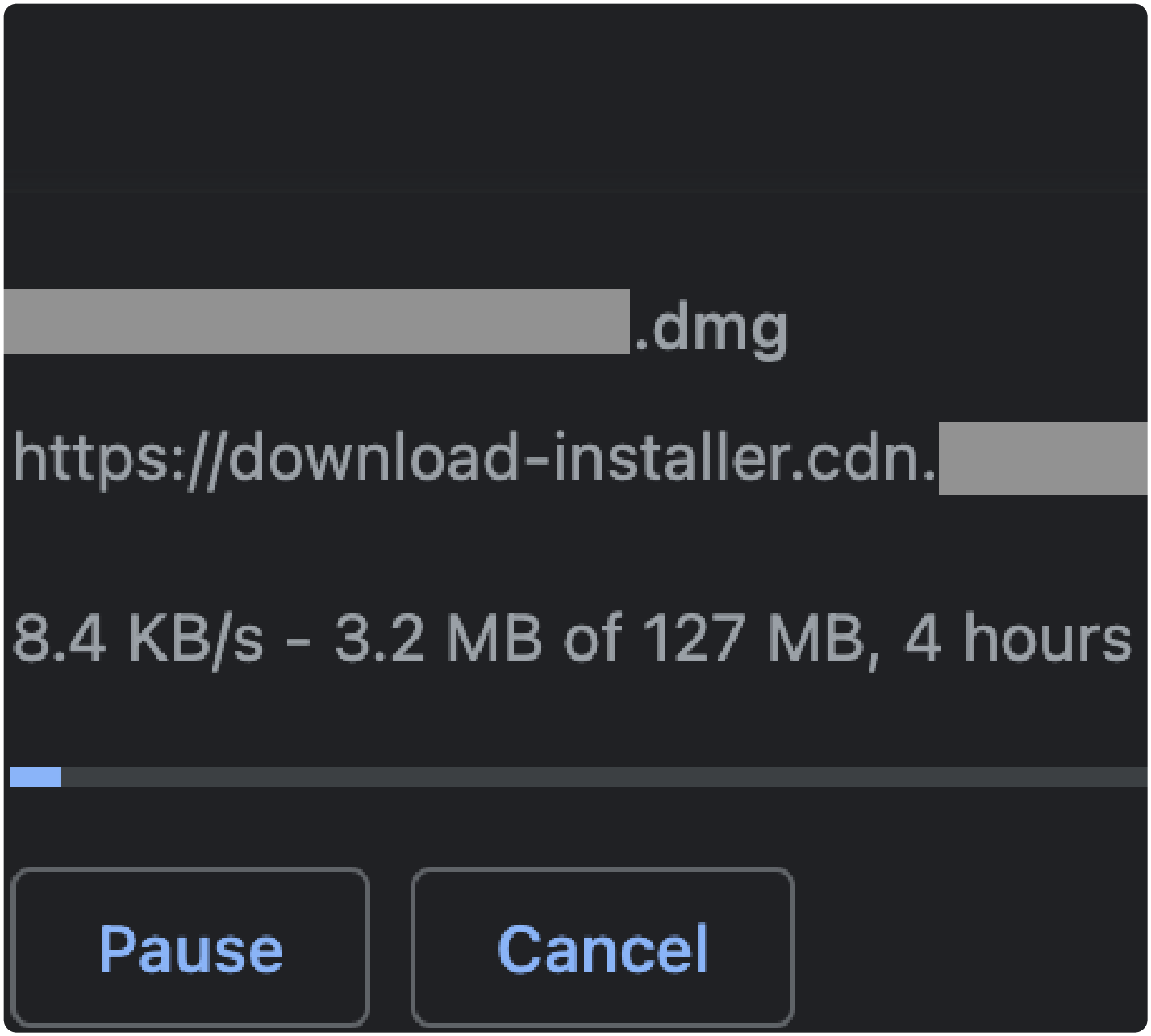


Technology South Pole

Engineering for Slow Internet

How to minimize user frustration in Antarctica.

brr
May 30, 2024

Hello everyone! I got partway through writing this post while I was still in Antarctica, but I departed before finishing it.

I'm going through my old draft posts, and I found that this one was nearly complete.

It's a bit of a departure from the normal content you'd find on brr.fyi, but it reflects my software / IT engineering background.

I hope folks find this to be an interesting glimpse into the on-the-ground reality of using the Internet in bandwidth-constrained environments.

Please keep in mind that I wrote the majority of this post ~7 months ago, so it's likely that the IT landscape has shifted since then.

Welcome back for a **~~bonus post~~** about Engineering for Slow Internet!

For a 14-month period, while working in Antarctica, I had access to the Internet only through an extremely limited series of satellite links provided by the United States Antarctic Program.

Before I go further, this post requires a special caveat, above and beyond my standard disclaimer:

Even though I was an IT worker within the United States Antarctic Program, ***everything*** *I am going to discuss in this post is based on either publicly-available information, or based on my own observations as a regular participant living on ice.*

I have not used any internal access or non-public information in writing this post.

As a condition of my employment, I agreed to a set of restrictions regarding public disclosure of non-public Information Technology material. I fully intend to honor these restrictions. These restrictions are ordinary and typical of US government contract work.

It is unlikely that I will be able to answer additional questions about matters I discuss in this post. I've taken great care to write as much as I am able to, without disclosing non-public information regarding government IT systems.

Good? Ok, here we go.

... actually wait, sorry, one more disclaimer.

This information reflects my own personal experience in Antarctica, from August 2022 through December 2022 at McMurdo, and then from December 2022 through November 2023 at the South Pole.

Technology moves quickly, and I make no claims that the circumstances of my own specific experience will hold up over time. In future years, once I've long-since

forgotten about this post, please do not get mad at me when the on-the-ground IT experience in Antarctica evolves away from the snapshot presented here.

Ok, phew. Here we go for real.

It's a non-trivial feat of engineering to get **any** Internet at the South Pole! If you're bored, check out the South Pole Satellite Communications page on the public USAP.gov website, for an overview of the limited selection of satellites available for Polar use.

South Pole's radomes, out in the RF sector. These radomes contain the equipment necessary to communicate with the outside world using our primary satellites.

If you're interested, perhaps also look into the 2021 Antarctic Subsea Cable Workshop for an overview of some hurdles associated with running traditional fiber to the continent.

I am absolutely not in a position of authority to speculate on the future of Antarctic connectivity! Seriously. I was a low-level, seasonal IT worker in a large, complex organization. Do not email me your ideas for improving Internet access in Antarctica — I am not in a position to do anything with them.

I do agree with the widespread consensus on the matter: There is **tremendous interest** in improving connectivity to US research stations in Antarctica. I would timidly conjecture that, at some point, there will be engineering solutions to these problems. Improved connectivity will eventually arrive in Antarctica, either through enhanced satellite technologies or through the arrival of fiber to the continent.

But — that world will only exist at some point in the future. Currently, Antarctic connectivity is *extremely limited*. What do I mean by that?

Until very recently, at McMurdo, nearly **a thousand people**, plus numerous scientific projects and operational workloads, all relied on a series of links that provided max, aggregate speeds of a few dozen megabits per second to the **entire station**. For comparison, that's less bandwidth shared by everyone **combined** than what everyone **individually** can get on a typical 4g cellular network in an American suburb.

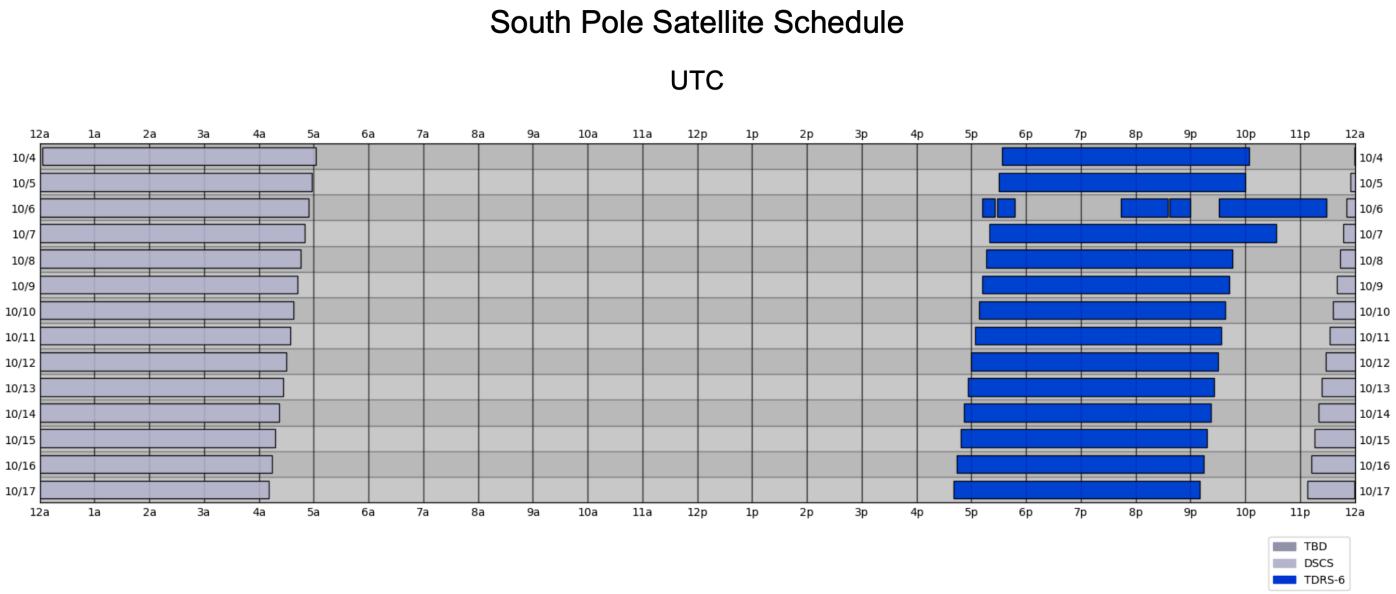
Things **are** looking up! The NSF recently announced some important developments regarding Starlink at McMurdo and Palmer.

I'm aware that the on-the-ground experience in McMurdo and Palmer is better now than it was even just a year ago.

But – as of October 2023, the situation was still pretty dire at the South Pole. As far as I'm aware, similar developments regarding Starlink have **not** yet been announced for South Pole Station.

As of October 2023, South Pole had the limitations described above, **plus** there was only connectivity for a few hours a day, when the satellites rose above the horizon and the station was authorized to use them. The satellite schedule generally shifts forward (earlier) by about 4 minutes per day, due to the difference between Sidereal time and Solar (Civil) time.

The current satellite schedule can be found online, on the South Pole Satellite Communications page of the public USAP.gov website. Here's an example of the schedule from October 2023:



South Pole satellite schedule, for two weeks in October 2023.

These small intermittent links to the outside world are shared by **everyone at Pole**, for operational, science, and community / morale usage.

Complicating matters further is the unavoidable physics of this connectivity. These satellites are in a high orbit, thousands of miles up. This means high latency. If

you've used a consumer satellite product such as HughesNet or ViaSat, you'll understand.

From my berthing room at the South Pole, it was about **750 milliseconds**, round trip, for a packet to get to and from a terrestrial US destination. This is about **ten times** the latency of a round trip between the US East and West coasts (up to 75 ms). And it's about **thirty times** the expected latency of a healthy connection from your home, on a terrestrial cable or fiber connection, to most major content delivery networks (up to 25 ms).

Seriously, I can't emphasize how jarring this is. At my apartment back home, on GPON fiber, it's about 3 ms roundtrip to Fastly, Cloudflare, CloudFront, Akamai, and Google. At the South Pole, the latency was over **two hundred and fifty times greater**.

I can't go into more depth about how USAP does prioritization, shaping, etc, because I'm not authorized to share these details. Suffice to say, if you're an enterprise network engineer used to working in a bandwidth-constrained environment, you'll feel right at home with the equipment, tools, and techniques used to manage Antarctic connectivity.

Any individual trying to use the Internet for community use at the South Pole, as of October 2023, likely faced:

- Round-trip latency averaging around 750 milliseconds, with jitter between packets sometimes exceeding several seconds.
- Available speeds, to the end-user device, that range from a couple kbps (yes, you read that right), up to 2 mbps on a **really good** day.
- Extreme congestion, queueing, and dropped packets, far in excess of even the worst oversaturated ISP links or bufferbloat-infested routers back home.
- Limited availability, frequent dropouts, and occasional service preemptions.

These constraints *drastically* impact the modern web experience! Some of it is unavoidable. The link characteristics described above are truly bleak. But — a lot of the end-user impact is caused by web and app engineering which fails to take slow/intermittent links into consideration.

If you're an app developer reading this, can you tell me, off the top of your head, how your app behaves on a link with 40 kbps available bandwidth, 1,000 ms latency, occasional jitter of up to 2,000 ms, packet loss of 10%, and a complete 15-second connectivity dropout every few minutes?

It's probably not great! And yet — these are real-world performance parameters that I encountered, under certain conditions, at the South Pole. It's normally better than this, but this does occur, and it occurs often enough that it's worth taking seriously.

This is what happens when you have a tiny pipe to share among high-priority operational needs, plus dozens of community users. Operational needs are aggressively prioritized, and the community soaks up whatever is left.

I'm not expecting miracles here! Obviously no amount of client engineering can make, say, real-time video conferencing work under these conditions. But — getting a few bytes of text in and out **should** still be possible! I know it is possible, because some apps are still able to do it. Others are not.

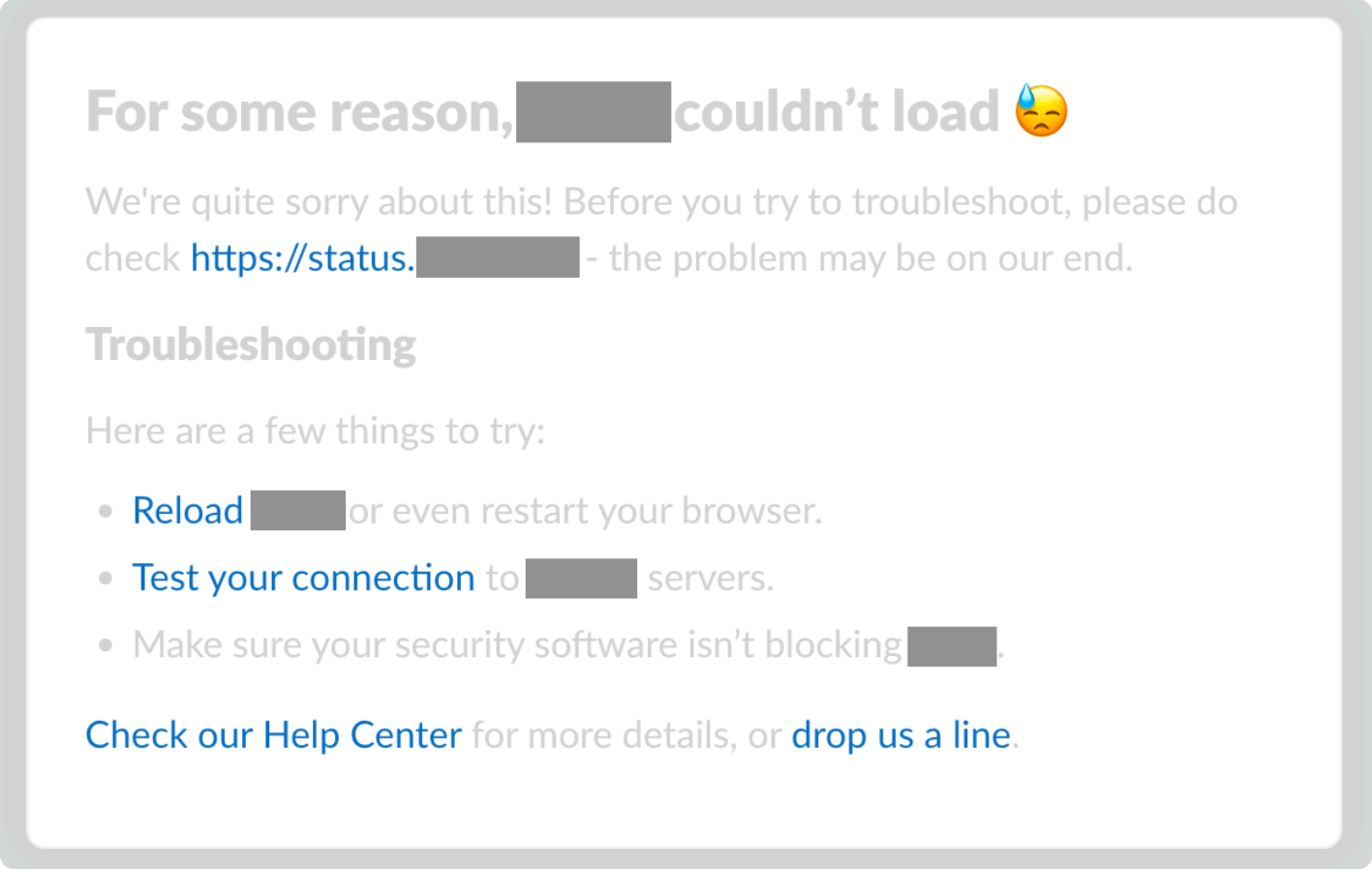
Detailed, Real-world Example

One day at the South Pole, I was trying to load the website of ***<$enterprise_collaboration_platform>*** in my browser. It's *huge*! It needed to load nearly 20 MB of Javascript, *just* to render the main screen! And of course, the app had been updated since last time I loaded it, so all of my browser's cached assets were stale and had to be re-downloaded.

Fine! It's slow, but at least it will work… eventually, right? Browsers do a decent job of handling slow Internet. Under the hood, the underlying protocols do a decent job at congestion control. I should get a steady trickle of data. This will be subject to the negotiated send and receive windows between client and server, which are based on the current level of congestion on the link, and which are further influenced by any shaping done by middleware along the way.

It's a complex webapp, so the app developer would also need to implement some of their own retry logic. This allows for recovery in the event that individual assets fail, especially for those long, multi-second total connectivity dropouts. But eventually, given enough time, the transfers should complete.

Unfortunately, this is where things broke down and got really annoying. *The developers implemented a global failure trigger somewhere in the app.* If the app didn't fully load within the parameters specified by the developer (time? number of retries? I'm not sure.), then the app **stopped, gave up, redirected you to an error page, dropped all the loading progress you'd made, and implemented aggressive cache-busting countermeasures for next time you retried.**



The app wasn't loading fast enough, and the developers decided that the app should give up instead of continuing to load slowly.

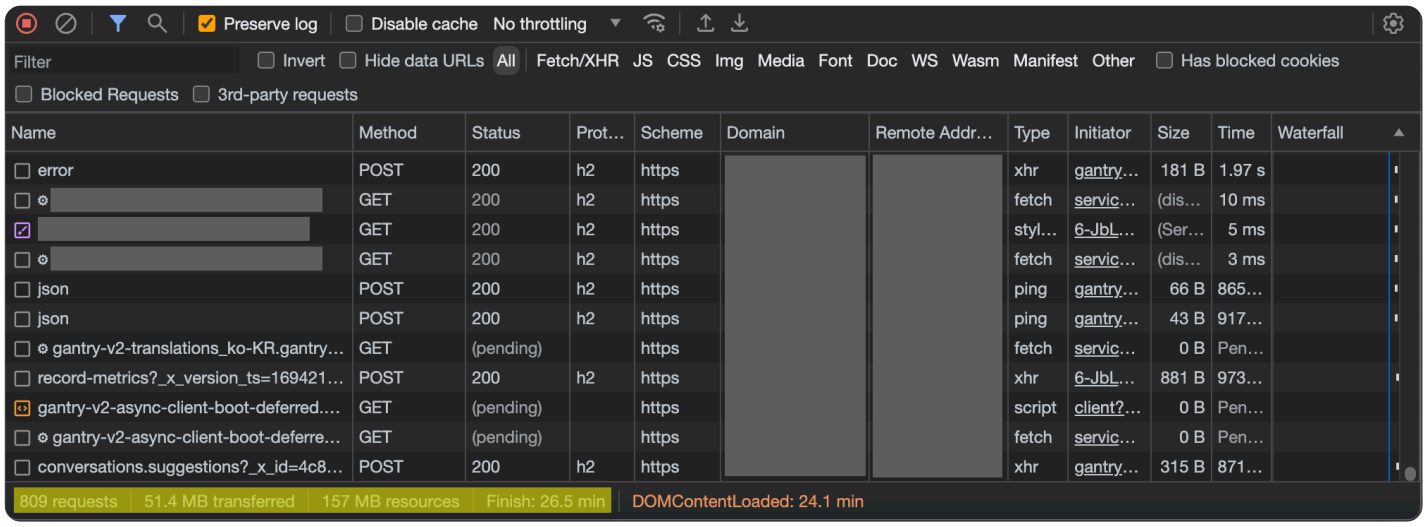
I cannot tell you how frustrating this was! Connectivity at the South Pole was never going to meet the performance expectations set by engineers using a robust terrestrial Internet connection. It's not a good idea to hardcode a single, static, global expectation for how long 20 MB of Javascript should take to download. Why not let me load it at my own pace? I'll get there when I get there. *As long as data is still moving, however slow, just let it run.*

But — the developers decided that if the app didn't load within the parameters they set, I couldn't use it at all. And to be clear — this was primarily a **messaging** app. The actual content payload here, when the app is running and I'm chatting with my friends, is measured in *bytes*.

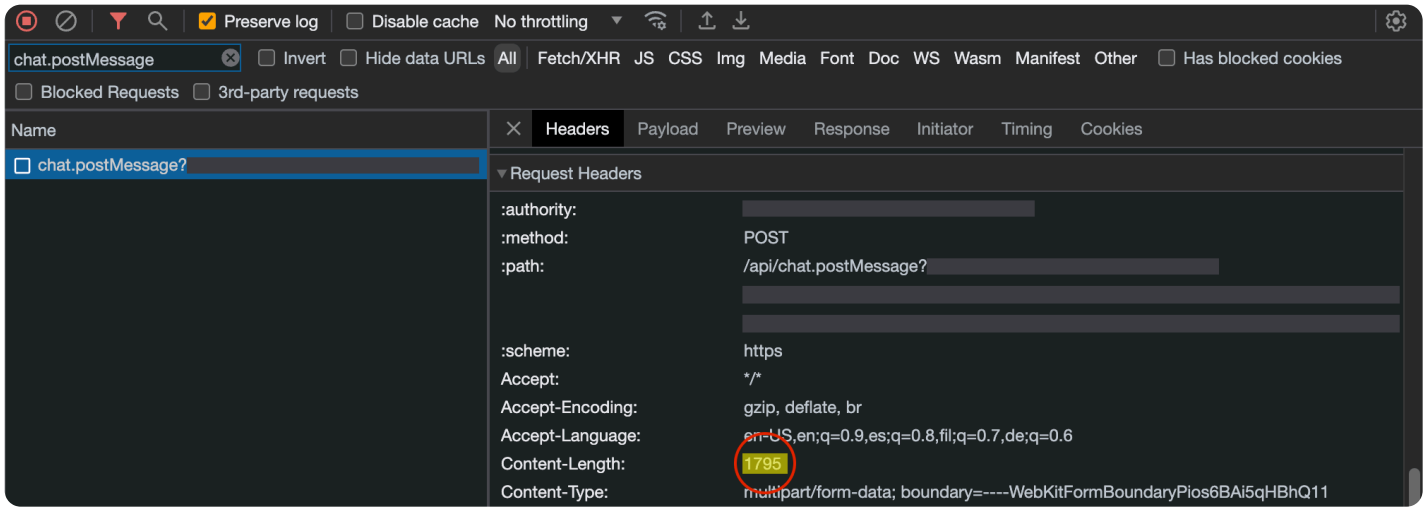
As it turns out, our Internet performance at the South Pole was *right on the edge* of what the app developers considered "acceptable". So, if I kept reloading the page,

and if I kept letting it re-download the same 20 MB of Javascript, and if I kept putting up with the developer's cache-busting shenanigans, *eventually* it finished before the artificial failure criteria.

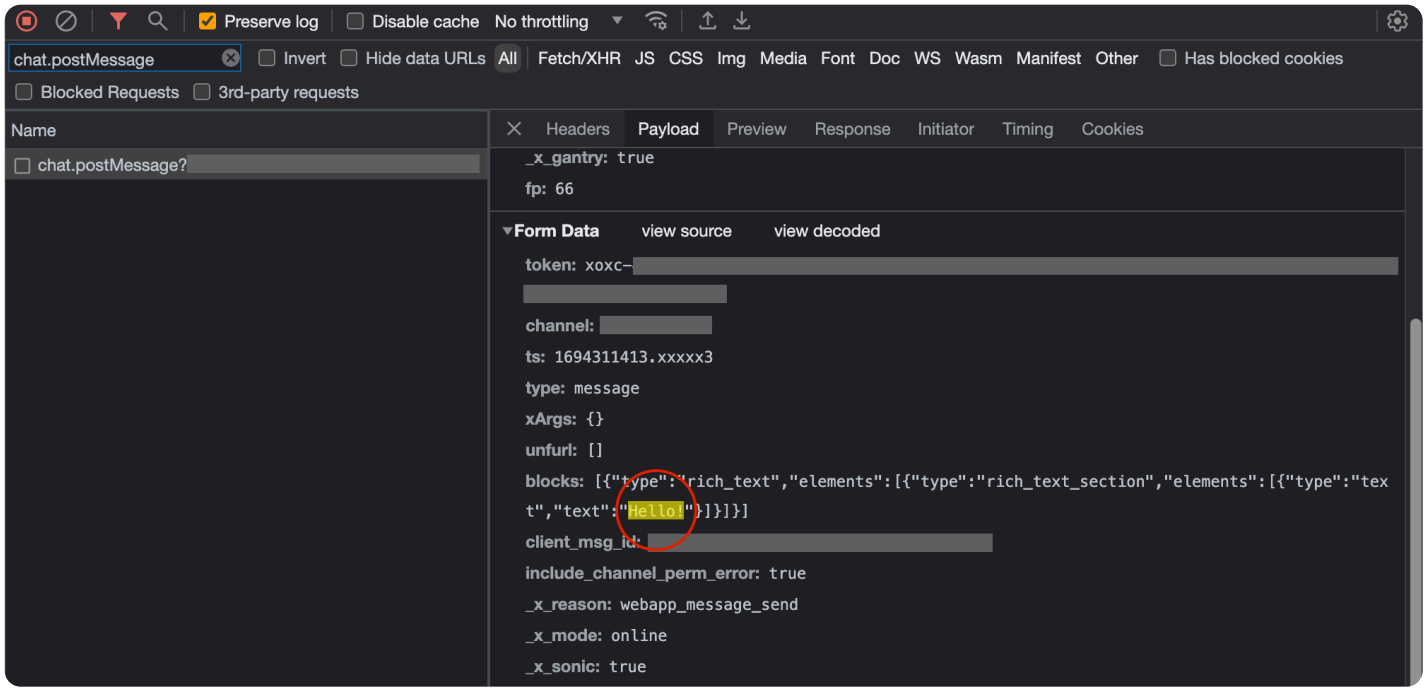
What this means is that I wasted *extra* bandwidth doing all these useless reloads, and it took sometimes **hours** before I was able to use the app. All of this hassle, even though, if left alone, I could complete the necessary data transfer in 15 minutes. Several hours (and a shameful amount of retried Javascript) later, I was finally able to send a short, text-based message to my friends.



A successful webapp load, after lots of retrying. 809 HTTP requests, 51.4 MB of data transfer, and 26.5 minutes of loading...



...all so that I could send a 1.8 KB HTTPS POST...



…containing a 6-byte message.

Does this webapp **really need** to be 20 MB? What all is being loaded that could be deferred until it is needed, or included in an "optional" add-on bundle? Is there a possibility of a "lite" version, for bandwidth-constrained users?

In my 14 months in Antarctica, I collected **dozens** of examples of apps like this, with artificial constraints built in that rendered them unusable or borderline-unusable.

For the rest of this post, I'll outline some of my major frustrations, and what I would have liked to see instead that would mitigate the issues.

I understand that not every app is in a position to implement all of these! If you're a tiny app, just getting off the ground, I don't expect you to spend all of your development time optimizing for weirdos in Antarctica.

Yes, Antarctica is an edge case! Yes, 750 ms / 10% packet loss / 40 kbps **is** rather extreme. But the South Pole was not **uniquely** bad. There are entire commercial marine vessels that rely on older Inmarsat solutions for a few hundred precious

kbps of data while at sea. There's someone at a remote research site deep in the mountains right now, trying to load your app on a Thales MissionLink using the Iridium Certus network at a few dozen kbps. There are folks behind misconfigured routers, folks with flaky wifi, folks stuck with fly-by-night WISPs delivering sub-par service. Folks who still use dial-up Internet connections over degraded copper phone lines.

These folks are worthy of your consideration. At the very least, you should make an effort to avoid **actively interfering** with their ability to use your products.

So, without further ado, here are some examples of development patterns that routinely caused me grief at the South Pole.

Hardcoded Timeouts, Hardcoded Chunk Size

As per the above example, **do not hardcode your assumptions about how long a given payload will take to transfer, or how much you can transfer in a single request.**

1. If you have the ability to measure whether bytes are flowing, and they are, **leave them alone**, no matter how slow. Perhaps show some UI indicating what is happening.
2. If you are doing an HTTPS call, fall back to a longer timeout if the call fails. Maybe it just needs more time under current network conditions.
3. If you're having trouble moving large amounts of data in a single HTTPS call, break it up. Divide the content into chunks, transfer small chunks at a time, and **diligently keep track of the progress**, to allow resuming and retrying small bits without losing all progress so far. Slow, steady, incremental progress is better than a one-shot attempt to transfer a huge amount of data.
4. If you can't get an HTTPS call done successfully, do some troubleshooting. Try DNS, ICMP, HTTP (without TLS), HTTPS to a known good status endpoint, etc.

This information might be helpful for troubleshooting, and it's better than blindly retrying the same end-to-end HTTPS call. This HTTPS call requires a bunch of under-the-hood stuff to be working properly. Clearly it's not, so you should make an effort to figure out why and let your user know.

Example 1 - In-App Metadata Download

A popular desktop application tries to download some configuration information from the vendor's website at startup. There is a hardcoded timeout for the HTTPS call. **If it fails, the app will not load.** It'll just keep retrying the same call, with the same parameters, forever. It'll sit on the loading page, without telling you what's wrong. I've confirmed this is what's happening by reading the logs.

```
09-14 12:26:18.503 F /AbstractTask:2555:d910/ Task
RequestProductsPermissions <0x600003b2ecd0>: new subtask is
RequestProductsPermissions <0>
09-14 12:26:18.503 F /WebPortalCommunication:2555:d910/ GET: https://
[redacted].[redacted].com/api/v1/product_permissions?
[redacted]
09-14 12:26:18.541 F /prl_client_app:2555:d910/ localhost: received
result for [DspCmdUserGetHostHwInfo]. RC = [PRL_ERR_SUCCESS]
09-14 12:26:33.799 F /WebPortalCommunication:2555:d910/ Request
finished with Network Error: 3 <0x6000087ce6c0>
09-14 12:26:33.801 F /WebPortalCommunication:2555:d910/ Request failed.
RC = PRL_ERR_WEB_PORTAL_UNEXPECTED (80047025), httpStatus = 0
 Header: QMap()
 Data: QVariant(Invalid)
 <0x6000087ce6c0>
09-14 12:26:33.801 F /AbstractTask:2555:d910/ Subtask 0
[RequestProductsPermissions] completed with RC = 80047025
[PRL_ERR_WEB_PORTAL_UNEXPECTED]
09-14 12:26:33.801 F /AbstractTask:2555:d910/ Finishing task
RequestProductsPermissions <0x600003b2ecd0> with RC = 80047025
[PRL_ERR_WEB_PORTAL_UNEXPECTED]
```

Excerpt from debug log for a commercial desktop application, showing a request timing out after 15 seconds.

Luckily, if you kept trying, the call would eventually make it through under network conditions I experienced at the South Pole.

It's frustrating that just a single hardcoded timeout value, in an otherwise perfectly-functional and enterprise-grade application, can render it almost unusable. The developers could have:

1. Fallen back to increasingly-long timeouts to try and get a successful result.
2. Done some connection troubleshooting to infer more about the current network environment, and responded accordingly.
3. Shown UX explaining what was going on.
4. Used a cached or default-value configuration, if it couldn't get the live one, instead of simply refusing to load.
5. Provided a mechanism for the user to manually download and install the required data, bypassing the app's built-in (and naive) download logic.

Example 2 - Chat Apps

A popular chat app ("app #1") maintains a websocket for sending and receiving data. The initialization process for that websocket uses a **hardcoded 10-second timeout**. Upon cold boot, when network conditions are especially congested, that websocket setup can sometimes take more than 10 seconds! We have to do a full TCP handshake, then set up a TLS session, then set up the websocket, then do initial signaling over the websocket. Remember — under some conditions, each individual roundtrip at the South Pole took multiple seconds!

If the 10-second timeout elapses, the app simply does not work. It enters a very long backoff state before retrying. The UX does not clearly show what is happening.

```
INFO  2022-12-30T00:06:33.961Z RetryPlaceholders.getExpiredAndRemove: Found 0 expired
items
INFO  2022-12-30T00:06:33.990Z retryPlaceholders/interval: Found 0 expired items
INFO  2022-12-30T00:06:35.454Z SocketManager: connecting authenticated socket
(hasStoriesDisabled=true)
WARN  2022-12-30T00:06:45.489Z SocketManager: authenticated socket connection failed with
error: Error: Connection timed out
    at [REDACTED]/Resources/app.asar/preload.bundle.js:126878:13
    at Timeout._onTimeout ([REDACTED]/Resources/app.asar/preload.bundle.js:9317:11)
    at listOnTimeout (node:internal/timers:559:17)
    at process.processTimers (node:internal/timers:502:7)
INFO  2022-12-30T00:06:45.489Z goBackToMainProcess: Already in the main process
```

Excerpt from debug log for chat app #1, showing the hardcoded 10-second timeout. We did indeed have Internet access at this time – it was just too congested to complete an entire TCP handshake + TLS negotiation + websocket setup + request within this timeframe. With a few more seconds, it may have finished.

On the other hand, a competitor's chat app ("app #2") does *very well* in extremely degraded network conditions! It has multiple strategies for sending network requests, for resilience against certain types of degradation. It aggressively re-uses open connections. It dynamically adjusts timeouts. In the event of a failure, it intelligently chooses a retry cadence. And, throughout all of this, it has clear UX explaining current network state.

The end result is that I could often use app #2 in network conditions when I could not use app #1. Both of them were just transmitting plain text! Just a few actual bytes of content! And even when I could not use app #2, it was at least telling me what it was trying to do. App #1 is written naively, with baked-in assumptions about connectivity that simply did not hold true at the South Pole. App #2 is written well, and it responds gracefully to the conditions it encounters in the wild.

Example 3 - Incremental Transfer

A chance to talk about my own blog publishing toolchain!

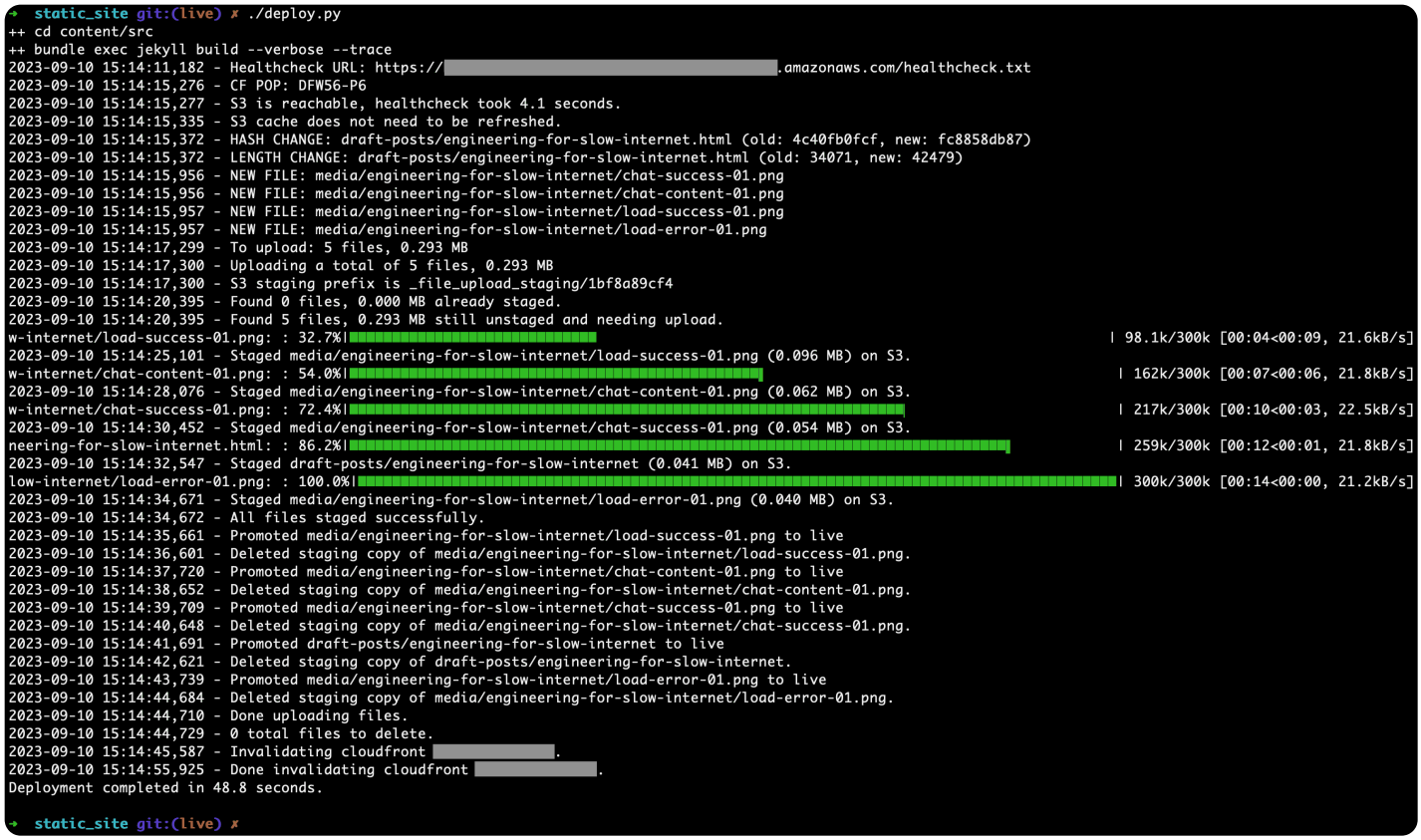
The site you're reading right now is a static Jekyll blog. Assets are stored on S3 and served through CloudFront. I build the static files locally here on my laptop, and I upload them directly to S3. Nothing fancy. No servers, no QA environment, no build system, no automated hooks, nothing dynamic.

Given the extreme connectivity constraints at the South Pole, I wrote a Python script for publishing to S3 that worked well in the challenging environment. It uses the S3 API to upload assets in small chunks. It detects and resumes failed uploads without losing progress. It waits until everything is safely uploaded before publishing the new version.

If I can do it, unpaid, working alone, for my silly little hobby blog, in 200 lines of Python… surely your team of engineers can do so for your flagship webapp.

It's amazing the usability improvements that come along with some proactive engineering. I had friends at Pole with blogs on commercial platforms, and who routinely shared large files to social media sites. They had to carefully time their day to maximize the likelihood of a successful "one-shot" upload, during a satellite window, using their platform's poorly-engineered publishing tools. Often it took several retries, and it's not always clear what was happening at every step of the process (Is the content live? Did the upload finish? Is it safe / should I hit "Post" again?).

Meanwhile, I was able to harvest whatever connectivity I could find. I got a few kilobytes uploaded here and there, whenever it was convenient. If a particular chunked POST failed, no worries! I could retry or resume, with minimal lost progress, at a later time. Once it was all done and staged, I could safely publish the new version.



My custom publishing script for this blog, to handle intermittent and unreliable Internet.

Bring-Your-Own-Download

If you're going to build in a downloader into your app, you have a high bar for quality that you have to meet. Otherwise, it's going to fail in profoundly annoying or catastrophic ways.

If I had to give one piece of advice: ***Let the user break out of your in-app downloader and use their own, if at all possible.***

Provide a manual download link, ideally one that leads to whatever differential patch file the app was going to download. Don't punish the user by making them download the full installer, just because your in-app patch downloader doesn't meet their needs.

This has the following benefits:

1. If your downloader fails, the user can still get the file manually, using a more robust downloader of their choice, such as a web browser.
2. The user can download the file one time, and share it with multiple devices.
3. The user can download the file on a different computer than the one running the application.
4. The user has the flexibility to schedule or manage the download based on whatever constraints they face.

Why is this all so important when considering users at the South Pole?

Downloads **are** possible at the South Pole, but they are subject to unique constraints. The biggest constraint is the lack of 24x7 Internet. While I was there, I *knew* we would lose Internet access at a certain time!

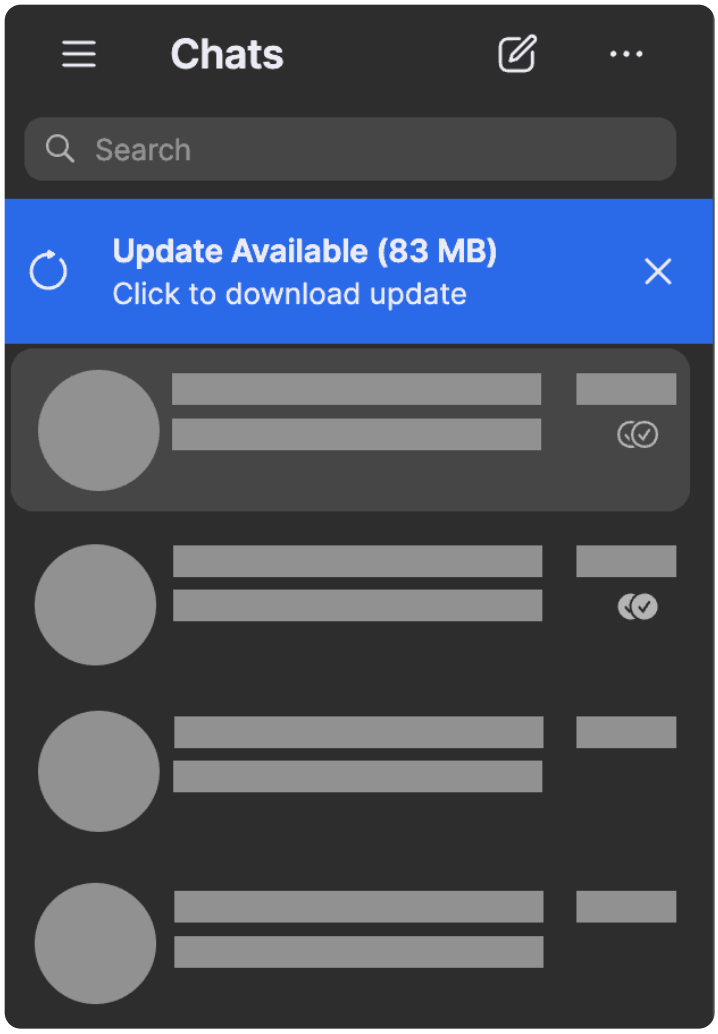
It's a frustrating reality: with most apps that do their own downloads, we were powerless to do anything about this known break in connectivity. We just had to sit there and watch it fail, and often watch all our progress be lost.

Let's say I had a 4-hour window, every day, during which I could do (very slow!!) downloads. If the total amount of data I could download in those 4 hours was **less** than the total size of the payload I was downloading, then there is *no way* I could complete the download in one shot! I'd *have* to split it over multiple Internet windows. Often the app wouldn't let me do so.

And that's not even considering the fact that access might be unreliable during that time! What if the underlying connection dropped and I had to resume the download? What if my plans changed and I needed to pause? I didn't want to waste whatever precious little progress I'd made so far.

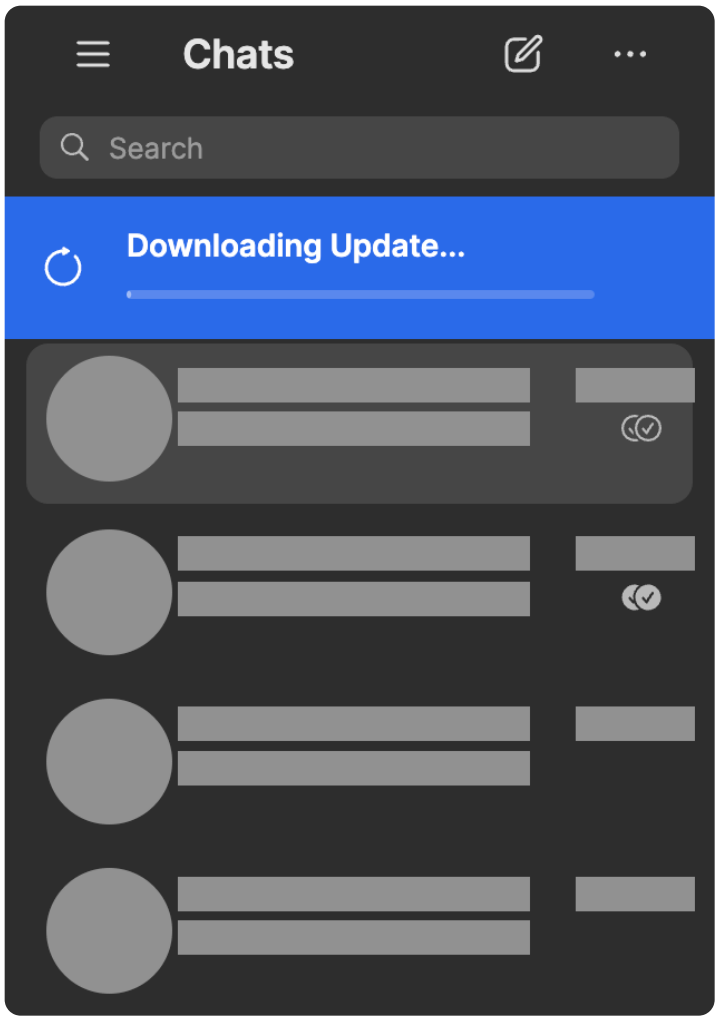
A lot of modern apps include their own homegrown, artisanal, in-app downloaders for large payloads. By "in-app downloader", I'm referring to the system that obtains

the content for automatic updates, patches, content database updates, etc. The common theme here is that the app transparently downloads content for you, without you being exposed to the underlying details such as the URL or raw file. This includes UI patterns such as *Check for updates*, *Click here to download new version*, etc.



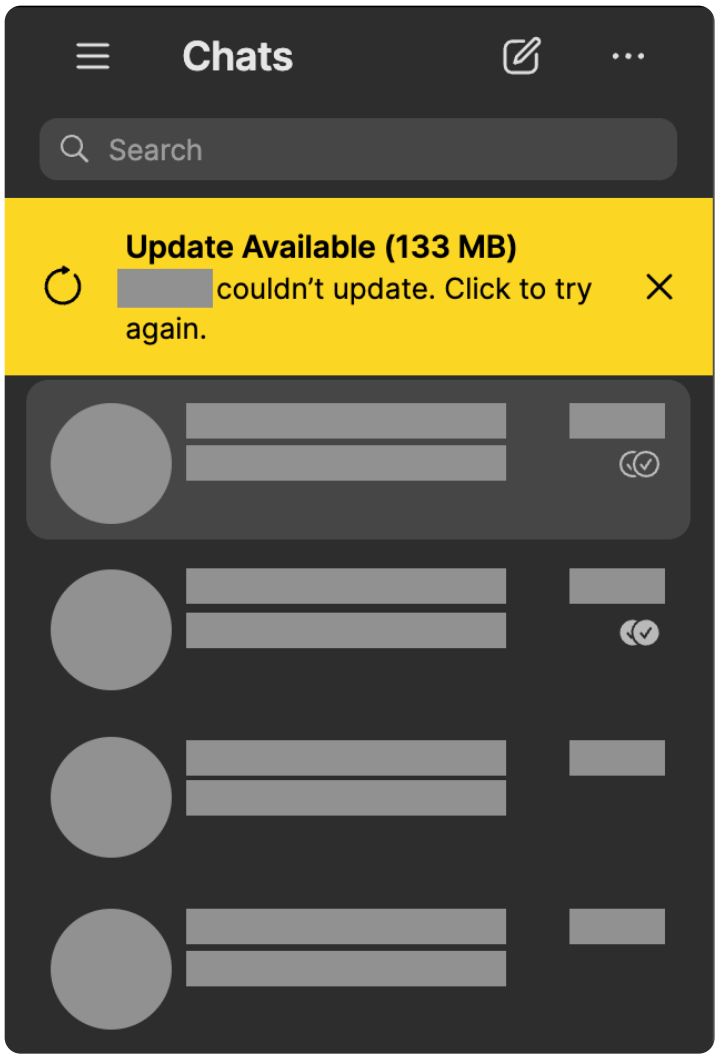
An in-app download notification for a popular chat application. This app apparently wants to download 83 MB of data in the background! This is tough at the South Pole. Will the UI be accommodating of the unique constraints at Pole?

Unfortunately, most of these in-app downloaders are woefully ill-equipped for the task! Many of them lack pause/resume functionality, state notifications, retry logic, and progress tracking. Many of them have frustrating restrictions, such as time limits for downloading the payload. While most of these issues are mere annoyances in the land of fast Internet, at the South Pole, they can make or break the app entirely.



Unfortunately, that's a resounding "no". There's no speed indication, no ETA, no pause button, no cancel button, no URL indication (so we can download manually), and no way to get at the underlying file.

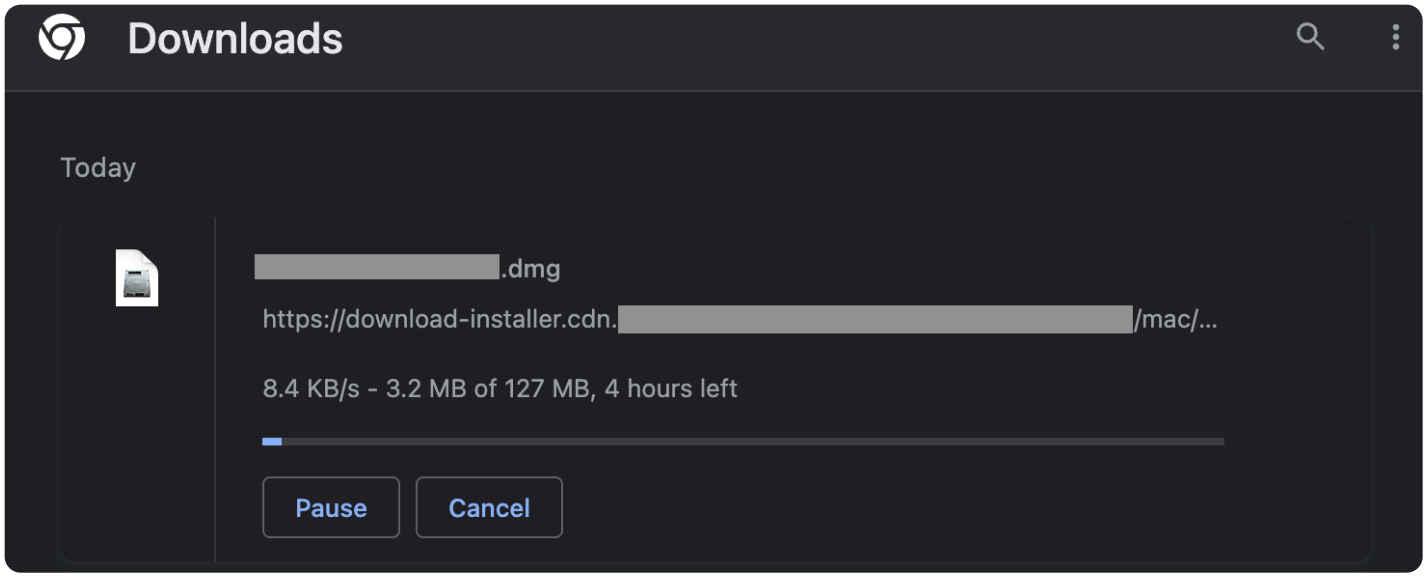
It was always frustrating to face down one of these interfaces, because I knew how much time, and data transfer, was going to be wasted.



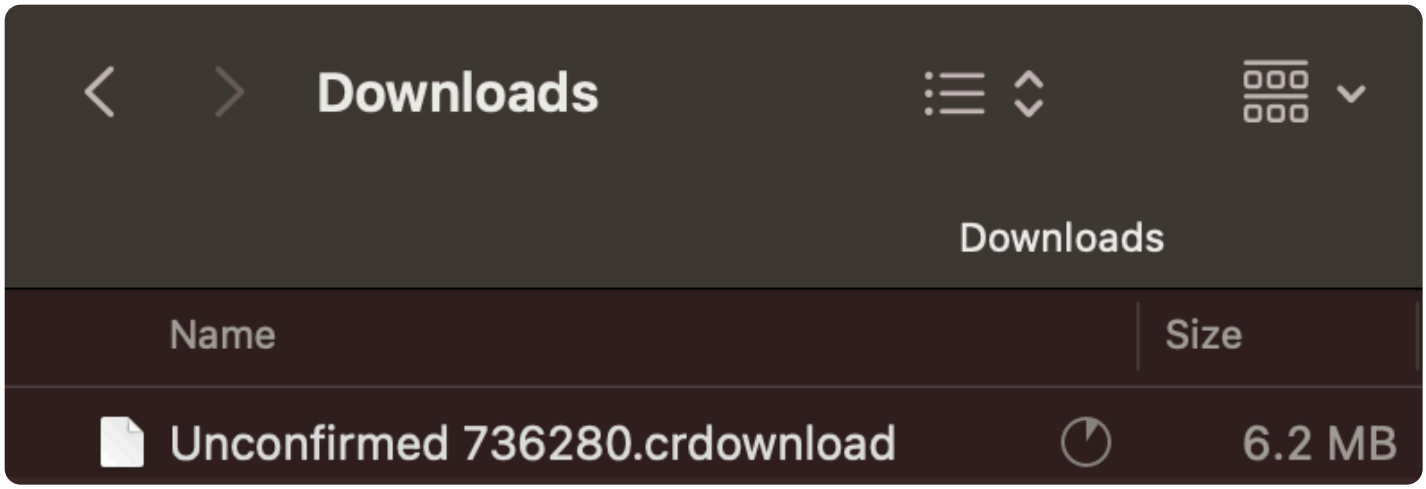
Darn, it failed! This was expected – an uninterrupted 83 MB download is tough. Unfortunately, all progress has been lost, and now it's not even offering a patch on the retry – the download size has ballooned to 133 MB, the size of the full installer.

Every app that includes an in-app downloader has to compete with an extraordinarily high bar for usability: **web browsers**.

Think about it! Every modern web browser includes a download manager that contains Abort, Pause, and Resume functionality. It allows you to retry failed downloads (assuming the content URL doesn't include an expiring token). It clearly shows you current status, download speed, and estimated time remaining. It allows you to choose where you save the underlying file, so you can copy it around if needed. And — it doesn't include arbitrary performance cutoffs! If you really want to download a multi-gigabyte file at 60 kbps, go for it!



A fully-featured download experience, from a major web browser. Downloading an app installer from the vendor's website. Note the status, speed, estimated time remaining, full URL, and pause / cancel buttons.



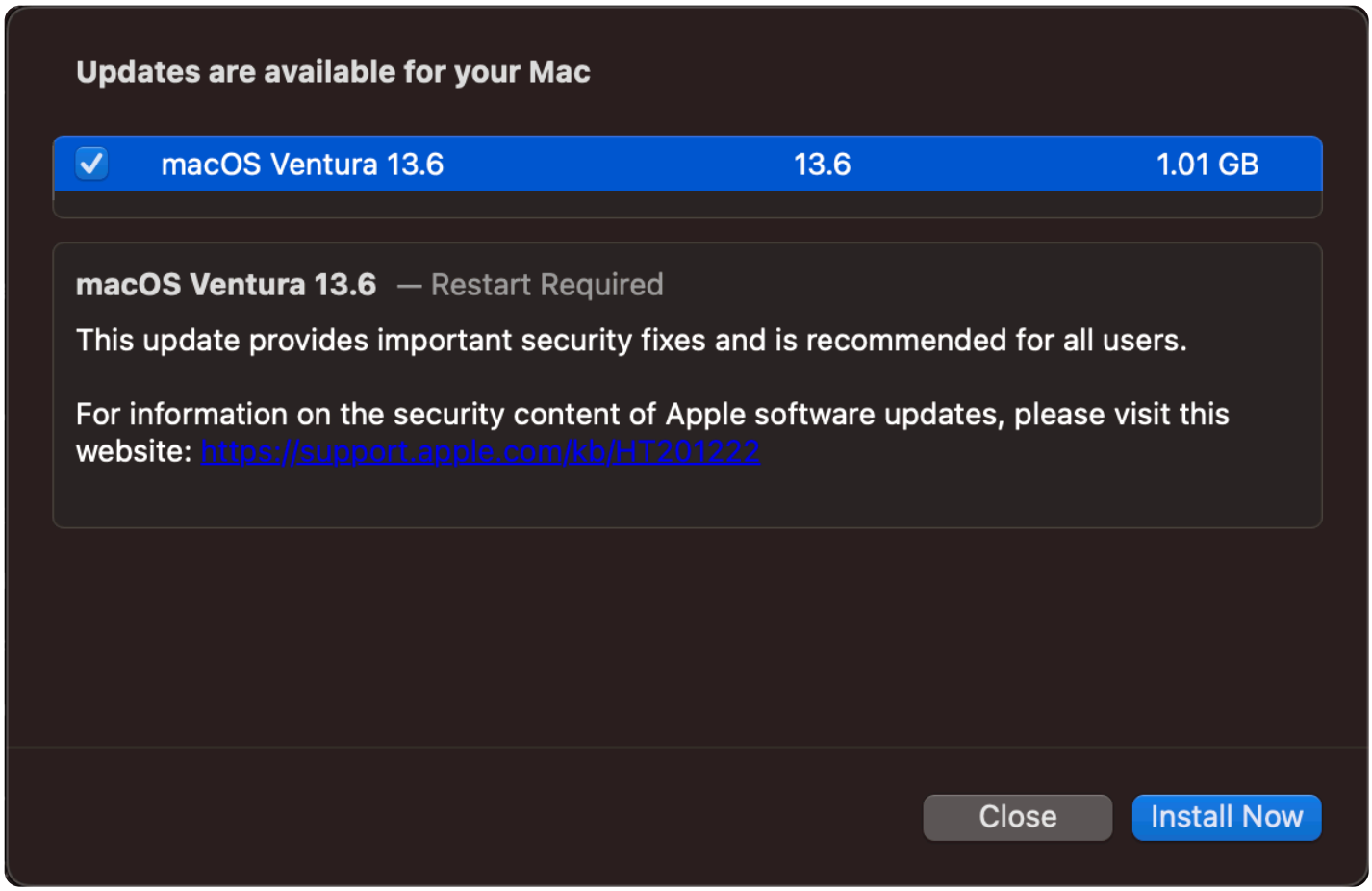
A partially-downloaded file, for the above-mentioned download.

Here are a few more examples of where in-app downloaders caused us grief.

Example 1 - macOS Updates

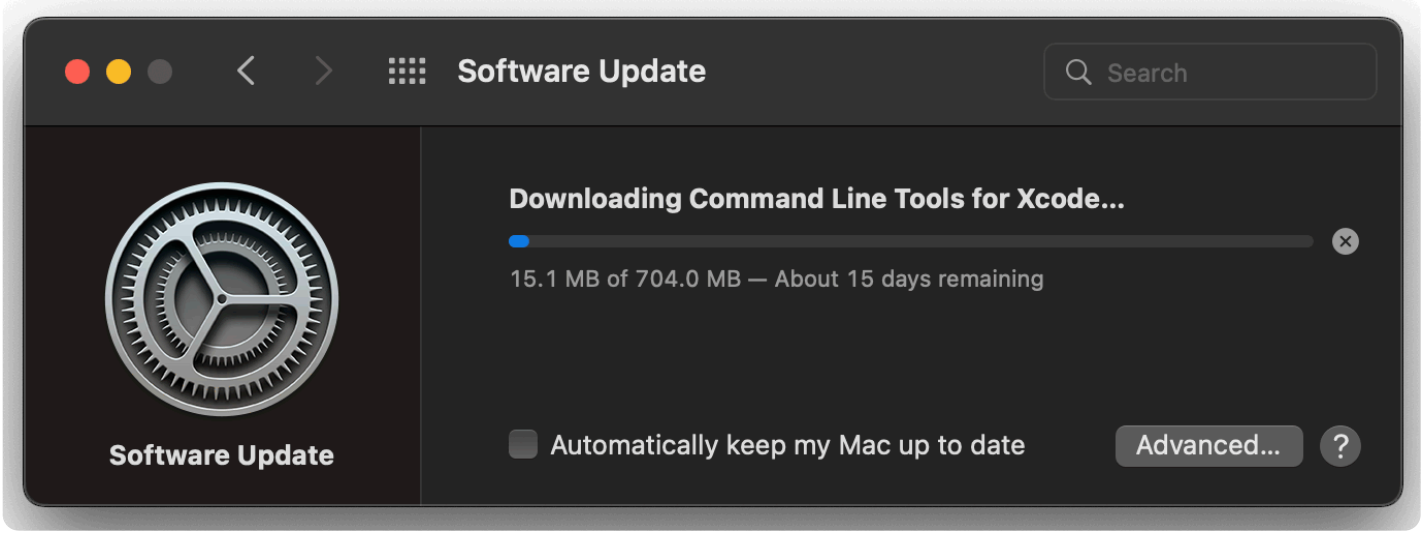
It's no secret that macOS updates are huge. This is sometimes even annoying back home, and it was much worse at the South Pole.

The patch size for minor OS updates is usually between 0.5 and 1.5 gigabytes. Major OS upgrade patches are sometimes 6+ gigabytes. Additional tools, such as Xcode, are often multiple gigabytes.



Sigh, yet another 1 GB patch for my personal macOS device at the South Pole.

If every single macOS device at the South Pole downloaded these updates, directly from Apple, we would have wasted a tremendous amount of bandwidth. And the built-in macOS downloader certainly wanted us to do this! Look at this interface — few controls, no way to break out and easily get the underlying patch files. If I canceled the download, or if it failed for some reason, it didn't always intelligently resume. Sometimes, I lost all my progress.



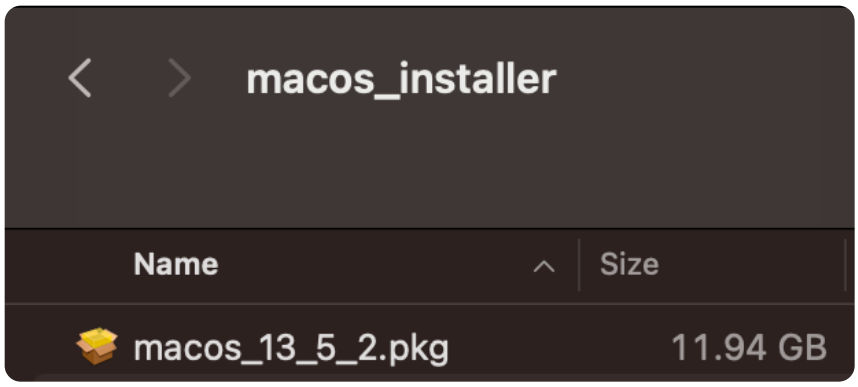
The macOS updater. No pause button! Was I expected to leave my laptop on, connected to the Internet, and untouched, for 15 days??

Now – Apple *does* have a caching server feature built into macOS. In theory, this should alleviate some of the burden! We should be able to leverage this feature to ensure each patch is only downloaded to the South Pole one time, and then client Macs will hit the cache.

I experimented with this feature in my spare time, with a handful of my own Apple devices. In practice, this feature still required each client Macbook to make a successful HTTPS call directly to Apple, to negotiate cache parameters. If this call failed, which it often did (*because of hardcoded short timeouts!!!*), then the client Mac just fetched the patch from public Apple servers. No retry, no notification. The client Mac just made a unilateral decision to bypass the cache, without any recourse or even a notification for the user. In practice, this initial cache negotiation call failed often enough at the South Pole that the caching feature wasn't useful.

What we *could* do was to fetch the full installer (12 gigabytes!) from Apple. Links to the full installer packages are conveniently aggregated on the Mr. Macintosh blog. We could pull the full installer down to the South Pole slowly and conscientiously: throttled, at low, background priority, using robust, interrupt-tolerant tooling, with support for caching and resumption of paused or failed transfers. Once we had the

file, we could distribute it on station. This process could take several days, but it was reliable.



The macOS full installer, painstakingly and conscientiously downloaded to the South Pole.

But *even this* didn't solve the problem! If the client Mac is Apple Silicon, it *still insisted* on downloading additional content directly from Apple, *even if* you ran the update using the full, 12 GB installer. There is no way to bypass or cache this. If the OS update required certain types of firmware updates or Rosetta updates, then *every Apple Silicon client Mac* would *still* download 1-2 GB of data directly from Apple during the install process.

Even worse, the download process was sometimes farmed out to a separate component in macOS, which didn't even report progress to the installer! Installing a macOS update at the South Pole meant staring at a window that said "installing, 32 minutes remaining", for *several hours*, while a subcomponent of macOS downloaded a gigabyte of un-cacheable data in the background.

Apple naively assumed that the 1 GB download would be so fast that they didn't bother incorporating download speed feedback into the updater's time estimate. They did not anticipate people installing macOS updates from a location where a gigabyte of downloads can take **several hours**, if not **days**.

You can't cache it, and you can't download it directly using a web browser (or other mechanism). You have to let Apple's downloader do it directly. And, of course, there's no pause button. It is a major inconvenience to users, and a major waste of

bandwidth, for each individual client Mac to download 1-2 GB of data in a single, uninterrupted shot.

Ways that Apple could make this significantly better for users with slow or otherwise-weird Internet:

1. Compute the required patch, and then give us a download link, so we can download it outside of Apple's downloader.
2. Improve the built-in update download tool with pause/resume functionality and intelligent state management, to ensure progress isn't lost.
3. Fix the full installer, so it includes *everything*, including all the currently-excluded items such as firmware and Rosetta updates for Apple Silicon Macs. It would be much more useful if it included everything. I could download it once, and then distribute it, without worrying about each Mac *still* needing to fetch additional data from Apple.
4. Improve the Apple Caching Server feature, so it's more reliable in situations where direct Internet access is unreliable. Give us more controls so that we can force a Mac to use it, and so that we can force the caching server to proactively download an item that we know will be needed in the future.

As it stands, it was a huge hassle for me to help people with macOS updates at the South Pole.

Example 2 - Samsung Android Phone OS Updates

My Samsung Android phone receives periodic OS updates. These updates include relevant Android patches, as well as updates to the Samsung UI and other OS components.

The updater is a particularly bad example of an app that fails to consider slow / intermittent Internet use cases.

Downloading an OS update for my Samsung Android phone at the South Pole.

First, the basics. There is no speed indicator, no numeric progress indicator (good luck counting pixels on the moving bar), no pause button, no cancel button, no indicator of the file size, and no way to get at the underlying file to download separately.

Second — if the download fails, it cannot be resumed. It will restart from the beginning.

In practice, at the South Pole, the phone could not download an entire OS update on a single satellite pass. So — it inevitably failed as soon as connectivity dropped, and I had to restart it from the beginning.

The **only way** I was able to get this done was by **turning off the phone entirely**, right before Internet access dropped, and then turning it back on when Internet access resumed at the next satellite pass. This tricked the phone into not giving up on the download, because it was totally off during the period without Internet. It never had a chance to fail. By doing this, I was able to spread out the download across multiple satellite passes, and I could complete the download.

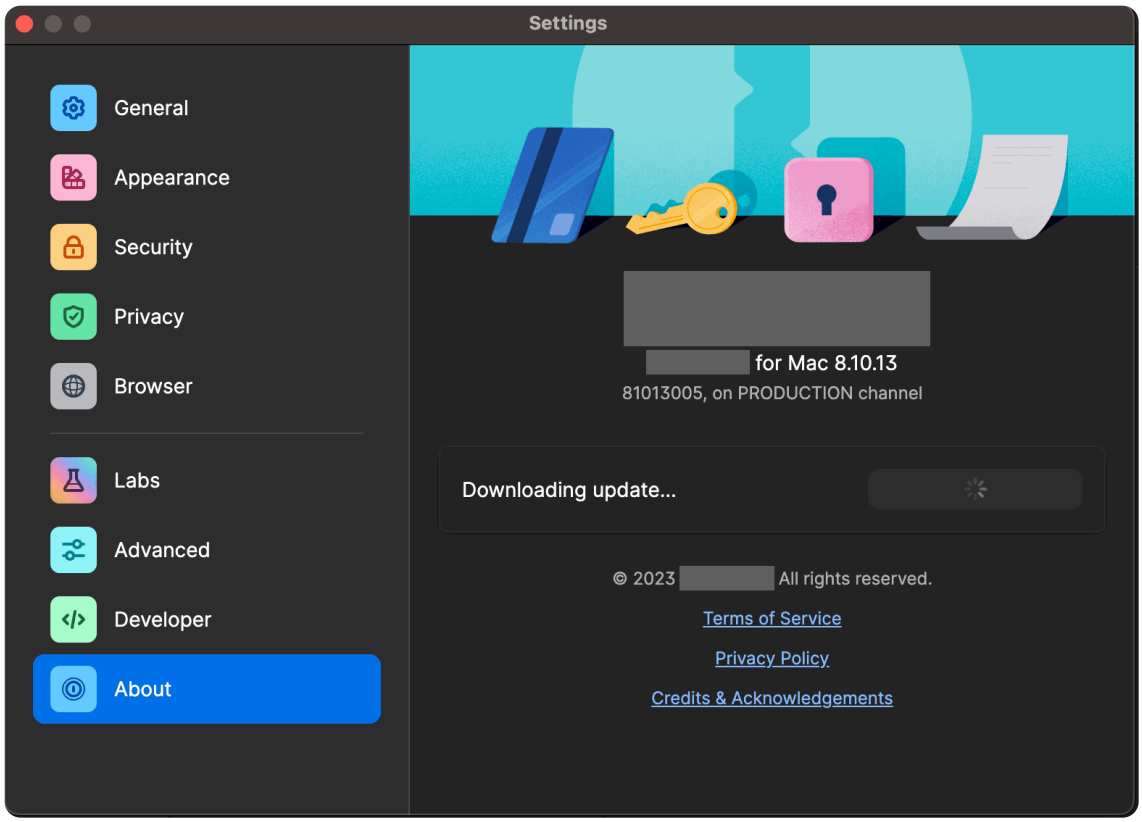
This is an absurd workaround! I should not have had to do this.

My US carrier (Verizon) does offer a downloadable application for macOS and Windows which should, in theory, allow me to flash the OS updates from my computer, instead of relying on the phone to download the patches. In practice, the Verizon app is even worse. Buggy, unreliable, and also insists on using its own in-app downloader to fetch the update files (sigh…).

I'm sure there's a way I could have gotten the images and flashed them manually. This is not an invitation for a bunch of Android enthusiasts to email me and explain bootloaders and APKs and ROMs and sideloading and whatever else is involved here. That's not the point. The point is — the mainstream tools that vendors ship are *hopelessly deficient* for users on slow Internet, and that's a bummer.

Example 3 - Small App Auto-Updater

A small desktop app has an in-app downloader for updates. Can you spot the issues?



Downloading an in-app update.

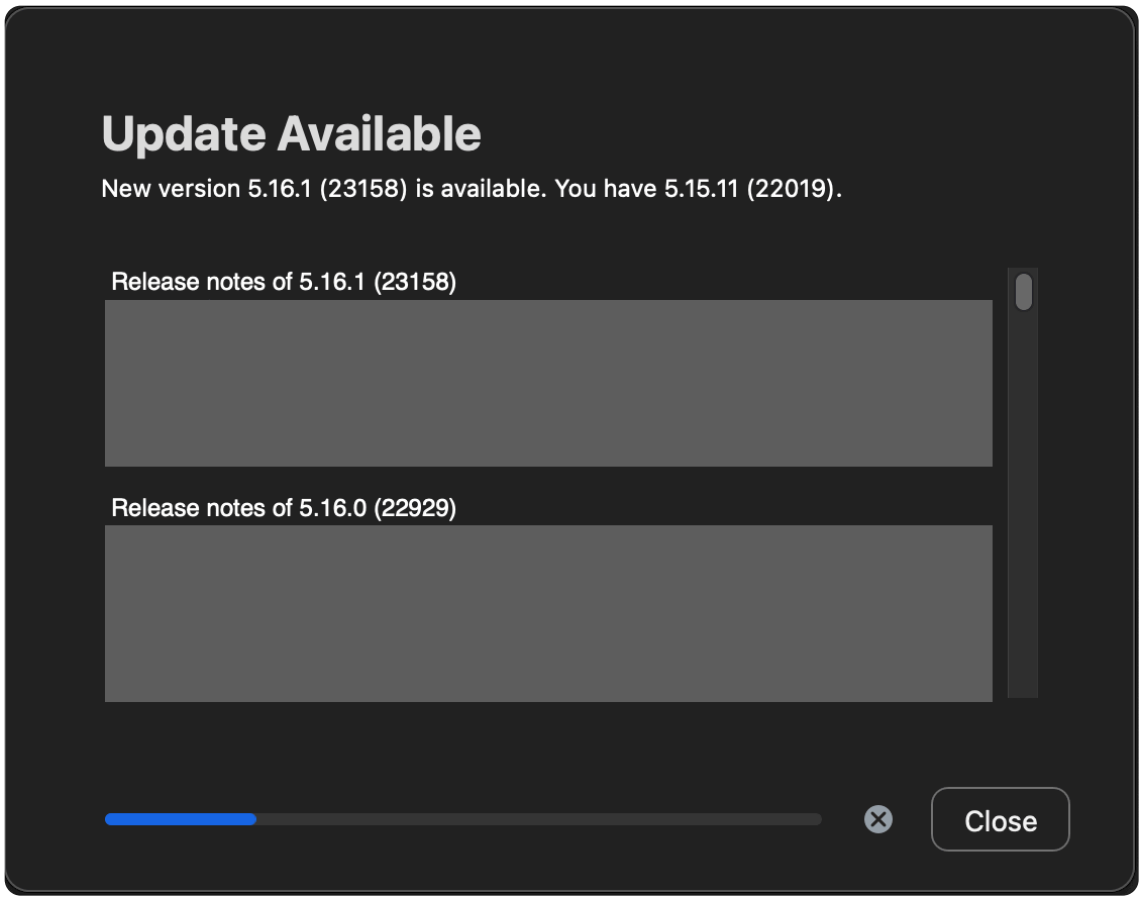
Let's count them:

1. No pause button.
2. No cancel button.
3. No progress indicator of any kind.
4. No speed / time remaining indicator.
5. No way to get at the underlying URL, so I can use my own downloader.
6. No progress tracking and no graceful resumption of an interrupted download.

This is actually one of my favorite desktop apps! It's a shame to call them out like this. A quick, easy way to make this MUCH better for users at the South Pole would be to provide a manual download link. Then, the developers wouldn't need to reimplement all the nice download features that my browser provides. I could just use my browser.

Example 4 - Yet Another App Auto-Updater

Here's another one!



Downloading another in-app update.

Let's count the issues:

1. No pause button.
2. No numeric progress / speed indicator.
3. No way to get at the underlying URL, so I can use my own downloader.
4. No progress tracking and no graceful resumption of an interrupted download.

It does have a few things going for it:

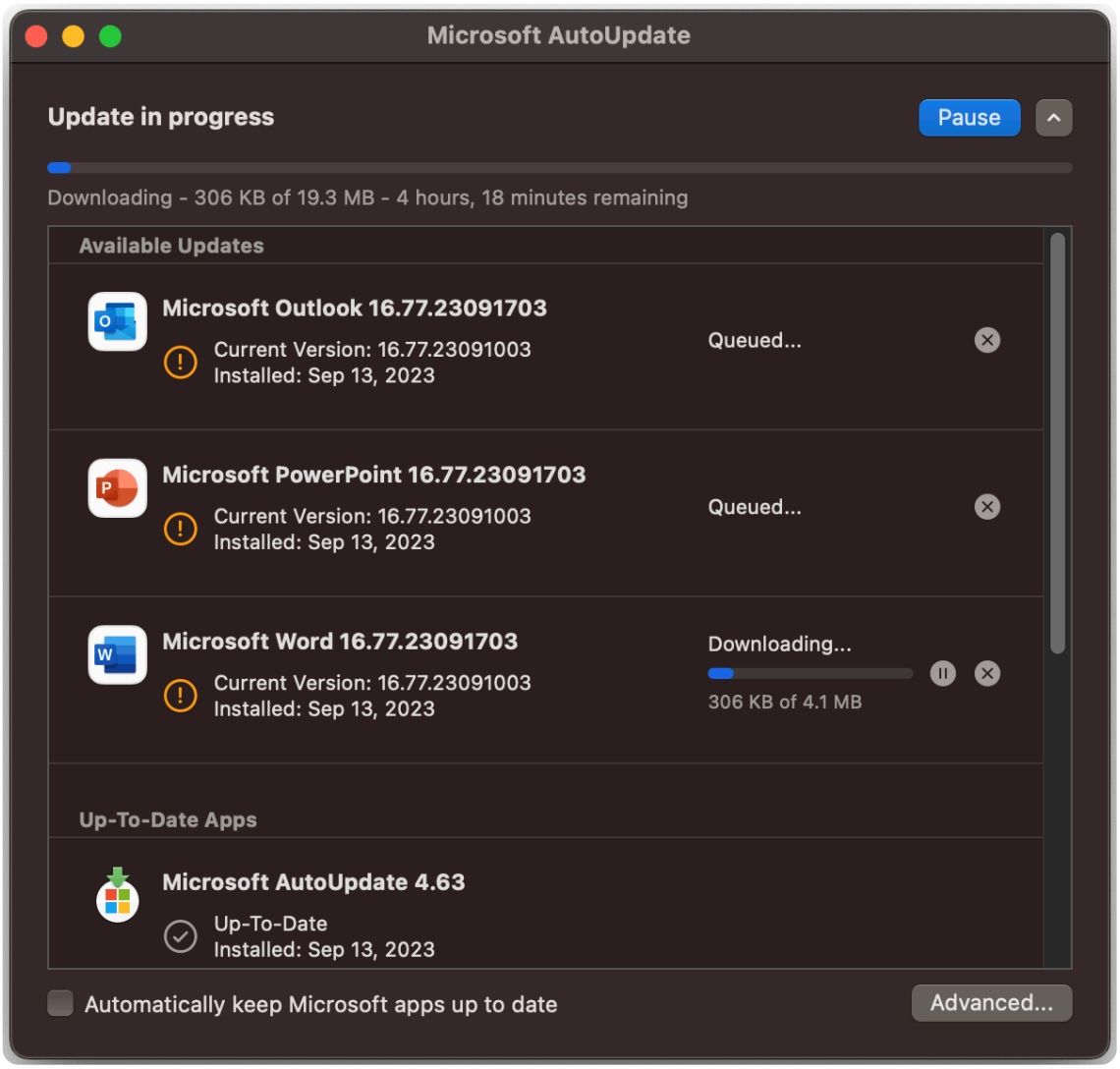
1. Cancel button.

2. Visual progress indicator.

But – overall, still a frustrating user experience for users with slow or intermittent Internet access.

Example 5 - Microsoft Office for Mac

Credit where credit is due – Microsoft has a GREAT auto-updater built into Office for Mac! Check it out:



Downloading Office for macOS updates at the South Pole.

Look at all these nice features!

1. Pause button!
2. Cancel buttons!
3. Progress indicator!
4. Speed and time remaining indicators!
5. Graceful resumption of interrupted downloads!

The only thing that could have made this better is a link to get at the underlying URL, so I could use my own downloader. But, given how good this interface is, I didn't mind using it, even at the South Pole.

Conclusion

I hope the examples I've shown in this post have been a helpful illustration of how minor oversights or under-developed features back home can become **major issues** in a place with slow Internet.

Again, I'm not asking that every app developer spend a huge amount of time optimizing for edge cases like the South Pole.

And I'm also definitely not asking for people to work miracles. Internet access at the South Pole, as of October 2023, was ***slow***. I don't expect immersive interactive streaming media to work under the conditions I described here, but it would be nice if apps were resilient enough to get a few bytes of text up or down. Unfortunately, what often ended up happening is that apps got stuck in a loop because of an ill-advised hardcoded timeout.

I hope everyone found this helpful, or at least interesting.

And thank you again to everyone who followed along with me on my Antarctic journey! I've been off-ice for about six months now, and going through my old posts

here have brought back fond memories.

I hope the current winter-over crew is doing well, and that everyone is enjoying the Polar Night. If the egg supply and consumption rate is the same as it was during Winter 2023, they should soon be finishing up The Last Egg.

I won't promise any more content, but I do have a handful of other half-finished posts sitting in my drafts. We'll see!

Keep reading

Older:

Redeployment Part Three

Off-continent after 446 days!

You may also like

Redeployment Part Three

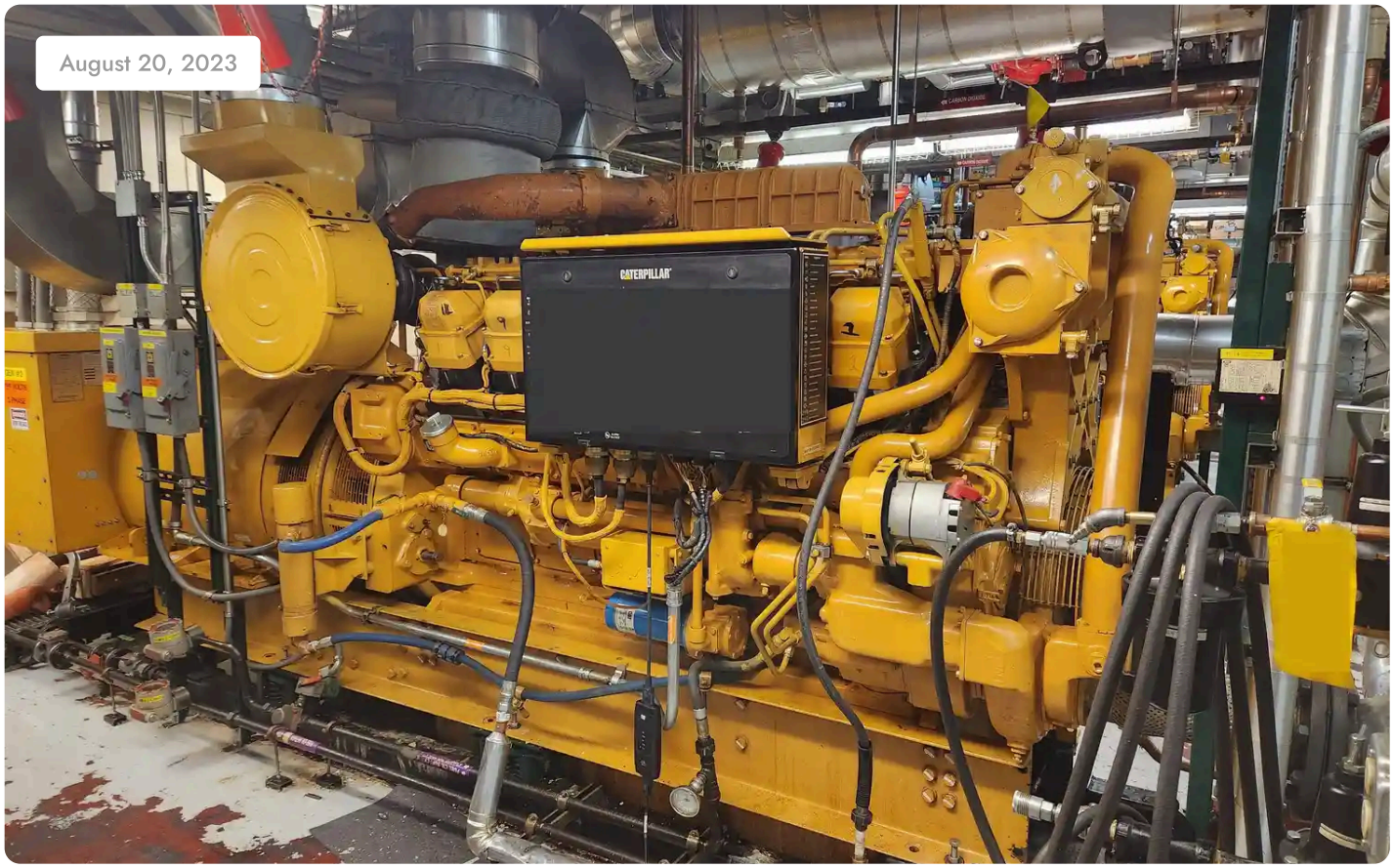
Off-continent after 446 days!

Redeployment Part Two

Station opening, and my flight out of Pole!

Redeployment Part One

Emerging from winter and preparing for our first flight!

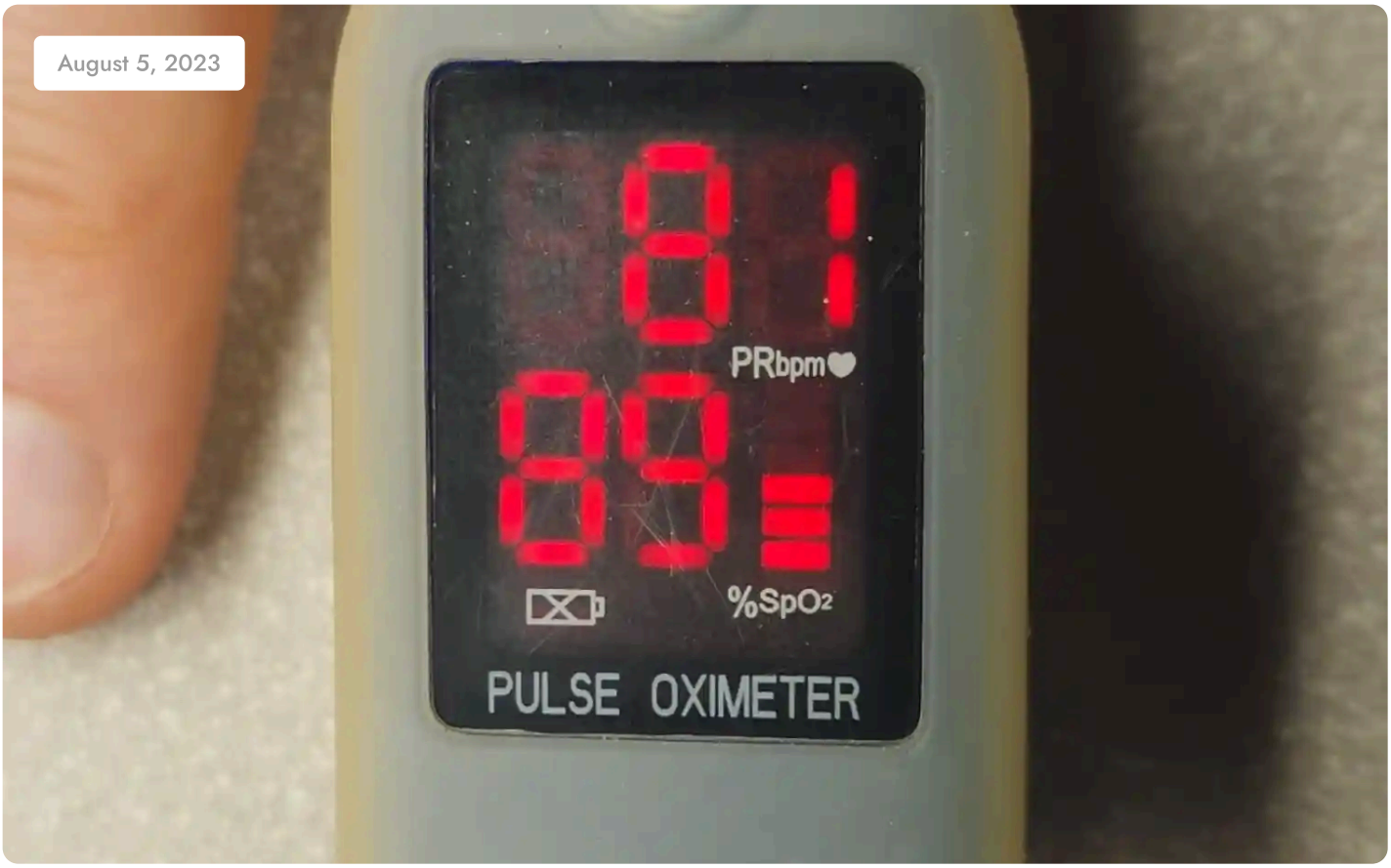


South Pole Electrical Infrastructure

Power generation and distribution at the South Pole.

Snowdrifts

4 days of blown snow into a doorway.



Pressure Altitude

Day-to-day variability at the South Pole.

brr