are liable to hit the same problem and result in another timeout. That just makes the user wait even longer for his error message. Most of the time, you should queue the operation and retry it later.

5.2 Circuit Breaker

Not too long ago, when electrical wiring was first being built into houses, many people fell victim to physics. The unfortunates would plug too many appliances into their circuit. Each appliance drew a certain amount of current. When current is resisted, it produces heat proportional to the square of the current times the resistance (I^2R). Because they lacked superconducting home wiring, this hidden coupling between electronic gizmos made the wires in the walls get hot, sometimes hot enough to catch fire. Pfft. No more house.

The fledgling energy industry found a partial solution to the problem of resistive heating, in the form of fuses. The entire purpose of an electrical fuse is to burn up before the house does. It is a component designed to fail first, thereby controlling the overall failure mode. This brilliant device worked well, except for two flaws. First, a fuse is a disposable, one-time use item; therefore, it is possible to run out of them. Second, residential fuses (in the United States) were about the same diameter as copper pennies. Together, these two flaws led many people to conduct experiments with homemade, high-current, low-resistance fuses (that is, a 3/4-inch disk of copper). Pfft. No more house.

Residential fuses have gone the way of the rotary dial telephone. Now, circuit breakers protect overeager gadget hounds from burning their houses down. The principle is the same: detect excess usage, fail first, and open the circuit. More abstractly, the circuit breaker exists to allow one subsystem (an electrical circuit) to fail (excessive current draw, possibly from a short-circuit) without destroying the entire system (the house). Furthermore, once the danger has passed, the circuit breaker can be reset to restore full function to the system.

You can apply the same technique to software by wrapping dangerous operations with a component that can circumvent calls when the system is not healthy. This differs from retries, in that circuit breakers exist to prevent operations rather than reexecute them.

In the normal “closed” state, the circuit breaker executes operations as usual. These can be calls out to another system, or they can be internal operations that are subject to timeout or other execution failure. If the call succeeds, nothing extraordinary happens. If it fails, however, the circuit breaker makes a note of the failure. Once the number of failures (or frequency of failures, in

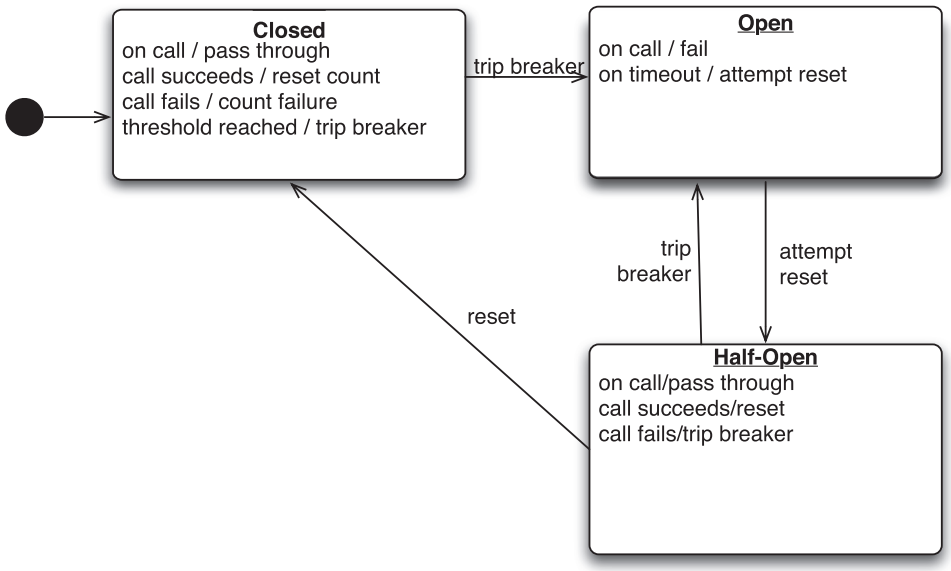

Figure 16—Circuit Breaker State Transitions

more sophisticated cases) exceeds a threshold, the circuit breaker trips and "opens" the circuit, as shown in Figure 16, *Circuit Breaker State Transitions*, on page 94.[3] When the circuit is "open," calls to the circuit breaker fail immediately, without any attempt to execute the real operation. After a suitable amount of time, the circuit breaker decides that the operation has a chance of succeeding, so it goes into the "half-open" state. In this state, the next call to the circuit breaker is allowed to execute the dangerous operation. Should the call succeed, the circuit breaker resets and returns to the "closed" state, ready for more routine operation. If this trial call fails, however, the circuit breaker returns to the "open" state until another timeout elapses.

When the circuit breaker is open, all calls will immediately fail. This should probably be indicated by some type of exception. To provide good feedback to the user, it is useful to throw a different exception when the circuit is open. This allows the calling code to handle this type of exception differently.

Depending on the details of the system, the circuit breaker may track different types of failures separately. For example, you may choose to have a lower threshold for "timeout calling remote system" failures than "connection refused" errors.

3. The Leaky Bucket pattern from *Pattern Languages of Program Design 2 [VCK96]* provides a wonderful implementation for this type of counter.

Circuit breakers are a way to automatically degrade functionality when the system is under stress. This can have an impact on the business of the system. Therefore, it is essential to involve the system's stakeholders when deciding how to handle calls made when the circuit is open. For example, should a retail system accept an order if it cannot confirm availability of the customer's items? What about if it cannot verify the customer's credit card or shipping address? Of course, this conversation is not unique to the use of a circuit breaker, but discussing the circuit breaker can be a more effective way of broaching the topic than asking for a requirements document.

The state of the circuit breakers in a system is important to another set of stakeholders: operations. Changes in a circuit breaker's state should always be logged, and the current state should be exposed for querying and monitoring. (See Chapter 17, *Transparency*, on page 231 for more detail.) In fact, the frequency of state changes is a useful metric to chart over time; it is a leading indicator of problems elsewhere in the enterprise. Likewise, operations needs some way to directly trip or reset the circuit breaker.

Circuit breakers are effective at guarding against integration points, cascading failures, unbalanced capacities, and slow responses. They work so closely with timeouts that they often track timeout failures separately from execution failures.

Remember This

Don't do it if it hurts

Circuit Breaker is the fundamental pattern for protecting your system from all manner of Integration Points problems. When there's a difficulty with Integration Points, stop calling it!

Use together with Timeouts

Circuit Breaker is good at avoiding calls when Integration Points has a problem. The Timeouts pattern indicates that there is a problem in Integration Points.

Expose, track, and report state changes

Popping a Circuit Breaker *always* indicates there is a serious problem. It should be visible to operations.[4] It should be reported, recorded, trended, and correlated.

4. See Chapter 17, *Transparency*, on page 231.

5.3 Bulkheads

In a ship, *bulkheads* are metal partitions that can be sealed to divide the ship into separate, watertight compartments. Once hatches are closed, the bulkhead prevents water from moving from one section to another. In this way, a single penetration of the hull does not irrevocably sink the ship. The bulkhead enforces a principle of damage containment.

You can employ the same technique. By partitioning your systems, you can keep a failure in one part of the system from destroying everything. Physical redundancy is the most common form of bulkheads. If there are four independent servers, then a hardware failure in one can't affect the others. Likewise, if there are two application instances running on a server and one crashes, the other will still be running (unless, of course, the first one crashed because of some external influence that would also affect the second).

Protect critical clients by giving them their own pool to call.

At the largest scale, a mission-critical service might be implemented as several independent farms of servers, with certain farms reserved for use by critical applications and others available for noncritical uses. For example, a ticketing system could provide dedicated servers for customer check-in. These would not be affected if other, shared servers are overwhelmed with "flight status" queries (as sometimes happens during severe weather). Such a partitioning would have allowed the airline in Chapter 2, *Case Study: The Exception That Grounded An Airline*, on page 9 to keep checking passengers at airports, even if channel partners could not look up fares for that day's flights.

In Figure 17, *Hidden Linkages*, on page 97, Foo and Bar both use the enterprise service Baz. Because both depend on a common service, each system has some vulnerability to the other. If Foo suddenly gets crushed under user load, goes rogue because of some defect, or triggers a bug in Baz, Bar—and its users—also suffer. This kind of unseen coupling makes diagnosing problems (particularly performance problems) in Bar very difficult. Scheduling maintenance windows for Baz also requires coordination with both Foo and Bar, and it may be difficult to find a window that works for both clients.

Assuming both Foo and Bar are critical systems with strict SLAs, it'd be safer to partition Baz, as shown in Figure 18, *Partitioned System*, on page 97.

Dedicating some capacity to each critical client removes most of the hidden linkage. They probably still share a database and are, therefore, subject to deadlocks across instances, but that's another antipattern.