Entwicklung einer Monitoring-Infrastruktur für ein verteiltes telerobotisches System für die minimal-invasive Chirurgie

Diplomarbeit im Fach Mechatronik

vorgelegt von
Jan Tully
geboren am 12. Juli 1983 in Schweinfurt

angefertigt am

Institut für Informatik

Lehrstuhl für Informatik 4
(Verteilte Systeme und Betriebssysteme)

Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer: Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat
Dr.-Ing. Jürgen Kleinöder
Dipl.-Ing. (FH) Stefan Jörg, Deutsches Zentrum für Luft- und Raumfahrt

Beginn der Arbeit: 1. Dezember 2009
Abgabe der Arbeit: 7. Mai 2010

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 7. Mai 2010 _____

Zusammenfassung

Ausgangspunkt ist ein modular aufgebautes Robotersystem, dessen Komponenten jeweils eigene Steuerungsrechner besitzen, die über ein kaskadiertes Kommunikationsnetz Daten – Befehle oder Ergebnisse – austauschen können. Aufgrund der Interaktion mit der realen Umwelt arbeiten die einzelnen Rechner mit Echtzeitbetriebssystemen. Zur Optimierung der Arbeitsprozesse und zur Fehleranalyse ist es wünschenswert, dass einzelne Module möglichst frei wählbare Informationen dem Benutzer oder einem Protokollierungsmechanismus zukommen lassen können.

Ziel der Arbeit soll es sein, dem vorhandenen System eine Infrastruktur zur Verfügung zu stellen, über die einzelne Module Daten in das Kommunikationsnetz einspeisen können, ohne dass die einzelnen Module Informationen über den Empfänger besitzen müssen. Von Interesse sind hier neben Debug-Informationen und Fehlermeldungen auch statistische Erhebungen zur Belastung der Roboter, beispielsweise in Form von durchschnittlichen Gelenkgeschwindigkeiten, Prozessorauslastung, Anzahl der Notstopps oder Betriebsstundenzähler. Außerdem sollen die Nachrichten ihre jeweiligen Entstehungszeitpunkte enthalten. Die Ausgabe der erfassten Daten soll – möglichst flexibel – auch an mehreren Stellen parallel erfolgen können und dort geeignet aufbereitet werden (z.B. zentrales Fehlerprotokoll oder Debug-Konsole). Das Echtzeitverhalten des Systems darf hierdurch nicht beeinträchtigt werden.

Zur Implementierung ist ein geeignetes Kommunikationsprotokoll zu entwickeln, das die Kaskadierungen des Gesamtsystems widerspiegelt und Datenpakete dementsprechend routet. Hierzu sind auf allen beteiligten Knoten des Netzes neue Kommunikationspunkte einzurichten, die einerseits Nachrichten über den aktuellen Zustand ihres Moduls an ihre übergeordnete Instanz senden können und andererseits Nachrichten untergeordneter Module weiterleiten können. Um den Echtzeitbetrieb des Systems nicht zu beeinträchtigen, muss gegebenenfalls eine Priorisierung der Nachrichten bzw., sofern möglich, ein anderer Kommunikationspfad gewählt werden. Insbesondere im Hinblick auf Grenzsituationen, in denen nur sehr wenig Rechenleistung zur Verfügung steht, aber viele Nachrichten verarbeitet werden müssen, muss dies sichergestellt sein und das System weiterhin bestmöglich arbeiten.

Abstract

Starting point is a modular robotic system, where each component has its own control computer, that exchanges data, such as commands or results, via a cascaded communication network. Due to the interaction with the real world, the computers work with real-time operating systems. To optimize the working processes and the error analysis, it is desired that individual modules can send arbitrary information to the user or a logging mechanism.

The aim of this thesis is to provide the existing system with an infrastructure, where modules can feed in data without any information about receivers. Besides debug information and error reports, statistical data regarding robot strains like average joint velocity, CPU load, number of emergency stops or uptime are of particular interest. Furthermore, messages are to contain the time of their generation. Output of adequately prepared data should be possible at multiple locations at the same time (e.g. central error log or debug console). Real-time constraints may not be violated hereby.

An appropriate communication protocol is to be developed, that reflects the system's cascading and routes data packages accordingly. This requires new communication points that have to be installed on every participating network node. Each of them is to be able to send information about its current state to superior instances and to forward subordinated modules' messages. In order to not interfere with the real-time operation of the system an appropriate prioritization of messages must be undergone and, if possible, a different communication path is to be selected. Particularly with regard to borderline situations in which very little processing power is available, but many messages have to be processed, this must be ensured and the operation of the overall system has to be guaranteed.

Inhaltsverzeichnis

1 Einleitung und Kontext **1**
1.1 Das MiroSurge Szenario 1
1.2 Systemaufbau 2
1.2.1 Sensor/Aktor-Ebene 3
1.2.2 Gelenk-Ebene 3
1.2.3 Echtzeitrechner-Ebene 3
1.2.4 Zusatzrechner-Ebene 4
1.3 Zielsetzung 4
1.4 Aufbau der Arbeit 5

2 Problemstellung **7**
2.1 Anforderungen 7
2.1.1 Integration in Software-Umgebung 7
2.1.2 Datenformat 7
2.1.3 Plattform-unabhängige Datenrepräsentation 8
2.1.4 Einfache Anpassbarkeit an Netzwerktopologie 8
2.1.5 Unabhängigkeit von Kommunikations-Hardware 8
2.1.6 Filter 8
2.1.7 Nachrichtenpriorisierung 9
2.1.8 Benutzerschnittstelle 9
2.1.9 Zeitstempel 9
2.1.10 Erkennung von Nachrichtenverlust 10
2.2 Randbedingungen 10
2.2.1 Neutralität bezüglich Benutzeranwendungen 10
2.2.2 Determinismus 10
2.2.3 Begrenzte Ausführdauer 11
2.2.4 Geringe Netzwerkbelastung 11
2.3 Zusammenfassung 12

3 Analyse existierender Lösungen **13**
3.1 Logging-Mechanismen 13
3.1.1 syslog-Protokoll 13
3.1.2 Logging-Frameworks am Beispiel Apache log4j 14
3.1.3 Google glog 15
3.1.4 AUTOSAR Diagnostic Services 15
3.1.5 Ergebnis 16
3.2 Lösungen zur lokalen Kommunikation 16
3.2.1 FIFO 17

3.2.2 POSIX Message Queues . . . 17
3.2.3 Shared Memory . . . 19
3.2.4 Message Passing Interface (MPI) . . . 19
3.2.5 Vergleich . . . 19
3.3 Lösungen zur Netzwerkkommunikation . . . 20
3.3.1 TCP . . . 20
3.3.2 UDP . . . 24
3.3.3 SCTP . . . 24
3.3.4 DCCP . . . 25
3.3.5 Qnet . . . 25
3.3.6 Vergleich . . . 25
3.4 Zusammenfassung . . . 26

4 Konzept **27**
4.1 Benutzerschnittstelle . . . 28
4.2 Datenformat . . . 28
4.3 Übertragungsinfrastruktur . . . 29
4.3.1 Aufbau . . . 29
4.3.2 Lokale Kommunikation . . . 30
4.3.3 Netzwerkkommunikation . . . 30
4.4 Zusammenfassung . . . 30

5 Ausarbeitung des Konzepts **33**
5.1 Datenformat . . . 33
5.1.1 Header . . . 33
5.1.2 Content . . . 37
5.2 Übertragungsinfrastruktur . . . 38
5.2.1 Client . . . 38
5.2.2 Server . . . 39
5.2.3 Datentransfer Client→Server . . . 48
5.2.4 Datentransfer Server→Server . . . 54
5.3 Benutzerschnittstelle . . . 56
5.3.1 Client . . . 56
5.3.2 Server . . . 57
5.4 Zusammenfassung . . . 60

6 Analyse der Implementierung **63**
6.1 Anforderungen . . . 63
6.1.1 Integration in Software-Umgebung . . . 63
6.1.2 Datenformat . . . 63
6.1.3 Plattform-unabhängige Datenrepräsentation . . . 64
6.1.4 Einfache Anpassbarkeit an Netzwerktopologie . . . 64
6.1.5 Unabhängigkeit von Kommunikations-Hardware . . . 64
6.1.6 Filter . . . 64
6.1.7 Nachrichtenpriorisierung . . . 65
6.1.8 Benutzerschnittstelle . . . 65
6.1.9 Zeitstempel . . . 66
6.1.10 Erkennung von Nachrichtenverlust . . . 66
6.2 Randbedingungen . . . 67
6.2.1 Neutralität bezüglich Benutzeranwendungen . . . 67

6.2.2 Determinismus 67
6.2.3 Begrenzte Ausführdauer 67
6.2.4 Geringe Netzwerkbelastung 68

7 Zusammenfassung und Ausblick **69**
7.1 Optimierungsmöglichkeiten und zukünftige Arbeiten 69
7.1.1 Zeichenkodierung 69
7.1.2 Filter 70
7.1.3 Zeitstempel 70
7.1.4 Reduzierung der Veröffentlichungsdauer 71
7.1.5 Reduzierung der Netzwerkbelastung 71
7.1.6 FPGA Implementierung 71
7.2 Überblick über Monitoring-Infrastruktur 71
7.3 Ergebnis 72

Kapitel 1

Einleitung und Kontext

Ausgangspunkt vorliegender Arbeit ist das MiroSurge Szenario des Deutschen Zentrums für Luft- und Raumfahrt [35] [36]. In diesem Kapitel soll der Kontext der Arbeit, sowie deren Zielsetzung vorgestellt werden.

1.1 Das MiroSurge Szenario

Der MIRO Roboterarm wurde zum Einsatz bei chirurgischen Operationen konzipiert, bei denen er assistierende und ausführende Aufgaben übernimmt. Insbesondere anspruchsvolle Tätigkeiten, die hohe Genauigkeit oder Kraft über einen längeren Zeitraum erfordern, sind prädestiniert dafür, maschinell unterstützt zu werden. Hierzu sind zwei Anwendungsmodi vorgesehen: Direkte, manuelle Führung, bei der Kräfte wie Gravitation und Reibung kompensiert werden, und ferngesteuerter Betrieb.

Aufgrund des knappen Raumes an einem Operationstisch und der gravierenden Folgen bei Kollision des Roboters mit dem Patienten oder anderen medizinischen Geräten wurde MIRO mit besonderem Augenmerk auf eine schlanke Form und geringes Gewicht (10 kg) entwickelt. Um die Bedienbarkeit intuitiv zu gestalten und damit die Einarbeitungszeit so kurz wie möglich zu halten, sind Gestalt und Bewegungsmöglichkeiten des Roboters einem menschlichen Arm nachempfunden. Am Ende des Armes sitzt an Stelle einer Hand eine Aufnahmevorrichtung für verschiedene, dem jeweiligen Aufgabenfeld anpassbare Werkzeuge, wodurch eine hohe Flexibilität des Systems erreicht wird.

Das MiroSurge Szenario (siehe Abbildung 1.1) ist eine Forschungsplattform bestehend aus MIRO Roboterarmen, die in dieser Arbeit zu Testzwecken eingesetzt wird. Es dient der endoskopischen Herzchirurgie und umfasst neben einer Kamera drei direkt am Operationstisch verschraubte MIRO Roboterarme, die mit aktiven endoskopischen Instrumenten ausgerüstet sind.

Um Standardkompatibilität zu gewährleisten, wurde beim Entwurf der Elektronik und Kontrollsoftware Wert darauf gelegt, Funktionalität auf General Purpose Architekturen zu implementieren. Alle Regler laufen dementsprechend

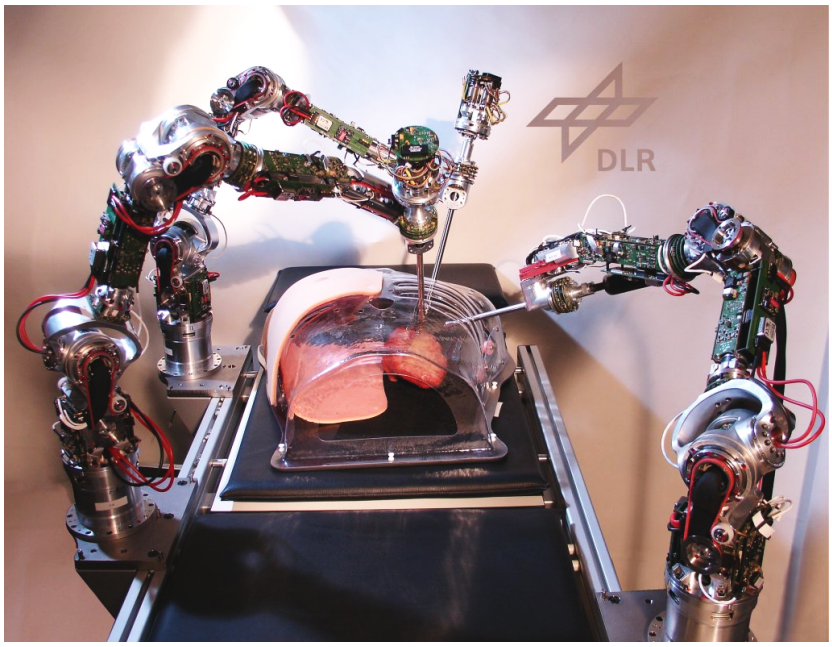


Abbildung 1.1: MiroSurge des DLR

nicht auf den für die Basisfunktionalität zuständigen Field Programmable Gate Arrays (FPGAs) der einzelnen Gelenke, sondern auf Standard PCs.

Aufgrund der harten Anforderungen dieses regelungstechnischen Systems (Regelschleifen mit bis zu 10 kHz) ist ein Betriebssystem notwendig, das die Ausführung der Algorithmen unter Echtzeitbedingungen ermöglicht.

Hier wird das Echtzeitbetriebssystem QNX Neutrino [15] verwendet, da dieses die erwähnten Echtzeitanforderungen erfüllt und durch vollständige Implementierung des POSIX Standards [5] die Verwendung von Software aus dem UNIX/Linux Bereich erlaubt. Speziell MATLAB/Simulink [10] ist damit als Codegenerator für den Entwurf der Regleralgorithmen einsetzbar.

1.2 Systemaufbau

In der MIRO Architektur wird zwischen Plattform- und Anwendungsfunktionalität unterschieden. Aufgabe der Plattform-Software ist die Verbindung der Steuerungsanwendungen mit der verteilten Elektronik, dargestellt durch Aktoren, Sensoren und die jeweiligen Ansteuerungen. Damit soll Anwendungsentwicklern die Implementierung von Low-Level Funktionen abgenommen und gleichzeitig wiederverwendbarer Code produziert werden.

Diese Struktur wird durch Unterteilung des Gesamtsystems in vier Abstraktionsebenen erreicht. (Siehe Abbildung 1.2.) Da die in vorliegender Arbeit erstellte

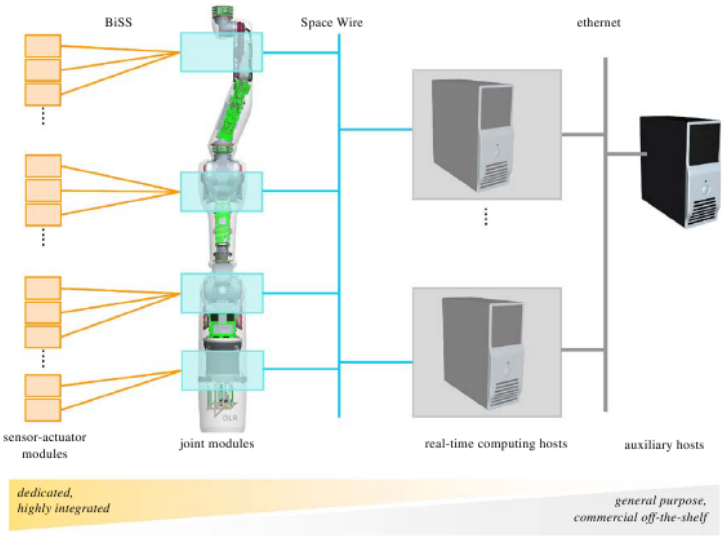


Abbildung 1.2: Architektur eines MIRO Roboterarms

Software mit allen Ebenen interagieren soll, werden deren Funktion, Architektur und Schnittstellen im Folgenden kurz aufgeführt.

1.2.1 Sensor/Aktor-Ebene

Die Sensor/Aktor-Ebene stellt die Schnittstelle zur physikalischen Welt dar. Hier überwiegen hoch optimierte, problemspezifische Lösungen zum Auslesen und zur Aufbereitung von Sensordaten und die Umsetzung von Aktorbefehlen in Bewegungen.

1.2.2 Gelenk-Ebene

In der Gelenk-Ebene werden Sensor- und Aktormodule zu Gelenkmodulen zusammengefasst. Diese Gelenke bilden die wichtigsten Grundbausteine des Roboters. Zentrale Steuereinheit der Gelenkmodule sind FPGAs, die vor Allem für die Übersetzung von Gelenkbefehlen in Aktorbefehle bzw. von Sensorinformationen in Gelenkinformationen zuständig sind.

Die Kommunikation mit den Sensor- und Aktormodulen erfolgt über das BiSS Protokoll [37], das speziell zur Anbindung von Sensoren und Aktoren an Standard-Hardware entwickelt wurde. Es bietet unidirektionale Master/Slave-Verbindungen bei geringem Overhead.

1.2.3 Echtzeitrechner-Ebene

In der Echtzeitrechner-Ebene befindet sich der Großteil der Anwendungsfunktionalität. Sie setzt sich zusammen aus miteinander vernetzten Standard PCs, die mit dem Betriebssystem QNX Neutrino arbeiten. Von hier geht die Steuerung der Gelenkmodule, sowie deren Regelung aus.

Die Rechner dieser Ebene kommunizieren mit den ihnen zugeordneten Gelenkmodulen über das SpaceWire Protokoll [30]. Es ist einfach genug, um auf den FPGAs der Gelenkknoten implementiert zu werden, und bietet hohe Bandbreite (1 GB/s) bei geringer Latenz (Roundtrip < 50 µs).

Auf dieser Ebene findet auch Kommunikation der Echtzeitrechner untereinander statt, da Aufgaben verteilt werden und Interaktionen notwendig sein können. So existieren im vorliegenden Fall dedizierte Rechner für jeden Roboterarm, die für dessen Positionsregelung verantwortlich sind und Aktionsbefehle von Rechnern der selben Ebene empfangen. Dieser Datenaustausch erfolgt über Punkt-zu-Punkt Ethernet-Verbindungen.

1.2.4 Zusatzrechner-Ebene

Zur Interaktion des Robotersystems mit Benutzern können an die Rechner der Echtzeit-Ebene zusätzliche Rechner angeschlossen werden. Von hier können beispielsweise Steuerungsbefehle gegeben oder Statusinformationen abgefragt werden. Echtzeitverhalten ist hier nicht erforderlich, so dass verschiedene Desktop-Linux-Distributionen als Betriebssystem eingesetzt werden können.

Der Datenaustausch zwischen Zusatz- und Echtzeitrechnern erfolgt ebenfalls über Ethernet, allerdings ohne Restriktion der Anzahl beteiligter Netzwerkknoten. Um die Kommunikation der Echtzeitrechner untereinander nicht zu beeinträchtigen, werden separate Netzwerkanschlüsse verwendet.

1.3 Zielsetzung

Das Gesamtsystem umfasst mehrere QNX und Linux Rechner verschiedener Abstraktionsebenen auf denen einzelne Software-Module ausgeführt werden. Somit findet nicht nur eine logische, sondern auch eine physikalische Trennung der Module statt. Ziel der Arbeit ist es, eine Kommunikationsinfrastruktur aufzubauen, über die trotz dieser Trennung Monitoring-Daten an eine oder mehrere Ausgabestellen übertragen werden können. Die Entkopplung der Komponenten soll hierbei jedoch gewahrt bleiben, so dass starre Verbindungen mit dem Ausgabegerät zu vermeiden sind.

Weiterhin soll eine Struktur der Monitoring-Daten (im Weiteren *Nachrichten* genannt) erarbeitet werden, die einerseits alle notwendigen Informationen zur Einordnung der Daten, und andererseits von Benutzern frei formulierbaren Text enthält.

Der Versand von Nachrichten darf die aufrufende Anwendung nicht unnötig verzögern. Ebenso müssen sich Prozesse der Monitoring-Infrastruktur neutral gegenüber Benutzerprozessen verhalten.

Da die Software-Module in C++ bzw. mit Hilfe von C++-Code-Generatoren entwickelt werden, soll die Einbindung der Monitoring-Funktionalität über eine C++-Bibliothek realisiert werden.

1.4 Aufbau der Arbeit

In **Kapitel 2** wird die Zielsetzung konkretisiert und in Anforderungen und gegebene Randbedingungen gegliedert.

In **Kapitel 3** wird untersucht, ob bereits existierende Lösungen ähnlicher Problemstellungen hier anwendbar sind. Dabei werden sowohl vollständige Lösungsansätze als auch Teillösungen betrachtet.

In **Kapitel 4** wird mit den in 2 und 3 gewonnenen Erkenntnissen ein eigenes Konzept entwickelt. Dabei werden Übertragungsinfrastruktur, Datenformat und Benutzerschnittstelle getrennt behandelt.

In **Kapitel 5** wird die Umsetzung des Konzepts aus 4 erläutert. Hier werden ebenfalls Übertragungsinfrastruktur, Datenformat und Benutzerschnittstelle getrennt bearbeitet.

In **Kapitel 6** wird die entstandene Implementierung mit den Voraussetzungen, die in Kapitel 2 aufgestellt wurden, verglichen und eine Bewertung vorgenommen, in wieweit die Aufgabenstellung erfüllt wurde.

In **Kapitel 7** wird zunächst noch einmal ein Überblick über die entwickelte Monitoring-Infrastruktur gegeben. Im Anschluss werden Lösungsansätze für nicht vollständig gelöste Teilprobleme und weitere Verbesserungsmöglichkeiten vorgeschlagen, sowie ein Ergebnis der Arbeit formuliert.

Kapitel 2

Problemstellung

In diesem Kapitel sollen zunächst die gewünschten Eigenschaften der zu entwickelnden Monitoring-Infrastruktur festgelegt werden. Für eine vollständige Spezifikation der Arbeit müssen auch die Randbedingungen berücksichtigt werden, die sich aus dem bereits vorhandenen System und dem Kontext der Monitoring-Infrastruktur ergeben. Die Randbedingungen müssen nachweisbar eingehalten werden, damit garantiert werden kann, dass die Monitoring-Infrastruktur die übrigen Teile des Gesamtsystems nicht beeinträchtigt.

2.1 Anforderungen

2.1.1 Integration in Software-Umgebung

Die Software-Module des MiroSurge Szenarios werden in C++ entwickelt. Es soll daher eine Bibliothek implementiert werden, die den Zugriff auf Monitoring-Funktionen aus C++-Anwendungen erlaubt. Zielplattform sind die QNX-Rechner der Echtzeit-Ebene, sowie die Linux-Zusatzrechner.

2.1.2 Datenformat

Ziel der Arbeit soll es sein, eine Möglichkeit bereitzustellen, Monitoring-Daten auszugeben. Der Begriff *Monitoring-Daten* kann jedoch nicht klar abgegrenzt werden und umfasst u.a. Fehler- und Statusmeldungen und Informationen zum Debugging. Die Daten sind demnach stark von der jeweiligen Verwendung abhängig und sollen deshalb frei formulierbar sein.

Um die Daten richtig interpretieren zu können, ist es jedoch notwendig zu wissen, an welcher Stelle und zu welcher Zeit sie generiert wurden. Diese Informationen sollen in einheitlichem Format an definierten Stellen einer Nachricht abgelegt werden.

Zur Klassifizierung der Nachrichten sollen diese außerdem Informationen über Typ und Dringlichkeit enthalten. Diese sind ebenfalls in einheitlichem Format an definierten Stellen einer Nachricht abzulegen. Um eine sinnvolle Klassifizierung zu ermöglichen, sollen für diese Informationen Werte vordefiniert werden, von denen je ein Wert einer Nachricht zugewiesen wird.

2.1.3 Plattform-unabhängige Datenrepräsentation

Es wird gefordert, dass Nachrichten sowohl bei ihrer Erzeugung als auch bei ihrer Ausgabe identisch dargestellt werden können. Das bedeutet insbesondere eine Adaptierung der Byte-Reihenfolge bei mehreren Byte großen Datentypen auf ein einheitliches Format, da verschiedene Rechnerarchitekturen dies auf unterschiedliche Weise handhaben [22]. Zudem ist die jeweilige Zeichenkodierung der Textsegmente anzugleichen [20].

2.1.4 Einfache Anpassbarkeit an Netzwerktopologie

Um die geforderte Modularität zu wahren, dürfen keine starren Kommunikationspfade verwendet werden. Ändert sich der Systemaufbau, muss die Monitoring-Infrastruktur ohne großen Aufwand angepasst werden können. Daraus folgt auch, dass die Empfänger einer Nachricht bei deren Versand nicht bekannt sein müssen.

2.1.5 Unabhängigkeit von Kommunikations-Hardware

Die Monitoring-Infrastruktur soll so ausgelegt werden, dass sie auf allen Ebenen des MiroSurge Szenarios (siehe 1.2) verwendet werden kann. Da hier verschiedene Techniken zur Kommunikation eingesetzt werden – SpaceWire und Ethernet – muss auch die Monitoring-Infrastruktur flexibel genug sein, um zumindest diese Techniken zu unterstützen.

Die vorhandenen Kommunikationswege können allerdings nur bedingt eingesetzt werden, denn zusätzlicher Verkehr könnte die Echtzeitanforderungen anderer Prozesse beeinträchtigen. Da dies nicht prinzipiell ausgeschlossen werden kann, muss die Infrastruktur so aufgebaut werden, dass auch auf alternative Kommunikationswege ausgewichen werden kann. Die Monitoring-Infrastruktur muss folglich einfach an neue Kommunikations-Hardware anpassbar sein.

Im Rahmen dieser Arbeit werden zunächst nur die Echtzeitrechner- und die Zusatzrechner-Ebene berücksichtigt. Zur Kommunikation soll die bereits vorhandene Anbindung aller Rechner an das institutsweite Ethernet genutzt werden, um die kritischen Wege zu umgehen. (Siehe Abbildung 2.1.)

2.1.6 Filter

Durch die Trennung von Nachrichtengenerierung und Ausgabe ist es nicht möglich, nur Nachrichten zu erzeugen, die auch für den Benutzer von Interesse sind. Beim Benutzer kommen deshalb alle Nachrichten an, die im gesamten Netzwerk generiert wurden. Da in den meisten Fällen jedoch nur eine bestimmte Gruppe von Nachrichten von Bedeutung ist, sollte dem Benutzer die Möglichkeit einer Filterung zur Verfügung gestellt werden. Die Parameter nach denen gefiltert werden soll, sollen dabei frei gewählt werden können. Stehen nicht genug Ressourcen für die Verarbeitung aller Nachrichten zur Verfügung, ist eine möglichst frühzeitige Filterung essentiell.

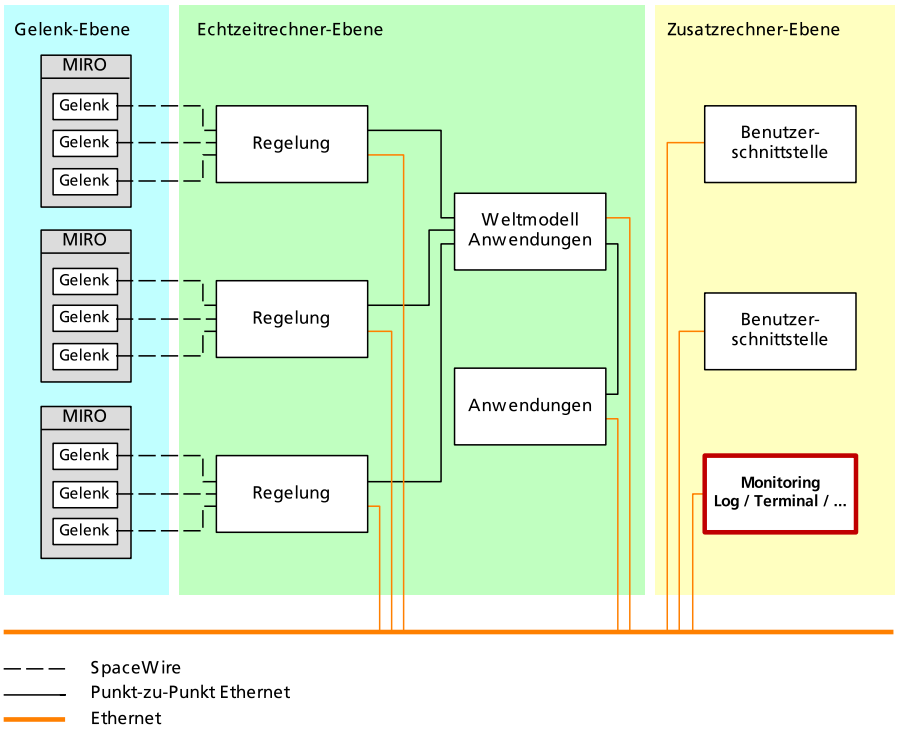


Abbildung 2.1: Schematische Darstellung des MiroSurge Szenarios mit Monitoring Erweiterung

2.1.7 Nachrichtenpriorisierung

Verschiedene Nachrichten besitzen in der Regel auch verschiedene Dringlichkeiten. Dies soll ebenfalls beim Transport berücksichtigt werden, so dass wichtige Nachrichten eine höhere Priorität erhalten und damit schneller und zuverlässiger zugestellt werden.

2.1.8 Benutzerschnittstelle

Die im Rahmen dieser Arbeit zu entwickelnde Software stellt dem Benutzer eine Möglichkeit zur Nachrichtenveröffentlichung bereit. Daher soll der Anwender auch nur eine dementsprechend einfache Schnittstelle vorfinden, die er ohne aufwendige Einarbeitung verwenden kann.

Auf der anderen Seite soll die Ausgabe und Filterung der Nachrichten so flexibel wie möglich gehalten werden. Dies schließt ein, dass auch mehrere Ausgabewege – bzw. Filter – gleichzeitig unterstützt werden sollen.

2.1.9 Zeitstempel

Ziel eines zentralen Monitorings ist es, Kausalketten von Ereignissen auch über Rechnergrenzen hinweg nachvollziehen zu können. Dazu ist es notwendig, dass die Reihenfolge, in der die Ereignisse aufgetreten sind, rekonstruierbar

ist. Daher soll jede Nachricht einen Zeitstempel tragen, der beim Erzeugen der Nachricht gesetzt wird. Um aufgrund von Zeitstempeln, die auf verschiedenen Rechnern gesetzt wurden, Rückschlüsse über die Ordnung ziehen zu können, müssen die Zeitgeber der Rechner synchronisiert sein. Dies ist [28] und [39] zufolge jedoch nicht exakt möglich. Damit kann bei nahe beieinander liegenden Zeitpunkten keine zuverlässige Aussage mehr über die Reihenfolge getroffen werden.

Da die Regelungsalgorithmen des vorliegenden Systems getaktet sind, sollte die Synchronisierung jedoch mindestens so genau sein, dass Takte verschiedener Rechner einander zugeordnet werden können. Die Qualität der Zeitsynchronisierung kann durch das Monitoring-System allerdings nicht beeinflusst werden und ist separat zu behandeln.

2.1.10 Erkennung von Nachrichtenverlust

Da nach 2.1.5 nicht bekannt sein kann, welche Übertragungsmedien eingesetzt werden, kann auch nicht garantiert werden, dass Datenpakete zuverlässig transportiert werden. Folglich können Nachrichten verloren gehen. Das ist tolerierbar, muss allerdings erkannt und dem Benutzer mitgeteilt werden.

2.2 Randbedingungen

2.2.1 Neutralität bezüglich Benutzeranwendungen

Eine Monitoring-Anwendung darf niemals den laufenden Betrieb der Anwendung stören, die überwacht werden soll. Folglich dürfen Prozesse der Monitoring-Infrastruktur nur dann ausgeführt werden, wenn kein anderer Thread bzw. Prozess lauffähig ist. Die Prioritäten der Monitoringthreads müssen also niedriger angesetzt werden als die niedrigste Priorität der Arbeit verrichtenden Threads bzw. Prozesse.

2.2.2 Determinismus

Auch die Übergabe einer Nachricht aus einer Benutzeranwendung an die Monitoring-Infrastruktur darf nicht zur Störung des Echtzeitverhaltens führen. Um die Auswirkungen einer Nachrichtenveröffentlichung im Voraus abschätzen zu können, muss vor Ausführung bereits bekannt sein, wie viel Zeit ein solcher Aufruf maximal benötigen kann [54]. Das impliziert u.a., dass auch keine externen, nicht determinierten Operationen, wie z.B. Speicherallozierung, durchgeführt werden dürfen. (Diese Einschränkung gilt nicht für Funktionen der Initialisierungsphase, da diese nicht echtzeitkritisch ist.)

Weiterhin können auch keine Mechanismen eingesetzt werden, die das Scheduling der Prozesse beeinflussen. Dies ist von Bedeutung, da bei einer Übergabe von Nachrichten an die Monitoring-Infrastruktur auf einen gemeinsamen Datenpuffer zwischen Benutzer- und Kommunikationsstruktur zugegriffen werden muss. Dieser Puffer muss auch bei mehreren gleichzeitigen Lese- und Schreiboperationen immer konsistent gehalten werden. (*Synchronisierung*)

Die meisten Synchronisierungsmechanismen, wie Mutexes (siehe [52] Abschnitt 2.3.6), Semaphoren (siehe [52] Abschnitt 2.3.5), etc., basieren jedoch auf der Sequentialisierung der Zugriffe. Dies kann allerdings zur Folge haben, dass ein hochpriorer Thread – hier die Anwendung – mit dem Zugriff warten muss, bis ein niederpriorer Thread – hier die darunter liegende Kommunikationsinfrastruktur oder eine andere Anwendung – seine Datenmanipulation abgeschlossen hat.Geeigneter ist es deshalb, die Datenstruktur schon so zu wählen, dass ein gleichzeitiger Zugriff erfolgen und somit auf explizite Synchronisierung verzichtet werden kann.

2.2.3 Begrenzte Ausführdauer

Das Monitoring-System soll auch auf den Rechnern eingesetzt werden, die im MiroSurge Szenario Regelungsaufgaben übernehmen. Wie in 1.1 erwähnt, besitzen die Regelschleifen Wiederholraten von bis zu 10 kHz. Daraus ergibt sich eine maximale Ausführungszeit eines Regelungsschritts von 100 µs. Zieht man hiervon noch die Zeiten zur Ausführung des Regelungsalgorithmus und der durch die Kommunikation entstehenden Verzögerungen ab, bleiben nur noch wenige Mikrosekunden für eventuelle Monitoring-Ausgaben. Die Zeit, die die Veröffentlichung einer Nachricht das Hauptprogramm maximal verzögern darf, ist also so gering wie möglich zu halten und sollte nicht über den einstelligen Mikrosekundenbereich hinausgehen. (Zum Vergleich: Eine Ausgabe durch die `printf` Funktion der Standard C-Bibliothek benötigt auf dem selben System ca. 200 µs.) Davon sind die Erstellung und das Befüllen der Nachrichten mit Nutzdaten ausgenommen, da diese Vorgänge schon in der Initialisierungsphase abgeschlossen werden können.

2.2.4 Geringe Netzwerkbelastung

Ebenso wie die CPUs der zu überwachenden Rechner, muss sich die Monitoringsoftware auch die Netzwerkinfrastruktur mit anderen, möglicherweise wichtigeren Anwendungen teilen. Aus diesem Grund muss die Belastung des Netzwerks gering gehalten werden. Es ist also nicht sinnvoll, jede wenige Byte große Nachricht einzeln in einem Transportpaket zu verschicken und damit zusätzlichen Overhead zu produzieren. Zweckmäßiger wäre eine Bündelung mehrerer Nachrichten in Pakete, deren Fassungsvermögen dem eines Transportpakets entsprechen.

Dieses Vorgehen ist allerdings nur bei hoher Frequenz der ausgehenden Nachrichten sinnvoll, da die Latenz zwischen Nachrichtengenerierung und Empfang beliebig groß werden kann, wenn mit dem Versand gewartet wird, bis eine Mindestanzahl an Nachrichten vorliegt. Zur Online-Überwachung ist es jedoch notwendig, dass diese Latenz gering gehalten wird. Außerdem würden bei einem Absturz des Rechners noch nicht versendete Nachrichten und damit Informationen, die Rückschlüsse über die Gründe des Absturzes zulassen, verloren gehen. Es muss also ein Kompromiss zwischen Ausnutzung der Netzwerkbandbreite und Optimierung der Verzögerung gefunden werden.

2.3 Zusammenfassung

In diesem Kapitel wurde der Kontext der Arbeit in verifizierbare Eigenschaften aufgeschlüsselt. Daraus ergaben sich einerseits Anforderungen, die sich direkt aus der Zielsetzung ableiten ließen, und andererseits Randbedingungen, die eingehalten werden müssen, um eine reibungslose Integration in das bestehende System zu ermöglichen. Wichtig bei der Zusammenstellung der Randbedingungen war insbesondere, dass das Monitoring-System im Hintergrund von echtzeitkritischen Anwendungen arbeiten soll.

Kapitel 3

Analyse existierender Lösungen

In diesem Kapitel werden zunächst existierende Mechanismen zur Übertragung von Log-Meldungen untersucht. Das geschieht in Hinblick auf die in 2 aufgeführten Kriterien. Weiterhin werden Eigenschaften dieser Mechanismen gesucht, die sich für eine eigene Implementierung adaptieren lassen.

Da der Transport von Nachrichten sowohl lokale Interprozesskommunikation als auch Netzwerkkommunikation erfordert, werden im Anschluss entsprechende Verfahren bzw. Protokolle analysiert.

3.1 Logging-Mechanismen

Logging-Mechanismen sind in praktisch jedem größeren (Software-) Projekt notwendig. Daher existieren entsprechend viele Bibliotheken, die verschiedene Logging-Probleme behandeln. In diesem Abschnitt sollen einige davon betrachtet werden. Aufgrund von Anforderung 2.1.1 werden nur C++-Bibliotheken näher untersucht, die auf UNIX-Systemen lauffähig sind.

3.1.1 syslog-Protokoll

syslog [5] [32] ist ein Standard zum Transport von Nachrichten über ein IP-Netzwerk und in praktisch alle unixoiden Betriebssysteme integriert. Der Transport erfolgt über eine Client/Server-Struktur. (Siehe [52] Abschnitt *1.7.5 Client-Server Model*.)

Clients sind dabei Nachrichten versendende Prozesse. *Server* übernehmen die Verarbeitung der Nachrichten, die von Clients erzeugt wurden. D.h. hier werden Nachrichten ausgegeben oder über das Netzwerk an andere Rechner versandt. Um Nachrichten von einer Netzwerkschnittstelle auch annehmen zu können, besitzen Server auch Empfangsfunktionalität.

Durch die Trennung von Nachrichtengenerierung und -verarbeitung wird die Dauer gering gehalten, die ein Anwenderprozess durch eine Nachrichtenveröffentlichung verzögert wird. Da Server sowohl Empfangs- als auch Sende- bzw. Ausgabefunktionalität besitzen, können Nachrichten über mehrere Server geleitet werden. Jedem Server muss demnach nur die jeweils nächste Station seiner Nachrichten bekannt sein, nicht jedoch der tatsächliche Zielpunkt. Damit ist es auch für Clients nicht notwendig den Ausgabepunkt ihrer Nachrichten zu kennen. Mit einer solchen Client/Server-Struktur sind folglich Anforderung 2.1.4 und Randbedingung 2.2.3 erfüllbar.

Ebenso wird Anforderung 2.1.2 durch syslog erfüllt: Eine syslog-Nachricht ist unterteilt in *Content* und *Header*. Der Content-Bereich trägt die eigentlichen Nutzdaten, die in strukturierter oder frei formulierbarer Textform gespeichert werden. Im Header werden Daten gespeichert, die zur Einordnung der Nachricht in einen Kontext notwendig sind. Konkret sind dies:

- eine Priorität
- eine Versionsnummer des verwendeten Protokolls
- ein Zeitstempel, der den Empfangszeitpunkt angibt
- Name oder Adresse des Netzwerkknotens von dem die Nachricht stammt
- der Name der Anwendung, die die Nachricht generiert hat
- Name oder Identifikationsnummer des Client-Prozesses
- ein Feld zur Identifizierung des Nachrichtentyps ohne feste Semantik

Der Zeitstempel einer syslog-Nachricht wird allerdings bei deren Empfang gesetzt, nicht bei der Erzeugung. Damit ist Anforderung 2.1.9 nicht erfüllt. Schwerwiegender ist jedoch, dass weder im syslog-Standard, noch in einer seiner Varianten (z.B. [11] [17] [18]), Determiniertheit (Randbedingung 2.2.2) gewährleistet wird. Damit ist syslog für vorliegenden Fall nicht einsetzbar. Die Client/Server-Struktur und der Nachrichtenaufbau eignen sich jedoch als Grundlage für eine eigene Implementierung.

3.1.2 Logging-Frameworks am Beispiel Apache log4j

Anders als syslog treffen Logging-Frameworks keine verbindlichen Aussagen über die Verarbeitung von Nachrichten. Sie implementieren lediglich Benutzerschnittstellen zur Nachrichtengenerierung und Programmierschnittstellen zur Anpassung der Ausgabe, sowie teilweise bereits vorgefertigte Ausgabemodule.

log4j [34] ist ein Logging-Framework der Apache Software Foundation und soll im folgenden repräsentativ betrachtet werden. Obwohl log4j primär für Java entwickelt wurde, existieren jedoch einige Portierungen nach C++ [7] [8] [9], so dass die Bibliothek aufgrund ihrer großen Verbreitung trotz ihres Java-Hintergrunds untersucht werden soll. Weitere Bibliotheken wie RLog [16], Pantheios [12] oder Boost.Log [2] sind ähnlich aufgebaut wie log4j und sollen daher nicht separat betrachtet werden.

Wie in syslog werden in log4j Nachrichten Prioritäten zugeordnet. Diese werden hier jedoch bereits bei der Nachrichtengenerierung ausgewertet, um so zu verhindern, dass Ressourcen verbraucht werden, obwohl die Nachrichten im Anschluss verworfen werden.

Die Verarbeitung der Nachrichten wird durch die Integration sogenannter *Appender* konfiguriert. Appender werden hier synchron in den Ablauf einer Benutzeranwendung eingebunden und ermöglichen einen hohen Grad an Flexibilität bei der Ausgabe und dem Versand von Nachrichten. Vorgefertigte Appender existieren beispielsweise zum Schreiben in Dateien oder Datenbanken, Versenden über Netzwerkverbindungen, zur Anbindung an das syslog-Protokoll und zur Zwischenspeicherung und anschließenden asynchronen Weitergabe an andere Appender. Andere Rechner als Zwischenstationen beim Nachrichtentransport sind nicht vorgesehen.

Keines der untersuchten Logging-Frameworks garantiert determiniertes Verhalten. Weiterhin müsste der Transport von Nachrichten über ein Netzwerk neu implementiert werden. Daher kommen auch existierende Logging-Frameworks hier nicht als Lösung in Frage. Das Konzept, lediglich Schnittstellendefinitionen bereitzustellen, und dadurch die Funktionalität den jeweiligen Anforderungen anpassen zu können, könnte allerdings in einer eigenen Implementierung berücksichtigt werden. Die Auswertung der Nachrichtenprioritäten auf Anwendungsseite ist nicht explizit gefordert, jedoch sinnvoll, um Ressourcen nicht unnötig zu verbrauchen.

3.1.3 Google glog

Google glog [4] schreibt Logging-Nachrichten wahlweise in Dateien oder auf den Fehlerausgabekanal `stderr`. Eine Weitergabe von Nachrichten über das Netzwerk ist nicht vorgesehen. Die Benutzerschnittstelle ist C++-Streams nachempfunden. Neben der Priorisierung der Nachrichten ist es außerdem möglich, die Ausgabe nur unter bestimmten Bedingungen auszuführen oder die maximale Anzahl der Ausgaben zu begrenzen.

Mit der *Raw Logging* Variante von glog können Nachrichten direkt auf `stderr` geschrieben werden, ohne dass Speicher alloziert oder Threads synchronisiert werden müssen. Damit wäre Determiniertheit garantiert. `stderr` kann jedoch auch von anderen Anwendungsteilen verwendet werden, so dass eine Umleitung des Fehlerausgabekanals – beispielsweise auf eine Netzwerkschnittstelle – nicht durchgeführt werden kann, ohne die Funktion von Benutzeranwendungen zu beeinträchtigen. Eine Verbreitung der Nachrichten über das Netzwerk ist folglich nicht möglich und glog somit hier nicht verwendbar.

3.1.4 AUTOSAR Diagnostic Services

AUTOSAR [1] ist ein offener Software-Standard der Automobilindustrie mit dem Ziel einheitliche Schnittstellen trotz der Vielfalt und Komplexität eingesetzter Baugruppen zu definieren. Unter anderem werden Schnittstellen zur Fehlerspeicherung (*Diagnostic Event Manager (DEM)*) und Diagnosekommunikation

(*Diagnostic Communication Manager (DCM)*) spezifiziert. Die Anforderungen an Zuverlässigkeit und Anwendungsneutralität sind bei sicherheitskrischen eingebetteten Systemen in Kraftfahrzeugen und medizinischen Geräten vergleichbar hoch.

Die Spezifikation sieht eine vollständige Abstraktion der Hardware und der Kommunikationswege vor, so dass Benutzern der Diagnosefunktionen nicht bekannt sein muss, wo die entsprechenden Module ausgeführt werden. Damit genügt der AUTOSAR-Standard den Anforderungen 2.1.3, 2.1.4 und 2.1.5. Sind Nachrichtenspeicher voll, werden alte Diagnosenachrichten entsprechend ihren Prioritäten verworfen, so dass auch Anforderung 2.1.7 erfüllt wird. Die Länge des Nachrichteninhalts kann variieren und zusätzlich mit ergänzenden Daten versehen werden.

Die erwähnte Abstraktionen werden jedoch nicht in den AUTOSAR Diagnostic Services implementiert, sondern bauen auf anderen AUTOSAR-Modulen auf. Ein Einsatz ist daher nur möglich, wenn der AUTOSAR-Standard systemweit eingehalten wird. In vorliegendem Fall ist dies nicht gegeben, so dass keine der existierenden AUTOSAR DEM und DCM-Implementierungen verwendet werden kann.

3.1.5 Ergebnis

Keine der untersuchten Logging-Bibliotheken kommt für einen Einsatz in vorliegendem Aufbau in Frage, da syslog und alle untersuchten Logging-Frameworks Randbedingung 2.2.2 missachten, google glog keinen Netzwerktransport bietet und AUTOSAR Diagnostic Services nur auf AUTOSAR-Plattformen funktioniert. D.h. es muss eine eigene Implementierung entwickelt werden, die alle Anforderungen und Randbedingungen erfüllt. Hierbei kann jedoch auf einige Lösungsansätze der vorgestellten Bibliotheken zurückgegriffen werden:

- Client/Server-Struktur
- Unterteilung in Nachrichten-*Header* und *Content*
- Flexibilität durch *Appender*/Erweiterungen
- Abstraktion der Kommunikationswege

Eine Client/Server-Struktur bedeutet die Aufteilung des Monitoring-Systems eines Rechners auf mehrere Prozesse. Damit ist – zusätzlich zur Kommunikation über Rechnergrenzen hinweg – lokale Interprozesskommunikation notwendig.

3.2 Lösungen zur lokalen Kommunikation

Im Folgenden sollen einige Mechanismen zur Interprozesskommunikation (*IPC*) untersucht werden. Da QNX das primäre Zielsystem ist, werden nur dort verfügbare Mechanismen betrachtet [14].

Die in Frage kommenden Implementierungen wurden bezüglich Datendurchsatz und Toleranz mehrerer gleichzeitiger Schreiber getestet. Es wurde jeweils die Zeit gemessen, die eine Implementierung zum Versenden bzw. Empfangen einer Nachricht mit fünf bzw. 800 Zeichen Inhalt benötigt. Soweit erforderlich wurden diese Versuche nicht nur mit einem, sondern auch mit acht konkurrierenden Schreibprozessen durchgeführt, während jedoch immer nur ein Leseprozess zum Einsatz kam. Unter QNX wurden alle schreibenden Prozesse auf der selben Prioritätsstufe gestartet, während der Leseprozess eine geringere Priorität erhielt und somit im Hintergrund der Benutzeranwendungen lief.

Die Zeitmessung wurde, wie in 3.1 gezeigt, auf einem ansonsten lastlosen System durchgeführt. Um zuverlässige Aussagen treffen zu können, wurden diese Tests je 8000 mal wiederholt.

```
1  ticks_before = clock.tickCounter();
2  send(message);
3  ticks_after = clock.tickCounter();
4  time = timeAt(ticks_after) - timeAt(ticks_before);
```

Quelltext 3.1: Zeitmessung (Pseudo-Code)

Die im Folgenden aufgeführten Diagramme stellen dar, welcher Anteil der 8000 Nachrichten innerhalb eines bestimmten Zeitraums versendet bzw. empfangen wurde. Da die Echtzeitanforderungen nur beim Versenden verpflichtend sind, wird der Nachrichtenempfang nur dann evaluiert, wenn der Versand bereits zufriedenstellend getestet wurde.

3.2.1 FIFO

Im POSIX-Standard werden Pseudodateien definiert, die zur gepufferten Übergabe von Daten an andere Prozesse verwendet werden können. Bei Leseoperationen werden die jeweils ältesten Daten zuerst zur Verfügung gestellt (FIFO = *First In – First Out*). Die Prioritäten der Nachrichten werden hierbei nicht berücksichtigt, wobei dies durch die Verwendung von einem FIFO-Puffer pro Prioritätsstufe kompensierbar wäre. Allerdings ist die Definition nicht Teil der Echtzeiterweiterung von POSIX, so dass die in 2.2.2 geforderte Determiniertheit nicht garantiert werden kann. Dies bestätigen die durchgeführten Messungen. (siehe Abbildung 3.1) Damit ist die Technik für diese Anwendung nicht brauchbar.

3.2.2 POSIX Message Queues

In der Echtzeiterweiterung des POSIX-Standards werden Datenpuffer ähnlich den zuvor beschriebenen FIFOs definiert. Hier können Nachrichtenprioritäten berücksichtigt werden, so dass eine Leseoperation jeweils die Nachricht mit der höchsten Priorität liefert; befindet sich mehr als eine Nachricht dieser Prioritätsstufe im Puffer, so wird die älteste von diesen bereitgestellt.

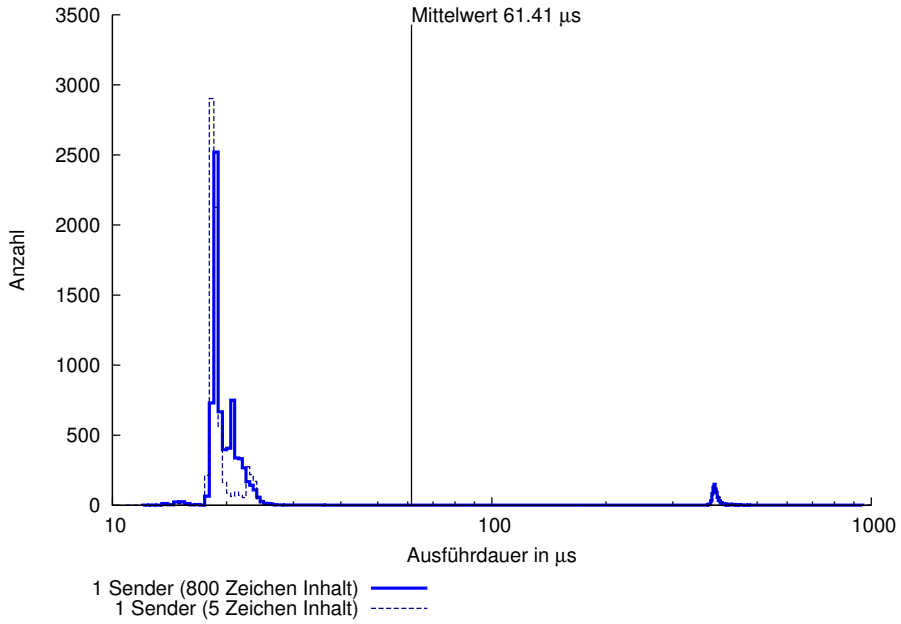


Abbildung 3.1: Nachrichtenversand über FIFO

Unter QNX können diese Message Queues auf zwei verschiedene Weisen eingebunden werden: Entweder mit Hilfe eines Prozesses auf Benutzerebene oder durch direkten Zugriff auf einen Puffer innerhalb des Kernels. Wird die erste Variante gewählt, erfolgen bei jedem Zugriff auf die Queue zwei Kontextwechsel: zum verwaltenden Prozess und zurück zur Anwendung. Dieser Overhead soll durch die alternative Variante vermieden werden. Es ist jedoch auch hier ein zusätzlicher Prozess notwendig, der die Queues im Kernel anlegt und bei Bedarf aufräumt. Kontextwechsel sind hier jedoch nur beim Erzeugen und Löschen eines Zugriffspunktes notwendig.

Eine Gegenüberstellung beider Implementierungen ist in den Abbildungen 3.2 und 3.3 zu finden. Der Geschwindigkeitsvorteil der direkten Zugriffsmethode ist vor allem bei mehreren konkurrierenden Schreibprozessen offensichtlich. Dies ist darauf zurückzuführen, dass bei einem Kontextwechsel vom Verwaltungsprozess zurück zum Schreiber nicht immer der Prozess wieder aufgenommen wird, der für den Aufruf des Dienstes verantwortlich war, sondern auch ein anderer Prozess der selben Priorität gestartet werden kann. Scheduling von Prozessen derselben Prioritätsstufe erfolgt unter QNX entweder nach FIFO oder dem Round-Robin Prinzip. In beiden Fällen ist es unwahrscheinlich, dass ein Prozess, der die Kontrolle der CPU an einen anderen Prozess abgegeben hat, als nächstes ausgeführt wird. Dies wäre nur der Fall, wenn kein anderer Prozess gleicher oder höherer Priorität zu diesem Zeitpunkt lauffähig ist. Der Erwartungswert von 1,53 µs sowie eine maximale Ausführdauer von 12 µs bei direktem Kernel-Zugriff genügen jedoch den geforderten Bedingungen weitgehend.

3.2.3 Shared Memory

Shared Memory bietet den IPC-Mechanismus mit der höchsten Bandbreite. Hier wird ein Speicherbereich in den Adressraum aller Prozesse eingeblendet, die sich bei einem Shared Memory Objekt registrieren. Ab dem Registrierungszeitpunkt erfolgt der Zugriff wie auf lokal allozierten Speicher. Allerdings bietet Shared Memory noch keinerlei Datenverwaltung. Diese muss zusätzlich implementiert werden. Die bereits erwähnten Anforderungen Determiniertheit, schneller Eintragvorgang, Toleranz bezüglich Nebenläufigkeit und Prioritäten berücksichtigendes FIFO-Verhalten sind durch die Datenstruktur zu gewährleisten. Messungen der Dauer von Lese- und Schreibzugriffen ergaben Werte von deutlich unter 1 µs. (Siehe Abbildung 3.4.)

3.2.4 Message Passing Interface (MPI)

MPI ist der native Nachrichtenübertragungsstandard unter QNX und Basis der meisten anderen IPC-Verfahren. Typenlose Daten beliebiger Größe werden hierbei direkt – ohne Zwischenspeicherung – in den Adressraum eines anderen Prozesses kopiert. Die Übertragungsrate ist dementsprechend nur durch die darunter liegende Hardware begrenzt. Der Transfer zwischen den Prozessen erfolgt synchron nach dem *Send→Receive→Reply*-Prinzip. Dies impliziert, dass ein Sender blockiert, bis der Empfangsprozess eine Antwort geschickt hat. Da der Verbindungsaufbau über statische Kanäle erfolgt, ist für jeden Sender eine separate Ankopplung an den Empfänger notwendig. Um nun schnellstmöglich Empfangsbestätigungen zu versenden, könnten auf Empfängerseite dedizierte Threads diese Verbindungen verwalten. Dies würde auf Grund der Kombination der erforderlichen Priorisierung der Nachrichten und der Prioritäten der Sendeprozesse, sowie der abzugebenden Garantie einer Antwort innerhalb einer festgelegten Zeitspanne komplex, und damit fehlerträchtig. Deshalb soll diese IPC-Form hier nur Referenzzeiten liefern.

Auf dem Testsystem benötigte die Sendefunktion bis zu ihrer Rückkehr – also der Antwort des Empfängers – reproduzierbar zwischen 1,60 und 1,65 µs. Der Geschwindigkeitsvorteil beim Transfer einer Nachricht mit fünf Zeichen Inhalt gegenüber einer Nachricht mit 800 Zeichen liegt im Bereich von 0,05 µs, ist also vernachlässigbar gering.

3.2.5 Vergleich

Wie eingangs erwähnt ist die Zeit, die ein Benutzerprozess zum Versand einer Nachricht benötigt, das wichtigste Kriterium bei der Auswahl eines geeigneten IPC-Verfahrens. Da diese Zeiten auf Grund diverser Einflüsse – z.B. dem Zustand des Prozessor-Caches, Pipelining, Seitenfehlern, unterschiedlichen Nachrichtengrößen, der Auslastung des Systems, Unterbrechungen, usw. – stark schwanken können, sind prinzipiell zwei Werte von Interesse: Ein Erwartungswert, der die im Normalfall zu erwartende Zeit angibt. Er kann durch Mittelwertbildung genügend vieler Messungen errechnet werden. Hier wurden alle Werte der im Vorangegangenen gezeigten Untersuchungen verwendet. Die Ergebnisse sind den jeweiligen Diagrammen oder Tabelle 3.1 zu entnehmen.

Weniger einfach zu ermitteln ist die maximale Ausführungszeit, *Worst Case Execution Time* (WCET) genannt. Hier muss der ungünstigste Fall ermittelt werden, also unter Berücksichtigung der erwähnten Einflüsse. Dies ist analytisch bereits bei geringer Komplexität nur noch schwer machbar. Müssen Faktoren wie Abhängigkeiten von anderen Prozessen auf Mehrkernprozessoren berücksichtigt werden, ist das auch mit sehr hohem Aufwand praktisch nicht mehr möglich. Ein ungefährer Vergleichswert kann jedoch ebenfalls durch genügend häufige Messungen unter verschiedenen Bedingungen ermittelt werden. Es ist bei dieser Methode allerdings nicht garantiert, dass die ermittelte WCET der tatsächlichen entspricht. Für die Ergebnisse der durchgeführten Messungen siehe Tabelle 3.1. Zu genaueren Informationen zur WCET-Ermittlung sei auf [54] verwiesen.

IPC-Mechanismus	Parameter	Mittelwert	WCET
MPI	– –	1,60 µs	– –
FIFO	– –	61,41 µs	– –
Message Queues	User-Level	15,66 µs	70 µs
Message Queues	Kernel-Access	1,53 µs	11,3 µs
Shared Memory	Keine Datenstruktur	0,23 µs	0,54µs

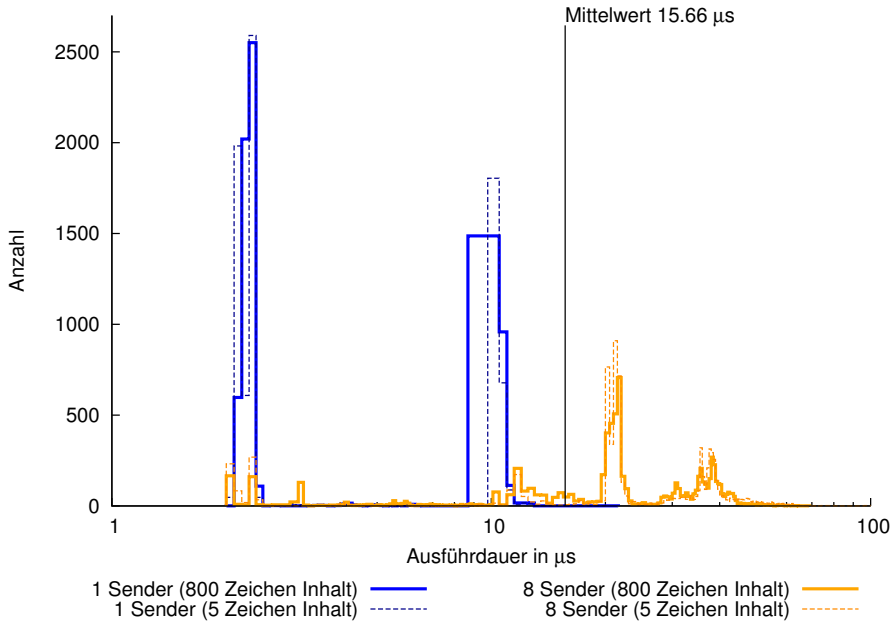
Tabelle 3.1: Erwartungswerte der Ausführdauer und WCETs (wenn determiniert) der Sendefunktionen verschiedener IPC-Mechanismen

3.3 Lösungen zur Netzwerkkommunikation

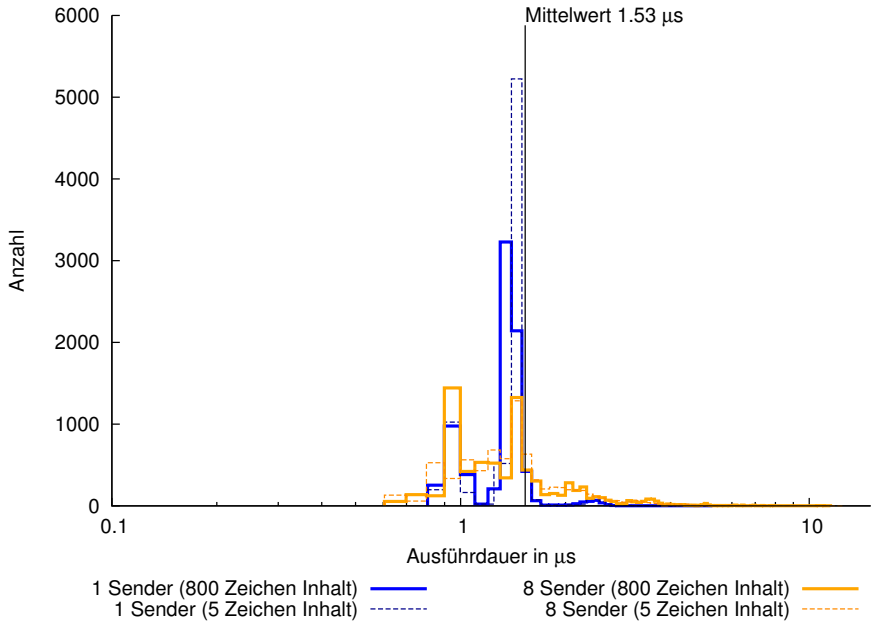
Zur rechnerübergreifenden Interprozesskommunikation werden üblicherweise Transportprotokolle als Abstraktion der Netzwerk-Hardware verwendet. Einige in Frage kommenden Protokolle sollen im folgenden Abschnitt kurz beschrieben und analysiert werden. Kriterien sind dabei Zuverlässigkeit, Overhead, Ausnutzung der Bandbreite und Verfügbarkeit auf verschiedenen Systemen. Overhead, der durch Protokolle ober- oder unterhalb der Transportschicht generiert wird, wird hier nicht betrachtet, da dieser für die meisten Transportprotokolle identisch ist, und somit für einen Vergleich vernachlässigt werden kann.

3.3.1 Transmission Control Protocol (TCP)

TCP [44] ist ein Protokoll, das eine zuverlässige Übertragung von Daten beliebiger Größe zwischen zwei Punkten (spezifiziert durch Adressen und Ports) ermöglicht. In ihm werden viele Probleme komplexer Netzwerkstrukturen adressiert und umgangen (z.B. Paketverlust, Flusssteuerung und Überlaststeuerung). Damit ist TCP geeignet zuverlässige Verbindungen in großen, unzuverlässigen Netzwerken aufzubauen. Es bildet die Grundlage der meisten Internetdienste und wird dementsprechend von allen gängigen Systemen unterstützt. (Siehe [53] Abschnitt 6.5.) Besonders zur Übertragung von Datenpaketen, die nicht in einem IP-Paket übertragen werden können, ist TCP geeignet, da diese automatisch auf mehrere Pakete aufgeteilt und beim Empfänger wieder zusammengesetzt werden.

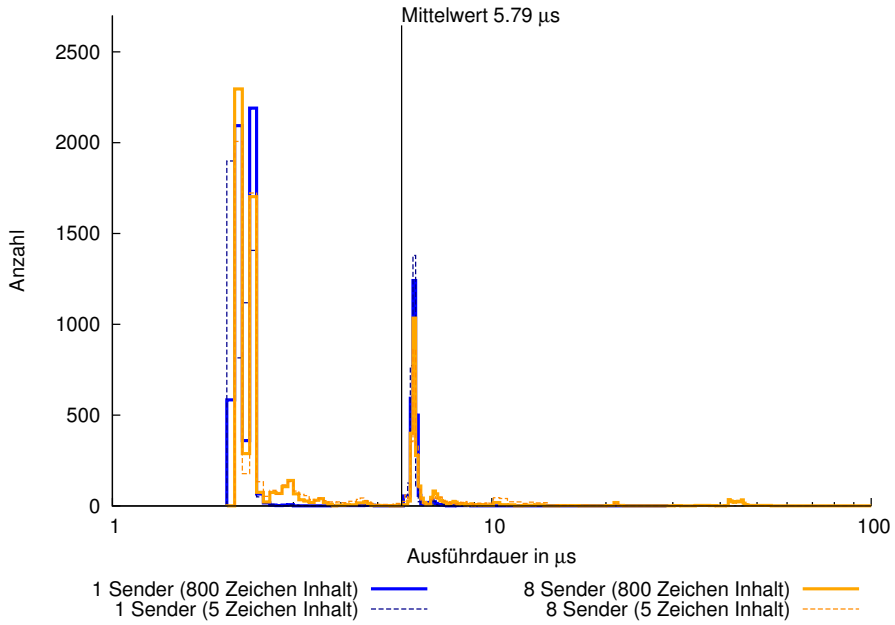


(a) externer Verwaltungsprozess

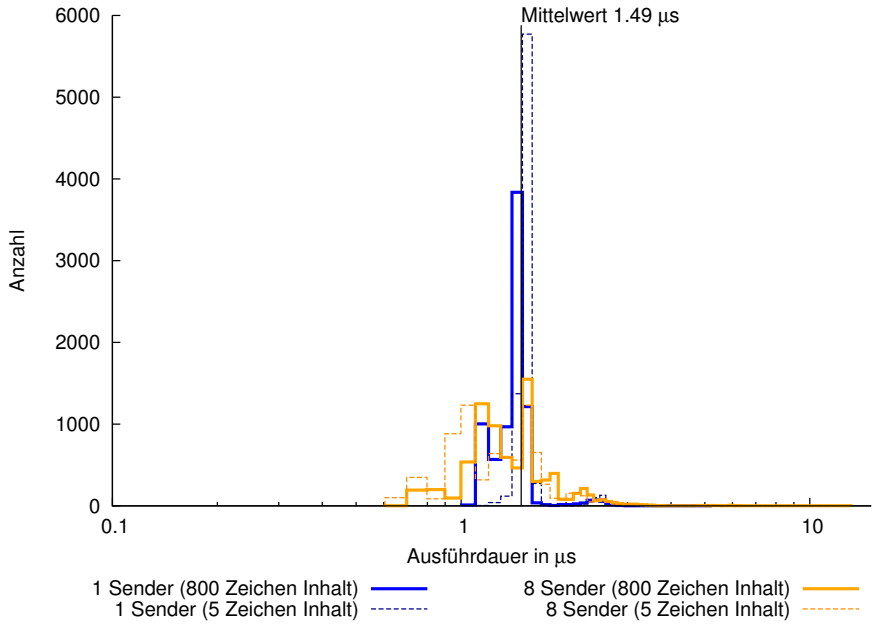


(b) direkter Kernel-Zugriff

Abbildung 3.2: Nachrichtenversand über POSIX Message Queues

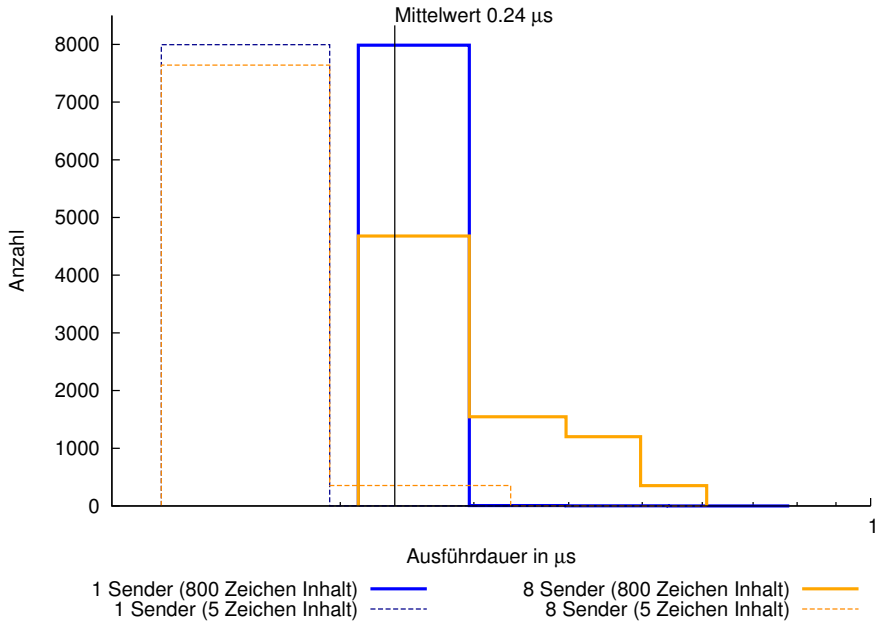


(a) externer Verwaltungsprozess

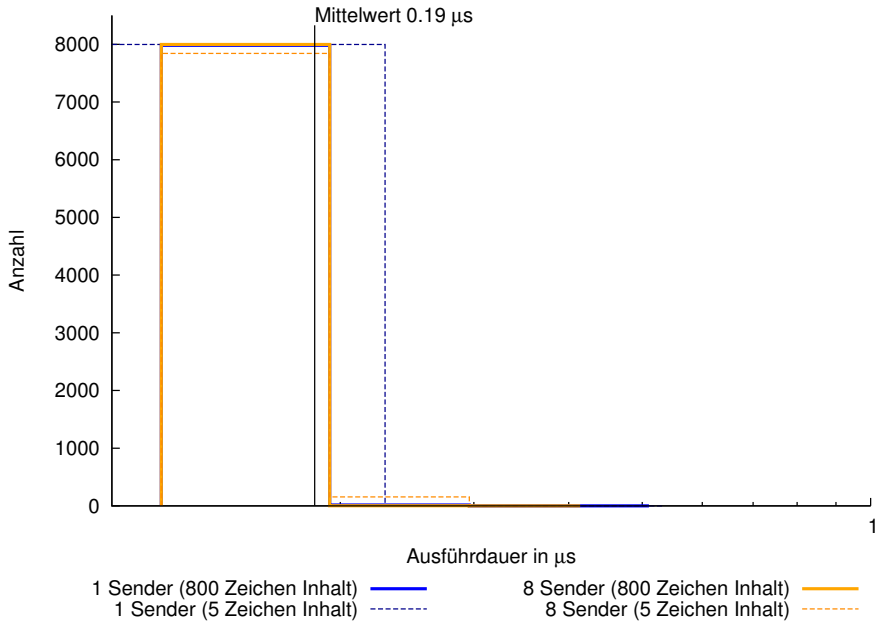


(b) direkter Kernel-Zugriff

Abbildung 3.3: Nachrichtenempfang über POSIX Message Queues



(a) Schreiben



(b) Lesen

Abbildung 3.4: Shared Memory

Auch der Verlust einzelner Teile wird durch erneute Übertragung kompensiert. Dies wird durch Empfangsbestätigungen der Teile ermöglicht.

Zusatzfunktionen benötigen allerdings zusätzliche Ressourcen: Empfangsbestätigungen benötigen Netzwerkbandbreite, Informationen über die richtige Zusammensetzung fragmentierter Pakete müssen im Paket-Header gespeichert werden, besondere Nachrichten wie zum Verbindungsauf- oder -abbau müssen durch entsprechende Felder im Header kenntlich gemacht werden, etc. Damit ergibt sich ein Overhead von mindestens 20–60 Bytes (je nach gewählten Zusatzoptionen) pro Teilpaket.

In vorliegendem Fall sollen jedoch nur Datenpakete versendet werden, deren Größe unter der eines IP-Pakets liegt. Außerdem ist das Institutsnetzwerk im Gegensatz zum Internet vergleichsweise klein, so dass komplizierte Fluss- und Überlaststeuerungen nicht notwendig sind. Weiterhin wurde spezifiziert, dass der Verlust einzelner Nachrichtenpakete zulässig ist, solange dies erkannt wird. Damit sind die Zusatzfunktionen, die TCP bietet, nicht erforderlich, die Nachteile, wie zusätzliche Informationen im Header und erhöhte Netzlast, bleiben jedoch erhalten.

3.3.2 User Datagram Protocol (UDP)

UDP [42], das zweite wichtige Standardtransportprotokoll des Internets, ist wesentlich einfacher aufgebaut als TCP. Es bietet weder Garantien bezüglich der Zustellung von Datenpaketen, noch Fluss- oder Überlaststeuerung. Übertragungen erfolgen verbindungslos, d.h. ohne vorherigen Aufbau eines Kanals. Daten werden auf der Transportebene nicht automatisch fragmentiert, so dass nur Pakete begrenzter Größe versandt werden können.

Daraus ergibt sich ein Minimum an Overhead: Im 8 Byte großen Header werden nur Zielport und eine Prüfsumme gespeichert. Durch die Prüfsumme können bei der Übertragung aufgetretene Fehler erkannt werden, so dass ein UDP-Paket entweder korrekt oder gar nicht beim Empfänger ankommt. Außer den Datenpaketen werden keine zusätzlichen Nachrichten verschickt. Dadurch ergibt sich eine gute Ausnutzung der verfügbaren Bandbreite bei geringen Verzögerungen.

3.3.3 Stream Control Transmission Protocol (SCTP)

SCTP [50] soll die Vorteile von TCP und UDP vereinen. Es arbeitet dabei verbindungsorientiert und zuverlässig wie TCP, sendet aber Datenpakete wie UDP. Diese Datenpakete setzen sich aus *chunks* zusammen, so dass mehrere (Steuerungs-)Nachrichten in einem Paket zusammengefasst werden können. Auch kann die strenge Einhaltung der Reihenfolge verschiedener Pakete deaktiviert werden, wodurch Pakete nicht unnötig verzögert werden. Das Problem des *head-of-line blocking* [33] von TCP wird in SCTP durch die Verwendung mehrerer paralleler Datenströme umgangen, so dass die Latenz weiter reduziert werden kann. QNX bietet SCTP-Unterstützung ab Version 6.3.0; Linux ab Kernel Version 2.6 [6].

Die funktionalen Eigenschaften von TCP bleiben weitgehend erhalten, so dass auch der entsprechende Overhead anfällt, wenn auch etwas geringer als bei TCP. Die Header-Größe setzt sich aus einem Paket-Header (12 Bytes) und den Headern der einzelnen chunks (je 4 Bytes) zusammen. Da alle chunks eines Pakets dem selben Prozess (bzw. dessen Port) zugestellt werden, ist in einem Paket auch üblicherweise nur ein chunk mit Nutzdaten enthalten. Wird nur dieser Teil betrachtet, besitzt ein SCTP-Paket einen Header von mindestens 16 Bytes. Wie bei TCP wird auch bei SCTP zusätzlicher Netzwerkverkehr durch Überlast- und Flusskontrolle, Empfangsbestätigungen etc. generiert. Damit ist SCTP zwar etwas besser für vorliegende Anwendung geeignet als TCP, jedoch auf Grund des Overheads immer noch weniger gut als UDP.

3.3.4 Datagram Congestion Control Protocol (DCCP)

DCCP [38] erweitert UDP um Staukontrollmechanismen. Dabei arbeitet DCCP verbindungsorientiert mit Steuerungsnachrichten, die gleichzeitig Nutzdaten übertragen können. Garantien bezüglich der Nachrichtenzustellung werden nicht gegeben. Linux unterstützt DCCP ab Kernel Version 2.6.14 [3]; Für QNX existiert noch keine Implementierung.

Auch bei DCCP entsteht ein größerer Overhead als bei UDP (12 oder 16 Bytes großer Header und zusätzliche Steuerungsnachrichten). Da Staukontrolle jedoch eingesetzt wird um einer Überlastung des Netzwerks vorzubeugen, wäre dieser Overhead akzeptabel. Die fehlende Unterstützung unter QNX schließt den Einsatz allerdings momentan aus.

3.3.5 QNX Native Networking (Qnet)

Qnet [13] ist eine netzwerktransparente IPC-Form unter QNX. Es ist einstellbar, ob die korrekte Übertragung der Daten garantiert werden soll oder nicht; eine Staukontrolle findet nicht statt. Zwei Qnet-Varianten müssen unterschieden werden: Die erste Variante setzt – wie die zuvor aufgeführten Transportprotokolle – auf dem IP-Protokoll auf; die zweite Variante direkt auf dem Ethernet-Protokoll. Die zweite Variante ist schneller als die erste, unterstützt jedoch kein Routing über lokale Netzwerkgrenzen hinaus. Da dies in vorliegendem Fall nicht notwendig ist, soll hier nur die direkt auf Ethernet aufsetzende Variante weiter betrachtet werden.

Der Header besteht aus mindestens 56 Bytes und zusätzlichen Namenseinträgen variabler Länge [23]. Dies kann jedoch nicht direkt mit den Header-Größen der anderen Protokolle verglichen werden, da hier der Header des IP-Protokolls (zwischen 20 und 60 Bytes) eingespart wird. Da auch keine Staukontrolle implementiert ist, ergeben sich keine deutlichen Vorteile gegenüber UDP, die eine systemspezifische Implementierung rechtfertigen würden.

3.3.6 Vergleich

Die vorgestellten Protokolle lassen sich grob anhand zweier Merkmale einteilen: Zuverlässigkeit und Fluss- bzw. Staukontrolle. (Siehe Tabelle 3.2.) Beide Eigenschaften bringen Overhead in Form von Daten im Protokoll-Header und

Belastung des Netzwerks mit Steuerungsnachrichten mit sich. Bei der Auswahl eines geeigneten Protokolls muss deshalb abgewogen werden, ob die zusätzlichen Funktionen tatsächlich mehr Vor- als Nachteile bringen.

Staukontrolle \ Zuverlässigkeit	ja	nein
ja	TCP	DCCP
nein	SCTP, Qnet	UDP

Tabelle 3.2: Vergleich verschiedener Transportprotokolle

3.4 Zusammenfassung

Es wurden einige gängige Logging-Mechanismen auf Übereinstimmungen mit den gegebenen Randbedingungen und Anforderungen untersucht. Da keine dieser Mechanismen alle geforderten Eigenschaften erfüllt, wurden die Prinzipien herausgegriffen, die auch in einer eigenen Implementierung hilfreich sein können.

Eines dieser Prinzipien ist die Trennung von Nachrichtengenerierung und -transport. Da damit lokale Interprozesskommunikation zwischen Clients und Servern notwendig ist, wurden die unter QNX verfügbaren Alternativen untersucht, ob sie die Einschränkungen einer Echtzeitumgebung einhalten und wie lange sie zur Verarbeitung einer Nachricht benötigen. Es zeigte sich, dass Shared Memory deutlich schneller als die anderen untersuchten Mechanismen ist, jedoch noch eine Struktur zur Verwaltung implementiert werden muss.

Zum Transport der Nachrichten über ein Netzwerk soll ein Standardprotokoll eingesetzt werden. Daher wurden TCP, UDP, SCTP, DCCP und Qnet betrachtet. Die Analyse ergab eine Unterteilung der Protokolle nach den Kriterien Zuverlässigkeit und Staukontrolle. Beide Funktionen erzeugen allerdings Overhead. Bei der Auswahl eines Protokolls soll daher darauf geachtet werden, nur geforderte Funktionen einzusetzen und ein ansonsten möglichst einfaches Protokoll zu bevorzugen.

Kapitel 4

Konzept

Die vorliegende Problematik lässt sich in drei Kernfragestellungen aufgliedern:

- Welche Daten sind für eine Monitoring-Umgebung von Interesse?
- Wie greift ein Benutzer auf die Daten zu?
- Wie werden die Daten zu den Ausgabestationen transportiert?

Durch die Aufteilung der Problemstellung ist ein modularer Ansatz durchsetzbar. Eine Spezialisierung dieses Prinzips auf die Entwicklung von Kommunikationsprotokollen ist das OSI-Schichtenmodell [19]. Einer der Vorteile des Modells ist, dass Instanzen einzelner Schichten einfach an neue Anforderungen angepasst bzw. ersetzt werden können, ohne dass Änderungen der anderen Schichten notwendig wären. Aus diesem Grund ist es möglich in vorliegender Arbeit nur die obersten drei Schichten zu implementieren, während für darunter liegende Schichten Standardimplementierungen verwendet werden können. (Siehe Tabelle 4.1.) Damit bleibt die Monitoring-Infrastruktur unabhängig von der verwendeten Hard- und Software.

#	*Schicht*	*Entsprechung in Monitoring-Infrastruktur*
7	Anwendungsschicht	Benutzerschnittstelle
6	Darstellungsschicht	Datenformat
5	Kommunikationsschicht	Übertragungsinfrastruktur
4	Transportschicht	
3	Netzwerkschicht	
2	Sicherungsschicht	
1	Bitübertragungsschicht	

Tabelle 4.1: OSI-Schichtenmodell mit Entsprechungen in Monitoring-Infrastruktur

4.1 Benutzerschnittstelle

> The easiest programs to use are those that demand the least new learning from the user – or, to put it another way, the easiest programs to use are those that most effectively connect to the user's pre-existing knowledge.
>
> *Rule of Least Surprise* von Eric Raymond [45]

Benutzer müssen nur zwei Stellen der Monitoring-Infrastruktur kennen: Zum einen die Schnittstelle zur Generierung von Monitoring-Nachrichten und ihrer Veröffentlichung (*Nachrichteneingang*) und zum anderen die Schnittstelle zur Darstellung dieser am Ausgabepunkt (*Nachrichtenausgang*).

Um den Nachrichteneingang intuitiv nutzbar zu gestalten, ist es sinnvoll, eine Abstraktion der Monitoring-Infrastruktur in Anwenderprogramme einzublenden und Nachrichten in Form von Objekten darzustellen [46]. Damit kann die Semantik eines Ausdrucks wie "Nachrichten an die Monitoring-Infrastruktur übergeben" im Quelltext beibehalten werden. Die notwendigen Komponenten werden in Form einer C++-Bibliothek (der *Monitoring-Bibliothek*) bereitgestellt und können so in beliebige Anwendungen integriert werden.

Die Funktionalität des Nachrichtenausgangs soll frei definierbar sein. Dabei soll ein Benutzer nur angeben müssen, *wie* Nachrichten ausgegeben werden, nicht jedoch *wann*. Der Nachrichtenausgang ist daher als eigenständiger Prozess zu implementieren, der vom Benutzer durch Funktionen zur Ausgabe der Nachrichten erweitert werden kann. Durch das Prinzip der Steuerungsumkehr – *inversion of control* oder *dependency inversion prinicple* [40] – ist es möglich, diese vom Benutzer erzeugten Funktionen so in den Kontrollfluss der Monitoring-Infrastruktur zu integrieren, dass diese für jede eingehende Nachricht ausgeführt werden. Der Benutzer muss somit nur für die Funktionalität sorgen, nicht für die Ausführung. Nach demselben Prinzip sollen auch frei definierbare Filter vor die Ausgabe geschaltet werden können.

4.2 Datenformat

Entsprechend 2.1.2 sollen Nachrichten einerseits frei formulierbare Inhalte und andererseits definierte Informationen zum Kontext ihrer Generierung enthalten. In Anlehnung an das syslog-Protokoll [32] werden Nachrichten daher in *Content*- und *Header*-Bereiche unterteilt.

Obwohl im Voraus nicht bekannt sein kann, wie lang der Text sein wird, den ein Benutzer versenden möchte, soll dynamische Speicherallozierung nicht verwendet werden. Dies würde Randbedingung 2.2.2 verletzen, da Speicherallozierung üblicherweise ein nicht-determinierter Vorgang ist. Der Speicher für den Content-Bereich soll deshalb bei der Nachrichtenerzeugung fest zugewiesen und nicht mehr in der Größe verändert werden.

In Abschnitt 2.1.2 wird gefordert, dass auch Informationen über den (physikalischen und logischen) Ort und den Zeitpunkt der Nachrichtengenerierung zu übertragen sind. Metadaten dieser Art werden in einem vom Content getrennten Teil – dem Header – gespeichert. Auch für die Übertragung erforderliche Daten sind im Header untergebracht: Beispielsweise wird jeder Nachricht eine Dringlichkeit zugewiesen, um so wichtige Nachrichten bei der Übertragung bevorzugen zu können. Da das Format der Header-Daten definiert ist, können auch die entsprechenden Speicherbereiche in ihrer Größe festgelegt werden.

Eine Nachricht besteht demnach aus einem Header- und einem Content-Bereich, die beide definierte Speicherbereiche belegen.

4.3 Übertragungsinfrastruktur

4.3.1 Aufbau

Bei der Verbreitung der Monitoring-Nachrichten soll die Modularität des Gesamtsystems erhalten bleiben. So sollen Anwendungsgruppen auch bei Änderungen des Gesamtsystems nicht neu konfiguriert werden müssen. Daher ist vorgesehen, dass jedes Modul lediglich seine direkten Nachbarn im Netzwerk kennen muss. Die Nachrichten sollen von jedem Netzwerkknoten aus an alle Nachbarn verteilt werden. Dadurch können alle Informationen des Gesamtsystems an beliebiger Stelle ausgelesen werden.

Da innerhalb einer Benutzeranwendung nicht festgelegt sein muss, ob Nachrichten lokal oder auf einem anderen Rechner ausgegeben werden (siehe 2.1.4), muss die Schnittstelle zwischen Anwendung und Ausgabeprozess in beiden Fällen identisch sein. Bietet ein Ausgabeprozess nicht nur die Möglichkeit der lokalen Ausgabe von Nachrichten, sondern wird dieser um eine Funktion zur Weiterleitung von Nachrichten zu Ausgabeprozessen anderer Netzwerkknoten erweitert, kann das Problem unbekannter Ausgabepunkte aus der Anwendung herausgenommen und in den Ausgabeprozessen behandelt werden . Dies erfordert von Ausgabeprozessen demzufolge die Möglichkeit Nachrichten zu versenden sowie Empfangsfunktionalität, um Nachrichten anderer Rechner verarbeiten zu können. Ein Prozess, der Nachrichten ausgeben und über eine Netzwerkschnittstelle versenden und empfangen kann, wird als *Server* bezeichnet. *Clients* sind alle Prozesse, die Nachrichten an einen Server übergeben.

Ein Server kann also sowohl Nachrichten von Clients des selben Netzwerkknotens, als auch Nachrichten von anderen Servern empfangen. Diese empfangenen Nachrichten können anschließend ausgegeben und/oder zu Servern anderer Netzwerkknoten weitergeleitet werden. Werden Nachrichten nun von den Servern nicht mehr direkt zum Ausgabepunkt gesendet, sondern über andere Server geleitet, genügt es, wenn jeder Server den nächsten Knoten Richtung Ausgabepunkt kennt. Damit sind Änderungen in der Netztopologie einfach, da jede Änderung nur an einem Netzwerkknoten durchgeführt werden muss.

Ein Beispielaufbau eines so konfigurierten Netzes ist in Abbildung 4.1 zu sehen. Hervorzuheben ist, dass jeder Nachrichtentransport über Rechnergrenzen hinaus ausschließlich über Server-Prozesse erfolgt.

4.3.2 Lokale Kommunikation

Wie erwähnt, ist zur Kommunikation zwischen Clients und Server lokale Interprozesskommunikation notwendig. In 3.2 wurden bereits in Frage kommende Mechanismen untersucht. Die Ergebnisse aus Tabelle 3.1 bringen zwei Mechanismen in die engere Auswahl: POSIX Message Queues mit direktem Kernel-Zugriff und Shared Memory. Beide bieten asynchrone Schreib- und Leseoperationen, gute Datendurchsatzraten und determiniertes Verhalten. Der hohe Datendurchsatz des Shared Memory ist aufgrund der fehlenden Datenstruktur nicht direkt mit dem der Message Queues vergleichbar. Da der Unterschied jedoch sehr deutlich ausfällt, verspricht eine Erweiterung des Shared Memory um eine Datenstruktur immer noch gute Ergebnisse.

Eine eigene Implementierung hat zudem den Vorteil, dass sie genau an die Anforderungen angepasst werden kann. So kann beispielsweise bei Message Queues nicht garantiert werden, dass immer Speicher für hochpriore Nachrichten reserviert bleibt, da dies nicht Teil des Standards ist. Bei der Implementierung des IPC über Shared Memory kann dies berücksichtigt werden.

4.3.3 Netzwerkkommunikation

Zur Kommunikation von Server zu Server sollen Standardtransportprotokolle eingesetzt werden. In Abschnitt 3.3 wurden bereits einige Möglichkeit untersucht, diese stellen jedoch nur eine Auswahl für den konkreten Anwendungsfall Ethernet dar. Da die Monitoring-Infrastruktur jedoch auch für andere Netzwerkarchitekturen offen gehalten werden soll (siehe 2.1.5) und auch bei einheitlicher Technik verschiedene Eigenschaften benötigt werden können, wird die Wahl des Protokolls den Benutzern überlassen.

4.4 Zusammenfassung

In diesem Kapitel wurde ein modulares Konzept zur Realisierung der gesteckten Ziele unter Einhaltung der Randbedingungen entwickelt. Die Benutzerschnittstelle soll zur einfachen Nutzbarkeit an existierende Schnittstellen mit ähnlicher Funktionalität angelehnt werden und dem Prinzip der objektorientierten Programmierung folgen.

Weiterhin wurde festgelegt, dass Nachrichten in Header- und Content-Bereich unterteilt werden: Im Header werden Metadaten zur Beschreibung der Nachricht gespeichert; im Content-Bereich Nutzdaten.

Ausgehend von dem Client/Server-Prinzip wurde eine Kommunikationsstruktur eingeführt, die einfache Änderungen der Kommunikationswege ermöglicht. Für die lokale Kommunikation zwischen Clients und Server soll eine Implementierung auf Shared Memory Basis entwickelt und mit den existierenden

Mechanismen verglichen werden. Zur Kommunikation zwischen Servern sollen von Benutzern wähl- und erweiterbare Standardprotokolle eingesetzt werden.

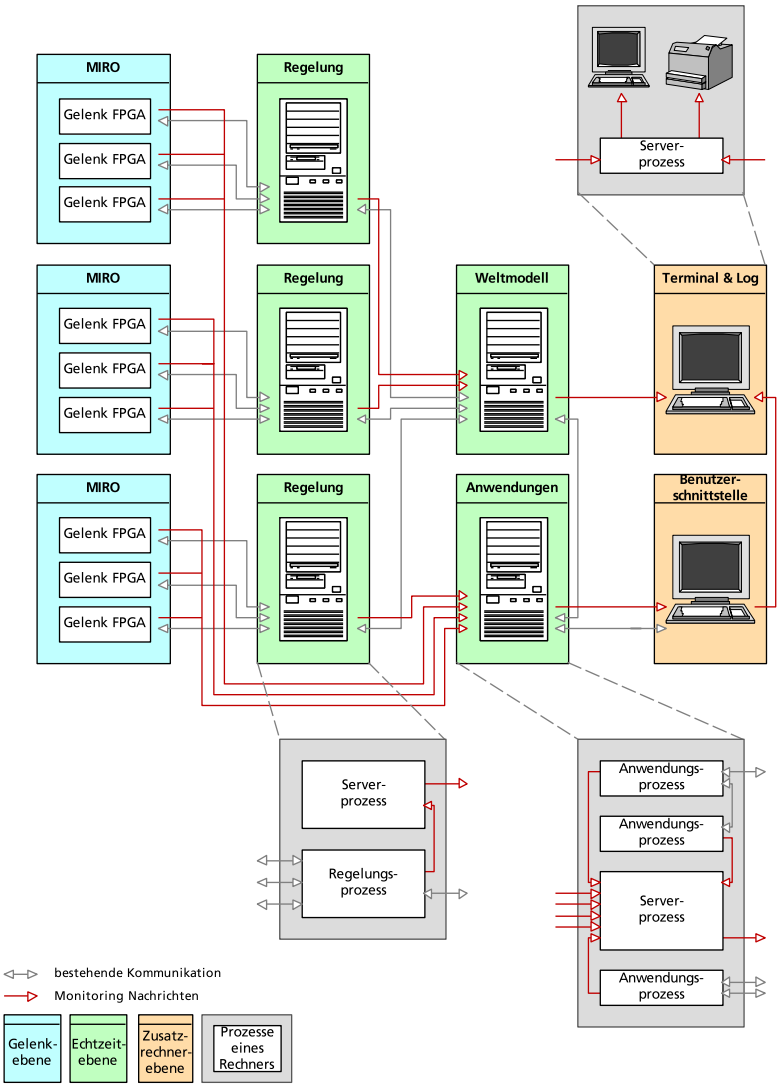


Abbildung 4.1: Kommunikationswege eines Beispielaufbaus

Kapitel 5

Ausarbeitung des Konzepts

In diesem Kapitel wird das in 4 erarbeitete Konzept konkretisiert. Es wird zunächst der Aufbau einer Nachricht erläutert. Anschließend wird die Übertragungsinfrastruktur aufgebaut, wobei Client, Server und die möglichen Kommunikationswege getrennt voneinander betrachtet werden sollen. Zuletzt werden die Benutzerschnittstellen von Client und Server beschrieben.

5.1 Datenformat

Wie bereits in Abschnitt 4.2 erwähnt, sind die Nachrichten in Header und Content unterteilt. Der Header wird für jede Nachricht automatisch vom System generiert und danach nicht wieder verändert. Lediglich zwei Einträge, die den Inhalt der Nachricht kategorisieren, müssen vom Anwender angepasst werden. Die eigentlichen Nutzinformationen werden vom Benutzer festgelegt und im Content-Bereich gespeichert. Da das System nur institutsintern eingesetzt wird und deshalb die mutwillige Störung des Betriebs durch Manipulation der Nachrichten ausgeschlossen werden kann, wird auf eine Integritätsprüfung zu Gunsten der Verarbeitungsdauer verzichtet.

5.1.1 Header

Im Folgenden werden die einzelnen Header-Einträge aufgelistet und kurz erläutert. Für eine Übersicht siehe Abbildung 5.1.

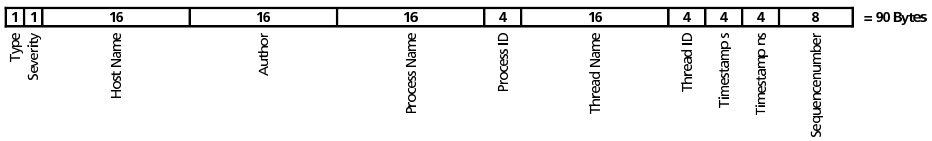


Abbildung 5.1: Speicheraufteilung des Headers

5.1.1.1 Typ

Der Typ einer Nachricht wird durch einen vordefinierten Schlüssel vom Benutzer spezifiziert und dient zur Einordnung und späteren Filterung der

Nachricht. Die Typen sind an die *Facilities* des syslog-Protokolls angelehnt, wurden jedoch an die Eigenschaften eines mechatronischen Systems angepasst. (Siehe Tabelle 5.1.)

TYPE_APP	Anwendung
TYPE_SYNCHRONIZATION	Synchronisierung verschiedener Rechner und Prozesse
TYPE_COMMUNICATION	Kommunikation
TYPE_CONTROL	Regelung
TYPE_HARDWARE	Hardware
TYPE_MONITOR	Monitoring-System
TYPE_HAL	Hardware-Abstraktionsschicht

Tabelle 5.1: Nachrichtentypen des Monitoring-Systems

5.1.1.2 Dringlichkeit

Da das Monitoring-System im Hintergrund der eigentlichen Anwendungen läuft, kann nicht garantiert werden, dass alle Nachrichten innerhalb einer festen Zeitspanne beim Ausgabepunkt ankommen. Verschiedene Nachrichten besitzen jedoch naturgemäß auch verschiedene Dringlichkeitsstufen. Um nun die Chance einer wichtigen Nachricht schnell zugestellt zu werden zu erhöhen, werden den Nachrichten Dringlichkeitsstufen zugewiesen. Diese Dringlichkeitsstufen bildet das Monitoring-System auf Prioritäten ab, die die Wahrscheinlichkeit einer schnellen Verbreitung der Nachrichten beschreiben.

Die möglichen Werte der Dringlichkeit wurden an die *Severity*-Stufen des Syslog-Protokolls (siehe [32]) angelehnt. Jedoch wurden ähnliche Stufen zusammengefasst um zu verhindern, dass unterschiedliche Begriffsauffassungen verschiedener Anwendungsentwickler zu falscher Priorisierung führen können. Daraus ergeben sich die Dringlichkeitsstufen aus Tabelle 5.2.

5.1.1.3 Netzwerkknoten

Um eine Nachricht eindeutig einem Rechner zuordnen zu können ist es notwendig, den Punkt im Netzwerk zu kennen, an dem sie generiert wurde. Auch wenn das Monitoring-System zunächst ausschließlich auf Standard PCs mit einer IPv4-basierten Netzwerkanbindung [43] zum Einsatz kommt, soll das Nachrichtenformat flexibel genug gestaltet sein, um auch Knoten anderer Netzwerktypen eindeutig identifizieren zu können. Vier Byte Speicherplatz, wie ihn eine IPv4-Adresse belegen würde, sind somit nicht in jedem Fall ausreichend. Andererseits soll der für den Header erforderliche Speicher so gering wie möglich gehalten werden, da die Gesamtgröße der Nachrichten begrenzt und der Platz für den Nachrichteninhalt maximal zu halten ist.

Eine Größe von 16 Bytes wird für sinnvoll erachtet, da dies auch die Speicherung von Adressen des IPv4-Nachfolgers IPv6 [26] erlaubt und zudem für einen von Menschen einfach lesbaren Rechnernamen ausreicht.

SEV_DEBUG	Debug-Ausgaben
SEV_NOTICE	zur Kenntnisnahme, jedoch keine Aktion erforderlich
SEV_WARNING	Hinweis auf mögliches Fehlverhalten
SEV_ERROR	Hinweis auf einen aufgetretenen Fehler; Eingreifen nicht zwingend erforderlich
SEV_CRITICAL	Hinweis auf einen aufgetretenen Fehler; sofortiges Eingreifen erforderlich um das System in einen sicheren Zustand zu bringen
SEV_EMERGENCY	Hinweis auf einen aufgetretenen Fehler; sofortiges Eingreifen erforderlich um Schaden an Mensch oder Maschine zu verhindern

Tabelle 5.2: Dringlichkeiten (geordnet von unwichtig zu wichtig)

5.1.1.4 Komponente

In der Regel laufen auf einem Rechner mehrere Prozesse, die Nachrichten hervorbringen können. Diese müssen auch in den Nachrichten unterscheidbar sein. Die Identifikation eines Prozesses eines modernen Betriebssystems kann entweder über den Prozessnamen oder über dessen Prozess ID erfolgen.

Für den Prozessnamen spricht, dass der Name einer ausführbaren Datei auch nach deren Beendigung bestehen bleibt. Allerdings ist der Name nicht immer eindeutig, da beispielsweise mehrere Instanzen einer Anwendung gestartet werden können, die dementsprechend denselben Namen tragen. Diese Instanzen bekommen jedoch systemweit eindeutige Identifikationsnummern zugewiesen und können anhand derer unterschieden werden. Da jedoch auch bei einem Absturz einer Anwendung Nachrichten noch eindeutig ihrem Ursprung zugeordnet werden können sollen, die IDs jedoch freigeben werden, wenn der zugehörige Prozess beendet wird, reicht auch diese Information allein nicht aus. Es ist also erforderlich, beide Informationen in den Header einzufügen, da ansonsten eine Identifikation nicht garantiert werden kann.

Die Prozess ID ist bei allen gängigen Betriebssystemen ein vorzeichenloser, ganzzahliger Wert, so dass vier Byte Speicherplatz im Nachrichten-Header ausreichend sind. Der Prozessname hingegen kann, je nach Betriebssystem, eine beliebige Länge annehmen. Dies stellt jedoch kein allzu großes Problem dar, da der Name nicht maschinell, sondern nur von Menschen interpretiert werden muss und im Normalfall bereits der Anfang eines Namens ausreicht um zu erkennen, welche Applikation sich dahinter verbirgt. Daher wurde der Speicherplatz für diese Information auf 16 Bytes festgesetzt.

Weiterhin ist es hilfreich den Entwickler der Software, die eine Nachricht erzeugt hat, zu kennen, da dieser die Nachricht am besten interpretieren und aufgetretene Fehler beheben kann. Es wäre nicht notwendig den Namen des Entwicklers mit in den Nachrichten-Header aufzunehmen, da dieser auch noch im Nachhinein ermittelt werden kann. Allerdings kann der Entwicklername

bei der Filterung nützlich sein, um beispielsweise nur Nachrichten angezeigt zu bekommen, die von eigener Software stammen. Da auch hier die ersten Buchstaben ausreichen um eine eindeutige Information zu erhalten, wurde der Speicher auf ebenfalls 16 Bytes begrenzt. In der vorliegenden Monitoring-Bibliothek wird der Entwickler aus der institutsinternern Bibliothek ausgelesen, in die er bei Initialisierung einer Anwendung eingetragen wird.

Da Prozesse in Threads unterteilt sein können, werden im Header auch Name und ID des Threads vermerkt, der zum Zeitpunkt der Nachrichtengenerierung aktiv war. Thread-Name und ID werden, analog zu Prozessname und Prozess ID, in 16 bzw. vier Byte großem Speicher hinterlegt.

5.1.1.5 Zeitstempel

Bei der Analyse von Systemfehlverhalten oder -abstürzen ist die Reihenfolge der verursachenden Ereignisse oft von entscheidender Bedeutung. Aus diesem Grund wird jede Nachricht bei ihrer Veröffentlichung mit einem Zeitstempel versehen. Um Zeiten verschiedener Systeme miteinander vergleichen zu können wird hierbei auf eine absolute Zeitrepräsentation, konkret die Unixzeit (Teil des POSIX Standards), zurückgegriffen. In acht Bytes ist eine eindeutige, nanosekundengenaue Zeitdarstellung bis über das Jahr 2100 hinaus möglich.

An dieser Stelle soll noch auf einige Probleme hingewiesen werden, die bei der Auswertung der Zeitstempel zu berücksichtigen sind: Zunächst ist zu bemerken, dass absolute Zeit nicht messbar ist und deshalb immer auf ein relatives Zeitsystem zurückgegriffen werden muss. Weiterhin ist eine exakte Synchronisation mehrerer Uhren nicht möglich; es kann jedoch ein maximaler Fehler angegeben werden. (Siehe [48].)

Weiterhin muss berücksichtigt werden, dass die Erstellung einer Nachricht selbst Zeit in Anspruch nimmt und durch Verdrängung des Prozesses durch höherpriore theoretisch beliebig lange verzögert werden kann. Um einen möglichst einheitlichen Zeitpunkt festzulegen, der zudem noch nah am auslösenden Ereignis liegt, wird der Zeitstempel als erster Schritt beim Veröffentlichen einer Nachricht gesetzt. Der Versatz durch Scheduling wird dadurch jedoch nicht vermieden, sondern nur die Wahrscheinlichkeit des Auftretens minimiert. Soll eine Verzögerung durch Scheduling komplett ausgeschlossen werden, liegt ein Locking über das Ereignis und die Veröffentlichung der Nachricht in der Verantwortung des Benutzers.

5.1.1.6 Sequenznummer

Wie in 2.1.10 gefordert, muss ein Nachrichtenverlust erkannt und dem Benutzer mitgeteilt werden. Hierzu werden Nachrichten eines Rechners mit fortlaufenden Sequenznummern versehen, da so Lücken, also fehlende Nummern, erkannt werden können. Durch die unterschiedliche Priorisierung der Nachrichten kann nicht mehr garantiert werden, dass die Generierungsreihenfolge auch der Reihenfolge des Eintreffens beim Empfänger entspricht. Eine Erkennung verlorener Nachrichten wird dadurch verkompliziert. Da das Monitoring-System jedoch FIFO-Verhalten innerhalb einer Prioritätsstufe garantiert, beheben zusätzlich

nach Prioritäten separierte Sequenznummern diese Schwachstelle. Lediglich ein Verlust der jeweils letzten versandten Nachricht, ohne darauf folgende, kann mit dieser Methodik nicht erkannt werden. Dies zu lösen würde eine Empfangsbestätigung der Gegenseite erfordern, was jedoch aufgrund der dadurch auftretenden Verzögerung hier nicht anwendbar ist.

5.1.2 Content

Im Folgenden werden die einzelnen Einträge des Content-Teils aufgelistet und kurz erläutert. Für eine Übersicht siehe Abbildung 5.2.

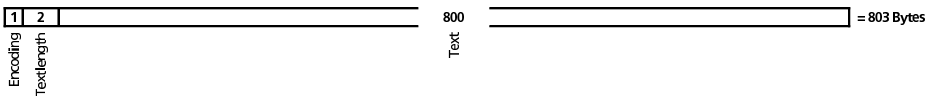


Abbildung 5.2: Speicheraufteilung des Content-Teils (mit beispielhaften 800 Bytes maximaler Textlänge)

5.1.2.1 Text

Der für den Nutzer interessanteste Teil einer Nachricht ist der von ihm frei formulierbare Text. Wie in 2.2.2 ausgeführt, darf innerhalb der Benutzerschnittstelle keine dynamische Speicherallozierung erfolgen, so dass die Länge des Textes durch den in der Initialisierungsphase zu diesem Zweck reservierten Speicher begrenzt ist. Die Größe dieses Speichers kann bei der Konfigurierung der Monitoring-Umgebung für jeden Rechner getrennt eingestellt werden. Es ist jedoch zu beachten, dass die geringste maximale Textlänge auf einem Pfad über mehrere Rechner die Länge des Textes begrenzt, die am Ende ausgegeben werden kann. (Siehe Abbildung 5.3.) Ist die maximale Textlänge auf einem Netzwerkknoten kleiner als die tatsächliche Textlänge einer eingehenden Nachricht, so wird lediglich der vordere Teil weitergeleitet. Dieses Verhalten stellt im vorliegenden Anwendungsfall jedoch keine Einschränkung dar, da Rechner mit geringen Ressourcen an den Enden der Kommunikationsketten zu finden sind (Gelenkebene und darunter) und also keine Nachrichten weiterleiten müssen.

Sind die Ressourcen eines Rechners nicht ausreichend, können statt ausformuliertem Text Symbole verwendet werden, die erst bei der Ausgabe, also auf einem anderen Rechner, in einen vordefinierten Text umgewandelt werden.

5.1.2.2 Textlänge

Als zusätzliche Information wird die tatsächliche Länge des Inhalts in der Nachricht gespeichert. Dadurch kann garantiert werden, dass auch ohne abschließendes Steuerzeichen nicht über den gültigen Speicher hinaus gelesen wird. Weiterhin ermöglicht die Kenntnis der Größe des interessanten Speicherbereichs einen schnelleren Kopiervorgang der Nachrichten, als wenn jedes Byte zunächst auf ein terminierendes Steuerzeichen hin überprüft werden müsste.

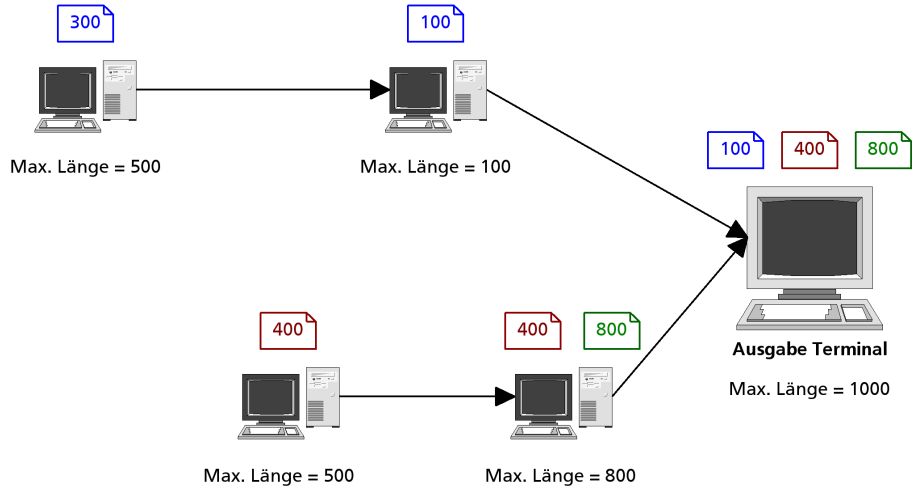


Abbildung 5.3: Verhalten der Textlänge einer Nachricht bei verschiedenen Maximallängen der Netzwerkknoten

5.1.2.3 Zeichenkodierung

In verschiedenen Anwendungen können – je nach Einstellung – auch verschiedene Zeichenkodierungen zum Einsatz kommen (z.B. ASCII [25], UTF-8 [55], etc.). Um eine einheitliche Darstellung von Text innerhalb des gesamten Netzes zu ermöglichen, wird deshalb die beim Erstellen verwendete Kodierung mit übertragen. So kann die Zeichenkodierung vor der Ausgabe gegebenenfalls konvertiert werden. Repräsentiert wird diese Information als einer von mehreren definierten Ganzzahlwerten. Dieses Feld wird ebenfalls verwendet um zu kennzeichnen, ob die Nachricht vollständigen Text oder ein Symbol enthält, das noch in Text umgesetzt werden muss.

5.2 Übertragungsinfrastruktur

5.2.1 Client

Ein Client des Monitoring-Systems ist eine Anwendung, die die Funktionen der Monitoring-Bibliothek nutzt. Dies beinhaltet zum einen die Erzeugung von Nachrichten, also Header-Generierung und befüllen mit Content, und zum anderen deren Veröffentlichung. Veröffentlichen meint in diesem Zusammenhang jedoch nicht die Verbreitung über das Netzwerk, Serialisierung, Filterung oder Ähnliches, sondern lediglich, dass die Nachrichten der Infrastruktur respektive dem Server zur weiteren Verarbeitung zur Verfügung gestellt werden.

Da dieser Ablauf sequentiell in die Ausführung der Anwendung eingebunden wird, ist hier Determiniertheit zu gewährleisten. Aus diesem Grund dürfen an dieser Stelle keine blockierenden Synchronisierungsmechanismen oder dynamisch allozierter Speicher verwendet werden. Da jeder Client eigene Nachrichten erstellt, ist eine Synchronisierung nur an der Schnittstelle zum Server

erforderlich. Dynamische Speicherverwaltung wurde bereits durch die in 5.1 beschriebene starre Struktur der Nachrichten überflüssig gemacht.

5.2.2 Server

Die Nachrichten, die von Clients erzeugt oder über eine Netzwerkschnittstelle empfangen werden, werden im Hintergrund der Echtzeitanwendungen in einem Server-Prozess weiter verarbeitet. Die Form der Weiterverarbeitung muss von einem Systemkonfigurator einfach einstellbar sein und reicht von einer einfachen Ausgabe bis hin zu einer komplexen Filterung mit anschließender Weiterleitung der Nachrichten an andere Netzwerkknoten. Die Aufgaben eines Servers lassen sich damit in drei Bereiche aufteilen:

1. Ausgabe der Nachrichten (lokal oder über das Netzwerk)
2. Empfang von Daten anderer Netzwerkknoten
3. Zusammenführen von lokal erzeugten und empfangenen Nachrichten unter Berücksichtigung der jeweiligen Prioritäten

Aus dieser Aufteilung der Aufgaben ergeben sich die drei Module eines Servers: Das Aufgabengebiet des *Senders* umfasst alle Funktionen, die mit der Ausgabe der Nachrichten zusammenhängen, *Empfänger* stellen Eingangsschnittstellen verschiedener Transportprotokolle dar, und der *Reorganizer* speist die von den Empfängern kommenden Daten in die lokale Kommunikationsinfrastruktur ein, so dass der Sender diese zusammen mit den von lokalen Clients erzeugten Nachrichten auslesen kann. Zusätzlich wurde ein Statistikmodul implementiert, bei dem eingehende oder verlorene Nachrichten registriert werden und bei Bedarf Verlustmeldungen generiert werden können. Ein Überblick über den Nachrichtentransport innerhalb eines Servers, sowie die verschiedenen Ein- und Ausgänge, ist in Abbildung 5.4 exemplarisch dargestellt und wird im Folgenden näher erläutert.

5.2.2.1 Sender

Der Sender ist als eigenständiger Thread innerhalb des Server-Prozesses implementiert. Seine Hauptaufgaben sind das Auslesen der einzelnen Nachrichten aus der lokalen Kommunikationseinheit, das Verpacken der Nachrichten in Netzwerkpakete und die Übergabe dieser Pakete an die Netzwerkkommunikation. Da alle eingehenden Nachrichten den Sender passieren, wurde hier auch die Tracking-Funktion zum Erkennen verlorener Nachrichten verortet. Sollen Nachrichten auch lokal ausgegeben werden, geschieht dies ebenfalls innerhalb des Senders. Eine Übersicht über den Ablaufplan des Senders ist in Abbildung 5.5 zu finden.

Eingang Da alle Aufgaben des Sendemoduls in der Verarbeitung von Nachrichten bestehen, ist eine Ausführung auch nur dann sinnvoll, wenn mindestens eine Nachricht am Eingang vorliegt. Sind alle vorhandenen Nachrichten verarbeitet, soll der Sender-Thread also blockieren, bis eine neue Nachricht in die lokale Kommunikationsstruktur eingetragen wird. Dies stellt ein Erzeuger-Verbraucher-Problem dar, das üblicherweise durch den Einsatz einer zählenden

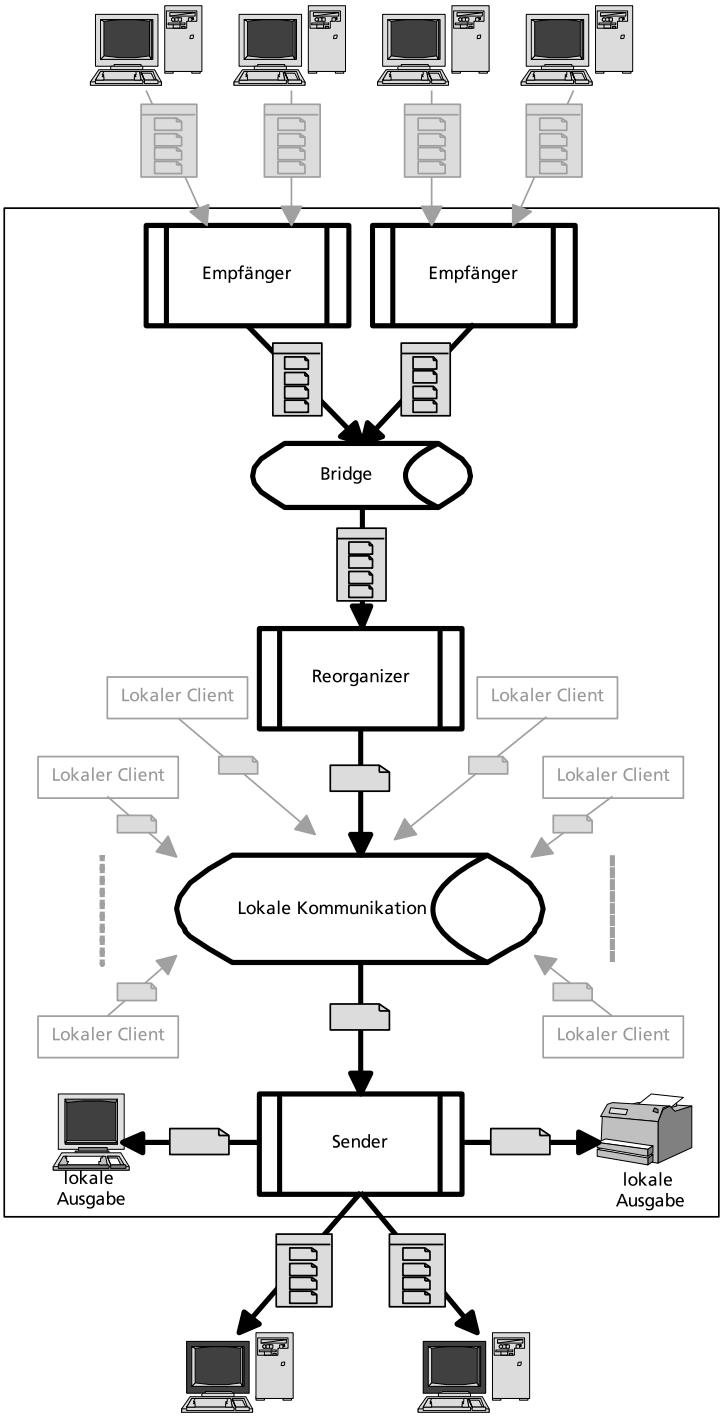


Abbildung 5.4: Server Aufbau

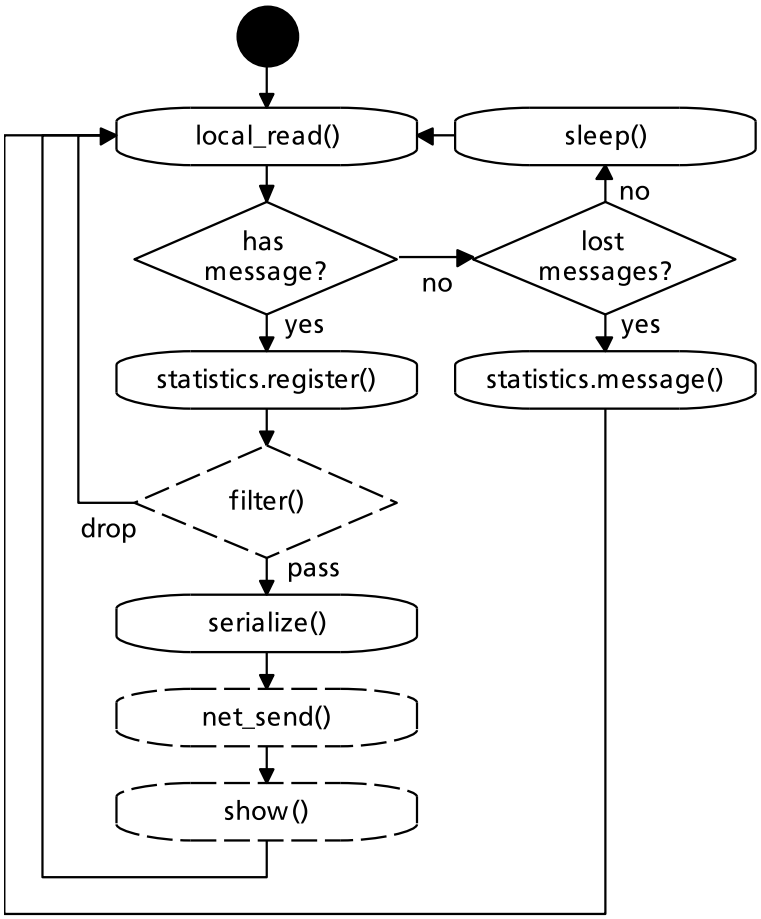


Abbildung 5.5: Sender Ablaufdiagramm: Blöcke mit durchgezogenen Linien sind fest; Blöcke mit gestrichelten Linien führen von Benutzern vorgegebene Funktionalität aus.

Semaphore gelöst werden würde [27]. Da eine Semaphore jedoch als eine gemeinsame Ressource aller Leser und Schreiber implementiert wird, und dementsprechend intern explizite Synchronisierungsmechanismen zum Einsatz kommen, die zur Blockierung eines oder mehrerer Prozesse führen könnten, soll aus den in 2.2.2 genannten Gründen hier darauf verzichtet werden. Eine andere Möglichkeit den Sender-Thread auf eine neu vorliegende Nachricht hinzuweisen, wäre die Verwendung von Signalen (Teil des POSIX-Standards). Das würde jedoch einen zeitlichen Mehraufwand auf Client-Seite bedeuten, so dass auch diese Lösung nicht in Frage kommt. (Vergleiche Randbedingung in 2.2.3.)

Solange in dem Puffer zwischen Client und Server noch Platz für weitere Nachrichten ist, ist eine sofortige Verarbeitung der eingehenden Nachrichten allerdings auch nicht erforderlich. Gelegentliches Testen – *Polling* – einer als leer bekannten lokalen Kommunikationsstruktur reicht somit aus. Das Intervall, in dem die Abfrage stattfinden soll, muss hierbei so an Puffergröße und zu erwartende Nachrichtenfrequenz angepasst werden, dass ein Auslesen vor dem Volllaufen des Puffers gewährleistet ist. Der Sender-Thread wird seine Ausführung nur aussetzen, wenn keine weiteren Aufgaben zu erledigen sind und keine neuen Nachrichten am Eingang vorliegen. Die durch Polling zusätzlich verbrauchte Rechenzeit ist zu vernachlässigen, da der Server-Prozess auf Grund seiner niedrigen Priorität im Verhältnis zu den Benutzerprozessen nur Ressourcen belegen kann, die ansonsten ungenutzt wären.

Nachrichten-Tracking Um erkennen zu können, ob eine Nachricht verloren wurde, werden allen Nachrichten beim ersten passieren eines Senders Sequenznummern zugewiesen. Jede über das Netzwerk eingehende Nachricht wird beim Statistikmodul registriert. Da alle Nachrichten eines Netzwerkknotens auf diese Weise erfasst werden, ist es so möglich, Lücken im Strom der Sequenznummern respektive verlorene Nachrichten zu finden. Liegen für den Sender keine weiteren Nachrichten zur Verarbeitung vor und wurden fehlende Nachrichten entdeckt, so wird eine Nachricht des Statistikmoduls erzeugt und in die lokale Kommunikationsinfrastruktur eingespeist.

Filter Wie in 2.1.6 gefordert, soll die Weiterleitung bzw. Ausgabe einer Nachricht nur erfolgen, wenn sie anhand definierbarer Kriterien dazu qualifiziert ist. Um unnötigen Rechenaufwand zu vermeiden, wird diese Filterung vor der weiteren Verarbeitung der Nachrichten vorgenommen. Filter sind vom Systemkonfigurator zu erzeugen und einzubinden.

Serialisierung und Marshalling Um Datenobjekte über ein Netzwerk versenden zu können, ist eine Repräsentation in Form eines zusammenhängenden Byte-Stroms notwendig. Dieser Vorgang wird als *Serialisierung* bezeichnet. Durch Umkehren der Serialisierung (*Deserialisierung*) muss sich ein semantisch zum Ausgangsobjekt identisches Objekt erzeugen lassen.

Auch wenn die Daten bereits in zusammenhängendem Speicher liegen, ist eine explizite Serialisierung auf Grund der Heterogenität verschiedener Betriebssysteme in puncto Speicher-Layout, Größe von Datentypen, etc. notwendig, da eine

Uminterpretierung (*Casting*) der Objekte als Byte-Strom diese Unterschiede nicht berücksichtigt.

Neben den Unterschieden verschiedener Software-Varianten muss auch die Heterogenität der Hardware bezüglich der Byte-Reihenfolge [22] berücksichtigt werden. Beim *Marshalling* werden die Daten in eine einheitliche Byte-Reihenfolge gebracht; hier *Little Endian*, da momentan nur x86-Rechner verwendet werden. *Demarshalling* bringt die Daten in die Byte-Reihenfolge der jeweiligen Architektur.

Zu Serialisierung und Marshalling einer Nachricht, werden drei verschiedene Verfahren angewandt:

Textfelder des Headers werden in voller Länge Byte-weise in den zu übertragenden Byte-Strom kopiert.

Numerische Daten werden zu Little Endian konvertiert – sofern die Prozessorarchitektur Big Endian ist – und anschließend Byte-weise in den Byte-Strom kopiert.

Das Textfeld des Contents wird auf Grund seiner Länge nicht komplett, sondern nur soweit gefüllt in den Byte-Strom kopiert. Die Anzahl der enthaltenen Zeichen wird zuvor (im Little Endian Format) in den Byte-Strom eingefügt, so dass beim Deserialisieren des Textes dessen Länge bekannt ist. So wird die Länge des Byte-Stroms variabel gehalten. Dadurch muss bei wenig Inhalt auch nur eine geringe Datenmenge übertragen werden.[1]

Die Deserialisierung bzw. das Demarshalling verlaufen analog.

Nachrichtengruppierung Die Nachrichten sollen über ein Netzwerk übertragen werden. Da Netzwerktransportprotokolle Datenpaketen immer auch selbst noch Header hinzufügen, erhöht sich damit die zu übertragende Datenmenge. Gemessen an den Nutzdaten verringert sich dadurch die Durchsatzrate. Ist die Größe der Nutzdatenpakete nun sehr gering, macht der Header des Transportprotokolls einen deutlich größeren Anteil der Netzwerklast aus, als dies bei sehr großen Datenpaketen der Fall wäre. Es ist also sinnvoll, mehrere Nachrichten gebündelt zu verschicken, um so die Belastung für das Netzwerk gering zu halten. (Vergleiche Randbedingung in 2.2.4.)

Andererseits erfolgt der Datentransport bei allen Netzwerktechniken, die auf dem OSI-Schichtenmodell (siehe [19]) aufbauen, ab der Vermittlungsschicht auf der Basis von Paketen als Transporteinheiten. Zwischen Transport- und Vermittlungsschicht besteht jedoch kein zwingender Zusammenhang in der maximalen Größe der Pakete, so dass prinzipiell ein Segment der Transportschicht auch größer als ein Vermittlungsschichtpaket sein kann. In diesem Fall wird die Übertragung entweder abgebrochen oder das Segment der Transportschicht wird

[1] Bei der (De-)Serialisierung des Contents müsste auch die Zeichenkodierung [20] angeglichen werden. Da dies im aktuellen System noch nicht notwendig ist, wurde eine Umwandlung noch nicht implementiert. Mit der Einführung eines entsprechenden Feldes im Content Teil der Nachrichten wurde eine Implementierung jedoch bereits vorbereitet.

auf mehrere Vermittlungsschichtpakete aufgeteilt. Da diese Teilpakete wieder jeweils eigene Header beinhalten, geht der Vorteil großer Pakete hierbei verloren (*Fragmentierung*).

Den beiden vorangegangenen Absätzen ist zu entnehmen, dass die optimale Größe eines Nachrichtenpakets in der maximalen Größe eines Vermittlungsschichtpakets (MTU = Maximum Transfer Unit [49]), abzüglich der Header-Größen darüber liegender Schichten besteht. Die MTU wird durch die verwendete Netzwerk-Hardware festgelegt und kann daher zwischen verschiedenen Netzwerkabschnitten variieren. Eine Fragmentierung der Pakete kann bei unzureichender Kenntnis der Infrastruktur somit nicht ausgeschlossen werden. Im vorliegenden Fall wurde die MTU mit dem für Ethernet üblichen Wert von 1492 Bytes angenommen. Der Overhead, der sich aus dem Protokoll der Transportschicht ergibt, muss bei der Wahl eines Protokolls gesetzt werden, um eine optimale Paketgröße zu gewährleisten.

Ein Nachrichtenpaket setzt sich aus mehreren, serialisierten und gemarshallten Nachrichten zusammen. Um die Nachrichten beim Empfänger wieder korrekt aus dem Paket extrahieren zu können, wird vor jeden Nachrichtenblock dessen Länge geschrieben. Die Länge wird in Bytes angegeben und durch einen 2 Byte großen Wert repräsentiert. Auch hier muss die Byte-Reihenfolge verschiedener Architekturen berücksichtigt werden. Daher werden diese Werte ebenfalls gemarshallt.

Steht nach einer serialisierten Nachricht der Wert '0' an Stelle der Länge der folgenden Nachricht, so weist dies auf das Ende des Paketes hin. Mit diesem Aufbau ist es nicht erforderlich, die Pakete vor dem Versenden komplett zu füllen. Dies ist von Vorteil wenn über einen längeren Zeitraum keine neuen Nachrichten eintreffen, sich jedoch bereits mindestens eine Nachricht im Paket befindet.

Ausgang Am Ausgang des Senders werden Datenpakete, die wie zuvor beschrieben erzeugt wurden, an die Transportschicht des OSI-Modells übergeben. Der tatsächliche Versand kann hierbei durch zwei Ereignisse angestoßen werden: Wenn das maximale Fassungsvermögen eines Nachrichtenpakets erreicht ist, wird dieses noch im selben Durchlauf versandt. Dies wird dadurch erkannt, dass eine eingehende Nachricht nicht mehr vollständig darin gespeichert werden kann. Diese abgelehnte Nachricht wird in das folgende Paket geschrieben. Stehen am Eingang keine neuen Nachrichten mehr bereit, ist auch dies Auslöser für einen sofortigen Versand. Da unbekannt ist, wann die nächste Nachricht eintreffen wird, ist es nicht sinnvoll ein bereits begonnenes Paket weiter aufzuhalten, und es wird mit den bis zu diesem Zeitpunkt enthaltenen Nachrichten verschickt.

Neben der Übergabe der Nachrichten an das Netzwerk werden auch lokale Ausgaben an dieser Stelle durchgeführt. Hierzu wird jede – unserialisierte – Nachricht an die Ausgabeschnittstelle übergeben und an alle benutzerdefinierten Ausgabeobjekte weitergeleitet.

5.2.2.2 Empfänger

Gegenstellen des Senderausgangs sind Empfängermodule anderer Server im Netzwerk. Da verschiedene Transportprotokolle unterstützt werden sollen, können auch mehrere Empfängermodule gleichzeitig in einem Server existieren. Um die Pakete möglichst schnell aus dem Puffer der Transportschicht zu lesen und so Platz für die nächsten eingehenden Daten zu schaffen, werden die Empfänger ebenfalls als Threads implementiert, die unabhängig von den übrigen Server-Teilen arbeiten. Für einen Überblick über die Funktionsweise eines Empfängers siehe Abbildung 5.6.

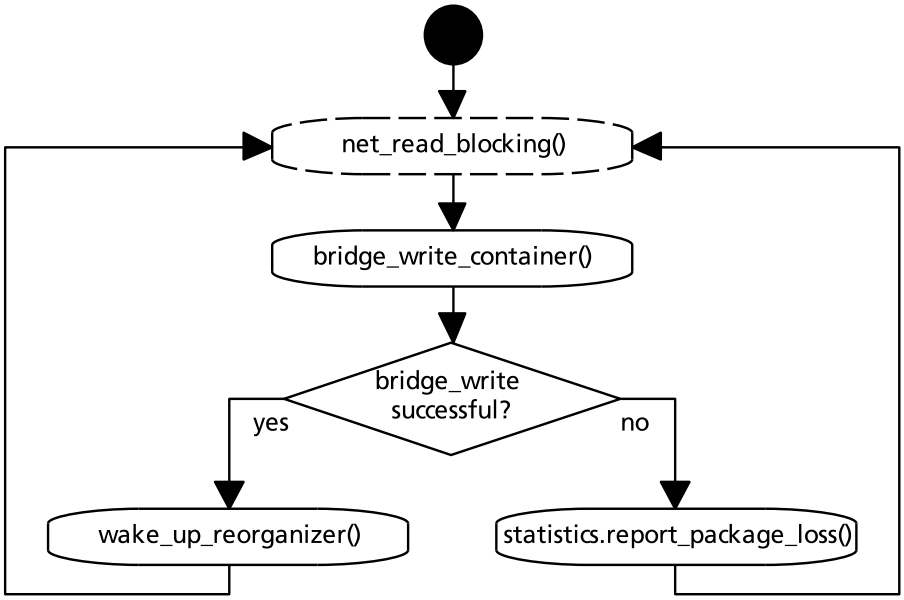


Abbildung 5.6: Empfänger Ablaufdiagramm

Die ankommenden Datenpakete werden von Empfängern nicht interpretiert, sondern lediglich in einen Zwischenspeicher (*Bridge*) kopiert, der dann vom Reorganizer ausgelesen wird. Einzige Anforderung an die Bridge ist FIFO-Funktionalität. Eigenschaften wie Geschwindigkeit, Speichereffizienz und Blockadefreiheit sind ebenfalls wünschenswert, unterliegen jedoch nur der Einschränkung, dass der Server als Ganzes alle eingehenden Nachrichten in der ihm zur Verfügung stehenden Zeit verarbeiten können muss. Ringbuffer erfüllen diese Bedingungen, so dass eine Instanz hier als Bridge zwischen Empfänger und Reorganizer zum Einsatz kommt.

Soll ein Paket in die Bridge übertragen werden, diese hat jedoch keinen freien Platz mehr zur Verfügung, so gibt der Empfänger-Thread die Kontrolle über die CPU ab, um es dem Reorganizer-Modul zu ermöglichen Platz freizugeben. Erhält der Empfänger die Kontrolle zurück, wird erneut versucht das Paket einzutragen. Dieser Zyklus wird bis zu 50 mal wiederholt. Ist das Eintragen danach immer noch nicht erfolgreich, wird das Paket verworfen und dem

Statistikmodul der Verlust der enthaltenen Nachrichten mitgeteilt. Die Pakete werden nicht sofort verworfen, da dies zur Folge hätte, dass nachfolgende Pakete mit hoher Wahrscheinlichkeit ebenfalls verloren gehen würden. Blockiert der Empfänger jedoch kurzzeitig, kann der Puffer der Netzwerkschnittstelle noch ausgenutzt werden. Die Anzahl der Wiederholungen wurde begrenzt um die Netzwerkschnittstelle zu entlasten, da diese unter Umständen mit anderen Anwendungen geteilt werden muss.

Um Polling zu vermeiden, und da die Funktion des Servers nicht echtzeitkritisch ist, kann hier eine Semaphore [27] verwendet werden, die dem Leser der Bridge neu eingetragene Nachrichtenpakete signalisiert und den gleichzeitigen Zugriff aus verschiedenen Threads synchronisiert.

5.2.2.3 Reorganizer

Aufgabe des Reorganizers ist das Entpacken der vom Empfänger kommenden Nachrichtenpakete und die Einspeisung der so erhaltenen Nachrichten in die lokale Kommunikationsstruktur. (Siehe Abbildung 5.7.) Auch der Reorganizer ist als zyklisch ablaufender Thread implementiert. Dieser synchronisiert sich auf die Semaphore, die vom Empfänger zur Signalisierung neuer Nachrichtenpakete verwendet wird. Befindet sich mindestens ein Paket in der Bridge, so liest der Reorganizer das älteste aus.

Aus diesem Paket werden nacheinander die einzelnen Nachrichtenblöcke ausgelesen, deserialisiert und demarshallt und in die lokale Kommunikationsstruktur eingefügt. Da die Nachrichten im Sender in der Reihenfolge ihrer Prioritäten ausgelesen und in ein Paket geschrieben werden, sind die Nachrichten auch innerhalb eines Pakets nach Priorität geordnet. So werden beim Auslesen im Reorganizer hochpriore Nachrichten auch zuerst an die lokale Kommunikation weitergereicht.

Schlägt die Weitergabe einer Nachricht an die lokale Kommunikationsstruktur fehl, so wird die Nachricht verworfen und dies dem Statistikmodul mitgeteilt. Ein Blockieren bis wieder Platz zur Verfügung steht ist nicht sinnvoll, da dies eventuell in nachfolgenden Paketen enthaltene höherpriore Nachrichten blockieren würde, obwohl für diese auf Grund ihrer Priorität noch Platz zur Verfügung stehen würde.

Sind keine Pakete mehr zu verarbeiten, blockiert der Thread bis er von einem Empfänger wieder aufgeweckt wird.

5.2.2.4 Statistik

In Abschnitt 2.1.10 wurde gefordert, dass der Verlust von Nachrichten erkannt werden muss. Zu diesem Zweck wurden Sequenznummern eingeführt. Das Statistikmodul dient nun zur Auswertung der Sequenznummern aller Nachrichten, die beim Sender eingehen.

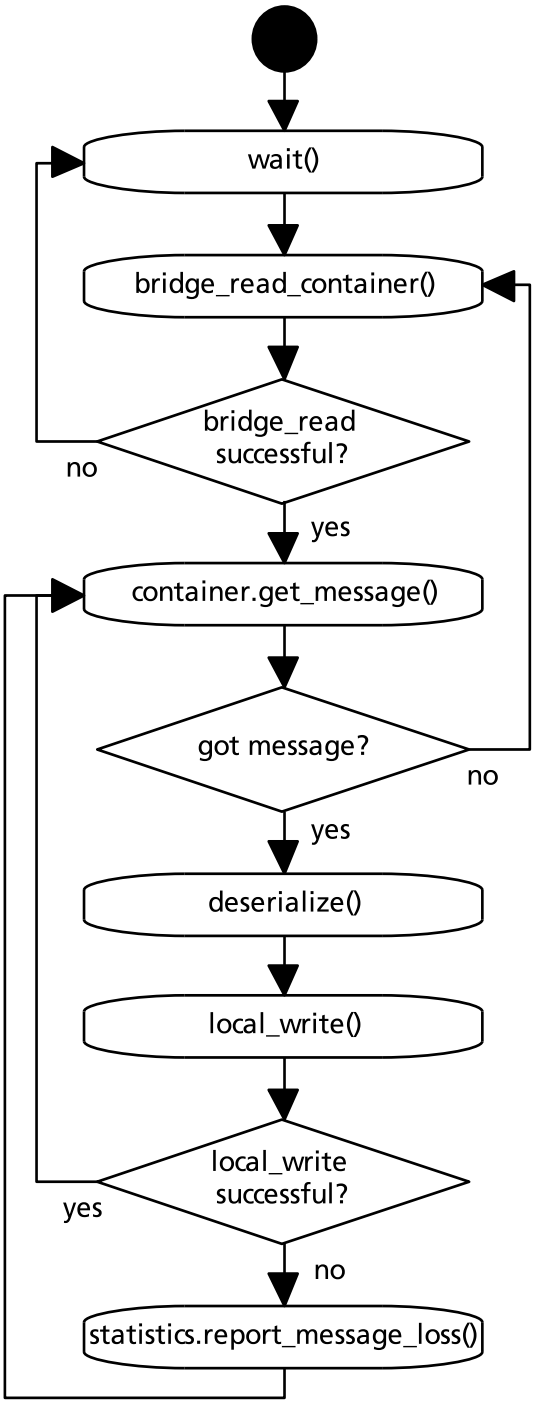


Abbildung 5.7: Reorganizer Ablaufdiagramm

Nachrichten, deren Header identische Werte für Netzwerkknotenname und Dringlichkeit (zusammen *Signatur*) enthalten, bekommen beim ersten Durchlaufen eines Senders aufeinander folgende Sequenznummern zugewiesen. Jede Nachricht müsste demnach eine Sequenznummer besitzen, die um eins höher ist als die der vorangegangenen Nachricht selber Signatur. Ist die Sequenznummer einer empfangenen Nachricht größer als die vorhergesagte Sequenznummer, wurden Nachrichten mit der Signatur dieser Nachricht verloren. Die Anzahl der verlorenen Nachrichten entspricht der Differenz aus tatsächlicher und vorhergesagter Sequenznummer.

Wurden verlorene Nachrichten erkannt, wird dies in eine Liste eingetragen. Diese wird auf Anforderung des Sendemoduls in eine Nachricht umgewandelt. Die Signaturen der verlorenen Nachrichten und die jeweilige Anzahl werden dazu in den Content-Bereich der Verlustnachricht geschrieben und aus der Liste des Statistikmoduls gelöscht. Die Dringlichkeit der Verlustnachricht wird auf die höchste Dringlichkeit der verlorenen Nachrichten gesetzt.

Um eine Verfolgung der Sequenznummern zu ermöglichen, werden für jede einmal aufgetretene Signatur die jeweils letzte Sequenznummer und der dazugehörige Zeitstempel gespeichert. Läuft ein Server lange Zeit und erhält er häufig Nachrichten von wechselnden Netzwerkknoten, wächst damit auch der Speicherverbrauch. Mit Hilfe des Zeitstempels ist es aber möglich, Einträge zu löschen, wenn lange Zeit keine entsprechenden Nachrichten empfangen wurden, da das in der Regel auf nicht mehr existierende Netzwerkknoten hinweist.

5.2.3 Datentransfer Client→Server

Nach der Erzeugung einer Nachricht in einem Client muss diese an den Server-Prozess des Netzwerkknotens übertragen werden. Dies muss von mehreren konkurrierenden Client-Prozessen gleichzeitig möglich sein, wobei die Konsistenz der Daten auch bei gegenseitigen Unterbrechungen der Prozesse garantiert werden muss. Zum anderen dürfen sich konkurrierende Prozesse nicht gegenseitig in ihrer Ausführung beeinträchtigen, so dass eine explizite Synchronisierung, beispielsweise mittels Semaphoren oder Mutexes, nicht in Frage kommt. Weiterhin gelten die in 2.2.2 und 2.2.3 erläuterten Randbedingungen.

In 3.2 wurde bereits festgestellt, dass eine Implementierung auf Shared Memory Basis gute Ergebnisse erwarten lässt. Im Folgenden wird nun eine Struktur zur Verwaltung des Shared Memory entwickelt, die die geforderten Eigenschaften besitzt.

Als Grundlage einer solchen Datenstruktur werden üblicherweise ein- bzw. zweifach verkette Listen, Stacks oder Ringbuffer verwendet. (Siehe [21] Kapitel *4 Data structures.*) Stacks sind nicht geeignet um FIFO-Verhalten zu gewährleisten, so dass diese nicht weiter betrachtet werden sollen. Lese- bzw. Schreiboperationen mit FIFO-Verhalten können sowohl mit Listen als auch mit Ringbuffern mit einer Komplexität von $O(1)$ implementiert werden. Bei der Verwendung von Ringbuffern wird jedoch die Anzahl der Elemente, die verwaltet werden können, implizit durch die Größe des Ringbuffers begrenzt; Listen

sind prinzipiell unbegrenzt. Da das Shared Memory nicht dynamisch in seiner Größe angepasst werden soll, ist eine Beschränkung in der Verwaltungsstruktur sinnvoll. Soll dieses Verhalten mit Listen implementiert werden, muss zusätzlich über die aktuelle Anzahl der Elemente in der Liste buchgeführt werden. Bei jedem Zugriff auf die Liste wären damit zwei Operationen notwendig: Manipulation der Liste und Manipulation der Elementenanzahl. Dies erfordert explizite Synchronisierung und ist somit nicht einsetzbar. Damit bleiben nur Ringbuffer als Grundlage einer Implementierung.

Ein Ringbuffer bietet zwar FIFO-Funktionalität, jedoch ohne Berücksichtigung der Prioritäten der Einträge. Durch den Einsatz von je einem Ringbuffer pro Prioritätsstufe kann dies kompensiert werden. Eine strikte Trennung der Prioritätsstufen könnte allerdings dazu führen, dass ein Ringbuffer voll läuft und deshalb Nachrichten verworfen werden müssen, obwohl auf einer niedrigeren Stufe noch ausreichend Speicher verfügbar wäre. Alternativ könnte der Schreibvorgang so lange blockiert werden, bis die Nachricht erfolgreich eingetragen wurde. Allerdings würde dieses Vorgehen das Kriterium der Determiniertheit verletzen. Der intuitive Ansatz zur Lösung dieses Problems – hochpriore Nachrichten notfalls auch auf niedrigeren Prioritätsstufen einzuhängen – kann zu Prioritätenumkehr führen, da eine hochpriore Nachricht hinter bereits im Buffer befindlichen niederprioren eingereiht würde und dementsprechend auch erst nach diesen wieder ausgelesen werden könnte.

Bei der Vergabe des verfügbaren Speichers müssen also zwei Kriterien erfüllt werden:

- Es müssen auch bei voller Auslastung der niederprioren Ringbuffer Reserven für auftretende höherpriore Nachrichten gehalten werden.
- Bei Erschöpfung des Speicherkontingents einer Prioritätsstufe muss auf verfügbaren niederprioren Speicher zurückgegriffen werden können, ohne dass dies zu einer Prioritätsumkehr beim Nachrichtentransport führt.

Eine Struktur, die diese Vorgaben erfüllt, kann durch die Trennung von Speicher und Verwaltung erreicht werden. Dazu wird der Speicher in Segmente aufgeteilt, die jeweils eine Nachricht fassen können und anhand ihrer Position, respektive über Indices, angesprochen werden. Zur Verwaltung werden je zwei Ringbuffer pro Priorität verwendet, von denen einer die Indices der von dieser Prioritätsstufe belegten Speichersegmente enthält, der andere die noch verfügbaren.

Eine Schreiboperation unterteilt sich demnach in drei Schritte (vergleiche Pseudo-Code in 5.1):

1. Index eines Speichersegments aktueller oder niedrigerer Priorität reservieren.
2. Nachricht an die reservierte Stelle kopieren.
3. Index in den Ringbuffer belegter Segmente der aktuellen Priorität eintragen.

```
1  bool write(message)
2  {
3   // reserve slot (1)
4   for (prio = message.priority(); prio >= 0; prio--)
5   {
6       index = available_slots[prio].read();
7       if (index != -1) break;
8   }
9
10  // if no slot was available, signal a failure
11  if (prio < 0) return false;
12
13  // copy message to reserved slot (2)
14  shared_memory[index] = message;
15
16  // propagate new entry (3)
17  occupied_slots[message.priority()].write(index);
18
19  // signal success
20  return true;
21 }
```

Quelltext 5.1: Shared Memory Schreiboperation (Pseudo-Code)

Die korrespondierende Leseoperation gliedert sich wie folgt (vergleiche Pseudo-Code in 5.2):

1. Belegtes Speichersegment höchster Priorität finden.
2. Nachricht in lokalen Speicher kopieren.
3. Index des gelesenen Segments auf niedrigst möglicher Prioritätsstufe verfügbar machen.

Die richtige Dimensionierung der verwendeten Ringbuffer relativ zueinander ist entscheidend, um die Kapazitäten der unteren Prioritätsstufen zu begrenzen und damit Reserven für darüber liegende Prioritäten zu halten. Der zur Verfügung gestellte Speicher wird zunächst gleichmäßig auf die Prioritätsebenen verteilt. Die Anzahl der freien Speichersegmente ($size$) pro Priorität, die maximal gehalten werden kann, berechnet sich dem zufolge nach Formel 5.1.

$$size = \frac{\text{Gesamtgröße}}{\text{Anzahl der Prioritäten}} \tag{5.1}$$

Ist die Gesamtgröße kein ganzzahliges Vielfaches der Prioritäten, so wird der verbleibende Rest der niedrigsten Priorität zugeordnet, da von dieser auch alle anderen Prioritäten Speicher beziehen können. Für eine exemplarische Dimensionierung der Ringbuffer zur Verwaltung der freien Speichersegmente siehe Abbildung 5.8.

```
1  bool read(message&)
2  {
3   // get occupied slot (1)
4   for (prio = max_prio; prio >= 0; prio--)
5   {
6       index = occupied_slots[prio].read();
7       if (index != -1) break;
8   }
9
10  // if no slot was occupied, signal a failure
11  if (prio < 0) return false;
12
13  // copy message to local memory (2)
14  message = shared_memory[index];
15
16  // make slot available for further write operations (3)
17  for (prio = 0; prio <= max_prio; prio++)
18  {
19      if (available_slots[prio].write(index)) return true;
20  }
21 }
```

Quelltext 5.2: Shared Memory Leseoperation (Pseudo-Code)

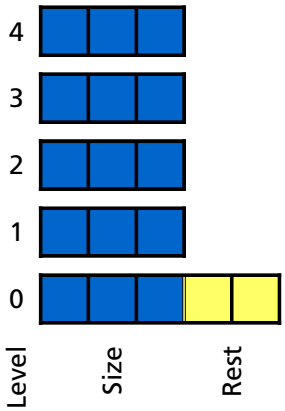


Abbildung 5.8: Ringbuffer zur Verwaltung der freien Speichersegmente für 17 Nachrichten bei 5 Prioritätsstufen

Da mit Steigerung der Priorität auch immer mehr Speicher zur Verfügung steht, müssen die Ringbuffer zur Verwaltung belegter Segmente so ausgelegt werden, dass der komplette auf ihrer und den darunter liegenden Stufen verfügbare Speicher von ihnen gehalten werden kann. Andererseits kann die Begrenzung des Speichers jeder Stufe implizit durch die Größe der Ringbuffer erreicht werden. Ein beispielhafter Aufbau ist in Abbildung 5.9 zu finden. Die tatsächliche Größe des gesamten Speichers muss je nach Systemleistung und zu erwartender Nachrichtenfrequenz individuell eingestellt werden.

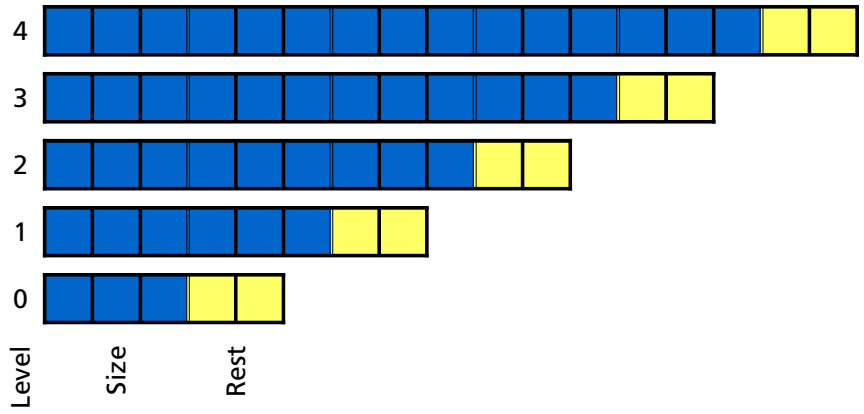


Abbildung 5.9: Ringbuffer zur Verwaltung der belegten Speichersegmente für 17 Nachrichten bei 5 Prioritätsstufen

Bei je einem Schreibe- und einem Leseprozess sind übliche Ringbuffer-Implementierungen prinzipiell nicht anfällig für *Race-Conditions*, da Schreibe- und Leseadresse nur von dem entsprechenden Prozess verändert werden. Im vorliegenden Fall kann die Anzahl der Prozesse jedoch nicht auf je einen begrenzt werden, da dies eine Begrenzung der Client-Anzahl nach sich ziehen würde. Sollen Ringbuffer von mehreren Clients gleichzeitig genutzt werden, muss dennoch die Konsistenz der Datenstruktur sichergestellt werden. Explizite Synchronisierungsmechanismen (siehe [52] Kapitel *2.3.2. Critical Regions*) wurden in 2.2.2 bereits ausgeschlossen. Eine Erweiterung üblicher Ringbuffer-Implementierungen gemäß [47] erlaubt jedoch den Zugriff durch beliebig viele konkurrierende Schreiber und Leser. Die Implementierung basiert auf der atomaren *Compare-And-Swap* Operation der CPU und der Reservierung von Schreib- bzw. Lesepositionen vor dem eigentlichen Zugriff und kann so auch ohne explizite Synchronisierung konsistente Daten garantieren.

Erste Messungen einer solchen Speicherverwaltung ergaben neben einer hohen Wertedichte im Bereich um 0,8 µs auch noch eine zweite Häufung mit deutlich höheren Werten (siehe Abbildung 5.10). Untersuchungen mit der Momentics Tool Suite (QNX Software Systems) ergaben, dass diese hohen Werte auf Seitenfehler (siehe [52] Kapitel *4.3 Virtual Memory*) zurückzuführen sind. Shared Memory wird unter QNX mit der Funktion `mmap()` in die jeweiligen Prozessadressräume eingeblendet. Nach [24] erfolgt die tatsächliche Verknüpfung des

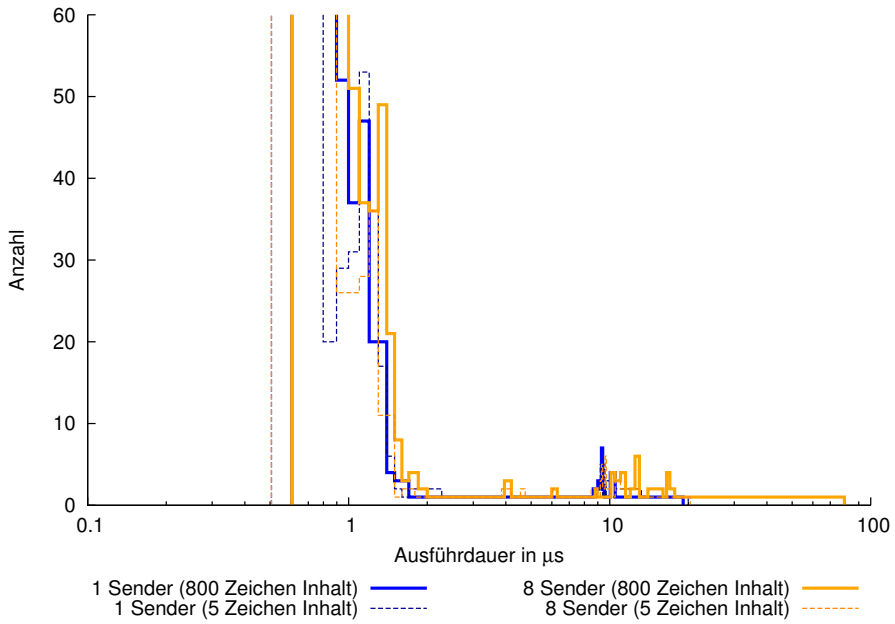


Abbildung 5.10: Nachrichtenversand über Shared Memory mit Ringbuffer-basierter Verwaltung

physikalischen Speichers mit der virtuellen Adresse (*Mapping*) jedoch erst beim ersten Schreibvorgang auf eine noch nicht verknüpfte Speicherseite. Die Dauer dieser Verknüpfung und die dazu notwendigen Kontextwechsel führen zu den beobachteten Verzögerungen:

> When you allocate memory with mmap(), it first allocates a virtual address range, and then [...] it will allocate the physical memory needed to back that object. What it doesnt do, though is setup all the page table mappings for the mapping [...]. In actual fact it will wait until the program first accesses a page before setting up a page table entry for it.
>
> aus *No fault of your own...* von QNX Entwickler Colin Burgess [24]

Werden dem Prozess, der `mmap()` aufruft, Ein-/Ausgabeprivilegien gegeben, erfolgt eine Verknüpfung sofort. Dies würde jedoch ebenfalls bedeuten, dass Speicher dieses Prozesses – und damit jedem, der die Monitoring-Bilbiothek nutzt – nicht ausgelagert werden kann. Das könnte bei hoher Speichernutzung dazu führen, dass Speicher von wichtigen Anwendungen vor unwichtigeren Anwendungen, die aber die Monitoring-Bibliothek nutzen, ausgelagert wird. Damit verletzt dieses Vorgehen die Randbedingung aus 2.2.2.

Stattdessen wurde dieses Problem in vorliegender Implementierung umgangen, indem direkt nach dem Aufruf von `mmap()` auf jeder Speicherseite mindestens ein Byte geschrieben wird. Damit erfolgt das Mapping der Speicherseiten bereits

in der Initialisierungsphase. Durchgeführte Messungen nach dem in 3.2 erläuterten Vorgehen ergaben die in den Graphen 5.11 und 5.12 abzulesenden Ausführdauerverteilungen. Sowohl Mittelwerte, als auch WCETs der Ausführdauern bleiben mit diesem Verfahren deutlich unter denen aller anderen in 3.2 untersuchten IPC-Mechanismen. Daher wird beschriebene IPC-Implementierung in der Monitoring-Infrastruktur zur Kommunikation zwischen Clients und Servern eingesetzt.

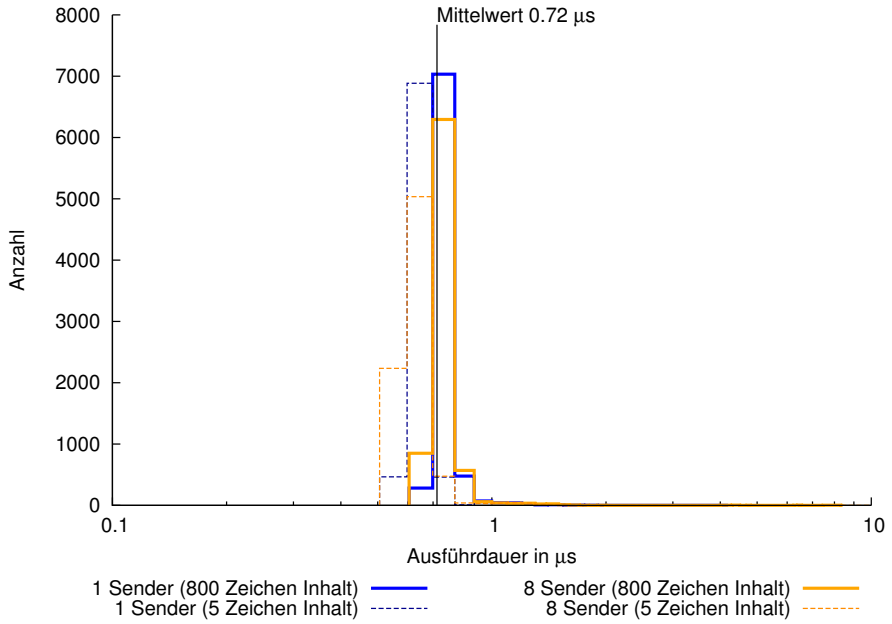


Abbildung 5.11: Nachrichtenversand über Shared Memory mit Ringbuffer-basierter Verwaltung

5.2.4 Datentransfer Server→Server

Wie in Abschnitt 5.2.2.1 erwähnt, werden Nachrichtenpakete über ein Netzwerk von Server zu Server weitergeleitet. Auf der Transportschicht sollen dabei Standardprotokolle verwendet werden. (Siehe Kapitel 4.) In 3.2 wurden einige in Frage kommende Protokolle analysiert. Da die Monitoring-Infrastruktur jedoch mit verschiedenen Protokollen arbeiten können soll, wird hier lediglich eines gewählt, das für vorliegendes Szenario geeignet ist. Die Schnittstelle der Server wird allerdings nicht darauf beschränkt, sondern offen gehalten.

Kriterien, die bei der Auswahl eines Transportprotokolls berücksichtigt werden müssen, sind Zuverlässigkeit, Overhead, Stauvermeidung und Verfügbarkeit auf verschiedenen Systemen. Eine Zusammenfassung der Auswertungen aus 3.2 hinsichtlich dieser Kriterien ist in Tabelle 5.3 zu sehen. Da Zuverlässigkeit immer auch erhöhten Overhead in Form von Header-Einträgen und Steuerungsnachrichten nach sich zieht, und Zuverlässigkeit für aktuellen Aufbau nicht gefordert ist, werden TCP, SCTP und Qnet ausgeschlossen. Stauvermeidung

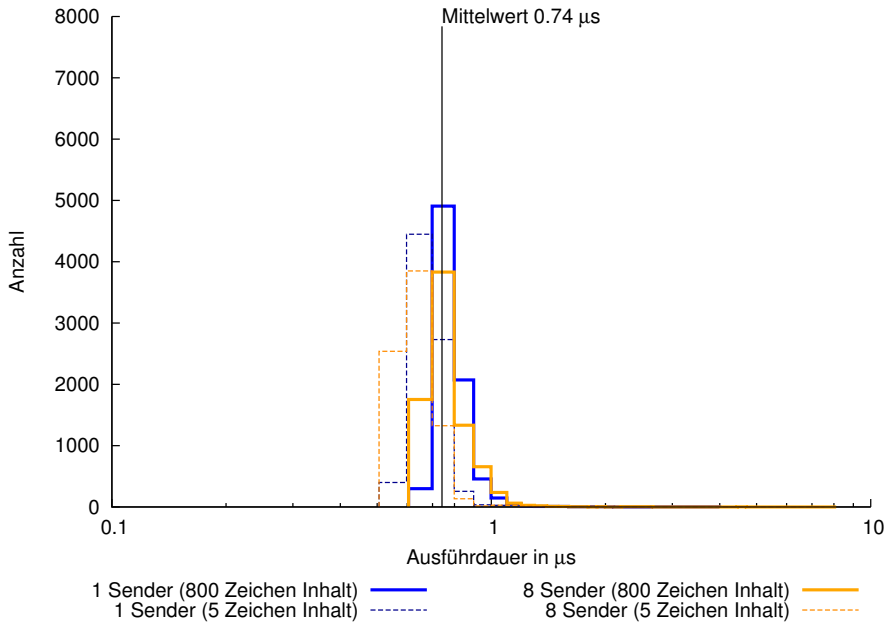


Abbildung 5.12: Nachrichtenempfang über Shared Memory mit Ringbuffer-basierter Verwaltung

	Zuverlässigkeit	Header	Steuerungs-nachrichten	Stau-vermeidung	QNX	Linux
TCP	ja	20–60 Byte	ja	ja	ja	ja
UDP	nein	8 Byte	nein	nein	ja	ja
SCTP	ja	16 Byte	ja	nein	ab v6.3.0	ab Kernel 2.6
DCCP	nein	12/16 Byte	ja	ja	nein	ab Kernel 2.6.14
Qnet	ja	min. 56 Byte (– IP-Header)	ja	nein	ja	nein

Tabelle 5.3: Transportprotokollvergleich

ist in Netzwerken mit hoher Last sinnvoll und wird implizit in 2.2.4 gefordert. DCCP ist unter QNX noch nicht verfügbar, so dass ein Einsatz hier nicht ohne Weiteres möglich ist. Bei der aktuellen Netzwerkauslastung ist Stauvermeidung nicht notwendig, so dass UDP in der aktuellen Konfiguration eingesetzt werden kann. Stehen ausreichend Ressourcen zur Verfügung, können die Vorteile eines aufwendigeren Protokolls genutzt werden.

5.3 Benutzerschnittstelle

5.3.1 Client

Die Monitoring-Bibliothek stellt ein Werkzeug dar, das genutzt werden soll, um den Zustand des Systems zu überwachen. Da die Bibliothek ihren Benutzern die Arbeit erleichtern soll, muss auch die Schnittstelle so einfach gehalten sein, dass sie ohne aufwändige Einarbeitung nutzbar ist. Die Schnittstelle muss eine Möglichkeit bereitstellen, Nachrichten zu erzeugen und diese an die Monitoring-Infrastruktur zu übergeben.

Eine Fehlerbehandlung ist für einen Benutzer der Monitoring-Bibliothek sekundär, da Fehler im Monitoring-System nicht innerhalb der Client-Anwendung behoben werden können. Weil aufgetretene Fehler lediglich das Monitoring-System beeinträchtigen können, ist es für die Funktion des Gesamtsystems auch tolerierbar, wenn diese unbehandelt bleiben. Dementsprechend werden keine *Exceptions* zur Fehlerbenachrichtigung verwendet, da diese behandelt werden müssten. Stattdessen wird auf Rückgabewerte und zusätzliche Funktionen zur Statusabfrage zurückgegriffen. So bleibt es dem Benutzer überlassen, ob er diese Informationen auswerten möchte.

5.3.1.1 Nachrichtenerzeugung

Eine Nachricht entspricht einer Instanz der Nachrichtenklasse `Message`. Vom Benutzer können jeder Nachricht Typ (siehe 5.1.1.1), Dringlichkeit (siehe 5.1.1.2) und Inhalt (siehe 5.1.2) zugewiesen werden. Dies kann direkt bei der Instantiierung durch entsprechende Konstruktorparameter geschehen. Ohne Angabe von Parametern werden die Standardwerte `TYPE_APP` und `SEV_ERROR` verwendet; der Inhalt bleibt leer.

Durch Aufruf von `set_type()` bzw. `set_severity()` an einem Nachrichtenobjekt können Typ und Dringlichkeit neu gesetzt werden. Der Inhalt kann durch `set_content()` überschrieben oder durch `reset_content()` geleert werden. Weiterhin ist es möglich den Inhalt sukzessive aufzubauen. Hierzu kann eine Referenz auf einen `ContentStream` durch `content_stream()` abgerufen werden. `ContentStream` wurde als Unterklasse von `std::ostream` implementiert; damit können alle Objekte an eine Nachricht übergeben werden für die ein entsprechender Operator existiert.

Die Header-Felder `Host Name`, `Author`, `Process Name` und `Process ID` werden bei der Erzeugung einer Nachricht bzw. beim Aufruf von `reset()` automatisch gesetzt. Die Felder `Thread Name`, `Thread ID`, `Timestamp` enthalten Informationen, die erst bei der Übergabe der Nachrichten an die Monitoring-Infrastruktur feststehen

und werden demnach auch erst zu diesem Zeitpunkt – ebenfalls automatisch – gesetzt.

5.3.1.2 Nachrichtenveröffentlichung

Die Monitoring-Umgebung wird in Form einer Singleton-Klasse (siehe [31] Kapitel *3 Creational Patterns*) `Monitor` in die Client-Software eingeblendet. Die Übergabe einer Nachricht an die Monitoring-Infrastruktur erfolgt durch den Aufruf von `Monitor::instance().publish()` mit dem Nachrichtenobjekt als Parameter. Fehler bei der Veröffentlichung werden im Rückgabewerte des Methodenaufrufs kommuniziert. Die Bedeutungen der Rückgabewerte sind in Tabelle 5.4 aufgeführt.

In Anlehnung an die C++-Ausgabeschnittstelle `std::cout` können Nachrichten auch über den Stream `mout` an die Monitoring-Infrastruktur übergeben werden. Das Ergebnis der jeweils letzten Veröffentlichung kann durch Aufruf der Methode `state()` am Stream-Objekt ausgelesen werden. Die möglichen Werte entsprechen ebenfalls denen in Tabelle 5.4.

OK	Veröffentlichung war erfolgreich
PUBL_FULL	Puffer zwischen Client und Server voll
PUBL_THREAD	Fehler beim Setzen der Thread Information
PUBL_TIME	Fehler beim Setzen des Zeitstempels

Tabelle 5.4: Mögliche Ergebnisse einer Nachrichtenveröffentlichung

Beispiel-Code zur Verwendung der Monitoring-Bibliothek ist in Quelltext 5.3 zu finden.

5.3.2 Server

Ein Server-Objekt wird durch Instantiierung der Klasse `Server` erzeugt. Diese Klasse sowie alle weiteren für den Betrieb eines Servers notwendigen Komponenten wurden in einer Bibliothek zusammengefasst, so dass einfach verschiedene Server-Konfigurationen mit identischem Grundgerüst erstellt werden können. Ein solches Server-Gerüst bietet selbst keine Funktionalität zur Netzwerkkommunikation, lokaler Nachrichtenausgabe oder Filterung. Es kann jedoch durch Aufruf der Methode `attach()` um die gewünschte Funktionalität erweitert werden. Gültige Parameter der `attach()` Methode sind Referenzen auf Objekte, die eine der Schnittstellen der jeweiligen abstrakten Basisklasse implementieren. (Siehe Tabelle 5.5.) Durch diesen modularen Aufbau ist es möglich, Server für verschiedene Anwendungsfälle zu konfigurieren und nur die Module einzubinden, die tatsächlich benötigt werden. Sender- und Empfängerimplementierungen der Transportprotokolle UDP und TCP sind bereits in der Server-Bibliothek enthalten. Eine grafische Darstellung der Nachrichten, wie in in Abbildung 5.13 zu sehen, wurde ebenfalls implementiert.

```
#include "monitor.h"
using namespace mtr;
using namespace mtr::message;

/*****************************************************************/

Message msg(SEV_ERROR, TYPE_HARDWARE,
   "Emergency Stop engaged");

mout << msg;
if (mout.state() != OK)
{
  /* error */
}

/*****************************************************************/

Message msg2(SEV_NOTICE, TYPE_HAL);
msg2.content_stream() << "set torques to: ";
for ( int i=0; i<6; ++i)
{
  msg2.content_stream() << desired_torques[i] << " ";
}

if (Monitor::instance().publish(msg2) != OK)
{
  /* error */
}

/*****************************************************************/

msg.set_content("consume ");
msg.content_stream() << hrl.timeStamp
     << "i: " << hrl.currents[0].iq.real()
     <<  ' ' << hrl.currents[1].iq.real()
     <<  ' ' << hrl.currents[2].iq.real()
     << " start: " << (unsigned int)hrl.start[0]
     << (unsignedint)hrl.start[1]
     << (unsigned int)hrl.start[2]
     << " reset " << (unsigned int)hrl.reset;

mout << msg;

/*****************************************************************/

msg.set_severity(SEV_NOTICE);
msg.reset_content();
msg.content_stream() << "Elbow Digital Board Temperature = "
   << _robot._rawData.joint1.temperatures.digital << "   C ";

Monitor::instance().publish(msg);

/*****************************************************************/
```

Quelltext 5.3: Nachrichtengenerierung und -versand

Abbildung 5.13: GUI zur lokalen Nachrichtenausgabe

Basisklasse	*rein-virtuelle Methoden*
`ReceiverAbstract`	`size_t receive(char* buffer, size_t size)`
`SenderAbstract`	`void send(Address& receiver, const char* data, size_t size),` `bool convert(Address& output, const string& text_line)`
`OutputAbstract`	`void show(MessageInterface& message)`
`FilterAbstract`	`bool filter(MessageInterface& message)`

Tabelle 5.5: Übersicht über Server-Erweiterungen

Zum Starten und Beenden aller Teile eines Server-Objekts können zwei Varianten verwendet werden: Entweder mit Hilfe der Methoden `go()` und `stop()` oder gekoppelt an den Gültigkeitsbereich eines `Runner`-Objekts (RAII-Prinzip: *resource acquisition is initialization* [51]). Erweiterungen können auch bei laufendem Server hinzugefügt (`attach()`) oder entfernt (`detach()`) werden.

Sollen Nachrichten an andere Server weitergeleitet werden, müssen zum einen die Adressen dieser Server und zum anderen die zu verwendenden Senderimplementierungen einstellbar sein. Hierzu wird für jeden Zielpunkt eine Zeichenkette an den Server-Prozess übergeben. Das Format dieser Zeichenketten entspricht:

```
Adresse [Key]
```

`Key` legt fest, welches Sendeobjekt verwendet werden soll. Der selbe Key muss bei der Instantiierung des Sendeobjekts gesetzt werden. Wird kein Key angegeben, wird UDP als Standard angenommen. Das Format der `Adresse` ist von der zu verwendenden Senderimplementierung abhängig. Eine gültige Zeichenkette für eine *TCP*-Verbindung zu Rechner *perses* und Port *12783* ist beispielsweise:

```
perses : 12783 [TCP]
```

Solche Zeichenketten können beim Start einer Server-Anwendung entweder direkt als Kommandozeilenparameter oder mit der Option `-f file` in einer Textdatei angegeben werden.

5.4 Zusammenfassung

In diesem Kapitel wurde die Implementierung der einzelnen Komponenten des Monitoring-Systems erläutert: Nachrichten werden mit Einsatz der Monitoring-Bibliothek in Clients erzeugt und zu Servern übertragen, die die Nachrichten weiterverarbeiten.

Zuerst wurde die Datenstruktur der Nachrichten festgelegt. Im 90 Byte großen Header werden Informationen gespeichert, die zur Einordnung der Nachrichten notwendig sind; im Content-Bereich kann beliebiger Text gespeichert werden.

Der Transport der Nachrichten wird von Server-Prozessen durchgeführt. Diese bestehen aus drei Teilen: Empfänger-Threads können Nachrichtenpakete von

anderen Servern empfangen und an den Reorganizer-Thread übergeben. Dieser extrahiert die einzelnen Nachrichten aus den Paketen und übergibt sie – wie ein Client – an die lokale Kommunikationsstruktur. Im Sender-Thread werden die Nachrichten aus der lokalen Kommunikationsstruktur ausgelesen, gefiltert und gegebenenfalls ausgegeben oder – in Pakete zusammengefasst – an andere Server weitergeleitet.

Zur lokalen Kommunikation zwischen Clients und Server wurde Shared Memory mit einer auf Ringbuffern basierenden Verwaltung implementiert. Für die Kommunikation zwischen Servern werden Empfänger- und Senderschnittstellen verwendet, über die die Monitoring-Infrastruktur einfach um neue Transportprotokolle erweitert werden kann. Aktuell wurden Schnittstellen für UDP und TCP implementiert.

Auf Client-Seite wird die Monitoring-Umgebung in Form eines `Monitor`-Objekts in Benutzeranwendungen eingeblendet. `Message`-Objekte können entweder durch Aufruf der Methode `publish()` des `Monitor`-Objekts oder mit Hilfe des `<<`-Operators eines `MonitorStream`s an die Monitoring-Infrastruktur übergeben werden.

Kapitel 6

Analyse der Implementierung

Im Folgenden wird vorliegende Implementierung mit den Anforderungen und Randbedingungen, die in 2 aus der Problemstellung abgeleitet wurden, verglichen. Werden alle Randbedingungen erfüllt, kann garantiert werden, dass die Monitoring-Infrastruktur ohne negative Auswirkungen auf Benutzeranwendungen oder das Gesamtsystem verwendet werden kann. Die Erfüllung der Anforderungen bestimmt den Grad, in wieweit die Aufgabenstellung umgesetzt werden konnte.

6.1 Anforderungen

6.1.1 Integration in Software-Umgebung

Die Komponenten, die zur Interaktion mit dem Monitoring-System notwendig sind, wurden in einer C++-Bibliothek – der Monitoring-Bibliothek – zusammengefasst. Da bei der Implementierung nur POSIX-konforme Schnittstellen verwendet wurden, ist die Monitoring-Bibliothek unter QNX, Linux und allen weiteren POSIX-Systemen nutzbar. Randbedingung 2.1.1 ist demnach erfüllt.

6.1.2 Datenformat

Durch den in 5.1 definierten Aufbau einer Monitoring-Nachricht ist es möglich frei formulierbaren Text zu versenden. die Länge des Texts ist jedoch begrenzt. Durch die entsprechenden Daten im Header einer Nachricht (siehe 5.1.1) ist eine Einordnung der Nachrichten in den Kontext ihrer Erzeugung durch einen Zeitstempel und Informationen über die Generierungsstelle bis auf Thread-Ebene genau möglich. Zudem wird vom Benutzer jeder Nachricht ein Typ aus Tabelle 5.1 zugeordnet. Dieser gibt an, aus welchem logischen Teil des Gesamtsystems die Nachricht stammt. Ebenso wird jede Nachricht mit einer Dringlichkeit aus Tabelle 5.2 versehen. Die Anforderungen aus 2.1.2 können demnach als voll erfüllt angesehen werden.

6.1.3 Plattform-unabhängige Datenrepräsentation

Durch die Serialisierung und das Marshalling aller Nachrichten, die über das Netzwerk verschickt werden, wird sichergestellt, dass auf allen Systemen eine Nachrichtendarstellung erzeugt werden kann, die in ihrer Bedeutung mit der Originalnachricht übereinstimmt. (Siehe 5.2.2.1.) Allerdings werden in vorliegender Implementierung die Zeichenkodierungen nicht beachtet. Da dies im aktuellen Anwendungsfall nicht notwendig ist und ein Feld zur Identifizierung von Kodierungen im Nachrichtenaufbau bereits vorgesehen ist (siehe 5.1.2.3), kann Randbedingung 2.1.3 trotzdem als erfüllt angesehen werden.

6.1.4 Einfache Anpassbarkeit an Netzwerktopologie

Alle Kommunikation über Rechnergrenzen hinweg wird von Servern abgewickelt. Ebenso wird hier festgelegt ob und wie Nachrichten lokal ausgegeben werden. (Siehe 5.2.2.) Dadurch ist es für die Clients nicht mehr notwendig die Empfänger ihrer Nachrichten zu kennen; sie kommunizieren nur mit einem Server, der auf dem selben Rechner ausgeführt wird. Durch die Fähigkeit der Server Nachrichten weiterleiten zu können, müssen Nachrichten nicht direkt an ihren Bestimmungsort übertragen werden, sondern können von Server zu Server weitergereicht werden, bis alle erforderlichen Stationen erreicht wurden. Dadurch müssen auch die Server die Ausgabepunkte nicht kennen. Servern müssen jedoch die nächsten Stationen der Nachrichten bekannt sein. Diese werden beim Start eines Servers festgelegt. (Siehe 5.3.2.) Für jeden geänderten Kommunikationsweg ist demnach ein Neustart eines Servers mit angepassten Parametern erforderlich. Somit ist ein manuelles Eingreifen notwendig, wenn die Netzwerktopologie geändert wird. Da dies allein durch Anpassung der Startparameter eines Servers möglich ist, ist Anforderung 2.1.4 erfüllt.

6.1.5 Unabhängigkeit von Kommunikations-Hardware

Da nur die obersten drei Schichten des OSI-Modells implementiert wurden, können Transportschicht und alle darunter liegenden Schichten einfach an die jeweils gegebenen Anforderungen angepasst werden. Dies ist durch die Erweiterbarkeit der Server um neue Sender und Empfänger einfach durchführbar. Dadurch ist es ebenfalls möglich, verschiedene Netzwerkschnittstellen und Transportprotokolle in einem Server parallel einzusetzen. (Siehe 5.2.2.1 und 5.2.2.2.) Für den aktuellen Systemaufbau wurden Schnittstellen für die UDP und TCP-Protokolle auf Ethernet-Basis implementiert. So müssen die Wege der Echtzeitkommunikation nicht belastet werden und es kann auf eine zweite Netzwerkanbindung ausgewichen werden. Da Implementierungen für SpaceWire und andere Kommunikationstechniken einfach zu ergänzen sind, ohne dass an der Server-Struktur Änderungen notwendig wären, ist Anforderung 2.1.5 voll erfüllt.

6.1.6 Filter

Filter werden in vorliegender Implementierung in Form von Rückrufmechanismen realisiert. (Siehe 5.2.2.1.) Damit sind sie vom Benutzer frei definierbar. Sie können auch zur Laufzeit dem Server hinzugefügt oder entzogen werden.

Es ist möglich, beliebig viele Filter in Reihe zu schalten. Eine Filterung auf Client-Seite ist momentan nicht möglich, zur Steuerung der Ausgabe und des Netzwerkverkehrs jedoch auch nicht nötig. Die Anforderung 2.1.6 ist erfüllt, könnte allerdings durch Client-seitige Filter noch verbessert werden.

6.1.7 Nachrichtenpriorisierung

Damit wichtige Nachrichten vor unwichtigen verarbeitet werden, wurde die lokale Interprozesskommunikation so ausgelegt, dass die Nachricht mit der jeweils höchsten Priorität zuerst vom Sender-Thread des Servers ausgelesen wird. Besitzen mehrere Nachrichten dieselbe Prioritätsstufe, werden diese nach dem FIFO-Prinzip abgearbeitet. (Siehe 5.2.3.) Es ist somit gewährleistet, dass die jeweils wichtigste und älteste Nachricht vor allen anderen in einem Transportpaket gespeichert wird. Daraus ergibt sich, dass die Dringlichkeit der Nachrichten innerhalb eines Paketes von vorne nach hinten abnimmt. Der Reorganizer auf Empfängerseite beginnt mit dem Auslesen der Pakete dementsprechend bei der wichtigsten enthaltenen Nachricht. Da die ausgelesenen Nachrichten wieder in die lokale – prioritätsgewahre – Kommunikationsstruktur eingefügt wird, werden wichtige Nachrichten auf ihrem gesamten Weg bis zur Ausgabe bevorzugt behandelt.

Es muss jedoch darauf hingewiesen werden, dass das Einspeisen einer Nachricht in die lokale Kommunikationsstruktur auch vom Scheduling-Verhalten abhängt. Das bedeutet, dass ein Prozess/Thread nur Nachrichten versenden kann, wenn er ausgeführt wird. Da der Server-Prozess im Hintergrund der Benutzeranwendungen arbeitet, kann es durch ungünstiges Scheduling vorkommen, dass Receiver oder Reorganizer nicht in der Lage sind, eingehende Nachrichten sofort zu verarbeiten. Um nun vor dem Versenden eine korrekte Priorisierung aller Nachrichten zu garantieren, müssten Receiver- und Reorganizer-Thread höhere Prioritäten zugewiesen werden als dem Sender-Thread. Dieses Vorgehen würde bei hohem Nachrichtenaufkommen jedoch dazu führen, dass die lokale Kommunikationsstruktur voll läuft, aber keine Nachrichten ausgelesen werden können. Um den unter diesen Bedingungen auftretenden Nachrichtenverlust zu vermeiden, arbeiten alle Threads des Servers auf der selben Prioritätsstufe, so dass auch der Sender-Thread ausgeführt werden kann, wenn kein Prozess höherer Priorität lauffähig ist. Die damit einhergehende Möglichkeit der Verletzung der Nachrichtendringlichkeiten wird zu Gunsten einer Vermeidung von Nachrichtenverlust in Kauf genommen.

Werden einem Server-Prozess genug Ressourcen zur Verfügung gestellt, um alle Nachrichten zu verarbeiten, werden hochpriore Nachrichten auch in jedem Fall bevorzugt behandelt. Ansonsten kann dies nicht garantiert werden. Anforderung 2.1.7 ist demzufolge mit Einschränkung erfüllt.

6.1.8 Benutzerschnittstelle

Die Schnittstelle zur Veröffentlichung von Nachrichten aus einer Benutzeranwendung heraus folgt dem Konzept objektorientierter Programmierung. Nachrichten werden als Objekte erzeugt, die an ein `Monitor`-Objekt übergeben werden, das die gesamte Monitoring-Infrastruktur abstrahiert. (Siehe 5.3.1.)

Durch diese Abstraktion wird die Komplexität der Nachrichtenverarbeitung vor Anwendungsentwicklern verborgen. Die Generierung einer Nachricht erfolgt durch Instantiierung eines `Message`-Objekts und muss im Weiteren vom Benutzer nicht mehr verändert werden. Es werden jedoch Schnittstellen angeboten, um eine nachträgliche Änderung des Typs, der Dringlichkeit und des Inhalts einer Nachricht neu zu setzen bzw. zu ergänzen. Neben der Möglichkeit den Nachrichteninhalt komplett zu überschreiben, kann dieser auch über eine `std::ostream`-Implementierung aufgebaut werden. In Bezug auf die Client-Seite kann Anforderung 2.1.8 als erfüllt betrachtet werden.

Zur Konfiguration der Nachrichtenverarbeitung werden vier abstrakte Basisklassen zur Verfügung gestellt. Spezialisierungen dieser Klassen können instantiiert und – in beliebiger Anzahl – in den Kontrollfluss eines Server-Prozesses eingebunden werden. (Siehe 5.3.2.) Auf diese Weise können Server einfach um frei definierbare Filter, Ausgabe- und Transportmechanismen erweitert werden. Hinzufügen und Entfernen solcher Objekte ist zu jedem Zeitpunkt möglich. Aufgrund dieser Flexibilität und da gemeinsame Funktionalität bereits in den Basisklassen enthalten ist, kann Anforderung 2.1.8 auch auf Server-Seite als erfüllt angesehen werden.

6.1.9 Zeitstempel

Jede Nachricht wird bei ihrer Veröffentlichung mit einem nanosekundengenauen Zeitstempel versehen. (Siehe 5.1.1.5.) Da der Zeitstempel der Systemzeit des jeweiligen Systems entspricht, hängt die Qualität des Zeitstempels von der Genauigkeit der Systemzeit ab. Wird angenommen, dass die Systemzeit aller beteiligten Rechner exakt genug ist, um die Reihenfolge verschiedener Nachrichten rekonstruieren zu können, ist Anforderung 2.1.9 erfüllt. Die Synchronisierung der Zeitgeber ist jedoch nicht Teil dieser Arbeit.

6.1.10 Erkennung von Nachrichtenverlust

Durch die Einführung von Sequenznummern werden verlorene Nachrichten erkannt. Dieser Verlust wird durch Generierung einer Monitoring-Nachricht bekannt gemacht, sobald dem Server ausreichend Zeit hierfür zur Verfügung hat. (Siehe 5.2.2.4.) Da verlorene Nachrichten nur erkannt werden, wenn Lücken im Strom der Sequenznummern auftauchen, ist eine Erkennung des Verlusts der jeweils letzten Nachricht mit diesem Prinzip nicht möglich. Soll sichergestellt werden, dass beim Transport über das Netzwerk keine Nachrichten verloren gehen, muss ein Transportprotokoll verwendet werden, das Datenzustellung garantiert.

Werden Nachrichten verloren bevor ihre Sequenznummer gesetzt wird – also vor dem ersten Auslesen einer Nachricht durch ein Sende-Modul – wird dies auf Server-Seite nicht erkannt. Bei korrekter Funktion der Monitoring-Infrastruktur ist der einzig mögliche Grund für den Verlust einer Nachricht ohne Sequenznummer das Fehlschlagen der Veröffentlichung auf Client-Seite. In diesem Fall wird jedoch der Client über den Fehler informiert (siehe 5.3.1.2), so dass dieser den Verlust geeignet behandeln kann. Anforderung 2.1.10 ist daher mit Einschränkung erfüllt.

6.2 Randbedingungen

6.2.1 Neutralität bezüglich Benutzeranwendungen

Unter QNX werden den Server-Prozessen Prioritäten unterhalb der Prioritäten der Benutzeranwendungen zugeordnet. Da an den Schnittstellen zu Benutzeranwendungen keine Synchronisierungsmechanismen eingesetzt werden, die das Prozess-Scheduling beeinflussen, und die Kommunikation zwischen Clients und Servern asynchron abläuft, ist unter QNX Randbedingung 2.2.1 erfüllt. Unter Linux wird Scheduling nicht strikt nach Prozessprioritäten gehandhabt, so dass die Beeinflussung der Benutzeranwendungen durch Monitoring-Prozesse nicht vermieden werden. Dies ist jedoch auch nicht im selben Maße notwendig wie unter QNX, da unter Linux keine echtzeitkritischen Prozesse ausgeführt werden.

6.2.2 Determinismus

Sowohl die Nachrichtengenerierung als auch die Nachrichtenveröffentlichung verwenden ausschließlich determinierte Funktionsaufrufe. Dies ist möglich, da die Datenstruktur der Nachrichten fest angelegt wurde und somit dynamische Speicherallozierung nicht verwendet werden muss. (Siehe 5.1.) Weiterhin werden keine Synchronisierungsmechanismen benötigt, die das Scheduling-Verhalten der Prozesse beeinflussen oder zur Blockierung einer Benutzeranwendung führen könnten. (Siehe 5.2.3.) Ein Nachweis hierzu ist in Abbildung 6.1 zu finden, in der erkennbar ist, dass alle Veröffentlichungen einer Nachricht innerhalb einer festen Zeitspanne abgeschlossen wurden. Randbedingung 2.2.2 ist demnach erfüllt.

6.2.3 Begrenzte Ausführdauer

Insbesondere die Ausführdauer einer Nachrichtenveröffentlichung ist begrenzt, da diese auch aus dem echtzeitkritischen Kontrollfluss von Benutzeranwendungen ausführbar sein muss. Die Erzeugung einer Nachricht kann hingegen schon in der nicht echtzeitkritischen Initialisierungsphase erfolgen. Daher wurde möglichst viel Funktionalität von der Nachrichtenveröffentlichung in die Initialisierung bzw. in den Server-Prozess verlagert. Alle Informationen, die schon vor dem Zeitpunkt der Veröffentlichung bekannt sein können, werden auch im Vorfeld gesetzt. So müssen bei einer Veröffentlichung nur noch Zeitstempel, Sequenznummer und Thread-Informationen gesetzt und die Nachricht in die lokale Kommunikationsstruktur kopiert werden. (Siehe 5.3.1.)

Die Zeiten, die hierfür nötig sind, wurden gemessen und mit denen einer entsprechenden Veröffentlichung mit syslog und einer Konsolenausgabe mit `printf()` verglichen. Die Ergebnisse dieser Messungen sind in Abbildung 6.1 zu sehen. Mit einem Mittelwert von 1,79 µs und einer WCET von 12,75 µs, ist vorliegende Implementierung ca. 25 mal schneller als QNXs syslog und ca. 120 mal schneller als `printf`. Weitere Tests im laufenden Betrieb des Robotersystems ergaben, dass diese Werte niedrig genug sind, um einen störungsfreien Betrieb zu ermöglichen. Randbedingung 2.2.3 ist damit erfüllt.

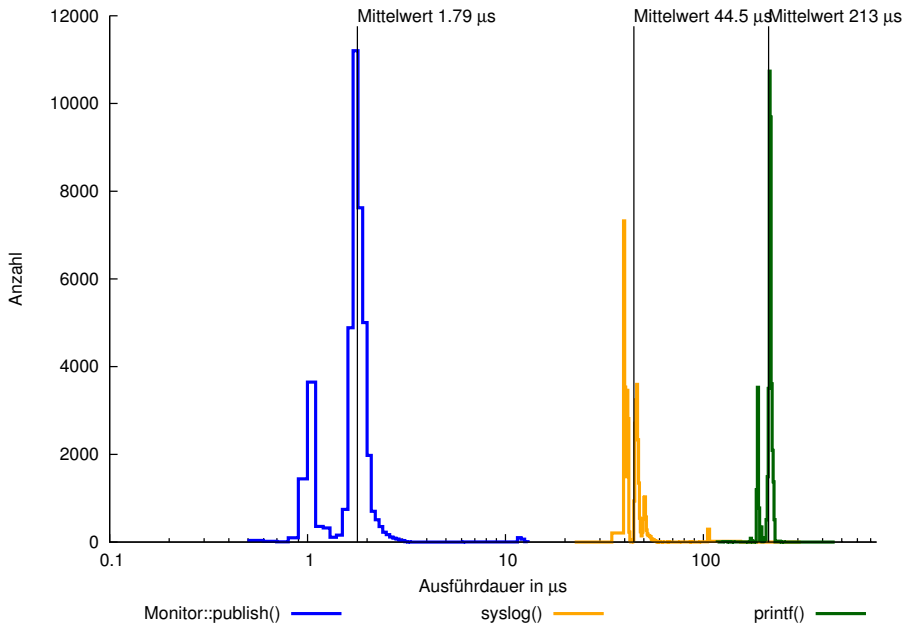


Abbildung 6.1: Vergleich der Dauer der Nachrichtenveröffentlichungen mit syslog, einer Konsolenausgabe mit `printf()` und der Monitoring-Bibliothek

6.2.4 Geringe Netzwerkbelastung

Nachrichten eines Servers respektive Netzwerkknotens werden, soweit dies sinnvoll ist, vor dem Versand zu anderen Servern gefiltert und gepackt. (Siehe 5.2.2.1.) Dadurch wird der Nutzdatenanteil im Vergleich zur Header-Größe des Transportprotokolls verbessert und damit die Netzwerkbelastung reduziert. Weiterhin werden von der Monitoring-Infrastruktur nur Nutzdaten über das Netzwerk übertragen; Steuerungsnachrichten oder ähnliches sind nicht notwendig.

Die tatsächliche Netzwerkbelastung hängt jedoch auch vom verwendeten Transportprotokoll und dessen Overhead ab. Da dieses einerseits frei wählbar ist und andererseits UDP – ein Protokoll mit geringem Overhead – als Standard verwendet wird, ist auch Randbedingung 2.2.4 erfüllt.

Kapitel 7

Zusammenfassung und Ausblick

Ziel dieser Arbeit war es, ein System aufzubauen, über das Nachrichten aus echtzeitkritischen Anwendungen heraus an andere Netzwerkknoten weitergegeben werden können. Die Kommunikationswege sollen dabei möglichst einfach an veränderte Systemkonfigurationen anpassbar sein. Nachrichten sollen einerseits frei formulierbaren Inhalt und andererseits definierte Informationen zum Kontext der Nachrichten enthalten.

Da alle Randbedingungen aus 2.2 in vorliegender Implementierung erfüllt wurden, kann garantiert werden, dass sich der Einsatz der Monitoring-Infrastruktur nicht negativ auf Benutzeranwendungen auswirkt. Die Anforderungen, die sich aus der Aufgabenstellung ergaben (siehe 2.1), wurden ebenfalls erfüllt, unterliegen jedoch teilweise noch Einschränkungen. Um dieses Einschränkungen aufzuheben und um das Gesamtsystem weiter zu verbessern, werden im Folgenden noch einige Möglichkeiten und Lösungsvorschläge für kommende Arbeiten im Zusammenhang mit der Monitoring-Infrastruktur aufgeführt. Abschließend wird nochmals ein Überblick über die entstandene Monitoring-Infrastruktur gegeben und ein Ergebnis der Arbeit formuliert.

7.1 Optimierungsmöglichkeiten und zukünftige Arbeiten

7.1.1 Zeichenkodierung

Momentan wird die Zeichenkodierung des zu übertragenden Textes nicht berücksichtigt. Eine Konvertierung zwischen verschiedenen Zeichenkodierungen erfordert zwei Informationen: Zum einen einen Tabellensatz, der für die verschiedenen Kodierungen die Übersetzung von Zahlenwerten in Schriftzeichen ermöglicht, und zum anderen einen Schlüssel, nach dem die Tabelle ausgesucht werden kann, die der gewünschten Zeichenkodierung entspricht. Ein Feld, um einen solchen Schlüssel zu übertragen, wurde im Content-Teil der Nachrichten bereits vorgesehen. Die Übersetzungstabellen könnten in der Monitoring-

Bibliothek hinterlegt werden. Bei der Ausgabe einer Nachricht müsste dann nur die entsprechende Tabelle zur Zeichendarstellung verwendet werden.

7.1.2 Filter

In der aktuellen Implementierung müssen Server-Anwendungen nach Änderungen an einer Erweiterung zur Netzwerkkommunikation, Filterung oder lokalen Ausgabe neu kompiliert werden. Bei Kommunikations- und Ausgabemodulen stellt dies kein Problem dar, da diese in der Regel dauerhaft bestehen bleiben. Im Fall der Filter kann es jedoch erforderlich sein, die Parameter, nach denen Nachrichten verworfen oder weiterverarbeitet werden, häufig zu ändern. Um dies zu ermöglichen wäre ein Filter wünschenswert, der mit Hilfe von Skripten konfiguriert werden kann. Die Interpretierung der Skripte könnte im laufenden Betrieb durchgeführt werden, was allerdings zu einer erhöhten Verarbeitungsdauer der Nachrichten führen würde. Alternativ könnten die Skripte – ähnlich der Empfängerliste – in der Initialisierungsphase der Server geladen werden. Dies würde die Flexibilität zwar wieder reduzieren, jedoch nur einen Neustart der Server-Anwendung erfordern und keinen erneuten Kompilierungsvorgang.

Weiterhin könnte es die Konfiguration von Server-Anwendungen erleichtern, wenn Sende- und Ausgabemodulen jeweils separat Filter vorgeschaltet werden könnten. So könnten beispielsweise Nachrichten hoher Priorität über ein zuverlässiges Transportprotokoll versendet werden, während unwichtigere Nachrichten ein weniger Overhead produzierendes Protokoll verwenden. Ein solches Verhalten ist mit der aktuellen Implementierung zwar auch möglich, muss jedoch direkt in die entsprechenden Sende- bzw. Ausgabemodule integriert werden.

Zusätzlich zu den Server-seitigen Filtern könnten einfache Filter innerhalb der Benutzeranwendungen dazu beitragen, Ressourcen nicht unnötig zu belegen. Eine Möglichkeit Filter auf Client-Seite zu implementieren ist, die Monitoring-Bibliothek nicht direkt zu nutzen, sondern ein Logging-Front-End wie Pantheios [12] als Schnittstelle zur Monitoring-Infrastruktur zu verwenden.

7.1.3 Zeitstempel

Wie bereits mehrfach erwähnt, ist ein Zeitstempel in einer Nachricht nur sinnvoll, wenn die Zeitgeber aller beteiligten Rechner synchronisiert wurden, da ansonsten eine Rekonstruktion von Ereignisabfolgen nicht durchgeführt werden kann. Dieses Problem kann jedoch nicht innerhalb der Monitoring-Infrastruktur gelöst werden, sondern erfordert die Synchronisierung der Systemuhren. Im aktuellen Aufbau wird dies mit Hilfe des NTP-Protokolls [41] bereits durchgeführt. Da NTP allerdings für die Verwendung in großen, unzuverlässigen Netzwerken wie dem Internet entwickelt wurde, könnte mit einem Verfahren, das auf kleine, zuverlässige Netzwerke optimiert ist, die Qualität der Uhrensynchronisierung vermutlich noch deutlich verbessert werden. In Frage kommt hierfür beispielsweise das *Precision Time Protocol* [29].

Da eine exakte Synchronisierung nicht möglich ist, wäre es sinnvoll, jeder Nachricht zusätzlich Informationen über die Unsicherheit ihres Zeitstempels hinzuzufügen, um so zumindest einen Anhaltspunkt bei der Auswertung der Daten

zu haben. Aktuell liegen solche Daten allerdings nicht vor; dies könnte jedoch bei der Implementierung eines anderen Synchronisierungsprotokolls berücksichtigt werden.

7.1.4 Reduzierung der Veröffentlichungsdauer

Um die Dauer, die zur Veröffentlichung einer Nachricht benötigt wird, weiter zu reduzieren, könnten einige zusätzliche Header-Daten vor oder nach der Veröffentlichung gesetzt werden. So ist dem Benutzer in der Regel bekannt, aus welchem Thread eine Nachricht verschickt werden wird. Er könnte diese Information also bereits bei der Nachrichtengenerierung setzen. Die Korrektheit dieser Daten läge dann jedoch in der Verantwortung des Benutzers.

7.1.5 Reduzierung der Netzwerkbelastung

Sollte das Netzwerk durch die Monitoring-Infrastruktur zu stark belastet werden, könnte dies durch zusätzliche Logik des Senders und des Reorganizers verringert werden. Beispielsweise könnten die serialisierten Nachrichten vor dem Versand komprimiert werden, um so die Menge der Daten, die übertragen werden muss, zu reduzieren. Ein anderer Ansatz könnte es sein, ähnliche Nachrichten zusammenzufassen. Besitzen mehrere Nachrichten z.B. den selben Inhalt und unterscheiden sich nur durch ihre Header, so würde es ausreichen statt aller Nachrichten nur eine zu verschicken und in dieser zu kennzeichnen, wie viele solcher Nachrichten aufgetreten sind.

7.1.6 FPGA Implementierung

In dieser Arbeit wurde die Monitoring-Infrastruktur für C++-Anwendungen auf QNX/Linux-Plattformen implementiert. Um auch Elemente der Gelenk-Ebene des MiroSurge-Szenarios überwachen zu können, ist eine Implementierung notwendig, die auf FPGAs eingesetzt werden kann. Außerdem ist die Implementierung eines SpaceWire-Empfängermoduls erforderlich, um Monitoring-Daten der Gelenk-Ebene in darüber liegenden Ebenen verarbeiten zu können.

7.2 Überblick über Monitoring-Infrastruktur

In vorliegender Arbeit wurden die obersten drei Schichten des OSI-Schichtenmodells zum Transport von Monitoring-Nachrichten über ein Netzwerk implementiert:

Die Benutzerschnittstelle (*Anwendungsschicht*), die in Anwenderprogrammen sichtbar ist, folgt dem Konzept der objektorientierten Programmierung. Die gesamte Monitoring-Infrastruktur wird hierbei durch ein `Monitor`-Objekt bzw. einen `MonitorStream` dargestellt, dem Nachrichtenobjekte übergeben werden können. Alle Attribute eines Nachrichtenobjekts werden automatisch gesetzt; Inhalt, sowie Typ und Dringlichkeit, können jedoch vom Benutzer überschrieben bzw. ergänzt werden. Die Konfiguration der Nachrichtenverarbeitung erfolgt über Rückrufmechanismen – ähnlich den Appendern aus log4j. D.h. Benutzer können

eigene Filter-, Ausgabe-, Empfangs- und Sendeobjekte erstellen und diese bei einem Server registrieren.

Das Format der Nachrichten (*Darstellungsschicht*) wurde dem des syslog-Protokolls nachempfunden, jedoch an die gegebenen Anforderungen angepasst. So wurden die Aufteilung der Nachrichten in Header und Content, sowie die starre Speicheraufteilung übernommen. Im Header werden Informationen zum Ursprung und der Art einer Nachricht eingetragen, also die Adresse des Netzwerkknotens, die Namen der Anwendung und des Verantwortlichen, Informationen zu Prozess und Thread, der Zeitpunkt der Veröffentlichung, eine Sequenznummer, Typ und Dringlichkeit. Im Content-Bereich wird ein Feld zur Verfügung gestellt, in das vom Benutzer frei formulierbarer Text eingetragen werden kann. Zusätzlich werden hier noch Informationen über Länge des Textes und verwendeten Zeichensatz hinterlegt.

Die Übertragungsinfrastruktur (*Kommunikationsschicht*) stellt eine Kombination aus der Client/Server-Architektur des syslog-Protokolls und des modularen Konzepts der meisten Logging-Frameworks dar. Anwenderprogramme, die die Monitoring-Bibliothek zur Generierung von Nachrichten nutzen, sind hierbei die Clients. Sie müssen in ihrem Kontrollfluss lediglich die Nachrichten erzeugen. Der Transport und die Ausgabe der Nachrichten wird von einem Server je Netzwerkknoten übernommen. Server bestehen aus drei Teilen: Vom Sender werden Nachrichten aus der lokalen Kommunikationsstruktur ausgelesen, gefiltert, gegebenenfalls ausgegeben und – serialisiert und in Pakete verpackt – über das Netzwerk an weitere Server gesendet. Receiver dienen zum Empfang von Nachrichtenpaketen, die von Servern anderer Netzwerkknoten versandt wurden. Diese Pakete werden an einen Reorganizer weitergegeben, der sie entpackt und – wie ein Client – in die lokale Kommunikationsstruktur einspeist. Filterung, Ausgabe, Empfang und Versand von Nachrichten werden durch Server-Erweiterungen realisiert. Die entsprechenden Methoden der Erweiterungen werden automatisch aus dem Server-Kontrollfluss heraus ausgeführt.

7.3 Ergebnis

In Kapitel 6 wurde nachgewiesen, dass durch vorliegende Implementierung alle Randbedingungen erfüllt sind, die einen sicheren Betrieb des Systems garantieren. Die Anforderungen wurden ebenfalls weitgehend erfüllt. Allerdings konnten einige Punkte nicht vollständig gelöst werden. Diese Anforderungen ergaben sich aus einer formalen Analyse der Aufgabenstellung, so dass erst durch den tatsächlichen Einsatz der Monitoring-Infrastruktur festgestellt werden kann, ob die Implementierung alle praktisch notwendigen Bedingungen erfüllt.

In einem ersten Test wurde die maximale Veröffentlichungsfrequenz ermittelt, bei der noch alle Nachrichten verarbeitet werden können. Hierzu wurde ein ansonsten lastloses QNX-System verwendet, auf dem eine Client-Anwendung in definierten Zeitabständen Nachrichten an einen Server übergab, der diese Nachrichten ungefiltert per UDP versendete. Durch sukzessive Reduzierung der Zeitabstände wurde ermittelt, dass auf dem Testrechner bis zu 80000 Nachrichten pro Sekunde verarbeitet werden können, ohne dass Nachrichten

verloren werden. Dies entspricht einer mittleren Zeitspanne von 12.5 µs von der Veröffentlichung einer Nachricht aus dem Client bis zum Versand durch den Server.

Da in diesem Testaufbau ein Doppelkernprozessor eingesetzt wurde und nur die beiden Prozesse Client und Server ausgeführt wurden, konnten hier Scheduling-Effekte noch nicht berücksichtigt werden. Um das Verhalten in einem realen Anwendungsfall zu untersuchen, wurde in einer 3 kHz Regelschleife eines Rechners des MiroSurge Szenarios in jedem Zyklus eine Nachricht veröffentlicht. Auch dies war im laufenden Betrieb des des MiroSurge Szenarios ohne Nachrichtenverlust möglich. Eine Nachrichtenveröffentlichung mit so hoher Frequenz stellt jedoch einen Extremfall dar, der üblicherweise nicht auftreten sollte. Damit erfüllt die Implementierung am Ende der Netzwerkpfade die Aufgabenstellung.

Die Weiterleitung von Nachrichten anderer Netzwerkknoten erfordert mehr Rechenzeit des Servers, da Empfänger, Reorganizer und Sender durchlaufen werden müssen. Der Empfang und die Ausgabe der Nachrichten des Regelungsrechners waren jedoch auf einem Linux-System noch möglich, so dass auch dieser Teil des Servers als ausreichend schnell angesehen werden kann.

Durch die Analyse der Prozesse des Regelungsrechners wurde weiterhin nachgewiesen, dass die Beeinträchtigung der Benutzerprozesse durch die Monitoring-Infrastruktur lediglich auf die Nachrichtenveröffentlichung zurückzuführen ist und gering genug ausfällt, um den Betrieb des Systems nicht zu stören. Damit kann die Aufgabenstellung als erfüllt angesehen werden.

Literaturverzeichnis

[1] *AUTOSAR*. http://www.autosar.org. – zuletzt abgerufen am 05.05.2010

[2] *Boost.Log*. http://boost-log.sourceforge.net. – zuletzt abgerufen am 19.04.2010

[3] *Datagram Congestion Control Protocol – Implementations*. http://www.read.cs.ucla.edu/dccp/#implementations. – zuletzt abgerufen am 19.04.2010

[4] *google-glog*. http://code.google.com/p/google-glog. – zuletzt abgerufen am 19.04.2010

[5] *IEEE Std 1003.1™–2008*. http://www.opengroup.org/onlinepubs/9699919799/. – zuletzt abgerufen am 19.04.2010

[6] *List of SCTP Implementations*. http://www.sctp.org/implementations.html. – zuletzt abgerufen am 19.04.2010

[7] *log4cplus*. http://log4cplus.sourceforge.net. – zuletzt abgerufen am 19.04.2010

[8] *log4cpp*. http://log4cpp.sourceforge.net. – zuletzt abgerufen am 19.04.2010

[9] *log4cxx*. http://logging.apache.org/log4cxx. – zuletzt abgerufen am 19.04.2010

[10] *MathWorks Simulink*. http://www.mathworks.com/products/simulink/. – zuletzt abgerufen am 19.04.2010

[11] *Modular Syslog*. http://msyslog.sourceforge.net. – zuletzt abgerufen am 19.04.2010

[12] *Pantheios*. http://www.pantheios.org. – zuletzt abgerufen am 19.04.2010

[13] QNX Software Systems: *QNX documentation – native networking (Qnet)*. http://www.qnx.com/developers/docs/6.4.1/neutrino/sys_arch/qnet.html. – zuletzt abgerufen am 19.04.2010

[14] *QNX Neutrino IPC*. http://www.qnx.com/developers/docs/6.3.0SP3/neutrino/sys_arch/kernel.html#NTOIPC

[15] *QNX Neutrino RTOS*. http://www.qnx.com/products/neutrino-rtos/neutrino-rtos.html. – zuletzt abgerufen am 19.04.2010

[16] *RLog*. http://www.arg0.net/rlog. – zuletzt abgerufen am 19.04.2010

[17] *RSyslog*. http://www.rsyslog.com. – zuletzt abgerufen am 19.04.2010

[18] *Syslog-ng logging system*. http://www.balabit.com/network-security/syslog-ng/. – zuletzt abgerufen am 19.04.2010

[19] *ISO/IEC standard 7498-1:1994*. November 1994

[20] *Internet Assigned Numbers Authority: Character sets*. Mai 2007. – zuletzt abgerufen am 19.04.2010

[21] Arndt, J.: *Matters computational*. http://www.jjj.de/fxt, November 2009

[22] Blanc, B. ; Maaraoui, B. *Endianness or Where is Byte 0?* White paper. Dezember 2005

[23] Boyd, A.: *QNX core networking – Quelltext*. http://community.qnx.com/svn/repos/core_networking/trunk/lib/io-pkt/sys/lsm/qnet/l4_lite/l4_resolver_en_ionet.c. – Revision 151

[24] Burgess, C. *No fault of your own...* http://sendreceivereply.wordpress.com/2008/04/18/no-fault-of-your-own/. April 2008

[25] Cerf, V.G. *ASCII format for network interchange*. RFC 20. Oktober 1969

[26] Deering, S. ; Hinden, R.: *Internet protocol, version 6 (IPv6) specification*. RFC 2460 (Draft Standard). Dezember 1998 (Request for comments). – Updated by RFC 5095

[27] Dijkstra, E.: The structure of the "THE"-multiprogramming system. In: *Commun. ACM* 11 (1968), Nr. 5, S. 341–346. – ISSN 0001–0782

[28] Dolev, D. ; Halpern, J. ; Strong, H.: On the possibility and impossibility of achieving clock synchronization. In: *STOC '84: Proceedings of the sixteenth annual ACM symposium on theory of computing*. New York, NY, USA : ACM, 1984. – ISBN 0–89791–133–4, S. 504–511

[29] Eidson, J. ; Kneifel, R. ; Moldovansky, A. ; Woods, S.: Synchronizing Measurement and Control Systems. In: *Sensors Magazine* 19 (2002), November, Nr. 11

[30] ESA: *SpaceWire*. http://spacewire.esa.int. – zuletzt abgerufen am 19.04.2010

[31] Gamma, E. ; Helm, R. ; Johnson, R. ; Vlissides, J.: *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995

[32] Gerhards, R. *The Syslog protocol*. RFC 5424 (Proposed Standard). März 2009

[33] Grinnemo, K. ; Andersson, T. ; Brunstrom, A.: Performance benefits of avoiding head-of-line blocking in SCTP. In: *Proceedings of the ICAS/ICNS 2005*. Papeete, Tahiti, 2005

[34] Gülcü, C.: *Apache Logging Services*. `http://logging.apache.org`. Januar 2007. – zuletzt abgerufen am 19.04.2010

[35] Hagn, U. ; Konietschke, R. ; Tobergte, A. ; Nickl, M. ; Jörg, S. ; Kübler, B. ; Passig, G. ; Gröger, M. ; Fröhlich, F. ; Seibold, U. ; Le-Tien, L. ; Albu-Schäffer, A. ; Nothhelfer, A. ; Hacker, F. ; Grebenstein, M. ; Hirzinger, G.: DLR MiroSurge: a versatile system for research in endoscopic telesurgery. In: *International journal of computer assisted radiology and surgery* 5 (2010), März, Nr. 2, S. 183–193. – ISSN 1861–6410

[36] Hagn, U. ; Nickl, M. ; Jörg, S. ; Passig, G. ; Bahls, T. ; Nothhelfer, A. ; Hacker, F. ; Le-Tien, L. ; Albu-Schäffer, A. ; Konietschke, R. ; Grebenstein, M. ; Warpup, R. ; Haslinger, R. ; Frommberger, M. ; Hirzinger, G.: The DLR MIRO: a versatile lightweight robot for surgical applications. In: *Industrial robot - An international journal* 35 (2008), Nr. 4, S. 324–336. – ISSN 0143–991X

[37] Haus iC: *BiSS Interface*. `http://www.biss-interface.com`. – zuletzt abgerufen am 19.04.2010

[38] Kohler, E. ; Handley, M. ; Floyd, S. *Datagram Congestion Control Protocol (DCCP)*. RFC 4340 (Proposed Standard). März 2006

[39] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. In: *Commun. ACM* 21 (1978), Nr. 7, S. 558–565. – ISSN 0001–0782

[40] Martin, R.: The dependency inversion principle. In: *C++ Report*. 1996, S. 61–66

[41] Mills, D. *Network Time Protocol (version 3) specification, implementation and analysis*. RFC 1305 (Draft Standard). März 1992

[42] Postel, J. *User Datagram Protocol*. RFC 768 (Standard). August 1980

[43] Postel, J.: *Internet protocol*. RFC 791 (Standard). September 1981 (Request for comments). – Updated by RFC 1349

[44] Postel, J.: *Transmission Control Protocol*. RFC 793 (Standard). September 1981 (Request for comments). – Updated by RFCs 1122, 3168

[45] Kap. Basics of the unix philosophy In: Raymond, E.: *The art of unix programming*. Addison-Wesley, 2003

[46] Rosson, M. ; Alpert, S.: The cognitive consequences of object-oriented design. In: *Human-computer interaction* 5 (1990), Dezember, Nr. 4, S. 345–379

[47] Rushworth, T. ; Telfer, A. *Multi-reader, multi-writer lock-free ring buffer*. Schutzrecht US 2009/0204755 A1. August 2009

[48] Kap. An overview of clock synchronization In: Simons, B. ; Welch, J. ; Lynch, N.: *Springer lecture notes in computer science: fault-tolerant distributed computing*. London, UK : Springer, 1990, S. 84–96. – ISBN 0387973850

[49] Slemko, M. *Path MTU discovery and filtering ICMP.* White paper. Oktober 2006

[50] Stewart, R. *Stream Control Transmission Protocol.* RFC 4960 (Proposed Standard). September 2007

[51] Kap. 14.4 Resource Management In: Stroustrup, B.: *The C++ programming language.* 3. Addison-Wesley, 1997

[52] Tanenbaum, A.: *Modern operating systems.* 2. Prentice Hall PTR, 2001. – ISBN 0130313580

[53] Tanenbaum, A.: *Computer networks.* 4. Prentice Hall PTR, 2003. – ISBN 00130661023

[54] Wilhelm, R. ; Engblom, J. ; Ermedahl, A. ; Holsti, N. ; Thesing, S. ; Whalley, D. ; Bernat, G. ; Ferdinand, C. ; Heckmann, R. ; Mitra, T. ; Mueller, F. ; Puaut, I. ; Puschner, P. ; Staschulat, J. ; Stenström, P.: The worst-case execution time problem – Overview of methods and survey of tools. In: *ACM transactions on embedded computing systems* 7 (2008), Nr. 3, S. 1–53. – ISSN 1539–9087

[55] Yergeau, F. *UTF-8, a transformation format of ISO 10646.* RFC 3629 (Standard). November 2003