Software Architecture Description

RePlay Framework

LAWO Informationssysteme GmbH
Stockfeldstr. 1
76437 Rastatt
Germany

Phone: +49 (7222) 1001-0

Version:	0.11
Date:	2015-06-24
Author:	Dipl.-Inform. Bastiaan Hovestreydt Dr.-Ing. Karsten Köth Development
Document:	N01409

Versions

0.00	BH	2015-03-09	Document created
0.01	BH	2015-03-16	Rough draft
0.02	BH, KDK	2015-03-16	Document number assigned, TCP ports defined
0.03	BH	2015-03-18	First releasable draft
0.04	BH	2015-03-19	Small errors corrected
0.05	KDK	2015-04-02	Input modules: link to project plan, CSS example
0.06	KDK	2015-04-10	More additions, marked in red: Stop-id more general, From LAWO XML to LTG XML changed.
0.07	KDK	2015-04-14	Handover from KDK back to BH
0.08	BH	2015-05-21	Separated stop ID and index, restructured composing and output layers
0.09	BH	2015-05-27	Added diagnostics
0.10	BH	2015-06-03	Extended LTG/Content
0.11	KDK	2015-06-24	NewConfig and KeepAlive element removed.

Contents

Versions 2

1. Overview 4

2. Input modules 5
2.1 IO-Module 6

3. Parser modules 6
3.1 Y-Module 6

4. Parsing layer interface 7
4.1 LTG XML 7
4.1.1 LTG/Content 8
4.1.2 LTG/Control 11

5. Controller (with cache) 11

6. Composing layer interface 12
6.1 LTG/Display 13
6.2 LTG/Sign 14
6.3 LTG/Audio 14

7. Webserver 14
7.1 Webserver configuration 14

8. Output layer interface 14
8.1 Infotainment 14
8.2 LED signs and audio 17

9. Diagnostics 17
9.1 LTG/Diagnostics 18

10. Examples 19
10.1 Line number as image 19
10.2 Stop request 20

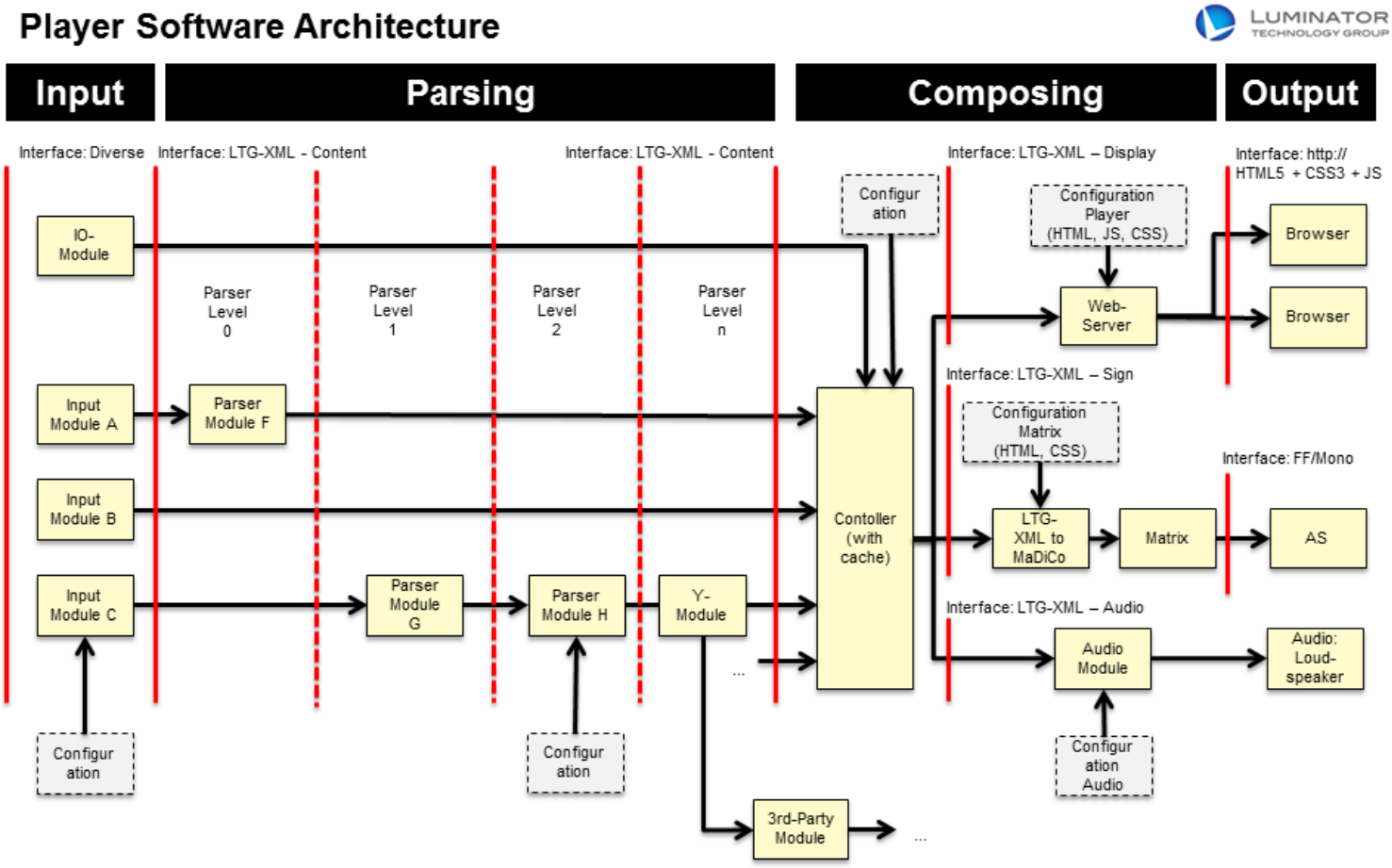
This document is work in progress and therefore subject to change.

1. Overview

The player software architecture is split into four main layers:

- Input
- Parsing
- Composing
- Output

The input layer handles the translation from OBU specific protocols to LTG XML. The parsing layer is organized in a pipe structure; information is sent as LTG XML from one parser module to the next. There can be any number of parsing modules, depending on the requirements of the specific project. However each parsing module has to be on its own parser level.



The composing layer gathers the bits of information that trickle down the pipe, combines them to a logical block of information and sends it to the output layer, where it is displayed.

The goal of this approach is to create a highly flexible system that is capable of handling the diverse requirements of our customers. In most cases choosing and configuring the existing modules should be sufficient to meet demands. At times a new input or parser module can be needed; however the source code of the modules in the composing and output layers should never have to be touched when adapting the system to a new customer.
Splitting the infotainment software in many small modules also makes it easier to debug and maintain each module.

2. Input modules

For each different protocol spoken by an OBU there is one dedicated input module, which handles the translation to LTG XML. This eliminates the need to handle different protocols further down the pipe and all the other modules (parsing, composing, output) can focus on their core function, instead of having to deal with lots of different protocols.

Necessary Input Modules:

- IBIS-Wagenbus (plain old OBU – minimal standard)
- InputISI (INIT OBU)
- SOAP (Trapeze OBU)
- IP-KOM (IVU OBU)
- SEP (Mobitec Access Web)
- MaDiCo-IS (GSP OBU)
- Inform (Airport busses OBU)
- IBISoverIP (Atron OBU, launching customer BVG)
- Ximple (Gorba Infotainments)
- GTFS (General Transit Feed Specification, see [1])
- GTFS-realtime
- GIRO HASTUS (Schedule generation software)
- … (See Project Plan for current list of input modules)

Possible Input Modules:

- TE (Trans Electronic OBU)
- ADV (Advertising)
- … (See Project Plan for current list of input modules)

Additional input modules can be implemented as needed by everyone inside the Luminator Technology Group.

As little work as possible is done in the input modules. Every function, that is not absolutely necessary in the input module should be implemented as a standalone parser module (e.g. replacement of numbers/strings with pictures, changing to or from all-caps, …). This way the functionality can be used from different input modules, when reconfiguring the system for another client.

[1] http://www.gtfs-data-exchange.com

2.1 IO-Module

The IO-Module is a special case of an input module. It communicates via MaDiCo[2] directly with the controller. It is used as hardware abstraction layer and sends updates about the state of the input signals like “stop request” and “door open”.

3. Parser modules

Recurring tasks that are independent of the communication with the OBU will be implemented in standalone parser modules. Each task should be realized in its own module. Each parser module can be used multiple times with different configurations inside the pipe structure.
Parser modules alter certain elements of the XML data that is passed through them. The selection of the altered elements can be done with a whitelist, blacklist or by using XPath as a query language. The way in which the data is altered is determined by the purpose of the parser module and its configuration.

Examples for parsing modules are:

- **Text replacement**
  The most important and most versatile module would be a simple text replacement module. It executes the preconfigured XPath query and applies a regular expression to the values of all matching elements.
- **Date and time formatting**
  This module is very similar to the text replacement module, except instead of applying regular expressions, it will parse date/time strings and put them back in the desired format, specified by configuration. As intermediate step, shifts of time due to time zones or daylight saving time can be applied. As an example, “2015-03-18T12:11:00+01:00” could become “in 10 minutes” or “Wednesday, March 18, 2015”.
- **Database lookup**
  This kind of module can be used to link transmitted information by the OBU to a database stored in the infotainment system. With this module the existing value of the selected XML element is used as a database key and replaced by the looked up value.

Parser modules can be implemented as needed by everyone inside the Luminator Technology Group.

3.1 Y-Module

In some cases, where the information of one input module has to reach more than one output module (e.g. infotainment screens, matrix signs, third party modules, …), the XML telegrams have to be duplicated by a special module in the parsing layer, called Y-Module.
The Y-Module is exactly like a normal parsing module (exists on a certain parser level, reads XML, outputs XML), except it allows for more than one parser levels to be specified as output level and it doesn’t change the XML data.

[2] See document IS_Doku_IOModule

4. Parsing layer interface

The modules in the parsing layer all share the same interface: a standard TCP/IP connection to transmit XML. The connection is kept open as long as its module is running and reestablished by the client if lost.
The parsing pipeline can have multiple sources (i.e. input modules), multiple intermediate stations (i.e. parsing modules) and multiple sinks (e.g. the controller or third party modules).
The pipeline is structured in different parser levels on which at most one parsing module can exist. Each parser level has its own TCP port, starting at 45000 for parser level 0, 45001 for parser level 1, etc. The sinks are organized in a similar way: each sink has its own TCP port, starting at 46000 for the controller.
This means each module has to know its place within the pipeline through configuration:

- The input modules need to know the parser level or sink to send their data to (to know which port to connect to)
- The parsing modules need to know the level they reside on (to open the corresponding port) and the parser level or sink to send their data to (to know which port to connect to)
- The sinks need to know which sink they are (to open the corresponding port)

It is possible to send any XML strings through the pipeline, as long as there is a sink that can handle it. For clarity reasons however, LTG XML should be used whenever possible.

For easier debugging, each module adds an attribute to the XML tags it has created or changed:

- Input modules add their name as the source attribute
  `src="Name of input module"`
- Parser modules add their name as an attribute representing their parser level (pl0, pl1, …) `pl0="Name of parser module"`

This attributes are not used programmatically, but only to reconstruct the path of information when debugging. In some cases it can be useful to include the action that was performed in parenthesis after the name:

```
pl0="Name (action performed)"
```

4.1 LTG XML

LTG XML is the main protocol spoken in the parsing layer. It is specified in the LAWO document N01707.
Each LTG XML telegram is enclosed in a root element called “LTG”.

```
<LTG>
    …
</LTG>
```

The two sub-elements interpreted by the controller are LTG/Content and LTG/Control.

4.1.1 LTG/Content

```
<LTG>
  <Content>
    ...
  </Content>
</LTG>
```

The LTG/Content sub-element is used to transport the informational content from the input modules (sources) to the composing layer (sinks). Values carried in LTG/Content do not need to be displayed right away (or at all), but rather describe the underlying information of what should be displayed. Only after the last parser module do the fields in LTG/Content represent exactly what should be displayed (if it is displayed).
LTG/Content XML text fields can contain any value as properly escaped XML string (e.g. “&”) which, after unescaping, is used directly as HTML in the output layer. Because of this, converting a line number to an image for example can be handled by a simple text replacing module in the parsing layer (see section 10.1). The escaping and unescaping of the strings is handled automatically by the XML library.

For every piece of information received from the OBU and sent through the parsing layer, LTG XML should contain an appropriate element.[3]

The defined elements (incomplete list) in LTG/Content are:

Element	Description
Time	Current time
Date	Current date
Line	Current line number
Via*	Parent element for an intermediate destination; can occur multiple times.
Destination	Destination of the vehicle
VehicleType	Type of the vehicle (like bus, tram, train, etc.)
InformationText	Additional information text relevant to the current line
TickerText*	Parent element for a text item of the news ticker; can occur multiple times.
Stop*	Parent element for a stop; can occur multiple times. Used to list the stops along the line.
CustomElements	Parent element for any elements that an input module may need to send that isn’t defined in LTG/Content otherwise. Any element can be used freely below this element.

The LTG/Content/Via, LTG/Content/Stop/Via and LTG/Content/Stop/Connection/Via elements can have the following sub-elements:

Element	Description
Index!	Index of the intermediate destination
Text	Text describing the intermediate destination

[3] LTG XML can be easily extended if not.
* Element can occur multiple times.
! Element must be used if the parent element is used.

The LTG/Content/TickerText element contains:

Element	Description
Index!	Index of the text item for the news ticker
Text	Text item for the news ticker

The LTG/Content/Stop element conrtains:

Element	Description
Index!	Index of a stop along the line. The index should be continuous throughout the line, starting at 1.
Id	ID to identify the stop. The stop ID is dataset unique.
Name	Name of the stop
Line	Line number at the stop (if the line number changes throughout the trip)
Via*	Parent element for an intermediate destination (if the list of intermediate destinations changes throughout the trip); can occur multiple times.
Destination	Destination of the vehicle (if the destination changes throughout the trip)
VehicleType	Type of the vehicle (if the type changes during the trip e.g. form bus to night bus)
ArrivalTime	Time at which the vehicle is scheduled to arrive at the stop
ArrivalDate	Date at which the vehicle is scheduled to arrive at the stop
RealArrivalTime	Calculated time of arrival at the stop, depending on the current position.
RealArrivalDate	Calculated date of arrival at the stop, depending on the current position.
DepartureTime	Time at which the vehicle is scheduled to leave the stop
DepartureDate	Date at which the vehicle is scheduled to arrive at the stop
RealDepartureTime	Calculated time of departure from the stop, depending on the current position.
RealDepartureDate	Calculated date of departure from the stop, depending on the current position.
Platform	Platform where the vehicle stops
ExitSide	Side of the vehicle, where passengers can exit
InformationText	Additional information text relevant at the stop
Connection*	Parent element of a connection; can occur multiple times. Used to list the available connections at the stop.

The LTG/Content/Stop/Connection element contains:

Element	Description
Index!	Index of the connection. The index should be a unique (within the parent stop element) number between 1 and 9.
Line	Line number of the connecting vehicle
Via*	Parent element for an intermediate destination of the connecting vehicle; can occur multiple times.
Destination	Destination of the connecting vehicle
VehicleType	Type of the connecting vehicle
ArrivalTime	Time at which the connecting vehicle is scheduled to arrive at the stop

ArrivalDate	Date at which the connecting vehicle is scheduled to arrive at the stop
RealArrivalTime	Calculated time of arrival of the connecting vehicle at the stop, depending on its current position.
RealArrivalDate	Calculated date of arrival of the connecting vehicle at the stop, depending on its current position.
DepartureTime	Time at which the connecting vehicle is scheduled to leave the stop
DepartureDate	Date at which the connecting vehicle is scheduled to arrive at the stop
RealDepartureTime	Calculated time of departure of the connecting vehicle from the stop, depending on its current position.
RealDepartureDate	Calculated date of departure of the connecting vehicle from the stop, depending on its current position.
Platform	Platform where the connecting vehicle stops
InformationText	Additional information text relevant to the connection
Guaranteed	Indicator whether the connection is guaranteed

Example:

```
<LTG>
  <Content>
    <Time>09:46</Time>
    <Date>2015-03-18</Date>
    <Line>123</Line>
    <Via>
      <Index>1</Index>
      <Text>Town center</Text>
    </Via>
    <Via>
      <Index>2</Index>
      <Text>Marketplace</Text>
    </Via>
    <Destination>Main Station</Destination>
    <Stop>
      <Index>1</Index>
      <Name>City Center</Name>
      <ArrivalTime>9:47</ArrivalTime>
      <Connection>
        <Index>1</Index>
        <Line>456</Line>
        ...
      </Connection>
      <Connection>
        <Index>2</Index>
        ...
      </Connection>
    </Stop>
    <Stop>
      <Index>2</Index>
      <Connection>
        <Index>1</Index>
        ...
      </Connection>
      ...
```

```
    </Stop>
    ...
    <Stop>
      <Index>23</Index>
      <Name>Main Station</Name>
      <ArrivalTime>10:28</ArrivalTime>
      <Platform>2b</Platform>
      ...
    </Stop>
    ...
    <CustomElements>
      <SomeKindOfCustomElement>xyz</SomeKindOfCustomElement>
      <AnotherOne>123</AnotherOne>
    </CustomElements>
  </Content>
</LTG>
```

4.1.2 LTG/Control

```
<LTG>
  <Control>
    ...
  </Control>
</LTG>
```

The LTG/Control sub-element is used to transmit control events to the controller module. It can contain the following elements:

Element	Description
StopIndex	Used to tell the controller at which stop the vehicle currently is. This way the controller can query the cache for previously sent LTG/Content/Stop elements and use StopIndex to select the next few stops on the line.
SequenceId	The SequenceId element can be used to switch between displaying different sets of information (e.g. next stops, connections, advertisement, …).
StopRequest	This flag can be used if the OBU tells the infotainment system whether a passenger has requested the vehicle to stop at the next stop. Normally this information can be obtained through the input signals of the infotainment directly.
GPS	Contains current GPS coordinate from external GPS receiver or simulator (Format: todo)
Identify	Used by the composing layer modules to identify themselves to the controller (see section 6) and the diagnostics module (see section 9).

5. Controller (with cache)

The cache module gathers and stores all information sent as LTG/Content XML. It accepts multiple TCP connections to allow the information from multiple input modules to flow together.

Each LTG/Content element triggers a change event after being read into the cache completely. This event notifies the controller which sub-elements have been updated.
The controller can react upon several different events:

- The change events from the cache
- Time based triggers
- Events sent by the IO-Module via MaDiCo
- Events from the OBU sent via LTG/Control

The configuration of the controller allows rules to be defined, that determine its behavior. These rules have three parts:

- **Events**
  Define which of the above events will trigger the execution of the rule (e.g. change of LTG/Content/Line).
- **Conditions**
  Only if these conditions apply, the rule will be executed (e.g. LTG/Control/SequenceId equals 3).
- **Output**
  The output part contains the data elements of the cache, which should be handed over to the webserver, matrix or audio module.
  - **Output of LTG/Display data elements**
    Contains a list of data elements stored in the cache, which should be sent via LTG/Display XML (see section 6.1) to the webserver module (e.g. send “LTG/Content/Line” as field “Line”).
  - **Output of LTG/Display visibilities**
    Contains a list of field names and corresponding visibilities to be sent via LTG/Display XML (e.g. field “StopRequest” is visible).
  - **Output of LTG/Sign**
    Contains a list of data elements stored in the cache, which should be sent via LTG/Sign XML (see section 6.2) to the matrix module.
  - **Output of LTG/Audio**
    Contains a list of data elements stored in the cache, which should be sent via LTG/Audio XML (see section 6.3) to the audio module.

6. Composing layer interface

As with the parsing layer interface, LTG XML is used as communication protocol within the composing layer as well. However in this case, the client and server roles are reversed in regard to the flow of information (the controller sends information to the webserver, matrix and audio modules, but acts as TCP server). This is done to prevent the need to specify which modules should be fed with information from the controller. Instead the modules register themselves with the controller to get updates. This also means that the controller can use the same interface (and port) to get information from the parsing layer as it uses to send information further down the composing layer.

The following steps occur during the communication:

- **Composing layer module (e.g. matrix module) connects to the controller**
  The connection is established via TCP on port 46000
- **Composing layer module identifies itself**

The module sends an LTG/Control structure specifying the type of the connecting module e.g.:

```
<LTG>
  <Control>
    <Identify>
      <Type>Sign</Type>
    </Identify>
  </Control>
</LTG>
```

The type can be `Infotainment`, `Sign` or `Audio` depending on the information the connecting module wants to receive.

- **Controller sends appropriate information to the connected module**
  The controller sends information that is relevant to the registered module over the existing connection. This happens when such information arises

The connection is left open as long as the composing layer module desires to be informed by the controller.

6.1 LTG/Display

The controller speaks to the webserver in LTG XML also, only in this instance using the LTG/Display node. The differences to LTG/Content are:

- The tags below LTG/Display are in a more or less flat structure, whereas LTG/Content has a hierarchical structure
- LTG/Display contains only information, that affects the content of the screen directly, while LTG/Content is yet to be transformed by the controller
- LTG/Content has a predefined structure and predefined fields, whereas the [FieldID] in LTG/Display/[FieldID] can take any name and refers dynamically to a field id in the DOM structure of output HTML file

LTG/Display is structured as follows:

```
<LTG>
  <Display>
    <Line>
      <Value>S1</Value>
      <Visibility>true</Visibility>
    </Line>
    <Destination>
      <Value>Karlsruhe Marktplatz</Value>
      <Visibility>true</Visibility>
    </Destination>
    <StopRequest>
      <Visibility>false</Visibility>
    </StopRequest>
    ...
  </Display>
</LTG>
```

Omitting the Value or Visibility sub-tags of an element will leave the respective property of the item untouched. Only listed elements will be updated. All listed elements in one LTG/Display XML telegram are changed in one action.
If an input module doesn't need the functionality of the cache and controller, it can bypass them by sending LTG/Display XML directly. This passes transparently through the controller.

6.2 LTG/Sign

TODO

6.3 LTG/Audio

TODO

7. Webserver

The job of the webserver is to receive the LTG/Display XML telegrams and relay the information to the connected browsers. It behaves as a single stateful display software and masks the existence of multiple browsers, which each could (but shouldn't) have their own state.
For this the latest value and visibility of each tag under the LTG/Display node is not only relayed directly to the connected browsers, but also stored inside the webserver module. When a browser (re)connects, all stored fields will be sent to bring the newly connected browser into the same state as all other connected browsers.

7.1 Webserver configuration

The customer specific layout of the infotainment is defined as part of the webserver configuration. Additionally the browser may see some files that are part of the framework.

File	Part of	Description
index.html	Customer configuration	Is delivered directly (with small modifications) from disc by the webserver
Static files (CSS, images, ...)	Customer configuration	Are delivered directly from disc by the webserver
core.js	Framework	Is embedded into the webserver and delivered from memory
custom.js	Customer configuration	Optional; used to override certain functions of core.js

The minimal configuration contains only an index.html. In this file the customer specific layout is defined (making use of additional static files like CSS and images).

8. Output layer interface

8.1 Infotainment

The connection between the webserver and the browser consists solely of well-established web technology. This means, that the browser has not to be custom built, but can instead use any modern browser engine (e.g. WebKit). Furthermore a normal desktop browser can be used for debugging, without any changes to the system or the browser.

The used technologies HTML5, CSS3, WebSockets and JavaScript all use HTTP as transport. This way only one port (default port 80) is needed.
The server transmits the base HTML/CSS files (containing the layout) when the browser connects. All content gets delivered incrementally via JSON over a WebSocket connection.

When delivering the index.html file, the webserver adds `<script>` tags to the HTML file to add the framework specific JavaScript code to the website.

Example for an original index.html:

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
    <link rel="stylesheet" type="text/css" href="layout.css">
  </head>
  <body>
    <div id="Line"       ></div>
    <div id="Destination"></div>
    ...
  </body>
</html>
```

Modified index.html delivered by webserver:

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
    <link rel="stylesheet" type="text/css" href="layout.css">
  </head>
  <body>
    <div id="Line"       ></div>
    <div id="Destination"></div>
    ...
    <script src="core.js" type="text/javascript"></script>
    <script src="custom.js" type="text/javascript"></script>
  </body>
</html>
```

Note that the `<div>`-tags are empty, since they will be filled by JavaScript code triggered through WebSocket messages.
The corresponding CSS file `"layout.css"` looks like:

```
    /* General settings */
    * {
            font-family: Helvetica, Arial, sans-serif;
            font-size: 30px;
            font-weight: bold;
            left: 0px;
            position: fixed;
            top: 0px;
            padding: 0px;
            border: 0px solid black;
            margin: 0px;
       }
    /* Line */
    #Line {
            color: white;
            background-color: rgb(31,27,32);
            text-align: center;
            left: 0px;
            top:  0px;
            width: 100px;
```

```
            height: 85px;
    }
```

The framework file core.js has the following tasks:

- Establish a WebSocket connection to the server and reconnect on connection loss
- Send regular WebSocket messages to the server to prevent a timeout and to give the server the possibility to detect errors[4]
- Handle received JSON messages via WebSocket

The WebSocket messages from the server can do one or more of these things:

- Update the value of an element with a given ID
- Update the visibility of an element with a given ID
- Trigger a reload of the page (only needed if an irrecoverable error is detected by the server)

The messages from the server are structured as follows:

```
{
  "Update":
  [
    {"ID":"Line", "Value":"S1", "Visibility":true},
    {"ID":"Destination", "Value":"Karlsruhe Marktplatz", "Visibility":true},
    ...
  ],
  "Reload":false
}
```

Omitting value or visibility of an element will leave the respective property of the item untouched. Only listed elements will be updated. If the message contains a reload flag and its value is true, core.js will trigger a reload of the website.
The structure of the JSON message allows it to be extended in the future.

core.js runs in the browser and handles the received messages:

```
var obj = JSON.parse(message);
beginUpdate();
if (obj.hasOwnProperty("Update") && Array.isArray(obj.Update))
{
  for (i = 0; i < obj.Update.length; i++)
  {
    if (!obj.Update[i].hasOwnProperty("ID"))
      continue;
    if (obj.Update[i].hasOwnProperty("Value"))
      setValue(obj.Update[i].ID, obj.Update[i].Value);
    if (obj.Update[i].hasOwnProperty("Visibility"))
      setVisibility(obj.Update[i].ID, obj.Update[i].Visibility);
  }
}
if (obj.hasOwnProperty("Reload") && obj.Reload)
  reloadPage();
endUpdate();
```

[4] 2015-03-12: keep alive message format not yet defined, should contain information about the currently displayed information in the website

In core.js the functions

- setValue(ID, value)
- setVisibility(ID, value)
- beginUpdate()
- endUpdate()
- reloadPage()

are implemented as default handlers. For example like this:

```
function setValue(ID, value)
{
  var ele = document.getElementById(ID);
  if (ele != null)
  {
    ele.innerHTML = value;
  }
}

function setVisibility(ID, value)
{
  var ele = document.getElementById(ID);
  if (ele != null)
  {
    ele.style.visibility = value ? 'visible' : 'hidden';
  }
}
```

custom.js can overwrite these methods to add customer specific behavior to a customer configuration, like CSS/JavaScript animations or changes to the content as it is received (e.g. write “7A” as “7_A”).

8.2 LED signs and audio

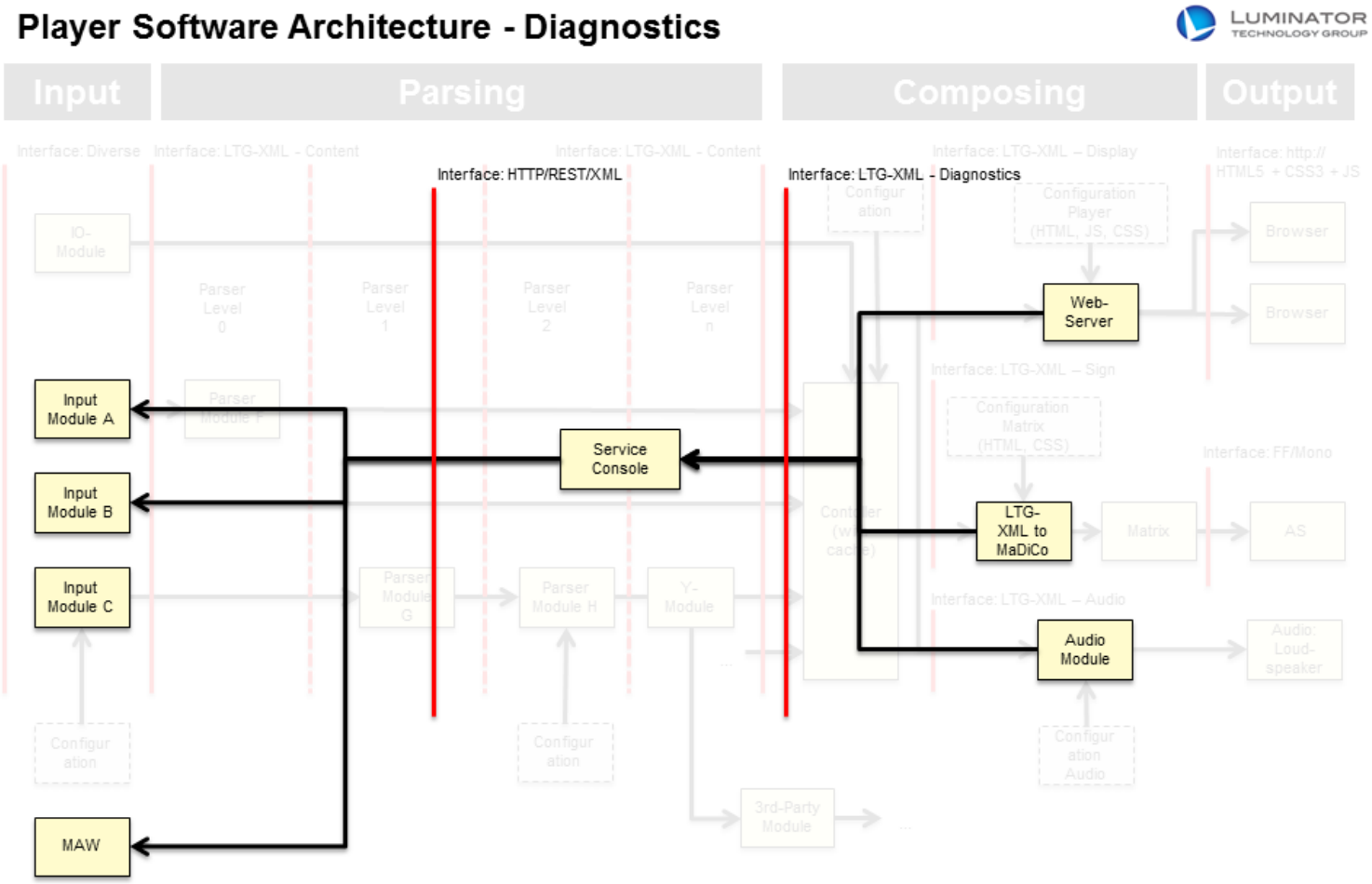
The connection between the Matrix-Module and the LED signs will be made by using existing protocols. This can be FF or Mono for example.
In a similar way the Audio-Module uses the existing architecture to drive the loudspeakers.

9. Diagnostics

To gather diagnostics data from the system, the existing module “Service Console” will be extended. It already has the ability to query certain bits of information about the system and present those via a HTTPS/REST/JSON interface to other software or via a website. For usage within the RePlay framework this interface will be extended to also accept HTTP (as opposed to HTTPS) connections without authentication, when the client connects from a specified network interface or IP address. Additionally the service console will be able to generate XML instead of JSON upon request.

This way the service console can continue to use its existing interface with minor changes, while other modules can easily use it to get data in their preferred format without the hassle of establishing a HTTPS connection.

The interface between the composing layer and the service console will be the same as the interface between the composing layer and the controller (see chapter 6. Composing layer interface): the service console acts as TCP server and the composing layer modules establish a connection. This is done to prevent the need to



specify which modules should be queried for diagnostics information from the service console. Instead the modules register themselves with the service console to provide information.

The following steps occur during the communication:

- Composing layer module (e.g. matrix module) connects to the service console
- Composing layer module identifies itself via LTG/Control/Identify
- Service console sends appropriate request for diagnostics information to the connected module, when such information is needed
- Composing layer module answers the request

The connection is left open for further requests for diagnostics information.

9.1 LTG/Diagnostics

TODO

10. Examples

10.1 Line number as image

Consider the following IBISoverIP telegram:

```
l17\r
```

It gets converted by the IBISoverIP input module the corresponding LTG XML:

```
<LTG>
  <Content>
    <Line src="IBISoverIP">17</Line>
  </Content>
</LTG>
```

In a parser module certain elements[5] of the XML structure are altered; in this case LTG/Content/Line. The text value of the selected element will be changed with text replacement[6] to represent an image:

```
<LTG>
  <Content>
    <Line src="IBISoverIP" pl0="Replace">&lt;img src=&quot;17.png&quot;/&gt;</Line>
  </Content>
</LTG>
```

The controller decides that this information should be displayed directly and triggers a new LTG XML telegram to alter the display state:

```
<LTG>
  <Display>
    <Line>
      <Value>&lt;img src=&quot;17.png&quot;/&gt;</Value>
    </Line>
  </Display>
</LTG>
```

The webserver sends this information as JSON formatted string over the WebSocket connection to each connected browser:

```
{
  "Update":
  [
    {"ID":"Line", "Value":"<img src=\"17.png\"/>"}
  ]
}
```

After receiving, core.js handles the changes specified in the JSON string by updating the DOM structure of the displayed HTML site:

```
<!DOCTYPE html>
<html>
  <head>...</head>
  <body>
    ...
    <div id="Line"><img src="17.png"/></div>
```

[5] These could be specified with XPath
[6] E.g. regular expressions

```
    ...
  </body>
</html>
```

This triggers the browser engine to fetch the image “17.png” from the server and redraw the page with the loaded image.

10.2 Stop request

The IO module notifies the controller that the value for the stop request input signal has changed:

```
<?xml version="1.0"?>
<MaDiCo>
  <InputReply>
    <InputSTOP>ON</InputSTOP>
  </InputReply>
</MaDiCo>
```

The controller decides that this information should be displayed directly and triggers a new LTG XML telegram to alter the display state:

```
<LTG>
  <Display>
    <StopRequest>
      <Visibility>true</Visibility>
    </StopRequest>
  </Display>
</LTG>
```

The webserver sends this information as JSON formatted string over the WebSocket connection to each connected browser:

```
{
  "Update":
  [
    {"ID":"StopRequest", "Visibility":true}
  ]
}
```

After receiving, core.js handles the changes specified in the JSON string by updating the style of the element with id “StopRequest” and setting its visibility to “visible”. This will trigger the browser engine to repaint the page with the “StopRequest” element visible.