COSC345 Software Engineering

Complex Systems and Exception Handling

Richard A. O'Keefe

July 13, 2017

References

- "Kode Vicious: Forced Exception Handling", George V. Neville-Neil, CACM 2017 vol.80 No.6 pp 31–32.
- "Simple testing can prevent most critical failures", Adrian Colyer, Blog "the morning paper", 6 Oct 2016.
- "Simple Testing Can Prevent Most Critical Failures: An Analysis Of Production Failures In Distributed Data-Intensive Systems", Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm, 11th USENIX Symposium on Operating Systems Design and Implementation, 2014.

What Yuan *et al* did

- They looked at 5 distributed systems: Cassandra, HBase, Hadoop Distributed File System, Hadoop MapReduce, and Redis.
- They selected 198 failures at random from a collection of user-reported failures.
- They tried to understand how single or multiple *faults* because user-visible *failures*.

How Complex Systems Fail

- What should we *expect* Yuan *et al* to find?
- We should expect failures to result from the interaction of multiple faults.
- We should expect the parts of the programs that are supposed to handle failures to be the parts that fail.

Findings 1

- Almost all failures require only 3 or fewer nodes to reproduce.
- Multiple inputs are needed to trigger the failures with the order between them being important.
- The error logs of these systems typically contain sufficient data on both the errors and the input events that triggered the failure, enabling the diagnose and the reproduction of the production failures.

Findings 2

- The majority of catastrophic failures could easily have been prevented by performing simple testing on error handling code . . . even without an understanding of the software design.
- Over 30% of the catastrophic failures would have been prevented [using] three simple rules.
- Almost all (92%) of the catastrophic system failures are the result of incorrect handling of non-fatal errors explicitly signalled in software.

The Three Flaws

- A handler catches errors but does nothing about them (other than writing a log entry).
- A handler has a catch that is too general and aborts the system.
- A handler has “TODO” or “FIXME” in it.

History 1

Exception handling is younger than procedure calls and garbage collection (1950s) but older than object orientation (1967).
(ERRORSET *form*) in Lisp 1.5 (1960) evaluates *form*. If an exception occurs, ERRORSET returns NIL, otherwise the value of *form*.
(ERROR *culprit*) raises a specific exception and saves *culprit* where handling code can find it.

History 2

Popularised by PL/I (1965). Present in Burroughs Algol (1965).
Language-defined fixed set of exceptions.
Elsewhere exceptions generally handled by GOTO (Fortran alternate returns) or callbacks (Algol 68 file error handling) or IF statements conditional on return codes (BCPL, C).
CLU (1974) exceptions were declared and bound to methods.

A key problem

All other name-based associations in a program are *static*. Use a variable? The name must be in scope.
Call a procedure? The name must be in scope.
Raise an exception? You have no idea where the handler will be.
CLU and Java try to tame this by forcing you to declare the exceptions a procedure can raise, but some exceptions are so common Java gives up on those.

Flaw 1: Ignoring errors

Often seen in student code:

```
try ... {
        ...
... } catch (Exn e) {
        System.err.println(e);
}
```

If the exception is raised, it is logged, but the program otherwise continues as if nothing had happened. Variables and objects are typically in a bad state.

Don't do that!

An exception handler **must** either

- restore all variables and objects that the **try** block may have changed/was supposed to set up to a usable state, or
- raise another exception, or
- kill the thread, or
- kill the process.

But how do we know what changed?

```
try . . . {
        foo.bar( . . . );
        ick.ack( . . . );
} catch (Exn e) {
        which statement raised e?
}
```

Keep **try** blocks simple.

The bigger the block, the less you know about what might have changed and where the exception might have come from.
Each **try** block should do *one* task simply.

How do we know what changed?

- Make variables **final** if possible so you know the **try** block couldn't change them.
- Use immutable objects as much as you can.
- Mutable objects that could be shared with other code should not be used in **try** blocks.
- Anything else must be assumed *dangerously wrong* after an exception.

Flaw 2: Over-general **catch**

- There is an exception class Exn with subclasses Exn1 and Exn2.
- You have a **try-catch** construct where Exn1 and Exn2 should be handled the same way.
- You write a single **catch** for Exn.
- But then or later, Exn also has a subclass Exn3 which should *not* be handled that way.
- Oops.

Don't do that

- Be *extremely* cautious about handling an exception that has subclasses.
- If two exceptions should be handled the same way, write two handlers calling a common procedure.
- If your programming language won't let you do that, find a better programming language.

Flaw 3: TODO/FIXME

- Why are you shipping code like that?
- You must test **all** your code, including exception handlers.
- Use test coverage tools to make sure you have done so.
- Ensure that there is a way to inject faults so that you can exercise handlers.

Alternative views

"The very real problems of survival after a component violates its specification is not addressed by exception handling." — Andrew P. Black, Thesis 1982, *"Exception Handling: The Case Against."*
"Let it crash!" — Joe Armstrong, designer of Erlang. Light-weight processes, no mutable variables, crashed processes can dump their state for debugging and can be automatically restarted by a supervisor process.