Project acronym: LADIO Project number: 731970 Work package: Deliverable number and name: D1.2: Distributed data management	Title: Real-time device data and metadata acquisition Work Package: WP1 Version: 1 Date: May 31, 2017
	Authors: Kristan Skarseth
Type: [ ] Report [ ] Demonstrator, pilot, prototype [ ] Website, patent filings, videos, etc. [X] Other	Co-Author(s): Stian Z. Vrba, Carsten Griwodz, Jonas Markussen, Benoit Maujean To: Albert Gauthier, Project Officer
Status: [ ] Draft [ ] To be reviewed [ ] Proposal [X] Final / Released to EC	Confidentiality: [ ] PU – Public [X] CO – Confidential [ ] CL - Classified
Revision: Draft	
Contents: Deliverable 1.2.	

Table of Contents

Introduction

At a very high level, data acquisition proceeds as follows (see Figure 1): Each shooting location is equipped with one SetBox which provides WiFi connectivity. The SetBox may also provide a link towards storage servers at the post-production house so that data is uploaded as soon as it is ready. CamBoxes and MiniBoxes (in the following collectively called Boxes) join the WIFI network and immediately go about to announce their presence, identity and capabilities. The SetBox registers them as Boxes and initiates a default data transfer mode. Whether the Boxes are notified by a SetBox or not, they are immediately acquiring raw data as well as time-varying metadata in real-time from miscellaneous recording devices connected to them. Such devices present on a movie set include cameras, audio recorders, LIDARs and 360 cameras. The Boxes store the acquired data locally in a format optimized for efficiency. Depending on the transfer mode that the SetBox prefers, they deliver raw data to the SetBox in push or pull mode. The SetBox converts it to the format(s) used in the production, and stores the converted data locally on its disk. The SetBox copies the converted data to back-end storage servers as soon as the link becomes available.

A shooting day can generate huge amounts of data, but Boxes have limited storage space, so data must be deleted from them periodically. In the first iteration, we do not attempt to automate this process: The SetBox will provide a management UI which will allow the operator to see the status of data files on each Box (copied/not copied to its “upstream”) and to delete the files.

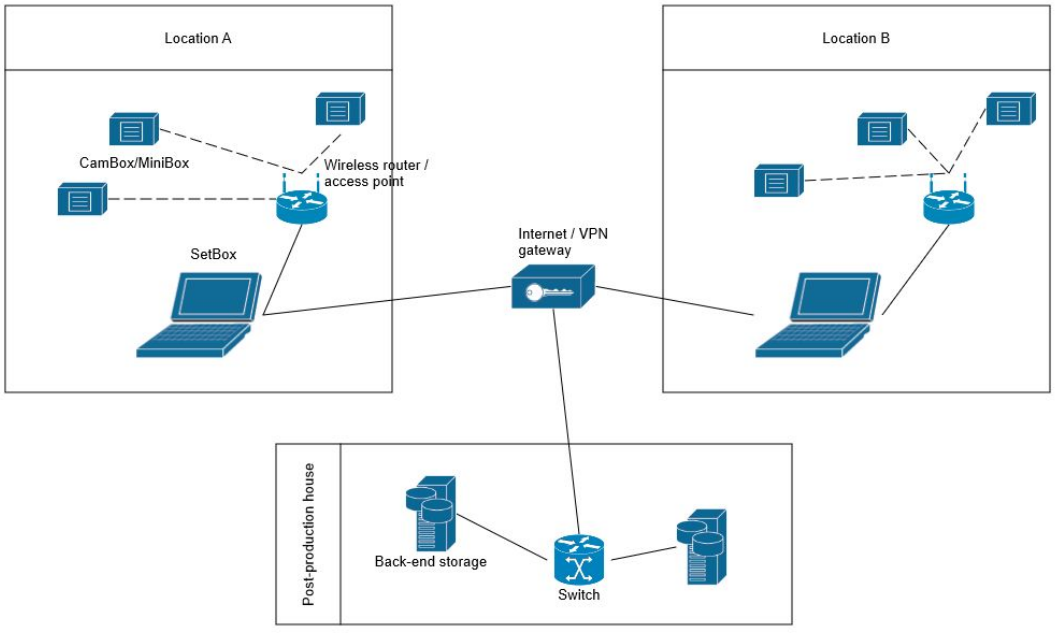

FIGURE 1: DATA ACQUISITION SCENARIO

In the rest of document, we describe the design of a distributed data management (DDM) system that allows for a simple and safe exchange of *objects* between nodes. Before presenting the design and implementation, we will introduce some definitions and describe assumptions behind the design.

The following elements are currently in a working state and available for LADIO partners:

- announcement and registration of Boxes

- capturing of one or more data feeds from cameras connected to a CamBox (main cameras and secondary cameras)
- local storage and streaming of data

These are implemented in the following private repositories:

- Main CamBox / MiniBox software
  https://github.com/alicevision/hal
  It stores data on SD cards or USB disk, streams over wireless as well as wired network.
- Support library for efficient data movement
  https://github.com/alicevision/libmarvin
  It implements both shared memory as well as protocol buffer communication.
- A SetBox-side implementation for controlling and receiving streams
  https://github.com/alicevision/halController

Status

The progress in work packages 1 and 2 suffers from the loss of application partner LABO, who was developing a solution named Gamp that was already used for data collection on some film sets, in particular also on sets of Norwegian public broadcaster NRK, who is a major contributor to the EBUCore development. We are working intensely with a new startup company to replace them, but in terms of CamBox/MiniBox hardware platform we are facing their completely new design and implementation, and in spite of our constructive and productive cooperation with them, progress is not ideal.

At this point, the code for the distributed data management supports CamBox/MiniBox registration as well as stream delivery are in use, but the transition from Google Protocol Buffers to a REST API that complies with the LADIO data model has not happened yet.

Definitions

Device	Any on-set device from which data acquisition is desired. Examples: studio camera, audio recorder, witness camera, inertial unit, lidar.
Data	Files obtained by acquisition from some device or supplied by the user.
Derived data	Files which are the result of applying computational steps on data or other derived data files. The inputs need not come from the same object. The computational pipeline must be described in metadata files.[1]
Metadata	Information generated by the DDM system or supplied by the user. Metadata files are separate from data files (as opposed to being embedded in data files).
Object	A collection of data and metadata files (i.e., a directory).
Object identifier	A piece of data (e.g., a string) that uniquely identifies an object.
Location	Single SetBox with established WiFi network and multiple Boxes acquiring data.
Back-end	Facilities at a post-production house (e.g., storage servers).

[1] Metadata must describe the exact software version and platform used in each step of the pipeline.

Assumptions and requirements

Back-end is secure.	When objects are delivered to the back-end, they are assumed to be secured and that they can be deleted from “Boxes”. Backup and similar routines on the back-end are responsibility of the user.
Locations are autonomous.	SetBox provides all DDM services on a location, *without assuming network connectivity to the back-end.* Some use-cases need certain objects to be available on a location (e.g., 3D assets for previz); in such cases, we require that users copy the needed objects to SetBox. For very small productions and testing, it should be possible to use a stand-alone SetBox without any back-end.
(Derived) data files are immutable.	In-place modifications of data and derived data files are forbidden. This aligns well with the industry workflows, and vastly simplifies DDM implementation because no conflicting data changes can occur between nodes.
Versioning of metadata.	Data files in LADIO cannot be edited and committed to the LADIO system without creating a new ProductionJob metadata entry and the creation of a new MediaSource metadata entry for the new content. When metadata is newly generated from data, this requires also a new ProductionJob and MediaSource. Lookup operations should therefore by default request the latest instance of identically named MediaSources.
Cheap updates.	It must be cheap to create new object versions.

Object identity and identifiers

On Boxes, we expect that files are organized based on camera clip (Roll, scene, shot/slate, take) and source type. The concrete filename scheme is not fixed but determined by a user-defined template. To safely distinguish files generated by devices connected to different Boxes, templated names must always include the GUID of the Box.

Some Boxes are expected to record the set continuously independent from the actual shots and takes from the point of view of the team on set. They may record between shots or run the whole day. These are distinguished by their separate NarrativeScene, which is a top-level identifier in the default name template and have only a constant (dummy) take and shot identifier.

Each Box can be connected to multiple devices:

- main camera
- several witness cameras
- FIZ unit (lens configuration with timecode)

Each of these source types can generate a set of metadata files for a clip.

Logical names for data objects, which comprise both data and metadata files and file segments, will finally fit into the hierarchical namespace that is spanned by the hierarchical information of scene (NarrativeScene), shot (Shot) and take (Take) according to the LADIO data model. However, it is not always possible or desirable to configure a Box to use this full name.

If the SetBox does not create a specific configuration for a Box, objects are named with the Box’s

GUID, dummy information for scene, shot and take, the UUID of the actual recording device connected to the Box, initial creation timestamp according to the Box's clock, the datatype of the file's content, and a sequence number for consecutive samples. Not all of this is encoded in the file name, but in the full path of the file.

Synchronization between Boxes and SetBox

The data transfer from Boxes to a SetBox can be handled in three different ways. Data can be sent pro-actively from Box to SetBox while it is recording in a streaming fashion, it can be downloaded on demand by initiative from the SetBox, or the Box data can be transferred by physically moving an SD card. These 3 choices are not mutually exclusive, and data that is collected by a Box should always be stored on it until it is explicitly deleted. The latter data transfer options can therefore also always be considered fallback options for the former options (ie. pull for push, and physical movement for pull).

In both online cases, it is required that the SetBox becomes aware of the Boxes, and the decision about the chosen data transfer mode is made through an application running on the SetBox. The discovery mechanism is described below in Discovery Mechanism for MiniBoxes and CamBoxes.

A pure pull implementation avoids problems of contention in the WiFi network on set, but it is not always feasible, for example because this would lead to a loss of live preview functionality as well as on-set previz. Pure push therefore requires a pull implementation as a fallback option. Also, even with small amounts of content, some push data will be lost or corrupted. An implementations of push-like streaming services inspired by MPEG DASH might be feasible but is known to have an undesirable streaming pattern that leads to transient but cyclic network problems in WIFI scenarios.

Mini-boxes will not necessarily always be available, thus data must be retrieved from them frequently or even continuously to make sure it is always backed up.

It should be noted that the combination of push and pull described in this document is suitable for RT use.

Metadata is recorded by a Box and stored to files. A REST API allows the SetBox to query the Boxes for new metadata files, and download it using a HTTP server. The SetBox will periodically poll the Boxes for new Metadata, and download when needed.

For on-set usage, the SetBox requires the metadata as soon as it is available. Boxes will need to push new metadata as soon as it is received from the recording device. This can include both per-frame metadata and sub-frame metadata such as zoom, focus and per audio sample metadata.

Due to potential WIFI issues we cannot assume that the SetBox will always receive all metadata when it is pushed from a Box. We need to be able to identify missing metadata, and push or pull it at some later point. This can be done either in parallel with receiving new push-data when the connection is stable again, or after recording is finished.

The Boxes on the main cameras will generally always be available throughout the day on a set, and we can rely on pulling missing metadata after a shot has been finished. Other recording devices may however be turned off between shots, or they may just be used for a couple of shots before being turned off for the rest of the day. In these cases we need to push the data from the Boxes to the SetBox as soon as possible, not at some random point after the shot is finished.

Implementation

We will start by implementing a pure pull solution where the SetBox periodically checks for new data on Boxes and downloads it using HTTP. This will cover most non-RT use cases, with the exception of when some Boxes are turned off between shots.

When this is working we may extend with push functionality where the Boxes can push notifications of new available data to the SetBox so periodic polling intervals can be increased.

Finally we will add functionality to push the metadata itself to the SetBox to allow for RT collection of metadata on the SetBox. Incomplete data due to WIFI issues will need to be handled with a combination of pulling from SetBox and additional pushes from the Boxes.

Sections below describe the requisite software that will be running on the SetBox and Boxes.

SetBox

SetBox runs several processes:

1. An NTP server as a global time source for the film location.
2. The "Registration" process continuously receives DNS-SD notifications from available devices. This is will be solved by the server implementation of the open source AllJoyn project.
3. The "Receiver" receives push notifications (including continuous data streams) and pulls new data from the Boxes. Boxes deliver data in a proprietary format (described below) and this data is stored to a temporary folder ("queue") on the SetBox.
4. The "Converter" is a batch process which examines the "queue" for new completed files, that is, a file fully transferred from a Box to the SetBox, and converts them to EBU-Core XML.
5. Thread to periodically (e.g., each minute) record local time to a file (for drift computation later)

These processes may either be a standalone daemon or integrated with the LADIO app.

MiniBox / CamBox

It also runs several processes:

1. NTP client. The Box runs ntpdate during boot and *before* starting the ntp client daemon for one-time synchronization.
2. Register itself in DNS-SD and announce services. This is solved by using the Tiny client implementation of the open source AllJoyn project.
3. "Collector" receives and stores metadata to a file on disk in real-time. In the later stage, it will also push data to the network.
4. HTTP server (nginx) with a two-fold role:
    a. Downloading completed files (pull).
    b. Running REST APIs (device control [where applicable], status queries, etc.)
5. Same thread as on SetBox, for clock drift estimation.

Each Box has its own unique GUID. For simplicity, the GUID is based on the primary WIFI card's MAC address. Although many cards allow a modification of this address, original address can still be assumed to be globally unique. In the unlikely case of a MAC address collision, WIFI will prevent

these devices from co-existing on a film set.

Deleting of files on Box will be done actively through SetBox gui. Can potentially be extended with automatic deleting when files are verified on SetBox/final destination(s).

Separation of metadata into files

The metadata collected on the Boxes must be stored in files. These files will be transferred to SetBox. In a traditional film production setting the files will be manually organized by file naming based around camera roll, scene, shot/slate and takes.

With all data correctly timestamped by the Boxes, which are synced with the SetBox NTP server, the files does not necessarily need to be named to facilitate this approach of organization. At the same time a substantial amount of additional data is generated on a modern film-set compared, and naming collisions can be a real issue. It is therefore proposed to use GUIDs as filenames, and let the timestamp of the file, in addition to the file contents, dictate where in the production timeline the file belongs, and indirectly which other files it conforms to.

Each SetBox has a constant GUID which uniquely identifies itself and any files which comes from it. Any files that are offloaded from a SetBox will inherently be from the same shooting location. Each Box also has its own GUID which helps distinguish the Boxes, but as one Box can collect data from multiple recording devices simultaneously, we will still need to separate files on each Box.

A recording device connected to a Box can be identified and recognized by hashing some combination of data describing the device. For most devices this will be the serial number and manufacturer name. For as long as the device is not physically altered this hash name uniquely describes the device in a satisfactory manner. A variety of physical alterations that can be made to a camera rig do not influence this uniqueness. These are instead transported in the metadata that is associated with the data that is captured by the recording device. This does include all the properties that can be described by MediaInfo such as focus position, zoom ring position and exposure settings, as well as physical position that can be recorded using LADIO extensions.

As soon as data and metadata are stored in a SAN, the SetBox GUID can potentially be removed. Although the SetBox is a device that is recognized in the preparation phase of the shoot, it does not act as a Recorder that creates content (at least the specific computer on set does not do that in its role as a SetBox), and it does therefore not appear in the metadata. All files collected by the SetBox should be linked together by other metadata. Might however be necessary to link some files more explicitly before the SetBox link is lost. Some files will span multiple film shots, and it would then be necessary to know which collection of film shots the file belongs to.

Disk file structure

We assume that Boxes can and will be moved from one recording device to another, possibly during the same shooting day. We also assume that some recordings may span several film takes, or that there may be several files for each film take on one Box.

On **a Box**, we can organize the files as follows: for each *take*, we generate a new directory named by a UUID. This UUID represents a combination of scene/shot/take/{UUID of connected device}. To account for individual habits of users (esp. DITs), who must be able to understand the file structure of disks and SD cards used for recording on Boxes if everything else fails, we support a user-defined

template for generation of the specific name. The UUID, however, must always be part the template. The REST API must be able to return the set of files newer than a datetime given as a parameter.

On the **SetBox**, we handle push data and pull data separately. The reason is that push data is probably not streamed completely because there will sometimes be network disruptions and frequently missing packets. Usually, a pull will be needed to download the complete file to the SetBox. This does not mean that the entire video file is required, but rather, an HTTP range request can be used to retrieve only the incomplete data blocks. A precondition for this is that SetBox-side naming of the pushed data is identical to naming used for storing on the Box side apart from a prefix, and name templates must uniquely identify the Box. This unique identification is guaranteed by enforcing that name templates include the unique UUID of the Box.

A DIT should also be able to manually transfer the files from an SD card (from a Box or any other device). While devices without LADIO software will require manual steps to identify the origin of the data, data from a Box must include the device identity. This implies that the each "volume" received from the same Box must contain the same Box GUID.

File formats

The original plan for LADIO was to use Google Protobuf for information transfer between Boxes and SetBox due to its platform independent and compact binary encoding (platform-independent binary encoding is also important for lossless transfer of floating-point data, since bit-identical binary floating point - text - floating point conversion may not be reliably implemented across compiler versions).

In conjunction with the development of metadata extensions to the EBUCore model, we discovered that libraries exist that encode JSON to a compact binary format, for example https://github.com/nlohmann/json (CBOR and msgpack). These are similar in platform independence and compression efficiency, but the use of JSON provides two advantages. The first advantage is that JSON is the currently the primary choice for data storage in Document Stores (such as MongoDB, Elastic, OrientDB), a storage option that we chose because it provides more efficient information access than flat files and can -in contrast to classical databases- work with partially structured data. Choosing a JSON-based transfer removes the need for transcoding. The second advantage is that JSON can be translated into XML representations, for example using the library Boost.PropertyTree (http://www.boost.org/doc/libs/1_61_0/doc/html/property_tree.html). This reduces the amount of code that must be manually written and adapted for every update of metadata specifications.

We therefore reconsidered our choice and choose binary-encoded JSON because of its simplicity and greater flexibility.

However, even binary-encoded JSON is not "compact" in that serialization must duplicate dictionary keys while serializing objects. This can be mitigated by recursively transforming objects to arrays by traversing the keys in alphabetical order. A schema giving the field names corresponding to array positions can be dumped separately and only once.

As an example, consider the following JSON object:

```
{
        "key3": 12,
        "key1": "asdf",
        "key4": [ 1,2,3 ],
        "key2":
                {
                "key6": "aa",
                "key2": 3.14
                }
        }
}
```

We first construct the “schema” object as follows, note the sorting of keys:

[“key1”, [“key2”, “key2”, “key6”], “key3”, “key4”]

Here, an array denotes a sub-object and the *first* name in the array is the name of the key in the *parent* object. Then we can send the data for each frame as a key-less array containing values and possibly nested arrays; note the correspondence of values with the keys given by the above array:

[“asdf”, [null, 3.14, “aa”], 12, [1,2,3]]

Sub-arrays having null as their first element correspond to sub-objects, while other arrays correspond to array values in the “current” object.

To begin with, we will use the ordinary (non-compacted) JSON serialization and store it in a compressed format. In this case, each compressed JSON object will be prefixed with a 16-bit length field.

API Definition

Polling for new files

SetBox polls Boxes periodically for information about potential new files.

Polling URI: /poll?newerThan=[*isoDateTime*]

Optional argument: *isoDateTime*

If *isoDateTime* is not provided, all files on the box are returned.

if *isoDateTime* is provided, all files newer than the given date are returned.

isoDateTime refers each file’s last modified time

Each file in a Box has 3 states: “writing”, “incomplete” and “complete”. “writing” state is for files currently being written to. If a “writing” file completes successfully, the state changes to “complete”.

If the box crashes, “writing”files will be changed to incomplete on restart. Filenames suffixed by the state label, like XXX.writing. The polling API call returns all incomplete and complete.

T=tmp, I=incomplete

data-file: T/I_A001_C001_XY0101.mov

hash-file: A001_C001_XY0101.mov_<hash_string_here> (file is empty, hash is stored in filename)

The following is the format of the JSON returned by Boxes to SetBox, describing files on the Box, ready for download.

```
[{
filepath,
size,
md5,
time_last_modified,
time_created
}]
```

Download File

Download URI: /download/<filepath>

argument filepath is retrieved from a json object returned by a poll-request.

Authentication

Currently there is no authentication required to communicate with the Box. As the Box contains the video proxies of the Live Action camera, it is important to ensure that anyone cannot download files or change parameters on the boxes.

Discovery Mechanism for MiniBoxes and CamBoxes

As a great number of different devices can be used on a recording set, requiring manual configuration by operators is not an approach that is feasible. For auto-discovery of Boxes, we use the open-source service discovery software named AllJoyn[2].

AllJoyn is an an open source software framework developed by the Allseen Alliance released under the Apache 2.0 license. It enables interoperability among heterogeneous “Internet of Things” (IoT) devices, by offering a consistent and universal method of bringing new devices onboard an existing network and discover available services. It also provides a common service for monitoring and managing devices, regardless of manufacturers. Being an industry-backed initiative, AllJoyn appears to be widely adopted and supported, and many previously used well-known IoT services, such as the

[2] https://allseenalliance.org

former Home Appliances & Entertainment service, have merged into AllJoyn. More recently, the Allseen Alliance has merged with the Open Connectivity Foundation as well.

AllJoyn is available for all major operating systems (Linux, macOS and Windows) and even comes in a minimalistic variant suitable for embedded devices with lack of features found in modern CPU architectures, such as multithreading. The framework offers a *zero-configuration* bootstrapping process to get access to a network shared between multiple devices, and has bindings to multiple programming languages (such as JavaScript, C++ and Java).

Access to the AllJoyn network is presented to an application running on a device as a logical bus, to which the application attaches itself. The framework uses a combination of multicast and broadcast UDP packets in order to create this bus. Once the device has attached itself to the bus, it can advertise its services using an About announcement. The announcement contains metadata in XML format about the device and supported interfaces. Other applications in the network can register About announcement listeners with specific filters. Services are registered on URI-like paths in the bus, similar to HTTP endpoints.

Specific services can be session-based or session-free. Upon discovering a service, a remote application can join a session, and the application offering the service has the option of accepting or rejecting the remote application. Sessions can be either point-to-point or multi-point. Security and authentication is handled at the application level, and the AllJoyn framework offers multiple authentication mechanisms.

An application offering a service implements a BusObject, and describes it in the About announcement metadata XML. The BusObject description contains a list of methods and their parameters and return values. After receiving an About announcement, a remote application can request a BusObject and call methods on it, similar to remote procedure calls and Java-style remote method invocation.

The advantage of AllJoyn compared to an mDNS-based discovery service written specifically for LADIO is that operating system independent and network technology independence has already been solved. Furthermore, service registration functions are included, enabling a very straightforward XML-based specification of the client-side (ie. MiniBox or CamBox) service.

The AllJoyn code that is appropriate for LADIO Boxes is the “Thin” client, which enables only the announcement of a service and can be installed on a client without adding a large number of software dependencies. The server side, which must run on the LADIO SetBox, has extensive software dependencies (libgtk2, libssl, xsltproc, libxml2, libcap, Python2.6, SCons, Uncrustify, Doxygen, Graphviz, TexLive, Gecko, Java, Junit, Google Text). While the removal of dependencies will be part of the SetBox application development, it is not critical since SetBox software is installed on desktop-class computers.