Development of a Communication Framework for optimized and guaranteed information exchange between train and wayside

Thesis report

Andrzej Koszela

10/27/2011
ABSTRACT

Modern train systems generate vast amounts of data. Analysis of this data may sometimes not be performed on the running train, and must be transmitted to other systems operated by the train management. The ability to transfer information to and from trains is often dependent on imperfect communication channels, such as mobile phone networks. Accordingly, there is an immediate need for an effective, robust and secure communication framework to abstract away these considerations from the information producers and consumers. This project attempts to define a framework for reliable data communication using unreliable communication channels, interacting with existing train systems. The result consists of a prototype design and a .Net implementation of such a framework, as well as a generic and extensible type library corresponding to the types of data generated by the Flytoget train system. Furthermore, several performance tests were undertaken; it is shown that while the use of such a framework may be detrimental on bandwidth usage, this negative impact may be lessened by the use of compression and message batching. The results presented in this thesis will be of general use on modern train systems, while the prototype was designed specifically for Flytoget.
# TABLE OF CONTENTS

ABSTRACT ........................................................................................................................................... 2

TABLE OF CONTENTS .......................................................................................................................... 3

1. BACKGROUND .................................................................................................................................. 5
   1.1 TRAIN DATA ................................................................................................................................. 5
       1.1.1 Process data ......................................................................................................................... 5
       1.1.2 Message data ....................................................................................................................... 5
   1.2 CURRENT SOLUTIONS ............................................................................................................... 6

2. CONVENTIONS ............................................................................................................................... 7

3. RELATED WORK .............................................................................................................................. 8

4. PROBLEM FORMULATION ............................................................................................................. 8
   4.1 DESIGN REQUIREMENTS .......................................................................................................... 8
       4.1.1 Req A .................................................................................................................................. 9
       4.1.2 Req B .................................................................................................................................. 9
       4.1.3 Req C .................................................................................................................................. 9
       4.1.4 Req D .................................................................................................................................. 9
       4.1.5 Req E .................................................................................................................................. 9
       4.1.6 Req F .................................................................................................................................. 9
   4.2 IMPLEMENTATION REQUIREMENTS ......................................................................................... 9

5. PROBLEM ANALYSIS ...................................................................................................................... 10

6. MODEL & METHOD ......................................................................................................................... 11
   6.1 DATA SOURCES ....................................................................................................................... 11
       6.1.1 ATC ..................................................................................................................................... 12
       6.1.2 GPS ..................................................................................................................................... 12
       6.1.3 TMS system information ..................................................................................................... 12
       6.1.4 Log files ................................................................................................................................ 13
       6.1.5 TSSP ..................................................................................................................................... 13

7. SOLUTION & IMPLEMENTATION .................................................................................................... 13
   7.1 INTERFACE .................................................................................................................................. 13
   7.2 TECHNOLOGIES USED ............................................................................................................ 14
   7.3 FUNCTIONALITY ....................................................................................................................... 15
       7.3.1 External ............................................................................................................................... 15
       7.3.2 Internal ............................................................................................................................... 16
   7.4 DATA TYPES & DATA ENCAPSULATION .................................................................................. 17
       7.4.1 Messages ............................................................................................................................. 17
       7.4.2 Notifications ........................................................................................................................ 19
1. BACKGROUND

A modern train system contains a large number of cooperating subsystems. Train control and monitoring involves these systems communicating with each other, and often communicating with a central train computer as well. The subsystems, in addition to generating normal operation data, often generate varying amounts of diagnostic information, of great use when unexpected situations may occur on the train. Analysis of this data is often more difficult if it is performed on the train, both due to the limited equipment onboard, as well as the additional costs for the train owner. In general, every unscheduled train outage is expensive, and even more so if sending technicians and support personnel to remote trains is required. If possible – given the current situation – the ability to analyze and diagnose any events remotely is then often an advantage.

This, however, requires the capability to transport large amounts of data, of various types, between train and wayside. This data may be generated in real time – such as monitored traffic between train subsystems – and should, in this case, be sent from the train as soon as possible; but data with less strict urgency requirements – such as statistics or logs where the data is collected by train side systems before being sent, perhaps on a daily basis – might also be of interest.

Since different train systems vary greatly with regards to complexity, functionality and purpose, the data they generate is also bound to vary from system to system. Since the majority of onboard systems on a modern train use digital electronics, the generated information may well be collected, processed if necessary and transmitted by computer systems, but one must still consider the mentioned variations regarding the amounts and structure of the data of interest.

This report proposes a software system that attempts to implement these capabilities.

1.1 TRAIN DATA

In modern trains, most data traffic is generally categorized as being of one of two types. These data paradigms differ in form as well as in function, and are briefly described here.

1.1.1 PROCESS DATA

Characterized by fixed structure and periodicity; information is sent cyclically, and may contain e.g. a representation of the internal state of a system. The periodicity is often 1-5 Hz. This type of data is often sent regularly between a main system and any relevant subsystems.

1.1.2 MESSAGE DATA

Describes data sent as a consequence of some internal or external event. Such an event may be new data being received from an outside source, the state of a system changing, or a previous request for data. Message data is structured, but may often contain fields of varying length, or
varying numbers of fields of a given structure. This is due to the information content of a message necessarily depending on the specific event that caused the message to be generated.

The desire to analyze generated data on the wayside, combined with the variance in the data, leads to a need of a general framework for efficient train-wayside data transport. The communication must be data neutral, and the framework must be extensible with regards to new types of data, if and when such a demand arises. An issue that must be considered is the general lack of stable and safe communication channels between train and wayside; historically, different analogue and digital radio systems have been utilized for this purpose, only offering very limited bandwidth. Lately, newer technologies such as 4G, UMTS, GPRS, etc. have made higher-bandwidth and more reliable communication possible, but the availability of these channels is far from universal. In the countryside – where train traffic is not uncommon – cellular network coverage is rarely perfect, and when the train encounters various obstacles (such as tunnels) the cellular network connection often goes down completely, which is a quite commonly occurring problem.

Due to complications such as these, the communication framework must support some kind of error control as well as reliable data transport.

1.2 CURRENT SOLUTIONS

Currently, the amount of log data transferred off trains is rather limited; in addition, the transfer is commonly initiated manually – a common scenario is that while the logging of communications between two train subsystems is automated, the transfer of the stored information is done manually if and when a problem is discovered by other means. The transfer itself is commonly done using commercially available solutions, such as VNC or other remote desktop solutions, which is often a rough solution and leads to several limitations. Commonly, if only a subset of the gathered data is of interest in the current situation, any filtering has to be performed on the wayside, after the data has been transferred off-train. Stored data can then only be used to diagnose errors that have already occurred, and cannot easily be used in a preventive manner. These factors contribute to a situation where most of the traffic and diagnostic information generated by the train systems is never examined; neither by automatic nor by manual means. This is because the analysis possibilities on the train are often severely limited, and the transfer of information to the wayside (where the investigative and analysis resources are much more powerful) is relatively complex.

The communication framework whose prototype is described is meant to combat and overcome these issues and allow for easy access to information generated by train systems. The goal is that the framework shall be integrated into and constitute an integral part of the general mobile train monitoring and diagnostics concept TMS (see 15, p. 41).
### 2. CONVENTIONS

This report uses the following terms, abbreviations and conventions:

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>bit</em></td>
<td>The atomic binary unit.</td>
</tr>
<tr>
<td><em>b</em></td>
<td>See <em>bit</em>.</td>
</tr>
<tr>
<td><em>byte</em></td>
<td>8-bit <em>word</em>.</td>
</tr>
<tr>
<td><em>B</em></td>
<td>See <em>byte</em>.</td>
</tr>
<tr>
<td><em>bandwidth</em></td>
<td>Network data rate. In this text, this term is used in the computing sense, e.g. as synonymous to <em>maximum throughput</em>.</td>
</tr>
<tr>
<td><em>word</em></td>
<td>Consists of a number of <em>bits</em>. Unless specified otherwise, a 4-byte (32-bit) word size is assumed.</td>
</tr>
<tr>
<td><em>Ki</em>, <em>Mi</em></td>
<td>IEC prefixes used for powers of two; <em>Ki</em> is $2^{10}$, <em>Mi</em> is $2^{20}$ etc. See [4].</td>
</tr>
<tr>
<td><em>ATC</em></td>
<td>Automatic Train Control, a train safety system including signaling and warning.</td>
</tr>
<tr>
<td><em>GSM</em></td>
<td>Global System for Mobile Communications; European and international standard for second-generation digital cellular networks.</td>
</tr>
<tr>
<td><em>GRPS</em></td>
<td>General Packet Radio Service; extension to GSM handling packet data traffic over GSM networks.</td>
</tr>
<tr>
<td><em>3G</em></td>
<td>A collection of third-generation digital cellular network standards.</td>
</tr>
<tr>
<td><em>TMS</em></td>
<td>TCMS Mobile Solution, an extension of TCMS that allows for remote train monitoring and diagnostics.</td>
</tr>
<tr>
<td><em>VNC</em></td>
<td>Virtual Network Computing, a commonly used type of desktop sharing software which enables viewing and interacting with the contents of a remote computer desktop.</td>
</tr>
<tr>
<td><em>TCMS</em></td>
<td>Train Control and Management System; a complete computerized train control solution.</td>
</tr>
<tr>
<td><em>XML</em></td>
<td>Extensible Markup Language, a standard of data encoding commonly used in internet scenarios.</td>
</tr>
<tr>
<td><em>SOAP</em></td>
<td>A collection of protocols and standards used to facilitate data exchange. Relies on XML and other internet-related protocols and commonly used in web service scenarios.</td>
</tr>
<tr>
<td><em>Framework</em></td>
<td>A collection of APIs, tools, and data types used by other applications to perform certain tasks.</td>
</tr>
<tr>
<td><em>Func()</em></td>
<td>Refers to a zero-or-more-argument function named <em>Func</em>. The parentheses do not automatically imply that <em>Func</em> is a zero-argument function. Function signatures are</td>
</tr>
</tbody>
</table>
3. RELATED WORK

Several solutions are currently used to transport diagnostic and other data between the train and the wayside.

Gray [1] has examined optimization of so called web services, and bandwidth usage when transporting general and XML data over HTTP. The publication compares different Java implementations, but since Java and .Net are relatively similar, and both environments greatly simplify data serialization and remote procedure calls while making much of the underlying work invisible to the library user, the presented results are still of great interest.

Davis, Parashar [2] note, in a comparative publication, that SOAP implementations in Java, Perl and .Net are similar when it comes to performance. This also implies that the main bottleneck is the SOAP-HTTP layer.

No major published performance study of MSMQ, a technology on which the prototype implementation of the framework relies, seems to exist. An often referred to source is however [3], a publication on the Microsoft Developer Network, wherein several performance considerations in an older version of MSMQ are presented. The MSMQ abstraction in WCF is not mentioned, but since the underlying technology is the same, similar performance considerations apply in this report.

See also 1.2 p. 6 for a description of the technologies currently used to implement train-to-wayside communication.

A thorough comparison between the effectiveness of different compression algorithms on XML-formatted data is found in [5]. While not specific to web service implementations, this paper is highly relevant to the presented results regarding compression and performance, and contains a theoretical discussion beyond the scope of this report.

4. PROBLEM FORMULATION

As an attempt at a solution to the above issues, a general communication framework will be designed, and a working prototype will be constructed.

4.1 DESIGN REQUIREMENTS

The initial design requirements (see 1 p. 5 for a discussion) are as follows.
4.1.1 **REQ A**

Data neutrality; the framework should, so far as is feasible, handle any different types of data.

4.1.2 **REQ B**

Security; encryption (transport and/or message layer) and/or authentication should be handled internally by the framework and be transparent to the user applications, on the server side as well as on the client side.

4.1.3 **REQ C**

Priority/QoS; the available bandwidth should be utilized efficiently by the framework.

4.1.4 **REQ D**

Reliability; the framework should handle at times imperfect connection conditions; information sent over the framework should be delivered when the connection improves. This must be transparent to the user.

4.1.5 **REQ E**

Addressing; the framework should handle a many-to-one scenario, where several clients communicate with a single server. The server must implement a scheme where the clients are differentiated and uniquely addressable.

4.1.6 **REQ F**

Optimization; due to limited bandwidth and unreliable connections, the framework should so far as is feasible utilize compression or other means of minimizing the amount of data sent over the wire. This must happen transparently to the user.

4.2 **IMPLEMENTATION REQUIREMENTS**

In addition to the above, the framework must be built in Microsoft .Net to facilitate interoperation with TMS, since much of TMS is built using this environment.

The goal is to develop and test, on actual running Flytoget (see 14, p. 39) train(s) in Norway, a working prototype that satisfies the above requirements.

For the Flytoget project, all data transport will be performed over a network where the existing infrastructure guarantees encryption and authentication. This is handled in the lower layers of the network. For this reason, the prototype is not required to satisfy Req. B, though some level of support for a future implementation of encryption should be built into the prototype.
The prototype system will consist of one or more client systems placed on running Flytoget trains, which will collect data from various subsystems (see 6.1 p. 11) on the trains, and communicate with a central server. The sets of train subsystems will be identical across all trains, but the generated data will of course vary.

In addition to the design and implementation of the prototype, a number of tests will be specified and performed to measure the performance and suitability of the framework for future use.

5. PROBLEM ANALYSIS

A solution to the above presented problem entails solving a number of sub-problems. First of all, a suitable design had to be developed, taking into account the requirements described in 4.1 above. In addition, several design decision not explicitly specified in the formal requirements had to be taken, to ensure the resulting communication framework is suitable for general usage, and easily extensible. Furthermore, this report was written.

After the design phase, the implementation of the prototype of the framework began. Some design choices were reevaluated during the development phase, leading to an iterative development workflow. This is often a necessity due to new knowledge and experiences, as well as new problems, encountered during the development phase. Several tests were performed to determine the general functionality, suitability and performance of the prototype.

Thus, the design, development, testing, and reporting phases often overlapped.

A general client-server design was chosen for the framework. This is a natural choice, due to the problem being inherently two-sided, and most of the data traffic the framework is expected to handle is one-way (from the trains to the central server). In a real-world scenario, one client system would be deployed per train, and one central server system would handle and communicate with these clients. The client systems use existing hardware – the TMS systems already present on the trains – while the server system is installed on suitable wayside hardware. These then act as endpoints for the data communication taking place in the framework.

Proven, existing technologies were used as far as is feasible. This was done with the hope of cutting down on development time, and ensuring a higher final product quality. Specifically, since the framework is required to be implemented in Microsoft .Net, WCF – Windows Communication Foundation – is used for the actual data transfer between the endpoints. WCF provides a number of different underlying transport technologies – termed bindings – and is often used to develop enterprise web services, using SOAP, REST or other web technologies for communication. However, other bindings also exist, e.g. simple transport over TCP sockets. WCF also handles object serialization as well as service discovery and remote procedure call.
6. MODEL & METHOD

Following the problem analysis, the framework has been implemented as a .Net class library, a client application, and a server application, both using this library. On the client side, the application collects data from the specified subsystems, and sends this over the framework. Eventually the data will reach the server side, where the server application reacts to new data in a suitable manner. In the prototype stage these applications are functional, albeit rudimentary, since the overall goal is for third party applications and project specific applications to use the framework for their special needs. The prototype whose design and development are detailed in this report is mainly a “proof of concept” implementation, and shows whether the suggested solution to the actual communication problems is reasonable.

The framework will be part of the TMS system; eventually most of the communication between TMS and wayside will be handled using the framework (exceptions to this being purely administrative traffic using Remote Desktop). In this report TMS mostly refers to the client side system, meaning the industrial PC unit (see 15, p. 41) physically installed on the train. Several other systems that are today part of TMS use the .Net environment, making this a natural choice for the communication framework as well.

The prototype has been designed to as far as is feasible act in an event driven manner. This means that the information sources on the client side call methods in the framework if and when new data is available, and a similar philosophy is applied on the server side. To make this possible some specific logic is required in the client and server side applications. In practice, this is implemented as callable methods on the client side, and exposed events on the server side. Data producers call the client side methods at their convenience, supplying new data, while data consumers subscribe to the server side events, which are then fired by the framework when the data has been transported over the framework. There are also corresponding server side methods and client side events to facilitate server-to-client data transfer.

The alternative to this event driven philosophy was using polling, meaning a system where the user applications as well as parts of the framework periodically query internal parts of the framework whether new data is available. This would imply additional traffic over the wire, since the server side polling requests would have to be forwarded to the information gathering component on the client side, leading to performance losses. The user applications, as well as some mechanisms in the framework, might still use polling internally, but when feasible, this is avoided.

6.1 DATA SOURCES

In the prototype design, requirement A (4.1.1, p. 9) has been specified as the possibility of sending the following data types through the framework, as these are the data producers currently available to TMS. The various data sources, and some characteristics of the data they produce, are described here.
6.1.1 ATC

ATC data is generated by a receiver mounted on the underside of the train. Whenever the train passes over a beacon – a special device mounted on the tracks which communicates with the receiver using radio – the internal state of the receiver is updated. The receiver continuously generates and transmits, with a frequency of 2 Hz, data packets to an evaluation unit. From this unit signals are then transmitted to a panel in the train cockpit (which displays information to the train driver), a recording unit, and if necessary to other train systems. On Flytoget, the implementation of this system is from the 1980s, and is older than most other electronic equipment; because of this, the signals sent between the units are encoded in a somewhat unfamiliar way. Logical values are represented by electrical current values (12–30 mA for a logical 1, 0 mA for a logical 0) and not electrical voltage values, which is the norm for modern electronic equipment. A specially constructed component is placed on an output port of the evaluation unit – a safety-certified part of the ATC system – in order to translate these signals into regular serial communication, though the translation process results in a symbol rate of 781,25 baud. This serial signal is then sent to the TMS system, where it may either be analyzed, sent to the wayside by the communication framework, or both.

The ATC data packets have a size of 32 bytes; any other packet size indicates either an error, or the system not being in operational mode, and the packets are then to be discarded. The communication framework will ignore these packets. The data content of the packets is structured, with fixed-length fields, and mostly numerical data (except for some oddities, such as two fields containing the raw output signals to a three-digit seven segment display each). Some of the fields are coded with a redundant 8-4 Hamming code; it is assumed that any communication faults between the beacon and the analysis unit are corrected by the latter, before the data is forwarded to TMS. For this reason, the redundant error-correction data is ignored, and no error checking is performed.

ATC falls into the scope of process data as described in 1.1.1 p.5.

6.1.2 GPS

The TMS system contains a GPS receiver, which generates updated positioning data. These are directly read by TMS and forwarded over the communication framework. The GPS accuracy is relatively good, but suffers from a lack of satellite coverage in tunnels, leading to intermittent accuracy loss. In the future this issue will be alleviated using a supplementary dead reckoning navigation system in TMS. GPS location information is considered as process data.

6.1.3 TMS SYSTEM INFORMATION

On occasion, TMS itself will act as an information source, and generate some diagnostic data. TMS contains a battery backup system for operating power, and the status of this system is queryable through the battery software. This information (including the current battery charge level, etc.)
will be sent, as soon as any value changes, to the wayside through the communication framework. This system information is considered as message data.

### 6.1.4 LOG FILES

The functionality to send files containing arbitrary information from TMS to wayside shall be included in the framework; in the prototype stage, this will mostly concern logged serial communication. These files vary greatly in size and contents. Today, much of the diagnostic information is stored in files on the train before being sent to the wayside; thus, supporting this method of operation is of great importance.

### 6.1.5 TSSP

TSSP is a general protocol for receiving train data from TCMS. Size and contents of the information received may vary, and may be transmitted by TCMS either as process or message data. A TSSP client application runs on the TMS system, and subscribes to various signals offered by TCMS. In the prototype stage, the set of signals of interest, as well as the subscription method (cyclic, on change, etc.), are known in advance.

Whenever the TSSP server application sends new data to TMS, the data is forwarded by the communication framework to any server side subscribers. The TSSP server is, depending on the type of the subscription selected by the TSSP client software, capable of producing both message as well as process data.

### 7. SOLUTION & IMPLEMENTATION

#### 7.1 INTERFACE

As mentioned earlier, the core of the communication framework is implemented as a .Net class library. To simplify usage the portion of the framework that is exposed to external users is minimized, it is hoped that this will reduce the confusion sometimes associated with using task-specific software frameworks. Since the framework uses a client-server model, the main visible portion of the framework is divided into two exposed interfaces; a client side interface and a server side interface. The idea is that the user, when constructing components cooperating with the framework, should be able to separate the client and server side sections of the component naturally.

To increase the level of modularity offered by the framework, each of the interfaces is further separated into two; a local and a remote interface. These are then referred to as “backends” – there is thus a client side local backend, a client side remote backend, a server side local backend, and a client side remote backend. The client/server terms describe which side of the framework the interface is used from, and the local/remote terms describe on which side of the communication channel the methods are, this is however handled internally by the framework, and all
applications using the framework will use the local interfaces only. As an example, data producing applications running on the client wishing to communicate with the wayside will use the client side local interface, while data consuming applications running on the server will use the server side local interface.

The interfaces associated with the communication framework are thus:

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITMS_Serverside_Local_Backend</td>
<td>Used by server side applications.</td>
</tr>
<tr>
<td>ITMS_Serverside_Remote_Backend</td>
<td>Used internally by the framework for client-to-server communication.</td>
</tr>
<tr>
<td>ITMS_Clientside_Local_Backend</td>
<td>Used by client side applications.</td>
</tr>
<tr>
<td>ITMS_Clientside_Remote_Backend</td>
<td>Used internally by the framework for server-to-client communication.</td>
</tr>
</tbody>
</table>

Two .Net classes implement these interfaces and constitute the core of the communication framework; these are the client side backend implementation, and the server side backend implementation. Internally, these implementations use a communication stack based on WCF and MSMQ (see Figure 1).

The framework is modularly designed. As long as the interfaces are implemented, the specific backend implementations may be arbitrary, and may use completely different communication channels internally. This will be completely transparent to the client and server side applications. A possible scenario is an implementation of the framework on top of a communication infrastructure where no TCP/IP communication stack is available on some part of the path between client and server. The specific framework implementation would then use some existing protocol for communication between the backends instead of the current stack (WCF over MSMQ over TCP/IP), without the applications using the framework having to adapt – or even be aware of – the underlying communication protocol. This modularity, combined with other extension capabilities, should make the framework easily adaptable to new scenarios.

7.2 TECHNOLOGIES USED

To avoid reinventing the wheel, existing solutions to encountered problems have, as far as has been feasible, been used. The specific technologies used in the prototype are as follows.

- **MSMQ**, a message queue implementation on top of TCP/IP;
- **WCF**, a web service framework, used to simplify data serialization, service discovery, and remote procedure calls;
- **LZMA** and **DEFLATE**, commonly used data compression algorithms (see 7.6 p. 21);
- **XML**, a common format for storing structured data in text format;
- **SHA256**, a common hash algorithm, used for file contents verification after transfer (see 7.8.1 p. 24).

Figure 1 shows a schematic overview of the entire communication stack used in the prototype.
7.3 FUNCTIONALITY

The functionality of the framework can be divided into two parts; the external functionality, visible to external applications, and the internal functionality, meant only for use by different framework components.

7.3.1 EXTERNAL

Seen from the outside, the exposed portion of the framework is relatively small. The user creates an instance of a backend implementation class, and calls methods exposed by this instance. The available methods are the ones specified by the backend interface. Following the general split of the framework into client side and server side functionality, there are also two APIs, to be used by server side and client side applications.

The client side API consists of the following methods and members (as defined by ITMS_Clientside_Local_Backend). The parameters and method return types are omitted for clarity.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SendMessageToServer()</td>
<td>Forwards a message to the framework, for transport to the server.</td>
</tr>
<tr>
<td>BatchMessageToServer()</td>
<td>Forwards a message to the framework, for later transport to the server.</td>
</tr>
<tr>
<td>SendNotificationToServer()</td>
<td>Forwards a notification to the framework, for transport to the server.</td>
</tr>
<tr>
<td>SendFileToServer()</td>
<td>Begins transmitting a local file to the server system.</td>
</tr>
<tr>
<td>OnReceivedNotification</td>
<td>Event fired upon the receipt of a notification; see 7.5, p. 20.</td>
</tr>
<tr>
<td>Open()</td>
<td>Opens and initializes the backend, and starts monitoring the client-side queue(s).</td>
</tr>
<tr>
<td>Close()</td>
<td>Closes all open handles in the backend.</td>
</tr>
<tr>
<td>ClientID</td>
<td>Unique numerical identifier assigned to this client. This identifier is configurable through XML, but the client and server must agree on the identifiers for all clients.</td>
</tr>
<tr>
<td>ClientName</td>
<td>Arbitrary client name assigned to this client.</td>
</tr>
</tbody>
</table>

The server side API consists of the following methods (as defined by ITMS_Serverside_Local_Backend).
Development of a Communication Framework for optimized and guaranteed information exchange between train and wayside

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SendNotificationToClient()</td>
<td>Forwards a notification (see 7.4.2, p. 19) to the framework, for transport to a selected client.</td>
</tr>
<tr>
<td>BroadcastNotificationToAllClients()</td>
<td>Forwards a notification to the framework to be broadcast to all known clients.</td>
</tr>
<tr>
<td>OnRecievedMessage</td>
<td>Event fired upon the receipt of a message; see 7.5, p. 20.</td>
</tr>
<tr>
<td>OnRecievedNotification</td>
<td>Event fired upon the receipt of a notification; see 7.5, p. 20.</td>
</tr>
<tr>
<td>OnFileComplete</td>
<td>Event fired upon the receipt of a file; see 7.5, p. 20.</td>
</tr>
<tr>
<td>Open()</td>
<td>Opens and initializes the backend, and starts monitoring the server-side queue(s).</td>
</tr>
<tr>
<td>Close()</td>
<td>Closes all open handles in the backend.</td>
</tr>
</tbody>
</table>

As mentioned earlier, care has been taken to make this API as implementation neutral as possible, and the API should be agnostic regarding the underlying transport technologies used.

7.3.2 INTERNAL

This subchapter will describe the internal implementation of the prototype, since the external API described above is implementation neutral.

Internally, the workings of the framework are defined by the actual backend implementations, and may vary greatly depending on the scenario to which the implementations have been adapted. The prototype implementation is, as has been described, based on WCF and MSMQ, and as such communicates with the WCF API.

On the client side, the backend implementation life cycle is simple; the client backend is started, the communication channels are initialized, the client application uses the framework for communication, and the backend is stopped along with its internal communication channels. These channels consist of interfaces to MSMQ queues. Two public queues are used on the server backend; a reliable, recoverable queue named queue_keep to which message delivery attempts are repeated in case of failure, and a non-reliable queue named queue. While the former is used for higher-priority messages, the latter is used for messages where guaranteed delivery is not a requirement.

With the use of WCF, the clients are configured (see 7.9, p. 25) with the addresses of the exposed queues on the server. At compile time, a client side class is generated that implements the remote server side interface ITMS_Serverside_Remote_Backend. The client instantiates an object of this class, and when transporting data to the server, calls methods exposed by this object. Regarding server-to-client communication, a similar process is applied, with the exception of the server being configured with the public queue addresses of all known clients; each client exposes only one queue used for all inbound notification communication. The interfaces and data types involved serve as contracts, which the communication counterparts guarantee to fulfill.
Thus, a simplified view of the internal framework functionality is that of a cross coupling, connecting the outbound local method on one side of the communication to the inbound remote method on the other, with WCF taking care of the actual data transport.

The WCF binding used is the \texttt{netMsmqBinding}, provided by WCF out-of-the-box. This binding serves as a wrapper around MSMQ, allowing use of this technology using standard WCF semantics.

\section*{7.4 DATA TYPES & DATA ENCAPSULATION}

The information transferred through the communication framework must be uniformly encapsulated to allow for data-agnostic processing, as well as simplify the data handling logic for both client as well as server side applications. To this end, a decision was made to use a message-based communication paradigm when designing the framework. The \textit{message} contains metadata – containing a timestamp, size information, and any other necessary information – as well as the payload, which is the actual data being forwarded. Messages are only sent from a client to the server.

In addition, a different, special type of small messages is also be used. These notify the various parts of the framework of various state changes in other parts – such as a client shutting down, or a TMS system reaching a low battery condition – and are mostly not be of interest to third-party applications using the framework for communication. These special messages are called \textit{notifications}, since they do normally not carry any payload as defined above. Notifications may be sent both from a client to the server as well as from the server to a client.

\subsection*{7.4.1 MESSAGES}

A class hierarchy has been defined to encapsulate the various message types forwarded by the framework. It is designed to be easily extendable with new classes to encapsulate any new types of data, although if this is unfeasible in specific scenario, a generic data container class is also available. This will however leave the responsibility of data serialization and deserialization to the user.

The fundamental information type contained by the message classes is the byte array. This choice was made for two reasons; first, it is the most general data type, since practically any computerized system onboard a modern train uses protocols wherein a sequence of bytes is the basic \textit{form of communication}; and second, transmitting data in this raw format helps to avoid unnecessary logic on the client side of the framework. The client side application only has to provide the following functionality:

1. Receiving of data from data producers (or, alternatively, generation of data internally);
2. Creation of the proper object in the class hierarchy, depending on the data source;
3. Filling the payload of this object with the raw data received from the data producer; and
4. Forwarding of this object to the communication framework.
With this approach, no deep data analysis has to be performed on the client side; instead, the data is forwarded to the server side “as is”. All translation and data analysis is performed on the server side, which both lessens the computational load on the client side system (which often is far less capable than the server side systems) as well as often leads to bandwidth conservation (see 8, p. 25 for a discussion on this subject).

However, if such a demand arises, client side data analysis is still possible. This would however not be carried out by the communication framework, but rather by either the data generator or by the client side application. A possible application where this functionality would be useful is data pre-filtration, where the client application performs some kind of data analysis to determine whether the received data should be sent to the server or discarded, based on the results of the analysis according to some predetermined criteria. Heuristic systems could also be used.

The following table describes the different message container classes in the hierarchy.

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATCMessage</td>
<td>This message contains unprocessed ATC data (see 6.1.1, p. 12) as sent to TMS by the ATC receiver system. In addition to the data, it contains the common message metadata fields: a timestamp, and a source ID field, identifying the sending client.</td>
</tr>
<tr>
<td>BatchMessage</td>
<td>This message contains a serialized collection of messages batched together by a client system for sending. It should not be used directly.</td>
</tr>
<tr>
<td>CompressedBatchMessage</td>
<td>This message type contains a serialized collection of messages batched together by the sending system, in the form of a BatchMessage, possibly compressed using one of the available algorithms (see 7.6, p. 21). After any decompression, the receiving server system will split the message collection into its constituent parts and handle each message separately. The payload of this message is a serialized BatchMessage object. In addition, this message also contains a field describing the compression algorithm used, if any, as well as the common metadata fields.</td>
</tr>
</tbody>
</table>
| FileMessage           | This message contains the contents of a file sent by the client system to the server system. The payload contains the file contents; in addition, the message also includes the common metadata and a file identification field, used by the receiving system to uniquely identify file transmissions. This message type will only be used for small files, where there is no need to split the file into pieces. While file transfers support compression, the messages do
Development of a Communication Framework for optimized and guaranteed information exchange between train and wayside

<table>
<thead>
<tr>
<th>Message Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FilePartMessage</strong></td>
<td>This message contains part of a file. The file ID is provided so that the receiving system may concatenate the file when all parts have been received.</td>
</tr>
<tr>
<td><strong>MessageBase</strong></td>
<td>This is the base message type from which all message classes inherit. It provides an array of bytes for holding the payload; a timestamp field which the client fills when the message is sent; a size field, and a source ID field, uniquely identifying the sending client. These fields are also contained in all derived message types, due to type inheritance.</td>
</tr>
<tr>
<td><strong>TSSPSubscriptionDataMessage</strong></td>
<td>This message contains unprocessed TSSP data (see 6.1.5, p. 13) as received by the TSSP client software running on the TMS system. Due to the large amounts of data normally provided by TSSP, these messages will most often be batched (see 7.7, p. 22 for a description of the batching mechanism). In addition to the payload field and the common metadata fields, this message type also contains a PDSubscription field, which includes details of the TSSP subscription the payload is associated with.</td>
</tr>
</tbody>
</table>

The message classes also contain various convenience methods used by the backends to facilitate message processing – for instance, the `CompressedBatchMessage` class contains a `ToMessages()` method, which applies any required decompression algorithm, resulting in a `BatchMessage` object, whose payload is then deserialized resulting in the original array of messages. These methods are used internally by the framework and should not be accessed by external applications.

Complex message types, such as `FilePartMessage` or `CompressedBatchMessage`, are never communicated to external applications through the exposed events (see 7.5, p. 20); instead, the framework handles these message types internally, and only sends the contained messages through to the external applications. For instance, a single `CompressedBatchMessage` received by the server may results in the `MessageEvent` event being fired multiple times, once with each message contained in the batch.

### 7.4.2 NOTIFICATIONS

In addition to messages containing data payloads, the framework also uses a second type of messages, termed *notifications*. These are used to communicate various state changes or
important events across the different parts of the framework, such as a client system starting up or going down. To avoid unnecessary overhead, these are not part of the MessageBase hierarchy, as they do not contain payloads, but constitute a separate class hierarchy. They are generally only intended for internal use by the framework, but events (see 7.5, p. 20) concerning the receipt of notifications are exposed and fired. Together with TMS system information messages (see 6.1.3, p. 12), this makes it possible to develop a server side application displaying the status and availability of all client systems. Unlike messages, notification traffic is bidirectional, and the server system may also send notifications to the client systems; additionally, the framework also contains a broadcast functionality for notifications, making it possible for the server system to broadcast a notification to all known client systems.

Notifications are also specifically used in connection with client-to-server file transfers, as described in 7.8, p. 22.

7.5 EVENTS

To simplify use of the framework, an event-driven model was adopted. When specific events occur – such as the receipt of a message, or the completion of a file transfer – interested parties are informed through the use of events. In the .Net environment, an event is an exposed member of a delegate type that can be subscribed to by external applications, and fired by the exposing type, at which point the subscribers are notified and can handle the received data as they see fit. The delegate types exposed by the framework are as follows:

- delegate void MessageEvent(MessageBase message) An event concerning the message message, such as the successful receipt of the message, has occurred. This is the delegate type most often used, for instance to notify server side subscribers that a message has arrived from a client. The source of the message is detailed in message.

- delegate void NotificationEvent(NotificationBase notification) An event concerning the notification notification has occurred.

- delegate void FileCompleteEvent(byte[] checksum, string path, FileInfoNotification info) A file transfer is complete. The file has been written to a file system at location path, the SHA256 checksum (see 7.8.1, p. 24) of the file contents is checksum, and additional file transfer information is supplied in info.

Using these delegate types, applications using the framework can define event handlers, which are methods with the signature defined by the delegate types, and are used to subscribe to the events exposed by the framework. The only event member exposed by the framework (as defined in ITMS_Clientside_Local_Backend) on the client side is as follows.

- event NotificationEvent OnRecievedNotification This event is fired whenever the client receives a notification from the server. The notification
On the server side, the framework exposes the following event members (as defined in ITMS_ServerSide_Local_Backend).

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MessageEvent OnRecievedMessage</code></td>
<td>This event is fired whenever the server has received a message. The message object is passed along when the event is fired. Not all received messages will generate this event (see 7.8, p. 22), while some may result in the event being fired multiple times due to batching (see 7.7, p. 22).</td>
</tr>
<tr>
<td><code>NotificationEvent OnRecievedNotification</code></td>
<td>This event is fired whenever the server receives a notification from a client. The notification object is passed along when the event is fired.</td>
</tr>
<tr>
<td><code>FileCompleteEvent OnFileComplete</code></td>
<td>This event is fired whenever a file transfer is complete.</td>
</tr>
</tbody>
</table>

All of the above mentioned event members are available to applications wishing to be notified of the corresponding events.

### 7.6 COMPRESSION

As discussed in depth below, serialization of generic data followed by transport over WCF may incur a significant bandwidth penalty. One way to combat this (the other being discussed below) is the use of lossless compression.

To this end, two compression algorithms are included in the framework prototype; **DEFLATE**, a commonly used general-purpose compression algorithm that has been proven to perform well in most scenarios, and **LZMA**, which is a newer and more complex algorithm that has been shown to outperform **DEFLATE** in some cases. While both these algorithms are variants of dictionary compression, they differ in some details.

**DEFLATE** uses the **LZ77** algorithm [6] followed by Huffman coding to reduce the length of commonly occurring symbols. This compressor is most commonly known for its use in the **PKZIP** utility, and will probably be familiar to most users. The actual implementation of the algorithm used is the one provided in the .Net framework.

**LZMA** uses a dictionary compression algorithm similar to **LZ77**, followed by a range encoder (a variant of arithmetic coding). The algorithm is commonly known for first being used in the **7-Zip** utility. The implementation used in the framework prototype is a slightly modified version of the reference implementation, available from [7].

A comparison of the effectiveness of these two compressors on simulated data sent through the framework is presented in 8, p. 25. A quantitative comparison of the execution time of the algorithms has not been included in this report.
7.7 MESSAGE BATCHING

The transparent data serialization features supplied by WCF, while very useful, may incur significant metadata overhead. This, combined with the generally expensive nature of network communication, leads to the need for message batching.

When an object is serialized by WCF and sent as a parameter to a remote method call (e.g. a MessageBase object being sent from the client as a parameter to ReceiveMessage() on the server), two types of metadata are added. First, type information is appended to the object to facilitate type-safe deserialization; second, before the data is sent on the wire, metadata regarding the target method is also added, and sent with the serialized data.

For large objects this is not much of a performance issue, but as the object size grows smaller the metadata makes up a larger portion of the transmitted data, which may lead to a large portion of the available bandwidth being used. This could work against requirement C (4.1.3, p. 9); thus, a way to minimize this overhead must be implemented.

The solution to these issues suggested in this work is message batching. This entails two optimizations; first, messages are concatenated together, serialized, and sent as one unit, hoping to lower the impact of the second metadata addition described above (as it will only be appended once per group of messages, instead of once per individual message); and second, these concatenated messages may be compressed (see 7.6, p. 21) resulting in a lower total size. Serialization metadata is still added to each message individually; however, the use of compression has proven to lessen the performance impact of this issue (see 8, p. 25).

7.8 FILE TRANSFERS

Functional file transfer is one of the most important requirements for, and features of, the framework. Conceptually, four steps need to be performed to transfer a file between two remote systems:

1. The sender has to read the file and perform any necessary transformations to the contents, to make them suitable for transmission;
2. The sender has to transmit the transformed contents;
3. The receiver has to receive the transmission, and reverse any transformations performed in point 1;
4. The integrity of the received file should be verified.

While this list is conceptual and the actual steps vary (for instance, it is assumed that a connection exists between the endpoints of the transmission), historically systems have often more or less adhered to these steps. For instance, for a time it was relatively common to encounter transmission channels that were not “8-bit clean”, meaning the channel itself assigned special meaning to one of the bits in each transmitted byte. This implied that the transmitted data had to be transformed to make sure that this special bit in each byte was available for the channel and
did not interfere with transmission\(^1\). The most common solutions to this problem are various binary-to-text encodings, which encode the data as 7-bit ASCII characters, leaving the eighth bit in each byte unused.

Binary-to-text encoding is also used in scenarios where binary data is sent over a channel that expects text, such as SOAP or XML-formatted information.

In the communication framework, the transformation performed is of a different kind\(^2\). Data serialization is handled by the underlying communication stack, but since communication through the framework is message-based, the contents of the file being sent have to be encapsulated as one or more messages. The trivial thing to do would be to create one message containing the whole file; however, for large files, this would possibly be detrimental on performance. Consequently, the file is split into chunks, each of which is contained in a separate message before being queued for transmission by the framework. The size of these chunks is configurable (see 7.9, p. 25).

In order to maximize bandwidth usage, according to requirement F (4.1.6, p. 9), data compression is optionally utilized for file transfers. Two compression algorithms are provided in the prototype, and care has been taken to make it easy to add additional algorithms. See 7.6, p. 21 for a discussion around compressing in general and the algorithms provided. When used, compression is performed on the initial file contents, before they have been split into chunks.

The reason for this is twofold. First, significant amounts of the data contained in normal log files (as well as other commonly encountered file types) may be redundant, or at the very least similar. Splitting the file into chunks prior to compression would likely also split some of the redundancy across several chunks, thus increasing the information entropy ratio in each individual chunk, which would lead to lower compression performance (see 8, p. 25). The second reason is that some compression formats may attach format-specific headers or metadata to the result of the compression; splitting the data prior to compression would then lead to this extra information being attached to each chunk individually, which would result in increased total overhead. These considerations thus mirror the ones regarding message batching (see 7.7, p. 22).

The performance implications of message batching and payload size are further discussed in 8, p. 25.

Thus, client-to-server file transmission in the communication framework prototype can be described with the following steps:

1. A client application calls \texttt{SendFileToServer()} exposed by the client backend, providing a path on the local file system, the compression type to use, the chunk size, and the message delay.

\(^1\) This is the case with several communication standards, such as historical email or RS-232.
\(^2\) However, the SOAP/XML binary-to-text transformation may be performed by the underlying WCF architecture in the prototype case.
2. The client backend computes a hash of the file contents.
3. The client backend compresses the file using the specified compressor, before splitting it into chunks of the given size or less.
4. The client backend generates a file identification number, and prepares a `FileInfoNotification` object, populating it with the following information:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileID</td>
<td>File identification number; only one file with the same identification number will be queued for transmission at any one time by an individual client.</td>
</tr>
<tr>
<td>Filename</td>
<td>Original path and filename of the file on the client system.</td>
</tr>
<tr>
<td>Compression</td>
<td>Compression used; the server will decompress the file contents using the same compressor upon receipt.</td>
</tr>
<tr>
<td>RawSize</td>
<td>Original size of the file (before compression), in bytes.</td>
</tr>
<tr>
<td>CompressedSize</td>
<td>Compressed size of the file, in bytes.</td>
</tr>
<tr>
<td>NumParts</td>
<td>Total number of chunks the file was split into.</td>
</tr>
<tr>
<td>Hash</td>
<td>Original hash value in string form.</td>
</tr>
</tbody>
</table>

5. The `FileInfoNotification` object and all chunks of the file (transmitted as `FilePartMessage` objects) are asynchronously queued for transmission using the underlying communication stack.
6. The receiving server system collects all received `FileInfoNotification` objects and file chunks on a per-ID basis.
7. When the notification has been received, together with all chunks constituting the file in question (the total number of chunks being transmitted as part of the notification), the server backend asynchronously joins the chunks together, in order, before decompressing the concatenated contents using the proper compressor.
8. The server backend computes the hash of the concatenated and decompressed file, and compares the result to the original hash. If the hashes match, the file transfer is deemed successful, the file is written to the server file system, and the `OnFileComplete` event is fired; otherwise, the file is discarded and any transmission error handling routines are invoked. In the prototype, only a diagnostic message is logged.

### 7.8.1 CHECKSUM HASHERS

To ensure reliable file transfer, a checksum is calculated at two points during the file transfer. First, before the file is split into chunks on the client side, a hash is calculated from the file contents, and sent along with the `FileInfoNotification`, notifying the server system of a beginning
file transfer. When all the pieces of the file have been transferred to the server system, and concatenated in the correct order, a second hash is calculated and compared to the first hash. If the hashes match, it can be assumed that the contents of the file now on the server system are identical to the original contents, and the transfer is deemed successful. Since the length of the hash is in the majority of cases shorter than the length of the file contents, the hash function is not injective; thus, cases where different file contents map to the exact same hash are not only possible but certain (since the set of all possible file contents is infinite, while the set of all possible hashes of a given length is not). Such a situation is termed a collision.

The hash algorithm used in the framework prototype is SHA256 [9]. This algorithm is commonly known, and used in various cryptographic and security scenarios. While theoretical attacks exist on reduced-complexity variants of the algorithm ([10], [11]), it is deemed more than secure enough for use in the framework prototype. The purpose of the hash comparison between the pre-transfer and post-transfer file contents is to detect transmission errors; the probability of a transmission error (changing either the file contents, or the transmitted first hash) resulting in a hash collision is, for all intents and purposes, negligible.

7.9 CONFIGURATION

Several pieces of information need to be supplied to the framework at run-time to enable successful operation. Each client system needs to be aware of the following:

1. Local MSMQ queue name, for incoming notifications. This queue is created at client start-up if it does not exist;
2. WCF base address for service metadata discovery;
3. Descriptive name of the client system;
4. A unique (among all clients associated with the current server) numerical ID of the client system;
5. A block size, used for file transfers (see 7.8, p. 22);
6. A batch size, used for message batching (see 7.7, p. 22);
7. A file block delay, used for file transfers (see 7.8, p. 22);
8. Endpoint addresses of the MSMQ queues on the server system; and
9. Various MSMQ configuration options.

These are stored in an XML file supplied with the prototype, and further information about each variable is supplied in the file.

8. RESULTS
In order to verify that the proposed framework prototype satisfies the requirements, a number of tests were undertaken. Two types of tests were performed; qualitative compliance tests, wherein it is determined whether the prototype can handle a given scenario, and quantitative performance tests, wherein the performance and effectiveness of the prototype are measured.

While the compliance tests were an important part of the development process, and were mostly undertaken on an *ad hoc* basis during development, the quantitative performance tests were mostly performed after the prototype was complete. They lead to interesting conclusions regarding the implementation specifically, but also regarding the suitability of techniques such as web services, message batching, and compression, as solutions to the current scenario and problems in general.

### 8.1 METHODOLOGY

The following performance test regime was performed on a “closed-circuit” network containing only two hosts, namely the client and the server. The former was an industrial PC intended to be placed on the train, while the server was a regular office PC. Both machines were running the respective framework backend implementation and associated test applications. The prototype used the reliable, recoverable queue named *queue_keep* on the server. After initiating data generation and transmission from the client side, actual network traffic (i.e. the number of bytes sent on the wire) was measured with commercially available tools. Each test was deemed completed when the following three conditions were met:

1. All messages have been generated and sent to the framework on the client side;
2. The incoming *MSMQ* queue on the server side was empty (thus, all generated messages had been forwarded to the framework by *MSMQ*); and
3. No relevant network traffic between the machines had occurred for 10 consecutive seconds.

The purpose of this series of tests was fivefold:

1. Determine the overhead associated with message serialization and transport, by comparing a known quantity of payload data with the actual quantity of data sent on the wire;
2. Determine the effect, if any, of different payload (and, consequently, message) sizes on the overhead measured in point 1;
3. Determine the effect, if any, of message batching, using batches of various sizes, on the overhead measured in points 1 and 2;
4. Determine the effectiveness and suitability of the available compression algorithms, by comparing the quantity of data on the wire when using a specific algorithm to the quantity observed when no compression is used, and all other parameters remain identical; and
5. Determine the effect, if any, of payload entropy on the results measured in points 1 through 4, by performing tests with all parameters identical, except for the payloads.
consisting of either pseudo-random data (with a high information entropy), or all-zero data (with a much lower information entropy).

Since the traffic was measured at a low abstraction level, all types of metadata and other transport overhead are included in the measurements, and part of the motivation behind these tests was indeed to measure such overhead. Since network traffic is not readily deterministic, the tests were repeated thrice each, and the presented measurements are averages rounded up to the nearest whole byte; however, the measurements did not vary between each individual sample and their relative differences never exceeded 10% of the highest sample.

In all tests, the payload size (and consequently also the message size) and total number of messages were controlled, so that the total quantity of payload was equal to 1 MiB. Thus, given only the payload size $S$, the number of messages $N$ sent as part of any test can be calculated by

$$N = \frac{2^{20}}{S}$$

In the following tables and graphs, $N$ is omitted for clarity. Random messages refer to messages where the payload consists solely of pseudo-random data generated by the .Net pseudo-random number generator, while Zero messages refer to messages with an all-zero payload. Batching refers to the mechanism described in 7.7, p. 22.

The averaged measurements are presented in Table 1 and Table 2, and are summarized and discussed below.

### 8.2 MEASUREMENTS

The series of tests presented in Figure 2 considers the average quantity of transmitted data when batching is not used. It is clear that payload size has a drastic effect on the measured quantity, and for small messages the added overhead is prohibitive; for the smallest message sizes, upwards of 72% of the transmitted data consists of overhead.

Additionally, it can be noted that payload information entropy plays no significant part in performance in this series of tests, as the values for both random and all-zero payloads are very similar. The results clearly show that the amount of transmitted data is not a linear function of the payload size. This implies that the generated per-message overhead is significant, as discussed in 7.7.

The next series of tests, presented in Figure 3 through Figure 6, was used to measure the effects of message batching and compression on transferred data amounts, where all messages contained random payloads. Figure 3 shows a summary of the data while the following figures present the results for each compressor used separately. It is shown that the both message batching and compression do not play a significant role when the payload size is sufficiently large; this is due to the overall information entropy of the transmitted data being high, as a large payload size leads to a low number of transmitted messages, thus limiting the amount of metadata transmitted. With a
high enough payload size the amount of transmitted data approaches the expected amount. However, for smaller payload sizes the amount of transferred data increases dramatically and the effects of both used techniques are drastic. Compression performance increases as payload size decreases, but is not enough to offset the increased metadata overhead for the smaller message sizes, and the **LZMA** compressor shows consistently better performance when compared to **DEFLATE**. Message batching is also shown to be a very effective technique, with the larger batch sizes consistently lowering the amount of data transmitted, due to less metadata overhead.

The data also suggests that neither technique is useful or necessary for payloads larger than 512 bytes, as the measured values for all batch sizes and compressors converge at this point.

Figure 4 through Figure 6 show that all three compressors have similar performance characteristics, while the **LZMA** compressor is superior for low payload sizes, and provides a slightly flatter curve overall.

The results presented in Figure 7 reinforce these conclusions. When transmitting a payload of 256 random bytes using different batch sizes and compressors, the choice of compressor is important, but less so than an increase in batch size. The **DEFLATE** compressor is also shown to only be a slight improvement on not using compression at all in this case.

The next series of tests, presented in Figure 8 through Figure 11 mirrors the results above, with some additional conclusions. In this series, payloads are all-zero; this drastically lowers the average information entropy of the transmitted data. Compression proves much more useful, and both compressors are able to lower the total amount of transmitted data significantly. Additionally, the results show that even when no compressor is used, less data is transmitted in total for payload sized of 16 to 512 bytes, when compared to the previous test series. The reasons for this are unknown. It is hypothesized that this relates to some optimizations in technologies in the underlying communication stack. Again, the values for all batch sizes converge, and batching has a negligible impact on performance when considering payload sizes greater than 1 KiB.

Compression is shown to be much more useful in this test series, and, for all but the smallest payload sizes, the effect of compression changes only slightly across the different batch size series; this change is most noticeable when using the higher-performance **LZMA** compressor.

For the smallest payload sizes, it is shown that while compression proves somewhat useful, the total amounts of transmitted data are still exceedingly high. This is due to the small payload size leading to the transmission of a large number of small messages; thus, more of the total transmitted data will consist of per-message metadata, with an information entropy significantly higher than the all-zero payloads. In these cases, the **DEFLATE** and **LZMA** compressors perform similarly, providing a nearly constant reduction in total data transmitted across all payload sizes. The **LZMA** compressor is consistently slightly more efficient, and the total amount of transmitted data again decreases with an increase of payload size, as discussed above. For large payload sizes, compression is shown to be a very effective means of transmitting low-entropy data.
An increase in batch size improves the efficiency of compression even further. Additionally, the **LZMA** compressor consistently outperforms **DEFLATE** by a larger margin with increasing batch sizes. Since the main sources of information entropy in this test series are the various types of message metadata, it can be expected that the average entropy of the transmitted data decreases with increasing payload and batch sizes (as discussed above); consequently, compression drastically lowers the total amount of data transmitted in these cases. With the maximum batch size (128 messages per batch) and payload size (8 KiB payload per message) tested, use of the **LZMA** compressor results in circa 4 KiB transmitted, as compared to 1 MiB of generated data. These conditions, and the low information entropy of the data, are however unlikely to be observed in general use.

The data presented in Figure 2 through Figure 12 are also available in Table 1 (random payloads) and Table 2 (all-zero payloads). The cells in these tables have been color-coded with respect to bytes-on-wire (red is higher, green is lower) to aid in reading.

### 9. FUTURE WORK

Several avenues of research have not been pursued in this thesis, and are suggested as topic which future work might explore:

1. A more comprehensive study of the compression effectiveness and subsequent bandwidth usage of real-life payloads with varied information entropy;
2. An in-depth investigation into the computational performance of the provided compression algorithms;
3. A comparison of the message queue-based approach to the problem presented in this thesis with other alternative solutions, such as RUDP [8].

Additionally, the framework prototype presented in this thesis is an open project, and will be adapted to future situations and environments.

### 10. SUMMARY

A framework design pursuant to the requirements has been presented, and a prototype implementation of the design has been completed and tested successfully; at the time of this writing, the prototype has not been field tested, although several bench and lab tests have been performed successfully. Results of performance tests are provided, showing that compression and message batching are effective tools to lower the overhead associated with the serialization and transport of large numbers of small messages over web services and WCF. Data types commonly encountered on the Flytoget train system have been investigated and presented, and a class library has been designed and implemented to allow the transport of these data types.
The author wishes to extend his thanks to Stefan Persson and Ola Sellin, for inspiration and an effective work environment, without whom the completion of this thesis would not have been possible. Björn Lisper has thanks for guiding the author through the thesis, and Max Leander-Koszela also provided valuable input and comments and has the author’s gratitude.

12. REFERENCES


13. FIGURES & TABLES

Figure 1: Summary of prototype communication stack

Figure 2: Random and zero data, no compression, no batching, against payload size
Development of a Communication Framework for optimized and guaranteed information exchange between train and wayside

Figure 3: Effects of batch size, compression and payload size on bytes-on-wire for random payloads

Figure 4: Effects of payload and batch sizes on bytes-on-wire for uncompressed random data
Development of a Communication Framework for optimized and guaranteed information exchange between train and wayside.

Figure 5: Effects of payload and batch sizes on bytes-on-wire for random data compressed with DEFLATE

Figure 6: Effects of payload and batch sizes on bytes-on-wire for random data compressed with LZMA
Development of a Communication Framework for optimized and guaranteed information exchange between train and wayside

![Graph showing effects of batch size, compression, and payload size on bytes-on-wire for all-zero payloads.](image)

**Figure 7:** Random data, 256 bytes payload per message, compression types, against batch size

**Figure 8:** Effects of batch size, compression and payload size on bytes-on-wire for all-zero payloads
Figure 9: Effect of payload and batch sizes on bytes-on-wire for uncompressed all-zero data

Figure 10: Effects of payload and batch sizes on bytes-on-wire for all-zero data compressed with DEFLATE
Figure 11: Effects of payload and batch sizes on bytes-on-wire for all-zero data compressed with LZMA

Figure 12: Zero data, 256 bytes payload per message, compression types, against batch size
<table>
<thead>
<tr>
<th>Payload size</th>
<th>Total msgs.</th>
<th>Compr.</th>
<th>Random, 16 msgs/batch</th>
<th>Random, 32 msgs/batch</th>
<th>Random, 64 msgs/batch</th>
<th>Random, 128 msgs/batch</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>65536</td>
<td>DEFLATE</td>
<td>9446990</td>
<td>5994238</td>
<td>4658380</td>
<td>3642490</td>
</tr>
<tr>
<td>16</td>
<td>65536</td>
<td>LZMA</td>
<td>7445314</td>
<td>4713126</td>
<td>3303598</td>
<td>2246026</td>
</tr>
<tr>
<td>16</td>
<td>65536</td>
<td>(none)</td>
<td>11287380</td>
<td>7684274</td>
<td>5970348</td>
<td>4916452</td>
</tr>
<tr>
<td>32</td>
<td>32768</td>
<td>DEFLATE</td>
<td>5537306</td>
<td>3873832</td>
<td>3002212</td>
<td>2641924</td>
</tr>
<tr>
<td>32</td>
<td>32768</td>
<td>LZMA</td>
<td>4501150</td>
<td>2894124</td>
<td>2109520</td>
<td>1710612</td>
</tr>
<tr>
<td>32</td>
<td>32768</td>
<td>(none)</td>
<td>6168486</td>
<td>4367560</td>
<td>3525938</td>
<td>3063004</td>
</tr>
<tr>
<td>64</td>
<td>16384</td>
<td>DEFLATE</td>
<td>3609948</td>
<td>2770282</td>
<td>2388312</td>
<td>1941952</td>
</tr>
<tr>
<td>64</td>
<td>16384</td>
<td>LZMA</td>
<td>2785366</td>
<td>2022422</td>
<td>1614760</td>
<td>1422654</td>
</tr>
<tr>
<td>64</td>
<td>16384</td>
<td>(none)</td>
<td>3700260</td>
<td>2729040</td>
<td>2295702</td>
<td>2081738</td>
</tr>
<tr>
<td>128</td>
<td>8192</td>
<td>DEFLATE</td>
<td>2426494</td>
<td>1959368</td>
<td>1812682</td>
<td>1546770</td>
</tr>
<tr>
<td>128</td>
<td>8192</td>
<td>LZMA</td>
<td>1970324</td>
<td>1600394</td>
<td>1385376</td>
<td>1295254</td>
</tr>
<tr>
<td>128</td>
<td>8192</td>
<td>(none)</td>
<td>2404656</td>
<td>1950050</td>
<td>1720242</td>
<td>1606130</td>
</tr>
<tr>
<td>256</td>
<td>4096</td>
<td>DEFLATE</td>
<td>2025444</td>
<td>1666292</td>
<td>1556466</td>
<td>1424296</td>
</tr>
<tr>
<td>256</td>
<td>4096</td>
<td>LZMA</td>
<td>1782144</td>
<td>1498318</td>
<td>1342632</td>
<td>1262462</td>
</tr>
<tr>
<td>256</td>
<td>4096</td>
<td>(none)</td>
<td>2004660</td>
<td>1662860</td>
<td>1480800</td>
<td>1424550</td>
</tr>
<tr>
<td>512</td>
<td>2048</td>
<td>DEFLATE</td>
<td>1582156</td>
<td>1414826</td>
<td>1388378</td>
<td>1257868</td>
</tr>
<tr>
<td>512</td>
<td>2048</td>
<td>LZMA</td>
<td>1466218</td>
<td>1324066</td>
<td>1239490</td>
<td>1202058</td>
</tr>
<tr>
<td>512</td>
<td>2048</td>
<td>(none)</td>
<td>1548266</td>
<td>1404044</td>
<td>1301212</td>
<td>1306066</td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
<td>DEFLATE</td>
<td>1347810</td>
<td>1260558</td>
<td>1234026</td>
<td>1192166</td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
<td>LZMA</td>
<td>1317792</td>
<td>1232590</td>
<td>1190134</td>
<td>1194388</td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
<td>(none)</td>
<td>1342802</td>
<td>1276126</td>
<td>1268066</td>
<td>1191294</td>
</tr>
<tr>
<td>2048</td>
<td>512</td>
<td>DEFLATE</td>
<td>1235114</td>
<td>1191948</td>
<td>1182466</td>
<td>1160048</td>
</tr>
<tr>
<td>2048</td>
<td>512</td>
<td>LZMA</td>
<td>1260624</td>
<td>1185592</td>
<td>1166320</td>
<td>1156554</td>
</tr>
<tr>
<td>2048</td>
<td>512</td>
<td>(none)</td>
<td>1232048</td>
<td>1189520</td>
<td>1171174</td>
<td>1158476</td>
</tr>
<tr>
<td>4096</td>
<td>256</td>
<td>DEFLATE</td>
<td>1181674</td>
<td>1158488</td>
<td>1155272</td>
<td>1142980</td>
</tr>
<tr>
<td>4096</td>
<td>256</td>
<td>LZMA</td>
<td>1181470</td>
<td>1163118</td>
<td>1154154</td>
<td>1176102</td>
</tr>
<tr>
<td>4096</td>
<td>256</td>
<td>(none)</td>
<td>1178730</td>
<td>1158362</td>
<td>1148258</td>
<td>1139796</td>
</tr>
<tr>
<td>8192</td>
<td>128</td>
<td>DEFLATE</td>
<td>1153990</td>
<td>1198652</td>
<td>1140806</td>
<td>1130976</td>
</tr>
<tr>
<td>8192</td>
<td>128</td>
<td>LZMA</td>
<td>1160384</td>
<td>1151680</td>
<td>1147862</td>
<td>1144878</td>
</tr>
<tr>
<td>8192</td>
<td>128</td>
<td>(none)</td>
<td>1151662</td>
<td>1141226</td>
<td>1136304</td>
<td>1134230</td>
</tr>
</tbody>
</table>

Table 1: Bytes on wire for random payloads using different compression types, payload sizes and batch sizes; table cells color-coded with respect to bytes on wire.
<table>
<thead>
<tr>
<th>Payload size</th>
<th>Total msgs.</th>
<th>Compr.</th>
<th>Zero, 16 msgs/batch</th>
<th>Zero, 32 msgs/batch</th>
<th>Zero, 64 msgs/batch</th>
<th>Zero, 128 msgs/batch</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>65536</td>
<td>DEFLATE</td>
<td>10413836</td>
<td>6092766</td>
<td>2593894</td>
<td>1791370</td>
</tr>
<tr>
<td>16</td>
<td>65536</td>
<td>LZMA</td>
<td>9469750</td>
<td>3769822</td>
<td>1816230</td>
<td>1086064</td>
</tr>
<tr>
<td>16</td>
<td>65536 (none)</td>
<td></td>
<td>13703280</td>
<td>8331432</td>
<td>5759560</td>
<td>4918564</td>
</tr>
<tr>
<td>32</td>
<td>32768</td>
<td>DEFLATE</td>
<td>4829616</td>
<td>2720134</td>
<td>1386128</td>
<td>988026</td>
</tr>
<tr>
<td>32</td>
<td>32768</td>
<td>LZMA</td>
<td>4803388</td>
<td>2755590</td>
<td>943624</td>
<td>549504</td>
</tr>
<tr>
<td>32</td>
<td>32768 (none)</td>
<td></td>
<td>7830094</td>
<td>4817310</td>
<td>3529410</td>
<td>3035290</td>
</tr>
<tr>
<td>64</td>
<td>16384</td>
<td>DEFLATE</td>
<td>2892270</td>
<td>1616340</td>
<td>703298</td>
<td>503184</td>
</tr>
<tr>
<td>64</td>
<td>16384</td>
<td>LZMA</td>
<td>2435738</td>
<td>1398890</td>
<td>943624</td>
<td>549504</td>
</tr>
<tr>
<td>64</td>
<td>16384 (none)</td>
<td></td>
<td>4456688</td>
<td>2295132</td>
<td>281124</td>
<td>1218594</td>
</tr>
<tr>
<td>128</td>
<td>8192</td>
<td>DEFLATE</td>
<td>1438326</td>
<td>875858</td>
<td>375086</td>
<td>255446</td>
</tr>
<tr>
<td>128</td>
<td>8192</td>
<td>LZMA</td>
<td>1206080</td>
<td>705706</td>
<td>233908</td>
<td>139146</td>
</tr>
<tr>
<td>128</td>
<td>8192 (none)</td>
<td></td>
<td>2798156</td>
<td>2182376</td>
<td>1606730</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>4096</td>
<td>DEFLATE</td>
<td>743328</td>
<td>289314</td>
<td>263826</td>
<td>191184</td>
</tr>
<tr>
<td>256</td>
<td>4096</td>
<td>LZMA</td>
<td>637940</td>
<td>215088</td>
<td>195634</td>
<td>112998</td>
</tr>
<tr>
<td>256</td>
<td>4096 (none)</td>
<td></td>
<td>2000996</td>
<td>1481682</td>
<td>1427456</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>2048</td>
<td>DEFLATE</td>
<td>380474</td>
<td>146758</td>
<td>141242</td>
<td>99164</td>
</tr>
<tr>
<td>512</td>
<td>2048</td>
<td>LZMA</td>
<td>321108</td>
<td>111026</td>
<td>100854</td>
<td>61770</td>
</tr>
<tr>
<td>512</td>
<td>2048 (none)</td>
<td></td>
<td>1548144</td>
<td>1331778</td>
<td>1256780</td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
<td>DEFLATE</td>
<td>194708</td>
<td>77694</td>
<td>75898</td>
<td>60652</td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
<td>LZMA</td>
<td>163038</td>
<td>57352</td>
<td>51106</td>
<td>31536</td>
</tr>
<tr>
<td>1024</td>
<td>1024 (none)</td>
<td></td>
<td>1344616</td>
<td>1232874</td>
<td>1214690</td>
<td>1192080</td>
</tr>
<tr>
<td>2048</td>
<td>512</td>
<td>DEFLATE</td>
<td>102512</td>
<td>45142</td>
<td>43242</td>
<td>33354</td>
</tr>
<tr>
<td>2048</td>
<td>512</td>
<td>LZMA</td>
<td>82358</td>
<td>30240</td>
<td>27526</td>
<td>19006</td>
</tr>
<tr>
<td>2048</td>
<td>512 (none)</td>
<td></td>
<td>1234440</td>
<td>1177608</td>
<td>1169518</td>
<td>1156618</td>
</tr>
<tr>
<td>4096</td>
<td>256</td>
<td>DEFLATE</td>
<td>56606</td>
<td>27660</td>
<td>26952</td>
<td>21532</td>
</tr>
<tr>
<td>4096</td>
<td>256</td>
<td>LZMA</td>
<td>41562</td>
<td>16112</td>
<td>13470</td>
<td>7618</td>
</tr>
<tr>
<td>4096</td>
<td>256 (none)</td>
<td></td>
<td>1177842</td>
<td>1152618</td>
<td>1148244</td>
<td>1139558</td>
</tr>
<tr>
<td>8192</td>
<td>128</td>
<td>DEFLATE</td>
<td>32948</td>
<td>20276</td>
<td>18714</td>
<td>17710</td>
</tr>
<tr>
<td>8192</td>
<td>128</td>
<td>LZMA</td>
<td>20970</td>
<td>9776</td>
<td>6796</td>
<td>4442</td>
</tr>
<tr>
<td>8192</td>
<td>128 (none)</td>
<td></td>
<td>1151738</td>
<td>1139100</td>
<td>1137330</td>
<td>1134454</td>
</tr>
</tbody>
</table>

Table 2: Bytes on wire for zero payloads using different compression types, payload sizes and batch sizes; table cells color-coded with respect to bytes on wire
The Airport Express Train
Development of a Communication Framework for optimized and guaranteed information exchange between train and wayside

When the new main airport for Norway opened in October 1998, NSB Gardermobanen AS put 16 express trains into service on the new rail link from Oslo to Gardermoen Airport.

The Airport Express Train will cover the 48 km stretch from Oslo to Gardermoen in just 19 minutes. The trainsets are designed to have a low internal and external noise level.

Each trainset, consisting of three cars, is equipped with 170 comfortable seats, with space reserved for wheelchair users. There will also be plenty of room for luggage. One or two intermediate cars, driven or trailer, can be added. A trainset comprises a maximum of four standard units in multiple operation.

The air conditioning system keeps the air in each car fresh and at a pleasant temperature. The Airport Express Train has pressure-sealed compartments, which means that it can pass through tunnels without any discomfort to the passengers. Both the designer and the producer have paid particular attention to reliability – passengers can relax in the knowledge that their journey will not be delayed. For this reason necessary maintenance will be guided by LCC (Life Cycle Cost) and RCM (Reliability Centred Maintenance).
15. APPENDIX 2: TMS SYSTEM OVERVIEW

PC Software overview – Landside/Onboard train

Hardware overview
Industrial vehicle PC

- **NEXCOM VTC-6100 (or similar)**
  - IP65 compliant (w/ VTK 60P)
  - CE, FCC and e13 approved
  - Operating Temperature:
    - -30°C ~ +60°C with CF or automotive HDD

- **Function**
  - Starts/shuts down when train is activated/deactivated (Wake On Power-function)
  - Contains the software that communicates with wayside