# Synchronised multi-room media playback and distributed live media processing and mixing

GStreamer Conference 2015, Dublin

9 October 2015

Sebastian Dröge <sebastian@centricular.com>

# Introduction

# Who?

- Long-term GStreamer core developer and maintainer since 2006
- Did the last few GStreamer releases and probably touched every piece of code by now

- One of the founders of Centricular Ltd
  - Consultancy offering services around GStreamer, graphics and multimedia related software

# What?

- We're going to talk about how to synchronize media between multiple devices over the network, in specific
  - Where is this useful? Scenarios
  - Clocks, why and which?
  - How to transport the media?
  - How to synchronise the media on different devices?
  - Set up of the relevant GStreamer elements
  - Interoperability with other solutions out there

# Scenarios

# Synchronised multi-room playback

# Live mixing and recording

CC-BY-2.0, Nayu Kim

# Clocks

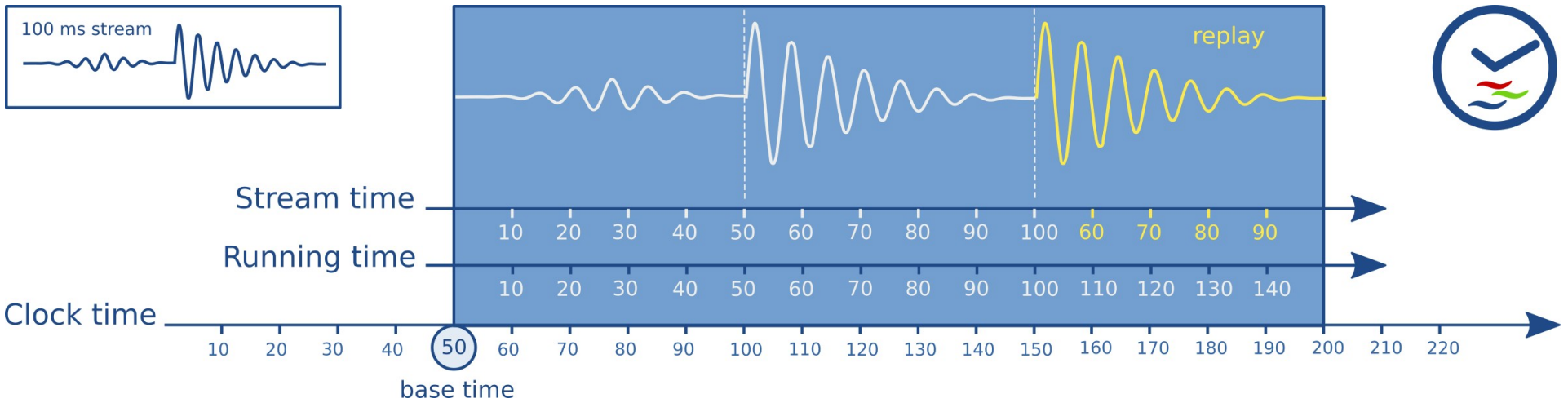
# Clocks

- For synchronising anything we need clocks
- No two clocks in the universe run at the same rate or show the same time
- We need a way to approximate the same clock on multiple devices and teach GStreamer how to use such a clock to synchronise media

# GStreamer Clocks

- Every pipeline has a single (master) clock
  - Used for synchronising all media in that pipeline
  - Usually used in sinks and live sources
- Automatically selected for you by default
  - But can be manually set too
- Used to generate the “running time” of the pipeline

# GStreamer Clocks



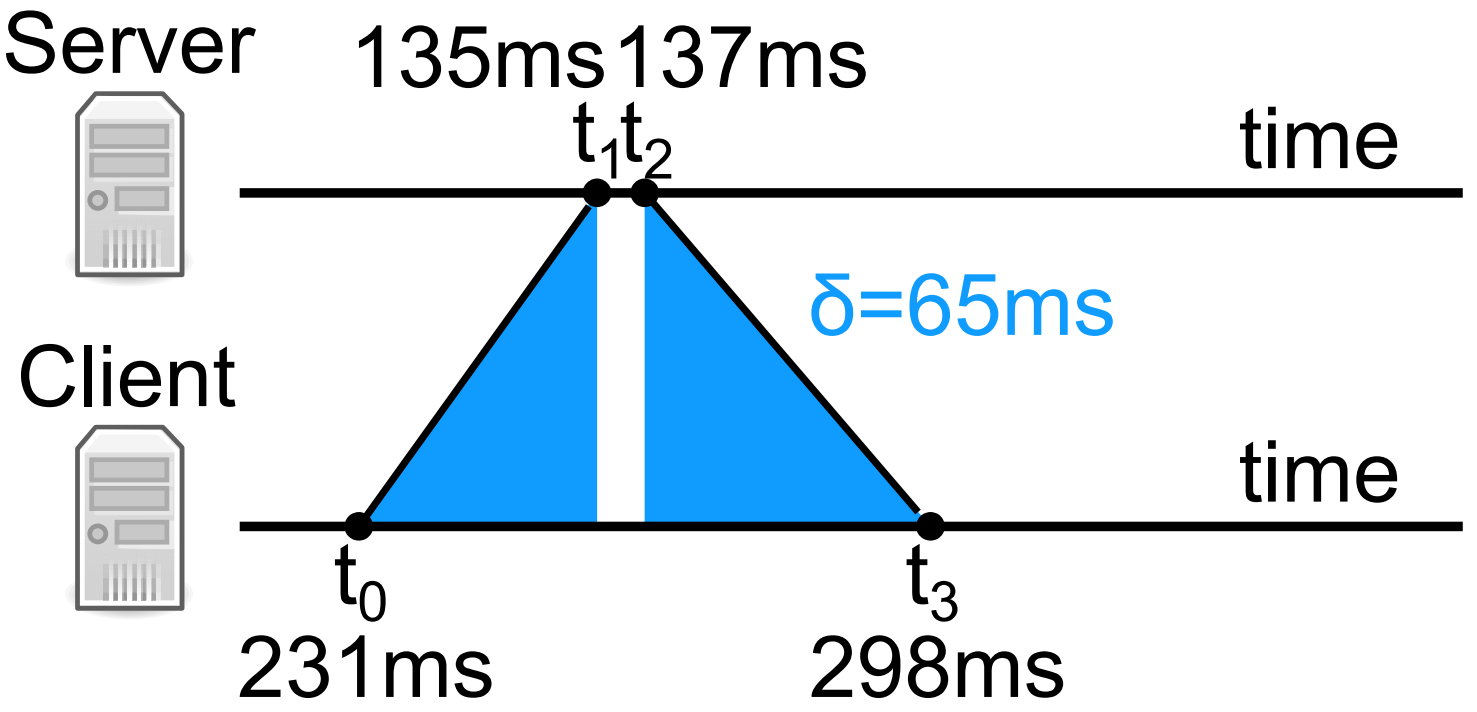
# GStreamer Clocks

- Each clock is a GstClock subclass
  - Needed: get_internal_time() virtual method that returns the current internal time of the clock in nanoseconds
  - Requirement: always running forwards
- Infrastructure for slaving one clock to another
  - Estimating relative clock rates and offsets between the two
  - Allows translating times between both clocks
- Idea: Create a clock that bases its internal time on observations from time on another machine on the network, slave it to the local system clock
  - Might've used that in the past already: this is how NTP is used by most people

# GStreamer Net & NTPv4 Clock

- NetClock existing since around 2005
- Custom protocol similar to NTPv4
  - UDP messages sent between client and server
  - Client “asks” the server for the current time & estimates the round-trip-time
  - Uses both the estimate the current time at packet receival
- NTPv4 (RFC4330) client clock since 1.6
  - Shares most of the code with the NetClock
- Tricky part: Filtering to handle networks with fluctuating RTTs

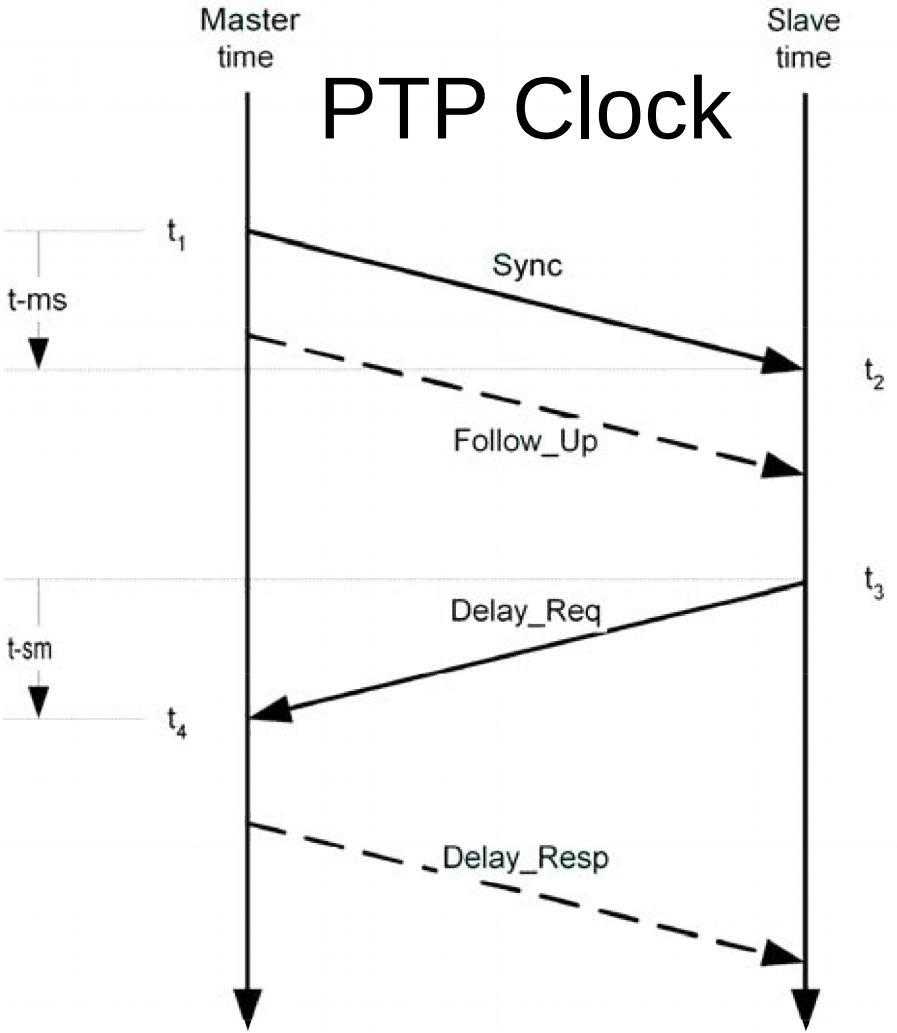
# NTPv4 Clock


Server
135ms
137ms
$t_1$ $t_2$
time
δ=65ms
Client
time
$t_0$
231ms
$t_3$
298ms

# PTP Clock

- PTP (IEEE1588-2008) client clock since 1.6
- For **local** systems that require higher accuracy than NTP
  - µs to ns range, NTP usually in the ms range
  - Possibility of driver and network hardware support to increase accuracy
- For distributing time to many devices in a local network without much overhead
  - Using multicast and only doing RTT measurements sometimes
  - Less robust for networks with fluctating RTTs (WiFi!)
- Distributed algorithm to automatically select the “best” clock of all available in the network

# PTP Clock




Timestamps known by slave

$t_2$

$t_1, t_2$

$t_1, t_2, t_3$

$t_1, t_2, t_3, t_4$

# GPS Clock

- GPS also needs all devices synchronised
- Could implement a GstClock around a GPS device

- Does not exist yet! Any volunteers?

# Media Transport

# Possible Choices

- HTTP
  - Easier to do buffering, TCP retransmissions
  - Not a problem for firewalls usually
  - Not ideal for low-latency live streams
- DASH/HLS
  - Mostly the same as HTTP
  - Easy CDN usage, multiple bitrates/resolutions/etc
- RTP/RTSP
  - Great for low-latency streaming
  - **We're going to focus on this for now**
    - others work less “automatic”. Ask me later!

# RTP/RTSP

- GStreamer RTSP server and source very easy to set up
  - “Write a server in 6 SLOC”
  - Easy to customize for everything we need here
- Stream configuration exchange via SDP and a HTTP-style request/reply protocol
- Media streaming via RTP
  - Custom RTP setup possible with GStreamer if needed, flexible enough for everything including WebRTC
- RTP has clearly defined timing semantics

# Synchronisation

# RTP

- Each packet has a timestamp
  - Random offset for each stream, based on the sender clock
- Each packet has a sequence number
  - Monotonously increasing with random starting point
- Usually transmitted in real-time
- Allows to synchronise every stream, detect packet loss and handle packet reordering
- Mappings for every major codec and metadata schemes

# RTP

- GStreamer default RTP time handling in rtpjitterbuffer
  - Take first packets timestamp and arrival time as “base”
  - Estimate sender clock based on network jitter, packet timestamps, packet arrival times
  - Slave estimated sender clock to the pipeline clock
- Not useful by itself for inter-device or even inter-stream synchronisation

# RTCP

- Bidirectional communication between sender and receiver
- Basic case
  - Sender report to the receivers, receiver reports to the sender
  - Sent/received packets, packet loss, RTTs
- Can be extended by other feedback / control (e.g. RFC4585)
  - Retransmissions, keyframe requests, …
  - Application specific extensions

# RTCP

- Interesting part for synchronisation
  - NTP timestamp and corresponding RTP timestamp in SR
  - Not necessarily based on an actual NTP clock
  - RTP timestamp interpolated for the NTP time when RTCP packet is sent
- Allows inter-stream synchronisation
  - Default rtpbin behaviour
- Allows inter-device synchronisation
  - All devices need to agree on the same “NTP” clock by some other means
  - Not the default mode! Details later

# RTP Header Extension

- RFC6051 defines an RTP header extension for carrying NTP timestamp
  - Allows to create a direct NTP ↔ RTP timestamp mapping
  - Allows immediate synchronisation before RTCP
  - Allows mapping for each RTP timestamp instead of just every now and then via RTCP
- Could create the same for PTP
- Main problem is still
  - Every device needs to agree on the same clock by some other means

# RFC7273

- Finally solves the remaining problem
  - But is not implemented in GStreamer… yet!
- Defines SDP fields for
  - Media clock that is used for timestamping packets
  - Offset between clock epoch and RTP timestamp
  - Supports NTP, PTP and others
- Plan is to implement it some time later this year

# Setup of GStreamer elements for RTCP-based synchronisation

# rtpbin

- “ntp-sync”: gboolean
  - Overrides rtpjitterbuffer behaviour to produce output timestamps based on the NTP clock times instead of packet arrival times
  - Only useful on receivers

# rtpbin

- “ntp-time-source”: enum
  - Useful for sender and receiver
  - **NTP**: Real NTP time based on real-time clock (NTP clock epoch)
  - UNIX: Same as NTP but using the UNIX epoch
  - Running time: Pipeline's running time
  - **Clock time**: Pipeline's clock time (running time + base time)

# rtpbin

- “buffer-mode”: enum
  - Only useful on receivers
  - None: Uses RTP timestamps, starting from 0
  - **Slave**: Sender clock estimation
  - **Synced**: Uses RTP timestamps, starts at arrival time of first packet

# rtpbin

- “rtcp-sync-send-time”: gboolean
  - Only useful on senders
  - Use capture or send time for the NTP ↔ RTP time mapping in RTCP
  - No difference for non-live pipelines
  - Latency is the difference for live pipelines
  - Set to FALSE if receiver **and** sender pipelines should be synchronised

# pipeline

- `gst_pipeline_use_clock()`
  - Force usage of a specific clock
  - Use same network clock on receivers and senders if not using RFC7273 integration
- `gst_pipeline_set_latency()`
  - Overrides default pipeline latency handling to use a static latency
  - Should be at least the maximum receiver latency
    - Network plus decoder plus sink latency!

# Setup Summary

- `test-netclock*.c` from gst-rtsp-server/examples
- Sender
  - Setup netclock provider (server)
  - Use system clock for that and the pipeline
  - “ntp-time-source”: clock-time
- Receiver
  - Setup netclient clock with sender's server
  - Use that for the pipeline and set 500ms fixed latency
  - “ntp-time-source”: clock-time, “ntp-sync”: TRUE, “buffer-mode”: synced
- Easily get < 1 frame synchronisation between receivers

# Interoperability

# RTCP

- NTP based synchronisation will work if
  - Sender and receiver agree on the same clock by some other means
- Works with many 3rd party solutions
  - Sometimes possible to configure an NTP server to use in senders
  - Completely based on RTP RFC, nothing else
- Synchronisation based on send or capture time usually undefined in other solutions

# Ravenna / AES-67

- IP based audio broadcasting standard
- Raw audio via RTP
- PTP clock
- RFC7273 for media clock distribution

- Supported by many professional broadcasting applications

# BBC IPStudio

- IP based live studio, broadcasting R&D project
- Video and audio via RTP
- RFC7273 and PTP clock
- Extensions for additional metadata
- Compatible with AES-67

# SMPTE 2022 / 2059

- IP based video transport standard
  - Remote, real-time video production
- RTP for audio / video
  - “SDI over IP”
- PTP clock
- FEC

# Thanks!

# Any questions?