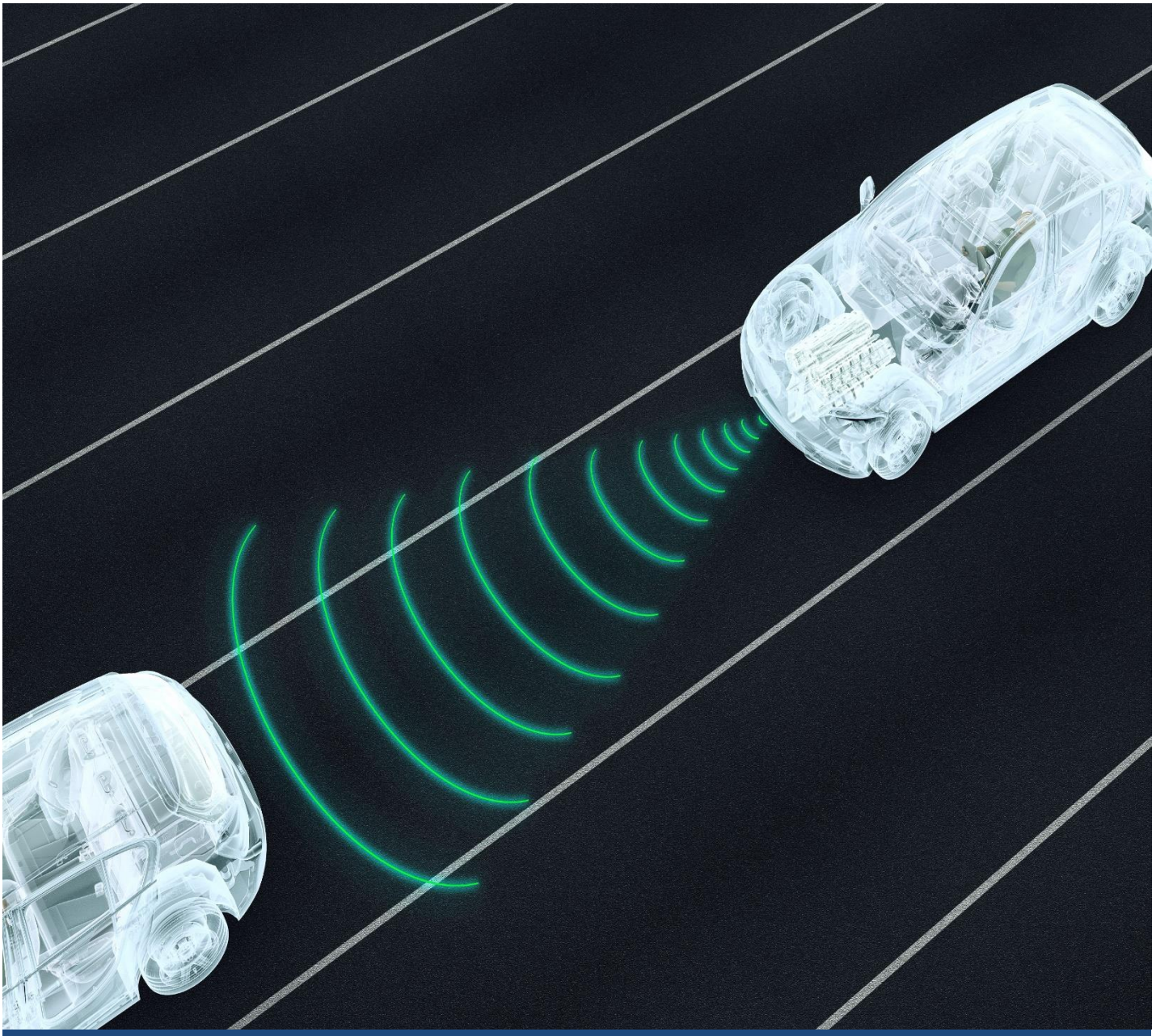

BlackBerry | QNX

CONCEPTUAL WHITEPAPER

# Automotive Functional Safety: No Hiding Place

Kerry Johnson
Chris Hobbs

BlackBerry QNX

5/2/18

BlackBerry | QNX

# Automotive Functional Safety: No Hiding Place

*Abstract: A modern vehicle has over 100 million lines of code, that's more than a Space Shuttle and the Large Hadron Collider. To make the necessary driving decisions, autonomous vehicles use sophisticated software running on the most powerful CPUs ever seen in the automotive industry. Since these processors are pushing the edge of performance and are prone to occasional failures, reliability has become a concern. In addition, the role that software is being asked to take can lead to problems caused by subtle software interactions. The occurrence of such hardware and software errors in an autonomous driving system can impact the system's safety. So how can we mitigate the impact of these errors?*

Advancements in automated driving are fueling the demand for the most powerful silicon on chips (SOCs) ever used in the automotive industry. To meet this demand, semiconductor manufacturers are pushing technology to the point where, for the first time in history, hardware is becoming less reliable. This problem arises from two major factors – physics and complexity.

On the physics front, SOCs run at faster clock speeds, producing more heat, and use smaller transistors, which can now be measured in number of atoms. Heat causes accelerated wear-out; the hotter the part operates, the sooner it fails. Die shrink leads to transistors that are extremely susceptible to faults caused by electromagnetic interference, the impact of alpha particles and neutrons, and cross-talk between neighboring cells. These problems also occur in Dynamic Random-Access Memory (DRAM) systems – in modern multi-gigabyte DRAMs, bit errors can be expected on the order of one per hour.

On the complexity front, manufacturers have been adding more and more inter-related functionality to each SOC. Unfortunately, SOCs ship with bugs, many of which are found only after the chip goes into production; known bugs are documented in the manufacturer's errata sheets. These bugs can affect computations and give erroneous results, thereby causing safety vulnerabilities. The probability of such errors directly impacts the ISO 26262 Automotive Safety Integrity Level (ASIL) rating.

## Verification

To detect and recover from these errors, system designers must implement compensation mechanisms. In one approach, the system performs each computation two or more times and then compares the results. Some microcontrollers implement a technique known as hardware lockstep, in which two processing elements within the SOC execute the same instructions at the same time, with dedicated hardware comparing the results. If the hardware detects a mismatch, an independent diagnostic routine determines which processor was faulty, and system software then takes remedial action. Unfortunately, this technique generally only supports replicas rather than diverse implementations. It also can't detect software bugs – both processing elements will "correctly" execute the buggy code – and it does not scale because the number of replicas is fixed by the hardware. Also, it isn't practical for today's high-performance hardware, where there are far too many internal states for a hardware checker to analyze.

In practice, a system can use software to verify the operation of the hardware. The developer implements a replicated copy of the software, and the two or more replicas are used to perform the verification. Each replica performs safety-critical computations – for instance, "given these

conditions, can acceleration be applied?" – and some middleware runs the computations with synchronization points invisible to the application.

Each replication scheme has its advantages and disadvantages. In the identical replica model, two identical computations running on different threads using different memory will yield the same correct result, except when a transient hardware or random software error occurs. In that case, the error will affect only one of the instances, not both. If there are only two replicas, then some form of error recovery action is required since it is not possible to determine which is correct. If there are more than two replicas, then a majority logic vote result can be used.

Of course, while this approach can detect and compensate for Heisenbugs in the software, it cannot correct for Bohrbugs in the software. Heisenbugs are defined as bugs that may or may not cause a fault for a given operation, while Bohrbugs are bugs that always cause a failure when a particular operation is performed.[1] To do so, a system could use fraternal replicas, which perform the same computations, but using different algorithms. If these replicas come to the same conclusion – for instance, both agree that acceleration can be applied – there is greater overall confidence that the result is correct.

## Implementing replication

A system designer can implement the replication schemes transparently, using middleware that is interposed into the communications path between components. In a microkernel OS, where all software components communicate with each other through message passing, the designer can take advantage of naturally occurring synchronization points that make it easy to interpose the middleware and check subsystem operation.

To ensure that the service is reliable and available, a replica-based system uses multiple instances of the server (see Figure 1). The role of the middleware is to ensure that messages from clients are delivered to all server instances in exactly the same order. From the client's perspective, there appears to be only one server; from the server's perspective, it believes it is executing alone. The middleware replicates the messages from the clients and distributes them to each server instance. The middleware then receives the responses from each server and compares the results to ensure the servers agree within permitted tolerances. Application developers do not need to concern themselves with replication or diversity schemes.

[1] https://www.cs.rutgers.edu/~rmartin/teaching/spring03/cs553/papers01/06.pdf

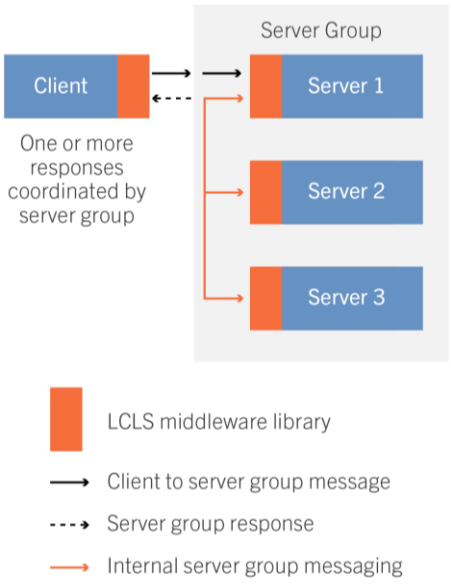


Figure 1: Loosely Coupled Lockstep (LCLS) middleware replicates the client messages that trigger a safety critical computation to each server in a server group. The LCLS middleware guarantees that all messages are delivered to all servers in the same order. The LCLS middleware also offers services to synchronize server state when new servers join a group.

Replication points, that are places to insert the middleware and support multiple replicas, must occur at the right level of granularity. Duplicating every mathematical computation or function call is too fine-grained, and results in higher development costs and slower runtime performance. The ideal replication granularity is implemented at the component level. In fact, the implementation of POSIX API functions in a microkernel OS serves as a good model for this approach.

Consider an application that opens a file and reads some data. Decoupling the application from the file system provides an excellent insertion point for the middleware.

If, instead of talking to the file system, the application talks to the middleware and the middleware talks to replicated file system servers, redundancy and checking can occur at that natural decoupling point, in a manner completely transparent to the application itself, and to the file system implementation. By designing other components within the system at a similar level of granularity, the system designer can insert replicas at the process-to-process communication boundaries. The flexibility offered by the OS is important here: if the OS allows replicas or diverse applications to be seamlessly partitioned across processor cores, the implementation of a replication strategy becomes easier.

## Server models

As mentioned, the system designer can choose between two main server models – identical and fraternal, with fraternal being further categorized as peer fraternal and monitor fraternal.

In the identical model, the servers can run on the same SOC, on different cores (processing elements) in a multi-core system or even on different systems connected by a network. This model offers a measure of scalability and potential redundancy in case of hardware failure: the same software must produce the same results; if it doesn't, then it's a hardware or software Heisenbug

problem. For example, as seen in Figure 2 below, two servers can compute the intended level of throttle and if both server instances agree within a specific tolerance, the operation can be considered safe.

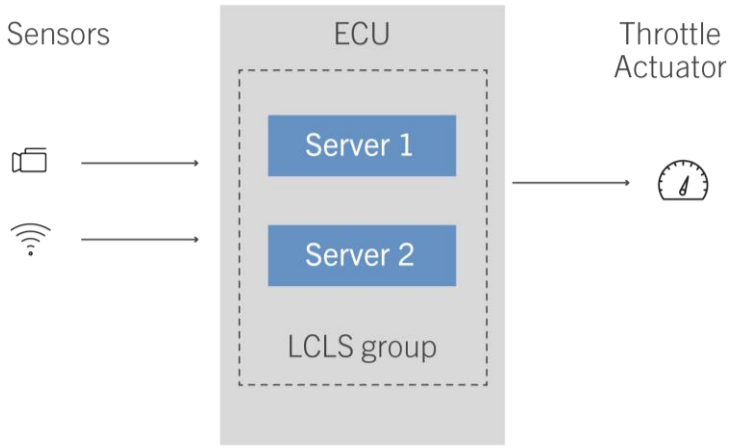


Figure 2: Identical replication model where the same computation is performed by Server 1 and Server 2.

In the peer fraternal model, the system uses diverse but fully functional versions of the servers; for example, the same source code compiled by different compilers. The expectation is that any failures would be in the implementation domain, effectively bugs in the implementation.

The monitor fraternal model also uses diverse servers, but some server instances have full functionality and the others have reduced monitor functionality.

While all three models are useful and can be intermixed, the monitor fraternal model offers an interesting cost savings. Consider the certification costs of each model:

- The identical model has no diversification, so the full development and certification process and cost for the ASIL rating must be borne by the software.
- The peer fraternal model uses diversification, so the overall combination of the multiple diverse instances contributes to both reliability and availability, but the software cost is double or more, depending on the number of diversities.
- The monitor fraternal model has successfully been used in other industries. The concept of a monitor is also known as a safety bag, a much simpler piece of software that ensures that the overall decisions being made are sane and safe. From a certification cost point of view, ISO 26262 ASIL decomposition can be used when certifying the main server software and the much simpler monitor, with less certification effort. The monitor effectively provides ASIL enrichment because it is another diverse instance.

The architecture described provides a very flexible and dynamic way of detecting random errors that have affected the safety of a system. The underlying principle is that of the strong ordering of events to all members of a server group. This provides software virtual lockstep that does not suffer from the limitations of hardware lockstep. As server instances may join and leave groups dynamically, the level of resilience can be tuned to the environment within which the system is operating. For example, when an automobile is driving on a highway, a different level of resilience may be required from when it is driving in a city center.

## Summary

Hardware lock step has been used in other applications to detect faulty CPU operation but this technique is not practical for today's high-performance hardware, where there are far too many internal states for a hardware checker to analyze in real time.

Hardware diagnostics on its own is not enough to detect all these errors. When paired with realtime software checking an efficient and complete means of verifying the system operation can be achieved.

About BlackBerry QNX

BlackBerry QNX was founded in 1980 and is a leading vendor of operating systems, development tools, and professional services for connected embedded systems. Global leaders such as Audi, Siemens, General Electric, Cisco, and Lockheed Martin depend on BlackBerry QNX technology for their in-car electronics, medical devices, industrial automation systems, network routers, and other mission- or life-critical applications. Visit www.qnx.com and follow @QNX_News on Twitter. For more information on the company's automotive work, visit qnxauto.blogspot.com and follow @QNX_Auto.

www.qnx.com