Distributed Systems course. This application implements an order and inventory management system using process threads. The server is initialized with an inventory list and assigned TCP port on which to listen for incoming client connections. The server allows clients to list items available, place orders for items, search for orders by customer name, and cancel orders. By using java.lang.Thread process threads the server is able to accept and process multiple concurrent client requests.

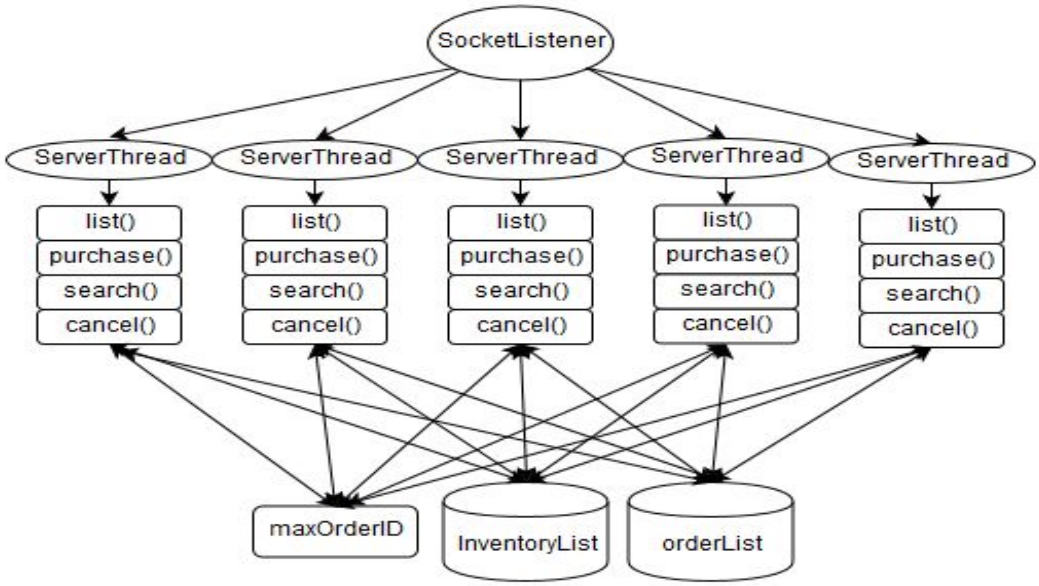
Technique

We began by forming a control flow graph for to represent the server source code. The server consists of several modules that are invoked dynamically based on messages received by the server.

We produced unit tests to satisfy traditional node coverage for all the application modules, integration tests to explore the interaction between modules, and finally multithreaded testing to simulate multiple clients simultaneously interacting with the server.

Illustration



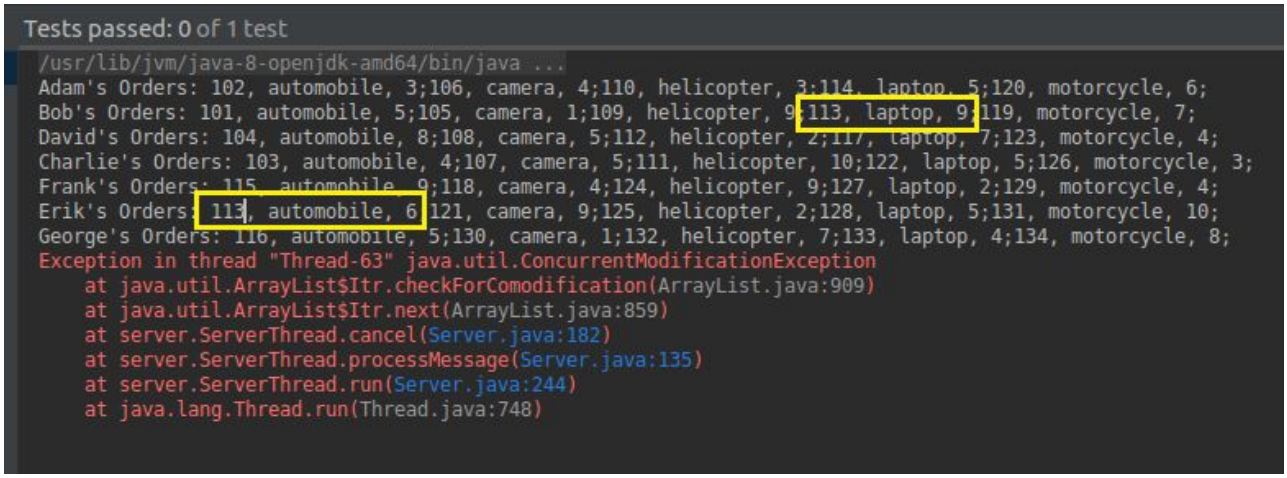


Evaluation

When the Server application was original written, the extent of testing performed involved starting a Client instance and sending various commands to see if anything broke; there was no methodical, systematic testing. After comprehensive testing with the test suite we developed, our results were mixed: the unit tests of each module performed as expected without error. The integration tests which evaluated module interaction also revealed no faults. It was only the multi-threaded tests where flaws began to emerge.

The transitive nature of the interaction between running threads resulted in cases where the test suite would pass one execution then fail the next. After some analysis of the failures, it became clear that the Server application was definitely not 'thread-safe'. The most glaring failures came in the form of frequent ConcurrentModificationException errors. These resulted from multiple clients attempting to place, search, or cancel orders simultaneously which updated single shared Inventory list and a single shared Orders list. To mitigate this flaw, a semaphore or other method could be implemented for both the Inventory and Orders lists to enforce exclusive access by a single thread at a time. Outside of this fundamental data-access issue, we also discovered a fault in the way order IDs were allocated which could be expressed when multiple threads attempted to simultaneously place orders. Occasionally, we would see the same order ID used for two different purchases. When this would occur and one of the threads would cancel its order, the result would have both orders with this order ID cancelled- even if one order belonged to a different Client.

Pictured: Bob and Erik both have orders identified by order number 113.



The identification of these faults would have been significantly more difficult without the use of an automated testing suite executing concurrent parallel threads to achieve parallel node coverage.

Conclusion

Parallel node coverage is an effective testing criteria for uncovering defects in multi-threaded applications which may not be detected by traditional coverage criteria.