Fachbereich 2 - Technik, Informatik und Wirtschaft

Studiengang Master Informationssysteme

Entwicklung einer Schnittstelle für verteiltes I/O in einer neuen on-board Computerarchitektur

Masterthesis

vorgelegt am 27.08.2013

Benjamin Weps

Matrikel-Nr. 194921

Betreuer (FH): Prof. Dr.-Ing. Cornelius Wille

Betreuer (extern): Daniel Lüdtke, DLR Braunschweig

Ehrenwörtliche Erklärung

Ich versichere, dass ich die vorliegende Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich habe die Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Braunschweig, 27. August 2013

Benjamin Weps

Zusammenfassung

Diese Arbeit stellt ein Teil des Projekt On-Board-Computer Next-Generation (OBC-NG) des Deutschen Zentrum für Luft- und Raumfahrt dar. In diesem Projekt wird ein Konzept für das verteilte Rechnen auf Raumfahrtsystemen angestrebt. Diese Arbeit beschäftigt sich dabei mit dem Interface-Node dieses Konzeptes, welcher die Schnittstelle zu den Sensoren und Aktuatoren des Raumfahrtsystems darstellt.

Zunächst wird in dieser Arbeit ein Überblick über die aktuellen Techniken im Bereich der verteilten Systeme gegeben. Aus den verfügbaren Techniken wird danach ein Softwarekonzept für die Verwendung in eingebetteten Systemen entworfen. Für die Evaluierung wird dieses Konzept in einem Prototyp für eingebettete Systeme und einer Simulationsumgebung umgesetzt. Anhand dieser Implementierungen findet eine Bewertung der Techniken für den Einsatz in dem OBC-NG System statt.

Diese Arbeit zeigt, dass die Techniken aus dem High-Performance-Computing Umfeld, in denen für gewöhnlich Ressourcenknappheit keine Rolle spielt auf die stark eingeschränkten Möglichkeiten der eingebetteten Systeme übertragen werden können.

Abstract

This work is a part of the On-Board-Computer Next-Generation (OBC-NG) project by the German Aerospace Centre. The OBC-NG project surveys a concept for distributed computing on space systems. This thesis is about the Interface-Node component which provides access to the sensors and actuators of the space system.

At first there will be an overview of existing techniques in the field of distributed systems. Out of this a concept will be developed for use in embedded systems. To evaluate the concept a prototype for a simulation evironment and for embedded systems will be implemented. Based on this implementation the applied techniques will be evaluated for the use in the OBC-NG System.

This thesis shows, that techniques which are used in High-Performance-Computing (HPC) Systemd can be adapted to embedded Systems with severe resource restrictions.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation . . . 1
1.2 Das Projekt OBC-NG . . . 1
1.3 Aufgabenstellung . . . 2
1.4 Aufbau der Arbeit . . . 2

2 Stand der Technik **4**
2.1 Hardware in Raumfahrtsystemen . . . 4
2.2 Netzwerktechnik in Raumfahrtsystemen . . . 4
2.3 Verteilte Ein- und Ausgabe . . . 5
2.4 Bereitstellungsmodi . . . 6
2.4.1 Push-Betrieb . . . 6
2.4.2 Pull-Betrieb . . . 7
2.5 Interprozess Kommunikation . . . 7
2.5.1 Shared-Memory . . . 8
2.5.2 Datenströme . . . 8
2.5.3 Remote Procedure Call . . . 8
2.5.4 Message-Passing . . . 9
2.6 Message Oriented Middleware . . . 9
2.7 Distributed Message Queue . . . 10
2.8 Message Marshalling . . . 11
2.8.1 Textdarstellung . . . 12
2.8.2 ASN.1 . . . 12
2.8.3 Protocol Buffers . . . 13
2.9 Transportsicherung . . . 13
2.9.1 Error Correction Codes . . . 13
2.9.2 Retransmission . . . 14
2.9.3 Fehlererkennung . . . 15

3 Systementwurf **17**
3.1 Sicht auf das Gesamtsystem . . . 17
3.2 Architektur des Interface Node . . . 18
3.2.1 Hardware Connector . . . 19
3.2.2 Network-Connector . . . 19
3.2.3 Timer-Service . . . 20

3.2.4 Kernkomponente 20
3.2.5 Configuration Handler 21
3.2.6 Einordnung in das OSI-Modell 21
3.3 Konfiguration 22
3.4 Event-System 23
3.5 Kommunikationssicherung 24
3.6 Publish/Subscribe System 25
3.7 Adressierung 26
3.8 Nachrichtentypen 27

4 Umsetzung 28
4.1 Simulationsumgebung 28
4.2 Prototyp 29
4.3 Sicht auf das Gesamtsystem 30
4.3.1 Master-Node 30
4.3.2 Switching-Node 30
4.3.3 Processing-Node 31
4.4 Ausführungskontexte 31
4.5 Nachrichtenstruktur und Marshalling 31
4.5.1 Datenfelder 32
4.5.2 Marshalling-Prozess 33
4.6 Speicherverwaltung 33
4.7 Fehlererkennung 33

5 Evaluierung 35
5.1 Speicherbedarf 35
5.2 Leistungsfähigkeit 35
5.2.1 Versuchsaufbau 36
5.2.2 Verarbeitungsdauer 36
5.2.3 Durchsatzrate 38
5.3 Timing 39
5.4 Determiniertheit 40

6 Fazit und Ausblick 41

Abkürzungsverzeichnis 42

Literaturverzeichnis 43

1 Einleitung

1.1 Motivation

Im Laufe der Geschichte haben sich die Anforderungen an Raumfahrtsysteme stark geändert. War der erste künstliche Satellit, Sputnik I, wenig mehr als ein Radiowellensender, erkannte man bald das wissenschaftliche Potential der Satelliten und benutzte sie für immer komplexere Experimente. Die Systeme mussten jetzt zum Einen verlässlich über einen längeren Missionszeitraum funktionieren, zum Anderen jedoch auch eine hohe Rechenleistung aufweisen.

Durch die fortschreitende Miniaturisierung und Verfügbarkeit der Halbleitertechnologien wurden solche Systeme möglich und zudem eröffnete sich die Möglichkeit einer kommerziellen Nutzung.

Die heutigen Möglichkeiten der Computertechnik eröffnen Wege, möglichst günstige und leistungsfähige Raumfahrtsysteme zu entwickeln, welche dennoch einen hohen Grad an Verlässlichkeit aufweisen können. Um diese Anforderungen erfüllen zu können, hat das Deutsche Zentrum für Luft-und Raumfahrt (DLR) das On-Board-Computer-Next-Generation (OBC-NG) Projekt gestartet zu welchem diese Masterarbeit beiträgt.

Das DLR ist das nationale Forschungszentrum der Bundesrepublik Deutschland mit den Schwerpunkten Luft- und Raumfahrt, Energie und Verkehr. Diese Arbeit wurde in der Abteilung Software für Raumfahrtsysteme und interaktive Visualisierung in der Forschungseinrichtung für Simulations- und Softwaretechnik des DLR angefertigt.

1.2 Das Projekt OBC-NG

Das OBC-NG Projekt ist aus der Studie Software und Hardware Architektur für rekonfigurierbare Computer (SHARC) [WLB13] hervorgegangen, welche die Grundlagen eines Hochleistungsrechnersystems für Raumfahrtanwendungen erforschte. Die Anforderungen an das System wurden bereits in dieser Studie festgelegt und bilden die Grundlage des OBC-NG Projektes und somit auch der vorliegenden Arbeit. Durch leistungsfähige Rechnernetze auf Raumfahrtsystemen wird es möglich, rechenintensive Aufgaben zu erfüllen. Beispiele für die Anwendung solcher Systeme sind die optische Navigation auf Himmelskörpern oder das Verarbeiten der wissenschaftlichen Daten direkt an Bord des Raumfahrtsystems.

Das Grundkonzept des Rechnersystems ist als Beispielaufbau in Abbildung 1.1 dargestellt. Es besteht aus Rechnerknoten (Nodes), welche in drei Typen nach ihren Aufgaben unterteilt sind. Die erste Art sind die Processing-Nodes (PN), diese übernehmen die Verarbeitung und Aufbereitung der Daten innerhalb des Raumfahrtsystems. Den zweiten Knotentyp bilden die Master-Nodes (MN), welche die Verwaltungsaufgaben des internen Rechnernetzes behandeln.

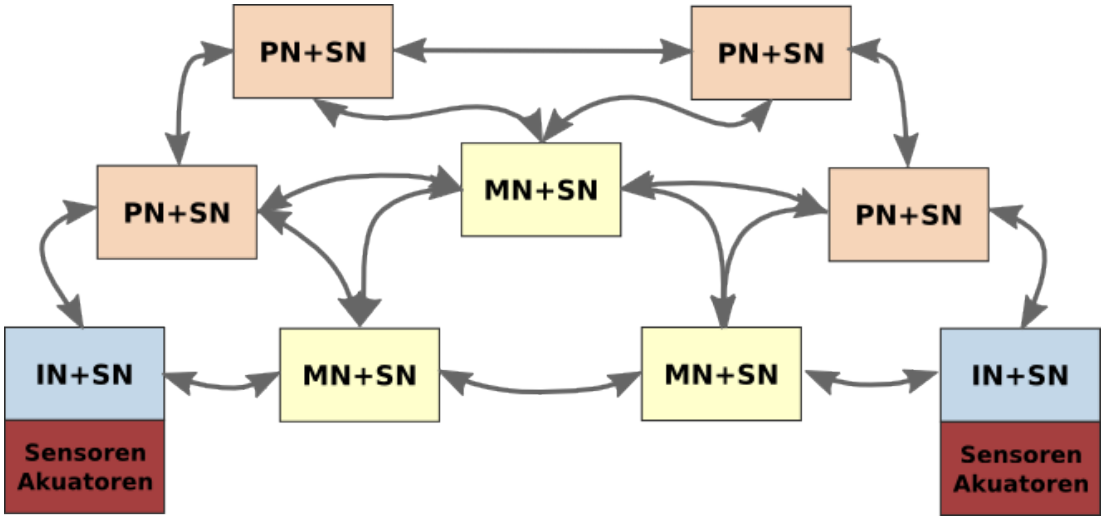

Abbildung 1.1: Beispielhafter Aufbau eines OBC-NG Netzwerks

Die Switching-Nodes (SN) sind für das Routing zuständig und leiten die Nachrichten im Netzwerk zwischen den Knoten weiter. Die letzte Gruppe der Knoten besteht aus den Interface-Nodes (IN); über diese Komponenten bekommt das Rechnersystem Zugriff auf die Sensoren und Aktuatoren des Raumfahrzeugs. Der Hauptteil dieser Arbeit wird die Interface-Nodes behandeln, da diese Knotenart hauptsächlich für die Verteilung von Informationen verantwortlich ist.

Im Kapitel Systemarchitektur wird genauer auf die Komponenten des OBC-NG Konzeptes eingegangen.

1.3 Aufgabenstellung

Die Aufgabe dieser Arbeit ist die Entwicklung eines Konzeptes zur Informationsverteilung innerhalb eines Raumfahrtsystem. Bei der Verteilung der Daten spielt im Zusammenhang mit OBC-NG die Interface-Nodes eine zentrale Rolle, weshalb hauptsächlich auf diese Knotengruppe eingegangen wird.

Bei der Entwicklung dieser Knoten muss auf die stark beschränkten Ressourcen von eingebetteten Systemen eingegangen werden. Zusätzlich zu den Bedingungen der eingebetteten Systeme müssen Anforderungen aus der Raumfahrt beachtet werden. Diese bestehen im Wesentlichen aus der Unterstützung von Redundanzen und dem deterministischen Verhalten des Systems. Nach der Auswahl geeigneter Technologien soll eine prototypische Implementierung des Systems in einem Netzwerksimulator und in Hardware vorgenommen werden, anhand welcher anschließend eine Evaluierung der Architektur stattfinden wird.

1.4 Aufbau der Arbeit

Der erste Teil der Arbeit gibt einen Überblick über geläufige Techniken im Bereich der verteilten Systeme, vor allem aus demjenigen der Unternehmensanwendungen und dem High Perfo-

mance Computing (HPC). Danach wird im zweiten Teil der Arbeit ein Auswahl der Technologien getroffen und damit die Systemarchitektur der Interface-Nodes beschrieben. Im dritten Kapitel beschreibt die Umsetzung der Konzepte in eine prototypischen Implementierung. Darauf folgt eine Evaluierung des Systems, welche auf die Anforderungen und Bedingungen der Aufgabenstellung eingeht. Den Abschluss bildet das Fazit mit einem Ausblick auf die Optimierungsmöglichkeiten dieses Konzeptes.

2 Stand der Technik

Die Technik, Aufgaben in einem Rechnernetz zu verteilen, ist vor allem im Bereich des High Performance Computing stark ausgeprägt. Dort sind Systeme zum verteilten Rechnen sowie für entfernte Dienste allgegenwärtig. Dieses Kapitel soll eine Übersicht über die derzeit vorhandenen Technologien geben.

2.1 Hardware in Raumfahrtsystemen

Für die Umgebungsbedingungen im Weltraum müssen die verbauten Hardwarekomponenten spezielle Anforderung erfüllen. Durch erhöhte elektromagnetische Strahlung, welche sich störend auf Halbleiter-Bausteine auswirkt, müssen spezielle, sogenannte strahlungsresistente Bauteile verbaut werden. Diese Bauteile werden meist mit abschirmenden Gehäusen versehen, sodass die Strahlung den Chip-Kern nicht erreichen kann. Zusätzlich zur Abschirmung sind die Systeme oft intern redundant ausgelegt, um im Fehlerfall auf ein Ersatzsystem wechseln zu können und die Mission nicht zu gefährden.

Speziell für Raumfahrtsysteme entwickelt sind die LEON Prozessoren, welche ursprünglich von der ESA entworfen wurden, und auf einem SPARC Befehlssatz basieren [Gai02]. Weiterhin existieren mehrere strahlungsresistente Versionen der Prozessoren mit MIPS und x86 Befehlssatz. Die Programmierung solcher Systeme unterscheidet sich prinzipiell nicht von den üblichen eingebetteten Systemen, da sich die Änderungen fast ausschließlich auf den physischen Aufbau des Chips auswirken, jedoch nicht auf den Befehlssatz.

2.2 Netzwerktechnik in Raumfahrtsystemen

Wie bei allen Hardwarekomponenten ist auch die Netzwerkstruktur der erhöhten elektromagnetischen Strahlung ausgesetzt. Dieser Umstand bedarf einer starken Abschirmung des Übertragungsmediums, und wirkt sich in einer hohen Fehlerrate bei der Datenübertragung aus. Gleichzeitig ist es wünschenswert, das Gewicht der Kabelbäume möglichst gering zu halten und den Stromverbrauch zu minimieren.

Bei der Auswahl der Infrastruktur muss ein Kompromiss zwischen mehreren Faktoren gemacht werden: Während eine vermaschte Netztopologie den höchsten Grad an Redundanz bietet, erhöht sich das Gewicht des Kabelbaums. Eine Sterntopologie hält zwar die Anzahl der benötigten Kabel niedrig, allerdings befindet sich mit dem zentralen Switch ein sogenannter „Single Point of Failure“, dessen Ausfall den Ausfall des kompletten Systems zur Folge hätte. Bei einer redundanten Auslegung des zentralen Switch erhöht sich dann wieder das Gewicht und der allgemeine Stromverbrauch.

Ein weit verbreitetes System, welches sich den Anforderungen in der Raumfahrt annimmt, ist der SpaceWire Feldbus. Die Datenrate des SpaceWire-Systems beträgt maximal 400 MBit/s, was es zum Beispiel für den Anwendungsfall der optischen Navigation, bei der große Datenmengen (Bilddaten) in kurzen Abständen übertragen werden müssen, an seine Grenzen bringt. Für die Zukunft ist ein Nachfolgesystem geplant, welches den Namen SpaceFibre trägt. Es soll auf Softwareebene kompatibel zum derzeitigen SpaceWire-Standard sein und dabei die Übertragungsrate auf bis zu 1 GBit/s steigern. Durch die Verwendung von Glasfaserkabeln zur Vernetzung der Teilnehmer, kann zusätzlich das Gewicht der Kabelbäume verringert und der Übertragungsweg unempfindlich gegen elektromagnetische Strahlung gemacht werden [PMS07].

2.3 Verteilte Ein- und Ausgabe

Ein zentrales Konzept des OBC-NG Projektes ist die Verteilung der Aufgaben auf mehrere Rechnenknoten. Während bei herkömmlichen Raumfahrtsystemen meist für jede Aufgabe dedizierte Rechenknoten verbaut werden, verfolgt das OBC-NG Projekt den Ansatz generischer Rechenknoten. Dabei werden die Aufgaben über alle Knoten verteilt. Das Thema der verteilten Ein- und Ausgabe befasst sich demnach mit der Verteilung der Informationen innerhalb des Netzwerkes. Dabei müssen die Rechnerknoten mit Informationen der Sensoren versorgt und entsprechend die Befehle dieser Knoten an die Aktuatoren delegiert werden.

Im Bereich der Unternehmensanwendung haben verteilte Anwendungen in den letzten Jahren stark an Bedeutung gewonnen. Dies lässt sich an den vielen Softwarearchitekturen und Frameworks erkennen, die sich in diesem Bereich gebildet haben (Java Enterprise Edition, Teile des .NET Frameworks, CORBA). Dennoch ist der Aufbau solcher Systeme unterschiedlich zu den Anwendungen in eingebetteten Netzen. Im Unternehmensanwendungen werden große, leistungsfähige Rechner eingesetzt, bei denen keine grundsätzliche Ressourcenknappheit herrscht, weshalb die Hauptaspekte dieser Architekturen in der Wartbarkeit und Erweiterbarkeit liegen. Das zeichnet sich beispielsweise in den verwendeten Protokollen wie XML ab. Bei eingebetteten Systemen muss der Fokus jedoch auch auf die Optimierung der Software liegen, da sowohl Rechenleistung, Arbeitsspeicher und oftmals auch Netzwerkleistung stark beschränkt sind.

Teilweise aus Ressourcenknappheit, aber auch weil Voraussetzungen (wie zum Beispiel ein Betriebssystem oder Systembibliotheken) oftmals nicht vorhanden sind, können die Frameworks für Unternehmensanwendungen nicht direkt verwendet werden. Jedoch ist es möglich Konzepte aus diesen Systemen zu übernehmen und diese für den Einsatz in eingebetteten Systems anzupassen. Die Thematik des verteilten Rechnens wird jedoch auch in eingebetteten Systemen behandelt, ein Beispiel dafür bilden Sensor-Netzwerke.

Bei diesen Netzwerken agieren viele kleine Sensorknoten, die gemeinsam durch Funk verbunden sind und Zustandsdaten verteilen. Der wissenschaftliche Artikel *Realtime Communication and Coordination in Embedded Sensor Networks* [SAL+03] beschreibt ein System für eben solche Netzwerke. In diesem Artikel werden Sensor-Netzwerke als „verteilte Rechnerplattformen mit starken Beschränkungen“ beschrieben, was ebenfalls auf das OBC-NG Systems zutrifft. Auf der Netzwerkschicht bestehen die Anforderungen der Sensor-Netzwerke vor

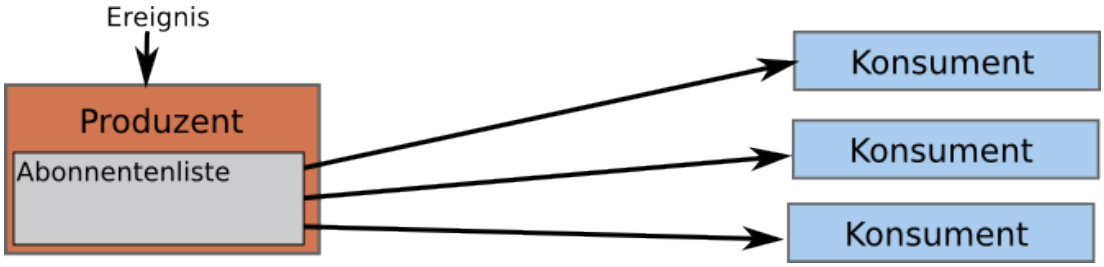

Abbildung 2.1: Prinzip des Push-Betriebs

allem darin, Informationen durch ein hochdynamisches Netz von Knoten zu verteilen, welche durch unzuverlässige Funkverbindungen vernetzt sind. Diese Probleme bestehen in dem Netzwerk des OBC-NG Systems nicht, da die Verbindungen statisch sind und Verbindungsabbrüche bei Kabelverbindungen deutlich unwahrscheinlicher sind.

Ein Aspekt, der sowohl für Sensornetzwerke als auch für Raumfahrtsysteme von Belang ist, ist die Energiesparsamkeit. Sowohl dem Funkknoten des Sensor-Netzwerkes, der über eine Batterie versorgt wird, als auch dem Interface-Node des Raumfahrtsystems stehen nur sehr begrenze Energiemengen zur Verfügung. Innerhalb der vorgestellten Arbeit wurde das Betriebssystem TinyOS aufgeführt, welches in Hinblick auf die stark eingeschränkten Ressourcen in Sensorknoten entwickelt wurde.

2.4 Bereitstellungsmodi

Die Bereitstellung von Informationen wird in verteilten Anwendungen wird als Dienst bezeichnet. Demnach werden die Teilnehmer nach Produzenten und Konsumenten dieser Dienste unterteilt. Allgemein gibt es zwei Betriebsmodi, um diese Dienste zu betreiben, welche sich durch den Auslöser der Datenübermittlung unterscheiden, den Push- sowie den Pull-Betrieb.

2.4.1 Push-Betrieb

Ein Betriebsmodus für verteile I/O Systeme ist der Push-Betrieb. Abbildung 2.1 zeigt den schematischen Aufbau einer Nachrichtenverteilung im Push-Betrieb. In diesem Modus bestimmt der Produzent der Informationen den Zeitpunkt der Übermittlung. Dazu ist es notwendig, dass er eine Liste aller Konsumenten besitzt, welche über die neuen Daten informiert werden wollen. Ein Konsument, der auf dieser Liste steht, wird als Abonnent bezeichnet. Wird eine Übermittlung ausgelöst, sendet der Dienstanbieter dann die neuen Daten an alle Abonnenten. Die Übermittlung kann dabei zyklisch ausgelöst werden (z.B. durch einen Timer-Event) oder durch äußere Einflüsse (z.b. durch neue Messdaten). Der Konsument besitzt dann zu jeder Zeit die aktuellsten Daten und wird zeitnah über Aktualisierungen informiert. Dieser Modus bietet sich für Anwendungen an, bei denen die Konsumenten neue Daten möglichst schnell erhalten sollen.

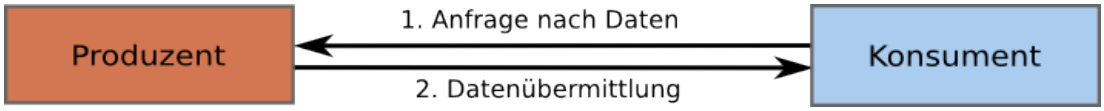

Abbildung 2.2: Prinzip des Pull-Betriebs

2.4.2 Pull-Betrieb

Die zweite Möglichkeit eine Datenübertragung auszulösen stellt der Pull-Betrieb dar, welche in Abbildung 2.2 dargestellt ist. Im Gegensatz zum Push-Betrieb löst in diesem Modus der Konsument die Übertragung der Daten aus, und bekommt vom Produzenten die aktuellen Daten übermittelt.

Es kann hierbei jedoch nicht sichergestellt werden, dass der Konsument jederzeit die aktuellsten Daten hat. In der Zeit, in der die Daten vom Produzent zum Konsumenten versendet und dort verarbeitet werden, können bereits aktuellere Daten bei dem Produzenten vorliegen. Der Konsument würde in diesem Fall im Gegensatz zum Push-Modus von der Aktualisierung nichts erfahren. Die Verwendung dieses Modus bietet sich bei Konsumenten an, die Daten nur relativ selten benötigen und tolerant gegenüber veralteten Daten sind.

Eine weitere Verwendungsmöglichkeit wäre, eine Pull-Anfrage für das Auslösen von Aktuatoren zu verwenden. Diese Verwendung würde eine Anpassung der Pull-Anfrage erfordern, da diese dann Nutzdaten mit sich tragen müsste. Das würde jedoch eine Zweckentfremdung des Pull-Mechanismus bedeuten, da die Pull-Anfrage lediglich als informelle Nachricht ohne Nutzdaten konzipiert ist.

2.5 Interprozess Kommunikation

Bei verteilten Systemen werden die Teilnehmer in der Regel in unterschiedlichen Prozessen ausgeführt. Sowohl lokale Prozesse, die beim Start jeweils ihreren eigenen Adressbereich zugewiesen bekommen, als auch Prozesse die über ein Netzwerk verteilt sind besitzen keinen gemeinsamen Speicherbereich. Im Gegensatz zu Threads, welche im selben Adressraum gestartet werden, können Prozesse daher nicht direkt miteinander kommunizieren. Es werden daher Mechanismen benötigt, die die Kommunikation unter den unabhängigen Prozessen ermöglicht.

Die Grundprinzipien des Datenaustausches von zwei unabhängigen Prozessen lassen sich unter dem Begriff „Interprocess Communication“ (IPC) zusammenfassen, dessen Grundlagen bereits 1986 von Leslie Lamport beschrieben wurden [Lam86]. Zu Beginn hat sich dieses Feld noch auf die Kommunikation lokaler Prozesse innerhalb eines Computers beschränkt, jedoch lassen sich diese Prinzipien ebenfalls auf Prozesse anwenden, welche in einem Rechnernetz verteilt sind.

Im Allgemeinen stellen Betriebssysteme, welche auch den Prozesswechsel verwalten, Mittel zur Kommunikation zwischen verschiedenen Prozesse zur Verfügung. Da man bei eingebetteten Systemen nicht immer davon Ausgehen kann, dass ein Betriebssystem zur Verfügung steht, muss die Kommunikation der einzelnen Prozesse selbst umgesetzt werden.

Im Zusammenhang mit der IPC soll der Fokus hierbei auf dem Informationsaustausch zwi-

schen Prozessen in einem Netzwerk liegen. Die Methoden, die in dieser Arbeit vorgestellt werden, sind der Shared-Memory, die Datenströme, der Remote-Procedure-Call und die Message Queue

2.5.1 Shared-Memory

Ein verbreiteter Ansatz der Kommunikation zwischen zwei getrennten Prozessen ist der Shared-Memory. Da beide Prozesse getrennte Bereiche im Speicher belegen und nicht direkt auf den Bereich des anderen Prozesses zugreifen können, wird ein Bereich im Speicher reserviert, welcher für alle beteiligten Prozesse zugänglich ist. Dieses System ist zunächst nur für lokale Prozesse möglich, da die beiden Programme direkten Zugriff auf dieses gemeinsamen Speicherbereich haben müssen. Die Technik des Shared-Memory auf lokalem Arbeitsspeicher wird von den meisten aktuellen Betriebssystemen unterstützt.

Um dieses Prinzip auf ein Netzwerk zu übertragen, existieren Konzepte, welchen die einen gemeinsamen virtuellen Speicherbereich über ein Rechnernetz ermöglichen [NL91]. Diese Technik wird „Distributed Shared Memory“ genannt und trägt im Bereich des HPC zur Skalierbarkeit großer Rechencluster bei. Bei der Verwendung von Shared-Memory treten die gleichen Probleme wie in der Programmierung mit Threads auf. Durch die Möglichkeit, jederzeit auf jede Stelle im gemeinsamen Speicher zugreifen zu können, müssen Mechanismen implementiert werden, welche die Konsistenz der Daten sicherstellt und Deadlocks (Verklemmungen) vermeidet.

2.5.2 Datenströme

Eine einfache und vielseitige Möglichkeit, Informationen zwischen Prozessen auszutauschen, ist diejenige der Datenströme. Der Sender schickt dabei Daten Byte für Byte über sogenannte Kanäle, die jeweils zwei Prozesse miteinander verbinden. Auf der anderen Seite des Kanals wartet der Empfänger auf die Daten in diesem Kanal. Die Zuordnung der Daten wird über diese Kanäle geregelt, was bedeutet, dass für jede Art Daten, welche übermittelt werden soll, ein eigener Kanal geöffnet werden muss.

Entweder werden reine Daten übermittelt, die der Empfänger direkt interpretieren und verwenden kann, oder es wird auf der Grundlage dieser Datenströme ein Protokoll aufgesetzt. Da Datenströme zunächst nur eine Folge von Bytes sind, ohne eine bestimmte Datenstruktur vorzugeben, sind sie sehr vielseitig, und können daher als Grundlage für höhere Protokolle genutzt werden. Bekannte Vertreter für Datenströme sind die Sockets, sowie die sogenannte Pipes für die lokale Prozesskommunikation.

2.5.3 Remote Procedure Call

Das Konzept des Remote Procedure Call (RPC) bildet einen Funktionsaufruf in einer Programmiersprache nach, welcher dann auf einem entfernten Prozess ausgeführt. Für die Schicht der Anwendungsprogrammierung sieht dieser Funktionsaufruf wie ein normaler lokaler Aufruf aus, und verhält sich entsprechend synchron, das heißt, die Funktion kehrt erst nach der

Ausführung im entfernten Prozess zurück. Unter dieser Schicht regelt das RPC-System das Verpacken der Methodenaufrufe und der Rückgabewerte in Nachrichten, welche über das darunterliegende Netzwerk versendet werden.

Für die Verwendung des Systems müssen alle Prozesse, welche Methoden entfernt aufrufen wollen, wissen, wo sich Prozesse befinden, die diese Aufrufe dann tatsächlich ausführen. Um diese Informationen zu veröffentlichen, wird in der Regel ein zentraler Verwalter eingesetzt, der auf Anfrage die verfügbaren Funktionen bereitstellt.

Für die Technik der RPC existiert ein Standard, der in der RFC 5531 definiert wurde [Thu09]. Bekannte RPC-Systeme sind CORBA, sowie im Java-Umfeld Java Remote Method Invocation (Java RMI).

2.5.4 Message-Passing

Eine weitere Art, Informationen zwischen getrennten Prozessen auszutauschen, ist das Message-Passing. Für das gesamte System wird im ersten Schritt eine einheitliche Struktur der Nachrichten (Messages) vorgegeben. Diese Datenstrukturen werden dann im Ablauf der Kommunikation in einen Bytestrom umgewandelt und dann über einen Datenstrom an den entfernten Prozess gesendet. Dort werden sie in die Datenstruktur der Nachrichten umgewandelt und entsprechend ihres Inhaltes abgearbeitet. Für die Umwandlung in einen Bytestrom und zurück in die Datenstruktur werden sogenannte Marshalling-Mechanismen benötigt, auf die später in diesem Kapitel eingegangen wird.

Diese Vorgehensweise benötigt nur einen Kanal zur Übertragung, da unterschiedliche Daten durch den Typ der Nachricht unterschieden werden können. Will man diese Kommunikation asynchron gestalten, muss man zusätzlich in die Software eine sogenannte Message-Queue einfügen. Diese Message-Queue übernimmt die Synchronisation der ankommenden und abgehenden Nachrichten.. Dafür wird eine Liste angelegt, welche alle Nachrichten enthält, die gesendet oder empfangen werden. Diese Liste wird dann sequentiell, oder wenn nötig priorisiert, abgearbeitet. Die Kommunikation mit Messages ist eine verbreitete Möglichkeit in verteilten Systemen Informationen auszutauschen. Die beiden Kommunikationskonzepte der Message Oriented Middleware und der Distributed Message Queue basieren beide auf dem Prinzip des Message-Passing

2.6 Message Oriented Middleware

Eine sehr stark vertretetene Architekturen im bereich der Unternehmensanwendung und HPC ist die sogenannte Message Oriented Middleware (MOM). Abbildung 2.3 zeigt das Prinzip der MOM. Dieses sieht einen zentralen Knoten, den sogenannten Broker vor, welcher die Zugriffe auf alle verbundenen Teilnehmer regelt.

Die verfügbaren Dienste werden bei diesem Konzept unabhängig von den Teilnehmern, welche die Dienste Anbieten verwaltet. Das heißt, dass der Broker einen Dienst über ein sogenanntes Topic anbietet, und die Teilnehmer sich jeweils für ein Topic als Konsument oder Produzent von Daten anmelden. Bei einer Anfrage nach Daten entscheidet der Broker dann dynamisch

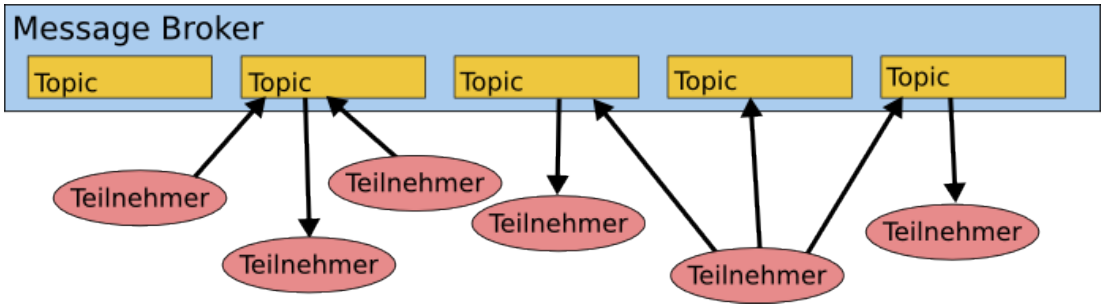

Abbildung 2.3: Prinzip einer Message Oriented Middleware

(z.B. nach der aktuellen Last der Teilnehmer), welcher der Teilnehmer, die sich als Produzenten für dieses Topic gemeldet haben, die Daten bereitstellt [Cur04].

Dieses Prinzip der losen Kopplung hat mehrere Vorteile. Die Teilnehmer dieses Systems müssen sich nicht gegenseitig kennen um Daten auszutauschen, sondern fragen diese Information jedesmal ab, und bekommen dynamisch die beteiligten Teilnehmer zugewiesen. Auch können Teilnehmer ausgetauscht werden, ohne dass dadurch der Betrieb des Systems beeinflusst wird. Weiterhin kann auch der Konfigurationsaufwand minimiert werden, da jeweils immer nur der Broker die aktuelle Systemkonfiguration kennen muss. Neben den Vorteilen existieren auch Nachteile, welche besonders bei einem Raumfahrtsystem zur Geltung kommen. Da der Broker die einzige Stelle ist, an der die Verwaltungsinformationen vorhanden sind, bildet der Broker einen Single Point of Failure. Dieses Problem kann durch die redundante Auslegung, z.B. einen zweiten Broker behoben werden. Weiterhin nachteilig an einem zentralen Konzept ist, dass dich an dem Broker der gesamte Datenverkehr konzentriert. Dadurch wird die Skalierbarkeit deutlich beeinflusst.

2.7 Distributed Message Queue

Eine Alternative zur Message Oriented Middleware stellt die Distributed Message Queue dar. Sie umgeht die Nachteile des zentralen Message Broker dadurch, dass jeder Teilnehmer des Netzwerkes seine eigene Liste von Nachrichten und Teilnehmern hält, welche sich für die angebotenen Daten abonniert haben. Abbildung 2.4 zeigt, wie die Knoten in einer Vernetzung dabei über die Subscriber-Listen miteinader verbunden sind.

Damit wird das Prinzip der losen Koppelung aufgegeben, da jetzt jeder Teilnehmer wissen muss, von wem er Daten bekommt, beziehungsweise wem er Daten senden soll. Jedoch wirkt sich dieser Nachteil bei einem statischen Netzwerk, dessen Teilnehmer sich nicht ändern kaum aus. Durch einen geeigneten Rekonfigurationsmechanismus kann die Ausfallsicherheit bei einem Ausfall eines Knotens erhöht werden und der Datenverkehr besser verteilt werden.

Eine Umsetzung dieses Konzept findet sich in 0MQ (Zero Message Queue). Dieses Framework bietet mehrere Möglichkeiten der Interprozess-Kommunikation, unter anderem auch das der Distributed Message Queue. Es baut dabei auf die bereits genannten Sockets auf. Sowohl für die lokale Kommunikation von Prozessen mit den Unix-Sockets, als auch für die Netzwerkkommunikation mit den Sockets des Transmission Control Protocol (TCP), welche eine

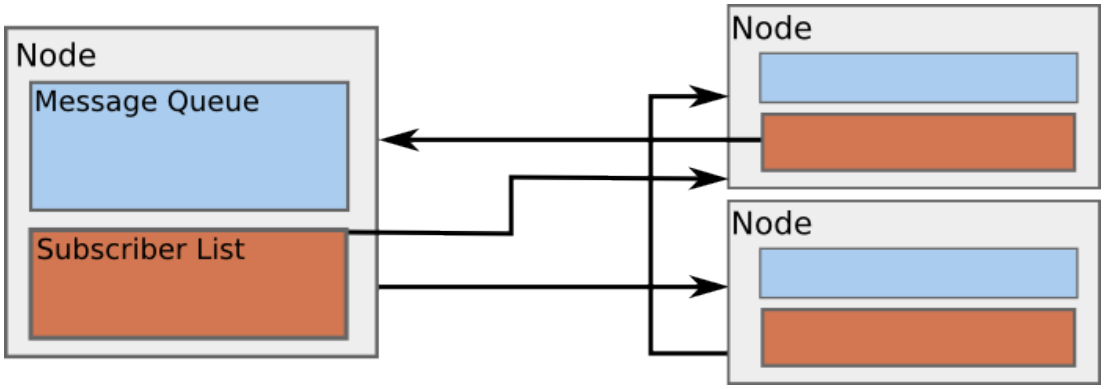

Abbildung 2.4: Beispiel der Vernetzung von Knoten einer Distributed Message Queue

verlässliche Verbindung zu den Teilnehmern bietet. Durch die Verwendung von TCP, welches wiederum auf dem Internet Protocol (IP) basiert, kann das Framework nicht ohne weiteres auf anderen Netzwerkprotokollen, wie beispielsweise SpaceWire verwendet werden. Es besteht zwar in diesem Fall die Möglichkeit IP und TCP in anderen Protokollen zu Kapseln, jedoch steigt dadurch der Overhead enorm.

2.8 Message Marshalling

Das Marshalling bezeichnet die Vorgehensweise und die Art, wie aus den Daten einer Datenstruktur Byte-Folgen für die Verwendung in den Datenströmen gebildet werden. Entsprechend wird der umgekehrte Fall, bei dem die Byte-Folgen aus dem Datenstrom wieder in die vorgegebenen Datenstrukturen gewandelt werden als Demarshalling bezeichnet.

Die einfachste und schnellste Methode des Marshalling einer Datenstrukur ist es, direkt den Speicherbereich als Byte-Folge in den Datenstrom zu schreiben. Da jedoch die Prozessoren in verteilten Systemen heterogene Architekturen besitzen können, und damit eine unterschiedliche Darstellung von Daten haben, würde das Speicherabbild auf einer anderen Prozessorarchitektur falsch interpretiert werden. Zu den Unterschieden zwischen den Architekturen zählen vor allem die unterschiedlichen Datenbreiten für Ganzzahl- und Gleitkommawerte, sowie die Reihenfolge der Bytes bei Daten welche aus mehreren Bytes zusammengesetzt werden, die sogenannte Endianess. Ebenfalls muss festgelegt werden wie Daten variabler Größe, wie beispielsweise Byte-Arrays oder Zeichenketten abgelegt werden.

Um diese Unterschiede zwischen den Systemen zu überbrücken muss eine für alle einheitliche Darstellung der Daten festgelegt werden, und die Datenstrukturen in dieses Format gewandelt werden. Für die Darstellung der Daten haben sich im Bereich der verteilten Systeme einige Konzepte etabliert. Im folgenden werden drei ausgewählte Darstellungen vorgestellt, die Texdarstellung, welche in Unternehmensanwendungen häufig verwendet werden. Für die binäre Darstellung von Daten wird die Beschreibungssprache ASN.1 vorgestellt, mit der viele Anwendungsprotokolle im Umfeld des Internets beschrieben sind, sowie die Google Protocol Buffers, die eine effiziente Form eines binären Protokolls bieten.

2.8.1 Textdarstellung

Eine einfache und verbreitete Möglichkeit der einheitlichen Darstellung von Daten ist die textbasierte. Dabei wird unter Zuhilfenahme einer einheitlichen Zeichenkodierung (beispielsweise dem 8-bit UCS Transformation Format (UTF-8) oder American Standard Code for Information Interchange (ASCI)) die Daten in eine Zeichenkette umgewandelt. Es kann dabei in eine einfache Abbildung des Datenstrom in den Bereich der darstellbaren Zeichen, wie beispielsweise bei der Base64-Kodierung oder in eine Auszeichnungssprache gewandelt werden, wie der eXtensible Markup Language (XML) oder der JavaScript Obejct Notation (JSON).

Textdarstellungen haben mehrere Vorteile: Wird eine einheitliche Zeichenkodierung festgelegt, kann die codierte Nachricht von jedem Teilnehmer im Netzwerk verstanden werden. Die Unterstützung von Zeichensätzen ist in vielen höheren Programmiersprachen enthalten. Weiterhin kann bei der Darstellung in eine Auszeichnungssprache das Format einfach erweitert werden, und die Daten bei der Übermittlung einfach eingesehen werden, was das Debugging erleichert. Auch ist es einfacher Daten von dynamischer Größe darzustellen, da bestimmte nicht darstellbare Zeichen als Trenner, sogenannte Delimiter, verwendet werden können(zum Beispiel das Byte mit dem Wert 0 um eine Zeichenkette abzuschließen).

Der große Nachteil ist der Overhead, der bei dieser Darstellung entsteht. Dadurch, dass nicht alle Kombinationen eines Bytes ausgeschöpft werden können, und bei Auszeichnungssprachen zusätzlich Metadaten über die Struktur des Dokuments eingefügt werden müssen, steigt die Menge an Daten die letztendlich übermittelt werden müssen signifikant an. Auch benötigt die Abarbeitung von textbasierten Nachrichten deutlich mehr Rechenleistung und Arbeitsspeicher als die Bearbeitung von binären Protokollen.

Textuelle Formate, insbesondere XML sind sehr verbreitet in Unternehmensanwendungen. Dort ist hohe Kapazität an Arbeitsspeicher, sowie eine leistungsfähige Netzwerkanbindung gegeben, womit mehr Wert auf Erweiterbarkeit und Wartbarkeit gelegt werden kann, als das auf Geschwindigkeit und Speicheroptimierung geachtet werden muss.

2.8.2 ASN.1

Den Ansatz einer binären Codierung der Daten verfolgt die Abstract Syntax Notation One (ASN.1) [Oli01]. Die ASN.1 wird von der International Telecommunications Union (ITU) in den Recommendations X.680 bis X.683 beschrieben. Sie bietet eine Beschreibungssprache mit der eine einheitliche Struktur für die Datenübertragungen definiert werden kann. Wurde das Nachrichtenformat festgelegt, werden im zweiten Schritt mit Hilfe der Encoding Rules die konkrete Codierung auf der Bitebene festgelegt. Aus der Kombination der Beschreibungssprache und den Encoding Rules erzeugt dann ein dazugehöriger Compiler den Quellcode für Datenstrukturen und Umwandlungsmethoden in der jeweiligen Programmiersprache. Die unterschiedlichen, vorgegebenen Encoding Rules sind in den Recommendations X.690 bis X.695 beschrieben.

Der Vorteil von abstrakten Beschreibungssprachen ist, dass die Definiton der Struktur unabhängig der darunterliegenenden Programmiersprache geschieht, und dann einfach für eine bestimmte Programmiersprache der entsprechende Code generiert werden kann. Dadurch ist

dann auf jedem System unter jeder Programmiersprache eine einheitliche Kommunikationsstruktur vorhanden.

ASN.1 ist weit verbreitet für die Definition von Nachrichtenstrukturen in Kommunikationssystemen. Für viele Internetprotokolle existieren Beschreibungen in ASN.1.

2.8.3 Protocol Buffers

Eine konkrete Umsetzung eines Marshalling Systems ist Protocol Buffers von Google. Ähnlich wie ASN.1 gibt es eine Beschreibungssprache für die Struktur der Datenübermittlung vor welche dann über einen Compiler in Datenstrukturen und Methoden der jeweiligen Sprache gewandelt werden. Protocol Buffers ist offiziell für die Sprachen C++, Java und Python verfügbar. Protocol Buffers benutzt eine binäre Darstellung der Daten und eine Little Endian Base 128 Codierung der Werte um eine effizintere Darstellung der Datenfelder zu erreichen. Es wurde in Hinblick auf Schnelligkeit in der Codierung und Effizienz in der Datengröße ausgelegt.

Protocol Buffers bietet durch die Verwendung einer eigenen Beschreibungssprache die gleichen Möglichkeiten wie ASN.1, mit dem Vorteil, dass es hierfür bereits optimierte Codegeneratoren gibt. Zusätzlich existiert bereits eine Umsetzung der Protocol Buffers für eingebettete Systeme. Diese sind durch die Verwendung von statischer Speicherreservierung, statischer Verlinkung und Ressourcensparsamkeit an die Bedingungen in eingebetteten Systemen angepasst, und daher sehr gut für diesen Anwendungsbereich geeignet [SP11].

2.9 Transportsicherung

Für manche Übertragungen ist es notwendig sicherzustellen, dass die Nachricht den Empfänger erreicht. Dafür werden Programmmechanismen benötigt, die einen Übertragungsfehler beheben, erkennen und melden können. Bei Übertragungsfehlern gilt es zu unterscheiden, ob die Übertragung verfälscht wurde, beispielsweise durch Übersprechen der Kommunikationsleitungen, oder ob die komplette Übertragung verloren gegangen ist, was der Fall bei einem Ausfall von Netzwerkkomponenten ist. Um verlässliche Verbindungen im Umfeld der Anwendungsprogrammierung umzusetzen wurden bereits mehrere Konzepte entworfen. Zwei geeignete Konzepte werden im folgenen erläutert:

2.9.1 Error Correction Codes

Die Error Correction Codes (ECC) sind eine Gruppe von Codierungen, welche die ursprüngliche Codierung der Binärdaten um zusätzliche Paritätsinformationen erweitert. Diese Paritätsinformationen werden aus den Nutzdaten gebildet und können nach der Übertragung zur Integritätsprüfung der Daten und darüber hinaus zum Korrigieren von fehlerhaft übertragenen Nachrichten verwendet werden.

Vorteilhaft an den ECC ist die Fähigkeit Übertragungsfehler zu erkennen, und bis zu einem bestimmten Grad zu reparieren, in dem aus den Paritätsinformationen wieder die korrekten Daten errechnet werden. Dadurch wird bei fehlerhaften Übertragungen kein erneutes Senden

notwendig. Sollte die Nachricht zu stark beschädigt sein, dass eine Wiederherstellung nicht möglich ist, kann der Fehler zumindest erkannt und gemeldet werden.

Nachteilig ist dagegen, dass Verbindungsabbrüche nicht erkannt werden können. Fällt eine Netzwerkkomponente aus, gibt es bei der alleinigen Verwendung von ECC keine Möglichkeit, zu erkennen, ob die Nachricht überhaupt zugestellt wurde. Für diesen Fall muss ein weiterer Sicherungsmechnismus eingebunden werden. Auch steigt durch die Verwendung von ECC die Größe der Daten, die übermittelt werden müssen und es bedarf Rechenzeit für die Umwandlung in den ECC und zurück. Ein Beispiel für einen effizienten Error Correction Code kann unter [Riz97] gefunden werden.

2.9.2 Retransmission

Im Gegensatz zum Error Correction Code wird bei der Retransmission Methode die Codierung der Daten nicht geändert. Das System beruht darauf, dass Nachrichten nach dem Versenden bei dem Sender zunächst gespeichert werden. Im Fehlerfall können sie dann erneut versendet werden, und werden erst gelöscht, wenn sichergestellt ist, dass Nachrichten fehlerfrei übermittelt wurde. Die wohl bekannteste Verwendung dieser Technik findet sich im Transmission Control Protocol (TCP) wieder. Durch geeignete Methoden zur Integritätsprüfung, wie beispielsweise eine Prüfsumme kann die korrekte Übertragung der Nachricht geprüft werden. Diese Technik der Integritätsprüfung wird später genauer erklärt. Welche Schritte danach erfolgen ist abhängig von der Art der Bestätigungen, welche für dieses System festgelegt werden. Die möglichen Bestätigungsarten sind die positive, negative, selektive und kumulative Bestätigung.

Positive Bestätigungen

Positive Bestätigungen werden versendet, wenn die Nachricht korrekt übertragen wurde. Nach einem Timeout geht der Sender davon aus, dass die Nachricht nicht zugestellt wurde und sendet die Nachricht erneut. Durch die Einführung eines Zählers, der die Anzahl der Zustellungsversuche erfasst, können ebenfalls Verbindungsabbrüche erkannt werden. Wird eine zuvor festgelegte Obergrenze an Zustellungsversuchen erreicht, wird davon ausgegangen, dass der Empfänger der Nachricht nicht erreichbar ist. Damit können beide Arten von Übertragungsfehlern erkannt werden. Jedoch ist es nicht sehr effizient, da neben der Bestätigung für jede gesendete Nachricht, selbst bei einem einzelnen fehlerhaften Bit die Nachricht erneut gesendet werden muss. Zusätzlich kann es passieren, dass Nachrichten doppelt versendet werden, falls die Bestätigung erst nach dem Timeout für das erneute Senden eintrifft. Eine Umsetzung dieser Bestätigung ist im TCP Protokoll implementiert [Pos81].

Negative Bestätigungen

Das Prinzip der negativen Bestätigungen sieht eine Benachrichtigung des Senders vor, falls die Integritätsprüfung fehlschlägt. Trifft bis zum Zeitpunkt des Timeouts keine solche Bestätigung ein, wird die Nachricht als erfolgreich zugestellt betrachtet [Roa05]. Diese Methode kann jedoch keine Verbindungsabbrüche erkennen, da bei einem Ausfall des Empfängers ebenfalls

keine Bestätigungen gesendet werden. Zusätzlich birgt diese Methode die Gefahr, dass Nachrichten verloren gehen, wenn die negative Bestätigung erst nach dem Timeout ankommt.

Selektive Bestätigungen

Eine Kombination aus den beiden vorherigen Methoden sind die Selektiven Bestätigungen. Jede Nachricht wird je nachdem, ob sie korrekt angekommen ist quittiert. Bei einem Timeout werden nicht-bestätigte Nachrichten erneut gesendet. Der Vorteil hierbei ist, dass im Gegensatz zu den reinen positiven Bestätigungen nicht bis zum Timeout gewartet werden muss, bis fehlerhaft übertragene Nachrichten erneut gesendet werden, und dadurch die erneute Zustellung schneller ausgelöst wird. Dabei bleiben die Erkennung von Verbindungsabbrüchen erhalten. Das TCP Protokoll implementiert diese Bestätigungsart in dem sogenannten “Selective Acknowledgement Option” [MMFR96].

Kumulative Bestätigungen

Kumulative Bestätigungen sind eine Form der positiven Bestätigungen. Bei dieser Art wird nicht jede einzelne Nachricht bestätigt, sondern die Bestätigung einer Nachricht wirkt sich ebenfalls auf die vorangegangenen Nachrichten aus, die noch auf eine Bestätigung warten. Diese Vorgehensweise senkt die benötigte Anzahl an Bestätigungen, die gesendet werden müssen, da einige Nachrichten gesammelt werden können, bevor die Bestätigung gesendet wird. Das TCP Protokoll benutzt diese Art der Bestätigung innerhalb sogenannter Übertragungsfenster bei der alle Nachrichten in diesem Fenster mit einmal bestätigt werden [Pos81].

2.9.3 Fehlererkennung

Um das Konzept der Retransmission umsetzen zu können muss es eine Möglichkeit geben fehlerhafte Übertragungen zu erkennen. Eine verbreitete Methode ist die Bildung einer Prüfsumme. Allgemein ist die Prüfsumme ein Wert, der reproduzierbar aus einem beliebigen Byte-Feld berechnet werden kann. Mathematisch handelt es sich dabei um eine surjektive Abbildung des Byte-Feldes auf den Prüfsummenwert.

Für die Erkennung von Übertragungsfehlern wird zunächst die Prüfsumme über den Daten, welche übertragen werden sollen gebildet. Diese Prüfsumme wird danach gemeinsam mit den Daten zum Empfänger gesendet. Dieser bildet über den Daten erneut mit dem gleichen Algorithmus die eine Prüfsumme. Um nun zu erkennen, ob die Übertragung fehlerhaft ist, wird die eben errechnete Prüfsumme mit der übertragen verglichen. Sind die beiden Werte identisch war die Übertragung fehlerfrei. Sind sie es nicht, wurde die Nachricht während der Übertragung beschädigt. Es können also Fehler nur erkannt werden, die Prüfsummen können erkannte Fehler nicht korrigieren.

Zur Berechnung existieren Algorithmen, die Übertragungsfehler mit hoher Wahrscheinlichkeit erkennen können. Ein Verfahren zur Berechnung der Prüfsumme ist der Cyclic Redundancy Check (CRC) [PB61]. Dieses Verfahren eignet um zufällige Bitkipper zu erkennen, die durch Rauschen auf dem Übertragungsmedium erzeugt werden können. Der CRC ist weit ver-

breitet und findet sich unter anderem zur Fehlerüberprüfung im Ethernet-Prokotoll oder in ZIP-Archiven wieder.

Wird zusätzlich zur Erkennung von zufälligen Änderungen in der Übertragung ein Schutz vor mutwilliger Veränderung durch einen Angreifer benötigt, reicht ein CRC nichtmehr aus. Der Algorithmus zur Berechnung der CRC-Prüfsumme macht es relativ leicht einzelne Bits in der Übertragung so zu modifizieren, dass der resultierende CRC-Wert identisch, der Inhalt der Daten jedoch ein anderer ist. Um diese Art der Modifikation zu erkennen wird ein sogenannter kryptographische Hashwert [BSNP+95] als Prüfsumme herangezogen. Die Berechnung der Hashwerte ist so ausgelet, dass schon ein Bitkipper in der Nachricht in einer starken Veränderung des Hashwerte auswirkt. Damit lässt der Hashwert keine Rückschlüsse auf die eigentliche Nachricht zu. Dieses Verfahren ist zwar aufwändiger zu berechnen als die CRC-Prüfsumme, jedoch ist es nun mit einem extrem hohen Aufwand verbunden eine Nachricht mit anderem Inhalt, aber der gleichen Prüfsumme zu erzeugen. Die Fähigkeit zufällige Fehler zu erkennen bleibt erhalten. Für die Anwendung in einem Netzwerk zwischen Raumfahrtkomponenten ist jedoch ein kryptografischer Hash nicht notwendig, da es keinen direkten Zugang zu dem internen Netzwerk gibt, und dadurch keine Manipulation an den Nachrichten möglich ist.

3 Systementwurf

In diesem Kapitel wird die Architektur des Interface-Node vorgestellt. Diese wurde aus einer Auswahl der Techniken entwickelt, die im letzten Kapitel vorgestellt wurden.

3.1 Sicht auf das Gesamtsystem

Der Interface-Node des OBC-NG Systems stellt für die anderen Knoten im Raumfahrzeug die Schnittstelle zu den Sensoren und Aktuatoren bereit. Die Hauptaufgaben der Interface-Nodes besteht dementsprechend darin, Information innerhalb des Netzes zu verteilen und Steuerbefehle zu verarbeiten. Diese Aufgaben benötigen in der Regel recht wenig Rechenleistung. Um den Bedarf an Rechenleistung nicht unnötig zu steigern sollte der Interface-Node auch ohne Betriebssystem lauffähig sein. Diese sogenannten „bare-metal-Systeme“ ermöglichen die Verwendung einer sehr minimalen Hardware, welche einen sehr geringen Energiebedarf hat.

Aus den Anforderungen der SHARC-Studie [WLB13] ist allgemein gefordert, dass Anwendungen „auf verschiedenen Plattformen lauffähig sind“. Im Zusammenhang mit dem Interface-Node bedeutet dies, dass die Software so ausgelegt sein muss, dass sie portabel für möglichst viele Prozessorarchitekturen ist. Die Portabilität der Software muss sich dabei jedoch auch auf das Netzwerk beziehen, auf dem der Interface-Node seine Daten versendet und empfängt.

Eine weitere wichtige Anforderung, die mit dem eben angesprochenen Punkt der Portabilität zusammenhängt, ist ein hoher Grad an Modularität. Wird die gesamte Programmarchitektur modular gehalten, senkt das den Entwicklungsaufwand für alle Knoten durch die Wiederverwendung bestehender Module.

Bei Raumfahrsystemen, sind sie einmal gestartet, ist der Zustand des Gesamtsystems von der Bodenstation aus, nur durch die Telemetriedaten ersichtlich. Daraus ergibt sich, dass die Konfiguration des Systems statisch ausgelegt sein muss, um zu erkennen in welchen Zuständen sich die einzelnen Knoten befinden. Änderungen der Konfiguration müssen dann von dem Master-Node aus vorgenommen und die Bodenstation darüber benachrichtigt werden. Es dürfen dabei keine dynamischen Änderungen (wie eine Änderung der Netzwerkrouten bei hoher Auslastung) selbstständig von den einzelnen Knoten vorgenommen werden.

Ebenfalls wichtig für Raumfahrsysteme ist die Unterstützung von redundanten Systemen, da Systemkomponenten nicht ausgetauscht werden können. Es müssen daher zusätzliche Knoten verbaut werden, die automatisch beim Ausfall des Hauptsystems aktiviert werden und die Aufgaben des defekten Knotens übernehmen.

Für den Austausch von Informationen in dem Netzwerk wird ein Protokoll auf Anwendungsebene benötigt, welches in dieser Arbeit entworfen wird. Dieses muss sowohl möglichst wenig Overhead erzeugen, als auch in der Bearbeitung möglichst schnell und ressourcensparend sein.

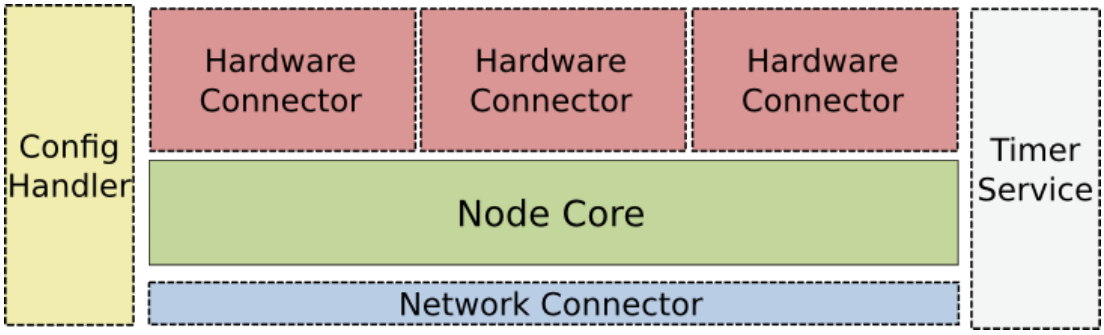

Abbildung 3.1: Architektur eines Interface Node

Alle diese Faktoren wirken sich auf den Entwurf des Interface-Nodes aus, welcher in den folgenden Abschnitten beschrieben wird.

3.2 Architektur des Interface Node

Die Architektur des Interface-Nodes ist in fünf Komponenten aufgeteilt, welche in Abbildung 3.1 dargestellt sind. Diese bestehen aus der konkret implementierten Kernkomponente (Node-Core) sowie den vier abstrakten Komponenten Hardware- und Network-Connector, Timer Service und Configuration Handler.

Um die Architektur und ihre einzelnen Komponenten zu erläutern eignet sich ein Anwendungsbeispiel, indem der Interface-Node als Schnittstelle zu einem Temperatursensor dient. Die Software läuft dann auf einem System, das sowohl einen direkten Zugriff auf den Sensor, sowie eine Verbindung zum Netzwerk mit den anderen Knoten hat. Als Netzwerk wird in diesem Beispiel SpaceWire verwendet.

Der Hardware-Connector ist in diesem Fall so implementiert, dass er die Messdaten des Sensors ausliest und unter einer programmierten Bedingung (beispielsweise nach einem Zeitintervall oder beim erreichen eines Temperaturwerts) die Sensordaten veröffentlicht. Diese Daten werden dann an den Node-Core weitergegeben. Dieser holt danach die Liste der Abonnenten, welche Daten von diesem Hardware-Connector erhalten sollen. Danach wird für jeden Abonnenten eine Nachricht erzeugt und in der Event-Queue abgelegt. Mit dem Einfügen der Nachrichten in die Event-Queue ist der Ausführungskontext des Hardware-Connector beendet.

Zusätzlich zum selbstständigen Versand implementiert der Hardware-Connector eine Methode, die bei einer Anfrage im Pull-Betrieb die Messdaten an anfragende Knoten versendet.

Die weitere Verarbeitung der Nachricht wird vom Ausführungskontext des Timer-Service durchgeführt. Als ersten Schritt wird dabei die Nachricht entnommen und in eine Byte-Folge gewandelt. Diese Byte-Folge wird dann an den Network-Connector weitergereicht, der diese in eine SpaceWire-Nachricht verpackt,

Auf der Empfängerseite kann sich nun zum Beispiel ein Interface-Node befinden, der diese Temperaturwerte benötigt. Der Network Connector dieses Knotens liest nun die Daten, die über das unterliegende Protokoll zunächst als Byte-Folge gesendet werden. Ist die Folge komplett, wandelt der Network-Connector diese Folge in eine Nachricht um und übergibt sie wie-

derum der Event-Queue des Node-Core. Danach kehrt der Ausführungskontext des Network-Connectors wieder zurück, und wartet auf neue Daten auf dem unterliegenden Protokoll. Die weitere Verarbeitung der Nachricht übernimmt wieder der Timer-Service. Dabei wird die Nachricht zunächst aus der Event-Queue entnommen und danach anhand Ihrerer Zieladresse einem Hardware-Connector zugeordnet und diesem übergeben. Danach entscheidet wieder die Implementierung des Hardware-Connector wie die Daten nach dem Empfang verarbeitet werden.

Die in diesem Beispiel genannten Komponenten der Architektur werden in den folgenden Abschnitten detaillierter erklärt.

3.2.1 Hardware Connector

Der Hardware Connector stellt die Verbindung zwischen der Hardware (d.h. der Sensoren und Aktuatoren) und der Kernkomponente bereit. Damit ist er der die Komponente, welche die Dienste des Interface-Node bereitstellt und ausführt.

Interface-Nodes können mit mehreren Sensoren und Aktuatoren verbunden sein, und damit mehrere Dienste gleichzeitig anbieten. Ein Interface-Node kann somit mehrere Hardware-Connectors besitzen. Der Zugriff auf jede Hardwarekomponente muss in einem eigenen Hardware Connector implementiert werden. Für baugleiche Hardware kann durch die Modulaität der Software der gleiche Hardware-Connector verwendet werden.

Für gewöhnlich besitzt ein Hardware-Connector einen eigenen Ausführungskontext. Dieser kann durch einen Hardware-Interrupt, einen Timer, oder durch einen Software-Interrupt ausgelöst werden. Hardware-Interrupts können durch die angeschlossenen Sensoren und Aktuatoren ausgelöst werden. Im Beispiel des Temperatursensors könnte der Interrupt durch das Erreichen eines bestimmten Wertes ausgelöst werden. Der Timer kann genutzt werden um wie im Beispiel angegeben innerhalb eines Zeitintervalls periodisch Daten zu veröffentlichen. Die letzte Möglichkeit den Ausführungskontext zu starten ist durch einen Software-Interrupt, der durch das Eintreffen einer Nachricht ausgelöst wird. Damit kann auf Plattformen mit Multithreading-Fähigkeit die Abarbeitung der Daten parallel zur den restlichen Aufgaben ausgeführt werden.

3.2.2 Network-Connector

Die zweite Connector-Komponente in der Architektur ist der Network-Connector. Seine Aufgabe ist die Verbindung des Interface-Nodes zu dem Netzwerk des Raumfahrsystems. Wie der Hardware-Connector ist dieser ebenfalls eine abstrakte Komponente. Damit wird die Portabilität auf verschiedene Netzwerke ermöglicht.

Um die Portabilität nicht zu beeinträchtigen unterstützt der Network-Connector zunächst lediglich die Möglichkeit Nachrichten im Unicast-Modus zu versenden. Das bedeutet, dass jede gesendete Nachricht nur einem einzigen Empfänger zugeordnet werden kann. Eine effizientere Möglichkeit mehrere Empfänger gleichzeitig mit Daten zu versorgen ist der Multicast. Dabei wird die Nachricht bei dem Sender einmal gesendet und dann durch die Netzwerkinfrastruktur verteilt. Um Multicast umzusetzen ist eine Unterstützung seitens des unterliegenden Protokolls, als auch der Netzwerkkomponenten erforderlich. Da nicht alle Protokolle Multicast-

Unterstützung besitzen, und die Unterstützung eine starke Auswirkung auf die gesamte Protokollverabeitung hat, wird diese Zustellungsmethode in dieser Arbeit nicht weiter verfolgt.

Ebenso wie der Hardware-Connector besitzt auch der Network-Connector einen eigenen Ausführungskontext. In ihm werden die Daten vom Netzwerk gelesen und an die Kernkomponente weitergeleitet. Auslöser für diesen Kontext ist in der Regel ein Interrupt, welcher durch die Hardware für den Netzwerkzugriff geregelt wird.

3.2.3 Timer-Service

Der Timer-Service besitzt zwei Aufgaben: die Abarbeitung der Event-Queue und die Bereitstellung eines Zeitstempels.

Im Zusammenhang mit der Transportsicherung und der Rekonfiguration muss es möglich sein Nachrichten zu einem späteren Zeitpunkt bearbeiten zu können. Der Timer-Service setzt diese Möglichkeit um. Eine detaillierte Beschreibung dieser Mechanismen findet sich in den Abschnitten 3.3 und 3.5.

Die zweite Aufgabe, die Bereitstellung eines Zeitstempels, wird für ebenfalls für die verzögerte Ausführung benötigt, um den absoluten Zeitpunkt anzugeben, an dem diese Nachricht bearbeitet werden soll. Darüber hinaus dient der Zeitstempel als Teil der Identifizierung einer einzelnen Nachricht. Die Identifizierung von Nachrichten wird im Abschnitt

Der Timer-Service ist, wie der Hardware- und Network-Connector als abstrakte Komponente in der Architektur vorgesehen. In einer eingebetteten Umgebung ist der Timer an den Timer-Baustein in der Hardware gekoppelt und muss daher speziell für diese Hardware-Komponente implementiert sein. Dieser Timer-Baustein löst dann mit einem Interrupt den Ausführungskontext des Timer-Service aus, der bereits im Beispiel beschrieben wurde, und startet damit die Protokollbearbeitung in der Kernkomponente.

3.2.4 Kernkomponente

Die Kernkomponente ist der einzige plattformunabhängige Bestandteil der Software, der auf jeder Plattform identisch ist. In ihr findet die Bearbeitung des Protokolls, sowie die Verwaltung des Interface Nodes statt.

Die Event-Queue ist der zentrale Punkt dieser Komponente. Hier werden alle eingehenden und ausgehenden Nachrichten abgelegt. Damit ist ein Wechsel der Nachrichten zwischen den verschiedenen Ausführungskontexten möglich.

Neben der Event-Queue hält die Kernkomponente ebenfalls die Abonnenten-Liste für die Hardware-Connectors, welche für das Publish-Subscribe System benötigt werden.

Da die Umsetzung des Protokolls auf jeder Plattform identisch ist, enthält die Kernkomponente die Datenstrukturen der Nachrichten. Dazu gehören, wie im Anfangsbeispiel beschrieben, die Methoden zur Umwandlung der Nachrichten in Byte-Ströme und zurück. Dieser Vorgang wird als Marshalling bzw. Demarshalling bezeichnet.

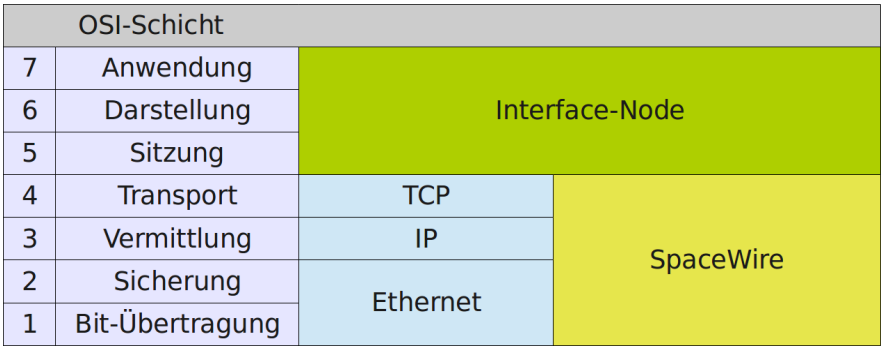

Abbildung 3.2: Einordnung der Protokolle im OSI-Modell

3.2.5 Configuration Handler

Der Configuration-Handler hat die Aufgabe die Komponenten des Interface-Nodes beim Eintreffen einer Rekonfigurationsbenachrichtigung in die angegebene neue Konfiguration zu bringen. Die Vorgehensweise zur Konfiguration wird in Abschnitt 3.3 genauer beschrieben.

Die Konfigurationsdaten können entweder direkt als Programmcode eingebunden werden oder der Handler wird als Parser einer Konfigurationdatei (beispielsweise im XML Format) implementiert. Das hat den Vorteil das dieser Parser dann auf anderen Nodes wiederverwendet werden kann.

3.2.6 Einordnung in das OSI-Modell

Im Open Systems Interconnect Modell (OSI-Modell)deckt der Interface-Node die obersten drei Schichten (Schicht 7 bis 5) ab [Int94]. Die Abbildung 3.2 zeigt die Einordnung des Interface-Node im OSI-Modell. Als Beispiel zu Protokollen, auf denen der Interface-Node aufsetzen kann, ist das TCP/IP und das SpaceWire Protocol aufgeführt.

Auf der obersten Schicht, der Anwendungsschicht, werden die verschiedenenen Dienste des Netzwerkes angeboten. Diese Aufgabe erfüllt hauptsächlich der Hardware-Connector indem er Dienste für den Hardwarezugriff anbieten.

Die darunterliegende Schicht, die Darstellungsschicht, legt die Byte-Darstellung der Nachrichten fest. Diese Schicht wird von der Kernkomponente behandelt, das diese die Methoden zum Marshalling und Demarshalling bereitstellt.

Die letzte der drei Schichten, welche vom Interface-Node behandelt werden ist die Sizungsschicht. Diese Schicht sorgt sich um die Kommunikation und Synchronisation der Prozesse im Netzwerk. Zuständig für diese Aufgabe ist ebenfalls die Kernkomponente, die mit den Abonnentenlisten die logischen Verbindungen im Netzwerk verwaltet und mit der Event-Queue für die Synchronisation von ankommenden und abgehenden Nachrichten sorgt.

Der Network-Connector stellt abschließend die Brücke zu der darunterliegenden Schicht vier sicher. Dabei ist seine Aufgabe die Nachricht aus den oberen Schichten weiterzuleiten, bzw. von den unteren Schichten entgegen zunehmen und an die oberen Schichten zu übergeben.

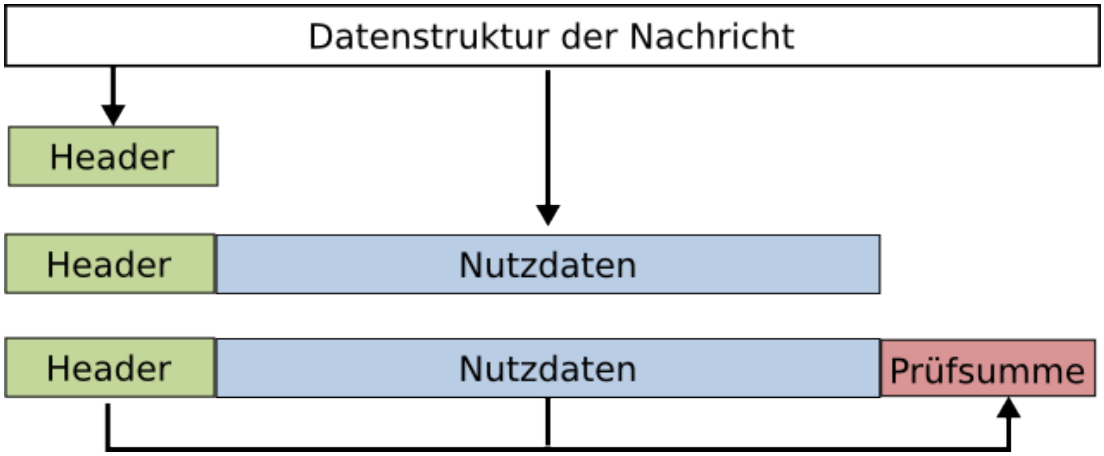

Abbildung 3.3: Die einzelnen Schritte des Marshalling-Prozesses

3.3 Konfiguration

Innerhalb einer Mission durchläuft das Raumfahrzeug unterschiedliche Phasen. Um möglichst wenig Energie zu verbrauchen ist es wichtig, dass nur die jeweils für die Missionsphase benötigen Geräte aktiviert sind, und alle anderen Verbraucher ausgeschalten werden. Auch werden in unterschiedlichen Phasen unterschiedliche Sensordaten bzw. Zugriff auf unterschiedliche Aktuatoren benötigt. Dafür muss es unter anderem möglich sein, die Abonnenten-Listen der einzelnen Hardware-Connectoren angepasst werden. Als letzter Aspekt der Konfiguration kommt die Fehlerbehandlung hinzu. Wird ein Fehler erkannt, muss er zunächst dem Master-Node gemeldet werden. Dieser entscheidet dann, welche Konfiguration den Fehler beheben kann und konfiguriert nachfolgend die anderen Knoten im System.

Wichtig in diesem Zusammenhang, dass der Zustand aller Knoten ausschließlich vom dem Master-Node verwaltet und geändert werden kann und die einzelnen Knoten nicht eigenmächtig ihren Zustand ändern.

Diese Arbeit befasst sich lediglich mit den Benachrichtigungen zu einem Konfigurationswechsel und den internen Abläufen beim Erhalt einer solchen Benachrichtigung. Die Verwaltung der Konfigurationen werden hier nicht behandelt, da dies zum Funktionsumfang des Master-Node gehört.

Zu dem konfigurierbaren Parametermetern des Interface-Node gehören die Hardware-Connectors, die aktiviert und deaktivert werden können. Ebenfalls müssen die Abonnenten-Listen angepasst werden um, wie bereits erwähnt, unterschiedliche Sensordaten oder Aktuatorzugriffe .

Die Hardware-Connectors werden über ihre internenZustände konfiguriert. Die drei Zustände in der sich ein Hardware-Connector befinden können sind Disabled, Paused und Running. Abbildung 3.4 stellt die möglichen Übergänge der Zustände dar.

Beim Erhalt einer Nachricht zum Konfigurationswechsel werden zunächst die Hardware-Connectoren in einen Paused-State gesetzt. In diesem Zustand können sie keine neuen Nachrichten veröffentlichen. Neue Daten können von dem Hardware Connector selbst zurückgehalten werden, das Verhalten ist aber nicht vorgegeben und von der jeweiligen Implementierung abhängig. Die neue Konfiguration selbst wird erst nach dem Ablauf einer Zeitspanne, der

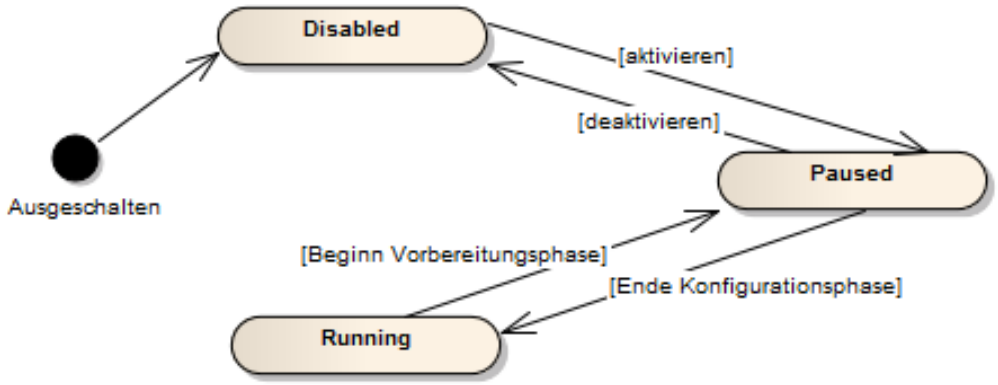

Abbildung 3.4: Zustandsübergänge der Hardware Connectoren bei dem Konfigurationsvorgang

sogenannten Vorbereitungszeit übernommen. Innerhalb der Vorberetungsphase können noch sämtliche Nachrichten abgesendet werden die sich bereits in der Event Liste befinden. Nach der Vorbereitungszeit werden alle Events in der Event Liste gelöscht. Bei dem Konfigurationsvorgang selbst werden die Hardware Connectoren, die deaktiviert werden in den Disabled-State und die aktivierten, in den Paused-State gesetzt. Erst nachdem die die Konfiguration des Interface-Nodes abgeschlossen ist, werden alle Hardware-Connectors, die sich im Paused-State befinden gleichzeitig wieder in den Running-State gesetzt. Um einen definierten Startzustand zu definieren, sind nach dem Hochfahren des Systems zunächst alle Hardware Connectoren im Disabled-State und starten erst, wenn der Master Node einen Wechsel in eine konkrete Konfiguration signalisiert.

Zusätzlich zu den verschiedenen Zuständen des Hardware-Connector, die vom Master-Node ausgegeben werden, existiert noch eine Konfiguration, in die sich der Interface-Node selbst versetzen kann. Dieser Zustand wird benötigt, um eine Selbstabschaltung umzusetzen. Der Interface-Node setzt die diesen Zustand, wenn die Verbindung zum Master Node abgebrochen ist. Dabei setzt der Knoten zunächst alle Hardware-Connectoren in den Disabled-State um danach auch sich selbst physisch abzuschalten.

3.4 Event-System

Der Kern des Message Passing Systems ist die Message-Queue. Sie übernimmt die Synchronisation der ein- und ausgehenden Nachrichten. Für die Anwendung im Interface Node muss diese um eine Zeit-Funktion erweitert werden um die verzögerte Ausführung von Nachrichten umzuseten. Die Erweiterung sieht vor, dass Nachrichten in eine Datenstruktur gekapselt wird, welche als Event bezeichnet wird. Die Datenfelder eines Events sind in der Abbildung 3.5 aufgelistet.

Neben der eigentlichen Übertragung enthält die Struktur eine Richtungsangabe, welche bezeichnet, ob es sich um ein ankommendes oder abgehendes Event handelt. Anhand dieser Rich-

Feldname	Größe in Byte
Eventrichtung	1
Ausführungszeit	8
Ausführungszähler	1
Übertragung	*

Abbildung 3.5: Datenfelder eines Events

tungsangabe wird die Nachricht bei der Ausführung entsprechend bei eingehenden Events an die Hardware-Connectors, oder bei ausgehenden Events an die Network-Connectors weitergeleitet.

Eine weitere Angabe ist die Ausführungszeit. Dieses Feld ist ein absoluter Zeitstempel, welcher angibt wann dieses Event frühestens ausgeführt werden kann. Soll ein Event sofort ausgeführt werden, wird die Ausführungszeit auf 0 gesetzt.

Zusätzlich enthält die Struktur ein Zähler, der angibt wie oft ein Event bereits behandelt wurde. Diese wird benötigt um in der Transportsicherung einen Verbindungsabbruch zu erkennen, wenn eine Nachricht mehrmals nicht zugestellt werden konnte.

Mit diesen Erweiterungen ist es möglich mit einer einzigen, nach Ausführungszeitpunkt sortierten Liste, Funktionen wie ein Resend nach einem Timeout, oder eine verzögerte Abarbeitung von Nachrichten umzusetzen.

3.5 Kommunikationssicherung

Bei der Auswahl von Mechanismen zur Umsetzung verlässlicher Übertragungen muss eine Abwägung zwischen der Übertragungsgröße, Latenz und der Verlässlichkeit gemacht werden. Im Abschnitt 2.9 wurden bereits Techniken vorgestellt, mit der eine verlässliche Zustellung von Daten umgesetzt werden kann. Für die Anwendung im Interface Node fällt die Wahl auf die Retransmission mit positiver Bestätigung, da diese im Gegensatz zum Error Correction Code den Vorteil bietet, dass sowohl fehlerhafte Übertragungen, als auch komplett verloren gegangene Übertragungen erkannt werden können. Die Verwendung von Error Correction Codes kann dagegen nur fehlerhafte Übertragungen erkennen und korrigieren. Zudem müssen bei einer korrekten Übertragung weniger Daten über das Netzwerk gesendet werden. Die Umsetzung der Sicherung läuft in zwei Phasen mit zwei Parametern und nutzt die Fähigkeit der verzögerten Ausführung des Eventsystems, wie in Abschnitt 3.4 beschrieben, aus. Der Ablauf ist im Sequenzdiagramm in Abbildung 3.6 dargestellt.

Die erste Phase ist die Retransmission-Phase. Sobald eine Übertragung gesendet wurde, wird sie erneut in die Event-Liste eingefügt. Vor dem einfügen wird die Ausführungszeit um einen Timeout erhöht, welcher der erste Parameter des Systems darstellt. Zusätzlich wird ein Zähler erhöht, der die Anzahl der Zustellungsversuche wiedergibt. Ist die Übertragung korrekt beim Empfänger angekommen, sendet dieser, wie bereits beschrieben, eine Bestätigung über den Erhalt der Nachricht zurück. Beim Eintreffen der Bestätigung löscht nun der Sender die bestätigte Nachricht aus der Liste der ausstehenden Events. Wurde eine Nachricht fehlerhaft, oder

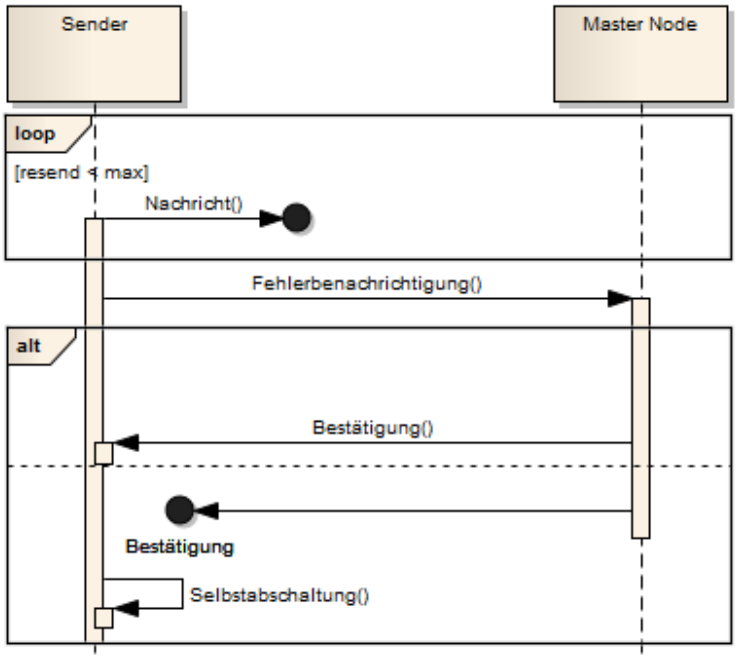

Abbildung 3.6: Sequenzdiagramm der Fehlerbehandlung

gar nicht zugestellt, bleibt die Bestätigung aus. Läuft nun der Timeout ab, beginnt diese Schleife mit dem Senden der Nachricht erneut. Dieses Verfahren wird solang wiederholt, bis der Zähler für die Zustellungsversuche einen Schwellwert erreicht hat, der als zweiten Parameter angegeben wird.

Beim Erreichen des Schwellwertes beginnt die zweite Phase der Fehlerbehandlung. In dieser Phase kann davon ausgegangen werden, dass der Empfänger der Nachricht nicht mehr erreicht werden kann. In diesem Fall wird eine Benachrichtigung über den Ausfall des Empfängerknotens an den Master Knoten gesendet. Diese Nachricht benötigt ebenfalls eine Bestätigung. Dadurch kann der Interface-Node erkennen kann, ob entweder der Empfänger, oder er selbst die Verbindung zum Netz verloren hat. Bekommt der Knoten nun keine Bestätigung vom Master Node über den Erhalt der Fehlerbenachrichtigung, schaltet sich der Knoten vollständig ab, um nicht unnötig Strom zu verbrauchen.

3.6 Publish/Subscribe System

Zur Umsetzung des Push-Betriebes der Knoten wird ein klassisches Publish/Subscribe System genutzt.

Die dafür benötigte Listen der Abonnenten wird Zentral von der Kernkomponente verwaltet. Dabei besitzt jeder Hardware Connector seine eigene Liste von Abonnenten, auf die er zugreift, wenn er Daten veröffentlichen will.

Diese Liste ist im Gegensatz zu den üblichen dynamischen Listen, bei denen sich die Teil-

nehmer frei an- und abmelden können, innerhalb einer Konfiguration statisch. Das bedeutet, dass die Liste der Abonnenten für einen Hardware Connector nur durch eine Rekonfiguration geändert werden kann. Auf diese Weise kann der Zustand der Abonnentenlisten jederzeit nachvollzogen werden.

3.7 Adressierung

Für die Addressierung der einzelnen Hardware-Nodes wurde eine zusammengesetzte Adresse aus einer globalen und lokalen ID entworfen. Die vollständige Adresse besteht aus einem 2-Byte Wert, wobei das Least-Significant-Byte die globale Adresse, und das Most-Significant-Byte die lokale Adresse bezeichnet. Beide Bestandteile beschreiben einen Hardware Connector eineindeutig.

Der globale Bestandteil beschreibt den physikalischen Interface-Node. Der Zweck der globalen ID liegt darin, Übertragungen auf den Routern des darunterliegenden Protokolls zu ermöglichen. Daher sind diese zunächst abstrakt, und die Bedeutung ergibt sich aus der Implementierung des Network-Connector. Beispielsweise kann die globale ID in eine Host-Adresse eines IP Netzwerk umgesetzt werden. Auch könnte sie mit einer SpaceWire-Adresse gleichgesetzt werden, welche ebenfalls ein Byte breit ist.

Der lokale Bestandteil der zusammengesetzten Adresse ergibt sich aus der Architektur des Interface-Node. Da mehrere Hardware Connectoren an einem Interface Node registriert werden können, muss zusätzlich zu der eben genannten globalen ID noch die lokale ID hinzugefügt werden. Diese beschreibt dann den jeweilgen Hardware Connector des Interface Node. Die lokale ID ergibt sich durch die Reihenfolge der Registrierung des Hardware Connector. Der Hardware Connector, der zuerst registriert wurde, besitzt die lokale ID 0, der zweite entsprechend 1, usw.

Durch die Verwendung von Adressen, die einfach in entsprechende Adressen des darunterliegende Protokolls gewandelt werden können, müssen die Switching-Nodes nicht das Protokoll der Interface-Nodes verstehen, sondern behandeln diese Daten als Rohdaten der unteren Protokollschicht. Dadurch können reguläre Switches und Router für das darunterliegende Protokoll genutzt werden. Beispielsweise kann bei einer Verwendung von SpaceWire die SpaceWire Node-Address direkt als globale ID genutzt werden.

Jeder Knoten in dem Netzwerk, welcher aktiv Daten produziert oder konsumiert, muss durch eine solche Adresse innerhalb des Netzes erreichbar sein. Das bedeutet, dass neben dem Interface Node auch Processing-Nodes eine Adresse benötigen. Ebenfalls müssen Master Nodes mit einer Adresse versehen werden, um von ihnen Konfigurationswechsel zu erhalten und Fehlerbenachrichtigung zu senden.

Die globale Adresse muss jedoch nicht zwangsläufig mit der physikalischen Adresse verbunden sein. So kann ebenfalls beispielsweise bei dem Path-Adressing wie es in der SpaceWire Spezifikation [Eur03, 85] beschrieben ist, im Network-Connector die Zieladresse durch den Pfad zum Zielknoten ersetzt werden. Die Pfadangaben müssen dabei dem Sender bekannt sein und können durch die Rekonfiguration geändert werden.

3.8 Nachrichtentypen

Nachdem die verschiedenen Konzepte zur Umsetzung der Kommunikation beschrieben sind, ergeben sich zusammenfassend sechs unterschiedliche Nachrichtentypen. Die Tabelle in Abbildung 3.7 gibt eine Übersicht über die Nachrichtentypen.

Zur Übertragung von Sensordaten, oder Befehlen für Aktuatoren werden zwei Typen von Nachrichten benötigt: die **zuverlässige Datenübertragung** und die **unzuverlässige Datenübertragung**. Die unzuverlässige Datenübertragung ist neben der Bestätigung die einzige Nachricht, die keine Bestätigung des Senders benötigt, bei ihr wird bewusst in Kauf genommen, dass Nachrichten verloren gehen können. Dabei können Ressourcen gespart werden, da diese Nachrichten nicht zurückgehalten werden müssen.

Für die Umsetzung des Pull-Betriebs wird weiterhin ein Nachrichtentyp für die aktive Anforderung von Daten vorgesehen werden, der **Pull-Request**. Um die Transportsicherung wie beschrieben umzusetzen wird zusätzlich ein Nachrichtentyp für die **Bestätigungen** benötigt. Dieser Nachrichtentyp hat die Besonderheit, dass er keine Nutzdaten besitzt und der Timestamp dieser Nachricht derjenigen entspricht, die bestätigt werden soll. So kann aus dem Sender und dem Timestamp der Bestätigungsnachricht ermittelt werden, welche Nachricht bestätigt wird.

Für die Meldung eines Verbindungsfehlers innerhalb der Transportsicherung wird ebenfalls eine **Fehlermitteilung** benötigt. Diese enthält in den Nutzdaten die Node ID des Knoten, welcher nicht erreichbar ist.

Der letzte Nachrichtentyp, die **Rekonfigurationsmitteilung** wird ausschließlich vom Master-Node immer an alle Knoten gesendet und beinhaltet die neue Konfigurations ID in den Nutzdaten.

Nachrichtentyp	**Nutzdaten**	**verlässliche Zustellung**
zuverlässige Datenübertragung	Ja	Ja
unzuverlässige Datenübertragung	Ja	Nein
Pull-Request	Nein	Ja
Bestätigung	Nein	Nein
Fehlermitteilung	Ja	Ja
Rekonfigurationsmitteilung	Ja	Ja

Abbildung 3.7: Tabelle der Nachrichtentypen

4 Umsetzung

Die Implementierung des Interface-Nodes ist eine prototypische Umsetzung der im letzten Kapitel vorgestellten Architektur sowie der Netzwerk- und Rekonfigurationsprotokolls. Der Fokus liegt bei der Evaluierung der Architektur und der Protokolle, weniger auf der Optimierung der Komponenten. Die Implementierung ist daher vornehmlich für die verwendete Simulationsumgebung geschrieben. Die Kernkomponente des Interface-Nodes ist dabei so implementiert, dass sie den Anforderung der Plattformunabhängigkeit entspricht. Dabei kann der der gleiche Programmcode sowohl in der Simulationsumgebung als auch auf realer Hardware verwendet werden. Zur Umsetzung wurde soweit wie möglich auf die Verwendung von plattformabhängigen Bibliotheken verzichtet. Als Programmiersprache für die Implementierung wurde C++ gewählt. Die Gründe dafür sind zum Einen die Verfügbarkeit der Sprache für praktisch alle bedeutenden Hardwaresysteme, zum Anderen benutzt das verwendete Simulations-Framework C++ als Sprache für die Implementierung der Simulationselemente. Zudem wird im OBC-NG Projekt voraussichtlich C/C++ als Programmiersprache für die Entwicklung der System- und Anwendungssoftware genutzt werden.

Die folgenden Abschnitte beschreiben die Umsetzung des Interface Node Systems.

4.1 Simulationsumgebung

Für die Simulation der Netzwerkkommunikation wird das OMNET++ Framework verwendet [OMN13]. Dieses Simulations-Framework bildet die Kommunikation verschiedener Netzwerkkomponenten auf Byte-Ebene anhand einer diskreten Simulation nach. Das Verhalten der einzelnen Teilnehmer, sowie die Verbindungen untereinander sind frei programmierbar.

In der Abbildung 4.1 ist ein beispielhafter Aufbau eines Netzwerkes mit mehreren Interface-Nodes und einem Switching- und Master-Node gezeigt.

Mit Hilfe dieser Umgebung ist es möglich die Kernkomponente, eingebunden in ein simuliertes OBC-NG Netzwerk, zu entwickeln. Daraus folgt, dass bei der Implementierung des Interface Node auch die Bedingungen an die anderen Knotentypen, welche bereits vorgestellt wurden, betrachtet werden können. Das eröffnet die Möglichkeit für OBC-NG , das komplette Kommunikationsmodell und dessen einzelne Teilnehmer zu simulieren. Weiterhin kann neben dem Verhalten des Kommunikationssystems im Normalfall Fehlerfälle und Ausfälle simuliert werden, um die Zuverlässigkeit des verwendeten Netzwerkprotokolls zu überprüfen.

Im Zusammenspiel mit der in dieser Arbeit entworfenen Architektur des Interface-Node, welche hohen Wert auf Modularität legt, kann der Simulator als Entwicklungsumgebung für den Interface-Node genutz werden. Der plattformunabhängige Code der Kernkomponente kann

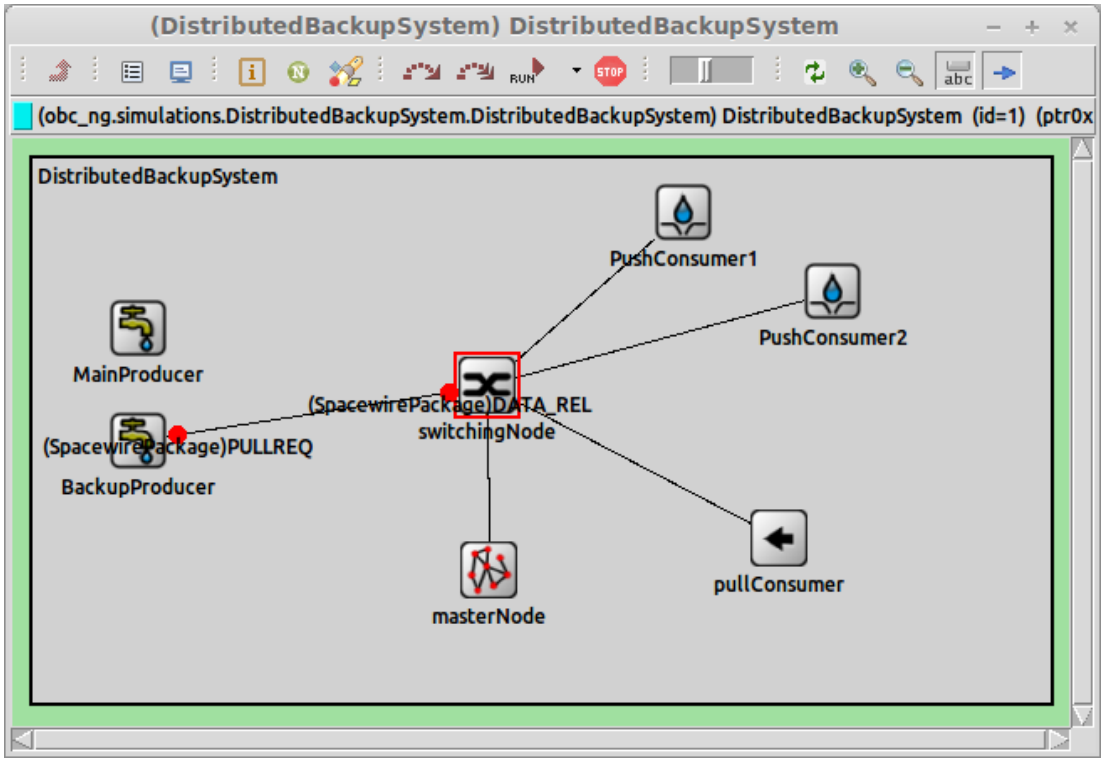

Abbildung 4.1: Beispielhafter Aufbau einer Simulation im OMNET++ Framework

dann in dieser Umgebung simuliert, getestet und debugt werden und danach direkt ohne Änderung am Code in der physischen Hardware verwendet werden.

Das event-basierte Modell der Simulation deckt sich sehr gut mit dem angesprochenen Event-System, welches im Interface Node verwendet wird. So kann die Timer-Komponente direkt Events in dem Simulationsmodell registrieren, die dann verzögert ausgeführt werden.

Durch die Darstellung der Übertragung auf Byteebene können ebenfalls darunterliegende Protokolle auf der Paketebene nachgebildet werden.

Da nach der Studie über die Bewertung der Netzwerktechnologien [SL12] SpaceWire eine der geeignetsten Technologien ist, die in OBC-NG eingesetzt werden können, sind die Pakete im Aufbau den SpaceWire Paketen nachempfunden. Als Referenz für den Paket aufbau gilt hierbei der SpaceWire Standard [Eur03].

4.2 Prototyp

Um die Lauffähigkeit der Sofware auf eingebetteten Systemen zu demonstrieren wurde die Software auf verschiedene Plattformen portiert. Die 3 Plattformen, welche zur Verfügung standen waren x86-64, ARM und PowerPC. Für die Übertragung kleinerer Datenpakete wurde eine Kommunikation zwischen den PowerPC und ARM Rechnern hergestellt. Im zweiten Anwendungsbeispiel mit großen Datenmengen, in diesem Fall 1 MiB große Bilddaten, wurden das ARM System zum Senden der Daten herangezogen, und der x86-64 Desktop Computer diente zum Empfang und Darstellung der Bilddaten.

Da zur Übertragung der Daten keine SpaceWire Hardware zur Verfügung stand, wurde

TCP/IP als Protokoll zur Übertragung genutzt. Durch die Verwendung dieses Protokolls kann die Portabilität auf andere Netzwerkprotokolle und Prozessorarchitekturen gezeigt werden, jedoch wären Leistungsmessungen der Übertragung nicht sehr aussagekräftig.

Weiterhin wurde das Linux Betriebssystem verwendet,um unter anderem die Threading und die Netzwerk-Bibliotheken zu Nutzen. Der Fokus dieser Arbeit liegt auf der Entwicklung des Protokolls, daher wurde auf eine reine Embedded-Implementierung ohne Betriebssystem verzichtet.

4.3 Sicht auf das Gesamtsystem

In der Simulation wurde neben dem Interface-Node auch die anderen Knoten des OBC-NG Systems eingebunden. Für den Master-, Switching- und Processing-Node wurde eine grobe Implementierung vorgenommen, die das Verhalten der Knoten nachstellt. Aus der Implementierung gehen die Vorgaben an das Verhalten der anderen OBC-NG Knoten hervor.

4.3.1 Master-Node

Der Master Node hat in diesem System die Hauptaufgabe, die Konfigurationen zu verwalten und die Rekonfiguration auszulösen. Er muss daher die Error-Notification-Nachrichten empfangen und verarbeiten können. Ebenfalls trifft der Master-Node die Auswahl der folgenden Konfiguration aus. Das bedeutet bei einer eintreffenden Fehlermeldung muss der Master-Node selbstständig erkennen können, mit welcher Konfiguration er diesen Fehler behandeln muss. Auf der Seite der Interface Nodes muss danach in der Implementierung des Configuration-Handler diese Konfigurationsinformationen decodiert und auf den Interface-Node umgesetzt werden.

Zu den weiteren Aufgaben, die über das interne Netzwerk hinausgehen, muss der Master-Node fähig sein einen Konfigurationswechsel der Bodenstation zu melden.

4.3.2 Switching-Node

Der Switching-Node muss bei dieser Umsetzung keine besonderen Fähigkeiten besitzen, da er lediglich das unterliegende Protokoll beherrschen muss und die Daten des höheren Protokolls als Rohdaten verwendet. Das ermöglicht die Verwendung von handelsüblichen Komponenten die lediglich das unterliegende Protokoll (z.B. SpaceWire oder Ethernet) behandeln. Die Verwendung von handelsüblicher Hardware, macht die redundante Auslegung der Komponenten einfach und kostengünstig.

Wird das SpaceWire Protokoll benutzt bestehen, wie bereits im Abschnitt 3.7 beschrieben zwei Möglichkeiten die Knoten zu adressieren: über die globale Adresse, oder über den Pfad durch die Router. Wird die Methode der globalen Adresse verwendet, müssen die Switching-Nodes ebenfalls konfigurierbar sein, um die Routen in unterschiedlichen Missionsphasen ändern zu können. Wird hingegen das Pfad-Routing eingesetzt kann die Konfiguration der Switching-Nodes wegfallen, da die Routen dann in der Network-Connector Komponente der Interface-Nodes angepasst wird.

4.3.3 Processing-Node

Da Processing-Nodes sowohl Daten anfordern, konsumieren und produzieren können sie im Kontext dieser Implementierung wie Interface-Nodes behandelt werden. Eine Möglichkeit wäre dabei, dass als Schnittstelle zum Datenaustausch eine Implementierung eines Interface-Node dem Processing-Node voransteht, welche dann die Daten an den eigentlichen Recheneinheiten weiterreicht. Daher kann aus Sicht des verteilten I/O und auch der Simulationsumgebung der Processing-Node zunächst wie ein Interface-Node betrachtet werden.

4.4 Ausführungskontexte

Wie bereits in dem Architekturbeispiel in Abschnitt 3.2 beschrieben findet die Verarbeitung von Nachrichten in verschiedenenen Ausführungskontexten statt. Die Umsetzung der Ausführungskontexten findet in dieser Arbeit auf drei Arten statt: mit Threads, über Interrupt und über Simulations-Ereignisse.

Bei Systemen mit Thread-Unterstützung ist der Ausführungskontext identisch mit dem Thread-Kontext. Das Programm hat in dieser Umsetzung keinen Einfluss auf den Kontextwechsel, da dieses vollständig vom Scheduler des Betriebssystems übernommen wird. Damit muss eine Thread-Synchronisierung in der Event-Queue stattfinden, da mehere Threads gleichzeitig auf die Elemente zugreifen können und inkonsistenten erzeugen können. Die Verwendung von Threads ermöglicht bei Hardwareplattformen mit mehr als einem Rechenkern die parallele Verarbeitung von Nachrichten.

Existiert für eine Hardwareplattform keine Thread-Unterstützung wird der Kontextwechsel durch Interrupts ausgelöst. Das Programm kann in diesem Fall Einfluss auf die Kontextwechsel ausüben, indem es die Interrupts aktiviert und deaktiviert. Damit kann eine Synchronisierung erreicht werden, indem die Interrupt-Routine vor dem Zugriff auf die Event-Queue alle Interrupts deaktiviert und nach dem Zugriff wieder aktiviert. Damit verhindert die Routine, dass sie ein anderer Interrupt unterbrechen kann.

Die letzte Umsetzung der Ausführungskontexte wird ausschließlich in der Simulationsumgebung genutzt. Da es sich um eine diskrete Simulation handelt, können die Simulationsereignisse wie Interrupts behandelt werden. Die Umsetzung der Ausführungskontexte in der Simulation sind also sehr ähnlich, da hier der Kontextwechsel durch ein Simulationsereignis ausgelöst wird. Ein Unterschied zu den Interrupts ist hierbei nur, dass Simulationsereignisse in ihre Abarbeitung nicht unterbrochen werden können. Selbst bei gleichzeitigem Auftreten mehrerer Ereignisse werden diese nacheinander ausgeführt. Eine Synchronisierung ist somit in der Event-Queue nicht nötig.

4.5 Nachrichtenstruktur und Marshalling

Ein essentieller Bestandteil eines jedes Netzwerkprotokolls ist der Aufbau der Nachrichten. Dieser einheitliche Aufbau stellt sicher, dass die versendeten Nachrichten für alle Teilnehmer des Netzes verständlich sind. Das Marshalling spielt bei der Darstellung und Umwandlung der

Feldname	Größe in Byte
Sender	2
Empfänger	2
Zeitstempel	8
Nachrichtentyp	1
Nutzdatengröße	4
Nutzdaten	Nutzdatengröße

Abbildung 4.2: Datenfelder der Nachrichtenstruktur

Nachrichten auf einzelne Bytes eine zentrale Rolle.

Unter dem Begriff Marshalling versteht man den Prozess, der die die Datenstruktur der Nachrichten in eine Byte-Folge wandelt. Dieses resultierende Byte-Folge wird dann als Rohdaten in ein Paket des darunterliegenden Protokolls verpackt und versendet. Für den Anwendungsfall dieser Arbeit ist es wichtig, dass die Nachrichten möglichst effizient gestaltet werden, was dabei sowohl die Minimierung der zu übermittelnden Daten, als auch die benötigten Ressourcen des Rechners betrifft.

Aus den verschiedenen Ansätzen, welche im Stand der Technik aufgeführt wurden, kommt in dieser Arbeit ein modifizierte Variante der embedded Protocol Buffers zum Einsatz. Diese Variante ist auf die Anforderungen in eingebettete Systemen angepasst [SP11]. Dadurch ist unter anderem die Speicherallokation vollständig statisch, was die Verwendung auf vielen eingebetteten System erst möglich macht.

Durch die Verwendung der Little Endian Base 128 Codierung (LEB128) von Integer-Werten bei Protocol Buffers wird die Übertragungsgröße minimiert. Bei dieser Art der Codierung werden Ganzzahlwerte nicht mit einer festen Datenbreite (beispielsweise 16 oder 32 bit) kodiert, sondern die Breite des Datenfelds vom Wert der Variablen abhängig gemacht. Es wird dabei immer die geringste Datenbreite gewählt in die der Wert passt.

Um die Größe der Übermittlungen noch kleiner zu halten, können zusätzlich die Nutzdaten durch geeignete Algorithmen komprimiert werden. Es wird jedoch keine generelle Kompression aller Übermittlungen vorgesehen. Die Gründe sind einerseits, dass eine Kompression überflüssig sein kann, wenn die Daten nur einige Bytes groß sind (wie bei Temperaturmessungen). Andererseits benötigt eine Kompression der Daten wiederum Ressourcen, welche in eingebetteten Systems nur sehr begrenzt zur Verfügung stehen. Eine Entscheidung, ob eine Kompression sinnvoll ist, hängt also von den Nutzdaten ab. Da die Nutzdaten nur Byte-Felder sind kann der Hardware-Connector also selber entscheiden, ob er die Daten vor dem versenden komprimiert.

4.5.1 Datenfelder

Abbildung 4.2 zeigt die Felder der Nachrichtenstruktur welche für jede Übertragung gleich ist. Die ersten beiden Felder geben die Adressen der beiden Partner der Übertragung an. Auf die Adressen folgt ein Zeitstempel, welcher den Zeitpunkt der Erzeugung der Nachricht angibt. Diese Kombination Empfänger und Zeitstempel ist für jede Nachricht eindeutig und wird

zur Identifizierung von einzelnen Nachrichten herangezogen. Dieses Merkmal wird für die Bestätigung von einzelnen Nachrichten benötigt. Das nächste Datenfeld beschreibt den Typ der Nachricht. Die letzten beiden Felder beschreiben die Nutzdaten der Übertragung. Da die Nutzdaten ein Array von Bytes sind, muss die Größe vorangestellt werden. Für den Fall, dass die Übertragung keine Nutzdaten enthält zeigt das Feld der Nutzdatengröße den Wert 0 an.

4.5.2 Marshalling-Prozess

Der Prozess des Marshalling wird in dem hier vorgestellten Protokoll in drei Stufen vollzogen. In Abbildung 3.3 sind die einzelnen Stufen dargestellt. In der ersten Stufe wird aus der Datenstruktur, welche die Nachricht enthält, die Metadaten mit Protocol Buffers in einen Byte-Array gewandelt. Hierbei fließen alle Daten aus der Tabelle 4.2, bis auf die Nutzdaten ein. Nach der Umwandlung wird das Byte-Array in den Ausgangspuffer kopiert

Der zweite Schritt des Marshalling kopiert die Nutzdaten aus der Datenstruktur in den Puffer hinter die Header-Daten. Da die Implementierung der embedded Protocol Buffers nur Byte-Arrays bis zu einer Größe von 128 Byte unterstützt, müssen die Nutzdaten getrennt behandelt werden.

Der letzte Schritt erzeugt aus den Header und Nutzdaten eine Prüfsumme und hängt sie an das Nutzdatenfeld an. Diese Prüfsumme wird nach der Übertragung zur Integritätsprüfung der Daten genutzt.

4.6 Speicherverwaltung

Ein Punkt der besonders für eingebettete Systeme relevant ist, ist die Speicherverwaltung. Für eingebettete Systeme bietet eine statische Speicherverwaltung mehrere Vorteile. Dadurch, dass zu Beginn des Programms alle benötigten Speicherbereiche alloziert werden eine Fragmentierung des ohnehin schon knappen Arbeitsspeicher verhindert. Zusätzlich kann der Bedarf an Arbeitsspeicher besser abgeschätzt werden, und damit die Systemvoraussetzungen bestimmt werden.

Innerhalb dieser Implementierung werden die Events in einer statischen Speicherverwaltung umgesetzt. Diese Verwaltung hält ein Array von statisch reservierten Objekten vor. Zusätzlich wird ein Flag zu jedem Objekt gespeichert, welches angibt, ob es Referenzen auf dieses Objekt gibt. Durch die Überladung des new und delete Operator der Event-Klasse können dann aus diesem statischen Speicherbereich die Referenzen auf freie Objekte aus dem Array zugegriffen werden, und entsprechen wieder freigegeben werden.

4.7 Fehlererkennung

Die Fehlererkennung ist ein wichtiger Bestandteil der Übertragungssicherung. Der komplette Sicherungsmechanismus verlässt sich dabei auf die Prüfsumme, wie in Abschnitt beschrieben. Als Verfahren zur Prüfsummenberechnung wird der CRC-32 Algorithmus verwendet Die Leistungsfähigkeit der CRC Prüfsumme, das heißt welche Fehler erkannt werden können, hängt

von dem verwendeten Polynom und dessen Grad ab [PB61]. Die zyklische Redundanzprüfung ist sehr gut geeignet um Übertragungsfehler durch Rauschen auf der Leitung (verursacht beispielsweise durch kosmische Strahlung) zu erkennen.

Die Berechnung und der Prüfsumme bietet einen Da die Prüfsumme lediglich an das Ende der Übertragung angehängt wird, kann die Berechnung und auf der Empfängerseite die Verifikation in Hardwarebausteinen vorgenommen werden. Für diese weit verbreitete Fehlerprüfung bestehen bereits viele Implementierung in Hardware zur Verfügung. Dies hätte zum einen die schnellere Berechnung zum Vorteil, andererseits wird dadurch die Nachrichtenverarbeitung des Interface Nodes bei fehlerhaften Nachrichten nicht in Anspruch genommen.

5 Evaluierung

Nachdem in den letzten Kapiteln die verwendeten Techniken und deren Umsetzung erläutert wurden, findet in diesem Kapitel eine Bewertung des Systems statt.

Im Fokus dieser Evaluierung steht die umgesetzte Architektur und das Kommunikationsprotokoll Diese soll zeigen, dass die Anforderungen der eingebetteten Systeme und der Raumfahrtsysteme wie in durch den Interface-Node erfüllt werden.

Die Evaluierung stützt sich dabei zum einen Teil auf Daten, welche aus der Simulation des Kommunikationsprotokolls gewonnen wurden und zum anderen Teil aus der prototypischen Implementierung.

5.1 Speicherbedarf

Da der Speicher, wie für eingebettete Systeme üblich, stark begrenzt ist, muss für jeden physikalischen Knoten zunächst der Bedarf an Arbeitsspeicher ermittelt werden. Durch die konsequente Verwendung von statisch reservierten Speicherbereichen lässt sich der Bedarf an Speicher recht gut ermitteln.

Der größte Teil des Arbeitsspeichers wird bei einem laufenden Interface Node für die Events benötigt. Zur Berechnung muss zunächst die Größe eines einzelnen Events ermittelt werden. Diese Größe ist zum wesentlichen Teil von dem Array abhängig, welches die Nutzdaten hält. Dieses Array für die Nutzdaten muss dabei mindestens die Größe der maximalen Nutzlast besitzen, die entweder versendet, oder empfangen wird (je nachdem welche Nutzlast größer ist). Sobald die benötigte Größe für ein Event ermittelt ist, lässt sich der gesamte Speicherbedarf in Byte mit folgender Formel ermitteln:

$$Speicherbedarf = Events(max) * (27 + Nutzdatenfeld)$$

Die Konstante 27 ergibt sich dabei aus den Verwaltungsdaten der Übertragung (17 Byte) und des Events (10 Byte). Siehe dazu Abbildung 3.5 und 4.2

Zusätzlich zu dem Speicherbedarf der Kernkomponente wird noch Arbeitsspeicher für die Arbeit der Hardware Connectoren und dem Network Connector benötigt. Dieser Bedarf ist jedoch von der Implementierung und der Anzahl der angebundenen Hardware Connectoren abhängig.

5.2 Leistungsfähigkeit

Ein wichtiger Faktor, welcher eine Aussage über Leistungsfähigkeit des Systems geben kann, ist die Durchsatzrate der Software. Diese gibt an, wie viel Nachrichten das System verarbeiten

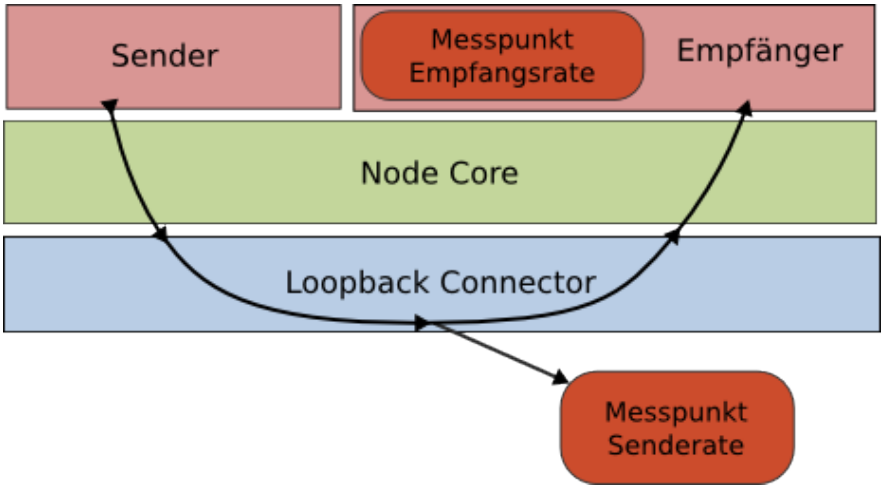

Abbildung 5.1: Schematischer Aufbau der Testumgebung

kann. Die Betrachtung kann dabei bei der maximalen Anzahl an Nachrichten liegen, die pro Zeiteinheit versendet werden können, oder davon abgeleitet, die maximale Datenmenge, die in dieser Zeiteinheit gesendet und/oder empfangen werden kann.

5.2.1 Versuchsaufbau

Um die Leistungsfähigkeit der Nachrichtenverarbeitung zu messen, wird für die Software eine Testumgebung aufgebaut, welche in Abbildung 5.1 dargestellt ist. Die Umgebung enthält einen Network-Connector, der als Loopback-Interface implementiert ist, das die serialisierten Nachrichten direkt zurückschreibt. Damit kann die Leistungsfähigkeit der Abarbeitung von Nachrichten ohne Einfluss des Netzwerkes gemessen werden. Zusätzlich notwendig ist die Implementierung zweier Hardware-Connectoren, die für das Senden und Empfangen der Nachrichten zuständig sind. Der Sender erzeugt dabei Datenpakete, verschiedener Größen um die Leistungsfähigkeit in Abhängigkeit der Nutzdatengröße zu bestimmen

Bei der Messung der Nachrichtenverarbeitung wird die Leistung in drei verschiedenen Modi gemessen. Die beiden Simplex Modi für Senden (TX) und Empfangen (RX) stellen dabei die Last auf reinen Sensorknoten (im TX-Modus) sowie reine Aktuatorknoten (im RX-Modus) nach. Der dritte Betriebsmodus ist der Vollduplex-Modus (FDX-Modus) in dem im gleichen Umfang Daten gesendet und empfangen werden, was ein Szenario für einen Processing-Node sein kann.

Als Hardwareplattform für die Evaluierung wird ein Raspberry Pi Modell B benutzt. Diese Plattform basiert auf einem ARMv6 Prozessor mit einer Taktfrequenz von 600Mhz und 512 MB Arbeitsspeicher.

5.2.2 Verarbeitungsdauer

Bei der Messung der Verarbeitungsdauer wird die Zeit gemessen welche die Nachrichten zur Verarbeitung benötigen Dabei wird der Weg von dem Hardware-Connector bis zum serialisierten Byte-Strom und/oder anders herum gemessen. In der Messung werden pro Nutzdatengröße

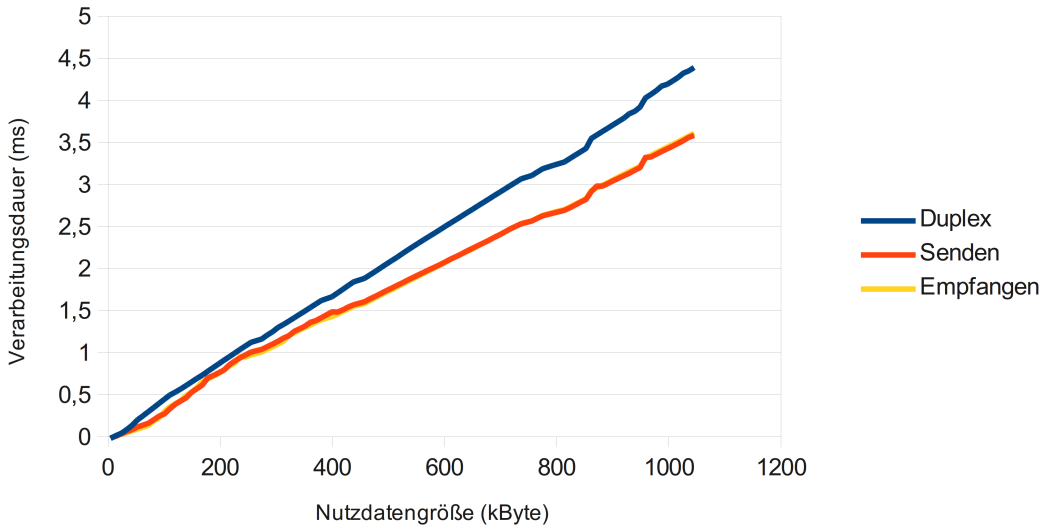

Abbildung 5.2: Verarbeitungsdauer einer Nachricht auf dem Raspberry Pi

die Zeit von 1024 Durchläufen gemessen und danach der Mittelwert errechnet um die Dauer für eine Nachricht zu ermitteln.

Abbildung 5.2 zeigt die Dauer der Nachrichtenverarbeitung in Abhängigkeit der Nutzdatengröße. Die Werte geben dabei in Millisekunden an, wie lang die Verarbeitung einer einzelnen Nachricht dauert. Aus dem Diagramm wird ersichtlich, dass die Verarbeitungsdauer linear zur Größe der Daten ansteigt. Dieser Zusammenhang ist durch die Notwendigkeit begründet, die Nutzdaten innerhalb des Systems zu kopieren, da der Aufwand des Marshalling der Nachrichten sich nicht auf die Länge der Nutzdaten auswirkt (siehe dazu Abschnitt 4.5).

In Senderichtung müssen beim Veröffentlichen der Daten durch den Hardware Connector das erste mal die Nutzdaten kopieren werden. Da davon Ausgegangen wird, dass die Daten nur innerhalb des Ausführungskontext des Hardware-Connectors gültig sind (siehe dazu die Architekturbeschreibung in Abschnitt 3.2), müssen die Daten für die Verwendung in der Event-Queue kopiert werden. Dabei gilt zu beachten, dass bei mehreren Empfängern für jeden Empfänger ein Event erzeugt wird und damit auch jedes mal die Daten kopiert werden müssen.

Diese Art der Speicherverwaltung wurde nicht implementiert, da sich diese Arbeit auf die Architektur und das Netzwerkprotokoll konzentriert. Der zweite Fall in dem die Nutzdaten kopiert werden müssen ist bei dem Marshaling-Vorgang. Hier müssen in den Byte-Strom neben dem serialisierten Header auch die Nutzdaten kopiert werden bzw. bei der Deserialisierung die Nutzdaten aus dem Empfangspuffer kopiert werden.

Bei der Betrachtung der Verarbeitungszeiten gibt es kaum Unterschiede zwischen dem Senden und Empfangen von Daten. Der FDX-Modus ist pro Nachricht langsamer als die beiden Simplex Modi, da mehr Daten gleichzeitig verarbeitet werden müssen. Bei der Betrachtung der Absolutwerte sind entsprechend die informellen Nachrichten ohne Nutzdaten (wie Bestätigungen). Die maximale Rate, mit der diese informellen Nachrichten versendet werden können liegen bei allen Modi weit über 10000 Nachrichten pro Sekunde. Für große Nutzdaten (bei

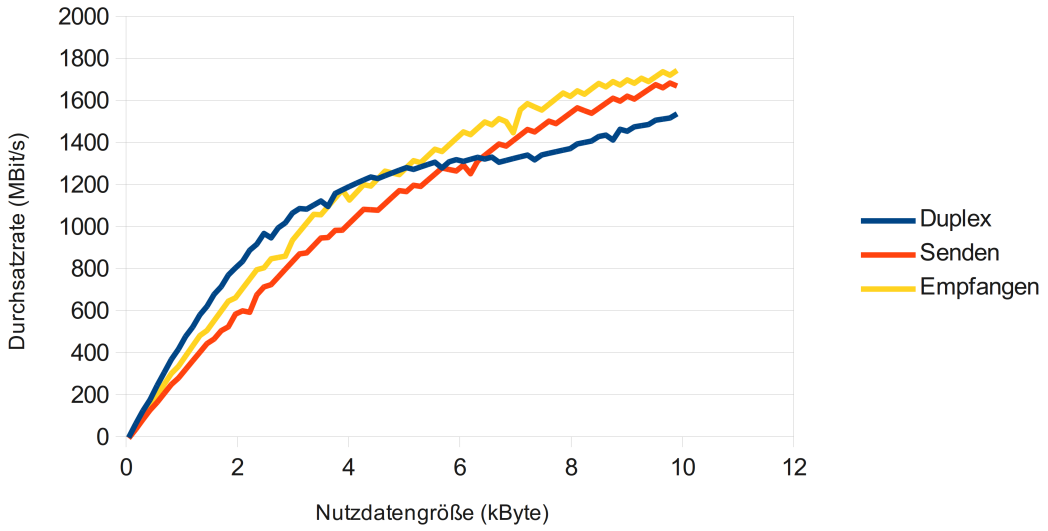

Abbildung 5.3: Durchsatzraten auf dem Raspberry Pi für kleine Nutzdaten

diesen Test bis 1 MB) werden in den beiden Simplex-Modi Raten von über 250 Nachrichten pro Sekunde und im FDX-Modus von über 200 Nachrichten erreicht.

5.2.3 Durchsatzrate

Aus der Verarbeitungsdauer, die im letzten Abschnitt besprochen wurde, lässt sich die Durchsatzrate ermitteln. Diese gibt an wie groß die Menge an Daten ist, die das System pro Sekunde verarbeiten kann. Mit dieser Maßzahl kann man das System mit den Netzwerkstrukturen vergleichen, die es unterstützen soll.

Das Diagramm in Abbildung 5.3 zeigt den Verlauf der Durchsatzrate bei kleinen Datenmengen. Bei dieser Betrachtung wird ersichtlich, dass bei kleinen Datenmengen die Durchsatzrate schnell ansteigt. Bei Nutzdaten bis zu einer Größe von ca. 4 KByte steigt die Durchsatzrate annähernd linear an. Dieser lineare Anstieg lässt sich damit erklären, dass für kleine Nutzdaten zunächst der Verwaltungsaufwand höher ist als der Aufwand für das Kopieren der Nutzdaten im Speicher. Das heißt, der Aufwand der Serialisierung bleibt, wie bereits erwähnt konstant, während die Nutzdatengröße linear ansteigt, ohne signifikante Auswirkung auf die Verarbeitungsdauer zu haben. Aus diesem Grund arbeitet der Duplex-Modus zunächst mit einer höheren Durchsatzrate als die Simplex Modi, da dieser mehr Daten gleichzeitig bei konstanter Verarbeitungsdauer bearbeitet.

Abbildung 5.4 zeigt die Entwicklung der Durchsatzraten bei größeren Datenmengen. Die Spitze zu Beginn der Kurve bei kleinen Nutzdaten erklärt sich dadurch, dass die hardwareseitigen Caches zum Zwischenlagern der Daten genutzt werden können. Durch die Verwendung von Caches muss nicht auf den langsameren Arbeitsspeicher ausgelagert werden, was eine enorme Steigerung der Durchsatzrate zur Folge hat. Werden die Nutzdaten zu groß für den Cache müssen sie in den Arbeitsspeicher kopiert werden. Die Durchsatzrate konvergiert bei

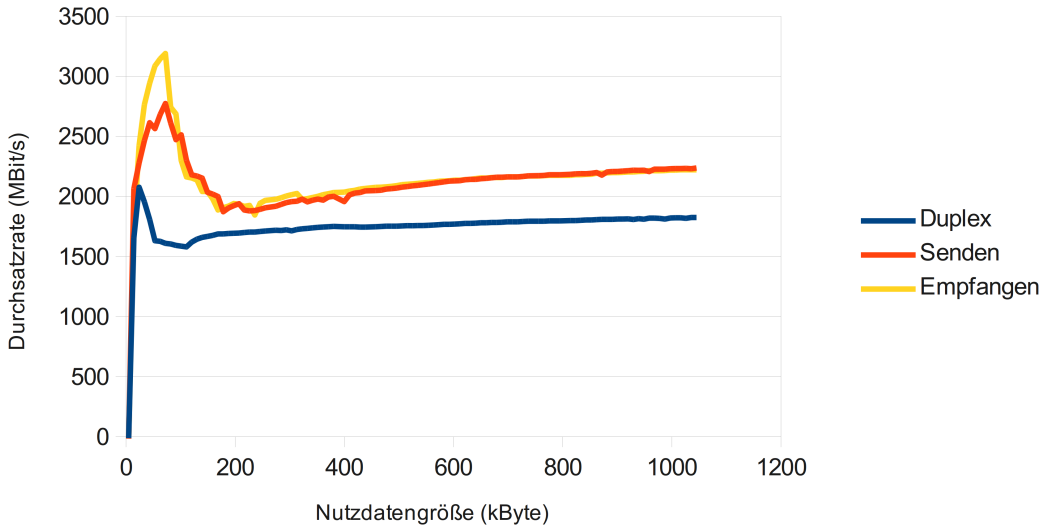

Abbildung 5.4: Durchsatzraten auf dem Raspberry Pi

großen Werten gegen den die maximale Durchsatzrate des Arbeitsspeichers, da das die Leistungsfähigkeit des Kopiervorgangs beschränkt. Die Durchsatzrate für den FDX-Modus ist bei großen Datenmengen nun langsamer, da wie bereits beschrieben mehr Daten gleichzeitig verarbeitet werden und damit auch der Aufwand für das Kopieren der Nutzdaten steigt.

Für die allgemeine Bewertung der Durchsatzrate wird der Maximalwert für große Datenmengen herangezogen. Dieser liegt in den Simplex-Modi über 2 GBit/s und in dem Duplex Modus bei über 1,5 GBit/s. Diese Werte können nun mit dem in Abschnitt 2.2 angesprochenen theoretischen Übertragungsraten von SpaceWire bzw. dessen Nachfolger SpaceFibre verglichen werden.

Bei dem Vergleich sieht man, dass bei großen Datenmengen ein einzelner Interface Node die komplette Durchsatzrate des Netzwerkes ausnutzen kann. Diese Annahmen werden immer unter der Hardwareplattform des Raspberry Pi getroffen. Für schnellere Netzwerke oder leistungsärmere Plattformen bieten sich im Bereich der Speicherverwaltung Optimierungsmöglichkeiten an die die Durchsatzrate erhöhen können.

5.3 Timing

Ein wichtiger Parameter, der in der Übertragungssicherung beachtet werden muss ist das sogenannte Resend-Timeout. Dieser Wert gibt an, wie lang auf eine Nachricht gewartet wird, bis die Nachricht erneut gesendet wird. Wie in Abschnitt 4.7 bereits beschrieben wurde, verwendet die Sicherung positive Bestätigungen. Damit geht das System davon aus, das die Übertragung fehlerhaft war, sollte innerhalb dieses Resend-Timeout keine Nachricht ankommen. Wird der Timeout zu kurz gewählt, wird die Nachricht erneut gesendet, obwohl sie bereits korrekt übertragen wurde. Diese Problem ist als Nachrichtenverdoppelung bekannt. Der Empfänger erhält dabei eine Nachricht zweimal, obwohl sie nur einmal zugestellt werden darf. Anderer-

seits muss der Timeout möglichst kurz gewählt werden um die Gesamtzeit zur Erkennung von Übertragungsfehlern und Verbindungsabbrüchen zu verringern, damit eine möglichst schnelle Reaktion ermöglicht wird.

Die Wahl des minimalen Timeouts ist dabei von dem verwendeten Netzwerk und die Route der Nachrichten abhängig. Das statische Routing hat dabei den Vorteil, dass die maximale Latenz konstant ist, da der Pfad immer über die gleiche Anzahl von Routern läuft. Um nun auszuschließen, dass Bestätigungen zu spät eintreffen, muss dazu die maximale Latenz zu allen Empfängern gemessen werden. Die doppelte Latenzzeit ergibt dann den minimalen Wert des Timeout.

5.4 Determiniertheit

Für den Einsatz in Raumfahrtsystemen ist es wichtig, dass zu jeder Zeit nachvollziehbar ist, in welchem Zustand sich jeder Knoten des Systems befindet, und sich innerhalb des selben Zustand deterministisch Verhält. Nur dadurch ist es möglich, vom Boden aus die Mission zu verfolgen und auf Fehler zu reagieren. Die einzelnen Knoten vorhersagbar zu halten ist Aufgabe des Konfigurationsmechanismus.

Die Vorhersagbarkeit des Systems ist an zwei Punkten gesichert. Der Erste Punkt ist die zentrale Verwaltung der Konfigurationswechsel. Dadurch, dass ausschließlich die Master-Nodes einen Wechsel in andere Modi vornehmen können, kann die aktuelle Konfiguration, in der sich das System befindet von der Bodenstation abgefragt werden.

Dieser Umstand allein reicht jedoch nicht aus um eine Vorhersagbarkeit zu erreichen. Zusätzlich muss auf der Seite der Interface-Nodes sichergestellt werden, dass sich die Knoten innerhalb einer Konfiguration statisch verhalten. Dieser Punkt muss bei der Implementierung der Hardware- und Network-Connectors beachtet werden. Es muss sichergestellt werden, dass nur der Configuration-Handler Parameter des Interface-Nodes festlegen darf (z.B. Einstellungen des Hardware Connector oder Netzwerk-Routen).

Durch diese beiden Punkte kann von der Bodenstation aus Vorhersagen über das Verhalten des Raumfahrtsystems gemacht werden, womit die Missionsphasen im Voraus geplant und simuliert werden können. Bei Verwendung dieser statischen Art der Konfiguration kann sich das System nur begrenzt an Umgebungsänderungen anpassen, nämlich durch einen Wechsel der Konfiguration. Dadurch müssen bei dem Entwurf des Gesamtsystems bereits Probleme wie Lastverteilung betrachtet werden müssen, welche andere Systeme dynamisch zur Laufzeit lösen können.

6 Fazit und Ausblick

Zusammenfassend konnte diese Arbeit zeigen, dass es möglich ist, Konzepte und Techniken aus dem Bereich der Unternehmenssysteme und des High-Performace-Computing an die Bedürfnisse der eingebetteten Systeme anzupassen. Daraus ist dieses Konzept einer Software- und Protokollarchitektur für distributed I/O entstanden.

Neben den Methoden zur Informationsverteilung konnten in dieser Arbeit ebenfalls Vorgaben für die anderen Knoten des OBC-NG umrissen werden. Daher kann diese Arbeit als Grundlage für die Entwicklung der restlichen Knoten im OBC-NG System verwendet werden. Ebenso ist es möglich, auf dem in dieser Arbeit entworfenen Netzwerkprotokoll aufzubauen und um weitere Nachrichten für andere Knoten zu erweitern.

Die Implementierung des Interface-Node hat gezeigt, dass das System eine verlässliche und effiziente Möglichkeit bietet die Informationen innerhalb eines Raumfahrtsystems zu verteilen. Durch den modularen Aufbau der Architektur kann sich das System sich an die Umgebung anpassen und reduziert damit gleichzeitig den Implementierungsaufwand, da die Module wiederverwendet werden können.

Die Evaluation haben gezeigt, dass sich die Architektur und das Protokoll die Anforderungen die Anforderungen der Verlässlichkeit und Leistungsfähigkeit in einem üblichen eingebetteten Umfeld erfüllen kann.

Darüber hinaus bietet das System noch Optimierungsmöglichkeiten, die jedoch über diese Arbeit hinausgehen. Zu diesen Möglichkeiten gehört eine effizientere Speicherverwaltung, bei der die Nutzdaten seltener kopiert werden müssen, und über Referenzen weitergegeben werden. Für diese Optimierung wird jedoch eine Verwaltung der Speicherbereiche der Nutzdaten benötigt, die mehrere Referenzen erkennt und entsprechend freigibt. Bei Testläufen, in denen keine Nutzdaten kopieren wurden, konnten Durchsatzraten bis zu 20 GBit/s auf dem Raspberry Pi erreicht werden. Dadurch wird ersichtlich, dass die Leistungsfähigkeit der Interface-Node noch deutlich gesteigert werden kann.

Neben den Optimierungen der Software, könnte die Leistung ebenfalls durch ein Multicast-fähiges Netzwerk gesteigert werden. Dafür wären jedoch in der Node-Core Komponente starke Änderungen nötig und ein entsprechend angepasster Network Connector.

Abkürzungsverzeichnis

ASN.1	Abstract Syntax Notation One
CRC	Cyclic Redundancy Check
DLR	Deutsches Zentrum für Luft- und Raumfahrt
ESA	European Space Agency
FDX	Full Duplex
HPC	High Performance Computing
IP	Internet Protocol
IPC	Interprocess Communication
MB	Megabyte
MOM	Message Oriented Middleware
OBC-NG	On-Board-Computer Next Generation
OSI	Open Systems Interconnection
RFC	Request For Change
RPC	Remote Procedure Call
Rx	Receiver
SHARC	Software- und Hardware Architektur für rekonfigurierbare Computer
TCP	Transmission Control Protocol
Tx	Transmitter

Literaturverzeichnis

[BSNP+95] BAKHTIARI, Shahram ; SAFAVI-NAINI, Reihaneh ; PIEPRZYK, Josef u. a.: Cryptographic hash functions: A survey. In: *Centre for Computer Security Research, Department of Computer Science, University of Wollongong, Australie* (1995) 16

[Cur04] CURRY, Edward: Message-Oriented Middleware. In: *Middleware for communications*. John Wiley & Sons, Ltd, 2004, S. 1–28 10

[Eur03] EUROPEAN COOPERATION FOR SPACE STANDARDIZATION: *SpaceWire - Links, nodes, routers and Networks*, 24 Januar 2003 26, 29

[Gai02] GAISLER, J.: A portable and fault-tolerant microprocessor based on the SPARC v8 architecture. In: *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, 2002, S. 409–415 4

[Goo12] GOOGLE INC.: *Encoding - Protocol Buffers - Google Developers*. `https://developers.google.com/protocol-buffers/docs/encoding`, 2012. – [Online; Zugriff: 20.08.2013]

[Int94] INTERNATIONAL TELECOMMUNICATION UNION: *Information technology - Open Systems Interconnection - Basic Reference Model: The basic model*. ITU-T Recommendation X.200. `http://www.itu.int/rec/T-REC-X.200-199407-I`. Version: 1994 (ITU-T Recommendation) 21

[Lam86] LAMPORT, Leslie: On interprocess communication. In: *Distributed Computing* 1 (1986), Nr. 2, 86-101. `http://dx.doi.org/10.1007/BF01786228`. – DOI 10.1007/BF01786228. – ISSN 0178–2770 7

[MMFR96] MATHIS, M. ; MAHDAVI, J. ; FLOYD, S. ; ROMANOW, A.: *TCP Selective Acknowledgment Options*. RFC 2018 (Proposed Standard). `http://www.ietf.org/rfc/rfc2018.txt`. Version: Oktober 1996 (Request for Comments) 15

[NL91] NITZBERG, B. ; LO, V.: Distributed shared memory: a survey of issues and algorithms. In: *Computer* 24 (1991), Nr. 8, S. 52–60. `http://dx.doi.org/10.1109/2.84877`. – DOI 10.1109/2.84877. – ISSN 0018–9162 8

[Oli01] OLIVIER DUBUISSON: *ASN.1: Communication Between Heterogeneous Systems*. Morgan Kaufmann, 2001 12

[OMN13] OMNET COMMUNITY: *OMNeT Network Simulation Framework*. `http://omnetpp.org`, 2013. – [Online; Zugriff: 23.08.2013] 28

[PB61] PETERSON, W.W. ; BROWN, D.T.: Cyclic Codes for Error Detection. In: *Proceedings of the IRE* 49 (1961), Nr. 1, S. 228–235. http://dx.doi.org/10.1109/JRPROC.1961.287814. – DOI 10.1109/JRPROC.1961.287814. – ISSN 0096–8390 15, 34

[PMS07] PARKES, Steve ; MCCLEMENTS, Chris ; SUESS, Martin: SpaceFibre. In: *International SpaceWire Conference Dundee 2007* (2007), September, S. 101–108 5

[Pos81] POSTEL, J.: *Transmission Control Protocol*. RFC 793 (INTERNET STANDARD). http://www.ietf.org/rfc/rfc793.txt. Version: September 1981 (Request for Comments). – Updated by RFCs 1122, 3168, 6093, 6528 14, 15

[Riz97] RIZZO, Luigi: Effective erasure codes for reliable computer communication protocols. In: *SIGCOMM Comput. Commun. Rev.* 27 (1997), April, Nr. 2, 24–36. http://dx.doi.org/10.1145/263876.263881. – DOI 10.1145/263876.263881. – ISSN 0146–4833 14

[Roa05] ROACH, A.B.: *A Negative Acknowledgement Mechanism for Signaling Compression*. RFC 4077 (Proposed Standard). http://www.ietf.org/rfc/rfc4077.txt. Version: Mai 2005 (Request for Comments) 14

[SAL+03] STANKOVIC, J.A. ; ABDELZAHER, T.F. ; LU, Chenyang ; SHA, Lui ; HOU, J.C.: Real-time communication and coordination in embedded sensor networks. In: *Proceedings of the IEEE* 91 (2003), Nr. 7, S. 1002–1022. http://dx.doi.org/10.1109/JPROC.2003.814620. – DOI 10.1109/JPROC.2003.814620. – ISSN 0018–9219 5

[SL12] STOHLMANN, Kai ; LÜDTKE, Daniel: SHARC - Bewertung von Netzwerktechnologien / Deutsches Zentrum für Luft- und Raumfahrt. 2012. – Forschungsbericht 29

[SP11] SCHWITZER, Wolfgang ; POPA, Vlad: Using Protocol Buffers for Resource-Constrained Distributed Embedded Systems. 2011. – Forschungsbericht. – available at http://www.in.tum.de/forschung/publikationen/technische-berichte.html 13, 32

[Thu09] THURLOW, R.: *RPC: Remote Procedure Call Protocol Specification Version 2*. RFC 5531 (Draft Standard). http://www.ietf.org/rfc/rfc5531.txt. Version: Mai 2009 (Request for Comments) 9

[WLB13] WESTERDORF, Karsten ; LÜDTKE, Daniel ; BÖRNER, Anko: SHARC Studie / Deutsches Zentrum für Luft- und Raumfahrt. 2013. – Forschungsbericht 1, 17