Fault Tolerance Philosophies and Strategies

Introduction

There are two fundamentally different philosophies we could adopt with regard to building a fault tolerant system. This document attempts to describe them along with their benefits and disadvantages. Each philosophy entails a set of strategies used to implement fault tolerance; some of those strategies are outlined along with their consequences for the design of the system.

It should be noted up front that the LSST processing systems will need to deal with failures as a common situation. As a back-of-the-envelope estimate, if we have 10,000 components (likely a low estimate), each of which has a (very conservative) 10 year mean time before failure (MTBF) with a uniform probability of failure in each time interval, the probability of a component failure somewhere in the system in any given hour is 0.076, and there is a 99% chance of a failure within 58 hours.

Philosophy 1: Assume Unreliability

We can assume that all components of the system are inherently unreliable and that failure is a normal occurrence. This seems to be Google's worldview. This implies that the system must be highly dynamic and self-organizing. Strategies used by this philosophy would include dynamic allocation of tasks to hardware nodes, including dynamic load balancing, and dynamic discovery of communication partners.

A design using this philosophy is likely to be more tolerant of all types of failures, including hardware, software, network, and even design failures. On the other hand, it is also likely to be less efficient in resource usage, particularly in the case when no failures occur. Forcing the system to be self-organizing may restrict the range of algorithms that can be easily implemented. Monitoring a highly dynamic system may be more complex.

Philosophy 2: Detect Failure and Correct

We can assume that all required components of the system are typically operational. The resulting system could be organized in a more static fashion. Strategies used by this philosophy would include detecting when failures occur and recovering from those failures.

A design using this philosophy is likely to be more efficient in the no-failure case. Its success is dependent on properly handling all failure modes. While a wider range of algorithms may be possible, they may still need to be implemented in a way that allows recovery and restart.

Commonalities

In both philosophies, processing will need to be broken down into granular tasks that can be restarted in the event of a failure. The level of granularity affects system downtime or unavailability (coarse-grain tasks increase this), the complexity of the management system (fine-grain tasks increase this), and the need for checkpointing (fine-grain tasks increase this). Overall system efficiency involves a trade-off between the overhead of fine-grain tasks and the wasted processing of coarse-grain tasks that need to be restarted.

Combinations

It is possible to combine the two philosophies. For example, a master controller using a failure-detection strategy could be used to dispatch tasks to a self-organizing pool of slaves.