Health Monitoring and Recovery

Introduction

Suppose that a system seems to be wedged ... it has not recently made any progress. How would we determine if the system was deadlocked?

- identify all of the blocked processes.
- identify the resource on which each process is blocked.
- identify the owner of each blocking resource.
- determine whether or not the implied graph contains any loops.

How would we determine that the system might be wedged, so that we could invoke deadlock analysis? It may not be possible to identify the owners of all of the involved resources, or even all of the resources. Worse still, a process may not actually be blocked, but merely waiting for a message or event (that has, for some reason, not yet been sent). And if we did determine that a deadlock existed, what would we do? Kill a random process? This might break the circular dependency, but would the system continue to function properly after such an action? Formal deadlock detection in real systems is:

Formal deadlock detection in real systems ...

a. is difficult to perform
b. is inadequate to diagnose most hangs
c. does not tell us how to fix the problem

Fortunately there is a simpler technique, that is far more effective at detecting, diagnosing, and repairing a much wider range of problems: health monitoring and managed recovery.

Health Monitoring

We said that we could invoke deadlock detection whenever we thought that the system might not be making progress. How could we know whether or not the system was making progress? There are many ways to do this:

- by having an internal monitoring agent watch message traffic or a transaction log to determine whether or not work is continuing
- by having the service send periodic heart-beat messages to a health monitoring service.
- by having an external health monitoring service send periodic test requests to the service that is being monitored, and ascertain that they are being responded to correctly and in a timely fashion.

Any of these techiques could alert us of a potential deadlock, livelock, loop, or a wide range of other failures. But each of these techniques has different strengths and weaknesses:

- heart beat messages can only tell us that the node and application are still up and running. They cannot tell us if the application is actually serving requests.
- an external health monitoring service can determine whether or not the monitored application is responding to requests. But this does not mean that some other requests have not been deadlocked or otherwise wedged.
- an internal monitoring agent might be able to monitor logs or statistics to determine that the service is processing requests at a reasonable rate (and perhaps even that no requests have been waiting too long). But if the internal monitoring agent fails, it may not be able to detect and report errors.

Many systems use a combination of these methods:

- the first line of defense is an internal monitoring agent that closely watches key applications to detect failures and hangs.
- if the internal monitoring agent is responsible for sending heart-beats (or health status reports) to a central monitoring agent, a failure of the internal monitoring agent will be noticed by the central monitoring agent.
- an external test service that periodically generates test transactions provides an independent assessment that might include external factors (e.g. switches, load balancers, network connectivity) that would not be tested by the internal and central monitoring services.

Managed Recovery

Suppose that some or all of these monitoring mechanisms determine that a service has hung or failed. What can we do about it? Highly available services must be designed for restart, recovery, and fail-over:

- The software should be designed so that any process in the system can be killed and restarted at any time. When a process restarts, it should be able to reestablish communiction with the other processes and resume working with minimal disruption.
- The software should be designed to support multiple levels of restart. Examples might be:
  - warm-start ... restore the last saved state (from a database or from information obtained from other processes) and resume service where we left off.
  - cold-start ... ignore any saved state (which may be corrupted) and restart new operations from scratch.
- The software might also designed for progressively escalating scope of restarts:
  - restart only a single process, and expect it to resync with the other processes when it comes back up.
  - restart all of the software on a single node.
  - restart a group of nodes, or the entire system.

Designing software in this way gives us the opportunity to begin with minimal disruption, restarting only the process that seems to have failed. In most cases this will solve the problem, but perhaps:

- process A failed as a result of an incorrect request received from process B.
- the operation that caused process A to fail is still listed in the database, and when process A restarts, it may attempt to re-try the same operation and fail agian.
- the operation that caused process A to fail may have been mirrored to other systems, that will also experience or cause additional failures.

For all of these reasons it is desirable to have the ability to escalate to progressively more complete restarts of a progressively wider range of components.

False Reports

Ideally a problem will be found by the internal monitoring agent on the affected node, which will automatically trigger a restart of the affected software on that node. Such prompt local action has the potential to fix the problem before other nodes even notice that there was a problem.

But suppose a central monitoring service notes that it has not received a heart-beat from process A. What might this mean?

- It might mean that the node has failed.
- It might mean that the process has failed.
- It might mean that the system is loaded and the heart-beat message was delayed.
- It might mean that a network error prevented or delayed the delivery of a heart-beat message.

Declaring a process to have failed can potentially be a very expensive operation. It might cause the cancellation and retransmission of all requests that had been sent to the failed process or node. It might cause other servers to start trying to recover work-in-progress from the failed process or node. And this recovery might involve a great deal of network traffic and system activity. We don't want to start an expensive fire-drill unless we are pretty certain that a process has failed.

- the best option would be for a failing process to detect its own problem, inform its partners, and shut-down cleanly.
- if the failure is detected by a missing heart-beat, it may be wise to wait until multiple heart-beat messages have been missed before declaring the process to have failed.
- in some cases, we might want to wait for multiple other processes/nodes to complain.

But there is a trade-off here. If we do not take the time to confirm suspected failures, we may suffer unnecessary service disruptions from forcing fail-overs from healthy servers. On the other hand, if we wait too long before initiating fail-overs, we are prolonging the service outage. These so-called "mark-out thresholds" often require a great deal of tuning.

Other Managed Restarts

As we consider failure and restart, there are two other interesting cases to note:

- non-disruptive rolling upgrades ... if a system is capable of operating without some of its nodes, it is possible to achieve non-disruptive rolling software upgrades. We take nodes down, one-at-a-time, upgrade each to a new software release, and then reintegrate them into the service. There are two tricks associated with this:
  - the new software must be up-wards compatible with the old software, so that new nodes can interoperate with old ones.
  - if the rolling upgrade does not seem to be working, there needs to be an automatic *fall-back* option to return to the previous (working) release.
  -
- prophylactic reboots ... It has long been observed that many software systems become slower and more error prone the longer they run. The most common problem is memory leaks, but there are other types of bugs that can cause software systems to degrade over time. The right solution is probably to find and fix the bugs ... but many organizations seem unable to do this. One popular alternative is to automatically restart every system at a regular interval (e.g. a few hours or days).

  If a system can continue operating in the face of node failures, it should be fairly simple to shut-down and restart nodes one at a time, on a regular schedule.