System Architecture for Variable Message Signs

Hannes Kulovits  
swarco Futurit Ges.m.b.H.  
Software Development  
Zichygasse 8  
A-1140 Vienna  
kulovits.futurit@swarco.com

Christoph Stögerer  
swarco Futurit Ges.m.b.H.  
Software Development  
Zichygasse 8  
A-1140 Vienna  
stoegerer.futurit@swarco.com

Wolfgang Kastner  
TU Vienna  
Institute of Automation  
Treitlstrasse 1  
A-1040 Vienna  
k@auto.tuwien.ac.at

Abstract

Traffic control and information systems are used in modern traffic technology for information propagation from a higher order control unit to the traffic participant. In up to date systems, the user interface for the traffic participant is provided via programmable signs displaying traffic jam warnings, speed limits or route diversion. These signs can be switched on or off and fed with arbitrary data corresponding to the present traffic situation. However, signs are manifold in size, functionality and means to communicate with them. This paper proposes a component based software architecture that allows a rapid integration of such (and future) signs into existing traffic management systems.

1. Introduction

Traffic signaling has its roots in the late 1970s when fiber optic signs for speed limit signalization appeared. At that time, traffic signs based on dot matrices were mainly accessed on-site via specialized hardware and digital I/O and were able to display different, but fixed signalization pictograms (Figure 1).

Soon, first simple serial communication protocols superseded digital I/O lines and offered to better fulfill upcoming demands for monitoring the sign facilities (e.g. the light source).

With the advent of light emitting diodes (LEDs) enabling rising performance regarding light output, reduction of power consumption and higher lifetime in contrast to the light sources used in the fiber optic technology, sign manufacturers changed over using LED-technology for signaling issues. At the beginning these diodes were mainly found in the advertising domain where monochrome LED-screens and later on full color technology composing pixels of three LEDs in red, green and blue (RGB) color were used. Shortly after, the concept was taken over to the traffic domain as well.

Starting from relatively simple traffic signs (e.g. again for speed limit signalization) with a limited number of depictable pictures (“limited signs”), applications were getting more complex. So called “programmable traffic signs” were coming up, providing the possibility to display text and pictures that are not predefined (Figure 2).

Figure 1. Traffic sign dot matrix

Figure 2. Programmable traffic sign

Since then signs for the purpose of displaying one of a number of legends that may be changed or switched off as required, were named Variable Message Signs in the context of road applications [13]. Besides Variable Message Signs (VMS), Dynamic Route Information Panels (DRIP) as outlined by Schouten et.al. [11] or Graphical Route Information Panels (GRIP) as presented by Alkim et.al. [2] were introduced.

Subsequently, different combinations and variations of limited and programmable traffic signs emerged with increasing complexity and demands for safety issues (reliability, dependability). For instance, the requirements changed from initially more or less simple monitoring of the light source to consistently supervising electronic...
components, inputs and outputs (e.g. data connection, power supply). Also detection units like humidity and luminosity sensors were integrated in such road traffic devices.

The underlying communication had to be adapted and re-designed as well. Regarding physical media and protocol standards, fiber optic communication lines and the use of the Internet Protocol (IP) became an option in the traffic control domain and broadened the range of possibilities for the development of new applications especially for remote monitoring and control.

Our work outlines problems of today’s traffic management systems regarding communication standardization as well as customization of VMS and presents a configurable system architecture to overcome increasing demands regarding flexibility and enabling short development times for new features.

The remainder of this paper is structured as follows: Section 2 presents the basic structure of a traffic management system and its components and discusses the main communication standards commonly used in traffic management systems. In Section 3 the goals for integrating VMS in a general framework are presented. While Section 4 explains the component based software architecture, Section 5 shows the use of factory patterns to achieve protocol independence and to setup a generic transmission framework. Finally, Section 6 gives an outlook on future directions within the scope of this work.

2. Traffic management systems

Traffic management systems are composed of multiple control and monitoring entities coupled by different communication facilities and protocols (Figure 3). At the highest level a Traffic Management and Information Center (TMIC) collects data from Sub-Stations (SS) and provides it to its users for global strategies concerning road traffic monitoring and control. Sub-Stations are responsible for intermediate data collection. They are interconnected to one or more Local Control Units (LCUs) that are in turn wired to sensors (e.g. detector loops, radar detectors) and actuators (e.g. gates, traffic lights, VMS) and are responsible for data processing and autonomous control jobs. A LCU is composed of a Control Module and (optional) I/O-Converters. It is the task of an I/O-Converter to translate incoming requests from the LCU to a vendor specific protocol and respond to the LCU adhering to a standardized communication protocol (see below).

The number of levels involved depends on size and complexity of the overall traffic management system. The minimum system configuration can be composed of some autonomously acting LCUs. Applications in this case may
care for the visualization of successive speed reduction for a specific section of road. This could be handled for instance via three VMS with 500 m in between displaying 130 km/h, 100 km/h and 80 km/h, respectively. The maximum system configuration consists of multiple levels of control and monitoring facilities, where each level is designed for autonomous operation as a kind of fallback in case of breakdown of a higher order level facility. Applications in this case include sophisticated traffic jam collision detection mechanisms and control of VMS with instructions for rerouting.

Throughout the hierarchical structure, communication facilities differ in requirements depending on the levels involved. After several pilot projects were started, some regional restricted (de-facto) standards regarding communication protocols and facilities were introduced. The most important ones are listed in Table 1.

### Table 1. Communication standards in the traffic domain: overview

<table>
<thead>
<tr>
<th>Communication standard</th>
<th>Country of origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS [5]</td>
<td>Germany</td>
</tr>
<tr>
<td>TLS over IP [4]</td>
<td>Austria</td>
</tr>
<tr>
<td>DAP [14]</td>
<td>Netherlands</td>
</tr>
<tr>
<td>NTCIP [1]</td>
<td>US</td>
</tr>
</tbody>
</table>

Up to now, these communication standards are dedicated to specific communication levels of a traffic management system structure. Thus, one single standard cannot be applied to a fully deployed traffic management system, i.e. from its highest level A to its lowest levels D or E (cf. Figure 3). Table 2 shows for each mentioned protocol its position and the devices included.

### Table 2. Communication standards in the traffic domain: levels and devices involved

<table>
<thead>
<tr>
<th>Communication standard</th>
<th>Levels</th>
<th>Data exchange between device/device</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS over IP</td>
<td>A</td>
<td>TMIC-SS</td>
</tr>
<tr>
<td>TLS over IP</td>
<td>B</td>
<td>SS-LCU</td>
</tr>
<tr>
<td>TLS</td>
<td>C</td>
<td>SS-I/O-Conv.</td>
</tr>
<tr>
<td>DAP</td>
<td>(A−C)</td>
<td>TMIC or SS or LCU</td>
</tr>
<tr>
<td>NTCIP</td>
<td>(A−C)</td>
<td>TMIC or SS or TMIC</td>
</tr>
</tbody>
</table>

#### 3. Goals

The remainder of this paper focuses on the communication between end devices, especially VMS, and depending on the system structure – control entities on the next higher level. In the following, these control entities are summarized under the term Higher Order Control Units (HOCU). Typical instances of HOCUs can be LCU s, Sub-Stations, and TMICs. Moreover, VMS can be subdivided into sign units of different types (e.g. monochrome/bi-color/RGB signs, limited signs) where each sign unit is handled by a specific sign controller and its proprietary protocol. An exemplary VMS could consist of one limited sign part and three text lines (Figure 4).

![Figure 4. Composition of a VMS](image-url)
4. Software architecture

From an economic point of view, the development costs (especially for future extensions) have to be kept as low as possible. In addition, the ability to react to new markets needs (e.g., integration of protocols for communication with HOCUs) has to be done as fast as possible. Therefore our approach is to integrate the I/O-Converter's functionality as part of the LCU using component technology with the goal of shortening reaction times when adopting different protocols in a configurative and flexible way.

The following higher order requirements had to be met:

- The solution had to be designed in a highly flexible, adaptable and distributed way.
- The coupling of the components had to be held as low as possible to ease software updates (integration of further protocols) and minimize error propagation.
- The solution had to provide a high degree of stability and reliability. Therefore it is necessary to execute the software in some kind of system mode and start running it automatically after computer restart without user actions being taken.
- Most notably, the solution had to be based on a MS-Windows embedded operating system.

To meet the main requirements the following technology decisions were taken:

- The Rational Unified Process [9] was used as the software life cycle, due to the fact that this approach allows for an architecture-centric development process.
- To provide a range of services for component interaction, from services promoting component integration on a single platform, to component interaction across heterogeneous networks, COM/DCOM [7] was chosen.
- As the components must be able to (1) run in the background of the system, (2) start automatically in case of a system reboot, (3) execute without user login, the decision was taken to use MS-Windows Services.

Finally, the software architecture was divided into three independent and self-contained components, namely PLC Service, Communicator and Controller discussed in more detail in the remainder of this section (Figure 5).

4.1. PLC Service

The PLC Service is responsible for interfacing with the underlying hardware (in our case this is a third party PLC system) and provides an (abstract) interface to it. Thus, components that use the PLC Service are completely hardware independent. Consequently, whenever the PLC system is to be replaced, the only part of the software architecture that has to be adapted is the PLC Service.
Moreover, the PLC Service cares for a publisher-subscriber mechanism. It provides notifications for interested subscribers each time a modification of the process data occurs. Such interesting events can be for instance the change of luminosity or the change of temperature.

Via the PLC Service it is possible to define for interesting parts of the process data (so-called I/O points) a name and description, a corresponding unit, a factor for scaling purpose, a range for the scaled value, as well as a time slot. In case of an updated value that is stable within the specified time slot, the component provides it to subscribers via a dedicated connection point. Each I/O point is configurable regarding its address, where its current values can be derived from. In addition, the PLC Service allows configuration issues supporting to categorize and store the connected I/O points in groups called I/O-Categories.

The benefit of the PLC Service lies in this configurability which enables a customization of the whole system without changing the PLC Service application itself. The setup process consists of two important phases. The first one copes with the creation of the appropriate PLC program in the underlying PLC application the PLC Service is connected to. The second one tackles the connection to the underlying PLC application (i.e. communication facilities) and logging issues by configuring the system using an XML configuration file.

4.2. Controller

The Controller component has to manage the communication to a configurable quantity of different sign controllers belonging to one or more physical VMS. Thus, the most important task of the Controller is to provide a uniform interface for communicating with them. This interface is bundled into the so-called FuturitCom protocol (Figure 6).

![Sign controller 1](image1)
![Sign controller 2](image2)
![Sign controller n](image3)

The communication to sign controllers can be handled either via a serial connection or via UDP/IP. To make each sign controller as well as the whole VMS addressable, the sign units and their corresponding Controller need a unique address. For this reason, the address structure $x.y$ was introduced where $x$ stands for the address of the Controller and $y$ for the address of a sign unit. To operate all sign units at once in the custody of a specific Controller the sub-address is set to zero.

The component offers the full command language set of the FuturitCom protocol via the IFuturitCom interface. The interface includes functions like Poll() which is used to determine the status of the sign, SetBrightness(), for setting the current luminosity of the sign, or SetContent() to upload a specific content (e.g. a bitmap). Following the address scheme, each method needs the address of the Controller and the sub-address of the sign unit to uniquely identify the corresponding part of the sign. As a result the Controller component enables an absolute transparent, uniform and coherent way of controlling and monitoring the sign connected to the system.

Furthermore, the component cares for an event interface, called IFuturitComEvents, which must be implemented by each user of the Controller component. This interface contains a counterpart for each method contained in the IFuturitCom interface. A method of the IFuturitComEvents interface is activated whenever a response of a call to the corresponding IFuturitCom method is ready. For instance, to determine the status of a sign, the Poll() command must be used. After the Controller processed the command, it notifies the caller by invoking the corresponding OnPoll() method of the IFuturitComEvents interface.

Last but not least, the Controller is responsible for autonomous control. Therefore, it reacts to data received from the PLC Service, such as controlling the brightness of a VMS autonomously. For this reason, it can subscribe to the events provided by the PLC Service.

The configuration of the Controller is divided up into two parts: The first part deals with the communication, i.e. what kind of communication (serial or UDP/IP) shall be used, and which sign units are driven by the Controller. The second part covers issues for autonomous control (e.g. automatically dimming the sign) and safety relevant features (e.g. cyclic testing of LED-displays).

4.3. Communicator

Figure 3 outlined that a VMS is integrated into a traffic management system where different units are involved. As mentioned at the beginning of this section, one important requirement is to cut down on the need to use an I/O-Converter to enable the communication with the VMS. However, without an I/O-Converter the VMS has to handle the different communication standards on its own (cf. Table 2). Thus, the HOCU directly connects to the VMS in order to control it via varying commands.

The interface to the HOCU is provided by the Communicator component. The main challenge of the Communi-
is to perform the following tasks independently of the protocol used (e.g. FuturitCom, TLS, SiTOS [12]):

- wait for an incoming request from a HOCU.
- validate the request, i.e. send a positive acknowledgment, if it is correct, return a negative acknowledgment, if the request is invalid, or ignore the request in all other cases.
- delegate the request to the Controller component which communicates with the underlying VMS and its sign units.
- wait for a response from the Controller component.
- return the response to the HOCU.

5. Communication based on factory patterns

The whole system must be configurable in a way that allows to choose between different command language sets before run-time. This requirement especially bothers the Communicator, for it has to react to commands requested by the HOCU. To solve this problem two abstract classes were defined. The first one, named GenericTelegram, forms the base class for the different telegrams of the supported command language sets. It provides an interface for retrieving the telegram byte stream and its length. The second class, called Transmission, offers an interface for establishing and closing a connection, and transmitting telegrams. It acts as base class for the different supported forms of communication mechanisms (serial and UDP/IP).

![Diagram: Telegram factory](Figure 7)

These two classes build the essential components of a transmission framework used to send and receive telegrams. Both, the Controller and Communicator use the transmission framework as an integral part of their functionality. The Controller utilizes the framework to communicate with its connected sign controllers and the Communicator needs it to interact with the HOCU.

The main functionality of the transmission framework (send and receive commands) seems not to be a thrilling job and suggests that there should be ready-to-use solutions (i.e. libraries) available. However, in our case two requirements make it a very special task and, as far as we know, unique: transmitting of commands must be performed in a very generic way in terms of (1) the used command language set and (2) the desired communication mechanisms. To solve this problem we propose a design where the transmission framework rests upon the abstract classes GenericTelegram (Figure 7) and Transmission (Figure 8), only. In addition, for both classes the Abstract Factory Pattern [6] is chosen. This decision enables a very flexible design that can be easily adapted in two directions: the first one is related to the desired way of communication and the second one opens facilities to rapidly react to upcoming command languages. Forthcoming extensions have never to touch the transmission framework regardless whether an additional way of communication (e.g. TCP/IP) or a further command language set (e.g. NTCIP) has to be implemented.

![Diagram: Transmission factory](Figure 8)

By means of the method CreateTelegram(), a telegram of the configured protocol (e.g. TLS) can be created from a given byte stream, which will be provided via a parameter. The method throws an exception in case the telegram is not valid, i.e. if it does not correspond to the command language set (e.g. FuturitCom) or is syntactically incorrect. If the telegram is valid, it returns an object of the created telegram represented by its superclass GenericTelegram.
Figure 9 shows, in a simplified way, how a command is processed by the system including the Communicator and Controller component. The Communicator uses the previously explained transmission factory to provide an interface (serial or UDP/IP) to the HOCU. This framework establishes the connection to the HOCU and thus provides the Communicator with the commands sent by the HOCU. Vice versa, it provides dedicated methods for sending commands to the HOCU.

A byte stream received by the Communicator is checked, whether it is valid (syntactically correct) and belongs to a supported command language. This is done by passing the byte stream to the protocol specific implementation (predefined by the configuration) of CreateTelegram() which creates the corresponding telegram and returns a reference to its base class (GenericTelegram). Next, a mapping to the corresponding COM method is accomplished. Therefore the kind of telegram (e.g. SetBrightness) and the parameters (e.g. brightness level) must be accessible to assemble the COM call. Afterward, the corresponding Controller method can be invoked. The corresponding response is delivered via the implementation of the IFuturitComEvents interface.

The Controller’s challenge is to carry out the command (e.g. adjusting the brightness of the sign) and to respond to the COM call via the event sink to notify the caller of the result of the processing. Depending on the configuration, the Communicator blocks until the response from the Controller arrives or a certain timeout expires. When the Controller replies within the timeout, the received response event is translated into the corresponding answer telegram of the configured command language.

6. Conclusion and outlook

The software architecture presented followed an extensive requirements analysis with the primary goal of eliciting today’s communication mechanisms in the traffic management domain. For describing the architecture, the Rational Unified Process and a five-view approach [8] which covers a logical view, implementation view, process view, deployment view and Use-Case view was applied. E. Gamma’s design patterns and the second volume of the POSA series [10] can be seen as the most important references for the system architecture and design.

Regarding the implementation, some limitations exist that pave the way for our future work. As already explained, the task of the Communicator is to accept commands (which, in general, are telegrams) of any standardized protocol, hand them over to the Controller for further processing and forwarding the response to the caller. Actually, this task could be embedded into an arbitrary I/O-Converter. However, in the current version of the implementation, such an I/O-Converter must be based on an architecture supporting a MS-Windows operating system. To become platform independent, the next step is to implement the Communicator based on the Apache Portable Runtime (APR) [3] and thus avoid future vendor specific lock-ins.

References
