Simulation of Process Control
Network Traffic

Viktor Andersson
Tommi Nylander

Department of Automatic Control
Abstract

The majority of industrial control systems today are distributed over networks. These systems tend to be complex, and can be difficult both to configure and to analyze; therefore, there is definitely a gain in being able to simulate the behaviour of such a system prior to the actual implementation.

This master’s thesis investigates the possibility to simulate the ABB automation system 800xA by expanding the Matlab/Simulink tool TrueTime. Throughout, the work has been focused on modeling the network communications and the internal behaviour of the control system nodes. Furthermore, the developed simulator has been used to examine a proposal to optimize traffic handling of the ABB specific protocol IAC.

Different performance measures have been investigated, both regarding control and network performance. One measure of particular interest has been the round trip time of a packet. The simulator has proven to be able to reproduce results of round trip times measured in an example system, explaining timing behaviour originating deep down in the control system nodes. Other simulations have involved different system settings, analyzing their impact on system performance.
Acknowledgements

We would like to express our special appreciation to our supervisor Svengunnar Tiljander for his constant support during our time at ABB. Also, we would like to thank Adam Norén for his curiosity in our master’s thesis and his ideas and help regarding the optimization of IAC traffic handling. For practical matters, Andreas Ekstrand has helped us and furthermore he presented the opportunity of doing the master’s thesis at ABB to begin with. Prof. Karl-Erik Årzén, our representative at LTH, has given much valuable ideas for the report.
# Contents

1. **Introduction** .................................................. 1  
   1.1 Background .................................................... 1  
   1.2 Goals .......................................................... 1  
   1.3 Individual Contributions .................................... 2  
   1.4 Outline ....................................................... 3  

2. **Overview of the 800xA System**  ............................................. 4  
   2.1 Usage ........................................................... 4  
   2.2 Hardware ...................................................... 5  
   2.3 Software ....................................................... 6  

3. **Network Structure in 800xA** ................................................ 10  
   3.1 Protocols used in 800xA ...................................... 12  
   3.2 Switched Ethernet ............................................. 12  
   3.3 RNRP - Redundant Network Routing Protocol ................... 14  
   3.4 UDP - User Datagram Protocol ................................ 16  
   3.5 TCP - Transmission Control Protocol ......................... 16  
   3.6 IAC - Inter Application Communication ....................... 23  
   3.7 MMS - Manufacturing Message Specification ................. 25  

4. **Simulation of Real Time Network Distributed Systems**  .................. 26  
   4.1 Simulation Tool Alternatives .................................. 26  
   4.2 TrueTime ........................................................ 28  

5. **Modeling and Simulation of 800xA** ......................................... 35  
   5.1 Helper File ................................................... 35  
   5.2 Implementation in TrueTime ................................... 35  
   5.3 Simulating RNRP with TrueTime ................................ 38  
   5.4 Simulating TCP with TrueTime ................................ 42  
   5.5 Simulating IAC with TrueTime ................................ 47  
   5.6 Simulating MMS with TrueTime ................................ 50  

6. **Results of TrueTime Simulations** .......................................... 53  
   6.1 Sensor/Actuator - Regulator Example .......................... 53
6.2 Measuring Performance ................................ 56
6.3 Handling of Packet Losses .............................. 60
6.4 Investigating RNRP Settings ............................ 62

7. Optimization of IAC traffic .............................. 67
   7.1 The Current Implementation ......................... 67
   7.2 The Proposed Optimization ......................... 68
   7.3 When Are Faster Round Trip Times Beneficial? .... 70

8. Results and Validation of IAC Optimization .......... 71
   8.1 Measurement Results ................................ 71
   8.2 Simulation Accuracy ................................ 77
   8.3 Comparison to Current Implementation .............. 87

9. Conclusions .............................................. 89
   9.1 Using TrueTime to Model 800xA ..................... 89
   9.2 Optimizing IAC Traffic .............................. 89
   9.3 Future Work ........................................ 90

Bibliography .............................................. 92
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>AC 800M controller</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>800xA operation station</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>Project in Control Builder</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>800xA network overview</td>
<td>11</td>
</tr>
<tr>
<td>3.2</td>
<td>OSI model</td>
<td>13</td>
</tr>
<tr>
<td>3.3</td>
<td>Switched Ethernet</td>
<td>14</td>
</tr>
<tr>
<td>3.4</td>
<td>RNRP network topology</td>
<td>15</td>
</tr>
<tr>
<td>3.5</td>
<td>UDP header</td>
<td>16</td>
</tr>
<tr>
<td>3.6</td>
<td>TCP header</td>
<td>17</td>
</tr>
<tr>
<td>3.7</td>
<td>TCP connection establishment and termination</td>
<td>18</td>
</tr>
<tr>
<td>3.8</td>
<td>The TCP send window</td>
<td>19</td>
</tr>
<tr>
<td>3.9</td>
<td>TCP error control</td>
<td>21</td>
</tr>
<tr>
<td>3.10</td>
<td>TCP congestion control</td>
<td>23</td>
</tr>
<tr>
<td>3.11</td>
<td>IAC client</td>
<td>24</td>
</tr>
<tr>
<td>3.12</td>
<td>IAC server</td>
<td>24</td>
</tr>
<tr>
<td>3.13</td>
<td>MMS variable value exchange</td>
<td>25</td>
</tr>
<tr>
<td>4.1</td>
<td>The TrueTime block library</td>
<td>29</td>
</tr>
<tr>
<td>4.2</td>
<td>Network Schedule in Raw Form</td>
<td>31</td>
</tr>
<tr>
<td>4.3</td>
<td>RTSys class overview</td>
<td>33</td>
</tr>
<tr>
<td>5.1</td>
<td>Class overview of the <code>protocol_helper.cpp</code> file</td>
<td>36</td>
</tr>
<tr>
<td>5.2</td>
<td>Packet class structure</td>
<td>39</td>
</tr>
<tr>
<td>5.3</td>
<td>RNRP forwarding logic</td>
<td>41</td>
</tr>
<tr>
<td>5.4</td>
<td>TcpConnection class overview</td>
<td>43</td>
</tr>
<tr>
<td>5.5</td>
<td>IacHandler class overview</td>
<td>50</td>
</tr>
<tr>
<td>6.1</td>
<td>Regulator example</td>
<td>54</td>
</tr>
<tr>
<td>6.2</td>
<td>Smooth network utilization</td>
<td>59</td>
</tr>
<tr>
<td>6.3</td>
<td>MMS versus IAC with 10% loss probability</td>
<td>62</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>6.4 RNRP redirecting network traffic</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>6.5 Network schedules when RNRP is redirecting network traffic</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>6.6 Simulation of experiment when number of allowed hello packets lost is set to 0</td>
<td>65</td>
<td></td>
</tr>
<tr>
<td>7.1 Threads in current IAC implementation</td>
<td>68</td>
<td></td>
</tr>
<tr>
<td>7.2 Threads in filtered IAC implementation</td>
<td>69</td>
<td></td>
</tr>
<tr>
<td>8.1 Measured RTTs with default firmware - 5 IAC and 50 MMS</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>8.2 Histogram of measured RTTs with default firmware - 5 IAC and 50 MMS</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>8.3 Measured RTTs with optimized firmware - 5 IAC and 50 MMS</td>
<td>73</td>
<td></td>
</tr>
<tr>
<td>8.4 Histogram of measured RTTs with optimized firmware - 5 IAC and 50 MMS</td>
<td>73</td>
<td></td>
</tr>
<tr>
<td>8.5 Measured RTTs with default firmware - 5 IAC</td>
<td>74</td>
<td></td>
</tr>
<tr>
<td>8.6 Histogram of measured RTTs with default firmware - 5 IAC</td>
<td>74</td>
<td></td>
</tr>
<tr>
<td>8.7 Measured RTTs with optimized firmware - 5 IAC</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>8.8 Histogram of measured RTTs with optimized firmware - 5 IAC</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>8.9 New measurement of RTTs with optimized firmware - 5 IAC</td>
<td>76</td>
<td></td>
</tr>
<tr>
<td>8.10 Histogram of newly measured RTTs with optimized firmware - 5 IAC</td>
<td>76</td>
<td></td>
</tr>
<tr>
<td>8.11 Simulation RTTs before model adjustment</td>
<td>77</td>
<td></td>
</tr>
<tr>
<td>8.12 Difference between measured and actual RTTs</td>
<td>78</td>
<td></td>
</tr>
<tr>
<td>8.13 Protocol handler timing explained - case 1</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>8.14 Protocol handler timing explained - case 2</td>
<td>81</td>
<td></td>
</tr>
<tr>
<td>8.15 Protocol handler timing explained - case 3</td>
<td>82</td>
<td></td>
</tr>
<tr>
<td>8.16 Protocol handler timing explained - case 4</td>
<td>83</td>
<td></td>
</tr>
<tr>
<td>8.17 Simulated RTTs - case 1</td>
<td>84</td>
<td></td>
</tr>
<tr>
<td>8.18 Simulated RTTs - case 1, no interfering tasks</td>
<td>84</td>
<td></td>
</tr>
<tr>
<td>8.19 Simulated RTTs - case 2</td>
<td>85</td>
<td></td>
</tr>
<tr>
<td>8.20 Simulated RTTs - case 3</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>8.21 Simulated RTTs - case 4</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>8.22 Simulated RTTs with filtered IAC implementation - case 1</td>
<td>87</td>
<td></td>
</tr>
<tr>
<td>8.23 Simulated RTTs with filtered IAC implementation - case 2</td>
<td>88</td>
<td></td>
</tr>
</tbody>
</table>
### List of Tables

6.1 PID settings ........................................... 56
6.2 Performance measures comparing MMS and IAC protocols ....... 61
7.1 Performance measures for studying the impact of optimized RTTs .... 70
# Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABB</td>
<td>Asea Brown Boveri</td>
</tr>
<tr>
<td>ACK</td>
<td>Acknowledgement</td>
</tr>
<tr>
<td>CA</td>
<td>Congestion Avoidance</td>
</tr>
<tr>
<td>CAN</td>
<td>Controller Area Network</td>
</tr>
<tr>
<td>CNCP</td>
<td>Control Network Clock Protocol</td>
</tr>
<tr>
<td>CSMA/AMP</td>
<td>Carrier Sense Multiple Access with Arbitration on Message Priority</td>
</tr>
<tr>
<td>CSMA/CD</td>
<td>Carrier Sense Multiple Access with Collision Detection</td>
</tr>
<tr>
<td>cwnd</td>
<td>Congestion Window</td>
</tr>
<tr>
<td>DES</td>
<td>Discrete Event Simulation</td>
</tr>
<tr>
<td>FBD</td>
<td>Function Block Diagram</td>
</tr>
<tr>
<td>FDMA</td>
<td>Frequency Division Multiple Access</td>
</tr>
<tr>
<td>FIN</td>
<td>Final</td>
</tr>
<tr>
<td>FR</td>
<td>Fast Recovery</td>
</tr>
<tr>
<td>HMI</td>
<td>Human Machine Interface</td>
</tr>
<tr>
<td>IAC</td>
<td>Inter Application Communication</td>
</tr>
<tr>
<td>IAE</td>
<td>Integral Absolute Error</td>
</tr>
<tr>
<td>IL</td>
<td>Instruction List</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>ISE</td>
<td>Integral Squared Error</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>ITAE</td>
<td>Intergral Time Absolute Error</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>LD</td>
<td>Ladder Diagram</td>
</tr>
<tr>
<td>MMS</td>
<td>Manufacturing Message Specification</td>
</tr>
<tr>
<td>MSL</td>
<td>Maximum Segment Lifetime</td>
</tr>
<tr>
<td>MSS</td>
<td>Maximum Segment Size</td>
</tr>
<tr>
<td>OSI</td>
<td>Open Systems Interconnection</td>
</tr>
<tr>
<td>PID</td>
<td>Proportional Integral Derivative</td>
</tr>
<tr>
<td>PLC</td>
<td>Progammable Logic Controller</td>
</tr>
<tr>
<td>PPP</td>
<td>Point-to-Point Protocol</td>
</tr>
<tr>
<td>RNRP</td>
<td>Redundant Network Routing Protocol</td>
</tr>
<tr>
<td>RTO</td>
<td>Retransmission Time-Out</td>
</tr>
<tr>
<td>RTT</td>
<td>Round Trip Time</td>
</tr>
<tr>
<td>SEQ</td>
<td>Sequence Number</td>
</tr>
<tr>
<td>SFC</td>
<td>Sequential Function Chart</td>
</tr>
<tr>
<td>SIL</td>
<td>Safety Integrity Level</td>
</tr>
<tr>
<td>SNTP</td>
<td>Simple Network Time Protocol</td>
</tr>
<tr>
<td>SS</td>
<td>Slow Start</td>
</tr>
<tr>
<td>ssthresh</td>
<td>Slow Start Threshold</td>
</tr>
<tr>
<td>ST</td>
<td>Structured Text</td>
</tr>
<tr>
<td>SYN</td>
<td>Synchronised</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>TDMA</td>
<td>Time Division Multiple Access</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
</tbody>
</table>
1

Introduction

This master’s thesis covers simulation of the network structure of the ABB control system 800xA, and what improvements that could be implemented using the results. The thesis is collaboration between ABB and the Department of Automatic Control, Lund University, where the simulation tool, TrueTime, utilized in this project was developed. The work was performed at ABB Business Center, Malmö.

1.1 Background

Modern control systems are to an increasing degree distributed over networks. Examples include control systems in large factories, oil rigs, air planes and cloud based control systems. These kinds of distributed systems tend to be very complex, and the task of analyzing and predicting the outcome of such systems is far from trivial, not to mention how difficult it can be to develop the systems. Hence, an effort to simulate the systems can be worth a try, as the simulation greatly can simplify system implementation and analysis. If successful, the simulation also lets the user experiment with different settings much faster and easier than in reality. Also, it can help the user to understand the behaviour of the system and could for instance be used for discovering ways to optimize the system.

1.2 Goals

The goal of this thesis is to model and simulate the 800xA distributed control system developed at ABB. In order to speed up development, a simulation tool TrueTime has been used. This is a Matlab/Simulink based tool, developed at the Department of Automatic Control at LTH, and works as a framework for our implementation. TrueTime is designed to simulate real time systems in general but has the feature of modeling networks in the system, at least for low level protocols. Also, the integration with Matlab/Simulink makes TrueTime ideal for simulating control systems acting on (simulated) physical processes. Though, the question remained, could TrueTime be used, in the way described above, to simulate the 800xA distributed
control system?

This master’s thesis started as a study in the simulation of the 800xA system only, but as time passed it also became an investigation of how the simulation could be validated and put to use. This finally led the thesis in the direction of studying the so called round trip times, i.e. the time it takes to get an answer to a request, when a network node reads a value from another unit. Especially, the details of the sending and receiving mechanisms of the system were studied and the simulated results were compared to those obtained from measurements of a real system. Furthermore, the gained knowledge from the simulation was utilized in an attempt to optimize the packet path in the system, with the intention of shortening the round trip times; thus, improving system performance.

### 1.3 Individual Contributions

This master’s thesis has above all been focusing on implementation and most of the time has been spent on discussing and coming up with solutions. Thus, a majority of the work has been done together. However, Tommi’s previous experience from data communication and networks has proven very valuable, especially when implementing the simulated TCP protocol. Viktor has been in charge of the overall structure and object oriented design of the program. Also, he has been responsible for the graphics used in the report. Furthermore, both authors have been involved in all phases of the thesis, from implementation to writing the report.
1.4 Outline

An introduction to the ABB control system 800xA is provided, focusing on the components involved and the system’s application areas is presented in Chapter 2 - Overview of the 800xA System. In Chapter 3 - Network Structure in 800xA, the overall network structure of 800xA is described, and important protocols on the network, transport and application layer are presented and explained thoroughly. Furthermore, Chapter 4 - Simulation of Real Time Network Distributed Systems covers the principles of network simulators. Also, the main simulation tool utilized in this thesis, TrueTime, is described in depth, and other alternatives are presented as well. Chapter 5 - Modeling and Simulation of 800xA describes implementation details of the simulation and modeling. This chapter follows Chapter 3 to a great extent, mapping theory to implementation. The use of the 800xA simulator is demonstrated in Chapter 6 - Results of TrueTime Simulations. This chapter also shows different scenarios that have been investigated. The current implementation of handling IAC traffic in the automation system is presented in Chapter 7 - Optimization of IAC traffic, and an optimized alternative is proposed with simulation results as background. In Chapter 8 - Results and Validation of IAC Optimization, the results of the optimized IAC implementation are discussed and compared to the simulations. The results of this project are evaluated and conclusions are drawn in Chapter 9 - Conclusions. Also, possible future work is proposed.
Overview of the 800xA System

The ABB 800xA system is a highly customizable automation system with focus on integration and safety. 800xA consists of several hardware and software components that together form the system. Covering this vast automation system completely is not possible, however, this chapter gives a brief overview and particularly presents the parts important to the thesis.

2.1 Usage

The 800xA system is used all over the world in larger facilities such as:

- factories
- dairy industry
- paper industry
- oil rigs
- nuclear power plants
- water dams

In general the 800xA system is used in process industry, especially in safety critical applications. To ensure the users a safe system, 800xA is SIL (Safety Integrity Level) classified at level 3, which means that the system has passed certain tests and that the system has been developed in a safe manner. For instance, if something goes wrong in a nuclear power plant, it is absolutely crucial that the automation system works properly.
In process industry, one is often controlling processes of a slow nature, and it is often more important to stay close to reference values during disturbances than being able to quickly follow a reference change. The actual physical process controlled could be almost anything, examples include:

- fluid levels in tanks
- heat in ovens
- pressure in gas/fluid containers
- flow of gases/fluids
- positions/velocities/forces in mechanical systems
- voltage
- biological processes
- chemical processes
- speed of conveyor belts

This has been taking into consideration in the simulations done in the thesis.

### 2.2 Hardware

The most important hardware units that make up the 800xA system are the controllers and I/O modules.

**AC 800M controllers**

The core of the 800xA system are the controllers, that is, the PLC processing units that receive measurement signals and calculate what to do in the system. The controllers utilized in this master’s thesis are the AC 800M controllers, more specifically with main unit type PM861A.

There are eight different main CPU models in the AC 800M family, with varying specifications and features including SIL-rating and redundancy support. The units’ clock frequencies (24-450 MHz) and RAM (8-256 MB) are small in comparison with a standard PC. The main unit used in this master’s thesis, the PM861A, has a clock frequency of 48 MHz, 16 MB RAM, a flash memory slot and CPU redundancy support [ABB, 2013c].

SIL-3 certification is obtained by the unit PM865A, and is by ABB called a High Integrity (HI) controller. It can be extended by a safety module SM811 that
Chapter 2. Overview of the 800xA System

runs all calculations in parallel with the main unit. This way the precision and reliability of the controller are greatly enhanced. This configuration of the units enables for both safety critical and standard process automation applications. It is also possible to attach and interconnect other units to all AC 800M main units, including I/O modules and fieldbuses, which are described later on in this section. See Figure 2.1 for a small High Integrity AC 800M set up.

Figure 2.1  An AC 800M controller set up, here with a main unit PM865A (second leftmost), with attached safety module (leftmost) and two I/O modules (to the right).

In an AC 800M controller a lot of different threads are running. Later in the report a closer look will be taken at some of them. The threads that are of particular interest are the ones handling network traffic and the ones that takes care of packets. The other threads will in the simulation be seen as interfering threads in the controller [System 800xA 5.1, Product Catalog (FP4 included), p. 5].

I/O Modules

The I/O modules of 800xA can both act as extensions to the AC 800M controllers, or be part of a distributed process I/O system that communicates with parent controllers over fieldbuses. As simple extensions, the I/O modules take care of the interface between AC 800M main unit and the input and output signals. However, on their own they also form the I/O systems S800 I/O and S900 I/O. These systems permit installation in the field, close to the sensors and actuators. This reduces the cost of cabling and is also useful for installations in hazardous areas [ABB, 2013d].

2.3 Software

The most important component in the 800xA automation system is perhaps the software, which enables the user to both create the control logic executed in the hardware and to supervise the processes.
2.3 Software

Process Portal A
The main tool that handles and configures settings related to the control applications is the Process Portal A (PPA). In this tool, the user can specify everything from user profiles to included hardware/software libraries, and manage the projects created in the Control Builder tool.

Control Builder
The Control Builder is a software tool developed for building applications in the control system. It is possible to create applications that run on the controllers, defining the logic and tasks.

The controller applications are created in projects, where all necessary components are available for configuration. See Figure 2.3 for an overview of a typical project. The main included parts are:

- libraries
- applications
- controllers

The Libraries concern both the software and the hardware. In the former case, the software libraries define e.g. data types, function blocks and control module types. When a software library is included in a project, these types become available in the applications. The hardware libraries define code closer to the hardware and the core functionality of the controller. An example is the protocol handler, part of BasicHWLib, which defines the controller communication handling.

The Applications contain the actual controller logic and are connected to tasks that execute on the controllers. The code is written in programs or diagrams. In this master’s thesis, programs have been used exclusively. In the programs it is possible to declare and set variables, communication variables and function blocks. The code can be written in 5 different languages, defined by the IEC-61131 standard [ABB, 2006, p. 158]:

- Instruction List (IL)
- Ladder Diagram (LD)
- Function Block Diagram (FBD)
- Sequential Function Chart (SFC)
- Structured Text (ST)
Chapter 2. Overview of the 800xA System

IL resembles assembly code, whereas LD, FBD and SFC are graphical languages. ST is a function oriented text based language, and is the only language utilized in the simple programs created in this master’s thesis. The Controllers define the hardware that the applications run on. It is possible to configure the network settings of the units, including the IP address of the controllers, and other hardware related settings as well. Tasks are created in each controller, and assigned e.g. priorities and period times. These tasks are then connected to the programs and applications, to define how they should run in the controller.

Operator Stations

The 800xA system also includes software for operator stations, which are complete HMI (Human Machine Interface) environments where plant operators can get an overview of the system status. The software provides the possibility to control the system remotely and take actions if needed. See Figure 2.2 for an example of a typical 800xA operator station.

**Figure 2.2** A typical 800xA operator station.
Figure 2.3  A project in the Control Builder software.
Network Structure in 800xA

The network in an 800xA system can be seen as four different parts: the plant network, the client/server network, the controller network and the fieldbus/field network. This master’s thesis focuses on the communication in the controller network, but to understand the system, at least some basic knowledge about the other networks is needed.

In the plant network, so called thin clients can connect to the system and are typically used for monitoring the system. The thin clients cannot access the rest of the network directly, they are isolated through an isolation device as seen in Figure 3.1. From this isolation device the thin clients can get data from the system.

The client/server network is used for communication between workplaces (clients)/server and server/server. The servers are called connectivity servers and handle the separation and communication between the different networks. To clarify, the client/server networks can communicate with the control network through the connectivity server(s). The workplaces, or clients, are used by the plant operators [ABB, 2012, p.22-24].

In the control network, all communication between the controllers is made. Also, the communication between controllers and connectivity servers is done in the control network. It is possible to include workplaces in the control network but it is almost never done in practice, since this can cause performance issues. However, in some rare cases when using a small system, it can be sufficient.

The fieldbus/field network is used for communication with modules in the plant, e.g. I/O units. These modules can communicate either directly with a controller or via a connectivity server [ABB, 2012, p.22-24]. 800xA supports third party field buses such as PROFIBUS, PROFINET, FOUNDATION Fieldbus, HART and MODBUS TCP [ABB, 2013b].
Both the client/server network and the control network have the feature of using redundant networks, i.e. one primary network and one secondary for backup. This is used when extra robustness is wanted for the system. The network redundancy is managed by the RNRp (Redundant Network Routing Protocol) protocol [ABB, 2012, p.28].
Chapter 3. Network Structure in 800xA

3.1 Protocols used in 800xA

The AC 800M controllers and connectivity servers in the network can communicate using the following protocols:

- **Application Layer Protocols**
  - MMS - Manufacturing Message Specification
  - IAC - Inter Application Communication
  - SNTP - Simple Network Time Protocol
  - CNCP - Control Network Clock Protocol

- **Transport Layer Protocols**
  - TCP - Transmission Control Protocol
  - UDP - User Datagram Protocol

- **Network Layer Protocols**
  - RNRP - Redundant Network Routing Protocol

- **Link Layer Protocols**
  - Switched Ethernet
  - PPP - Point-to-Point Protocol

Here, the different layers are the ones defined in the standard OSI (Open Systems Interconnection) model. That is, a model that divides a network into seven different layers: Application, Presentation, Session, Transport, Network, Data Link and Physical, as seen to the left in Figure 3.2. Read more about the OSI model at [The OSI Model’s Seven Layers Defined and Functions Explained 2014].

All protocols relevant to this project are described in the following sections. The CNCP and SNTP protocols are used for time synchronization in the system. Local clocks are not perfect and drift if nothing is done to prevent it. In this master’s thesis, these protocols together with the link layer protocol PPP have not been investigated [ABB, 2012, p.22-24].

3.2 Switched Ethernet

Ethernet, or CSMA/CD, is in this thesis defined as a standard for data communication on the data link layer over LANs. Ethernet is also associated with the Ethernet cable, which is a standard physical media for data communication. This, however,
3.2 Switched Ethernet

Switched Ethernet is an upgraded version of the CSMA/CD, that avoids collisions on the network by dividing a network of \( N \) clients into \( N \) collision domains by using a switch, see Figure 3.3. In this way, collisions due to interfering traffic between clients are eliminated, as long as the switch is able to multiplex the different channels [Forouzan, 2013, p. 363-375].

![Figure 3.2 The TCP/IP stack's correspondence to the OSI model.](image)
Chapter 3. Network Structure in 800xA

Figure 3.3 The Ethernet switch separates clients into different collision domains to avoid collisions. Inspired by [Forouzan, 2013, p. 375].

3.3 RNRP - Redundant Network Routing Protocol

In a controller network each node can be connected with redundant networks, i.e. to one primary network and one secondary backup network. To handle this, ABB has created a network layer protocol called RNRP (Redundant Network Routing Protocol). This takes care of re-routing the traffic from one network to the other if some kind of problem occurs, e.g. that a link is broken. To be able to do this each controller in the network must build its own routing table, containing data about which other nodes that exist in the same network area. This is done by letting all nodes broadcast so called hello messages. These messages simply say that the sending node is alive in the network. The hello messages are broadcast both on the primary and secondary network. All nodes listen for the hello messages and in this way they can get a picture of how the network looks like, and keep track of how long time it has gone since a hello message was received from a certain node.

The hello messages are broadcast over UDP. Because of this, one might argue that it is not completely true that RNRP is just a network layer protocol. However, all routing protocols must communicate somehow and as RNRP works in the background in the way it does, it should be considered as a network layer protocol, see Figure 3.2.

Besides from handling network redundancy, RNRP also lets nodes route messages between different network areas. It also detects if some error has occurred on the network and can send this information to applications [ABB, 2012, p. 37-38].
3.3 RNRP - Redundant Network Routing Protocol

Figure 3.4 RNRP network topology, inspired by [ABB, 2012, p. 40].

If a network error occurs between two nodes in a network area, RNRP can only save the connection if at least one of the networks, primary or secondary, is working for both nodes. In Figure 3.4 five nodes exist in a network area. In this example, it does not matter if the nodes are controllers or workstations, as long as they are network nodes and use RNRP. As seen in Figure 3.4 node A and C have lost their connection to the primary network and node E has lost the connection to the secondary network. In this case, node A and C can communicate with all nodes except E. The same thing applies to E, i.e. it can communicate with both B and D but not with A and C. Node B and D have a fully redundant connection [ABB, 2012, p. 40].

In the case of a network error, RNRP separates between two kinds of errors: link down and node down. Node down is when a controller is not responding at all, e.g. if it crashes or loses power. This is detected by other controllers by noticing that hello messages are not sent from that controller. As default, three missed hello messages are interpreted as a node down event. Link down could for example be if an Ethernet cable is physically removed or cut off. A link down can be sensed by a network node and in case of a link down, RNRP can instantly redirect traffic, that is, without waiting for three (as default) lost hello messages.

There is a bit more to know about RNRP, but this is out of scope for this report. For
more information see [ABB, 2012].

3.4 UDP - User Datagram Protocol

UDP is a connectionless and simple transport protocol. It is, along with TCP, used in the transport layer of the ABB 800xA network structure. UDP has no features such as flow, error or congestion control, however, its header does include a checksum as seen in Figure 3.5. The resending of the lost or corrupted packets is not performed by UDP, but has to be taken care of by protocols at a higher level (such as IAC). As a result, there are no acknowledgements sent and information might be lost or arrive out of order, at least at the transport layer.

UDP is a useful alternative to TCP when it is not disastrous if some packets arrive out of order or do not reach the destination at all. Under these conditions, UDP is preferable since it does not require connection establishment and termination packets to be sent and thus giving a more efficient transfer. Another case where UDP could be preferred over TCP is when the user wants to be able to configure the resend time-outs. This is not possible in TCP, but since UDP does not resend packets at all, the resend scheme can be determined by a higher level protocol.

![UDP header format](image)

**Figure 3.5** UDP header format. Inspired by [Forouzan, 2013, p. 738].

3.5 TCP - Transmission Control Protocol

TCP is a connection-oriented transport protocol [Forouzan, 2013, p. 999]. It is, along with UDP, used in the transport layer of the ABB 800xA network structure. TCP is a reliable protocol with features such as flow, error and congestion control. It is responsible for process-to-process communication and multiplexing is
3.5 TCP - Transmission Control Protocol

achieved using port numbers. The data is sent in byte-oriented segments ("stream of bytes") between the sending and receiving buffers. The TCP header segment format is shown in Figure 3.6.

Figure 3.6 TCP header format. Inspired by [Forouzan, 2013, p. 748].

Connection Establishment and Termination
The TCP connection allows for transmission of data in both directions, which means that the communication must be initialized (and terminated) by both the server and the client. Figure 3.7 shows events 1-7 described below.

Connection Establishment using three-way-handshaking

1. The client performs an active open by sending the first segment with a SYN flag set, requesting a connection from client to server. This consumes one sequence number.

2. The server responds with a SYN+ACK segment, which both requests a connection from server to client as well as acknowledges the client’s SYN request. This consumes one sequence number.

3. The client responds with an ACK segment, acknowledging the server’s SYN request. The connection is now open in both directions, and this segment does not consume a sequence number.
Figure 3.7  Connection establishment and termination for a common TCP scenario. Inspired by [Forouzan, 2013, p. 760].

**Connection Termination using four-way-handshaking**

4. The client performs an active close by sending a segment with a FIN flag set,
3.5 TCP - Transmission Control Protocol

requesting to terminate the connection from client to server.

5. The server responds with an ACK segment, acknowledging the client’s FIN request. The connection is still open from server to client, and data transfer is still possible in that direction.

6. The server sends a FIN segment, requesting to terminate the connection from server to client.

7. The client responds with an ACK segment, acknowledging the server’s FIN request. In order to prevent delayed packets being accepted by a later connection, the client enters a TIME-WAIT state for 2 Maximum Segment Lifetimes (MSL). After the 2MSL time out and the ACK is received by the server, the connection is terminated in both directions.

Flow Control
Flow control is achieved by using send and receive windows, situated at both the client and server side of the communication, adding up to a total of 4 windows. The send window, as seen in Figure 3.8, defines which bytes that can be sent. When acknowledgements for the outstanding bytes arrive, the send window slides to the right and allows for new bytes to be sent. The receive window works in a corresponding way; the window slides as newly received bytes arrive. The size of the receive window is advertised by the receiver and affects the size of the sender’s send window according to:

$$\text{window}_{send} = \min (\text{window}_{congestion}, \text{window}_{receive})$$  \hspace{1cm} (3.1)

![Flow Control Diagram](image)

**Figure 3.8** The TCP send window. Inspired by [Forouzan, 2013, p. 761].

Error Control
The error control feature of TCP guarantees that all segments arrive at the receiving process uncorrupted and in order. To achieve this, TCP includes mechanisms for
detecting and resending corrupted and/or lost segments, and storing of out-of-order segments until the missing segments arrive. This is done by making use of check-sums, acknowledgements and time-outs.

The key component of the error control is the retransmission of segments. This is performed when any of the two events occur:

- **Retransmission Time Out (RTO)**
  When the retransmission time out for the connection expires, TCP resends the bytes in the entire send buffer.

- **Three Duplicate ACKs**
  When three duplicate ACKs arrive, TCP resends the bytes in the entire send buffer. This feature is called *fast retransmission*, and helps to speed up the retransmission process.

To clarify the error control procedure, a simple example of how the TCP protocol handles lost messages is given. Consider the following case, a network node, called A, wants to send three bytes of data to another node B. To simplify the packet sequence numbers, each segment only carries the (unrealistic) value of one byte. This means that the three bytes of data will be sent in three different segments, with the sequence numbers 1, 2 and 3. In a real case scenario, a packet typically carries about 1500 bytes, hence a segment will cover an interval of sequence numbers. With this being said, the following list of events in combination with Figure 3.9 will explain the error control feature of TCP.

1. The three segments are sent from A. Segment 1 and 2 get lost on the way.
2. Segment 3 arrives out of order at B and is stored in a temporary receive buffer. B sends ACK 1 to indicate that A should resend from segment 1.
3. A retransmission time out occurs, and the three segments are resent. Since A does not know that segment 3 has arrived at B, segment 3 is also sent again. This time, segment 2 is lost.
4. B receives segment 1 and 3. Since segment 1 is in order, it can be delivered to the application layer. ACK 2 is sent to indicate that A should resend from segment 2.
5. Two ACK 2 arrive at B. Notice that this means that the duplicate ACKs counter is set to one. A new retransmission time out occurs, and since A knows that segment 1 has arrived, only segment 2 and 3 are resent. Segment 3 gets lost on the way.
6. Segment 2 arrives, hence both segment 2 and 3 are in order and can be delivered to the application layer. Notice that segment 3 did not arrive but was
stored previously. ACK 4 is sent to indicate that A should send from segment 4.

7. Only three segments should be sent, hence A is finished sending.

Figure 3.9 An example of how TCP error control handles lost packages.
Chapter 3. Network Structure in 800xA

Congestion Control

Congestion control is achieved in TCP by making use of a congestion window at sender site, \( cwnd \), which when necessary limits the send window. The most common TCP implementation today, Reno TCP, uses a congestion control scheme as seen in Figure 3.10.

The TCP connection begins in the slow start (SS) phase with \( cwnd = 1 \) MSS (Maximum Segment Size). In this example the slow start threshold (\( ssthresh \)) is set to 16 MSS. When an ACK arrives in the SS phase, \( cwnd \) gets multiplied by 2 which leads to an exponential increase. As seen in the figure at \( t = 3 \) RTTs (Round Trip Times), this increase is interrupted when a time-out occurs, resetting \( ssthresh \) to \( cwnd / 2 \) and then setting \( cwnd \) to 1. The SS phase then starts again before reaching \( ssthresh \) at \( t = 5 \) RTTs.

This causes TCP to reach the congestion avoidance (CA) state. In this state \( cwnd \) is updated according to \( cwnd = cwnd + 1 \) when an ACK arrives, leading to an additive increase. This goes on until either a time-out occurs or three duplicate ACKs arrive. The former case is not shown in the figure, however that would force the connection back to the SS phase again.

At \( t = 13 \) RTTs three duplicate ACKs arrive, which causes the connection to enter the fast recovery (FR) state. Here the \( ssthresh \) value is set according to \( ssthresh = cwnd / 2 \), before updating \( cwnd \) to \( ssthresh + 3 \). In the FR state, the increase of \( cwnd \) behaves as in SS until a new non-duplicate ACK arrives. This occurs at \( t = 15 \) RTTs, and causes the connection to re-enter the CA state, but this time with \( cwnd \) set to the current value of \( ssthresh \). In this example, the connection stays in the CA state until the figure ends at \( t = 20 \) RTTs. However, it is obviously still possible for the TCP connection to switch between the SS, CA and FR states when the events as described above occur.
3.6 IAC - Inter Application Communication

IAC is an application layer protocol, used for reading and writing variables between controllers in the 800xA network. This is done over UDP, as opposed to the MMS protocol which utilizes TCP. This makes IAC an alternative that is simpler to use than MMS, since no connection has to be established before the variable exchange can begin. In order to initialize a variable communication over IAC, it is only necessary to declare the direction of the variables in each controller; in (i.e. read), out (i.e. write) or in/out (i.e. read/write).

The communication models for the client and server side can be seen in Figure 3.11 and Figure 3.12 respectively. The model is based on three main parts: the communication protocol, a protocol handler thread and a scheduler thread. The communication protocol defines the format of the sent messages, and allows for multiple variables to be sent in each message. The protocol handler is responsible for the communication between the client and the server, and implements the functions necessary for creating and receiving IAC messages. The scheduler thread runs...
asynchronously to the protocol handler thread and is responsible for storing and fetching the variable values. This is done by performing cyclic copy-ins to 1131 (short for IEC-61131) variable memory on the client side and copy-outs from the variable memory on the server side. When the server receives a request for a variable value, the protocol handler thread performs a look-up to fetch the latest stored value (copy-out) [ABB, 2013a, p. 24-26].

**Figure 3.11** The interaction of the client and the IAC protocol handler. Inspired by [ABB, 2013a, p. 26].

Since UDP has no resending mechanism, IAC itself has to be responsible for the error control. A request is resent by IAC if no response is received within the timeout interval, normally set to the period of which the IAC requests are sent. The resend time out is, however, never by default set to a higher value than 500 ms. IAC does indeed guarantee that a request is met by a response, but not that they arrive in order [ABB, 2010, p.49-56].
3.7 MMS - Manufacturing Message Specification

The MMS protocol is built upon TCP and is used for exchanging variable data. It is implemented in the application layer of the OSI model (see Figure 3.2).

Before doing any variable value exchanges, the MMS protocol has to create a connection, both in the TCP connection that it is built upon and at the MMS layer. When getting a variable value over MMS a client first has to send a request to a server which holds the variable to be read. Then the server sends back the value of the variable, see Figure 3.13. In general, an MMS server can periodically send data without requests but this is not used in the 800xA system. Thus, a server can only send data if it has received a request for it first. Note that in this context, both client and server refer to a role that an AC 800M controller takes. That is, a controller can be both a client and a server depending on if it sends or receives a request. The server in this case should not be confused with e.g. a connectivity server [ABB, 2010, p. 31-48, 165].

![Exchange of variable value with MMS](Sisco, 1995, p. 10).

The MMS layer also limits the number of request on a TCP connection to three.
In this master’s thesis, the main tool used is TrueTime, a Matlab/Simulink based simulator for real-time and networked systems [Cervin et al., 2003]. TrueTime has been developed at the Department of Automatic Control at Lund University and supports co-simulation of controller task execution in real-time kernels, network transmissions, and continuous plant dynamics.

### 4.1 Simulation Tool Alternatives

As mentioned, TrueTime has been chosen as a tool for the simulation in this thesis project, but there are several alternatives available. Below, a list of different network simulators is presented. Also, some of the most commonly used network simulators are described in corresponding subsections later in this chapter [Pan, 2008]. Notice that these are network simulators, not emulators. The difference is that a simulator catches the overall behaviour while an emulator mimics the reality "exactly". Since we are interested in looking at the overall behaviour, a simulator is preferred.

- NetSim - [tetcos, 2014], [Waupertitsch et al., 2006, p. 2135]
- ns-1 - [Floyd, 1999], [*ns-3 Project Goals* 2006]
- ns-2 - [Information Science Institution, 2011], [*ns-3 Project Goals* 2006]
- ns-3 - [*ns-3 Homepage* 2014], [*The ns-3 Manual* 2014], [*ns-3 Project Goals* 2006]
- OPNET - [*Riverbed Homepage*]
Most network simulators are based on something called discrete event simulation (DES). This basically means that a number of events, and how the system reacts to these, are defined. In this way, very complex system can be modeled. Besides from network simulations, discrete event simulation is often used in stress testing, finance, manufacturing and health care [Rouse, 2012]. Another method to simulate networks is by a Markov chain simulation. This is usually faster, but less detailed and flexible than discrete event simulation [Gkantsidis et al., 2003].

**ns-2**

The ns-2 network simulator was made in collaboration between UC Berkeley, LBL, USC/ISI, and Xerox PARC, called the VINT project. It is an open source tool and it is supported by DARPA. The simulator is written in C++ and it uses OTcl for commands and configuration. ns-2 is a newer version of ns-1 which is based on the REAL network simulator [Information Science Institution, 2011, p. 1].

**ns-3**

The ns-3 network simulator is open source software and is widely used in research. It is programmed in C++ and was developed as a replacement for ns-2. Simulations made with ns-3 can be written in C++ and/or Python code. For more information, see [ns-3 Homepage 2014], [The ns-3 Manual 2014] and [ns-3 Project Goals 2006].

**OPNET**

OPNET is a commercial network simulator made by Riverbed. It is, according to Riverbed’s own website, the fastest discrete event based network simulator on the market. It has an open interface so that integration with other tools and simulators is easy. It has support for several protocols and devices and has rich visualization of simulation results. The tool is made in C/C++ [Riverbed Homepage].

**OMNET++**

OMNET++ is actually not just a network simulator, but a discrete event simulator in general. However, it is commonly used as just a network simulator. It consists of components in C++ but it is used with a high level language called NED. It also has some GUI support and an IDE based on the Eclipse platform [Omnet++ Homepage].
Why TrueTime Was Chosen

Even though there are many other good alternative tools for the simulation and TrueTime has some drawbacks, we consider TrueTime to best fit this project. This is especially thanks to the integration of TrueTime with Simulink which makes it very easy to simulate physical processes in combination with the network and control system. TrueTime can also be used to model the behaviour of the AC 800M controller with the right amount of details. TrueTime lacks built in support for simulation of protocols above the link layer, however, it has been reasonably "easy" to implement this. Another desirable feature is that TrueTime is open source software, thus, access to the entire source code is provided.

4.2 TrueTime

TrueTime is a simulation tool based on Matlab/Simulink. It is developed at the Department of Automatic Control at LTH in Lund and is made for simulating real-time control systems.

Structure of a Simulation in TrueTime

A simulation with TrueTime is constructed with Simulink blocks from the TrueTime library, see Figure 4.1. Right now, the available blocks are:

- TrueTime Kernel
- TrueTime Network
- ttGetMsg
- ttSendMsg
- TrueTime Battery
- TrueTime Wireless Network
- TrueTime Ultrasound Network

In this master’s thesis the only blocks used are the kernel and network blocks. The kernel block represents a computing node that runs some user defined code, see Section 4.2, and the network block corresponds to the physical and data link layer of a (wired) network. TrueTime also gives the user the possibility to run the simulation with either just Simulink blocks, or in a combination with code written in C++ or MATLAB m-files. These blocks and files can then be used together with standard Simulink blocks, e.g. a transfer function block representing a physical process. Then the TrueTime kernel and network blocks can act as a distributed control system that controls the physical process [Cervin et al., 2010, p. 9].
The Network Block

In the TrueTime network block, a local area network is modelled. The block represents the physical and data link layer of the OSI-model (see Figure 3.2). In the block settings it is required to specify:

- Network Type
- Network Number
- Number of Nodes
- Data Rate
- Minimum Frame Size
- Loss Probability
Chapter 4. Simulation of Real Time Network Distributed Systems

- Initial Seed
- Other Network Type Specific Parameters

The network type sets what kind of data link layer protocol to be used. The current network types supported are:

- CSMA/CD (Ethernet)
- CSMA/AMP (CAN)
- Round Robin
- FDMA
- TDMA
- Switched Ethernet
- FlexRay
- PROFINET
- NCM

The network number is just an ID for the network and the number of nodes is, as the name indicates, how many nodes (kernel blocks) that are connected to the network. Data rate sets the transfer speed of the network in bits/s and minimum frame size is the least amount of bits one packet is allowed to have. Loss probability is the probability of losing a packet when sending something over the network. This parameter and the data rate are what model the physical layer of the network. There is no possibility to specify a certain physical media, but it is not important to simulate such details in this project anyway.

The Network block also outputs a network schedule. The schedule shows which network nodes that are currently active. An example of a network schedule is seen in Figure 4.2.
Figure 4.2 A schedule over a simulated network in TrueTime. The different colours indicate different nodes in the network. When the schedule is at an integer value the node of that number is idle. If the schedule shows a value of $X + 0.5$ where $X$ is an integer, that node is currently using the network.

In addition to the network block there is a wireless network block. With this one can, as the name suggests, model a wireless network. There are a couple of parameters for modeling typical attributes for a wireless network, such as path loss function and signal strength. It also features the wireless network standards 802.11b WLAN and 802.15.4 ZigBee [Cervin et al., 2010, p. 16-26].

The Kernel Block

In TrueTime, the kernel block represents a real time kernel running some specified tasks (see Section 4.2 for information on tasks). What tasks the kernel should perform is set in the so called init function, which is written in a MATLAB m-file or in C++. There are also a few parameters to set for each kernel block:

- init Function (as mentioned above)
- init Function Argument
• Number of Analog Inputs and Outputs
• Number of External Triggers
• (Network and) Node Numbers(s)
• Local Clock Offset and Drift

With the \texttt{init} function argument it is possible to send data to the \texttt{init} function. The number of analog inputs and output sets the amount of channels used to interact with other Simulink blocks. External triggers can be used for programming in Simulink rather than in the \texttt{init} functions. The network and node numbers tells TrueTime what network(s) that kernel belongs to and what number (basically an IP) the kernel has in that network. When using multiple networks for one kernel this should be an $n \times 2$ matrix where $n$ is the number of networks connected to the kernel. Each row then contains a network number and a node number. Local clock offset and drift sets the behavior of the clock used in the kernel.

The kernel block can output, in addition to the analog output(s), a schedule of the running tasks and the kernels power consumption [Cervin et al., 2010, p. 15].

**User Defined Code**

As mentioned earlier, TrueTime provides the possibility to control what each kernel block should run in the simulation by specifying a user defined code written in either a MATLAB m-file or in C++. The latter alternative is compiled with the Matlab MEX interface linked to an external C/C++ compiler (e.g. gcc or Visual Studio C++).

In the user defined code, tasks can be set to a kernel. These tasks can be performed both periodically and on events. In the case of event driven task, a task could e.g. be triggered by an incoming packet on a network. In the code it is also possible to specify the priorities and scheduling of tasks.

**TrueTime Core**

This section describes the internal underlying machinery of TrueTime, i.e. how the simulations are executed in the kernels.

**The RTSys Class** The main TrueTime core class that handles the simulations in a kernel is called RTSys. When the S-function corresponding to a specific kernel is initialized, i.e. the so called \texttt{init} function, an instance of RTSys is created and stored in the UserData field of the kernel block between simulation steps. It contains, among others, the attributes seen in Figure 4.3. The RTSys instance handles the actions that are executed in \texttt{runKernel()}, which is the main method where the tasks are run.
4.2 TrueTime

**Figure 4.3** An overview of the RTSys class. The class responsible for running the simulation of the TrueTime kernels.

**Task Execution** TrueTime kernel tasks are handled by the readyQ and timeQ member variables in the RTSys class. Both queues are sorted linked lists and contain the tasks, as well as the timers, sorted in release and expiry time priorities. The readyQ is used for the tasks and timers that are ready for execution and the purpose of the timeQ is to keep track of the tasks and timers that are to be released [Cervin et al., 2010, p. 36-41].

The tasks in TrueTime are divided into segments. When a task is run, the task code will be called repeatedly with an increasing segment number, starting with segment one. It is not until the function returns FINISHED, i.e. -1, that the task code stops getting called. Due to this behaviour, a task code function typically consists of a switch statement:
double taskCodeFunction(int seg, void* data)
{
    switch (seg) {
    case 1:
        // Do something in the first "round"
        return someExecTime;
    case 2:
        // Do something in the second "round"
        return someMoreExecTime;
    default:
        return FINISHED;
    }
}

The return value of the code function is of type double and represents the task execution time. Note that this is not the actual execution time required by Matlab to execute the simulations, rather it is an estimation provided by the user to represent the actions taking place. The void pointer data is used for passing variables to the code function. Since it is a void pointer, anything can be passed, including arrays and structs to pass multiple values.

A task is always provided with a function pointer, which points to some user defined code that specifies the actions that task should take. A simple pseudo-code example of adding a function pointer to a task is shown below:

double codeFunctionToBeAttached(int seg, void* data)
{
    mexPrintf("This task is running!");
    return execTime;
}

    ttCreatePeriodicTask("task_name", startTime, period,
                      codeFunctionToBeAttached);

Summary
TrueTime is a powerful tool for simulation of control system distributed over networks. It provides the user a good base to create highly customizable simulations. However, due to its openness it requires a lot of knowledge from the user, both about the system to be simulated and about TrueTime.
Modeling and Simulation of 800xA

This chapter describes the implementation of the 800xA network structure in TrueTime, which mainly consists of adding the features of higher level protocols such as TCP and RNRP. In addition to the network protocols, the main threads in the controllers are also considered in order to model the behaviour of the controllers as accurate as possible.

5.1 Helper File

In order to make it easier to create different simulation setups, a helper file has been created to encapsulate the behaviour of general 800xA components. For example, the behaviour of how to handle a TCP connection is put in the helper file, while the init script of a kernel contains the connections it needs to use. In other words, each kernel defines its behaviour in the init script, but uses the predefined utilities of the helper file. The helper file could thus be viewed as an extension to TrueTime with the features of higher level protocols and the AC 800M controller.

Worth mentioning is that the helper file has been made so that not too much flexibility is lost. It is still easy to e.g. write custom PID code for a single kernel.

The helper file has a rather complicated class structure. This can be seen in Figure 5.1. Notice that this is just an overview, more detailed diagrams and explanations are found in corresponding sections.

5.2 Implementation in TrueTime

TrueTime has built in support for simulation of the physical and data link layers in networks. For simulation of communication with higher layer protocols such as
TCP and MMS, a file `protocol_helper.cpp` has been implemented. To simulate the internal behaviour of the controllers, tasks are utilized to model the most important threads.

**Kernel Setup and `initVital()` Function**

When using the helper file some parts of the kernel setup are vital for the other functions to work properly. Some of these essential commands, the ones that every kernel must run, have been collected in a function called `initVital()`. This function is responsible for defining events, starting tasks and network handlers for dual networks. A full list of its responsibilities is shown below:

- Creating Internal Mailboxes:
  - `tNet0ActionType`
  - `tNet0`
  - `tNet0Down`
  - `iacSendDownSignalMail`
5.2 Implementation in TrueTime

- Defining Events:
  - TransferState
  - MmsOk
  - UpAgain

- Creating Tasks:
  - Periodic:
    * tNet0
    * main
    * scdt
    * hello_task
    * check_task
    * interference
  - Aperiodic:
    * empty_task

- Creating Time-Out Handlers
  - timeOutHandler
  - timeOutHandlerIac

- Two Network Handlers, One For Each Subnet

The mailboxes are used for inter task communication. In order for tasks to be able to react to events, the events must first be defined. This is why there are events being defined in the `initVital()` function.

Some of the tasks created are doing some important work for the behaviour of the kernel, such as the `hello_task`. Others are created as dummy tasks that exist just for putting a load on the CPU of the kernel. Also, to mirror the behaviour of the AC 800M TCP/IP stack, all network traffic has to go through the `tNet0` thread, except when using an IAC filter. Read more about the IAC filter in Chapter 7.

The time-out handlers set the behaviour of what happens when a task does not finish before its deadline. The network handlers make sure an event is triggered when a packet arrives over that network.
5.3 Simulating RNRP with TrueTime

In this section, the implementation of the routing protocol RNRP (see Section 3.3) in TrueTime is described. The following aspects of the protocol have been considered:

- Hello messages
- Creating routing tables
- Routing using table lookup
- Network failure redirection

The implementation of keep-alive hello messages is achieved simply by sending broadcast messages from each alive kernel, using the TrueTime predefined function `ttSendMsg()` (see [Cervin et al., 2010, p.95]). The messages are sent periodically with a default 1 s period by making use of a periodic task created by the predefined function `ttCreatePeriodicTask()` (see [Cervin et al., 2010, p.65]). Since every kernel has to send these hello messages, this is part of the method `initVital()` (see Section 5.2). The messages consist of static pointers to instances of HelloPackets, a class that inherits from the base class Packet. The HelloPacket class is identical to Packet, apart from that it overrides the `getType()` method to signal that it is part of a hello message. The member variables of the Packet class can be seen in Figure 5.2.
5.3 Simulating RNRP with TrueTime

Figure 5.2 Overview of the class structure used for sending packets in protocol_helper.cpp

- Packet
  - +size: int
  - +receiver: string
  - +kernelData: KernelData
  - «virtual»
  - +getType(): void

- Data
  - «virtual»
  - +handle(): void

- HelloPacket
  - +getType(): int

- lacPacket
  - +sourcePort: int
  - +destinationPort: int
  - +data: lacData*
  - +msgNbr: int
  - +timerName: char[MAXCHAR]
  - +isResolved: bool
  - +getType(): int

- TcpPacket
  - +sourcePort: int
  - +destinationPort: int
  - +seqNbr: int
  - +FIN: int
  - +flag: int
  - +timerName: char*
  - +data: Data*
  - +ackNbr: int
  - +window: int
  - +isSent: bool
  - +getType(): int

- MmsData
  - +name: string
  - -isRequest: bool
  - +handle(): void
  - +handleRequest(): void
  - +handleReceivedData(): void

- lacData
  - +name: string
  - -isRequest: bool
  - +handle(): void
  - +handleRequest(): void
  - +handleReceivedData(): void

- <template> MmsVariable<T>
  - +data: T
  - +handleRequest(): void
  - +handleReceivedData(): void

- <template> lacVariable<T>
  - +data: T
  - +handleRequest(): void
  - +handleReceivedData(): void
Chapter 5. Modeling and Simulation of 800xA

The class RoutingTable handles the storage of nodes in routing tables, and each entry is described by an instance of KernelData. Additionally, the class also stores two timestamps and booleans, in order to keep track of the time a node was last heard from and what network that is up. The creation of routing tables is performed when receiving RNRP hello messages. If the sender is previously unknown, a new entry in the array of type RoutingTable is added, otherwise the time since it was last heard from is updated. The routing tables are also managed by a periodic check hello task, which is responsible for checking the time when a node was last heard from. The task, that also is part of the initVital() method, traverses through the current entries of the routing table and checks if a node (on the primary or secondary network) has exceeded the node down-timer set to 4 seconds. If so, the node on that network is regarded as dead, and the boolean prim-/secIsAlive is set to false.

When e.g. IAC or MMS messages are sent over the network, they always utilize the forwarding logic of RNRP (see Figure 5.3), to reach the correct node and network. As seen in the figure, the RNRP protocol either sends the messages on the primary or secondary link depending on which network that is up. If the message is to be broadcast (i.e. a hello message), it skips the logic and is assigned the TrueTime broadcast address '0'. If a receiver is not found in the routing table, the message is forwarded to the default router in order to direct it to its correct destination.

If a network node fails during execution, this is taken care of thanks to the fact that the forwarding logic is performed for each message sent. This means that a new path, if it exists, will be assigned to the message.
Figure 5.3 The forwarding logic of RNRP.
5.4 Simulating TCP with TrueTime

This section describes the way the TCP protocol has been implemented in the protocol_helper.cpp helper file.

The TCP Connection Class

The TCP connection class is responsible for handling the traffic between two kernels over a certain TCP connection. An overview of the class can be seen at Figure 5.4. It contains information of:

- The IP of the host and destination
- The name of the host and destination
- The source port of the host and destination
- Sequence count (Next segment to send)
- Integer of the last ACK that has been sent
- Integer of the last segment that has been received
- A send window and a receive window
- A state of the connection (INIT, TRANSFER, CLOSE or REINIT)
- A counter of number of duplicate ACKs in a row
- Send-, receive- and a temporary (for unordered packets) buffer
- A list of all transmitted TCP packets (for later deletion)
- An MMS variable counter
Figure 5.4 An overview of the TcpConnection class, the class responsible for handling a TCP connection in the protocol_helper.cpp file.
Connection Establishment and Termination

Before transmitting data over the TCP connection, a connection must be established from both client and server kernel. This has been implemented by a call to `Kernel::openTcpConnection("OtherKernel", sourcePort, destinationPort)`, which initializes all required variables and adds the connection to the list of TCP connections in the kernel. This also initiates a three-way handshake as described in Section 3.5, and during the establishment process the state of the connection is set to INIT where further data transfer is blocked. When the connection establishment is finished in both ways, the event TransferState is triggered and puts the connection to state TRANSFER allowing for data communication. In the same way, a connection can be terminated by a call to `Kernel::closeTcpConnection(TcpConnection* tcp)`. This issues a four-way handshake termination process, as well as freeing used memory and removing the connection from the TCP connections list in the kernel.

Sending and Receiving Data

The sending and receiving mechanism has been made in layers. This is mainly for simplifying the rather complex logic but it also follows the design a real TCP/IP stack is using. The job of the TCP connection is to go from Application layer, in this case the MMS protocol, to the RNRP protocol at the network layer (see Figure 3.2). To facilitate the implementation of the TCP/IP stack, the MMS protocol functionality was merged into the TcpConnection class, but this part is covered in Section 5.6.

To send a packet over an established connection, one must first check the TCP status with a call to `TcpConnection::send()`. Then, to actually start the sending process, a call to `TcpConnection::put()` is made. The application code in the sender can look something like this:
5.4 Simulating TCP with TrueTime

TcpConnection* tcp = kernel->getTcpConnection(port);

switch (seg) {
    case 1:
        status = tcp->send(length, currentStateOfConnection)
        // Handle status..
        if(Has to wait) {
            // Go back to this code segment
            // in next method call
        }
        if(Send buffer is full) return FINISHED;
        if(No problems) return 0.0;
        return someWaitingExecTime;
    case 2:
        tcp->put(msgData, length);
        return someExecTime;
    default:
        return FINISHED;
}

The send() method’s responsibility is to tell if it is ok to send a packet or not. It can result in any of the following three statuses:

- NO_PROBLEM
- HAS_WAITED
- BUFFER_FULL

If the send buffer is full (has less space than the number of packets to be sent), nothing will happen and the status returned will be BUFFER_FULL. Else, if the connection is not in TRANSFER state, the transmitting will have to wait. Hence, the program will start waiting for the event ”TransferState” and return the status HAS_WAITED. Otherwise, if everything went as it should, the send() method will return NO_PROBLEM.

In the put() method, the message is split into segments if the size is greater than MAXPACKETSIZE. Each segment is created as a new TcpPacket pointer, with variables defined in Figure 5.2. The only segment that actually carries the payload Data pointer is however always the last one; the previous ones are only dummy segments with a size and sequence number as if they also were carrying data. The data carrying segment is issued with a flag FIN set to true, in order to make it possible to retrieve the data at the receiver side. After the segmentation, the segments are put into the sending buffer and the method TcpConnection::sendIt() is invoked. There, the segments in the buffer that are within the send window are sent using
Chapter 5. Modeling and Simulation of 800xA

tNet0Send().

However if the to the put() method passed Data pointer is null, the packet is treated as an ACK and instead directly passed to the TcpConnection::sendAck() method. The ACK packets are thus not entered into the send buffer, rather they are sent independently as they are not included in the flow control. Read more about this in Section 5.4.

On the receiver side, the code could roughly look like this:

```cpp
TcpConnection* tcp = kernel->getTcpConnection(port);
switch (seg) {
  case 1:
    tcpPackets = tcp->get();
    return someExecTime;
  default:
    return FINISHED;
}
```

The only thing the get() method does, is to loop through the receive buffer until a packet with FIN flag set to 1 is found.

**Flow Control**

Flow Control is implemented in TrueTime using two instances of class Buffer (both send and receive) in each kernel. The send and receive windows are implemented as simple integers in the TcpConnection class. This means that they represent absolute sequence numbers rather than intervals, i.e. if the seqCount variable is smaller than sendWindow, it is regarded as within the allowed send window.

**Error Control**

The TCP protocol must ensure that data arrives and is in correct order. As mentioned in Section 3.5, the TCP protocol uses an ACK mechanism to ensure packet arrival and order. The ACK packets are created and sent in the sendAck() method upon arrival of a new segment.

The resending due to time-outs is taken care of by the interrupt handler timeOutHandler and its code function timeOutCode(). As each sent packet is assigned with a resend timer, by default set to 1 s, the expiration of such a timer will trigger an interrupt and thus execute the time-out code. This function calls TcpConnection:::resendAllInSendBuffer(), which utilizes the sendIt() method to resend the packets and reset all timers.
Simulating IAC with TrueTime

The other reason for a resend that is implemented is the arrival of three consecutive duplicate ACKs. This is implemented by a duplAcks counter in the TcpConnection class, which is increased each time when the ACK arrived is equal to lastAck. When it reaches a value of three the method resendAllInSendBuffer() is invoked and the counter is reset. The counter is also reset when a new ACK, e.g. greater than lastAck, arrives.

To guarantee that the segments arrive in order at the application layer, they are stored in a temporary receive buffer tmpRcvBuffer that is part of the TcpConnection class. Upon arrival of each segment, the tmpRcvBuffer is sorted and all segments that are in order and greater than the lastRcv segment are emptied from the tmpRcvBuffer into the (final) rcvBuffer.

Congestion Control
The implementation of congestion control in our version of TCP in TrueTime is left out. This is due to the fact that congestion in the network is not very likely since the messages are typically sent with a ~1 second periodicity.

5.5 Simulating IAC with TrueTime

As mentioned in Section 3.6, the IAC protocol is connectionless and is based on UDP. The IAC protocol however, has its own resending mechanism. The IacHandler can be used in a similar fashion as the TcpConnection, and an overview of the class can be seen in Figure 5.5. To set up an IAC handler simply write the following in an init script of a kernel:

```c
1 kernel->createIacHandler("OtherKernel", sourcePort,
2     destinationPort, analogChannel);
```

Here the analogChannel specifies what channel to use when handling IAC messages. To use the IAC handler, one must first set up the IAC handler in the two kernels that should communicate. This is done as described above. Then in some task code, a packet must be created and sent from the sender and received at the receiver. This is shown in the pseudo code below:
Chapter 5. Modeling and Simulation of 800xA

Sender

```
IacHandler* iac = kernel->getIacHandler(port);

switch (seg) {
    case 1:
        iacReq = new IacVariable<MeasurementToControlSignal>(
            "variable_name", value, isRequest);
        return someExecTime;
    case 2:
        iac->waitCheck();
        return 0.0;
    case 3:
        iac->createAndSendIacPacket(kernel, iacReq, size);
        return someExecTime;
    default:
        return FINISHED;
}
```

First, the IAC handler must be obtained by extracting it from the kernel, given a certain port number. Then in a first code segment (for an explanation of the segment structure used in this code, see Section 4.2), an IacVariable is created with the name and value of the variable to be exchanged. Also, a flag indicating that the packet is a request is set to true. The `waitCheck()` method in the second segment is handling the possible waiting due to network failures. In the third and last segment, the packet is sent. The `IacHandler::send()` method checks for a setting IAC_FILTER, see Chapter 7. If the filter is activated, the `send()` method ignores the normal way to send a packet, i.e. through the tNet0 task, and just sends the packet with the TrueTime predefined function `ttSendMsg()`. If the filter is not activated the normal call to `tNet0Send()` is used.
5.5 Simulating IAC with TrueTime

Receiver

```c
IacHandler* iac = kernel->getIacHandler(port);
IacPacket* iacPacket;
Data* tmp;
IacVariable<MeasurementToControlSignal>* var;

switch (seg) {
    case 1:
        iac->waitCheck();
        return 0.0;
    case 2:
        iacPacket = iac->get();
        var = dynamic_cast<IacVariable<MeasurementToControlSignal>*>(iacPacket->data);
        tmp = iacPacket->data->handle(kernel, port);
        return someExecTime;
    default:
        return FINISHED;
}
```

As in the case of the sending kernel, the receiving kernel must obtain the IacHandler instance with getIacHandler(port). In the first segment, a waitCheck is performed to handle possible network failures. In the next segment, the packet can be received and data can be extracted. This is done with the IacHandler::get() method that simply gets a packet from the IAC handlers receive buffer. In addition, the data can be "handled", which means that the receiving kernel somehow uses the data and it can also return a value used for a reply to the sending kernel. This reply can, for example, be a control signal.

When an IAC packet is created, a timer with an attached time-out handler is added to the packet. If the timer expires, i.e. a packet does not arrive in time, a special time-out handler code, called iacTimeOutCode(), is executed. This code is responsible for resending this individual packet (not a group of packets as in the TCP case).
Chapter 5. Modeling and Simulation of 800xA

Figure 5.5 An overview of the IacHandler class that facilitates an IAC connection.

5.6 Simulating MMS with TrueTime

The sending and receiving of MMS packet is very similar to the way sending and receiving of TCP packets is done. Below a pseudo code example is shown:
5.6 Simulating MMS with TrueTime

Sender

```cpp
TcpConnection* tcp = ctrl-&gt;getTcpConnection(port);

switch (seg) {
  case 1:
    // TRANSFER is the wanted state
    status = tcp-&gt;mmsSend(size, TRANSFER);
    // Handle status..
    if (Has to wait) { // The waiting now includes TOO_MANY_MMS wait
      // Go back to this code segment
      // in next method call
    }
    if (Send buffer is full) return FINISHED;
    if (No problems) return 0.0;
    return someWaitingExecTime;
  case 2:
    tcp-&gt;mmsPut(mmsReq, port);
    return someExecTime;
  default:
    return FINISHED;
}
```

Here, the TcpConnection::mmsSend() and TcpConnection::mmsPut() basically calls the TcpConnection::send() and TcpConnection::put() but with the added functionality that the MMS connection only allows three outstanding MMS packets. Thus, for example, the TcpConnection::mmsSend() added a status TOO_MANY_MMS to indicate that the sender must wait for a currently outstanding MMS packet to be received before trying to send another one.

Receiver

```cpp
TcpConnection* tcp = kernel-&gt;getTcpConnection(port);

switch (seg) {
  case 1:
    tcpPackets = tcp-&gt;mmsGet();
    return someExecTime;
  default:
    return FINISHED;
}
```

The TcpConnection::mmsGet() does, unlike TcpConnection::mmsSend() and TcpConnection::mmsPut(), use the TCP version of the method. This is because the use of TcpConnection::get() will extract any packet from the receive buffer.
In the MMS case, only the MMS packets should be extracted.
6

Results of TrueTime Simulations

This chapter covers the utilization of the implemented simulation. An example setup has been made for model experimenting, see Section 6.1. The chapter also includes analysis of the obtained simulation data and how to interpret some measures of performance.

6.1 Sensor/Actuator - Regulator Example

An example simulation has been made containing two controllers communicating over a redundant network with MMS and IAC. This has been chosen to keep things as simple as possible, but so that it still produces some interesting results.

One of the controllers takes the role as a sensor/actuator node and the other acts as a regulator node. The sensor/actuator works as a client that periodically takes a measurement from a "physical" process. Then it sends the measurement to the regulator as a request for a control signal. The regulator computes the control signal and sends it back to the sensor/actuator which sets the control signal to the process. To make it easier to compare the performance of IAC and MMS, the example has two identical processes, one controlled by sending IAC packets and one with MMS packets.

The simulation can be run with disturbances of the following kind:

- Measurement noise
- Packet loss probability
- Network interference load
- Interfering kernel tasks
Chapter 6. Results of TrueTime Simulations

Figure 6.1 The regulator example Simulink blocks.

The process has been chosen to have a proper time constant so that interesting results are obtained from a simulation. The typical ABB customer system is some kind of process industry. In such a system it is usually more important to stay close to a certain reference during load disturbances rather than quickly respond to reference changes. This has been taken into consideration in the simulations and a process with a square wave as load disturbance is used.

The regulator node uses a discretized PID control algorithm with tracking based anti wind-up, implemented in the code below:
First, the P, I and D parts of the control signal are calculated. Then, the signal is saturated, and finally, the "old" values of the PID parts are stored. However, notice that the I part uses a tracking-based algorithm for anti wind-up. The observant reader notices the use of the PidData class. The PidData class contains information of the PID settings used. It can also be used to "store" a function describing a control signal algorithm as the one above. An excerpt from the PidData class is shown below:
### Chapter 6. Results of TrueTime Simulations

```cpp
class PidData : public Data {
public:
    // ctrl params
    double K, Ti, Td, N, h, ad, bd, beta, Tr;

    // ctrl states
    double yold, Iold, Dold, u;

    double (*calcFunction)(PidData* pid, double y, double r);

    PidData(string name, double k, double ti, double td,
             double tr, double beta, double n, double h,
             double (*calcFcn)(PidData* pid, double y, double r))
        : K(k), Ti(ti), Td(td), Tr(tr), beta(beta), N(n), h(h),
    yold(0.0), Dold(0.0), Iold(0.0), u(0.0), calcFunction(calcFcn)
    {
        variableName = name;
        ad = Td / (N * h + Td);
        bd = N * K * ad;
    }

    // Some other stuff here
};
```

In the simulations made in this master’s thesis, the following PID settings found in Table 6.1 have been used. These have been manually tuned for reasonable performance of the system.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K$</td>
<td>0.85</td>
</tr>
<tr>
<td>$T_i$</td>
<td>3.5</td>
</tr>
<tr>
<td>$T_d$</td>
<td>0.5</td>
</tr>
<tr>
<td>$T_r$</td>
<td>$\frac{T_i + T_d}{2}$</td>
</tr>
<tr>
<td>$N$</td>
<td>100</td>
</tr>
<tr>
<td>$\beta$</td>
<td>1</td>
</tr>
<tr>
<td>$h$</td>
<td>1</td>
</tr>
</tbody>
</table>

**Table 6.1** The PID settings used in the simulations.

### 6.2 Measuring Performance

To get some kind of result from a simulation, a way of measuring performance is needed. Hence, it is logical to separate between network, kernel and control perfor-
6.2 Measuring Performance

The control performance is directly dependent on the other two but it is still good to be able to analyze them separately.

**Network Performance**

To measure the network performance, two aspects have been studied: network utilization and round trip times (RTT). The network utilization is calculated using the network schedule from the network block in Simulink. The calculation is done by a MATLAB script that first defines a plot interval and makes a time weighted average for the scheduling values in each interval. Notice that due to the nature of discrete event simulation there will be a lot of values in some time intervals and fewer in other. This is why the time weighted average is done. After this a smoothing filter is used on the averaged values.

This might seem a bit unscientific and it is too. There is no good definition of utilization and when you think about it, utilization in reality can only be 1 (something is happening) or 0 (nothing is happening). This smoothing is just making it easier for a human to interpret the result. Below the code for calculating "smooth" utilization is shown.

function [ smooth_util plot_t ] = calc_util( schedule, plot_interval )
%CALC_UTIL Calculates a smooth util from schedule
    t = schedule.time;
    util = zeros(max(t)/plot_interval, 1);
    momentan_util = schedule.signals.values;
    util_size = size(momentan_util);
    util_rows = util_size(1);
    util_cols = util_size(2);
    d = diag(1:util_cols);

    %norm to interval 0 - 0.5
    momentan_util = momentan_util - (ones(util_size)*d);

    %for total requested util
    momentan_util = momentan_util > 0.13;
    total_momentan_util = zeros(length(momentan_util), 1);

    for j = 1:util_cols
        total_momentan_util = total_momentan_util + momentan_util(:,j);
    end

    i = 1;
    val = total_momentan_util(1);
    while(i <= util_rows)
        i_old = i;
        i = find(total_momentan_util(i:end) ~= val, 1) + i - 1;
        if(isempty(i))
            break;
        end
        util(ceil(t(i)/plot_interval)) = val*(t(i) - t(i_old));
        val = total_momentan_util(i);
    end

    util = util./plot_interval;
    smooth_util = smooth(util, 50, 'lowess');
    plot_t = plot_interval:plot_interval:max(t);
end

The smooth command with the 'lowess' parameter is using a local regression with weighted linear least square to smooth the data. See http://www.mathworks.se/help/curvefit/smooth.html for more information about the smooth command. In Figure 6.2 a smoothed network utilization plot is shown. Compare this to Figure 4.2 which is the schedule used for the smooth plot.
6.2 Measuring Performance

The round trip times are calculated by comparing the time of when a request is sent from a client at application layer, to the time when the reply of this request is received at the application layer.

**Kernel Performance**

The same code described in the network performance section above can be used to calculate CPU utilization from the schedule given by the kernel blocks.

**Control Performance**

There are several ways to measure the control performance of a system and some of the most common ones are overshoot, settling time and rise time of a step response. However, in process industry, which is where the 800xA system is mostly used, one is often interested to stay as close to a static reference as possible rather than following a reference change. Due to this, it is more interesting to look at performance measures that somehow integrate the error term over time. Below are the chosen measures in order to evaluate control performance:

- IAE - Integral Absolute Error
Chapter 6. Results of TrueTime Simulations

- ISE - Integral Squared Error
- ITAE - Integral Time Absolute Error

\[ IAE = \int_{0}^{T} |e(t)| \, dt \]  

\[ ISE = \int_{0}^{T} (e(t))^2 \, dt \]  

\[ ITAE = \int_{0}^{T} t|e(t)| \, dt \]

IAE is simply integrating the distance from the reference. ISE has the property that large deviations from the reference will have a much larger impact on the ISE value than smaller deviations. In ITAE the time \( t \) is started at some kind of step, either in reference or in a load disturbance. The weighting with \( t \) makes ITAE sensitive to overshoot and oscillations.

6.3 Handling of Packet Losses

In a real network, packets do not always get to their destinations. This is referred to as a packet loss. In TrueTime, each network can set the loss probability, that is, the probability of losing a packet when sending it. As the MMS and IAC protocols use different ways of handling lost packets, it is interesting to measure and compare the performance of systems using MMS and IAC to communicate.

Simulations have been run with four different loss probabilities: 0 %, 0.1 %, 1 % and 10 %. For each setup, plots of the measurement signal and control signal have been studied as well as the performance measures described in Section 6.2.

Below in Table 6.2, the performance measures from the simulations are presented. In general, the IAC protocol is better than the MMS protocol for handling frequent packet losses. But with low loss probabilities, the difference is very small. In a real network, the loss probability is often very small. For instance, in the measurements done on a real system in this master’s thesis, no packet losses were recorded.

At 10 % loss probability, the IAC protocol is a clear winner as the MMS protocol has 16.26 % higher IAE, 27.84 % higher ISE and 19.79 % higher ITAE. This is, however, an extreme case with very high loss probability. Though, it should be noticed that the process to be controlled could be of a more difficult nature. That is, the 10 % loss probability might be extreme, but the process in a real system could
be more sensitive to lost packets, thus a lower but non-zero packet loss probability can be enough to distinguish between the protocols. This difference in system performance is mostly due to the different resending strategies of IAC and MMS. As stated in Section 3.6, IAC has a maximum retransmission time of 500 ms and no restrictions on number of outstanding packets, as opposed to the MMS protocol that is forced to use the error control features of TCP, see Section 3.5. In addition, the MMS protocol’s limitation of three outstanding packets could deteriorate control performance greatly.

Furthermore, in the case of 0 % loss probability, there should be no difference at all due to packet loss handling, but there still is some differences in the performance measure. This indicates that there is some other mechanism involved in the results, such as task timing in the controllers. Thus, one should be careful not to put too much weight on the numbers given in Table 6.2.

<table>
<thead>
<tr>
<th>Loss Probability</th>
<th>IAE / $10^3$</th>
<th>ISE / $10^3$</th>
<th>ITAE / $10^5$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IAC</td>
<td>MMS</td>
<td>IAC</td>
</tr>
<tr>
<td>0</td>
<td>1.5827</td>
<td>1.5883</td>
<td>5.2008</td>
</tr>
<tr>
<td>0.001</td>
<td>1.5799</td>
<td>1.5883</td>
<td>5.1780</td>
</tr>
<tr>
<td>0.01</td>
<td>1.6085</td>
<td>1.6111</td>
<td>5.2684</td>
</tr>
<tr>
<td>0.1</td>
<td>1.8231</td>
<td>2.1195</td>
<td>5.7977</td>
</tr>
</tbody>
</table>

Table 6.2  Performance measures from simulations comparing the MMS and IAC protocols, handling different probabilities of packet loss.

In Figure 6.3, measurement- and control signals are plotted for the processes controlled with systems using MMS and IAC respectively, when loss probability is set to 10 %. In this case, it is clear that the IAC protocol’s retransmission mechanism outperforms the one of MMS.
Chapter 6. Results of TrueTime Simulations

Figure 6.3  When the packet loss probability is set to 10 % the system performs poorly, but the IAC protocol handles the lost packets better than the MMS protocol.

6.4 Investigating RNRP Settings

As mentioned, the 800xA system has the possibility to use redundant networks. This is handled by the RNRP protocol and as complex as the system is, it is hard to tune the RNRP settings. Hence, a possibility to quickly test different setups with help from a simulation is desirable. In this section a closer look is taken at some of the attributes of RNRP.

Network Failure

A possible situation in a system is that a controller’s network stops working. In Figure 6.4 and Figure 6.5, a simulation example of when this happens is provided. In a simple system like this there is not much to analyze, but in larger systems with many nodes, investigating what happens when a link down event occurs can be more interesting. At least, this simple simulation shows that the MMS and IAC protocols handle a link down event differently. The IAC protocol takes action more
6.4 Investigating RNRP Settings

quickly, which leads to a faster but a bit more oscillating recovery.

Figure 6.4 Control and measurement signals when the Sensor/Actuator nodes primary network is destroyed at \( t = 50 \) s. RNRP redirects network traffic for the Sensor/Actuator controller to network 2. Also, notice the slightly different handling of the recovery from switching by the MMS and IAC protocols.

Figure 6.5 clearly shows that all traffic is stopped for the Sensor/Actuator node (Node 1 in the plot) in Network 1, while the traffic is increased in Network 2. A plot of the total network utilizations is left out as the total load on the network is not interesting in this case. Also, the change in network load is rather small, as Network 1 still is loaded with traffic of hello packets from the Regulator node.
Figure 6.5 The schedules for Network 1 and Network 2 respectively when the Sensor/Actuator nodes primary network is destroyed at $t = 50$ s. RNRP redirects network traffic for the Sensor/Actuator controller to network 2 which is illustrated by the lack of traffic for node 1 (The Sensor/Actuator controller) in Network 1 and the increased traffic in Network 2 for the same node.

Lost Hello Packets Allowed

In Section 3.3 it is described how the RNRP protocol handles lost hello packets. As default, three hello packets must be lost before RNRP switches to the backup network. Though, this number has been decided in an ad hoc manner and it would be nice to be able to experiment with different numbers of allowed last hello packets. It is preferable to keep robustness against occasionally dropped packets, but on the other hand, a fast detection of a broken node is also desirable. The first would require a high number and the latter a low, i.e. it is a matter of opinion of what is most important. A typical argument would be that it is more important to keep robustness against false alarms rather than a fast recovery from a node down event as this is something that should not happen very often. Also, switching between networks takes one hello packet period to do, which definitely should be avoided. This is of course very subjective and the importance of each aspect is highly dependent on
what system that is to be controlled. Anyway, a simulation could ease the decision.

In order to trigger node down events often enough to see something interesting in the simulations, the packet loss probability has been set to 10% which is a lot considering how good modern networks are. Then, the number of lost hello packets allowed has been changed from 0 to 3. At 3 allowed lost hello packets, no node down events were triggered, making it meaningless to run simulations with more than 3 allowed lost hello packets. With this in mind one can come to the conclusion that 3 allowed lost hello packets is a very robust setting, making false alarms of node down events almost impossible, even on bad networks.

Figure 6.6 shows simulation data from a setup using 0 lost hello packets allowed. This introduces a lot of network switching, but it occurs in such a way that it does not affect system performance in our example. Simulation data from a system with default setup, that is, 3 allowed lost hello packets, is seen in Figure 6.3.

![Simulation data](image)

**Figure 6.6** Simulation data of process control when the number of allowed hello packets lost is set to 0.
Chapter 6. Results of TrueTime Simulations

Actually, when looking at the performance measures of the simulations using different numbers of allowed lost hello packets, there is no noticeable difference. This is probably because of the relatively slow and easy controlled process that is used for the simulations. Hence, a network switching event, causing lost network traffic for one second, does not have an impact on the system as it is just at very few time instances that such a delay would affect the control performance.

Now why use such a process in simulations if there is no impact from different RNRP settings? The answer is simply that the process is typical for an ABB customer. Process industry is usually slow and robust. At least this shows that the RNRP settings are not of great importance when looking at control performance, but in general network switching should still be avoided. Therefore, the default setting of three allowed lost hello packets is probably to prefer.
7

Optimization of IAC traffic

In order to test the accuracy and usefulness of the simulator, and at the same time possibly improve the 800xA system, an attempt of optimizing the handling of IAC traffic has been made.

7.1 The Current Implementation

As it is now, all network traffic, incoming and outgoing, must pass a thread $t_{\text{Net0}}$ before being handled by the protocol handler thread. The $t_{\text{Net0}}$ thread is relatively low prioritized and thus, this mechanism may slow traffic handling down. Interaction between the different threads in a controller is seen in Figure 7.1.
Chapter 7. Optimization of IAC traffic

Figure 7.1  In the current implementation all traffic has to go through the low prioritized tNet0 thread.

7.2 The Proposed Optimization

The IAC protocol is, among other things, made for fast communication. Hence, it would be beneficial to skip the tNet0 thread to speed things up. In order to do this, a filter must be added just after the network interrupt. Then if the packet is an IAC packet, the tNet0 is skipped, see Figure 7.2.
In the optimized implementation a filter has been added to separate IAC traffic from other traffic. This way, IAC traffic does not go through the low prioritized tNet0 thread which reduces round trip times.

In practice, the handling of incoming packets could be achieved by letting the interrupt handler post a job to a filter thread, which will store the packet in a local buffer if it is an IAC packet. Then, when the protocol handler thread wants to read from the IAC packet, it can read from this buffer instead of reading from the TCP/IP stack. In our implementation, there is no separate filter thread, but an existing thread "Exception" has been borrowed for this purpose. This is done due to the difficulties of adding another thread in the complex thread structure of the controller. The optimized implementation is thus not a finished product, but more a proof of concept.

The handling of outgoing packets is probably a bit more complicated and has been left out in this master's thesis. It would at least somehow involve keeping track of
other controllers in the network with a routing table. The stack skipping workaround could probably be implemented in a similar fashion as with the incoming packets.

7.3 When Are Faster Round Trip Times Beneficial?

When optimizing the round trip times one must have in mind that it is only in certain system setups that the shortened round trip times actually affect the system as a whole. As seen in Figure 3.11 - 3.12, a short round trip time will only have an impact when the 1131 task is run with a relatively short period. For instance, if the 1131 task period is 10 seconds, the probability that a shorter round trip time will affect the system at application layer is very small. Though, when pushing a system with frequent 1131 task execution it may definitely boost performance.

An example simulation shows that using an 1131 task period of 20 ms, that is, the same as the protocol handler period, performance measures improve slightly. For instance, in this example, IAE was 0.96 % higher for the unfiltered version of IAC traffic handling. With an 1131 task period of 100 ms the IAE is only 0.35 % higher. Though, turning the 1131 period up to 500 ms, the relative difference is 0.36 % which is a bit higher, but this is probably due to that the process is getting close to being unstable which makes the slightly better round trip times more valuable. At an 1131 period of one second, the process is unstable. Plots with comparisons of the results have been left out as the curves using the same 1131 period look almost identical, though, the performance measures are presented in Table 7.1.

<table>
<thead>
<tr>
<th>1131 Task Period</th>
<th>IAE / 10(^3) Filter</th>
<th>IAE / 10(^3) No Filter</th>
<th>ISE / 10(^3) Filter</th>
<th>ISE / 10(^3) No Filter</th>
<th>ITAE / 10(^4) Filter</th>
<th>ITAE / 10(^4) No Filter</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 ms</td>
<td>1.5763</td>
<td>1.5915</td>
<td>4.9243</td>
<td>5.0054</td>
<td>7.7270</td>
<td>7.8076</td>
</tr>
<tr>
<td>100 ms</td>
<td>1.6274</td>
<td>1.6331</td>
<td>5.1783</td>
<td>5.1968</td>
<td>8.0028</td>
<td>8.0307</td>
</tr>
<tr>
<td>500 ms</td>
<td>2.9925</td>
<td>3.0032</td>
<td>9.6784</td>
<td>9.7132</td>
<td>15.839</td>
<td>15.894</td>
</tr>
</tbody>
</table>

Table 7.1 Performance measures comparing filtered and unfiltered versions of the IAC protocol using different periods on the 1131 task during simulations. Shorter round trip times have a larger impact on systems using short 1131 task periods, unless the system is on the verge of being unstable.
Results and Validation of IAC Optimization

In order to know if the new filtered version of the IAC traffic handling implementation works, a validation must be performed. This has been done in steps:

1. Take measurements from the experimental system
   a) With original implementation
   b) With filtered version of implementation

2. Analyze the measurement data
   a) Compare filtered and unfiltered round trip times
   b) Find possible patterns of behaviour in the system

3. Compare measurements to simulation results

8.1 Measurement Results

To enhance the impact of the traffic filtering on the round trip times, it was decided to send more than one value between the controllers. Hence, the measurements have been taken with a system communicating with either five IAC communication variables, or five IAC communication variables and 50 MMS variables. The IAC communication variables are using the setting "fast", i.e. they are updated with a period of 1 second. The MMS variables are read every 200 ms.

Below is a selection of interesting measurement data. For each measurement both raw data and a histogram are shown for the convenience of the reader. First is Figure 8.1 - 8.4 with data from a system using both IAC and MMS variables. The first two figures show data from the system with the original firmware, that is the unfiltered
version, and the last two show data from the system using the filtered implementation.

![Measured RTTs in a test system](image1)

**Figure 8.1** Measured round trip times in a reference system using default firmware. The system is communicating with five IAC communication variables with setting fast and 50 MMS variables with cyclic reading every 200 ms.

![Histogram of measured RTTs in a test system](image2)

**Figure 8.2** Histogram of measured round trip times in a reference system using default firmware. The system is communicating with five IAC communication variables with setting fast and 50 MMS variables with cyclic reading every 200 ms.
8.1 Measurement Results

Figure 8.3 Measured round trip times in a reference system using modified firmware. The system is communicating with five IAC communication variables with setting fast and 50 MMS variables with cyclic reading every 200 ms.

Figure 8.4 Histogram of measured round trip times in a reference system using modified firmware. The system is communicating with five IAC communication variables with setting fast and 50 MMS variables with cyclic reading every 200 ms.

Next, the measurements from a system using only five IAC variables are presented, see Figure 8.5 - 8.8. As before, the measurements of the system with original firmware are shown first, then the filtered version.
Chapter 8. Results and Validation of IAC Optimization

**Figure 8.5** Measured round trip times in a reference system using default firmware. The system is communicating with five IAC communication variables with setting fast.

**Figure 8.6** Histogram of measured round trip times in a reference system using default firmware. The system is communicating with five IAC communication variables with setting fast.
8.1 Measurement Results

**Figure 8.7** Measured round trip times in a reference system using modified firmware. The system is communicating with five IAC communication variables with setting fast.

**Figure 8.8** Histogram of measured round trip times in a reference system using modified firmware. The system is communicating with five IAC communication variables with setting fast.

As seen in the previously presented figures, the results seemed a bit odd. Thus, to certify that no strange disturbances on e.g. the network was taking place, another measurement was made with the exact same setup as in Figure 8.7 - 8.8.
Figure 8.9  Another measurement of round trip times in a reference system using modified firmware. The system is communicating with five IAC communication variables with setting fast.

Figure 8.10  Histogram of the other measurement of round trip times in a reference system using modified firmware. The system is communicating with five IAC communication variables with setting fast.

This is obviously not very similar to the previous measurement with the same settings, as the result differs with a factor of two. Understanding this behaviour was not easy but a possible explanation to the phenomenon was found. This involves the timing of the different tasks in the controllers, especially the protocol handler
8.2 Simulation Accuracy

Before measuring the round trip times, the simulation showed that the round trip times should be about 1-6 ms, see Figure 8.11. As the real measurements gives round trip times around either 20 ms or 40 ms, this model of the system is obviously not satisfactory.

Figure 8.11 The simulated round trip times from the model used before measuring round trip times from the system. The result is way of both quantitatively and qualitatively. The data comes from a simulation using the unfiltered solution.

Given that the measured round trip times were always around 20 ms or 40 ms, and that the same setup could give different results, an idea came up that some kind of timing mechanism with periodicity was going on. After much consideration, a hypothesis was formed. The explanation to the behaviour is that the protocol handler thread is periodic with a period of 20 ms and depending on how the two controllers are started in relation to each other, a pattern of 20/40 ms round trip times will emerge. Henceforth, a period of the protocol handler thread, that is 20 ms, will be known as one period.

Consider the following four scenarios:

1. Controller A is started slightly before controller B.
2. Controller A is started slightly after controller B.

3. Controller A and B are running with roughly half a period of the protocol handler thread.

4. Both controllers are started (almost) at the same time.

One important thing to keep in mind when analyzing these round trip times, is how a round trip time actually is defined. To be able to measure the round trip times at all, some changes in the firmware needed to be done and the way of measuring round trip times that was found possible to implement, utilized the protocol handler. Thus, a measured round trip time is the time from the packet arrival at the protocol handler at controller A, to the time that a response comes to the protocol handler at controller A. In Figure 8.12, the difference between this round trip time and the time it takes to go from application layer back to application layer again is illustrated.

![Figure 8.12](image)

**Figure 8.12** There is a difference between the round trip times at application layer and the measured round trip times from experiments.

In this way, the measured round trip times will always be approximately one or two periods and a bit shorter than a "real" round trip time. Also, notice that the measured round trip time will not be exactly one period as the starting and stopping time of the round trip time is taken somewhere in the middle of a protocol handler task execution. This means that a measured round trip time can be shorter than one or two periods, unlike the effect caused by delayed task executions, which only adds time to the round trip times.

In the first and the second case, the hypothesis is that, depending on how much the execution of the protocol handler task is delayed by interfering tasks, the round trip time will be either one or two periods, leading to a result like the round trip times
in Figure 8.5 - 8.6. In Figure 8.13 - 8.14 the two cases are explained.

The timing of the first case results in the round trip times always being one period on the non-interfered controller. As seen in Figure 8.13 this is because there is always enough time to not miss the execution of the protocol handler task, both from A to B and from B to A. However, if the protocol handler tasks are delayed "randomly" from interfering tasks, there are two different outcomes. If lucky, the delay caused by interfering tasks will not affect enough for the packet to miss a protocol handler task execution. But if the protocol handler task execution is delayed a bit more in A than it is delayed in B, the packet will miss the protocol handler task execution in B and will have to wait an extra period.
Chapter 8. Results and Validation of IAC Optimization

Figure 8.13 A timing diagram illustrating the behaviour when controller A is started slightly before controller B. Depending on the timing of how the protocol handler task execution may be delayed, different outcome in round trip times are retrieved. In the non-interfered case, round trip times are always around one period. When other tasks interfere, the round trip times can be either of one period or two.

As illustrated in Figure 8.14, the behaviour in the second case is much like the one in the first case. Only here, an extra period will be added to the round trip time if the protocol handler task is delayed more in B when receiving the request than in A when the response is to be received. In this way, the packet must wait for the protocol handler in controller A to execute before being sent to the application layer.
8.2 Simulation Accuracy

Figure 8.14 A timing diagram illustrating the behaviour when controller A is started slightly after controller B. Depending on the timing of how the protocol handler task execution may be delayed, different outcome in round trip times are retrieved. In the non-interfered case, round trip times are always around one period. When other tasks interfere, the round trip times can be either of one period or two.

In the third case, interfering tasks will not affect the timing enough to make a difference, thus, the round trip times will always be approximately one period long, that is 20 ms. As visualized in Figure 8.15, there is simply too much margin for the delay caused by interfering tasks to have an impact on the system.
Chapter 8. Results and Validation of IAC Optimization

Figure 8.15 A timing diagram illustrating the behaviour when controller A and B are started one half of a period apart. Both in the non-interfered case and in the interfered case, round trip times are always around one period. Since the controllers are started one half of a period apart, there is too much margin for the delays due to interfering tasks to have an impact on the system performance.

Finally, in the last case, the execution of the protocol handler task in both controllers will occur at almost the same time, resulting in them always just missing the other thread’s execution. As seen in Figure 8.16, this leads to round trip times of two periods, i.e. 40 ms. The reason that the randomness of the delays do not result in some lucky round trip times in this case is because a system with the protocol handlers in sync will also have the interfering tasks in sync. This makes the delays in A and B the same as indicated by the dotted vertical lines in Figure 8.16.
8.2 Simulation Accuracy

Figure 8.16 The timing between the two controllers protocol handler threads when both controllers start their protocol handler thread at the same time. Both with and without interfering tasks the round trip time will be about two periods long. The fact that this is the case even with interfering tasks, is due to that both controllers other tasks are timed identically and thus the delay caused by interfering tasks will always be the same for both A and B.

The explanations provided are just hypotheses. To get some support for the ideas, the phenomena were recreated in simulations. These simulations, unlike the ones used earlier, added the protocol handler threads to the controllers.

The first case was found using a setup where the Sensor/Actuator (Controller A in the figures above) was started one ms before the Regulator (Controller B). A histogram of the round trip times from this setup with the unfiltered implementation of IAC is shown in Figure 8.17.
Figure 8.17  Simulated round trip times from an unfiltered system with interfering tasks, where the Sensor/Actuator node is started 1 ms after the Regulator controller. This setup generates round trip times that alternate between one and two periods, just as the hypothesis described in this section predicts.

This shows the phenomenon of round trip times alternating between one and two periods. Just like the hypothesis predicts, a simulation of controllers without interrupting tasks gives round trip times of approximately one period, see Figure 8.18 below.

Figure 8.18  Simulated round trip times from an unfiltered system without interfering tasks, where the Sensor/Actuator node is started 1 ms after the Regulator controller. This setup generates round trip times that always are one period long, just as the hypothesis described in this section predicts.
The same behaviour can be seen for the second case. In Figure 8.19, a histogram is shown using a setup with the Sensor/Actuator (Controller A) starting one ms after the Regulator (Controller B). Also, the simulation using non-interfered controllers gives round trip times of one period in this case as well.

![Histogram of Round Trip Times](image)

**Figure 8.19** Simulated round trip times from an unfiltered system with interfering tasks, where the Sensor/Actuator node is started 1 ms before the Regulator controller. This setup generates round trip times that alternate between one and two periods, just as the hypothesis described in this section predicts.

The hypothesis is once again confirmed in the third and the fourth case. Figure 8.20 shows the round trip times of a system starting the controller one half of a period apart. In Figure 8.21, the simulation results of a setup where the controllers are started at the same time are presented. Both setups use the unfiltered implementation and the non-interfered results are consistent with the hypotheses.
Figure 8.20  Simulated round trip times from an unfiltered system with interfering tasks, where the two controllers are started with one half of a period apart. This setup generates round trip times that always are one period long, just as the hypothesis described in this section predicts.

Figure 8.21  Simulated round trip times from an unfiltered system with interfering tasks, where the controllers are started at the same time. This setup generates round trip times that always are two periods long, just as the hypothesis described in this section predicts.

Given the results presented in this section, one can obtain high accuracy after adjusting the simulation model. However, without prior knowledge about the system, the simulation model is usually rather poor and as a result, simulation data are not to be trusted.
8.3 Comparison to Current Implementation

As the results obtained from measurements show such varying behaviour, there is not much to analyze regarding the efficiency of the IAC traffic filter. However, as the adjusted simulation hopefully gives accurate results, a comparison between the filtered and unfiltered implementations can be done.

The huge impact the periodic protocol handler thread has on the round trip times cannot be magically removed. Though, in the case of round trip times alternating between one and two periods, as in case 1 and 2 described in the previous section, one might expect more "luck" resulting in more cases of one period round trip times. The simulated round trip times of a filtered versions corresponding to the systems of Figure 8.17 and Figure 8.19 are shown in Figure 8.22 and Figure 8.23. Here it can be seen that the systems using the same setup but with the filtered version of IAC traffic handling, outperforms the systems using the unfiltered implementation. However, these are the only cases substantially benefiting from the filtered IAC implementation.

Figure 8.22  Unlike the unfiltered version, this simulation data using filtered IAC traffic shows that round trip times in case 1 as defined in Section 8.2, using otherwise same settings, results in the round trip times always being one period long instead of alternating between one and two periods.
Unlike the unfiltered version, this simulation data using filtered IAC traffic shows that round trip times in case 2 as defined in Section 8.2, using otherwise same settings, results in the round trip times always being one period long instead of alternating between one and two periods.

In practice, it is hard to validate the hypotheses and simulation results with measurements. This is due to the difficulties of controlling the starting times of the controllers manually. The controllers are restarted by pressing and holding a button, so to systematically start controllers in a certain timing setup on purpose, e.g. one controller one ms before the other, is impossible. One could theoretically do a vast amount of restarts of the systems randomly and take measurements from this, but in practice this would take way too much time and effort as the measurement technique used in this thesis has been partially manual.
Conclusions

This chapter evaluates the results obtained in the thesis, as well as presenting some of the issues that were encountered during the workflow. In addition, possible future improvements are discussed.

9.1 Using TrueTime to Model 800xA

It has proven difficult to model a complex system as 800xA, as it requires a lot of knowledge of the system. The attempts on experimenting with different system settings in simulations, have resulted in insights regarding the effect packet loss probability has on control systems using MMS and IAC protocols. However, other simulation studies, e.g. experiments regarding RNRP, have been less rewarding.

Despite questionable gain of the studies done on overall system impact in this master’s thesis, the simulations of higher level protocols and controller timing behavior have been successful. As a consequence, the performed work can be used as a foundation for further studies of the 800xA system.

9.2 Optimizing IAC Traffic

Even though empirical evidence supporting the success of the optimized version of IAC traffic handling has not been obtained, there are reasons to believe that the partially finished optimization has a performance boosting effect on the round trip times. This conclusion comes from the round trip times given by simulation data. The reliability of the simulations is supported by the fact that simulated round trip times have been shown very similar to the experimental measurements, both qualitatively and quantitatively.

The impact of shortened round trip times on overall system performance however remains unclear. This is heavily dependent on process characteristics and low level task timing behaviour in the controllers.
9.3 Future Work

Unfortunately, time always limits what can be done in projects like this. Hence, ideas for future work are provided in this section.

**Improvements to TrueTime**

One obvious disadvantage of using TrueTime compared to other network simulators, is its lack of support of standard higher level protocol such as TCP. A lot of effort was put on implementing the TCP protocol in the simulations, so integrating this to the core of TrueTime would definitely be a nice feature.

Something that has not been mentioned in this master’s thesis, is the fact that the 64-bit version of Matlab has some issues with TrueTime. This problem has been avoided by using the 32-bit version of Matlab, but fixing this bug would be desirable.

**800xA Simulations**

The somewhat paradoxical fact of simulations, is that the simulation data is usually bad and unreliable until a good model of the system is obtained. At the same time, simulation data is most valuable when there is not much knowledge about the system. The result of this is that the modeling and simulation of a system becomes an iterative process, where validation and analysis of an old simulation leads to model adjustments and new simulation data. The new simulation results might give the users better understanding of the system, making it possible to further adjust the simulation model. This is the way the modeling of IAC round trip times was improved. This is also the way that every other aspect of the 800xA system simulation model can be enhanced. To sum up, almost endless future work can be spent on improving the simulations of the 800xA system.

**Expanding the IAC implementation**

The optimized version of IAC traffic handling developed is limited to optimizing the incoming IAC packets. Of course, it is preferred to optimize the packet route for outgoing packets as well. However, the implementation of such an optimization would probably require a bit more effort than for the incoming packet optimization. This is because when sending a packet, information of where to send the packet must be obtained. This is far from trivial in a system like 800xA, as the redundant network requires the controller to keep track of its peers in the network. Getting this information without going through the normal TCP/IP stack involves, for example, building routing tables.

Also, further investigations on the impact that shortened round trip times have on the overall system performance is needed. This demands either a system setup mech-
9.3 Future Work

anism which is able to set the start times of the controllers in the network on at least a millisecond scale, or large scale statistics of randomly started controllers. Furthermore, an analysis of how other threads in the controller are affected by the filter thread should be done to prevent degrading system performance in other components.
Bibliography

ABB. *System 800xA 5.1, Product Catalog (FP4 included).* URL: http://www05.abb.com/global/scot/scot354.nsf/veritydisplay/a7dc1ca53bd960e2c1257b9d003d63a0/$file/3BSE062937_E_en_System_800xA_5.1_Product_Catalog_(FP4_included).pdf.


ABB (2012). *System 800xA Network Configuration - System Version 5.1.*

ABB (2013a). *DoF - Inter Application Communication.*


J-Sim Homepage. URL: http://j-sim.cs.uiuc.edu/.

ns-3 Homepage (2014). URL: https://www.nsnam.org/.


Omnet++ Homepage. URL: http://www.omnetpp.org/.


QualNet Homepage. URL: http://web.scalable-networks.com/content/qualnet.

REAL Homepage. URL: http://www.cs.cornell.edu/skeshav/real/overview.html.


The OSI Model's Seven Layers Defined and Functions Explained (2014).

Abstract

The majority of industrial control systems today are distributed over networks. These systems tend to be complex, and can be difficult both to configure and to analyze; therefore, there is definitely a gain in being able to simulate the behavior of such a system prior to the actual implementation.

This master’s thesis investigates the possibility to simulate the ABB automation system 800xA by expanding the Matlab/Simulink tool TrueTime. Throughout, the work has been focused on modeling the network communications and the internal behaviour of the control system nodes. Furthermore, the developed simulator has been used to examine a proposal to optimize traffic handling of the ABB specific protocol IAC.

Different performance measures have been investigated, both regarding control and network performance. One measure of particular interest has been the round trip time of a packet. The simulator has proven to be able to reproduce results of round trip times measured in an example system, explaining timing behaviour originating deep down in the control system nodes. Other simulations have involved different system settings, analyzing their impact on system performance.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

Language
English
Number of pages
1-93

Recipient’s notes

Security classification

http://www.control.lth.se/publications/