Software Defences Against Hardware Failure

6th Scandinavian Conference on System & Software Safety

Chris Hobbs

QNX Software Systems

V1.0

May 20, 2018

Prolegomenon: Random Software Errors

Most of the safety standards assume:

- that hardware errors can be random.
- that all software errors are systematic.

(Mechanical) hardware errors appear to be random because we don't know the states of all the molecules and don't have the processing power to deduce the outcome.

Prolegomenon: Random Software Errors

Most of the safety standards assume:

- that hardware errors can be random.
- that all software errors are systematic.

(Mechanical) hardware errors appear to be random because we don't know the states of all the molecules and don't have the processing power to deduce the outcome.

An OS has many more states than there are nucleons in the universe.

Prolegomenon: Random Software Errors

Most of the safety standards assume:

- that hardware errors can be random.
- that all software errors are systematic.

(Mechanical) hardware errors appear to be random because we don't know the states of all the molecules and don't have the processing power to deduce the outcome.

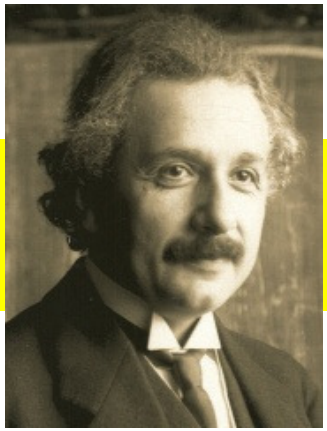
An OS has many more states than there are nucleons in the universe.

I have a 33 line C program that fails roughly every 300,000,000 times it is run. It does not use (pseudo-)random numbers. How does this differ from hardware failure?

The Problems

1. *The increasing frequency of random hardware and software errors*
2. *The increasing complexity of the two aspects of system design*

Probleme kann man niemals mit derselben Denkweise lösen, durch die sie entstanden sind.
Albert Einstein



The First Problem

The increasing frequency of random hardware and software errors

Modern processors come with 20+ pages of *errata*: "Sometimes instructions are executed out of sequence. Work-around: None."

The First Problem

The increasing frequency of random hardware and software errors

Modern processors come with 20+ pages of *errata*: "Sometimes instructions are executed out of sequence. Work-around: None."

Cosmic rays, cross-talk and EMI cause bit, byte, word, row and column flips. These errors affect DRAM, cache, registers, etc. STOP PRESS: Last Thursday researchers at the Vrije Universiteit, Amsterdam, shewed how to use row-hammering from Javascript in a browser to hack a smart 'phone.

The First Problem

The increasing frequency of random hardware and software errors

Modern processors come with 20+ pages of *errata*: "Sometimes instructions are executed out of sequence. Work-around: None."

Cosmic rays, cross-talk and EMI cause bit, byte, word, row and column flips. These errors affect DRAM, cache, registers, etc. STOP PRESS: Last Thursday researchers at the Vrije Universiteit, Amsterdam, shewed how to use row-hammering from Javascript in a browser to hack a smart 'phone.

Multi-threaded applications (on multi-core processors) produce random software errors.

The First Problem (cont)

According to *"DRAM Errors in the Wild"* we can expect 25,000 to 70,000 bit-flips per 10^9 hours per Mibit.

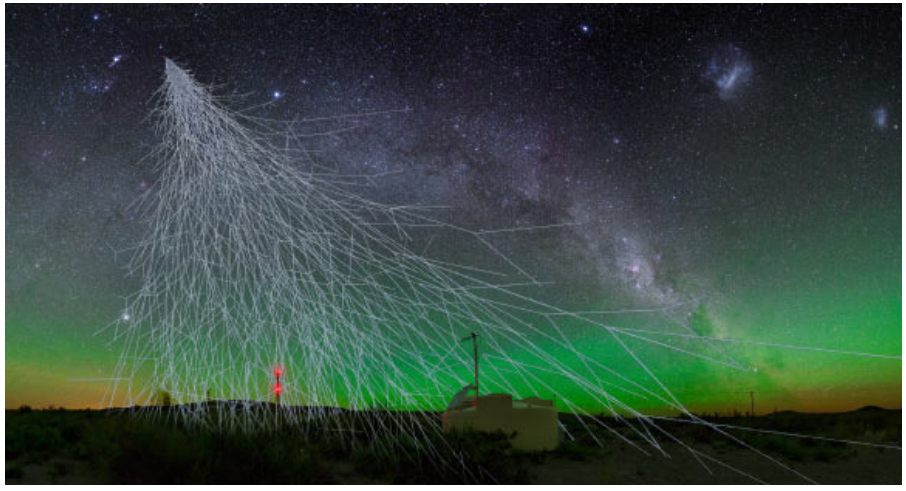


Image credit: A. Chantelauze / S. Staffi / L. Bret / Pierre Auger Observatory.

In an embedded system with 2 Gibytes RAM, a bit error can be expected every hour or so. We can't bring the system into its Design Safe State that often.

"All software running on a modern microprocessor is non-deterministic" (Rob Ashmore, SCSS'17, Bristol, February 2017)

The Second Problem

The increasing complexity of the two aspects of system design

1. The calculation of the safety-critical values.
   "Given our speed, the wet road, the slight incline, the distance to the car in front, etc., we should apply a brake force of 450.3 Newtons"
2. The availability/reliability balance of the system.
   "We are travelling on the highway: availability is more important than reliability: 1oo3 configuration"

We need to decouple these two aspects of the system.

The Second Problem

The increasing complexity of the two aspects of system design

1. The calculation of the safety-critical values.
   "Given our speed, the wet road, the slight incline, the distance to the car in front, etc., we should apply a brake force of 450.3 Newtons"
2. The availability/reliability balance of the system.
   "We are travelling on the highway: availability is more important than reliability: 1oo3 configuration"

We need to decouple these two aspects of the system.

Requires mathematical expertise

The Second Problem

The increasing complexity of the two aspects of system design

1. The calculation of the safety-critical values.
   "Given our speed, the wet road, the slight incline, the distance to the car in front, etc., we should apply a brake force of 450.3 Newtons"
2. The availability/reliability balance of the system.
   "We are travelling on the highway: availability is more important than reliability: 1oo3 configuration"

We need to decouple these two aspects of the system.

Requires mathematical expertise

Requires statistical expertise

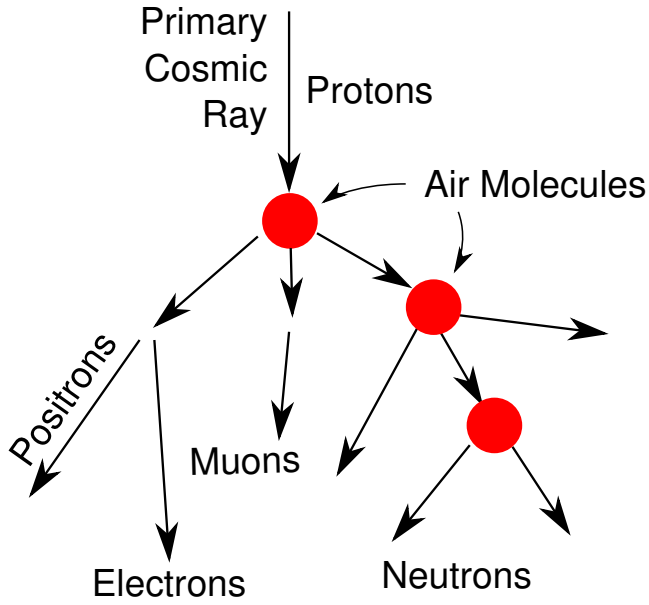
The Second Problem (cont)

Conflating these two problems leads to over-complex, sub-optimal and inflexible implementations.

Making them independent solves these problems and makes verification (including testing) much more effective.

The Changing Question — 1 of 2

Has a random hardware error occurred?



The Changing Question — 1 of 2

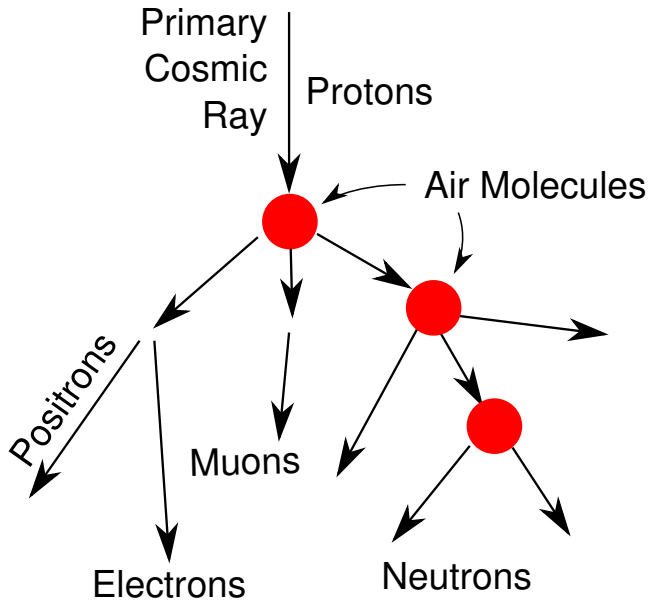
~~Has a random hardware error occurred~~?

They occur too often and their effect cannot be predicted, especially in an accidental system.

Today's important question:

Has something happened that has negatively affected the safety of our system?

So can we ignore hardware errors?



The Changing Question — 2 of 2

How can we build the required dependability into this application?

The Changing Question — 2 of 2

~~How can we~~ build the required dependability into this ~~application~~?

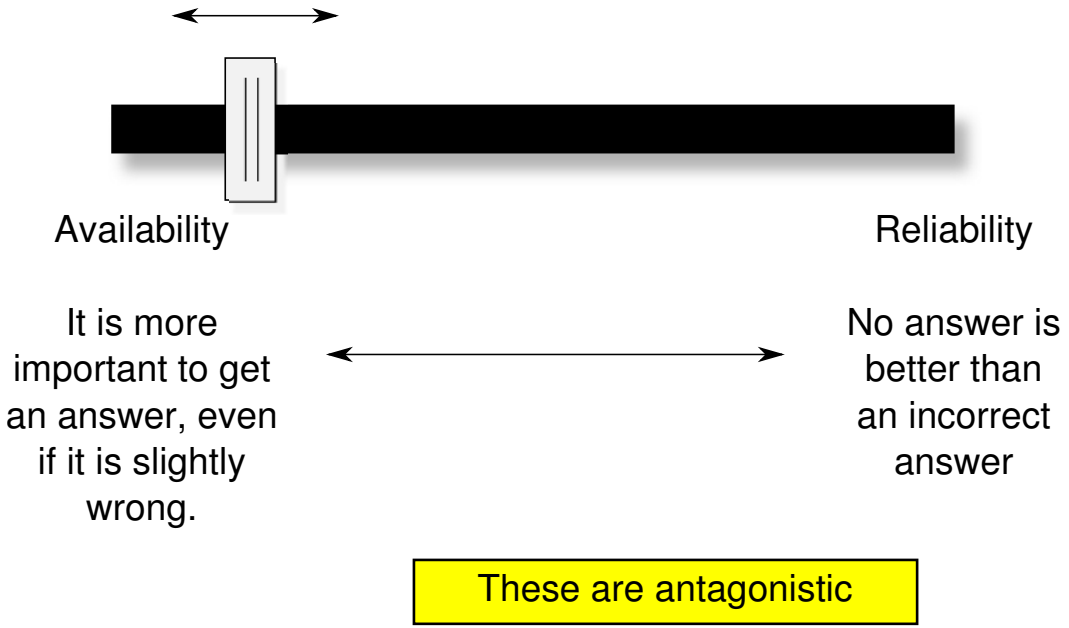
The dependability is independent of the application.

Today's important question:

How can we separate the design of application from the design of the system dependability so that we can tune one without affecting the other?

Hardware Lock-Step: an inadequate solution

- The number of replicas is defined by the hardware and cannot be changed: the system cannot be tuned to local conditions.



Hardware Lock-Step: an inadequate solution

- The number of replicas is defined by the hardware and cannot be changed: the system cannot be tuned to local conditions.
- It is impossible to design the system around *diverse* implementations.

  Hardware Lock-Step generally supports only *replicas*.

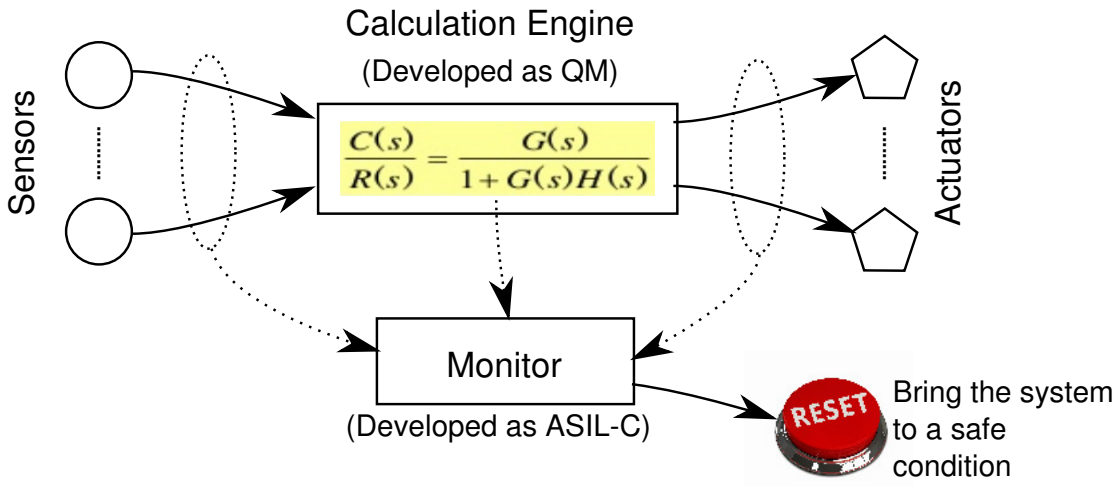
  Without diversification it is impossible to implement the monitor pattern ("Safety Bag").

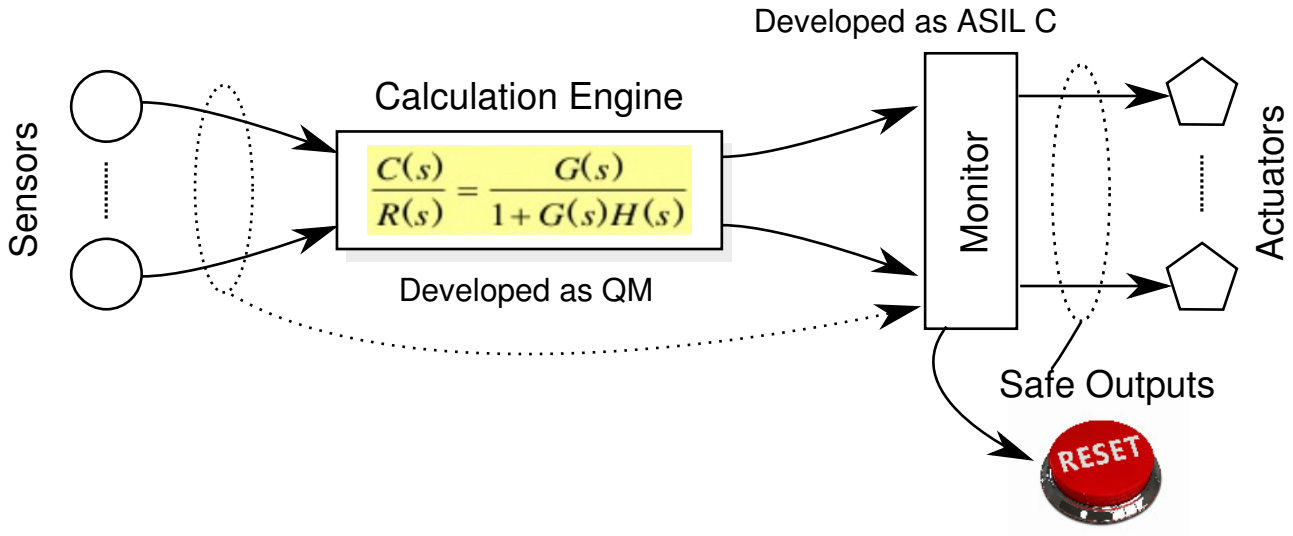
Hardware Lock-Step: an inadequate solution

- The number of replicas is defined by the hardware and cannot be changed: the system cannot be tuned to local conditions.
- It is impossible to design the system around *diverse* implementations.
- Hardware lock-step does not defend against software errors ("Heisenbugs").

  If one replica hits a Heisenbug, then all replicas hit it at the same time.

Reference Architectures



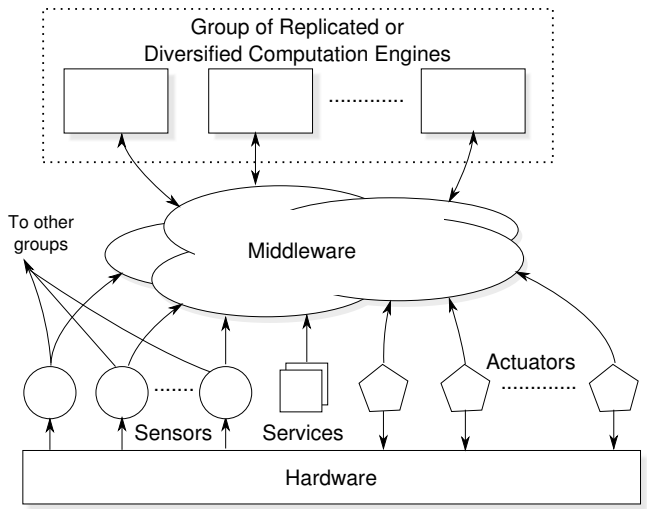


The Road Towards a Solution?

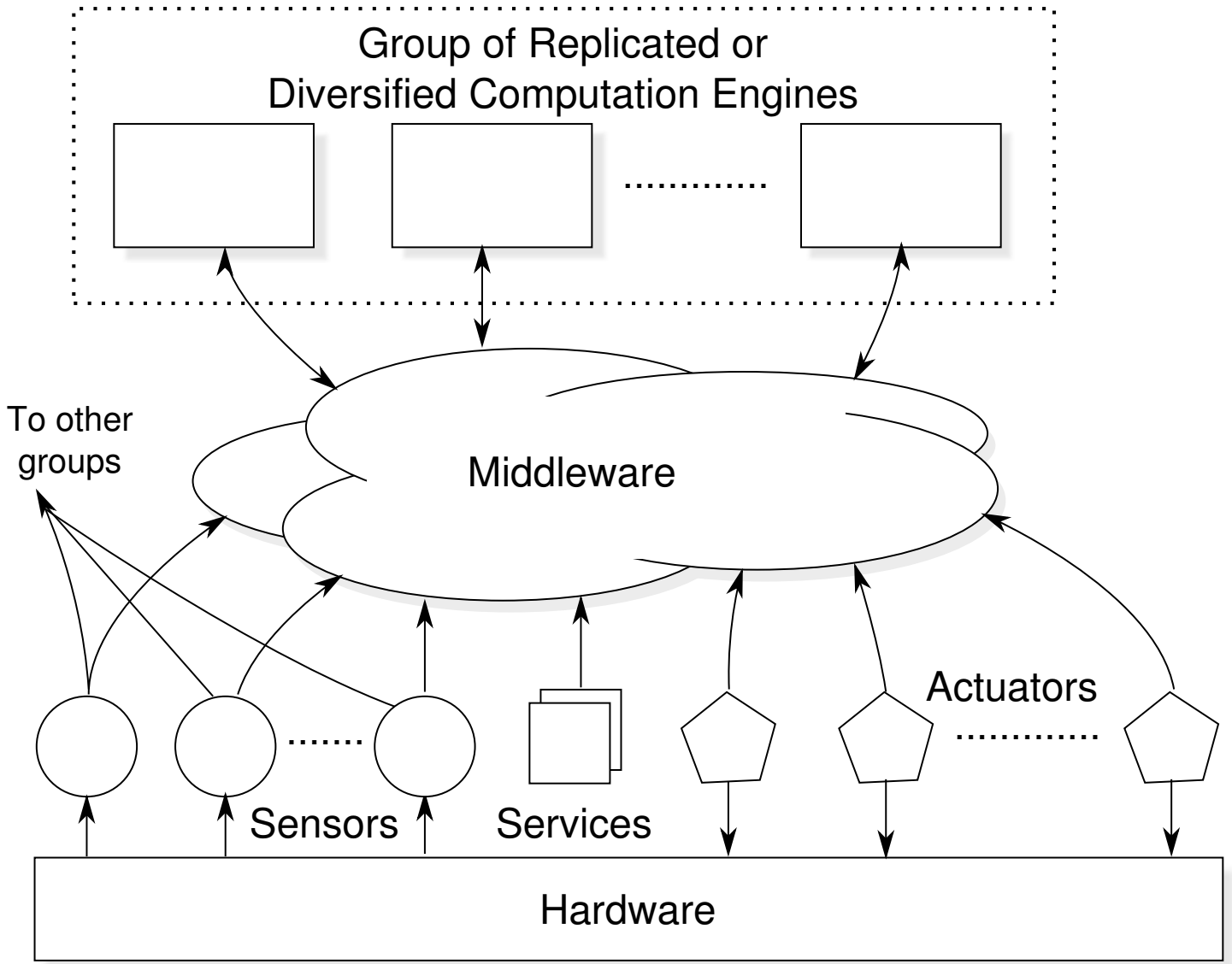
We would like to detect and handle errors, whether software or hardware, that affect the safety of our system.

Since the early 1990s, the data storage industry has been using "Virtual Synchrony" to solve similar mission-critical problems.

Does Virtual Synchrony also work in the embedded world?



Virtual Synchrony?



How does Virtual Synchrony Appear?

Ken Birman (1987):

> *In a virtually synchronous environment, routines . . . will behave as if distributed actions were performed instantaneously and in* ***lock-step.*** *. . . It will appear to any observer – any process using the system – that all processes observed the same events in the same order.*

I.e., Virtual Synchrony, when viewed by an external observer, appears to provide lock-step operation.

But this *virtual* lock-step does not suffer from the disadvantages listed above for hardware lock-step.

QNX calls this "Loosely-Coupled Lock-Step" (LCLS) operation.

System-Tuning

Availability

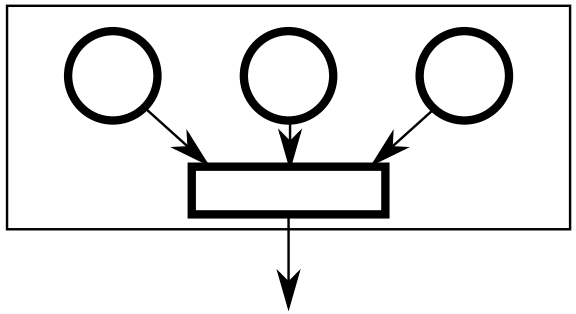
Reliability

It is more important to get an answer, even if it is slightly wrong.

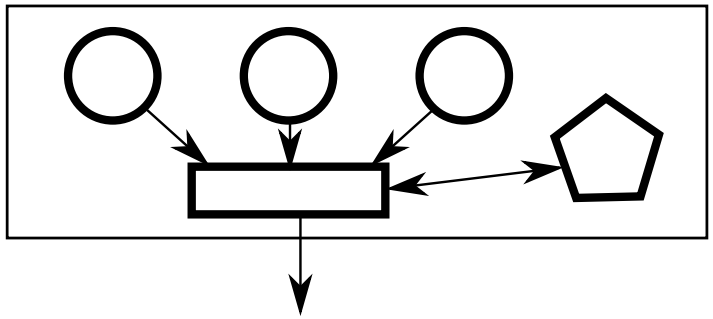
No answer is better than an incorrect answer

These are antagonistic

Dynamic LCLS



2oo3



3oo3 with Monitor

Availability ⟷ Reliability

On the Motorway

In the Town Centre

LCLS Implementation

Goals:

- As few changes to the application as possible.
- Static or dynamic changes to the balance between availability and reliability.
- Independence of the server and dependability implementations.

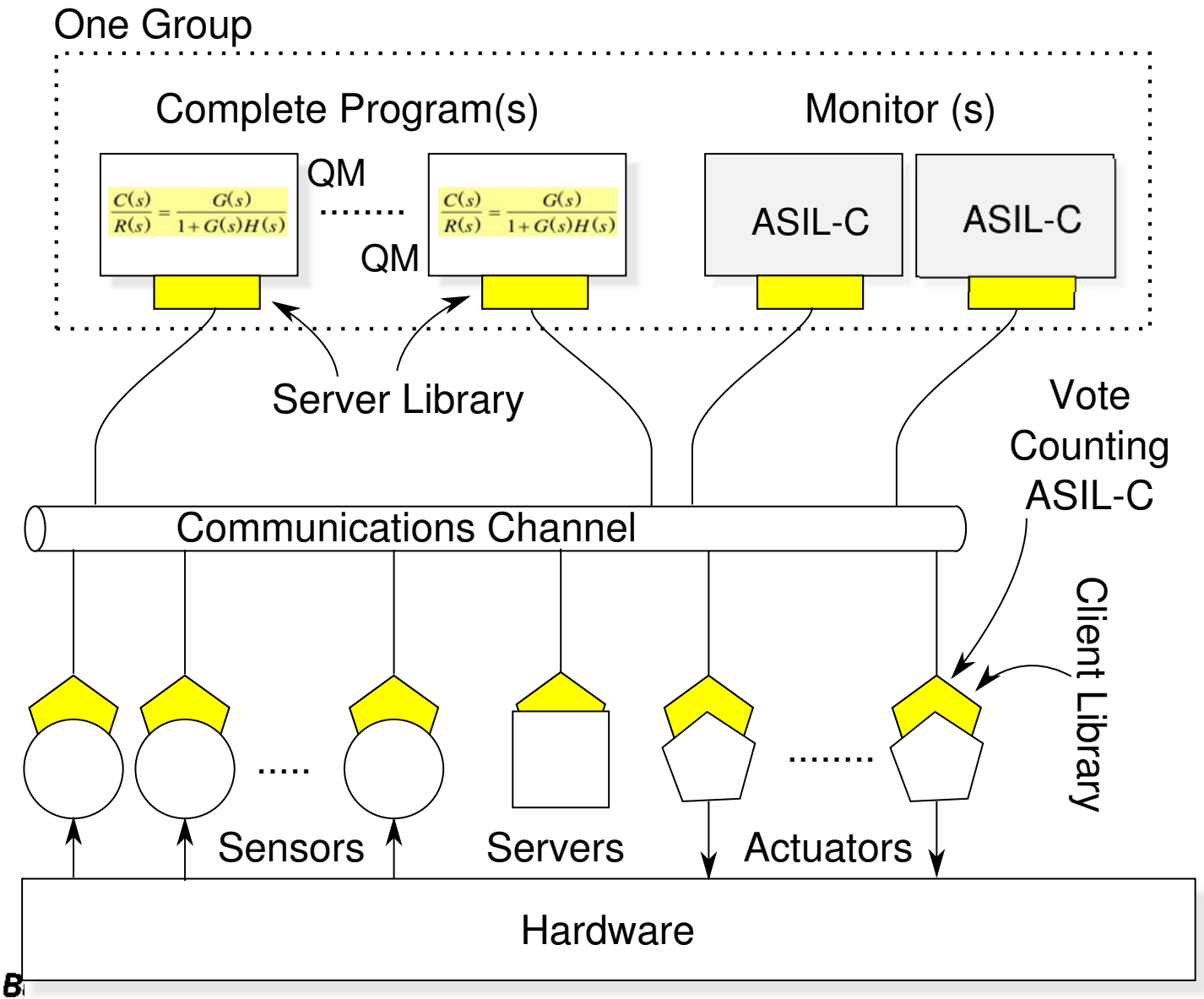
LCLS Implementation

Goals:

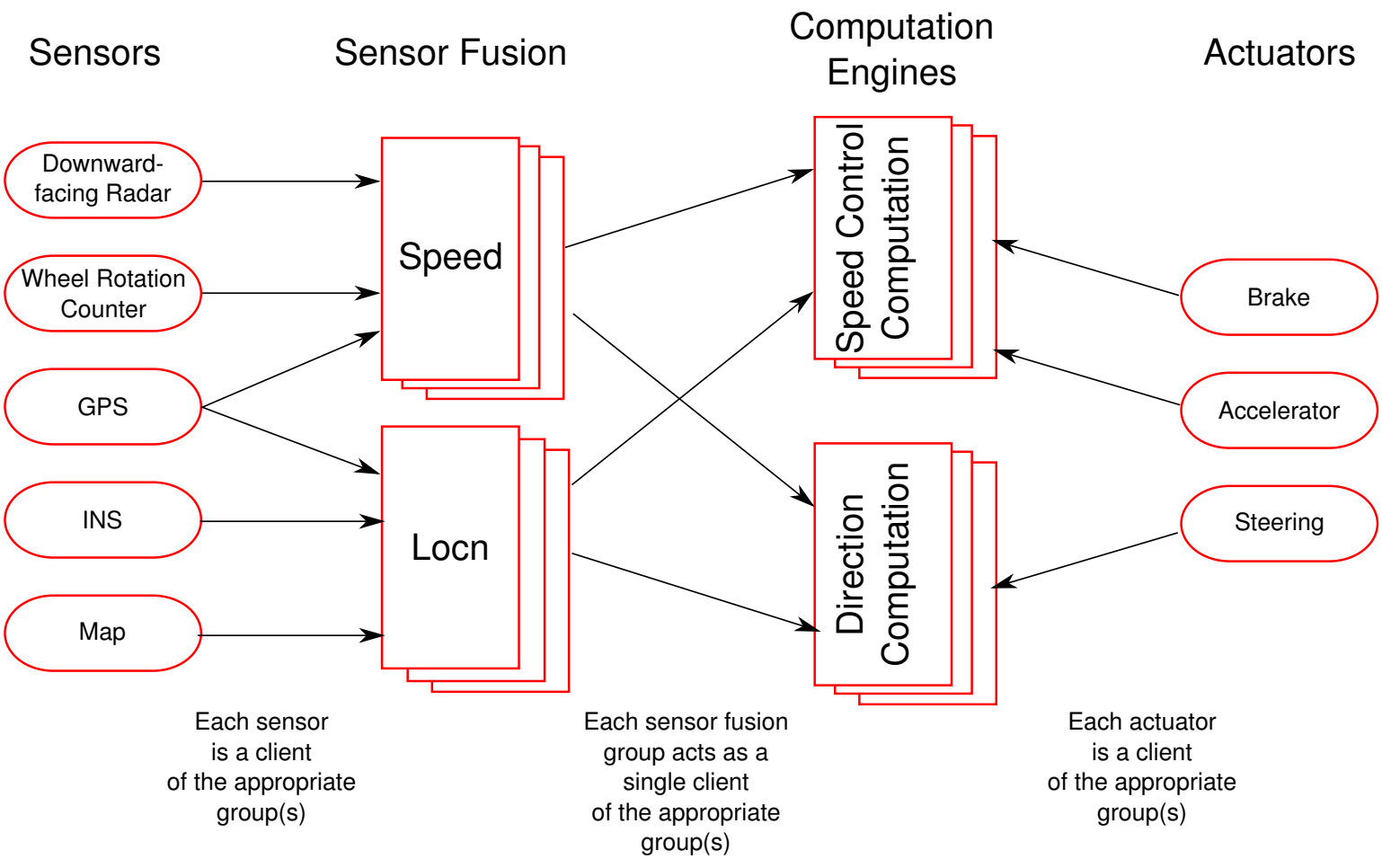
- As few changes to the application as possible.
- Static or dynamic changes to the balance between availability and reliability.
- Independence of the server and dependability implementations.

The number of instances of the Server can be selected dynamically. Because the lock-step is only *virtual*, it is extremely unlikely that a random error will hit two or more instances simultaneously.

LCLS: Configuration 1 (of many)



An Automotive Configuration



Summary

All software is subject to random hardware **and software** errors.

The most important question is not whether such errors have occurred, but whether they have affected the safety of our system.

The LCLS architecture, based on proven Virtual Synchrony algorithms, addresses this important question.

Because LCLS only provides "virtual" synchrony, it can defend against both hardware **and software** errors.

LCLS cleanly divides the two areas of expertise required to build a safe embedded system: the mathematical functionality and the statistical balance of availability and reliability.

Many Thanks

Chris Hobbs
QNX Software Systems
chobbs@qnx.com