SSC23-II-01

Over-the-vacuum Update – Starlink's Approach for Reliably Upgrading Software on Thousands of Satellites

Akash Badshah, Natalie Morris, Matthew Monson
Space Exploration Technologies
1 Rocket Rd, Hawthorne, CA 90250; 310-363-6000
akash.badshah@spacex.com

ABSTRACT

Starlink operates the world's largest constellation of over 4000 satellites, all of which receive regular software updates to deliver new capabilities, improve reliability and performance, and maintain security. Updating the software across the constellation requires solving two core challenges: safely updating an individual satellite in the harsh environment of space, and orchestrating thousands of updates without impacting users of the system.

Variations on these problems have been addressed for large scale terrestrial compute systems by the broader software industry, and we have leveraged practices of that state of the art to develop a novel spacecraft software update system that delivers updates to the entire fleet of spacecraft on a rapid cadence.

We developed a fault tolerant update system that is resilient to a breadth of failure classes, ensures consistency across a satellite composed of many independent computers, and is autonomous and self-correcting across a variety of traditionally challenging space operations scenarios. We leverage that system to adopt software industry standard practices like canary testing and progressive rollout.

We have used this software update system to make over 200 updates to continuously deliver new functionality and improve performance of the fleet with no satellites lost due to failed software update.

INTRODUCTION

Starlink is a global broadband internet constellation, comprised of over 4000 satellites, each of which is contains many individual computers, some of which are multi-core Linux systems-on-chip and others are bare-metal low-power microcontrollers.

Starlink satellites were first launched in 2019 but given the considerable amount of time required to launch enough satellites to provide continuous coverage from LEO, public use was not available until late 2020. The time gap between first launch and operational availability required software update capability to the satellites, so that even the oldest launched satellites could continue to improve as the system developed. Even now, with millions of people around the world using Starlink every day, we are actively deploying new capabilities to the system and maintain security by leveraging weekly software updates to these spacecraft.

Updating thousands of computers while preventing device failure or regression to the end-user is not a new problem; it is well understood by software engineering companies that have servers deployed across the world, or end-user devices that need to be remotely updated like smartphones.

However, unlike updating these terrestrial devices, physical access to recover from a bad update is not possible with spacecraft, even as a last resort. Additionally, the harsh environment of space means things like radiation induced CPU faults or memory corruption are experienced more frequently, and satellites will nominally go hours at a time without connecting to our ground systems. These challenges specific to the space environment require a robust and autonomous update procedure to allow adoption of terrestrial practices without compromising speed or scale.

FAULT TOLERANT UPDATES

First it is helpful to define the concept of fault tolerance. To be fault tolerant, a system must maintain operating capability in the presence of a fault. A fault can be a temporary or permanent failure in hardware, or an error in the software itself. Here are some specific scenarios that can (and do) occur on the satellites:

1. A single bit in memory is flipped – rendering that data untrustworthy
2. A software process crashes
3. An emergency situation requiring an urgent reboot

These issues can occur due to the radiation environment in space; in addition, software bugs can cause processes to crash or other misbehavior that results in the software acting contrary to our desires. In the presence of any of the above, a software update could at best fail to execute; at worst, it could permanently corrupt the computer memory and render further updates impossible. When we considered how to update the software on a single computer on a single satellite, we designed around the following paradigm: the satellite must fall back to a known-good software image in the event of a failure.

We generally think of our satellite computers as glorified servers in space. Each computer has a processor and some non-volatile memory in which we store the software program that runs on the processor. As most readers will intuit, the industry-standard way to provide data redundancy is to keep multiple separate copies. We chose to divide the non-volatile memory on a satellite computer into redundant partitions, which we'll refer to as A and B.

A/B Boot

We used these redundant partitions to implement a concept of operations for a "self-healing" software boot system, called "A/B Boot." The key questions that the design needs to address are:

1. To which partition will the computer write a newly received update?
2. How will the bootloader decide the partition from which to boot?
3. How will the computer detect if the new software is functional?

For the first question, consider the following system: The computer always writes to partition A and only falls back to partition B if A has failed. This would be suboptimal because it forces the computer to write to the partition from which it has most recently booted. The software that is currently running, by definition, is the last-known-good software. We don't want to write over it until we have established a new last-known-good. In the concept we designed, the computer will always write its new software to the opposite partition.

Suppose that version 1 is in partition A, and we have just written version 2 to partition B. We want to reboot the computer and boot up into partition B. How does the computer know which partition to boot from? We could set a flag that tells the bootloader which partition to use when it reboots. Let's say we set the flag to B, reboot, and enter partition B. This is obviously an issue; if version 2 has a bug and crashes, how will it update the flag to tell the bootloader to return to partition A? We could add some re-try and fallback logic, but this is overcomplicating. The simplest method is to program the bootloader to always toggle between partitions on every power cycle.

In the scenario where version 2 has a bug and crashes, we want the computer to return to version 1. We need a mechanism to reboot the computer in the event of a crash, so we can take advantage of the alternating boot partitions. Since the satellites are in space, there is no option to send someone to the server room and reboot the computer by hand. With no software running, the satellite has no way to communicate with an operator on the ground. To ensure that the computer is rebooted if the process crashes, we implemented a hardware watchdog that toggles the power reset when tripped. The software is responsible for "petting" the watchdog; if the process crashes, the watchdog will trip. This feature also allows us to neatly handle other types of software bugs by intentionally crashing the process to trigger the watchdog. You might say this is our answer to the classic troubleshooting question, "Have you tried turning it off and back on again?"

To summarize A/B boot: the computer will write new software to the opposite partition. It then executes a reboot to load that new software. If the new software crashes, a hardware watchdog will trip and reboot back to the known-good software.

CONSISTENCY ACROSS A SATELLITE

Each satellite is a collection of computers (referred to as "nodes") in control of various subsystems on the vehicle. The earliest satellites have 80 nodes, the newest have 237 nodes, and future designs approach 500 independent nodes. These nodes have bidirectional communications interfaces with one another to achieve coordinated tasks. For example, the flight computer sends commands to a remote node to change the angle of the solar array and receives sensor inputs from that node describing the state of the solar array. The A/B boot system ensures a single computer receives an update and remains resilient to faults either in the process or contents of that update. The primary mechanism for attaining that resilience is the reliance on a known-good backup version, but this creates a new challenge: how do we guarantee that every computer on the satellite will be running compatible software, such that the bidirectional interfaces continue to operate?

If every computer follows the A/B boot scheme independently, some nodes might boot to a new version

while others crash and fall back to known-good. And even if all nodes might successfully boot to a new version, a single node might subsequently experience a radiation-induced power cycle. A/B boot scheme changes the boot partition unconditionally on each boot, so that node would boot up into the previous "last known good" after rebooting. The independence of each node, along with the hundreds of interfaces between them, means we must solve the problem of "version tears" between nodes where each side is expecting a different version of the interface.

One way to solve this problem would be to use a versioning scheme on the interfaces that allow for continued communication in both a "forwards" and "backwards" compatible way. This principle is applied across the internet, especially in places where updates cannot guarantee coordination (like apps on smartphones communicating with a central server). To achieve this, however, a significant amount of overhead is usually required in both the data on the wire and in the logic handling these different versions. These are constrained resources in an embedded vehicle environment. This is also not particularly useful in the context of nodes on an individual vehicle where, unlike in the smartphone-server relationship, we don't expect them to operate in this version-tear state for extended periods of time.

The solution we chose was to prevent these version tears from occurring in the first place. We do this by delegating a "central" node (the flight computer) as the arbiter of what software *should* be running the vehicle. The central node receives its software image, and the image for every other node, and uses the full A/B boot scheme. Every "remote" then inherits its configuration from the central node.

First the central node writes the new images to the opposite partition and reboots the entire vehicle. Then as each remote node powers up into its bootloader, it checks with the central node which image it should be running. The remote node compares the hash of the image in its nonvolatile memory to that of the image the central node has for it. If the hash differs, the remote node fetches the image from the central node and proceeds with the bootup.

This "hub-and-spoke" distribution model allows the entire vehicle to be protected by the A/B boot property by inheriting it from the central node and avoids the pitfalls of coordinating hundreds of possibly independent interfaces with complex and cumbersome compatibility logic.

AUTONOMOUS UPDATES

After having developed the primitives of a fault tolerant update system (using A/B boot) and an atomic coordination mechanism across the vehicle, we can begin to treat software updating satellites in space in a similar manner to updating servers on Earth. One more paradigm we need is a system to orchestrate the update, similar in concept to the automatic installer system your smartphone or desktop computers use. This system is responsible for downloading the update and determining when to update each partition.

On its face, our software installer is very similar to those terrestrial systems. It begins with a process on the satellite that checks in with our management servers on the ground to determine if new software is available. If so, the central node will proceed to download that update into memory, and once the download is complete and the integrity of the downloaded binary is verified, it will commit that to non-volatile memory (using the A/B principle described previously). Once it is done committing it, the vehicle will reboot into the new software using the "hub and spoke" method and then begin executing on the new software.

On top of this standard model, our system has some features that are uniquely valuable in the space environment, where high energy particles cause everything from bit flips to unplanned reboots. We generally want to minimize the amount of time where the A & B partitions have different software, so that an unplanned reboot doesn't have the unintended consequence of effectively rolling a satellite back to older versions. On the other hand, we need to ensure a sufficiently high bar to clear before we commit a version of software to both partitions.

We balance these priorities by requiring that any given version of software can only be written to both A and B partitions if that software can provably execute *another* software update to a partition. The mechanism for enforcing this requirement is that we don't allow direct copying from the A partition to the B partition (or vice versa). Let's say that the B partition is running version 2, while A is running version 1. Clearly version 1 can fetch new software and write it to partition B. For version 2 to commit itself to the A partition, it must fetch and validate itself from our ground systems. This guarantees that the new version is even capable of establishing a secure and stable connection to the ground. Once we have asserted this requirement, we know that however bad the new software version is, it can at least update to a different build, presumably one that fixes any bugs. If a version fails this test, perhaps because it introduced a bug that disables contact with the ground, it will be unable to write itself to the "other" partition. And after some reasonable amount of time trying and failing to contact the ground, it will reboot into known good software.

Another way in which our system differs from terrestrial systems is that we expect that the software itself will be corrupted at rest every now and then. A known good software version in one of the partitions may be corrupted by radiation-induced bit flips. The central node monitors for this condition by comparing a hash of the image located in non-volatile memory to the hash expected by the ground, so if the hash differs (even if there is no planned update) the satellite will fetch the "update" and write it, thus "healing" itself. The remote nodes, as mentioned above, have a similar protection since they compare their versions to the central node's versions on every boot.

One way in which our system might seem to be different than terrestrial update systems is that the connection may be intermittent and unstable, especially as the satellite hands off between ground stations or passes over the ocean. However, we have determined that standard HTTPS download processes are reasonably resilient to this kind of intermittent disconnection, and with suitably tuned retry settings can pause and resume downloads seamlessly.

Finally, given that these satellites are continuously in use to serve internet to customers of the Starlink product, we want to avoid reboots for known reasons (like software updates) at inopportune times. We have built a coordination system for satellites informing our ground planning algorithms that it needs to reboot, and waiting until the planner has intentionally offloaded all users from the satellite before it reboots. This cannot protect us in the case of an unexpected reboot, but the vast majority of planned weekly updates events can be proactively coordinated in this way.

Taken together, this installer architecture means that all an operator on the ground needs to do to update a satellite is change the configuration in our management layer, and the satellite will autonomously recognize the need for an update, perform it without impacting the broader system, and assert the basic safety of that new build before fully committing to it.

ROLLOUT ACROSS THE FLEET

With such a large constellation of assets, and the vehicle safety ensured by our update system, in the normal course of our operations we are generally more concerned about marginal degradation of system-wide performance because of software updates rather than adverse behavior on individual satellites. Here again, we borrow from the state-of-the-art techniques applied to servers or remote device management on Earth.

Generally, to screen for low probability issues, most large-scale update systems will start by updating a small "canary" population of devices and monitoring key metrics on those devices to ensure overall performance is in-family (or improved) relative to baseline. We adopt the same technique by randomly sampling a small set of our satellites, updating them by leveraging our safe & autonomous system, and monitoring how those satellites impact end user experience.

Once a build has been screened against this canary process, we begin a process of rolling out more significantly across the fleet. As with other industries, we adopt a process of "progressive rollout" – where an update is rolled out to the whole fleet over time, as opposed to having everything update in a short window. This allows us to continue screening for ever lower probability bugs and prevent adverse system level effects of changing too much at once. If a regression is detected in software at any point in our analysis & monitoring, we can change the ground-based configuration and rely on the autonomous satellite update system to roll back to a known good.

CONCLUSION

In this paper, we've described our adoption of terrestrial software management practices to treat a constellation of satellites very similarly to a fleet of servers in data centers. We developed an update architecture that is resilient to the unique constraints of spacecraft but requires low operator intervention. In so doing, we have enabled a rapid pace of development without sacrificing safety or user experience. Other operators considering the development of large-scale spacecraft systems should use our experience as guidance to lean into the existing state of the art for terrestrial software systems rather than developing bespoke technologies uniquely suited to space.

ACKNOWLEDGEMENTS

Many brilliant engineers were needed to develop the software update systems described in this paper. In writing this paper, we the authors merely put to words their considerable efforts. We wish to thank every SpaceX engineer past and present who has contributed in some way to making these efforts successful.