TesterHome | IT大咖说

Creating OpenSTF

MTSC 2018

About me

- Hi, I’m Simo Kinnunen
- Created OpenSTF with Gunther Brunner in 2013
- Check out github.com/sorccu for some of my past work

What is OpenSTF?

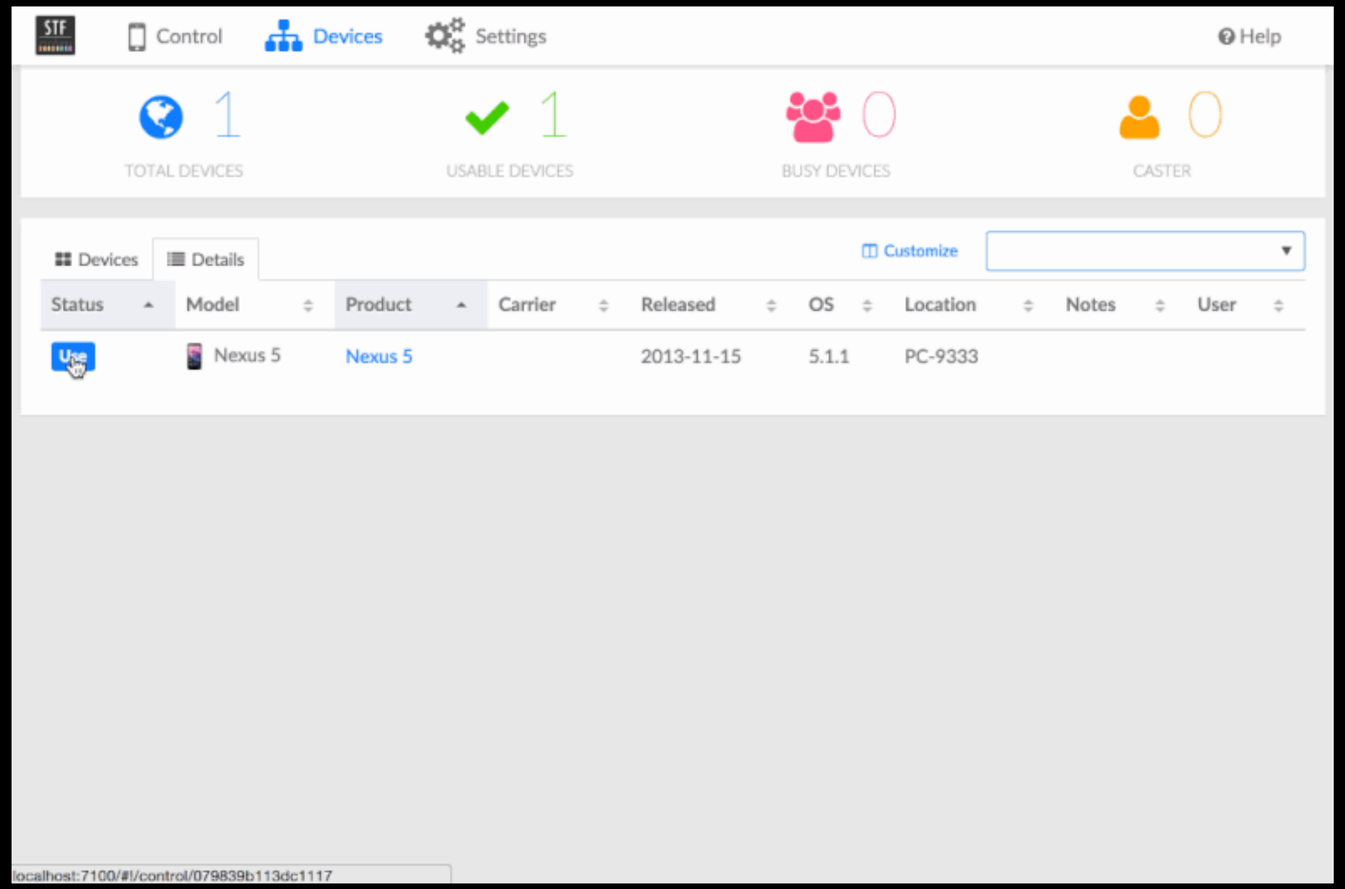
What is OpenSTF?

- OpenSTF is an **on-premises**, **free**, **open source** device management platform
  - Apache 2.0 license
  - 7,500+ stars on GitHub
- Compare to e.g. AWS Device Farm, Google Cloud Test Lab

What does it do?

- Provides effortless remote access to your test devices
  - Remote control for manual testing and debugging
  - Remote monitoring and recovery of test runs
- Integrates with automation and development tools
  - Appium, Jenkins, Android Studio, ADB, etc.

This is what it looks like



This is how you use it

Why use OpenSTF?

- Scales to thousands of devices
- Runs on commodity hardware
- Easy to use (though perhaps not to set up...)
- Open source

OpenSTF can be good for you

But, I’m not here to
sell it to you

Our focus today

- To discuss how OpenSTF was created and how it evolved over time
- With the power of hindsight, evaluate how we've done and what we could do better now
- Provide insights into things we might do in the future

To understand the
choices we've made...

...we have to go back.

Origins of OpenSTF

Origins of OpenSTF

- OpenSTF originally started as a test automation platform back in 2013
  - It's called **Smartphone Test Farm** after all
- Remote control became a priority later
  - Realized it was a better fit for our use case at the time
  - Decided while building a feature to record test runs

Landscape in 2013

Android in 2013

- Android 4.3 had just been released
- Android 2.3 was still alive and supported
- Samsung Galaxy S2 and S3 were popular phones
- LTE wasn’t supported on all phones
- Surprising variety; many manufacturers have since exited the business

Mobile testing in 2013

- Very little tooling existed
  - Tools that did exist were mostly immature and unstable
- Very little documentation existed
- Almost nothing was open source
- As a result, few tests were actually written

Our challenge

Our challenge

- To create a system that supports every single Android device
  - Including Android 2.3, of course
- Make it usable on any machine without having to install anything
- Integrate with existing tooling
  - We don’t want to redo everything

Initial prototype

- An initial prototype was running within a few months
  - Used adb’s undocumented framebuffer command for screen capture
  - Monkey tool for input
  - Basic adb commands glued together

It sucked.

What didn’t work

- Unpredictable USB disconnections
- Physical devices were a huge mess
- Only about 10% of our devices were “fully functional”
  - Touch events didn’t work at all on some devices, or certain views like the home screen and settings were not functional
    - Obviously no multitouch
  - Super slow

Solving the hardware side

Disconnections

- Modern Intel CPUs assign relatively few resources to the built-in USB host controller
  - Practical limit of around 8-12 devices per machine
- Most USB hubs don’t provide enough power to both charge the phone and keep a stable data connection
- Sometimes USB cables break
- ADB is horrible on macOS

CPU issues

- Our solution:
  - PCIe cards with built-in USB host controllers
- Other options:
  - More machines
  - On some machines you can enable a hidden second USB host controller
  - Use another CPU vendor (though that may bring other issues)

USB hub issues

- Our solution:
  - A Battery Charging 1.2 compatible USB hub (up to 1.5A per port) for most devices
  - A spec-violating 1A hub for older devices
  - A relatively expensive, programmable USB hub
- Other solutions:
  - Plug directly into a machine. Usually doesn’t help much.
  - Reduce power usage (e.g. screen). Almost never works.
  - Fewer devices per machine

Operating system

- Just use Linux, you'll save a lot of time.
- Avoid macOS at all costs. It works until it doesn't.

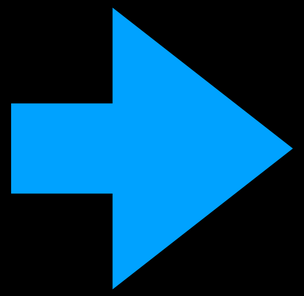
Bandwidth limits

- ADB’s USB bandwidth used to max out at roughly 5MB/s regardless of device
- A single Full HD RGBA framebuffer is about 8MB
- Clearly, we couldn’t even do 1 FPS even if the capture itself was relatively fast (it wasn’t)

Organization

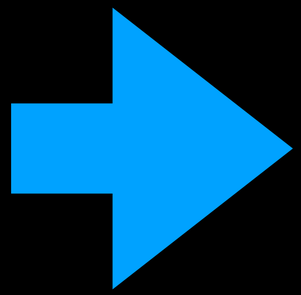
- It turns out that nobody sells a device shelf, especially for 100+ devices
- Our solution:
  - Design a custom shelf by ourselves


ORICO
USB 3.0 SUPER SPEED HUB
docomo
SONY
XPERIA
Sony Ericsson
Shift
Ctrl

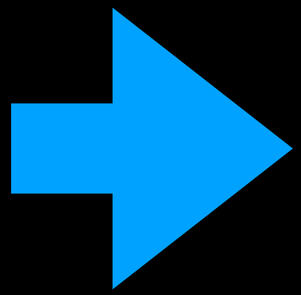


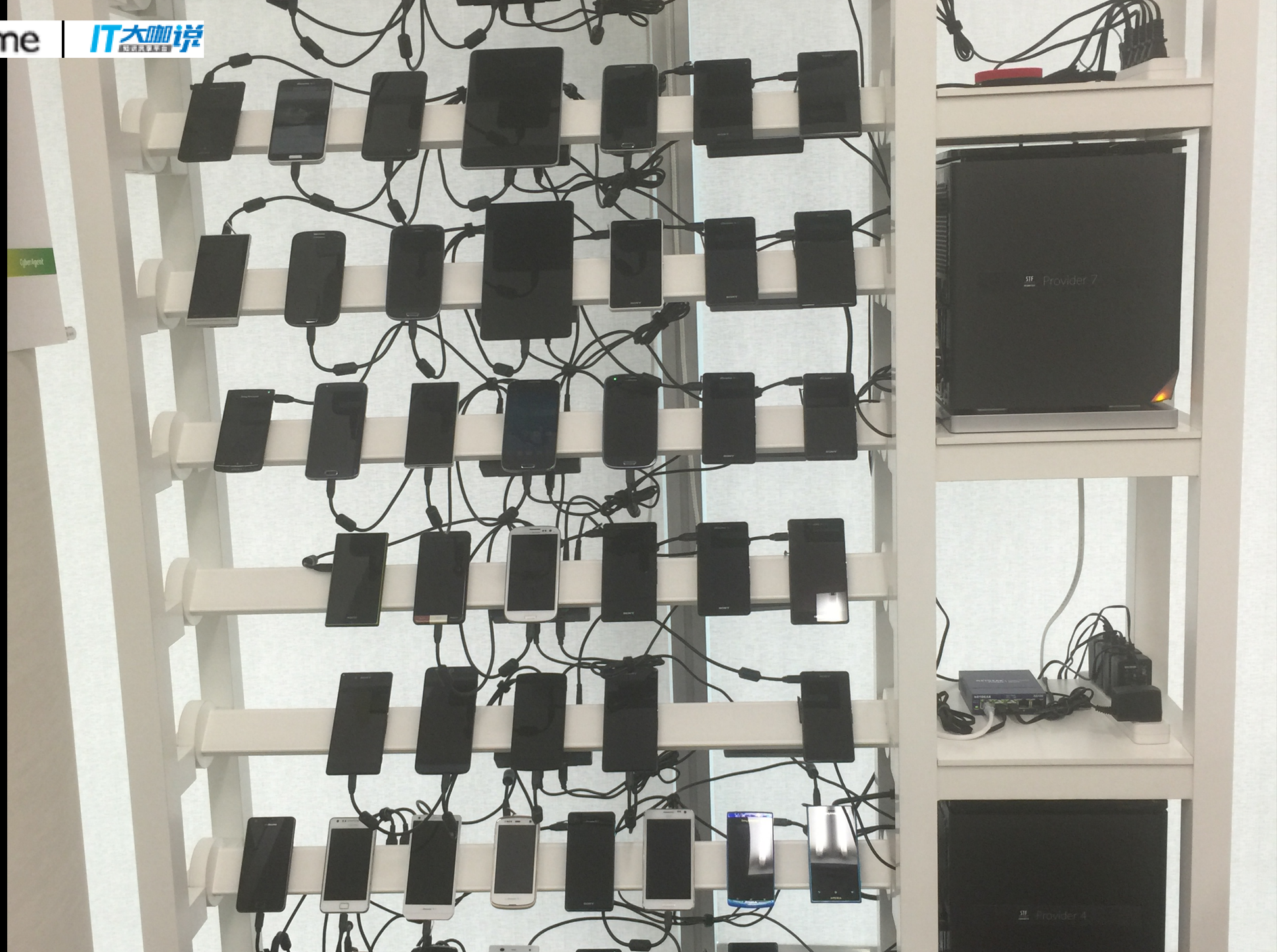

CX430 M
plugable
plugable
plugable
plugable


NETGEAR
CX430M
plugable




22 38
SAMSUNG
LG
SONY
HTC





Software side

Reliable touch input

Reliable touch input

- Monkey is far from reliable
- Turns out that the adb **shell** user belongs to the **input** group
  - The input group has r/w access to /dev/input/event*
  - One of those input devices is the touchscreen, and writing to it will generate touch events
  - Read events and describe devices with the built-in **getevent** command

Reliable touch input

- Our solution: minitouch, a small, custom NDK-built C program
  - Automatically detects the correct touch device
  - Exposes a socket interface, translates commands to raw Linux kernel touch events
  - Supports real-time multi-touch
  - Additional requirements on some devices, e.g. Xiaomi
  - Fixed >90% of the non-functioning devices we had

Reliable touch input

- Minitouch still works well and supports almost all devices
- Viable options in 2018:
  - Grant **INJECT_EVENTS** permission via adb and use Android Java APIs

Fast screen capture

Fast screen capture

- Originally used adb framebuffer, wasn’t working out
- The adb **shell** user belongs to the **graphics** group
- The graphics group is allowed to connect to SurfaceFlinger, a private service behind private APIs

Fast screen capture

- Our solution: minicap, a small C++ NDK binary
  - Uses private APIs to create a virtual display on Android 4.3 and later
  - Uses the ScreenshotClient private API on older Android versions
  - Converts framebuffers to JPG via turbojpeg, which is surprisingly fast. Modern devices reach 40+ or higher FPS easily.

Benefits of minicap

- Near real-time
- Works on any Android version including 2.3
- Can pretend to be a “secure” screen, meaning it can capture views with FLAG_SECURE
- Relatively small JPG file size doesn’t overwhelm the USB bus
- Simple to use

Disadvantages of minicap

- Uses private APIs
  - Has to be recompiled for each new Android version
  - Developer previews in particular are annoying, because Google doesn’t release the real source code they’re using
  - A small number of devices don’t work, but we’ve fixed plenty
- JPG is pretty low-tech in 2018

Tooling integration

Tooling integration

- Most tooling integrates with **adb**
  - As long as the device is on ADB, it'll usually work
- OpenSTF provides "fake" devices that you can **`adb connect`** to
  - Identity is confirmed via adb keys
  - Requests are proxied to the real device
  - Implemented in adbkit, one of our projects

Tooling integration

- Create an API token
- Register your adb key
- Retrieve “adb connect” URL via API
- Run “adb connect”
- The device is now connected to your local adb and is usable in e.g. Android Studio and Appium.

Lessons learned

Lessons learned

- Overall, many of the components have held up pretty well and are still useful today.
- But, if OpenSTF was started today, what would I do differently?

Things I’d do differently

- No private APIs. It was cool to make it work, but it’s been a pain to maintain over the years
- Rethink the architecture a bit to make it easier to deploy the project
  - We get questions all the time
- Don’t cheap out on hardware, it’ll pay for itself over time

Things I’d do differently

- We expected more contributions due to using a relatively easy language (Node.js/JavaScript)
  - In reality, while there have been some, there have definitely been fewer contributions than we thought
  - With that in mind, I’d probably go with Rust now
- AngularJS was a decent choice at the time, but now I’d definitely go with React

Things I'd do differently

- OK to abandon full backwards compatibility if it makes sense, or new development easier
- Community management and responding to issues has been difficult at times
  - Welcome early contributors and make them part of the team if possible
- Lack of analytics has been an issue

Things I’d do differently

- Rather than a browser app, focusing on native apps first would make more sense now
  - Native development is easier with advanced features such as video streaming, and a better user experience overall

Insights

Things we want to do

- Android P support (soon)
- Get rid of JPEG rendering, use h264 over WebRTC instead (eventually)
- Ability to properly reset devices (eventually)
- iOS support (eventually)

That’s all!

To stay up to date

- Check github.com/openstf/stf every now and then for future updates