UBICAST

Case study

Using Gstreamer for building Automated Webcasting Systems

26.10.10 - Gstreamer Conference

Florent Thiery - Ubicast

Agenda

- About Ubicast
- Easycast
  - Goals & Constraints
  - Software architecture
- Gstreamer
  - As webcasting framework
  - As automation framework
- Python & gstreamer
- Main challenges

About UbiCast

- 3 years old french company ~10 people (6 devs, 3 gst)
- Produces automated webcasting systems
  - Turnkey, end-to-end solution
  - Designed for mass video production
  - Easy to use
  - Automated capture features
  - Automated publishing workflow
- Applications
  - Education
  - Corporate training
  - Conference webcasting
- Products
  - EasyCast capture station
  - WebTV

Solution overview

- Presentation capture
- Transparent
- End-to-end
- Rich Media

What we sell

- Touchscreen appliances with accessories
- Services (training, suport, WebTV + third party hosting services, custom dev)

How it looks

Easycast

- Touchscreen GUI
- Robotic network camera support
- Tracking features
- Simultaneous XGA & A/V capture
- One-push publishing

WebTV

- Live & VOD
- Remote control
- Metadata editing
- Stats ...

Goals

- Ease of use by non-specialists
  - Touchscreen
  - Autodetection
  - Production & post-production automation
- Turnkey solution
  - Appliance
  - Integrated encoding, streaming, processing
  - Hardware integration (station, accessories)
  - Web/SaaS integration
- Technology agnostic
  - Video formats
  - Third party providers
  - Unobtrusive (hardware capture, « passive tracking »)
  - Open standards (RTP/http/ftp/...)

Project constraints

- "Small startup friendly"
  - OSS software based
  - Run on "commodity hardware"
  - Scripting language
- Parallel, heavy tasks
  - Heavily multi threaded
  - Fully asynchronous (GUIs hate http)
  - Low-level language core

Easycast Software Stack

- Appliance -> Linux (Ubuntu-based)
- Web integration -> twisted
- Touchscreen / rich multimedia interface -> clutter
- Multimedia
  - Decoding, encoding, streaming ... -> gstreamer
  - Image analysis -> OpenCV
  - Audio analysis -> gstreamer plugins
- Gobject MainLoop
- DBus (NetworkManager, HAL, utility daemons)
- Gnome technologies: gconf, gnomevfs, ...
- python : bindings for everything

Gstreamer as Webcasting framework

- Encoding, Transcoding & Streaming : many implemented protocols, codecs & muxers
  - « Classic » pipeline (1x video, 1x audio, local encoder, rtp/h264 encoder)
- Hardware support
  - capture cards
    - audio: overall good support for single channel devices
    - video: good V4L/1394 support
  - network devices friendly: good results with most network devices (http-mjpeg-multipart/rtsp-h264); work done for elphel open hardware cameras (http://code.google.com/p/gst-plugins-elphel/)
- Image compositing using gst-plugins-gl

Gstreamer as automation framework

- http multipart metadata parsing (SONY movement metadata extraction)
- OpenCV
  - Largest open source image processing library
  - Limitations : mostly scientific, input/output layers are large patched blobs, packaging/modularity issues, hard to share resources with other apps
- OpenCV & gstreamer
  - gst-opencv http://github.com/Elleo/gst-opencv
  - Keeps the core of opencv in a compact package
  - Shares resources
  - gst events: great api for forwarding results upper layers
  - Great plugin api
- Audio filtering / analysis

Miscellaneous uses

- Asynchronous / automagically threaded
  - image conversion/resizing
  - signal probing
  - large file copy with pauseability and progress reporting (which AFAIK gnomevfs does not provide) – gnomevfssink too simple for ftp
- Port scanner

Python & Gstreamer

- The tremendous power of gst.parse_launch
  - Prototype on the command line
  - Quickly port and interact
  - Result: gstreamer python programming is 80% string manipulation (concatenating pipelines portions) ; elements naming is crucial
- gstmanager (http://code.google.com/p/gstmanager/)
  - Simple api wrapper
  - gst.event forwarding (broadcasted)
  - Debug helper (print gst-launch-compatible reconstructed pipeline description)
  - Overlay plugin system, but hard to get it right
- Python bindings are very good but some low levels feature make it crash (ex: notify on queue filling states), sometimes simpler is better (e.g. property polling)

Main challenges

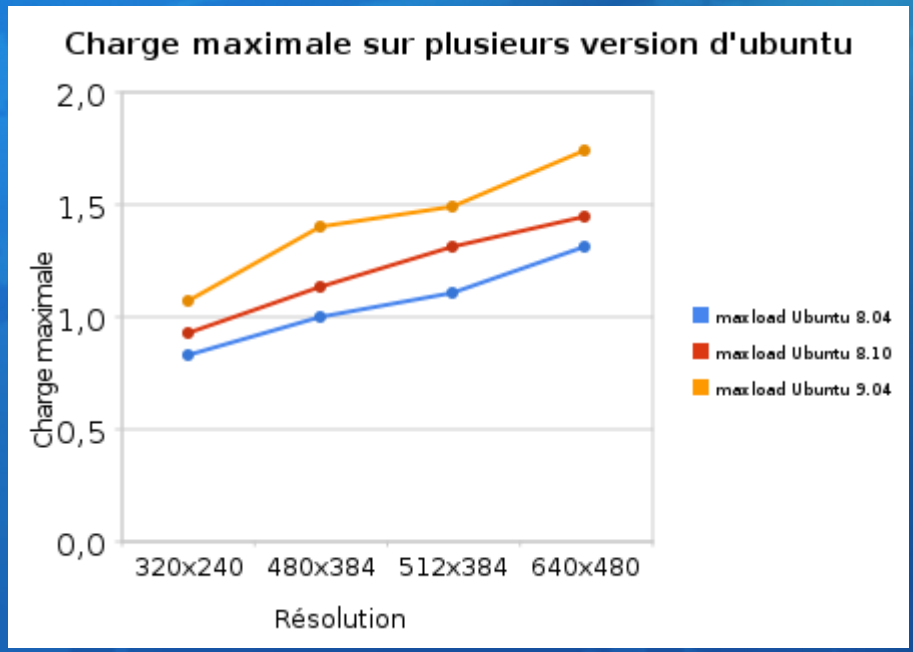
- Learning curve
  - It's a long road just covering the basics (tools, doc, debugging, ...)
  - Writing small apps helps discovering. Tool: http://code.google.com/p/gst-gengui/
- As a company, gst skills are hard to find
- A/V desync is live pipeline's worst nightmare
  - Developped "clap" software for long run tests
  - Failed detecting drifts automatically
- MT safetyness – gobject.idle_add is your friend, especially with twisted / clutter mix
- Debugging blockings on very large pipelines is hard to figure out (queue uses). Tool: http://code.google.com/p/gst-viewperf/
- For consistent behaviours
  - Better to stick with one single native recording format
  - Find lowest common denominator for caps
- Non linear editing (gnonlin) is hard ; we ended up used third party utilities (oggtools)
- Many small hacks for safety (e.g. check target file size is really growing, ...)

Main challenges: hardware support

- Ok, not directly gstreamer related but it's a pain to find professional devices supporting Linux. Testing/torture is mandatory
- Most professional A/V manufacturers don't know/don't care about gstreamer (not the same in embedded world !)
- Some of them have V4L apis (but no HAL/udev rules, limited V4L compliance, kernel hacks...)
- The others have proprietary APIs (-> MediaMagic – space for ecosystem)
- Most of them didn't offer Linux support at all 3 years back, but this is changing !
- Sometimes unreliable behaviour but most of the time lower level problems than gstreamer (kernel)
- Hardware often causes system freezes
- Hardware-specific additional latency → delayer « hackish » element

Main challenges : The version choice

- Performance and behaviour will vary among releases
- For an appliance, validating/developing against a single distribution is easier (e.g. Ubuntu 8.10 – assuming tests done by vendor)
- Many tests required to stabilize a version
- Having performance-oriented benchmarking routines would help choosing versions
- How to apply small patches without compromising distro stability/integrity ? Features/fixes propagation delay → often easier to use hacks in production



Main challenges : dynamic pipelines

- Dynamic pipelines ~= adding/removing branches
  - Why ? Because you can't (easily) share hardware ressources between pipelines
  - adding is quite straightforward
  - removing without noticeable hiccups is harder
  - pad blocking / unlink / unlock ... not easy with a/v pipelines !
- The recording case: the muxing issue
  - muxers can't reset timestamps dynamically (bug https://bugzilla.gnome.org/show_bug.cgi?id=561224)
  - Restarting a modified pipeline worked very well for us (KISS), but care for hardware liberation delays (e.g. usb audio) !
- Found it easier to run parallel pipelines (ex : xga processing)

To sum up

- Gstreamer is a wonderful framework, incredible potential
- Gstreamer + Python is a powerful combination
- Stable VS Latest problematic/frustrating in production context
- KISS works
- We underestimated the testing effort
- We underestimated what users can do → « safety cream »
- Not yet easy to use dynamically

UBICAST

Thank you. Any questions ?

Please come and check it out !