Concurrency and testing – central principles and strategies

Matti Vuori, Tampere University of Technology

(translation Mikko Tiusanen)

TAMPERE UNIVERSITY OF TECHNOLOGY

Contents

Purpose of the slide set

- Slides consider testing issues related to concurrency
- Assumes that reader knows basics principles and techniques of testing – these will not be repeated.
- Approach is technology & testing tool independent and practical.
- Goal is to help find correct attitudes, principles, and strategies that help find the right testing methods to assess the most relevant facts for each situation, while understanding limitations of testing environments.

Challenges of testing

- Problems due to concurrency are day-to-day occurrences.
  - And locating errors in the code is hard.
  - Even repeating error situations is often hard.
- Testing concurrency problems using traditional methods is hard.
  - Traditional unit testing that treats methods and functions does not work, tools do not support this and do not find its problems.
    - Unit testing isolates the test object, whereas with problems of concurrency the issue is the delicate interaction of many things.
    - Test tools, monitoring, and logging change the timings.
  - Test environments differ from production use environments – production use will unearth new problems.
  - Testing knowlege needs to be applied creatively.
- If somewhere ”the devil is in the details” then in concurrency!

Central principles 1/2

- Understand where the program and component will be used.
  - What kind of use.
  - What kind of environment.
  - What is going on there.
  - How does the architecture operate.
- Assume that each component has faults.
  - Your own and every single one of the others.
- Load will unearth the faults.
  - Put the software under stress in testing.
  - Make sure in tests that software is not under stress in use (that is, it is not loaded too much in the run-time environment, based on the tests).
- Expect the worst in all operations that treat shared resource – and test that even the worst will be handled correctly.

Central principles 2/2

- Assume that all the other processes, threads, etc, can do anything and test that this is tolerated and handled correctly.
- Everything may turn upside down if the run-time environment is changed – so, change it in testing.
  - Assumed speed of execution.
  - Load.
  - The behaviour and speed of other components.
- Value randomness and variation in testing.
  - Bring out the problems by changes.
- The tighter the timing constraints, the more sensitive the program is to disturbances.
- Exaggerate everything in testing – and then some more...
- Test the adaptability to future changes.
  - Say, the processor changes...

Code reviews important

- Code reviews are not testing, strictly speaking, although they are sometimes counted as static testing.
- Code reviews are an important step for early quality assurance.
- They are the more important, the harder the testing of the object is – and concurrent ones are just this.
- The next slides treat various concurrency related themes that need to be checked in code reviews.
  - Check lists are generally a good idea to support reviews.
  - These can be built for various application areas, taking the known technological pitfalls of each into consideration.
  - It is obvious that maximizing the level of warnings from the compiler and static code checking programs is recommendable.

Code review topics 1/2

- Code clarity… complicated code is susceptible to problems of concurrency.
- General strategy for robustness.
- Checking all aspects of concurrency.
- Architecture.
- Right strategy for multiprocessing – if there are options.
- Correct implementation of ”patterns” implementing low-level concurrency.
- Right strategy for asynchronous operations – if there are options.
- Right message passing strategy – if there are options.
- Critical sections, locking.
- Avoiding unnecessary locking of resources.
- Avoiding locking resources for extended periods.

Code review topics 2/2

- Checking that resources are ready before allocation.
- A defensive strategy for resources being free.
- Resource allocation.
- Thread performance.
- Computationally heavy operations – synchronous ones, in particular – can they be split?
- Timeouts.
- Recovering from waits.
- Identification of API function calls risky for concurrent execution.
  - Thread safety etc.
  - Using tools for this is recommended.
  - Suitable as a part for integration testing.

Code review on other levels of system abstraction

- A similar review is useful when performed on all levels of system abstraction.
  - From the smallest method to the mass use of databases and the overall architecture.
- Evaluation of the architecture looks at the total construction.
  - How to handle various scenarios:
    - Heavy load conditions.
    - Communication being slow or erratic.
  - May use specific architecture assessment methods, but even a good review with check lists is advantageous.

Characteristics of testing

- With concurrency there are many ”moving parts" and it is useful to think that not all phenomena are known in advance.
  - Therefore, one cannot rely only on preplanned test cases.
- Testing needs to be directed reactively in response observed behaviour. When finding something funny, investigate its causes – is it a symptom of something bigger that may cause the program to blow up shortly.
  - This is so called exploratory testing, http://en.wikipedia.org/wiki/Exploratory_testing

Testing levels

- Testing has traditionally been divided to levels, say:
  - Unit testing.
  - Integration testing.
  - System testing.
  - Acceptance testing.
- Concurrency shows up differently on different levels.
- Traditional unit testing isolated the test object from the others, but in testing concurrent systems it is important to create interactions!

Testing environment

- The testing environment needs to correspond to the production use environment as closely as possible.
  - Performance.
  - System set up and configuration.
  - The smallest of change in the set up can make a difference.
- Variation of timing:
  - In different environments.
  - With different other processes.
  - With different system clock (speed).
  - Slowing down other processes and events.
  - More speed – what if the processor is upgraded?
- Varying the environment parameters helps to see effects of changes.

Loading

- For example, the VR (Finnish railways) systems are told to have got stuck completely when database row locks were escalated to relation and relation page levels (VR problems 2011).
- A contributing factor was a platform bug that was realized under heavy load.
- That is: more processes, users, load in operations.
  - Bigger files, more data.
  - Slower communications for system, faster for generating load.
- Performance / load testing.
  - Testing under various loads.
  - For how many processes the performance is still ok.
  - What happens as the number is increased.

Robustness

- Asynchronous tasks.
  - Interrupt handling.
  - Testing exception handling.
- Disturbances in other components.
  - Mock-ups, simulators as aids for testing.

Simulation 1/2

- Simulator and emulator testing do not tell much.
  - Speed is not the same as in the production environment.
- But!
  - It is valuable to simulate other components, other software.
  - Simulation of machine and automation systems can exaggerate their behaviour in ways that is not possible to test in a real environment.
  - Produce random events.
  - Produce disturbances – it only needs a small stuck part to mess up even railroad traffic.
- The traditional tool is a stub that realizes the behaviour of another component in a simplistic way.
  - For each stub it is useful to consider building a more variable simulation – and introducing disturbances to the system using it.
  - Mock-up is a reasonably current and relevant concept here: http://en.wikipedia.org/wiki/Mock_object

Simulation 2/2

- Model-based testing.
  - http://en.wikipedia.org/wiki/Model-based_testing
  - Make (state) models of other parts of the system and vary their behaviour.
  - Make a model of the own part.
    - A model to cover the essential issues to be tested that can be put into interaction with the environment.

Observe during testing 1/2

- Test cases and tools are needed for this, including various monitoring tools.
- ”Visible things” – behaviour.
  - Will the expected things be realized at all:
    - Does everything occur.
    - Do all messages leave.
    - Do all expected things get output to screen / log.
    - Etc…
  - Ordering – things happen in correct order.
  - Being error-free – no errors in any situation (timing, load).
  - Observations of time: delays, speed, speed variation.
  - All kinds of phenomena in load testing.

Observe during testing 2/2

- Under the hood:
  - Data integrity when being transferred between concurrent processes.
  - Exception handling.
- Monitoring.
  - Load – processor, communications, etc…
  - Number of processes, threads and others.
  - Speed of execution.
  - Free space for buffers, queues and variation of it.
  - Resource use and utilization.
  - Execution profiling.
    - Bottlenecks.
    - Variations under different circumstances.
  - Exceptions.

Five keys of testing concurrent program

1. Even small changes can affect the behaviour of the program. Starting with the stage of the moon…
2. Testing <u>cannot</u> prove the correct behaviour of a program in the target environment in a particular situation, never mind for a concurrent program. It can only provide (some) information about errors and situations where the program has been observed to operate correctly.
3. Testing needs to be brutal and to create changes, disturbances, and variations, exaggerating these so that a necessary robustness can be ”assured”.
4. The tester needs to be sensitive to different phenomena in testing so that results can lead to the potential problems in the program.
5. Code reviews are important and it is useful to employ them.