Have you tried turning it off and on again?

A software article by Efron Amber Licht

July 2023.

ALL ARTICLES

LICENSE

Feeds

- RSS
- ATOM
- JSON

1. Introduction: “Restart, Reboot, Reinstall”

Whenever, wherever electronics are used, they break. And when they break, the holy cry resounds:

> “Have you tried turning it off and on again?”

And if that didn’t work, the next step is usually:

> “Try unplugging it and plugging it back in.”

The third and final step is usually:

> “Try reinstalling it.”

This holy trinity: “Restart, Reboot, Reinstall” - has a higher success rate than any other debug or repair strategy since the first MOS 6502 rolled off the assembly line in 1975. **They are remarkably universal and effective repair strategies**.

They are so universal and effective that we sometimes don’t think of them as strategies at all.

How many times over the last few days have you had to do one or more of these things? All of these events took place over the last 72 hours:

device	program	problem	restart, reboot, reinstall?	did it work?	time to fix?	total time
amd64/win pc	docker through wsl	wouldn't start	reboot	✅	60s	60s
amd64/win pc	xbox game bar	froze on startup	reinstall	✅	180s	240s
amd64/win pc (windows)	visual studio code	incorrect syntax highlighting	restart	✅	40s	280s
amd64/win pc	steam	wouldn't load game	restart	✅	20s	300s
amd64/win pc	visual studio code	incorrect import cache	restart	✅	40s	340s
amd64/win pc	discord	failed to load recently-joined server	restart	✅	10s	350s
amd64/win pc	wifi adapter	failed to load recently-joined server	reboot	✅	370s	720s
arm64/android phone	youtube	wouldn't load video	restart	✅	30s	750s
arm64/android phone	youtube	wouldn't refresh list of videos	restart	✅	30s	780s
mazda3 infotainment	bluetooth	wouldn't connect to phone	reboot	✅	300s	1080s
mazda3 infotainment	everything	crashed	reboot	✅	300s	1380s
sony ps5	spotify	wouldn't load	reboot	✅	180s	1560s
linksys router	firmware?	wifi down	reboot	✅	300s	1860s

That's a **31 minutes**: a half hour of of my life - spent on "Restart, Reboot, Reinstall" so far this week. By the end of the year, this will be nearly a full day.

I am a computer programmer, so I have to deal with more software than most people, but *no one's life is free of software* nowadays. Everyone is sitting in their car or their office or on their couch, turning software off and on again. And yet, we don't design software with this in mind. We design software as if it will never break, and if it does, it will be a rare and exceptional event. It's *all* buggy. It's **software.**.

Even the longest-lived, most-used, most-loved software has bugs, from sudo to task manager. The linux kernel, one of the most scrutinized and fought-over pieces of

software, has 3709 bugs that *we know about* listed in it's issue tracker as of 2023-07-12. And a surprising amount of these bugs can be mitigated by "Restart, Reboot, Reinstall".

Maybe it's time that we started admitting that our designs will fail, and we should design for failure by making it as easy as possible to get back on the happy path. More concretely:

- software should be quick and easy to install and uninstall
- it should boot as quickly as possible
- hardware should be designed to do a cold boot as quickly as possible
- and should make that cold boot as easy as possible, preferably with a single button press.

We'll get into more details about *how* to do this in a second, but first, a few stories about **turning it off and on again**... in production.

2. telecom: file-handle leaks

While working for a large telecom, we had a couple of dozen servers that were slowly leaking memory and filehandles over a few days. This was a slow process, but after about three days the server would be pretty much useless.

We noticed that after deploys, all of our servers would perform well. That is, **if you turned them on and off again**, it fixed the problem. I wrote a quick script to kill and reboot individual servers while we started tracking down the bug, but after a few weeks, we still hadn't found it.

In a fit of frustration, I wrote a library that would kill and reboot servers at random. It looked more or less like this:

```
 1  package russianroulette
 2
 3  // On a timescale determined by stddev and mean, roll a 6-
    sided die. If the result is 0, call the provided cancel
    function, (usually context.CancelFunc) and return.
 4  func Roulette(stddev, mean time.Duration, cancel func()) {
 5      rng := rand.New(rand.NewSource(time.Now().UnixNano()))
 6      for {
 7          duration :=
    time.Duration(math.Abs(rand.NormFloat64()) * stddev + mean
 8
 9          time.Sleep(duration)
10          if rng.Intn(6) == 0 {
11              log.Printf("russianroulette: bang! triggered
    shutdown")
12              cancel()
13              return
14          }
15          log.Printf("russianroulette: click")
16      }
17  }
```

(it wasn't actually called `russianroulette`: the boss shut that one down).

The context being cancelled would trigger a graceful shutdown of the server, which would restart itself. We tweaked the numbers to have each server restart itself roughly every two days (using the normal distribution so that servers wouldn't 'sync up' and all restart at the same time and drop traffic).

It was **kind** of a joke, but we figured what the hell, we'd try it. It solved the problem. *permanently*. We never figured out what was causing the leaks, but we never had to. It turned out our servers just worked better when we **turned them off and on again** every so often.

The practice spread to more servers as various leaks were discovered, and before we knew it most of our services had some variant of `russianroulette` in service. Our response times were down and our availability was up.

As far as I know, the technique, while crude, is still in use today.

3. IOT: electric bikes

3.1. what happened?

A few years ago, I was working at a company that made and rented electric bikes, which we controlled over the internet via a pair of microchips. One, the "client", had a cellular connection, which received commands from our servers via tcp/ip and forwarded them to the second, microchip, the "controller", which had a bluetooth connection and was wired to the bike's electronics.

Shortly after I started, I was asked to look into a problem where bikes would stop responding to TCP/IP commands over a few days. I pretty quickly discovered that the client would become unresponsive over time. Chips would start out working fine, but after a few days, they would gradually get laggier, then eventually stop responding to commands altogether. They would 'seem' alive, but they wouldn't respond to any commands. Even removing the battery and plugging it back in wouldn't fix it, which should have been a hard boot.

Since we had no access to the firmware, we couldn't find out WHY it was happening or write a patch ourselves. We contacted the manufacturer of the client, but while they gave us some time and additional documentation, we were a low-volume customer and they didn't have much capacity to help us. The additional documentation noted that the client chip had a small, non-replaceable backup battery that would preserve the contents of the chip's memory for a few days after the main battery was removed. Good in theory, but what it was *actually* doing was keeping our bikes in a zombie state for days after we removed the battery: neither completely alive nor completely dead.

In theory, we now had a perfect solution to the problem: every night, when you plugged the bike in to charge, just hit the reboot button on the client. Only, of course, there *was* no reboot button, because the microchip had not been designed to have a bug. Not only was there no reboot button, there was a thick layer of epoxy between us and the microchip, sealing it onto a board in an effort to protect it from wind, rain, and dust.

In our haste to 'protect' the chip, we had made them impossible to fix. An electric drill and some elbow grease *could* get you in, but it was harder work and more expensive than buying the fucking bikes in the first place. And, of course, that was assuming you didn't accidentally break the electronics the rest of the way by drilling them out in the first place. I'm sure *someone* could have done it, but not anyone we could afford.

And even if the software would be trivial to fix for the next revision, we still needed some way to mitigate the pain for the thousands of bikes we still had! We were well and truly SOL. **If the bikes had designed to have a physical reset switch, we would have saved hundreds of thousands of dollars**.

3.2. how bad was it?

I would estimate that between 20% and 30% of the bikes that would otherwise be available for service were sitting in a warehouse at any given time, waiting for their backup batteries to die. Let's say 20%. Since we had to buy more to keep our desired availability, that's like adding a 25% surcharge to the cost of every bike! As a startup, you can't really afford to burn cash like that. Let's do a little table of costs here:

cost/bike	% idle	total bikes, USD	wasted money , USD
$100	20	5000	$100,000
$100	30	5000	$150,000
$100	20	20,000	$500,000
$100	30	20,000	$750,000

If we had focused on making it **easy and fast to turn on and off again**, we never would have had such a problem. A few months down the line, we had a big meeting where we were going to design a new bike. Everyone got to pitch features. I asked for a big button that physically disconnected power to everything. Everyone laughed. I told them I wasn't joking, and that it was crucial to reliability. They said they'd think about it. I never got to find out, because instead of a new button, we got a nice round of layoffs when the company ran into financial trouble. Who knows - maybe if we'd had to buy fewer bikes, we'd have had more money to keep going.

C'est la vie. Let's talk about booting up and shutting down.

4. practical tips

Hopefully you're convinced that we should make it easy to Restart, Reboot, and Reinstall. Let's talk about how to do that.

4.1. RESTART/REINSTALL: languages

Choice of language matters. Some languages naturally lend themselves to programs which are self-contained and easy to reinstall and restart.

4.1.1. JVM-based (Java, Kotlin, Scala, Clojure, Groovy)

These are the slowest to start of popular languages. The JVM is extremely heavyweight and takes ages to boot up. It's not uncommon for a simple "hello world" to take multiple seconds to start up. This is unacceptable. Lately there have been tools like graalvm to fully ahead-of-time compile Java and it's friends to native code. You can then statically link these executables to make a single binary, which is easier and faster to reinstall and less likely to have 'orphaned' dependencies. Prefer this wherever possible.

4.1.2. .NET (C#, F#, VB.NET)

.NET's story is rather similar to Java's. It's a bit faster to start up, but still pretty slow. Microsoft has made a concerted push over the last few years to migrate these languages to compiling native code with dotnet native. Again, prefer this wherever possible. As far as I'm aware, .net does not yet support fully static binaries, but it seems to be in the pipeline.

4.1.3. Go

See the article mentioned above: start fast: booting go programs quickly. Again, prefer fully static binaries wherever possible, using pure go libraries `CGO_ENABLED=0` where possible and building C dependencies into the binary with `ldflags` and `musl` where not.

4.1.4. Compiled languages without runtime (C, C++, Rust, Zig, ...)

These languages are generally very fast at runtime and support fully static linking via LLVM/Clang. It's still your responsibility to make the boot time fast, but you have the tools to do it.

4.1.5. Traditional scripting languages (Python, Ruby, Perl, PHP, Javascript)

Scripts often boot slowly. While booting the interpreter itself takes some time, that's not usually the bottleneck. Instead, it's the fact that interpreters are:

- slow
- don't do concurrency well,
- must initialize at runtime or import time things other languages could compile away.

Worse yet, scripting languages generally have minimal standard libraries and rely on a large ecosystem of overlapping third-party libraries, each of which must be individually loaded on each run. It's also a nightmare to reliably package these dependencies. Minimizing dependencies and lazy-loading imports can lead to dramatic speedups.

Solutions exist, but often come in the form of things like docker containers, which are slow and heavy if not carefully managed. My artticle, Docker should be fast, not slow: a practical guide to building fast, small docker images may be of some assistance here.

4.1.6. Minimal scripting languages (bash/sh, lua)

The tips written above mostly apply, except the interpreters themselves will boot very quickly.

4.2. dependencies

I've mentioned dependencies a few times, so let's go in to a bit more detail.

4.2.1. Bundle dependencies with your program

Embedded in the binary is best. Within an installer or zip is tolerable. You should be able to *cleanly* install your program by clicking a single button or running a single command. Making it easy to install makes it easy to *reinstall*, which is important for reliability. A single binary is the easiest thing to cleanly install or uninstall. I do not believe the conventional wisdom about not vendoring dependencies. I would rather waste disk space than people's time, and I *don't* waste disk space because I have as few dependencies as I can.

4.2.2. Have as few dependencies as possible

You don't have to load something you don't have. Your program boots at *best* as fast as it's slowest dependency. Consider whether you *really* need yet another DB or external service.

4.2.3. Statically link dependencies where possible

Linking takes very little time, but it's not instantaneous. More troubling is the way DLLs and .so files tend to go missing and break people's installations in hard-to-debug ways. Dynamically loaded dependencies make the "reinstall" part of "Restart, Reinstall, Reboot" much less reliable.

4.2.4. Load binary dependencies concurrently with your program

That is, instead of starting `postgres` and then starting your program, start them both at the same time, and have a bit of logic in your program to block until the database is ready before, say, opening a connection to it. This is extra work for the *first* dependency, but the approach scales well to multiple dependencies.

Your processor is generally wildly underused during program initialization. Use it to do work in parallel.

4.2.5. shutting down dependencies

Many dependencies are not designed to be shut down gracefully - they're long-living services that might be relied upon by other software. This is a pain and I don't know of any perfect solutions.

4.3. starting a program quickly

My first article in this series goes into more detail, but here's a quick summary:

- measure your initialization time to see if and where you have a problem
- use traditional optimization techniques to shave time off hot-spots
- use nonblocking eager initialization or lazy initialization to free the main thread during startup
- store assets in a way that makes them easy to load quickly

5. shutting down your program

- Always respect signals from the operating system. SIGINT means "stop soon". SIGTERM means "stop now". I usually don't handle SIGTERM and let it just kill the program.
- On receipt of a signal, *immediately* stop accepting 'new work' and begin cleanup.
- Where possible, do cleanup concurrently. It is often possible to begin cleanup 'before' receiving a shutdown signal, by autosaving user data, for example.
- Wire all long-running tasks to a signal-handling mechanism that can prime them for cancellation. (In go, this is usually context.Context.Done()).
- Shutdowns, like initialization, should be measured for speed. They are *even more crucial*, because if you don't stop fast enough, someone will do it for you.
- Design your program to be randomly killed. Even if your users don't deliberately exit the program, power & hardware failures happen to everyone.

This sounds a bit abstract, so I'll close with a few concrete examples (in Go). We'll start with a basic HTTP Server:

5.1. shutdown example 1: http server

```go
// an example of how to use signal.NotifyContext to gracefully shutdown a server.
// https://go.dev/play/p/_NzUaJqCGgm
func main() {
    // When the OS sends us an interrupt signal (i.e., via Ctrl+C),
    // sigCtx will be cancelled.
    sigCtx, cancel := signal.NotifyContext(context.Background(), os.Interrupt)
    defer cancel()

    server := &http.Server{
        Addr: ":8080",
        Handler: http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
            w.Write([]byte("Hello World"))
        }),
        // BaseContext is the default context for incoming requests.
        // Since we are using a signal.NotifyContext, new connections will have an automatically-cancelled context
        // after we receive an interrupt signal.
        BaseContext: func(_ net.Listener) context.Context { return sigCtx },
        // any server, no matter how trivial, should have timeouts.
        ReadTimeout:  500 * time.Millisecond,
        WriteTimeout: 500 * time.Millisecond,
        IdleTimeout:  time.Second,
    }
    log.Println("Starting server on :8080")
    go func() {
        defer cancel()
        server.ListenAndServe()
    }()
    <-sigCtx.Done()
    // we've received an interrupt signal.
    // let's give the server 350ms to finish off it's current connections.
    ctx, cancel := context.WithTimeout(context.Background(), 350*time.Millisecond)
    defer cancel()
    if err := server.Shutdown(ctx); err != nil {
        log.Fatal(err)
    }
}
```

6. shutdown example 2: video game

For a slightly more complex example, let's look at how my game, Tactical Tapir, handles graceful shutdown. Nearly the first thing we do in main is register signal handling:

```
var shutdownState atomic.Int64 // atomics make sure that a write can't be lost due to a race and will eventually be seen by all goroutines.
const (
    shutdownNone int64 = 0 // default value
    shutdownTriggered int64 = 1 // we've received a shutdown signal
    shutdownStarted int64 = 2 // main goroutine has started the shutdown process (i.e., its waiting for all shutdown tasks to complete)
    shutdownComplete int64 = 3 // shutdown tasks have completed: main goroutine can exit.
    shutdownTimeout int64 = 4 // shutdown tasks failed: main goroutine must exit.
)

func main() {
    { // signal handling
        log.Println("main: registered SIGINT (Ctrl+C) handler")
        sigCh := make(chan os.Signal, 1)
        signal.Notify(sigCh, syscall.SIGINT)
        go func() {
            <-sigCh
            gamelog.Infof("caught SIGINT, exiting")
            shutdownState.Store(shutdownTriggered)
        }()
    }
}
```

And then, in the main game loop, the first thing we do on every frame is check if we've received a shutdown signal. Most of our branches are pretty simple:

```
// update game logic for frame; don't draw anything or play sounds yet
func (g *Game) Update() error {
SHUTDOWN:
    switch shutdownState.Load() { // atomic
    default:
        panic("unreachable") // we should never get here.
    case shutdownComplete:
        return errors.New("shutdown OK")
    case shutdownTimeout:
        return errors.New("shutdown failed: timeout")
    case shutdownNone, shutdownStarted:
        break SHUTDOWN // nothing for us to do.
```

But one is a bit more complex. Let's look at it in detail. If you're not familiar with the CAS/Compare and Swap operation, you might want to read up on it.

We want to start the shutdown signal exactly once.

```
 1       case shutdownTriggered:
 2           // we've received a shutdown signal. we only want
    to start the shutdown process once.
 3           if !
    shutdownState.CompareAndSwap(shutdownTriggered,
    shutdownStarted) {
 4               break SHUTDOWN
 5           }
 6           // we are the first and only goroutine to start the
    shutdown process.
 7
 8           const timeout = 500*time.Millisecond // force-
    shutdown timeout
 9           gamelog.Warnf("got an interrupt: shutting down in
    %s", timeout)
10
11           go func() { // force-shutdown goroutine.
12               time.Sleep(timeout)
13               // tell main that the shutdown failed due to
    timeout.
14               // we CAS for the following reason: if graceful
    shutdown succeeded between when we started the timer and
    now, we don't want to overwrite it and tell the user it
    failed when it actually succeeded (avoid false positives)
15               shutdownState.CompareAndSwap(shutdownStarted,
    shutdownTimeout)
16           }()
17           go func() { // graceful shutdown goroutine.
18           // this will 'race' with the force-shutdown
    goroutine: whichever one finishes first will set the
    shutdown state.
19           // each shutdown task is done in parallel so that
    we can finish as quickly as possible.
20           // right now we only have two tasks, a rotating
    autosave and saving the console history,
21           // but adding more would usually be as simple as
    adding another goroutine and wg.Done() to this function.
22               wg := new(sync.WaitGroup)
23               wg.Add(2)
24               go func() {
25                   defer wg.Done()
26                   err := g.PlayState.AutoSave()
27                   if err != nil {
28                       gamelog.Errorf("error autosaving: %v",
    err)
29                   }
30               }()
31               go func() {
```

```
32                  defer wg.Done()
33                  err := g.Console.SaveHistory()
34                  if err != nil {
35                      gamelog.Errorf("error saving history:
    %v", err)
36                  }
37              }()
38
    wg.Wait() // synchronization point for autosave and history
    save. if we get past here, they're done.
39              shutdownState.Store(shutdownComplete) // NOT a
    CAS: we want this to 'win' over the timeout goroutine.
40              // after all, if we time out AFTER this, we
    still gracefully shutdown before the deadline.
41          }()
42      }
43      // rest of game loop goes here
44  }
```

After graceful shutdown or an *extremely* short timeout (like 500ms), it should **forcefully shut down**. It's your responsibility to make sure that the graceful shutdown process is as fast as possible. If you don't do that, people will find a way to hard-kill it without you. If your shutdown process is quick & reliable, people will use it. If it's awful, they'll merely pull the plug.

7. conclusion

Turning it on and off again is the lived reality of software engineering, whether we like it or not. Let's stop pretending our programs won't fail, and design them to make that failure as painless and transitory as possible.

MORE ARTICLES