Sane-airscan: the future of Linux driverless scanning
Few words about myself

Name: Alexander Pevzner

E-mail: pzz@apevzner.com

Known in Open Source world as author of sane-airscan and ipp-usb

Linux user since 1.2.13 kernel times.

30+ years in software industry, focusing mostly in systems software, network protocols, software, that works with hardware.
What is driverless scanning

- Driver still needed
- But it speaks vendor-neutral protocol
- Can work with compatible devices from many vendors
- Can be compared with USB flash, SATA Hard Drive, IPP printer
- User plugs the device and it just works, regardless of brand, manufactured and model
Current state of the art

- In the ideal world, there should be only one common protocol for any particular class of hardware.

- In reality, two protocols already exist and widely used:
  - ESCL (Apple AirPrint scanning) from HP and Apple
  - WSD (Web Services for Devices) from Microsoft and W3C

- Two more are coming:
  - IPP-scan from PWG (Printer Working Group)
  - TWAIN Direct from TWAIN Group
The important value of having these standards: full codification of scanner device as a programmable object

Although standards a different, the software model of scanner is very similar

Before that, there were no clear understanding, what software interfaces are supported by scanner

In result, SANE treats options as something that can be show to user as a human-readable text. TWAIN allows driver to implement its own GUI dialog. And there is no reliable way for non-interactive program to guess, for example, does device support ADF or not

Hope, SANE standard will be eventually updated with this new understanding in mind

In the next slides I will tell a little bit about each of these protocols
- Promoted by Apple, part of Apple AirPrint standard
- Unlike printing part of AirPrint standard, specification is not published, but reverse engineered
- Relies on DNS-SD for device discovery
- Simple XML-based protocol
- Supported by every Apple device, which is good motivation for hardware vendors to support it
WSD (WS-Scan)

- Promoted by Microsoft, natively supported by Windows
- Specification is publicly available
- Also published as W3C standard, and this is slightly different dialect (uses different XML namespaces)
- Printers seems to support both dialects, but Windows driver uses Microsoft version
- XML-based
- Relies on its own discovery mechanism (WS-Discovery, based on XML multicasting over UDP)
- Very overcomplicated, specification hard to read, examples contradict with written specs
- Extensively uses XML namespaces
- In general, hardware implementations are more buggy than for eSCL
- Created by PWG (Printing Working Group)
- Fully documented
- Designed as extension of IPP printing protocol for scanning
- Relies on DNS-SD for device discovery
- Uses IPP binary representation for communications with devices
- Currently not implemented neither in hardware nor in software
- Created by TWAIN working group
- Based on Google Cloud Print (_privet._tcp), like IPP-scan based on IPP-print
- Documentation is publicly available
- Can be used for both local and remote (cloud) scanning
- Relies on DNS-SD for device discovery
- Uses JSON messages for communication with device
- Exists in a form of software simulator for Windows, freely available from TWAIN, but closed source
- May arrive on hardware within 1-2 years
On previous slides I was speaking about network protocols (all HTTP-based)

But what about USB-only devices?

They also are not lost

There is IPP-over-USB protocol, which is essentially HTTP over USB

IPP printing, eSCL scanning and even device web console works well over USB

WSD doesn’t work
sane-airscan: the implementation

- Implements the standard SANE backend, works with any SANE frontend (simple-scan, xsane, etc)
- Supports eSCL and WSD
- If device supports both protocols, chooses protocol automatically
- Relies on Avahi for DNS-SD and implements own search engine for WS-Discovery (WSD)
- Extensible. New HTTP-based protocols can be easily added
sane-airscan: few words about architecture

- Consist of the following major components:
  - Discovery engine
  - Protocol-independent state machine
  - Protocol handlers for each supported protocol
  - Logger, which is very important for troubleshooting without physical access to device
  - Supporting infrastructure (HTTP client, memory management etc)
Proper implementation of discovery is not so trivial

One physical device may be found multiple times (IP4/IP6, WiFi/Ethernet, eSCL/WSD)
  • Merges found device instances by UUID

DNS-SD network names are user-friendly, WSD are not (WSD uses UUID as device name)
  • Name taken from DNS-SD world, if possible

DNS-SD is fast (reads from Avahi cache), WSD is not (with multicast discovery there is a trade-off between speed and reliability)
  • Uses hints from DNS-SD world: if device announces IPP printer with scanner support (_ipp._tcp with scan=t in the TXT record) or _scanner._tcp service, it makes sense to continue discovery until compatible scanner is found
Fortunately, all supported protocols have quite similar workflow:

- Obtain device capabilities (paper size, resolutions, color modes etc)
- Send scan request
- Download image
- Decode image (or multiple images, if scanning from ADF)
- Tell device to cleanup after scan
- Error reporting and recovery

It allows to make a clear split between protocol-independent state machine and protocol handlers, that perform actual work. Protocol handlers are simple (~1000 C lines each), all complexity is in the common layer.
Protocol handlers

- Responsible for creation and decoding HTTP requests

- Decide next step to perform, allowing some flexibility in workflow (say, eSCL requires in a case of error to explicitly request an error reason, while WSD brings it instantly)
Good logger is “must have” for this project, because in most cases I have to debug by e-mail.

Standard SANE approach (logging to console) implies certain limitation on a level of detailing.

Logging to console still used, but not very detailed.

If enabled, very detailed log (protocol trace) created in the separate log file.

Binary data (downloaded images) goes into separate .tar file (.tar writer is very easy to implement).

In most cases, it’s enough to ask user to enable protocol trace and to send resulting files.
Supporting infrastructure

- HTTP client, image decoders, convenience wrappers for memory allocation – all goes here
- Overall driver size is relatively large, more that 20K lines
In the following few slides I will tell about our future plans
The present state. Please note, everything runs within a single process:

- Client App (frontend)
- libsane-dll.so (a simple dispatcher)
- libsane-XXX.so (device driver)
- libsane-YYY.so
The Future

- Where we go. Please note, a daemon process added to the picture:
What it gives us

- Faster client startup: no need to wait for drivers initialization
- No more “access denied” problems for USB devices, as server has enough privileges
- Ability to change SANE API without need to break hundreds of existing drivers
  
  • sane-airscan will perform a required translation

- Hardware driver together with IPP-scan server can be packed as a snap. The similar approach will be used in the printing infrastructure
Why IPP-scan

- Unlike eSCL, fully documented
- Unlike WSD, documentation is clear
- Backed by respectable organization, PWG
- Based on established standard, IPP
- Hope, eventually will be supported by hardware, like IPP-print
GitHub: https://github.com/alexpevzner/sane-airscan

Email: pzz@apevzner.com