Fault Tolerance and Robustness in Concurrent Systems

4010-441
Principles of Concurrent Software Systems

Faults, errors, failures, and fault tolerance have many different definitions.

What working definition should we use for fault?

What does it mean to be fault-tolerant?

What are possible faults in the Sleeping Barber system?

If not handled, faults can exhibit themselves in a system in a number of different ways.

- Actions – the wrong actions are performed
- Timing – the right actions are performed but at the wrong time
- Sequence – the right actions are performed but in the wrong sequence
- Amount – the wrong number of actions are performed

Fault-tolerance is a system level attribute that needs to be designed in rather than tacked on.

In a broad sense, what are the two major categories of activities that have to go on to achieve fault-tolerance?

A simple software watchdog is a first detection mechanism.

Software components are required to report a heartbeat to their supervisor or to a central monitor. The assumption is that as long as the heartbeat is received the component is working.

How much does this tell us about the operation of the component?

What could be an extension to the simple watchdog concept that could tell us more?

There are a number of responses that can be taken once you find out that something is wrong.

What are some approaches that can be used to deal with a broken component, and an operation that may not have been done correctly?

What concerns do you have to consider?

First, we will establish some terminology.

- Cancellation
  - ***Task level termination***
  - ***May or may not result in stopping threads***
- Interruption
  - ***Thread level termination***
  - ***Get a thread to terminate with or without completion of the current operation***
- Shutdown
  - ***Application or service level termination***
  - ***Stop all tasks, and associated threads, with or without completion***

These definitions are not necessarily universally accepted.

If you are not using a framework with fault handling, you will have to deal with it all yourself.

- A framework without fault handling may not give you many options
- Define cancellation and interruption policies
  - ***How to do it, when it is checked, what is done***

It may look like Java gives you some tools at the thread level, but not really.

- Everything will have to be cooperative

In the Thread class, take a look at:

`public void interrupt()`
Interrupts this thread.
`public static boolean interrupted()`
Tests whether the current thread has been interrupted.
`public boolean isInterrupted()`
Tests whether this thread has been interrupted.

How do you use these to initiate an interruption, and to process it? Why is this a cooperative approach?

You have some design decisions to make regarding how to handle being interrupted.

- At the task level
  - ***Finish current work or stop immediately***
  - ***Does it own the thread?***
    - Yes, end the thread?
    - No, i.e. it's running from a thread pool, let thread manager handle it for the thread
      - Preserve interrupted status
      - Throw InterruptedException
- At the thread level
  - ***Propagate interrupt if where it is detected does not implement interruption policy***
  - ***Otherwise, implement interruption policy***

There are other things that you need to consider if you want to build a fault-tolerant system.

- What is the most common indication that your program had a problem?

```
Exception in thread "main" java.lang.SomeException
        at com.example.myproject.Class1.method1(Class2.java:16)
        at com.example.myproject.Class2.method2(Class3.java:25)
        at com.example.myproject.TopClass.main(TopClass.java:14)
```

If it is operationally critical that the system keeps running, tries to recover from errors, or at a minimum does a graceful, failsafe shutdown, what do you do?

Shutdown of a service should take down all tasks and threads that it owns.

- At the task level
  - ***Let a running task complete?***
  - ***Let scheduled but not started tasks complete?***
  - ***Provide information about what work was not finished.***
- Once tasks are handled, interrupt threads in pool
- `ExecutorServices` provide some support
  - ***`shutdown()`***
  - ***`shutdownNow()`***
  - ***`awaitTermination()`***