COMPSCI 630 Systems **Spring 2017**

Lecture 15: Fault Tolerance

Lecturer: Emery Berger *Scribe(s): Rohith Pesala, Shruthi Yashavanth*

15.1 Fault

The definition of fault is ”violation of a specification”, an unintended behavior. These specifications can be formal, informal or latent.

Formal These specifications are very strict and complete. They an be in terms of First-order logic or higher-order logic. These specifications are hard to formulate but they are concrete and complete with proofs if necessary. An example of such specification would be the Fibonacci series

$$fib(n) = \begin{cases} 1 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & n > 1 \end{cases}$$

Informal Informal specifications are mostly verbal specifications that are generic. These specifications are very high level and involve a lot of ambiguity.

Latent These specifications are somewhere in between formal and informal. These can be more of implementation details asking for specific components in a system.

Examples If a user requests for a webpage and it returns with a ”404” error or if it closes the browser. Any such crashes or unintended behaviors are called faults.

15.2 Verification

To verify that a systems works properly, there are multiple methods to prove it’s correctness. The two main methods are Formal/Total verification and Partial correctness.

Formal Verification We can prove a system’s correctness in using this method when we have the underlying specification that is complete and can deal with all the states of system. Then, we compare our system’s output against the specification with same input. This is only theoretically feasible because if we had the underlying specification, then we would use that itself.

Partial Correctness In this method, we try to avoid known problems and faults that generally occur such as buffer overflows and null dereferences. This is practically feasible because we can aim to avoid the well known faults making system as robust as we can make it.

15.3 Types

There are many types of faults that we encounter while developing a system. Generally, we try to avoid crashes, buffer overflows, null dereferences, etc., Let's consider the case of web browsers. Some of the faults that can occur are overload (too many clients/users that the server can't handle), Timeline (service depends on when the request is sent), Server (Depends on which server the request is routed to). We can check for divergence in browsers by blurring and down sampling and comparing. One of the standard approaches to such transient failures is to retry/refresh the request. This is an idempotent action. An idempotent action, f, is defined as $f(x) = f(f(x))$.

15.4 Fault tolerant systems

15.4.1 Reliability vs Availability

Fault tolerance of a system can be measured in terms of reliability and availability. A reliable systems always produces the correct result but an available system does something as long as it can. Reliable systems are necessary in processes like transactions and where the result needs to be absolutely correct. Available systems are useful when the system just needs to keep on running for example a telephone line. It is acceptable to have some noise in between packets of sounds rather than waiting for the correct packet to arrive.

15.4.2 Tandem

At tandem, they focus heavily on fault tolerance and they produce a Non stop system that is highly fault tolerant. The main aspects of this system is to modularity and fault isolation. They also claim that most bugs that occur in practice are Heisenbugs and that they can be resolved by a restart. The most crucial part of their system is using process pairs where they run the same input on different systems with certain type of message passing and when one fails, the other can continue processing and output the correct result. This system is actually inspired from the concept of N-version programming.

15.4.3 N-version Programming

In this method, we have n different implementations of the same system and we run all of them simultaneously. If one of the system diverges from the others, we consider it to be faulty and we discard that implementation. This method is reliable when all these different implementations are independent of each other but in practice that is not the case. The dependence is inherent because they all are given the same details of what to implement and whatever bugs occur are in the edge cases where it is tough to program and all the implementations have a high probability of failing in those areas just because it is hard to implement. There's two ways to deal if we see a system disagrees with the majority. They are fail-stop (kill the system as soon as you see it's faulty on one input) and fail-safe(ignore the system that disagrees with the majority for that input and let it run for the next inputs). If we employ a fail-stop system, this method can only deal with n-1 faults.

15.4.4 DieHard

In this paper, a mechanism and a system are introduced to avoid memory errors in unsafe languages such as C and C++. These languages are susceptible to memory errors such as dangling pointers (freeing of live objects), buffer overflows (out-of-bound writes corrupting contents of live objects), heap metadata overwrites (when heap metadata is closer to heap, overwrites can occur), uninitialized reads (reads to newly allocated or un-allocated memory), invalid frees (illegal addresses being freed), and double frees (repeated calls to free objects).

DieHard has two modes of operation - stand-alone, which avoids many memory errors, and replicated, where two replicas of the program are executed and the outputs are compared to check the correctness in terms of avoiding memory errors. The stand-alone mode replaces the memory allocator.

DieHard, introduces the concept of 'probabilistic memory safety' (probabilistic guarantee to mitigate memory errors) and presents a runtime system that provides this. Allocation of the memory is randomized across the entire heap. This reduces Buffer overflow errors and reduces the chances of dangling pointers.

Tests using diehard with real and injected errors show that in the vast majority of cases, programs running with DieHard are tolerant to faults completely and run to completion without crashes that occur 100% of the time outside of the Diehard memory management system.