Open Traffic Systems - An Extensible Middleware for Distributed Traffic Management Systems

Daniel Bültmann, Reinhold Gebhardt, Communication Networks (ComNets), RWTH Aachen University, Faculty 6, Aachen, Germany, {dbn|geb}@comnets.rwth-aachen.de

Hanfried Albrecht, AlbrechtConsult, Aachen, Hanfried.Albrecht@AlbrechtConsult.com

Thilo Schön, GEVAS software, Munich, thilo.schoen@gevas.de

ABSTRACT

Public bodies in the role of buyers and operators of traffic management systems are bound to public tendering laws stipulating a competition for the system procurement to benefit from cost reduction and new features. This situation results in manufacturer-mixed systems and distributed system architectures. Communication protocols are one of the key enablers for successful deployment of such systems. Traffic management systems with their long life-cycles impose special demand on extensibility and adaptability of communication protocols. In this paper a middleware as one essential element of such distributed communication architectures will be described. The overall concept is called Open Traffic Systems (OTS).

INTRODUCTION

City traffic control and management systems are migrated increasingly from monolithic to distributed systems. Subsystems as parts of such distributed systems accomplish independent tasks and cooperate with other subsystems to accomplish overall tasks in the functional sharing. To meet the increasing functional and qualitative requirements of operators of such systems, framework conditions must be created so that suppliers of specific functions or subsystems can integrate their products into existing systems. Integration and cooperation of subsystems requires corresponding mechanisms. This can often be satisfied by the provision of supporting communication protocols. These must meet the requirements of open standards, so that manufacturers can develop and offer products independently of any other supplier.

A second, not less important requirement on protocol standards is not to restrict further growing of the range of functions or of the functions complexity. Therefore, a protocol architecture is needed which can be adapted to changed requirements flexibly and which supports compatibility to existing realizations in the context of a quality assured process.

In the context of the R&D initiative “Traffic Management 2010” of the German Federal Ministry of Economics and Technology (BMWi) the project Dmotion [2][3] aims to realise a coordinated strategy for traffic guidance and control in Greater Düsseldorf between the city of Düsseldorf and the state North Rhine-Westphalia which is responsible for the motorway system. In addition, private traffic information provider shall be able to request information on the general traffic conditions. Communication related tasks within this project are to acquire, analyse and assess all requirements on communication by subsystems and to use the results for the definition of a communication architecture. In this paper a middleware as one essential element of such a communication architecture will be described. The overall concept is called Open Traffic Systems (OTS). Necessary steps to meet the general requirements for an open specification, adaptability and extensibility will be discussed.

RELATED WORK

The need for communication standards in traffic management systems gets visible by the activities of the respective standardization groups. DATEX II [4] represents a European standardization initiative which also integrates predecessor activities like OTAP (Open Travel data Access Protocol). It aims to improve the exchange of traffic and travel information between traffic management and traffic information centres. The primary focus of this standardization activity is on data models, rather than on communication protocols.

Pilot projects for the integration of traffic data sources are in progress or planned in the European room. The Netherlands National Datawarehouse (NDW) project [5] of the Ministry Rijkswaterstaat was started in 2005. It shall collect and make available information on traffic densities and traffic flows in provinces and city areas. The efforts are quite similar to the upcoming German meta data platform project of the Federal Ministry of Traffic.

With its primary focus on a general purpose and light-weight communication protocol suite OTS complements DATEX II very well. In a communication layer view OTS protocols are to be placed beneath the data model and data access layer of DATEX II. In fact, a proposal for combining OTS and DATEX II within the meta data platform project is currently being prepared. OTS will provide a communication platform for realizing high level communication concepts like DATEX II in heterogeneous manufacturer mixed system networks.

OPEN TRAFFIC SYSTEMS

The OTS (Open Traffic Systems) concept can trace its origins to the 2002 founded Open Traffic Systems City Association (OCA, see [6]). This is an association of purely public bodies and operators that currently has 35 members from Germany, Austria and Switzerland. As clients for the creation and/or expansion of traffic management systems, the pertinent public authorities are almost always confronted with the necessity of having to mix manufacturers. This normally entails problems on both the client and contractor sides since the manufacturers involved are in competition with each other and act accordingly.

The necessity of having to mix manufacturers on the one hand results in public tendering laws stipulating a competition for the procurement of new system components, even if these have to be integrated into existing system landscapes. On the other hand, systems from different public authorities (e.g. municipality, county, private) have to be functionally combined with – historically developed - systems from different manufacturers since this is the only way to provide the desired traffic management services.

The OTS concept is aimed at precisely this initial situation and the uncomplicated realisation of a mixture of manufacturers. Firstly, it offers a process model that has been especially aligned to the specifications, public tendering process, test and approval of mixed-manufacturer systems, and secondly it contains a reference model for the design of an open, distributed system architecture for a traffic management system. One integral element is the OTS-interface concept with corresponding OTS-interface specification for cooperation between subsystems involved in a mixed-manufacturer overall system.

SYSTEM ARCHITECTURE

The OTS protocol specification consists of concepts from communication protocols up to data models. The application oriented data model covers control data, status data and measured or computed data within the field of traffic management. Extensions are always possible and can be processed compatibly. A special branch is concerned with configuration information, especially for traffic signalling and control systems. The data model specifies an addressing scheme for the identification of objects to which the transmitted data refer. Extensions of the data model and its addressing scheme are planned in the context of integration with Datex II.

Adequate application centric communication patterns and their protocols for establishing communication, retrieving configuration data, subscription to traffic management data and its distribution as well as delivery of control commands are defined. These application centric protocols are built upon the functionality offered by the OTS-Transport and OTS-Session layer, described in the further sections. It is envisaged that OTS-Transport and OTS-Session build a framework that allows for rapid development and consistent deployment of future communication patterns and their application-centric protocols.

Figure 1 shows an exemplary scenario for the application of OTS to traffic management systems. It shows a traffic information application (middle node) that receives traffic measurements from road sensors via the server of the traffic signalling and control system and is able to control information displays based on this input data. The data elements that are used here are traffic measurements and information messages.

Each node within this scenario takes a specific role with respect to the applied communication pattern. The traffic information server behaves as a data subscriber for traffic measurements, subscriptions are managed by the traffic signalling server. The sensors behave as data publishers that forward data to the server of the traffic signalling and control system which distributes this data to the respective subscribers. Together, the three nodes use the publish-subscribe communication pattern [1]. The command pattern is formed by three nodes that act as commander, command forwarder and command consumer. Respective protocols are provided by OTS entities within the application-centric communication pattern layer.

All OTS applications are based on OTS-Transport and OTS-Session. These protocol building blocks are the main focus of this paper and will be described in the next sections. The key functionality these blocks provide with respect to system integration is the provision of abstract transport channels (based on different host protocols, e.g. TCP/IP and SOAP) and the flexible operation of these within a single session.

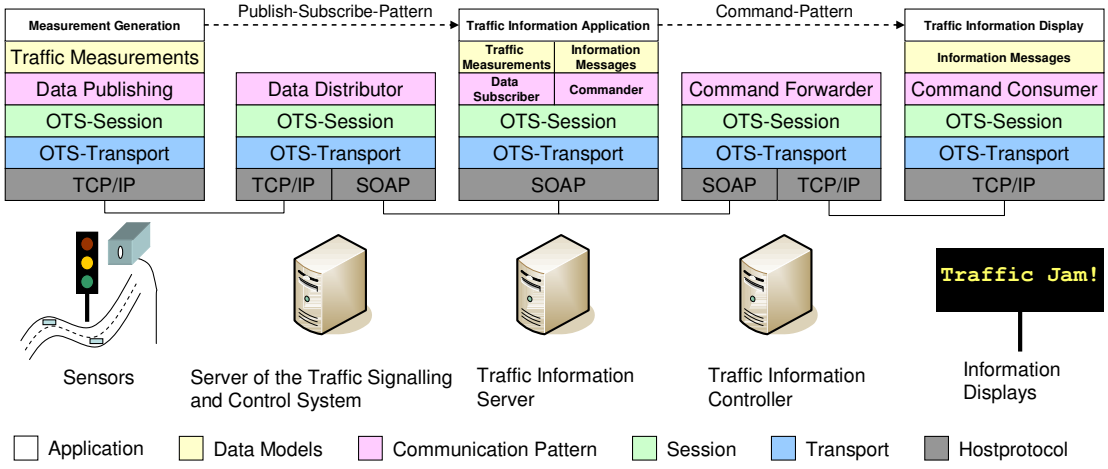

Figure 1: Exemplary Scenario

OTS PROTOCOL SUITE

OVERVIEW

Figure 2 shows the OTS protocol architecture. The central paradigms of this architecture are

1. Build small hermetic protocol layers with clearly defined functionality. Build protocol blocks.
2. Make these blocks generic. This promotes reusability and allows building a protocol toolbox.
3. Build whole protocol stacks by selecting and stacking appropriate blocks from the toolbox. Since blocks are small you only pick functionality that you really need for your application.
4. Don't extend blocks by modifying them. Extend by defining new extension blocks following item 1.

The key protocol block of OTS described within this paper is the OTS-Transport layer. The OTS-Transport layer manages access to different host protocols while offering one unified Service Access Point (SAP) to upper layers. Upper layers make use of OTS-Transport Channels which offer unified asynchronous and synchronous messaging services no matter what host protocol is used. For the time being OTS protocols are based on either the Simple Object Access Protocol (SOAP) or directly on TCP/IP. Only this allows for extending the application level protocol blocks independently of the supported host protocols.

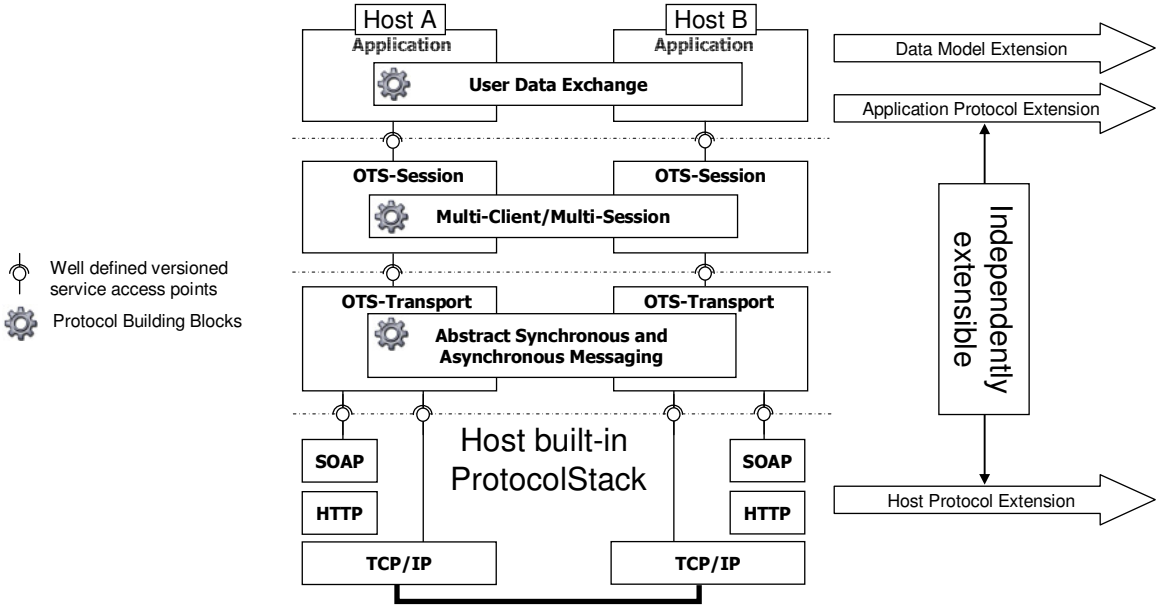

Figure 2: OTS Approach - Toolbox of reusable Protocol Functions

OTS-Session is based on OTS-Transport. It realises a session service that allows applications to be multi-client and multi-session capable. This capability is so important to distributed systems that an independent protocol function was established. It allows for dynamic exchange of transport channels during the lifetime of a session. This is useful to trade implementation complexity against bandwidth requirements. If an application level protocol has high bandwidth requirements it can choose to send its bulk data using an efficient binary encoding on a TCP/IP based OTS-Transport channel. Low bandwidth applications will make

use of OTS-Transport channels on top of SOAP to leave data encoding and stub generation to the SOAP framework.

Application level protocols for traffic management systems will primarily need to realise Publish-Subscribe [1] measurement distribution services and support for Request/Commit transactions to securely configure remote devices.

The next section will give a detailed view on how the abstraction of different host protocols is realised within OTS-Transport. It will be shown how establishment and release of OTS-Transport channels is implemented both with TCP/IP (built-in connections) and with SOAP (no connection concept). The OTS-Session layer will not be described here in further detail due to the limited space. Yet, it is important to note that a prototype for the OTS Protocol Suite has been developed at ComNets which provides all the aforementioned functionality. Continuous automated protocol testing has been used to guarantee the correctness of this prototype. The prototype and testing framework have been developed in Python and with support of the Twisted [7] networking engine and runs both on Windows and Linux.

OTS TRANSPORT LAYER

The key to manufacturer diversity and independence of technology is a common communication basis. OTS Transport is designed for this purpose. It is a minimalist abstraction layer for different host protocols, yet powerful enough to support a plethora of higher level communication protocols. It offers an asynchronous, symmetric message transfer service that supports a connection concept. Symmetric means there is no distinction such as client or server once a transport channel has been established. Messages may be sent by each communication partner at any time. It will be shown here that it is possible to realise this service both on TCP/IP (connection-oriented, asynchronous, symmetric, and stream-based) and on SOAP (connection-less, synchronous, asymmetric, and message-based). Claiming that integration of further host protocols is not a problem, thereby seems plausible.

Figure 3 shows a sequence diagram of a typical communication sequence. OTS-Transport has five generic functions that are used during such a communication sequence:

1. Passive Open: OTS-Transport users that offer a service to other OTS-Transport users must indicate to the OTS-Transport layer that they are willing to accept incoming channel requests.
2. Active Open: OTS-Transport users that want to contact the service offered by another OTS-Transport user can initiate the channel establishment through this function. After this function is executed a transport channel is established and both client and server can refer to this channel by using the negotiated *TransportID*.
3. Asynchronous Message Transfer: After a channel has been established both the client and server may send messages to each other at any time in any direction. The transport channel that is to be used for message transfer is identified by the *TransportID*. Synchronous message transfer can be realised simply by blocking the application locally on TData calls. This does not affect the protocol and is neglected herein.
4. Disconnect: Both communication partners may close an OTS-Transport channel at any time. The *TransportID* becomes invalid.
5. Passive Close: If a service is no longer offered the server may stop to accept incoming OTS-Transport channels.

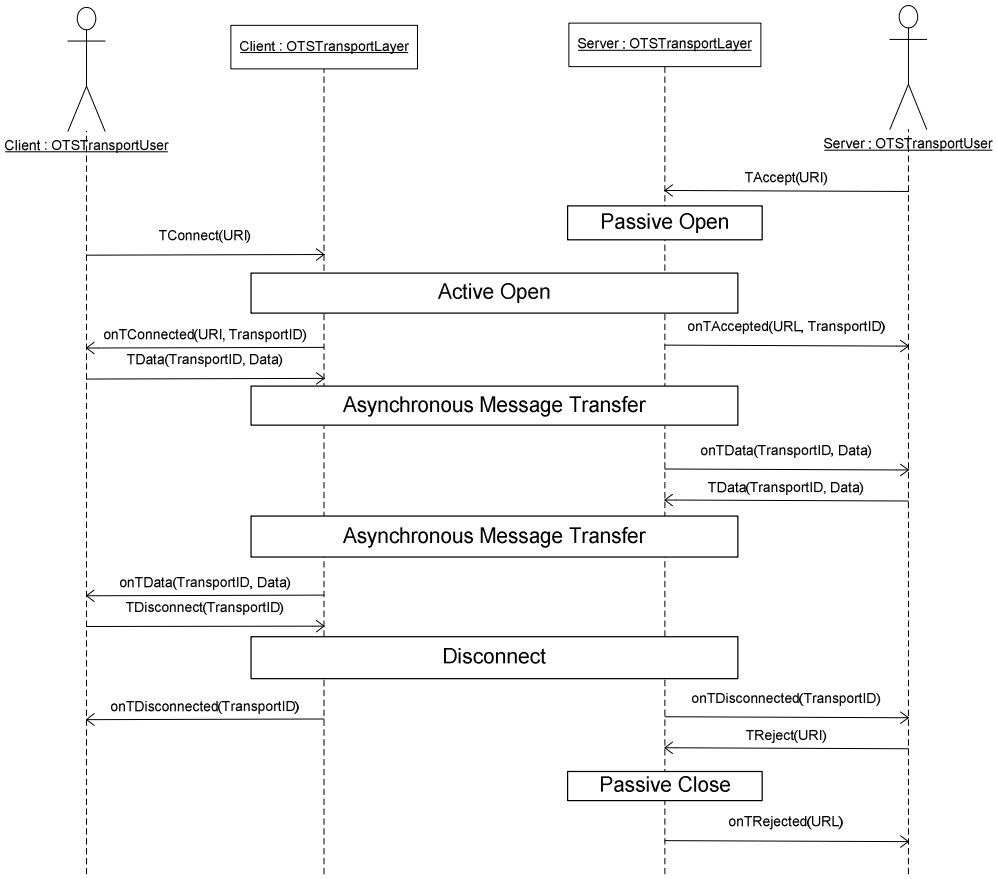

Figure 3: OTS-Transport Protocol Functions

Realisation

OTS-Transport users are addressed by using the well known Uniform Resource Identifier (URI) [8]. Table 1 shows the general structure. URI addressing is only used during the Passive Open and Active Open phases.

Schema		Username		Password		Hostname		Port	Path		Query		Fragment
otssoap:	//	daniel	:	mypassword	@	comnets.de	:	80	/dbn	?	a=b	#	example

Table 1: Structure of the Uniform Resource Identifier (URI)

OTS-Transport determines the channel type by interpretation of the URI's Schema field which can be either 'otssoap' or 'otstcp'. Username and password are used by OTS-Session for authentication. Hostname, port and path are used by OTS-Transport as will be described in the following. Query and fragment fields are not used by now.

Once a transport channel has been established communication partners can refer to the channel by using the *TransportID* which is negotiated between client and server during the *Active Open* phase. The *TransportID* is composed by a *ClientPart* and a *ServerPart* (e.g., integers) to ensure that it is unique both in client and server. This approach is similar to the source and destination port concept of TCP and UDP.

Protocol Binding for TCP/IP

The OTS-Transport service can be realised very easily with a connection-based host protocol like TCP/IP. The *Passive Open* function is simply realised by binding to the port and IP address associated with the hostname and listening to the created socket. OTS-Transport may use the blocking function `accept()` of TCP/IP to wait for incoming connections. The *Active Open* function uses `connect()` to establish a TCP connection to the peer host. After the underlying TCP connection is established negotiation of the *TransportID* is started. Suppose we use a XML encoding on the TCP connection the client would send "`<channelSetup>`*`ClientPart`*`</channelSetup>`" to the server. It will respond with "`<channelSetupResponse>`*`TransportID`*`</channelSetupResponse>`". Please note that *ClientPart* will contain a unique number chosen by the client while the server responses with the complete *TransportID,* i.e. the *ClientPart* concatenated with the *ServerPart* chosen by the server. Message transfer is simply done by writing data to the associated sockets. Messages are separated by encapsulating the payload within message tags. The *Passive Close* and *Disconnect* functions both use TCP/IP's `close()` method. Associated *TransportIDs* will be marked invalid when a TCP connection closes.

Protocol Binding for SOAP

The SOAP protocol is a connectionless protocol. Therefore OTS-Transport needs to realise connection management when binding to the SOAP protocol. SOAP implements the remote procedure call (RPC) communication pattern. A SOAP *Service* is a collection of several procedures. A *Service* is made available by deploying it at a given URI, i.e. a SOAP server will accept incoming TCP connection on a given port. Multiple Services may be deployed on the same TCP port. SOAP servers dispatch calls to different service by the path element of the URI. SOAP request may either be made by using HTTP's GET or POST method.

To realise connection management the service dispatching capability by URI path element of SOAP servers is exploited. The protocol functions *Active Open* and *Disconnect* build the connection management service of OTS-Transport which is always deployed at a fixed path of the URI (e.g. otssoap://ots.comnets.de:5678/OTS/Transport/ConnectionManagement). OTS-Transport will deploy this service within the *Passive Open* function.

Clients that want to connect to a SOAP server do this by activating the *Active Open* function. OTS-Transport calls the connect method of the connection management service available at the given host and port combination. It generates the *ClientPart* of the *TransportID* and passes it along. The server receiving that call will construct the *TransportID* by generating and appending the S*erverPart*. The complete *TransportID* is then returned to the client. Additionally the server deploys a new messaging service solely responsible for this transport channel at an URI that includes this *TransportID.* (e.g. otssoap://ots.comnets.de:5678 /OTS/Transport/Channels/*TransportID*). When the client receives the response of the connect call it knows that a messaging service has been deployed by the server at that URI.

Subsequent activations of the *Asynchronous Message Transfer* function within the client function are then simply mapped on the respective SOAP method of the transport channel's messaging service.

The *Asynchronous Message Transfer* function within the server can be realised in two ways:

1. Pull-style realisation: The server offers methods that allow the client to poll for new messages. The server must buffer messages until the client actively pulls them. Polling is solely handled within OTS-Transport. Its behaviour towards upper layers remains

asynchronous, though users might experience increased message delays. This approach allows for slim clients that do not need to deploy a SOAP server of their own but it is not truly asynchronous.

2. Push-style realisation: Clients deploy a messaging service of their own after completing the *Active Open* function. The server can immediately deliver messages by calling the respective method within the client. This approach is symmetrical and implements truly asynchronous messaging, circumvents message buffering within servers, but results in higher client complexity.

The *Disconnect* and *Passive Close* functions are simply realised by removing the respective messaging service or connection management service.

CONCLUSIONS & OUTLOOK

The work presented within this paper is a result of the German Dmotion project. The presented extensible middleware solves the following key problems. First, it gives clear extension points to allow for system composition with equipment from different vendors. Second, it allows for independent extensions both on application level and on the host transport protocol. And third, it encourages reusability by building up a protocol toolbox of common communication patterns. By decoupling extensions in host protocols and in application protocols, it enables evolving systems.

ACKNOWLEDGEMENT

This work has been funded by the German BMWi and was carried out as part of the German Dmotion project [2][3].

REFERENCES

[1] P.T. Eugster et al., “The Many Faces of Publish–Subscribe,” ACM Computing Surveys, vol. 35, no. 2, 2003, pp. 114–131.
[2] Düsseldorf in Motion (Dmotion), “Integrated Strategy Management for Enhanced Mobility”, http://www.dmotion.info/download/dmotion_flyer_englisch.pdf
[3] "Forschungsinitiative VM 2010 - Dmotion Gesamtvorhabensbeschreibung"; Düsseldorf, Dezember 2004; Landeshauptstadt Düsseldorf, Amt für Verkehrsmanagement et al.
[4] Datex II project web page. http://www.datex2.eu
[5] Nationaal Datawarehouse web page, http://www.nationaaldatawarehouse.nl/
[6] Open Traffic Systems City Association (OCA) webpage, http://www.oca-ev.org
[7] Twisted Matrix Labs, “A framework for networked applications”, http://twistedmatrix.com
[8] IETF, RFC 3986 “Uniform Resource Identifier (URI) : Generic Syntax”, January 2005