SpinDance
Collaborate. Iterate. Succeed.

User Experience (UX) Centric IoT Software Update

2016 IoT Software Update Workshop

Eric Smith, Mike Stitt, Robert Ensink, and Karl Jager
SpinDance, Inc.

Revision: B
6 June 2016

ABSTRACT:
This position paper describes the software update user experience (UX) needed for new IoT devices to be better than legacy electro-mechanical devices, and thus, achieve widespread adoption and ubiquity. We develop requirements that describe pleasant and quiet IoT devices that just work. These requirements then drive software security approaches, update protocols, software architectures, and silicon. We describe oracles that automatically control software updates and device configuration. We describe redundant loader approaches for reliability and availability during software updates. Great user experience requires silicon capabilities that increase cost, but are needed for IoT devices to be accepted into our lives.

KEY WORDS:
User Experience (UX), User-Centered, Software Update, Firmware Download, Security, IoT, Things, Interoperability, Firmware Download Oracle, System Provisioning Oracle, Loadable Modules, Silicon, Public Key Infrastructure (PKI), Symmetric Encryption, Asymmetric Encryption, Authentication, Authorization, Irreversible Commands, Boot Selector, Boot Loader, Application Slot, Application Loader, Highly Available, Highly Reliable, Pleasant, Quiet, Automatic Software Update, Automatic Firmware Update, Firmware Over-The-Air (FOTA) Update, Decision Fatigue, Interruption of Service.

1. Introduction

We are pleased to provide this position paper on user experience (UX) centric software updates. By considering the user experience needed when things are ubiquitously connected to the Internet, we can discover software update requirements, which then drive software update protocols, software architectures, and even silicon requirements.

2. User Experience - Decision Fatigue

Decision fatigue is real. When we make more and more decisions in our day, the quality of our decisions decrease and decrease.

Each person has a limited number of things that are important enough to them to care about their precise configuration, tailoring, and maintenance. For some, such things may include their mobile phone, television, laptop computer, automobile, and perhaps, their coffee maker. We would prefer to limit the number of things we need to maintain. We don’t want ubiquitous smart things to result in ubiquitous unnecessary decision making.

Pleasant and quiet IoT devices need to just work. We will invite them into our lives when they do their jobs without getting in the way and without requiring us to think about them.

Software updates must be automatic, requiring minimal user participation and decision making.

3. User Experience - Interruption of Service

Despite decision fatigue, there will be some circumstances where we might want to make a decision about the configuration of our connected things. When we’re holding a party or giving a presentation, we don’t want our plans interrupted by a scheduled loss of service. Rather, we might prefer to schedule our software updates at convenient times.

Software update systems must allow users to prevent interruptions of service when they interfere with events in their lives. Our preference is for highly available systems that automatically schedule and apply software updates in such a way that the risk to our schedules is minimized and that the use of our things is not interrupted.

4. Wandering Cats - Interoperability is Hard

IoT system administrators would like to be able to simultaneously control and coordinate changes to things and the other things to which they connect (e.g., other devices, clouds, mobile/web/desktop applications, manufacturing equipment, etc.). We would like to update everything at once, so that we can minimize interoperability challenges. But, we can’t. Some things are in stock at the manufacturer, some are in distribution or in stores, some are unpowered with users, some are powered, but their Internet connection is down. We can’t herd the cats. We can’t even lay out a saucer of milk for the cats to gather around. In a large system of cats, nobody is in full control of when new cats, or new versions of cats, are born into the system. The software updates are at the mercy of the combined actions of all of the people who possess the things and influence their paths to the Internet. We will always have some things connecting to our systems that are running old versions of application software and software update software.

We can derive at least three requirements from our wandering cats. 1) The software update system must be resilient to changing components; it must consist of components that are compatible with past and future versions of components in the system. 2) We must select interoperable sets of application software; the software update system must deploy and connect versions of things and clouds that are compatible with each other. 3) We must monitor software update deployments and firmware operation, so that we can understand whether or not updates are being successfully delivered and consumed, how well devices are operating in the field, and whether or not system or device anomalies are present.

5. Herding Cats - Oracles

At SpinDance, we use two entities to herd the cats. To indicate that all the cats should listen to the entities, we call them Oracles. We have: 1) a Firmware Download Oracle that specifies what version of software to load and 2) a System Provisioning Oracle (similar to a Device Manager) that specifies which things should connect to which clouds.

6. Security and Trust

There is widespread agreement across the IoT industry that IoT systems must be secure. And, as an industry, we are correct, when we project Internet security challenges into IoT. Reliability, security, and privacy must be automatically and seamlessly woven into systems and their user experience; our things must be secure without requiring our users to fuss over or configure them. Security must be approached holistically, from a system-wide perspective: breaches are inevitable and must be detectable, isolatable, recoverable, and fixable. We must continuously monitor our IoT systems for security breaches.

We prefer security systems where things and clouds both mutually authenticate (each verify the authenticity of the other's identity) and verify that they should trust the commands and data of one another. The recipient of a command must both authenticate the sender's identity and verify that the sender is authorized to issue such a command. We prefer public key infrastructure (PKI) to authenticate identities, verify authorizations, and establish symmetric session keys for encrypted communication channels.

A "software update" command is an irreversible command (or at least a hard-to-reverse command), in that it causes the contents of non-volatile memory to be altered. In most systems, the "software update" command is not the only irreversible command. A common example is associating a device with a new user account; when this happens, data associated with the previous account on the device is lost. Another example is updating certificates or other credentials.

When designing a generic software update system, it is common to place an authentication, authorization, and encrypted communication solution into an update system separate from the application; however, the application typically requires the same level of authentication, authorization, and encryption, because of the irreversible commands it also performs. To minimize memory requirements, and to allow the update to occur without interruption of service, in our systems, we have placed the software update system within the application-oriented secure communication system (i.e., the update system is part of and integrated with the application, not in a separate image [e.g., boot loader], and uses the same communications channel).

In systems where there is a mesh of similar devices with limited communication bandwidth, we can envision a mixed solution, where the command and control for an update is within the application's main communication channel, and the transport of the software image is in another channel tailored to the limited communication bandwidth.

7. Firmware Download and System Provisioning Oracles

We believe things will verify that they trust the clouds made by their manufacturers, or other organizations the manufacturers designate, that keep track of what software should be loaded in what devices. There will be clouds trusted by the devices to control software updates. The situation is too complicated to be the responsibility of the user.

Pleasant and quiet devices will trust the Firmware Download Oracle which defines which software version should be used. Users will not tend to specify the minor version and patch fixes, just like they don't control individual versions of device drivers within operating systems for desktop computers or mobile devices. More and more, we let the mobile device and operating system vendors automate the entire software update system. IoT software update should be fully automatic.

Smart, connected devices are often designed and built with multiple components from different vendors that each may receive software updates (e.g., Wi-Fi, cellular, Bluetooth, application processors, etc.). The software update solution must accommodate these products: allowing software updates for individual components or the device as a whole, obtained from a single or different sources. Companies invest significant time designing and integrating components to deliver a service and experience: one component's updated software may subtly or catastrophically break the device, degrading the user's experience and brand perception. Updating devices is hard and dangerous; therefore, we prefer update systems that allow device vendors, or their delegate(s), to specify which software resides in relevant components of their devices (e.g., via a manifest or device configuration), to determine when their devices are updated (e.g., initiated via the cloud), and to control how updates are obtained and applied (e.g., via the application processor on the device). We expect the same for the software in other parts of our systems: that someone is in control that ensures everything works together well. Ultimately, the vendor of a given IoT device always is responsible for providing a fully integrated product.

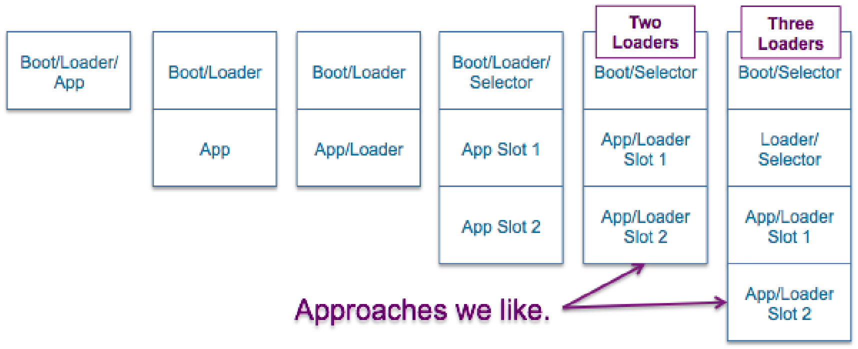
During development, we move IoT devices from one cloud to another. We move them between our continuous integration, quality assurance, alpha test, beta test, and production clouds. We move them from cloud to cloud under control of our System Provisioning Oracle. We suspect that we might use the System Provisioning Oracle to control which production systems IoT devices connect to as they change software versions.

8. Costs

Are we willing to pay extra for the redundancy necessary for highly available, secure, and interoperable things? Because price increases result in reduced sales, our first instinct is to say, "No." But, with our current set of largely electro-mechanical things, we are used to secure, highly available, highly interoperable things. Things like door handles, latch bolts, strike plates, light switches, and light sockets don't tend to fail. They last decades or centuries. They don't require maintenance, security patches, or software upgrades. For smart things to become widely adopted, their security, maintainability, and reliability must be similar to what existing electro-mechanical things achieve, with the same "just works" user experience, regardless of what's actually happening "under the hood."

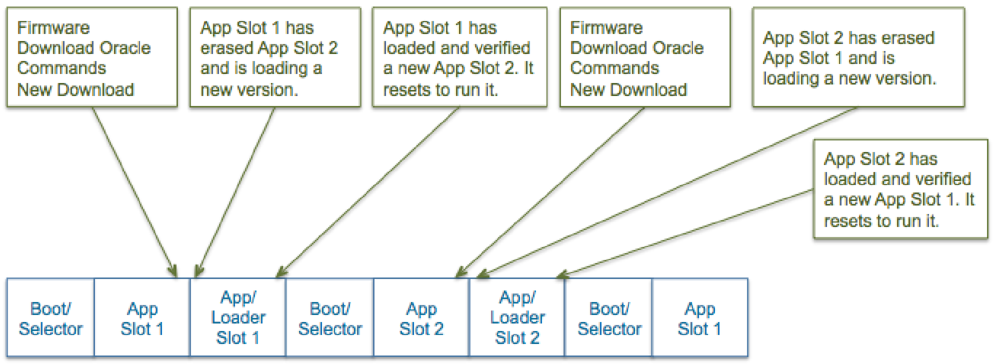
9. IoT Device Loadable Module Approaches

Things must be highly reliable, highly available, quiet, pleasant, and update automatically to deliver the best user experience. To accomplish this, IoT device software must be partitioned into multiple loadable modules, as shown below.



The "two loader" approach allows software download without interrupting service, while allowing recovery from most faults (e.g., a corrupted application image or an image that fails to run well). In this approach, the Boot Selector simply selects which Application/Loader to run, and each Application Slot can download overtop the other and then reset the device to let the Boot Selector run the new slot. Our intent is to make the Boot Selector so small that it will never need to be replaced. Each loadable module has a role to play with failure detection and recovery; however, the Boot Selector is ultimately responsible for selecting the appropriate image to run, based upon version, fault, and runtime information available to it (e.g., self-reported validity of image, hard faults per unit of time, CRC, diagnostic results, etc.).

Loader approaches have an infinite loading loop: for as long as the device is in use, it is running and loading generation after generation of applications and/or loaders. Unfortunately, in a "two loader" approach, when a loader has erased the other slot, we are a single permanent fault away from breaking the loop (i.e., "bricking" the device). This risk may be mitigated by thorough and appropriate testing.


Firmware Download Oracle Commands New Download
App Slot 1 has erased App Slot 2 and is loading a new version.
App Slot 1 has loaded and verified a new App Slot 2. It resets to run it.
Firmware Download Oracle Commands New Download
App Slot 2 has erased App Slot 1 and is loading a new version.
App Slot 2 has loaded and verified a new App Slot 1. It resets to run it.
Boot/ Selector
App Slot 1
App/ Loader Slot 1
Boot/ Selector
App Slot 2
App/ Loader Slot 2
Boot/ Selector
App Slot 1

To further reduce this risk, this leads us to prefer the “three loader” approach. This approach extends the “two loader” approach by adding a fallback Loader/Selector that provides a separated means to load Applications/Loaders; in this way, we always have two pieces of software able to perform a load, one of which may recover from more subtle application defects.

10. Silicon Implications

Great user experience is necessary for IoT devices to become widely adopted, but will increase IoT silicon costs. To build reliable, available systems with three loaders, we prefer processors with instruction sets tailored for efficient position independent code. We prefer processors with full-featured Memory Management Units (MMUs) that can fetch from any part of flash, while simultaneously writing or erasing any other part of flash; however, we have implemented such approaches on the larger set of ARM Cortex-M class devices with Memory Protection Units (MPUs) that do not fully support such flash operations, as well as on devices without any memory protection at all. We prefer silicon devices with lots of flash divided into small sectors. Flexible security approaches require a surprising number of computationally expensive asymmetric cryptographic operations, which need to be performed quickly for a great user experience. While it is not required for all applications, and it is possible to do everything in software (and we have), the small processors in IoT devices need hardware acceleration to efficiently perform symmetric and asymmetric cryptography.