COMP9242 – Exam

November 9, 2010

Paper 1

CuriOS: Improving Reliability through Operating System Structure

Summary

This paper presents a system called *CuriOS*, designed for recovering from OS server faults and errors. When discussing OS servers the authors indicate they are using the microkernel concept of traditional OS services outside of the kernel. The paper is motivated by the fact that after an error occurs, simply restarting a microkernel service can result in loss of state that will have an adverse effect on all clients of the service. As inherent in the paper title, the system attempts to restructure common microkernel design in order to present a robust system that can recover from OS server failure transparently to clients. Even if a request by one client cannot be recovered, other clients to the server should not be affected by the error.

CuriOS wraps around a minimal kernel called *CuiK* which handles all memory permission management. Fault tolerance is achieved by implementing user-level servers, called *protection objects*, that are accessed by a kernel interface of *protected method calls*. Under CuriOS, an OS Service is the combination of kernel, protected object and protected method calls. Protected objects execute in unprivileged mode and have their own private heap. They also have their own private stack per thread that accesses the protection domain. When an error is detected within a protection domain an exception is thrown, which is caught by CuriOS. CuriOS will then restart the protected object in-place and re-call any methods that were being executed, so that the failure is nearly transparent to any clients. Clients are suspended whilst the recovery occurs, so time differences can be noticed. Additionally any operations that may start a mechanical process (such as printing) will also be noticeable.

To avoid loss of state, CuriOS maintains any client state required for the OS service to restart the method in a portion of the client's memory referred to as a *Server State Region*(SSR). The client cannot access the SSR. When a protection domain is accessed, an SSR is created in client memory and mapped in virtual memory to the protection domain. Once the method is complete, the SSR is unmapped and removed. SSRs are not required for stateless server operations. It is possible that an SSR is corrupted during an error, but the authors argue that this will only have an effect on the client that caused the erroneous operation and other clients will not perceive an error. To detect SSR corruption, magic numbers and consistency heuristics are present.

The types of errors recoverable by CuriOS include infinite loops and processor signaled errors. Functional errors such as deadlock or incorrect behaviour are not considered.

Several servers are implemented and tested for CuriOS, including a scheduler, timer, file system, LWIP networking stack and serial port driver. Test are conducted using fault injection via bit flipping and instruction injection to cause memory faults.

The paper takes the approach of fault recovery rather than assuring correctness of important drivers initially, with the implicit assumption that OS services will never be fault free. Additionally, it assumes that service interruption is due to error that requires restart, not correct server behaviour or non-terminal errors.

Successes

CuriOS successfully implements server restarts that are effectively transparent to the client. Faults particular to one client do not have an effect on other clients and the system recovers from most of the tested injected faults.

Limitations

Applications need to be altered significantly to use the CuriOS application model.

OS services need to be substantially altered to deal with SSRs and to be able to recover from any uncorrupted state of an SSR. Changes are still required for non-SSR services, like those made to the ext2 filesystem server, however the details of these changes are not made clear in the paper.

The overheads of the protection domains and SSRs are very high, without including the overheads of restarting servers and checking SSRs for corruption. According to the paper, making a call to a protection domain without SSR is comparable to the CuriOS performing two context switches. However from the stated data a call into a protection domain has a 32% overhead when compared to the cost of two context switches. In addition, when dealing with servers that use SSRs, there is increased overhead on calls to the protected object. The paper does not make clear the significance of this overhead, especially when considering that half of their servers user SSRs. From the measurements presented, the overhead of a protected object call with SSR is 206% in terms of instructions and 94% in terms of time. As CuriOS does not appear to be a particularly fast microkernel makes the overhead incredibly high.

The paper states that CuriOS takes 148 microseconds for two context switches. This value seems quite high, especially compared to OKL4s 151 clock cycle IPC (on ARM926, which is the same CPU as found on the OMAP1610) which is analogous to a CuriOS context switch. With a 96MHz clock speed an equivalent OKL4 round trip IPC would be $2 * 151/96000000 \approx 3$ microseconds.

In the case that a method fails after repeated protection domain restarts, an error is returned to the client indicating that the operation is not possible. This has the implication that any calls to an OS server must handle possible errors in spite of CuriOS' reliability model, although likelihood should be greatly reduced.

Criticisms

One of the requirements established by the paper for an OS to transparently recover in the event of failure is that the system should have *transparency of addressing*, such that the service can be accessed by the same reference. The authors note that using a name server or similar identifying technique would be sufficient to mitigate the requirement, avoiding the need to reload failed servers into the same memory addresses.

CuriOS states that is being developed as a reliable OS designed for embedded systems, yet the recovery functionality has high overheads in memory and performance with no recognition of the overheads or faster systems. The related work section ignores direct competitors, as the latest L4 paper referenced is from 2005. Additionally, the paper refers to L4 as the L4 of the original Liedtke paper, instead of referring to L4Ka::Pistachio which it appears they are actually talking about. As a result the analysis of L4 appears to generalise the entire family of microkernels to one set of results.

The measurements section lacks any performance comparison at all, and overall system performance is avoided. The only metric provided is how successful the system is at recovering from the faults it is designed to

recover from and how long it takes to get in and out of a protection domain. Whilst fault injection experiments are documented, the micro benchmarks used to measure calls into the projection domain are lacking and not reproducible, whilst also giving no indication of overall performance. Instruction counts from QEMU are misleading as they measure the number of instructions executed rather than the number of clock cycles taken to execute those instructions.

Fault injection is used by categorising the faults to be general faults in servers or devices, but there is no indication of how this system, with its high overheads, would perform under real loads or how the injected faults cover the broard range of real faults.

Conclusion

CuriOS presents a successful system for restarting faulting OS services. However the system presents significant overheads that would appear to discourage its use in real world systems in favour of other methods like device driver synthesis or static/dynamic analysis of drivers and servers. The poor performance of the microkernel appears to reinforce the bad name given to user-level device drivers in general. It is possible that the overheads are less significant than they appear, but appropriate benchmarks and assessments are not present in the paper.