

**Entwurf und Implementierung einer
Architektur für die Anbindung von
Nutzerschnittstellen an die Steuerung
eines Robotiksystems für die Chirurgie**

Masterarbeit

Florian Roeske



MASTERARBEIT

ENTWURF UND IMPLEMENTIERUNG EINER ARCHITEKTUR FÜR DIE ANBINDUNG VON NUTZERSCHNITTSTELLEN AN DIE STEUERUNG EINES ROBOTIKSYSTEMS FÜR DIE CHIRURGIE

Freigabe:

Der Bearbeiter:

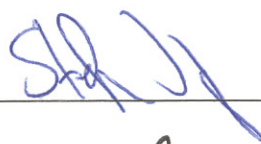
Unterschriften

Florian Roeske



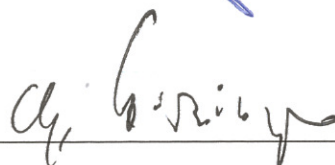
Betreuer:

Dipl.-Ing. (FH) Stefan Jörg



Der Institutsdirektor

Prof. Dr. G. Hirzinger



Dieser Bericht enthält 88 Seiten, 52 Abbildungen und 8 Tabellen



hochschule mannheim



Deutsches Zentrum
für Luft- und Raumfahrt e.V.

Masterarbeit

**Entwurf und Implementierung einer
Architektur für die Anbindung von
Nutzerschnittstellen an die Steuerung eines
Robotiksystems für die Chirurgie**

Eingereicht am 30. August 2011
von Florian Roeske

Matrikelnummer 1027671
Studiengang Informationstechnik
Fakultät für Informationstechnik
Hochschule Mannheim

Betreuer	Prof. Dr. Eckhart Körner	(Hochschule Mannheim)
	Dipl.-Ing. (FH) Stefan Jörg	(Deutsches Zentrum für Luft und Raumfahrt)

Institut für Robotik und Mechatronik
Deutsches Zentrum für Luft und Raumfahrt e.V.

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Masterarbeit

**„Entwurf und Implementierung einer Architektur für die Anbindung von
Nutzerschnittstellen an die Steuerung eines Robotiksystems für die Chirurgie“**

selbständig und ohne Benutzung anderer als die angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen (einschließlich elektronischer Quellen) direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht worden. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer an deren Prüfung noch nicht vorgelegt worden.

Oberpfaffenhofen, den 30. August 2011

Florian Roeske

Zusammenfassung

Das modular aufgebaute MiroSurge-System für die Chirurgierobotik des DLR besteht aus einer Vielzahl von verteilten Regelungskomponenten. Da an diesem System aktiv geforscht und weiterentwickelt wird, muss dieses besonders flexibel und einfach umstrukturierbar sein. Aus diesem Grund wird in dieser Arbeit ein Konzept zur Anbindung von redundanten Benutzerschnittstellen an ein verteiltes Regelungssystem aus Simulink-Modellen vorgestellt. Es wird gezeigt, dass durch die Verwendung der gewählten Middleware ROS, die nötige Anpassbarkeit und Robustheit erreicht wird, die von einer vielseitigen Forschungsplattform erwartet wird. Mit einem Codegenerator werden die Schnittstellen zwischen der Middleware und den Regelungsmodellen generiert.

Inhaltsverzeichnis

1	Einleitung.....	1
1.1	Das DLR MiroSurge-Szenario.....	1
1.2	Systemarchitektur.....	2
1.3	Zielsetzung.....	5
1.4	Aufbau der Arbeit.....	6
2	Problemanalyse.....	7
2.1	Schnittstellen zur Roboterhardware.....	7
2.2	Regelungskonzept.....	8
2.3	Reglerzustandsmaschinen.....	10
2.4	Benutzerschnittstellen des MiroSurge-Systems.....	11
2.5	Diskussion.....	13
3	Auswahl der Middleware.....	15
3.1	Grundlagen zu Middleware.....	15
3.2	Auswahlkriterien für eine Middleware-Lösung.....	17
3.3	Existierende Middleware-Lösungen.....	18
3.3.1	Objektorientierte Middleware-Lösungen.....	18
3.3.2	RPC-Lösungen.....	19
3.3.3	Messaging Bibliotheken.....	19
3.3.4	Middleware für die Robotik.....	20
3.4	Diskussion.....	22
4	Konzept für die Anbindung von Nutzerschnittstellen.....	24
4.1	Konzept.....	24
4.1.1	Übertragung von Parametern und Befehlen an ein Programm.....	24
4.1.2	Zwei Programme mit sich überschneidenden Schnittstellen.....	28
4.1.3	Exklusiver Zugriff.....	29
4.1.4	Design der Simulink-Blöcke.....	30
4.1.5	Codegenerator.....	32
4.2	Validierung des Konzepts mittels einer Beispielanwendung.....	32
5	Implementierung der ROS-Simulink-Schnittstelle.....	36
5.1	ROS Pakete und IDL.....	36
5.2	ROS Topics.....	37
5.3	ROS C++-Schnittstelle.....	37
5.4	Portierung der ROS-Bibliotheken auf QNX 6.5.....	39
5.5	Simulink S-Functions.....	40
5.6	Aufbau der Simulink-Blöcke.....	41
5.6.1	Initialisierung der Simulink S-Functions.....	42
5.6.2	Initialisierung der S-Functions für den External-Mode.....	42
5.6.3	Bearbeitung eintreffender Nachrichten.....	43
5.6.4	Aufbau des Subscriber-Blocks.....	43
5.6.5	Aufbau des Service-Blocks.....	44
5.6.6	Aufbau des Publisher-Blocks.....	45
5.7	Aufbau des Codegenerators.....	46

6 Analyse und Test der Implementierung.....	49
6.1 Zeitmessungen der Simulink-Blöcke.....	49
6.1.1 Zeitmessungen des Publisher-Blocks.....	50
6.1.2 Zeitmessungen des Subscriber-Blocks.....	52
6.1.3 Zeitmessungen des Service-Blocks.....	53
6.1.4 Diskussion.....	55
6.2 Integration der Schnittstellen in die MIRO-Einarmapplikation.....	59
6.2.1 Entwurf der Schnittstellen für die MIRO-Einarmapplikation.....	59
6.2.2 Integration der Simulink-Blöcke in das Regelungsmodell.....	61
6.2.3 Client-Anwendungen für die MIRO-Einarmapplikation.....	62
6.2.4 Test der überarbeiteten MIRO-Einarmapplikation.....	64
6.2.5 Ergebnisse.....	66
7 Weiterführende Arbeiten und Ausblick.....	67
7.1 Weiterführende Arbeiten.....	67
7.2 Ausblick.....	68
8 Zusammenfassung.....	69
Literaturverzeichnis.....	70
Abbildungsverzeichnis.....	74
Tabellenverzeichnis.....	77
Anhang.....	78

1 Einleitung

Robotiksysteme für die Chirurgie werden mit der Intention vorangetrieben postoperative Traumata und Infektionsrisiken bei Patienten zu verringern. Teleoperative Systeme für die Chirurgie sollen den Chirurgen dabei unterstützen. Durch einen schnelleren und präziseren Eingriff soll der Genesungsprozess des Patienten verkürzt werden. Medizinroboter können im endoskopischen minimal-invasiven Bereich oder bei offenen Operationen verwendet werden.

1.1 Das DLR MiroSurge-Szenario

Im Rahmen des MiroSurge-Projekts [1] [2] [3] arbeitet das Deutsche Zentrum für Luft- und Raumfahrt seit geraumer Zeit an einer Forschungsplattform für Anwendungen in der Medizinrobotik. Das MiroSurge-Szenario (siehe Abbildung 1.1) bildet ein Anwendungsszenario für die Forschung an der minimal-invasiven Chirurgie.

Die MiroSurge-Forschungsplattform besteht aus einer Vielzahl verschiedener Hardware- und Softwarekomponenten. Der Grundstein des Systems wurde mit dem MIRO-Roboterarm [4], einem speziellen Leichtbauroboterarm mit 7 Freiheitsgraden, gesetzt. An ihm können wechselbare Werkzeuge oder ein Endoskop angebracht werden.



Abbildung 1.1: DLR MiroSurge-System mit künstlichem Herzen.

Die MICA [5] ist ein am DLR entwickeltes Werkzeug für das MiroSurge-Szenario. Sie ist mit Sensoren und Aktuatoren ausgestattet und kann mit verschiedenen Zangen und Schneidewerkzeugen ausgestattet werden. Zudem kann die MICA zur Laufzeit mit dem MIRO-Roboter verbunden oder gewechselt werden.

Außerdem gibt es die Eingabekonsolle, von der aus der Chirurg einmal Operationen durchführen soll. Sie besteht aus zwei haptischen Eingabegeräten, mit denen sich die Roboterarme mit den angeschlossenen Werkzeugen steuern lassen, und einem 3D Monitor (siehe Abbildung 1.2). Auf dem Monitor wird das Bild eines Stereo-Endoskops als dreidimensionales Bild dargestellt mit dem der Chirurg die Positionen seiner Werkzeuge im Patienten besser einschätzen kann.

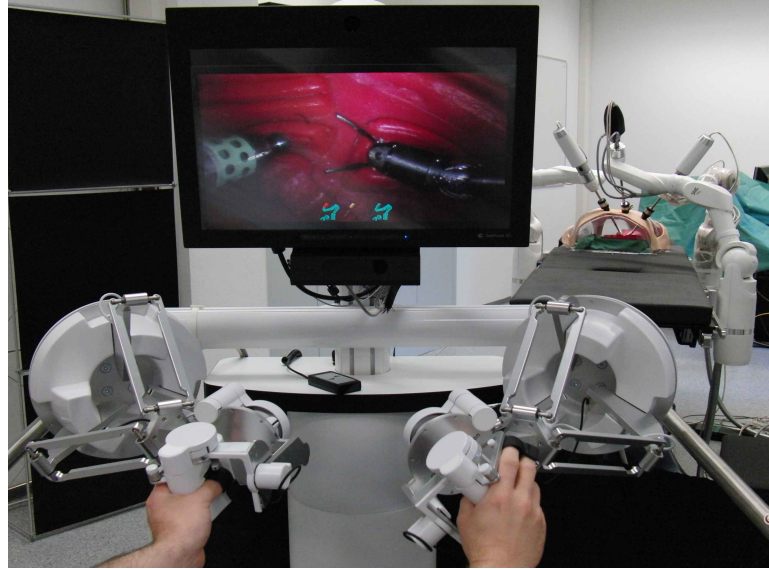


Abbildung 1.2: Eingabestation des DLR MiroSurge-Szenarios mit 3D-Monitor und haptischen Eingabegeräten.

Der Fokus des Hardwaredesigns richtet sich bei diesem System darauf, ein leichtes und vielseitiges System mit geringen Ausmaßen zu entwickeln. Dadurch wird in beengten Operationssälen nicht zu viel Platz eingenommen. Der Roboterarm besitzt ein Gewicht von ca. 10kg und ist frei am Operationstisch positionierbar. Dies ermöglicht die flexible Anpassung des Systems an verschiedene Operations-Szenarien.

1.2 Systemarchitektur

Für die teleoperative Steuerung und Regelung von drei MIRO-Armen, einer davon mit Endoskop und zwei mit MICA-Werkzeugen ausgestattet, ist eine komplexe Systemarchitektur erforderlich. Das MIRO-System ist eine Forschungsplattform, die ständig weiterentwickelt wird. Aufgrund der andauernden Veränderung der mechatronischen Komponenten und der Systemkonfiguration, hat man die Architektur des Systems in verschiedene Abstraktionsebenen unterteilt und die Steuerungen der einzelnen Hardwarekomponenten logisch wie auch physikalisch getrennt. Dies hat zur Folge, dass die Regelungsanwendungen der einzelnen Komponenten auf getrennten PCs ausgeführt werden. Um zusätzliche Flexibilität zu gewährleisten, werden Standard PCs eingesetzt.

Die Steuerung der Roboter erfordert aufwändige Regelungsanwendungen. Für die Modellierung der Regelschleifen wird Matlab/Simulink [6] verwendet. Mit dem Real-Time

Workshop¹ [7] von Matlab/Simulink wird aus den Modellen Code generiert. Dieser wird auf Standard PCs mit dem Betriebssystem QNX Neutrino [8] ausgeführt und erfüllt harte Echtzeitkriterien.

Diese Kombination aus einem POSIX kompatiblen Echtzeitbetriebssystem und Matlab/Simulink bietet eine komfortable Möglichkeit, modellbasierte Softwareentwicklung für die Regelungstechnik der MiroSurge-Komponenten zu betreiben und gleichzeitig die harten Echtzeit Anforderungen, mit Regelschleifen von bis zu 10kHz, zu erfüllen.

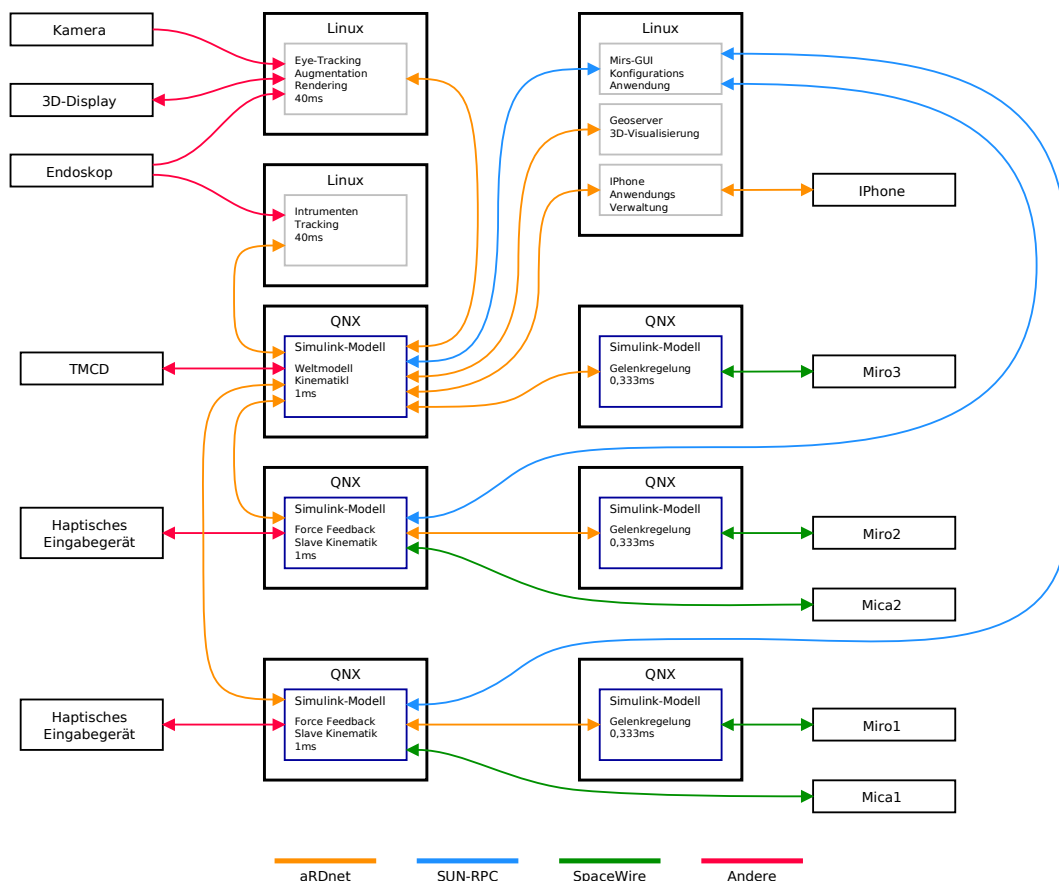


Abbildung 1.3: Architektur des verteilten MiroSurge-Regelungssystems mit den verschiedenen Kommunikationsarten.

Die MiroSurge-Forschungsplattform besteht aus mehreren QNX Rechnern für die Regelungsanwendungen und Linux Rechnern für die Steuerung und Überwachung des Systems. Die Basis der Regelungsanwendung bildet ein System aus verteilten Simulink-Modellen, zwischen denen Daten mit 1kHz ausgetauscht werden.

Abbildung 1.3 zeigt ein Übersichtsdiagramm der verteilten Regelungsanwendungen. In dieser Abbildung lässt sich erkennen, dass jeder Roboterarm ein eigenes Simulink-Modell für die Positionsregelung besitzt, das auf physikalisch getrennten Rechnern ausgeführt wird.

¹ Der Real-Time Workshop ist seit der Matlab/Simulink Version R2011a mit dem Stateflow Coder verbunden und in Simulink Coder umbenannt worden.

Zwei weitere Modelle nehmen die Benutzereingaben der haptischen Eingabegeräte entgegen und steuern jeweils einen Roboterarm. Zusätzlich fließen Daten aus dem gesamten Szenario in ein sogenanntes Weltmodell. Von hier aus wird das gesamte System für Anwendungsszenarien gesteuert. Das Weltmodell bietet die Möglichkeit Einfluss auf das gesamte System zu nehmen.

Durch die Notwendigkeit des Zusammenführens von Daten im Weltmodell muss ein Datenaustausch zwischen den verschiedenen Simulink-Berechnungsmodellen stattfinden. Dieser Datenaustausch wird mit einer Rate von 1kHz durchgeführt.

Somit besteht das gesamte verteilte Regelungssystem aus sechs Simulink-Modellen, die auf jeweils einem Standard PC mit dem Echtzeitbetriebssystem QNX ausgeführt werden.

Für die Kommunikation der Simulink-Modelle wird aktuell aRDnet [9] (agile Robot Development) verwendet. Die aRDnet Kommunikation ist ein am DLR entwickelter Ansatz für den Datenaustausch zwischen komplexen mechatronischen Systemen. Über einen UDP (User Datagram Protocol) basierten Kommunikationsstream werden Daten eines Shared Memory Bereiches zu einem anderen Rechner transferiert. Mit diesem Kommunikationsmodell lassen sich verteilte Systeme erstellen. Es bietet Kommunikationsblöcke für Simulink und C Bibliotheken für Windows, Linux, QNX und iOS. Zudem unterstützt aRDnet die für die Regelung des MiroSurge-Projekts geforderten Datenraten von 1kHz zum Austausch der Daten zwischen den Modellen.

Neben der echtzeitfähigen Kommunikation zwischen den Simulink-Modellen, sollen während der Laufzeit Modifikationen in den diesen Modellen über Benutzerschnittstellen erfolgen, die das Einstellen der Regler ermöglichen. Dies beinhaltet das Starten und Stoppen von Reglern sowie das Wechseln zwischen verschiedenen Reglermodi. Darüber hinaus ist die Möglichkeit gefordert, Daten aus den Modellen in anderen Anwendungen zu nutzen. Zum Beispiel zur Visualisierung der aktuellen Pose der Roboterarme oder zum Debuggen.

Der Zugriff auf die Reglereinstellungen erfolgt im MiroSurge-Projekt durch eine RPC-Schnittstelle (Remote Procedure Call). Hierfür wird Sun RPC² verwendet. Daten für die Visualisierung und zum Debuggen werden über eine aRDnet Kommunikation aus dem Weltmodell gesendet.

Für die Echtzeitkommunikation und die Steuerungs- und Überwachungsaufgaben zwischen den QNX und Linux Systemen gibt es zwei getrennte Ethernet-Netzwerke. Die QNX Rechner sind über die erste Netzwerkkarte mit dem Institutsnetzwerk verbunden und mit der zweiten an ein eigenes Netzwerk, das nur für die Echtzeitkommunikation zuständig ist. Das Echtzeitnetz dient nur für die aRDnet Kommunikation zwischen den Modellen und besteht aus Punkt-zu-Punkt Verbindungen. Die Echtzeitfähigkeit wird dadurch erreicht, dass nur ein Prozess pro Rechner auf die Netzwerkschnittstelle zugreift. Über das Institutsnetzwerk werden von den Linux Rechnern aus Einfluss auf die Regelungsanwendungen genommen.

² Sun RPC oder ONC RPC (Open Network Computing Remote Procedure Call) ist ein System für RPC Aufrufe und unter Unix/Linux meist standardmäßig installiert.

1.3 Zielsetzung

Das Gesamtsystem des MiroSurge-Projekts besteht aus mehreren Linux und QNX Systemen, auf denen verschiedene Simulink-Modelle für die Regelungsanwendungen ausgeführt werden.

Ziel dieser Arbeit ist es, ein flexibles Konzept für die Anbindung der Benutzerschnittstellen an die mit Matlab/Simulink erzeugten Regelungsmodelle zu entwickeln. Die Architektur für die Echtzeitkommunikation zwischen den Modellen, sowie die Modelle selbst, sollen unverändert bleiben. Das erstellte Konzept soll die Möglichkeit bieten, Einstellungen in den Modellen vorzunehmen und auch Daten aus den Modellen an verschiedene Benutzerinterfaces zu übertragen.

Die beiden verwendeten Kommunikationsframeworks, aRDnet und SUN-RPC, bieten keine komfortable Möglichkeit IP-Adressen und Ports zu verwalten. Damit sind die Einstellungen der Szenarien statisch auf eine Netzwerkkonfiguration zugeschnitten. Änderungen in der Architektur der Regler oder der Netzwerkkonfiguration erfordern eine Neukonfiguration des entsprechenden Simulink-Modells und von Shell-Sripten für das aRDnet. Gerade zu Visualisierungs- und Debuggingzwecken soll aber eine flexible Möglichkeit bestehen nicht auf eine bestimmte Konfiguration angewiesen zu sein. Durch die Verwendung einer Middleware können IP-Adressen und Port-Nummern abstrahiert werden und die Flexibilität des Systems dadurch gesteigert werden.

Teil der Arbeit ist es damit, das aktuelle System zu analysieren und die Anforderungen herauszuarbeiten. Verschiedene Middleware-Lösungen müssen abschließend entsprechend diesen Anforderungen untersucht werden.

Mit Hilfe der ausgewählten Middleware-Lösung soll ein neues Konzept für die Anbindung von Nutzerschnittstellen an das System entwickelt werden. Zudem soll das erarbeitete Konzept, beispielhaft an einem Teil des MiroSurge-Systems implementiert und getestet werden.

1.4 Aufbau der Arbeit

In **Kapitel 2** wird das Regelungskonzept erläutert und es werden die Probleme der aktuellen Anbindung der Nutzerschnittstellen an das MiroSurge-Projekt analysiert, um die Anforderungen an eine Middleware-Lösung und das Lösungskonzept zu bestimmen.

In **Kapitel 3** wird ein Überblick über Middleware-Lösungen gegeben. Es werden verschiedene Lösungen nach den gegebenen Kriterien untersucht und anhand der Anforderungen eine passende Lösung ausgewählt.

In **Kapitel 4** wird mit den in 2 und 3 gewonnenen Erkenntnissen ein eigenes Konzept zur Anbindung von redundanten Benutzerschnittstellen an ein Simulink-Modell erarbeitet.

In **Kapitel 5** wird die Implementierung der ROS-Simulink-Anbindung vorgestellt. Dabei wird der Aufbau der einzelnen Blöcke erläutert und ein Codegenerator zum automatischen Erzeugen der Simulink-Blöcke gezeigt.

In **Kapitel 6** wird die Implementierung der Simulink-Blöcke analysiert. Es werden Messungen durchgeführt, um sicherzustellen, dass die Implementierung die Echtzeitfähigkeit des Systems nicht beeinträchtigt. Zudem wird das gesamte Konzept auf ein konkretes Beispiel angewendet und getestet.

In **Kapitel 7** wird ein Überblick über das in dieser Arbeit entstandene Konzept zur Anbindung von Benutzerschnittstellen an die verteilte Regelungsanwendung gegeben. Dazu werden Verbesserungsmöglichkeiten vorgeschlagen, sowie ein Ausblick für langfristige Ziele gegeben.

2 Problemanalyse

In diesem Kapitel wird zu Anfang weiter auf das MiroSurge-System und die zu Grunde liegenden Konzepte eingegangen, um die Randbedingungen der Arbeit spezifizieren zu können. Dazu wird auf das Konzept für die Regelungsanwendungen eingegangen und die Anbindungen der verschiedenen Benutzerschnittstellen vorgestellt. Dabei wird untersucht, wie die Anbindungen der verschiedenen Benutzerschnittstellen an das Gesamtsystem stattfinden, um später eine passende Middleware-Lösung auswählen zu können.

2.1 Schnittstellen zur Roboterhardware

Im MiroSurge-System wird grundsätzlich zwischen Anwendungs- und Plattformfunktionalität unterschieden. Mit der Plattform wird die Struktur des Systems, von den Sensoren und Aktoren bis hin zu den PCs sowie die zu verwendenden Kommunikationsprotokolle vorgegeben. Die Struktur der Plattform wird durch Unterteilung des Gesamtsystems in insgesamt vier Ebenen erreicht (siehe Abbildung 2.1). Die unterste Ebene wird durch die Sensoren und Aktoren gebildet. Auf der darüber liegenden Ebene werden logische Einheiten, wie z.B. Gelenke, gebildet, die von FPGAs gesteuert werden. Die FPGAs kommunizieren mit den Sensoren und Aktoren über eine BiSS-Schnittstelle (Bidirektional/Seriell/Synchron) [10].

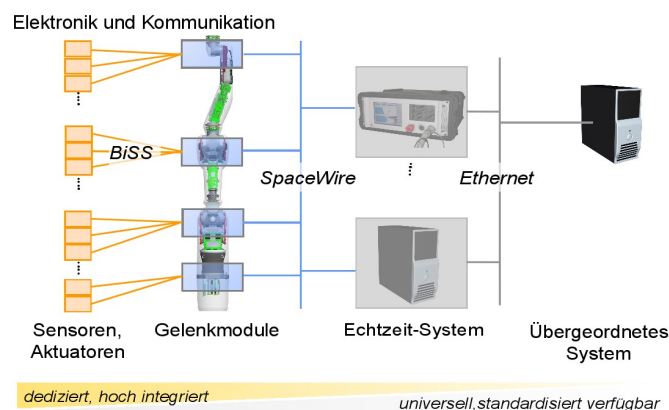


Abbildung 2.1: Kommunikationsebenen der MIRO Hardware.

Die Verbindung zwischen den Echtzeitsystemen und den einzelnen Gelenken stellt die HAL (Hardware Abstraction Layer) dar. Die Kommunikation erfolgt über den SpaceWire-Bus [11], welcher eine geringe Latenz (Roundtrip $< 50 \mu\text{s}$) und eine hohe Bandbreite (1Gbit/s) bietet und auf den FPGAs der Gelenkmodule implementiert ist.

Die Regelungsanwendungen werden auf den QNX Echtzeitsystemen ausgeführt und bilden die Anwendungsebene. Von dieser Ebene kann auf die einzelnen HAL-Schnittstellen der Gelenke zugegriffen werden. Die Schnittstellen werden in Form von Matlab/Simulink Blöcken zur Verfügung gestellt. Damit können die Anwendungsentwickler bequem auf die

verschiedene Hardware zugreifen, ohne sich um die unterste Hardwareebene kümmern zu müssen. Die oberste Anwendungsebene bilden die nicht echtzeitfähigen Linux Systeme.

2.2 Regelungskonzept

Die Regelungsanwendungen werden mit Matlab/Simulink entworfen. Diese Tools werden im MiroSurge-Projekt für die Entwicklung und Simulation der Regler für die hochdynamischen, mechatronischen Anwendungen verwendet. Die Programmierung in Simulink geschieht Datenstrom-orientiert. Daten werden in Quellen erzeugt oder in das Modell hinein geleitet. Entlang an einem benutzerdefinierten Pfad werden Berechnungen durchgeführt. An einer Senke werden die Ergebnisse der Berechnungen angezeigt oder zur Weiterverarbeitung herausgegeben. Damit besteht ein Simulink-Modell aus einem Netz von Kommunikationsblöcken die Ein- und Ausgänge besitzen und miteinander verbunden werden können. Die Blöcke können auch in sogenannten Subsystemen verschachtelt werden. Subsysteme entsprechen Blöcken, in denen weitere Blöcke abgelegt werden können. Somit ist es möglich, Modelle in verschiedene Ebenen einzuteilen und logisch zu gruppieren. Die Berechnungsreihenfolge der Blöcke und die Art der Berechnung wird mit dem sogenannten Solver gewählt. Solver sind Algorithmen zur numerischen Integration, mit deren Hilfe das durch das Modell definierte System berechnet wird. Simulink enthält Solver, mit denen sich zeitkontinuierliche (analoge), zeitdiskrete (digitale) Systeme berechnen lassen. Bei den Komponenten im MiroSurge-Projekt wird der diskrete Solver verwendet. [12]

Mit Simulink selbst wird eine große Anzahl an Modulen für verschiedenste Anwendungen mitgeliefert. Des Weiteren besteht die Möglichkeit, eigene Blöcke und Blockbibliotheken für Simulink, den sogenannten „S-Functions“ (System Functions) zu erstellen. Diese können in der eigenen Matlab Programmiersprache oder in C/C++ erstellt werden. Mit selbst erstellten Blöcken ist es so zum Beispiel möglich, aus den Modellen heraus direkt auf Hardwaretreiber zuzugreifen.

Mit dem Realtime Workshop (RTW) besteht die Möglichkeit mit Hilfe von Codegeneratoren aus den getesteten Simulink-Modellen ausführbare Programme für verschiedene Zielplattformen zu erzeugen. Auf diese Weise werden die auf den Echtzeitrechnern lauffähigen Programme erzeugt. Mit Simulink kann zum Debuggen der Regelungsanwendungen, eine Verbindung zu den Echtzeitrechnern hergestellt werden. Dies wird als „External-Mode“ bezeichnet. In diesem Modus werden Berechnungen der Regelungsanwendung nur auf der Zielplattform vorgenommen und über eine UDP Verbindung in das Modell der Benutzeroberfläche zur Visualisierung und Manipulation übertragen.

Das Design der Regler erfolgt nach einem domänenspezifischen Designmodell und wird als „virtueller Pfad“ (Virtual Path) [13] bezeichnet. Dieses Entwurfsmodell ist im Rahmen des MiroSurge-Projekts am DLR entstanden. Das Ziel dieses Entwurfsmodells ist es, den Entwickler bei der Erzeugung von robusten und deterministischen Regelungsanwendungen zu unterstützen. Der virtuelle Pfad beschreibt die Regelschleife, die auf dem Echtzeitsystem

ausgeführt wird. Er beginnt mit einem Sensor und endet mit einem Aktor in der physikalischen Welt (siehe Abbildung 2.2).

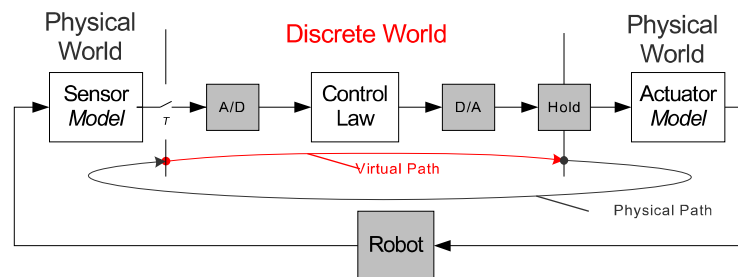


Abbildung 2.2: Der virtuelle Pfad beginnt in der physikalischen Welt beim Sensor und endet beim Aktor. [13]

Das Berechnungsmodell des Reglers wird mit dem Sensor an der physikalischen Zeit synchronisiert. Somit ist die Berechnungsreihenfolge im Regler festgelegt und ein unidirektionaler diskreter Signalfluss wird erzeugt. Die Synchronisierung der Zeit in den Simulink-Modellen erfolgt durch die HAL-Blöcke die den Berechnungstakt (Rate) der Modelle vorgeben und auf die Sensorik der Hardware synchronisiert sind. In einem Berechnungstakt wird das gesamte Modell einmal berechnet. Es wird also ein synchrones Berechnungsmodell genutzt. Im Regler selbst ist der Algorithmus für die Regelung des Systems implementiert. Im Aktor werden die errechneten Signale überprüft und somit sichergestellt, dass das System nicht in einen kritischen Zustand versetzt wird. Dazu befindet sich im Aktor der jeweiligen Hardware eine Zustandsmaschine. In der Aktorzustandsmaschine (siehe Abbildung 2.3) wird die Fehlerüberprüfung des Eingangssignals vorgenommen. Diese umfasst eine Überprüfung des Stellwerts und Zeitstempels. Ist zum Beispiel der Zeitstempel zu alt oder ein Wert außerhalb des gültigen Bereiches, wird dies als Fehler erkannt und der Aktor wird gestoppt.

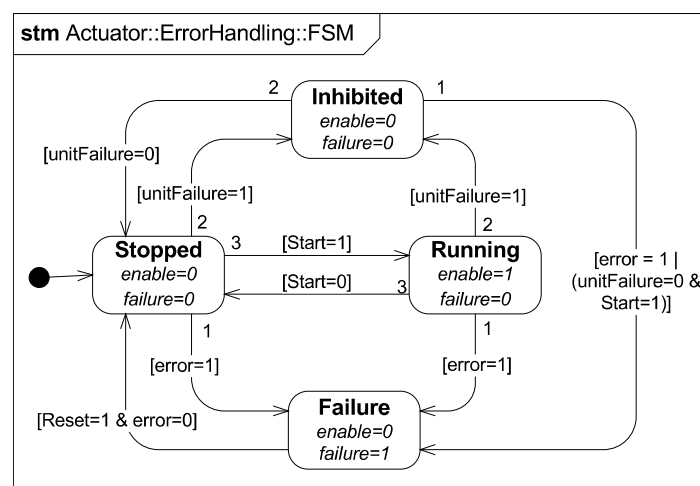


Abbildung 2.3: Aktor Zustandsmaschine. [13]

Die Zustandsmaschine besitzt vier Zustände. „Stopped“, „Running“, „Failure“ und „Inhibited“. Die HAL bietet eine Schnittstelle zu der Aktorzustandsmaschine und reicht die Running- und Failure-Signale weiter. Neben dem Steuerungssignal hat die HAL noch die Signaleingänge „Start“, „Stop“ und „Reset“, um die Hardware zu Steuern.

Dieses Konzept unterstützt die Entwickler bei dem Entwurf von Regelungen für komplexe mechatronische Systeme, indem ein synchrones, deterministisches und sicheres Design in die Praxis umgesetzt wird.

2.3 Reglerzustandsmaschinen

Neben der im vorherigen Teil erwähnten Zustandsmaschine in der HAL, zur Validierung der Steuerungssignale, sind in den Reglern noch weitere Zustandsmaschinen vorhanden. Die wichtigste Zustandsmaschine befindet sich im Regelkreis, der für die Steuerung der Gelenkmodule des MIRO-Roboters zuständig ist. Diese Regelungsanwendung besitzt die folgenden drei Reglermodi um die Positionen der MIRO-Arme zu Steuern:

- **Position Control:**

Positionsregelung, es wird eine kartesische Zielposition angesteuert.

- **Gravity Compensation:**

Schwerkraftkompensation, der Arm hält eine bestimmte Position „locker“ so, dass ein Benutzer den Roboterarm führen kann.

- **Impedance Control:**

Wie die Schwerkraftkompensation, allerdings fährt der Arm nach verschieben des Benutzers seine Soll-Position wieder an. Zudem erzeugt der Roboter eine Gegenkraft beim verschieben die mit der Entfernung von seiner Soll-Position zunimmt.

Die dazugehörige Zustandsmaschine zum Umschalten zwischen den Positionsregelungen besitzt daraus resultierend vier Zustände „position interface“, „impedance interface“, „gravity compensation“ und „halt“.

Die Problematik beim Umschalten zwischen den verschiedenen Positionsregelungen ist das Überprüfen des Umschaltprozesses. Es muss sichergestellt werden, dass das Umschalten sicher ist. Schnelle oder große Positionsänderungen beim Umschalten der Regler hätten bei einem Medizinroboter der direkt am Patienten arbeitet fatale Folgen. Dazu laufen alle Positionsregelungen parallel und beim Umschalten der Regler werden die Steuerungssignale zwischen denen umgeschaltet werden soll auf zu große Abweichungen überprüft. Ist die Abweichung im Toleranzbereich wird der Zustand und somit auch der Modus gewechselt. Falls dies nicht eintritt verbleibt die Regelung in ihrem vorherigen Zustand.

Beim Umschalten der Reglerzustandsmaschinen ist zu erkennen, dass das Signal vom Benutzer gesendet wird und einem Änderungswunsch der Parameter im Reglermodell entspricht. Dieser Änderungswunsch ist asynchron zum Berechnungstakt des Reglermodells und muss daher nicht sofort umgesetzt werden. Die Reaktion des Systems ist daher auch ein asynchrones Ereignis.

2.4 Benutzerschnittstellen des MiroSurge-Systems

Das MIRO-System benötigt für die Teleoperation und die Vorgabe der Reglersollwerte verschiedene Benutzereingabegeräte. Für ein teleoperativ gesteuertes Robotersystem bildet das haptische Eingabegerät die wichtigste Schnittstelle zwischen Roboter und Benutzer. Im MiroSurge-Projekt werden zwei Eingabegeräte vom Typ Sigma7 der Firma Force Dimension [14] (siehe Abbildung 2.4) verwendet.

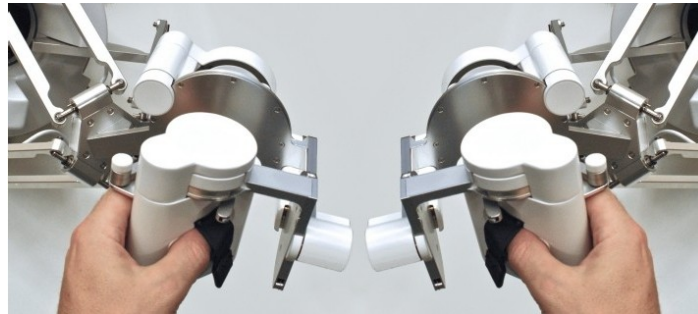


Abbildung 2.4: Haptisches Eingabegerät Sigma7.

Das Sigma7 besitzt 7 Freiheitsgrade und bietet mit einem Force Feedback Modus eine intuitive Möglichkeit Rückmeldungen aus dem Roboter an den Benutzer zu leiten. Das Sigma7 interagiert direkt mit den Reglermodellen der MIRO-Roboterarme sowie den MICA Werkzeugen und ist daher mit einer Taktrate von 1kHz direkt mit den Echtzeitsystemen verbunden (vgl. Abbildung 1.3).

Robotiksysteme die nahe am Menschen im Einsatz sind, müssen aus Sicherheitsgründen immer über einen Notausschalter verfügen. Am MIRO-System ist das am DLR entwickelte TMCD (Tele-Manipulation Command Device) im Einsatz (siehe Abbildung 2.5).



Abbildung 2.5: TMCD, Notausschalter mit konfigurierbaren Tasten und SpaceMouse.

Zusätzlich zu einem Notausschalter nach Industriestandard bietet das TMCD noch weitere Eingabemöglichkeiten durch den Benutzer. Auf dem TMCD ist eine SpaceMouse als weiteres Eingabegerät mit sechs Freiheitsgraden angebracht. Zudem stehen sechs frei programmierbar und beleuchtbare Tasten zur Verfügung. Der Notaus des TMCD ist direkt mit dem Steuerungsrack des Roboters verbunden. Bei einem betätigen des Schalters werden alle Motoren sofort gestoppt und die Bremsen aktiviert. Das TMCD ist über eine serielle Schnittstelle an den Rechner angeschlossen, auf dem das Weltmodell ausgeführt wird. Der

Zugriff auf die Daten dieses Gerätes ist über eine C-Bibliothek und entsprechende Simulink-Blöcke möglich.

Das Mirs-GUI (Minimally Invasive Robotic Surgery - GUI) ist eine mit Qt [15] entwickelte Benutzeroberfläche (siehe Abbildung 2.6). Sie bildet die eine wichtige Schnittstelle zur Konfiguration der Regelungsanwendung.

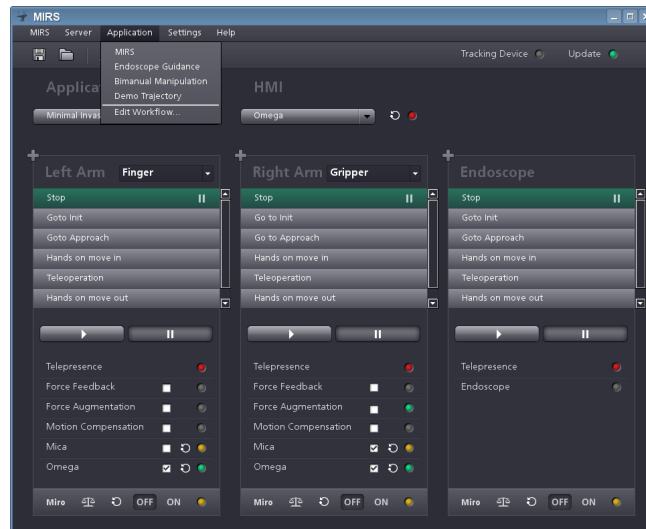


Abbildung 2.6: Mirs-GUI Anwendung für die Steuerung des MiroSurge-Systems.

Mit der Mirs-GUI wird die Roboter-Hardware von einem Linux Computer aus gestartet und gestoppt. Die Anwendung ermöglicht den Wechsel der MICA Werkzeuge und das Umschalten der verschiedenen Regler. Die Verbindung zur Regelungsanwendung erfolgt über eine RPC Schnittstelle mit dem Weltmodell und den MICA Modellen (vgl. Abbildung 1.3). Die Aktualisierung der GUI erfolgt über zyklische RPC Aufrufe (Polling). Zudem übernimmt die Mirs-GUI Teile des für eine Operation notwendigen Arbeitsablaufes, wie z.B. das Anfahren von bestimmten Posen.

Für Operationsplanungsvorgänge gibt es die Möglichkeit direkt vom Operationstisch aus mit einer iPhone-Anwendung auf die Roboter zuzugreifen. Zu den Planungsaufgaben gehören zum Beispiel das Scannen des Patienten mit einem Laserscanner der an einem Roboter angebracht wird. Die gewonnenen Daten können für die Positionierung der Roboter während einer Operation verwendet werden. Diese Anwendung direkt am Roboter ist für Planungsaufgaben sehr wichtig, da sie später einmal direkt am Patienten durchgeführt werden sollen und es in diesem Fall eher umständlich ist zwischen den Robotern, dem Patienten und den Workstations mit der Planungsanwendung zu wechseln. Das iPhone ist über eine aRDnet-Verbindung mit einem Linux-Rechner verbunden der die Anfragen an andere Anwendungen und Simulink-Modellteile weiterleitet.

Der Geoserver ist eine Visualisierungsanwendung mit dem das gesamte MiroSurge-Szenario in einer 3D-Umgebung dargestellt werden kann. Für die Darstellung der MIRO-Arme werden daher die aktuellen Gelenkwinkel benötigt. Die Gelenkwinkel werden über eine aRDnet-Schnittstelle aus dem Weltmodell an den Geoserver gesendet.

2.5 Diskussion

Für die spätere Auswahl einer geeigneten Middleware-Lösung und für den Entwurf des Lösungskonzepts, ist eine weitere Analyse der Anwendungen notwendig. Dazu wird zwischen Benutzerschnittstellen, die sich direkt auf die Regelkreise auswirken und Benutzerschnittstellen zum Aufrufen von Funktionen oder setzen von Parametern unterschieden. Das haptische Eingabegerät wird mit einer hohen Rate abgetastet und auch selbst geregelt um die Reaktionszeiten in der Telepräsenz gering zu halten. Die Mirs-Anwendung wird dagegen über eine RPC Verbindung mit Daten aus dem Simulink-Modell zyklisch aktualisiert und kann auf Benutzerwunsch Befehle an das Modell senden um z.B. einen Regler umzuschalten.

Die im gesamten System verwendeten Kommunikationsarten können in drei Kategorien aufgeteilt werden:

- **Signal** - Harte Echtzeit ($>1\text{kHz}$)
Diskretes Signal für die Regelung. Wird zwischen den Modellen über aRDnet versendet.
- **Parameter** – Zustandsänderungswunsch ($<1\text{Hz}$)
Sporadisch durch den Benutzer ausgelöste Parameteränderung im Modell die sicher ankommen und ausgeführt werden muss. Zum Beispiel die Anbindung einer GUI an ein Simulink Modell.
- **Zustände** - Zyklische Daten ($1\text{-}50\text{Hz}$)
Daten die zyklisch aktualisiert werden. Zum Beispiel Systemzustände die in einem GUI angezeigt werden. Diese Daten können in beide Richtungen gesendet werden. Also vom Modell zum GUI und umgekehrt.

Die Infrastruktur der Echtzeitsignale soll in dieser Arbeit nicht verändert werden, daher werden die Benutzerschnittstellen die direkt mit Signalen mit einem Regler kommunizieren nicht weiter beachtet. Mit den verbleibenden, nicht echtzeitfähigen, Benutzerinterfaces werden Zustandsänderungswünsche und zyklische Daten versendet und empfangen. Daher sollten für diese Datenübertragungen keine Signal-Verbindungen genutzt werden die nicht die benötigte Übertragungssicherheit bieten, wie es Beispielsweise bei den iPhone Anwendungen der Fall ist. Bei Benutzerschnittstellen geht es nicht um Übertragungen in einer möglichst kurzen Zeit sondern vielmehr um eine sichere Übertragung und das sichere setzen von Werten. Alle untersuchten Benutzerschnittstellen sind zur Übersicht in Tabelle 2.1.

Benutzerschnittstelle	Sendet	Empfängt	Anbindung
Sigma7	Signale	Signale	USB
Mirs-GUI	Parameter	Zustände	SUN-RPC
TMCD	Parameter und Zustände	Zustände	Seriell
Geoserver	-	Zustände	aRDnet
iPhone	Parameter	Zustände	aRDnet

Tabelle 2.1: Übersicht der Benutzerschnittstellen mit der verwendeten Kommunikationsart.

Anwendungen sollten mehrfach und auch gleichzeitig auf dem selben oder verschiedenen Rechnern verwendbar sein. Die Benutzerschnittstellen müssen hierfür synchronisiert werden damit sich alle Anwendungen gleich verhalten und ein konsistenter Systemzustand erreicht wird. Auch lässt sich teilweise bei den Benutzerschnittstellen eine Redundanz erkennen. Zum Beispiel können mit dem TMCD und dem MIRS-GUI die Regler der MIRO-Roboter umgeschaltet und die Roboter Hardware gestartet und gestoppt werden. Dies kann später durch die Verwendung von universellen Schnittstellen welche mehrere Client-Anwendungen gleichzeitig bedienen können, ausgenutzt werden.

Die Kommunikationsblöcke für die angebundenen Benutzerschnittstellen liegen auf Grund ihrer Größe und Inflexibilität zentral im Weltmodell. Dies bedeutet, dass Daten für die Benutzerschnittstellenkommunikation aus anderen Modellteilen bis zum Weltmodell weitergereicht werden müssen. Folglich steigern Kommunikationsschnittstellen, welche direkt in die Modelle integrierbar sind, die Übersichtlichkeit und Wartbarkeit des Gesamtsystems.

Darüber hinaus gibt es Systemparameter in den Modellen für die es keine Anbindung an Benutzerschnittstellen gibt, die aber dennoch gelegentlich zur Systemoptimierung verändert werden müssen. Besteht die Notwendigkeit nicht angebundene Modellparameter zu ändern, wird oft der External-Mode von Simulink verwendet. Dieser Simulink Modus wird bei vielen Einstellungen verwendet weil der Aufwand eine netzwerkfähige Schnittstelle in das Modell zu integrieren mit den gegebenen Mitteln sehr hoch ist.

Zudem wird der Aufbau des gesamten Systems über verschiedene Shell Scripte konfiguriert. Beim Systemstart wird eine starre Netzwerkkonfiguration der Komponenten erzeugt. Falls eine Komponente abstürzt ist es unter Umständen nötig das gesamte System neu zu starten.

In der folgenden Übersicht sind noch einmal abschließend die ermittelten Kriterien für die Anbindung der Benutzerschnittstellen zusammengefasst:

- Einfache Integrierbarkeit an die verteilten Simulink-Modelle, ohne dabei die bestehende Architektur des Gesamtsystems zu verändern
- Sichere Übertragung von Parametern in das Modell
- Übertragung von zyklischen Zustandsdaten
- Anbindung an Linux und QNX
- Dynamischer Verbindungsaufbau
- Synchronisation von redundanten Benutzerschnittstellen
- Einfache Erzeugung neuer Schnittstellen

Die hier aufgeführten Probleme werden in den folgenden Teilen dieser Arbeit mit der Auswahl einer geeigneten Middleware und der Entwicklung eines Konzepts gelöst.

3 Auswahl der Middleware

Im folgenden Teil der Arbeit wird eine geeignete Middleware-Lösung für die Forschungsplattform des MiroSurge-Projekts ausgewählt. Zu Beginn dieses Kapitels werden dazu einige Grundlagen über Middleware erläutert um den Einstieg zu erleichtern. Dabei werden die Unterschiede von verschiedenen Ansätzen für Middleware-Lösungen aufgezeigt. Anschließend werden die Auswahlkriterien für eine passende Middleware aufgestellt und existierende Lösungen untersucht.

3.1 Grundlagen zu Middleware

Der Begriff Middleware beschreibt eine Softwaretechnologie, die entworfen wurde um das Verwalten der Komplexität und Verschiedenartigkeit in verteilten Systemen zu vereinfachen. Sie ist eine Zwischenschicht, die sich zwischen der Anwendung und dem Betriebssystem befindet (siehe Abbildung 3.1). Im OSI-Schichtenmodell befindet sie sich auf der Anwendungsschicht (Schicht 7). Die Middleware besteht aus einer Sammlung von Diensten, die es Prozessen erlaubt miteinander zu kommunizieren. Die Prozesse können hierbei nicht nur innerhalb eines Computers kommunizieren, sondern auch über Rechnergrenzen und verschiedene Plattformen hinweg. Mit dem Einsatz einer Middleware sollen einheitliche Schnittstellen für die Anwendungskommunikation entstehen. Die Komplexität für verschiedene Kommunikationsarten wird in der Middleware selbst verborgen. [16] [17]

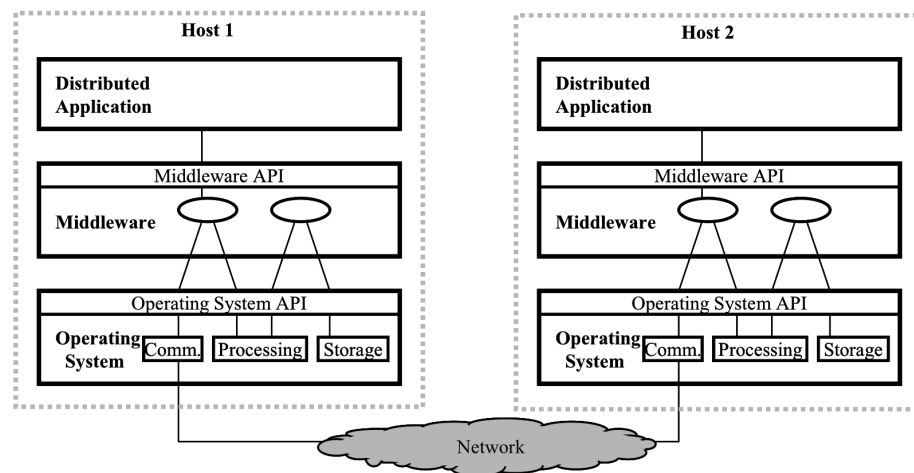


Abbildung 3.1: Middleware im Zusammenhang eines verteilten Systems. [16]

Der Begriff Middleware steht mehr für ein Abstraktionskonzept als für eine definierte Technologie. In der Literatur gibt es verschiedene Kategorisierungen [17] [18].

Die verschiedenen Ansätze lassen sich nach den verwendeten Kommunikationsarten (synchron oder asynchron) und, nach ihren zu Grunde liegenden Paradigmen ordnen:

- **Funktionsaufruf-orientiert** (Remote Procedure Call (RPC)):

Die Funktionsaufruf-orientierte Middleware ermöglicht den Aufruf von Funktionen, die sich auf einem entfernten System befinden als ob sie lokal vorhanden seien. Es wird meistens eine synchrone Kommunikation genutzt. Eine asynchrone Kommunikation ist möglich.

- **Nachrichten-orientiert** (Message Oriented Middleware (MOM)):

Die Nachrichten-orientierte Middleware basiert auf dem Austausch von Nachrichten zwischen lose gekoppelten Kommunikationsendpunkten nach dem Publish-Subscribe-Modell. In diesem Modell werden Nachrichten von Sendern (Publisher) veröffentlicht und von Empfängern (Subscriber) abonniert. Ein Broker (Vermittler) übernimmt die Nachrichtenverteilung an die Subscriber. Die Nachrichten werden asynchron von den Publishern veröffentlicht. Die Empfänger und deren Anzahl sind dabei nicht bekannt. Ein oder mehrere Subscriber können die veröffentlichten Nachrichten abonnieren (1-zu-n-Kommunkation). Die Filterung der Nachrichten beim Subscriber kann Topic (Thema) basiert oder Content (Inhalt) basiert geschehen.

Neben der Publish-Subscribe-Kommunikation ist es mit manchen Nachrichten-basierten Middleware-Lösungen ebenfalls möglich, synchrone RPC-Aufrufe nach dem Request-Reply-Modell auszuführen.

- **Objektorientiert** (OOM):

Die objektorientierte Middleware entstand aus den RPCs und erweitert dieses System um objektorientierte Konzepte. Sie stellt eine Abstraktion von einem Objekt, das sich in einem anderen Prozess und auf einem anderen System befinden kann, bereit. Bei der Programmierung kann dies wie ein lokales Objekt behandelt werden mit samt Methodenaufrufen, Verkapselung, Vererbung und Polymorphie. Diese Art der Middleware erlaubt eine transparente Nutzung von verteilten Objekten, auf die synchron oder asynchron zugegriffen werden kann.

Als bekannte Lösungen sind hier die von der Object Management Group (OMG) [19] entwickelte Spezifikation CORBA (Common Object Request Broker Architecture) [20], die Remote Method Invocations (RMI) von Java und das Component Object Model (COM) von Mircrosoft zu nennen.

- **Anwendungsorientiert:**

Diese Kategorie geht über die objektorientierten Konzepte hinaus und hat vor allem die Unterstützung verteilter Anwendungen als Ziel. Komponenten werden hierbei Modelle und Dienste zur Verfügung gestellt. Beispiele sind J2EE (Java Enterprise Edition Plattform) [21] und das .Net-Framework von Mircrosoft [22].

3.2 Auswahlkriterien für eine Middleware-Lösung

Aus der Problemanalyse der Arbeit lassen sich nun die Kriterien für die Auswahl einer geeigneten Middleware-Lösung zusammenstellen.

Es wird eine Middleware-Lösung gesucht die keine Umkonfiguration der bestehenden Architektur der verteilten Regelungsanwendung erfordert.

Zudem ging aus der Analyse der aktuellen Benutzerschnittstellen hervor, dass für die Anbindung der Benutzerschnittstellen Zustandsänderungswünsche und zyklische Daten benötigt werden. Die Zustandsänderungswünsche müssen sicher in das Regelungssystem übertragen werden. Die zyklischen Daten sollen asynchron an die Client-Anwendungen gesendet werden. Zudem sollen mehrere Client-Anwendungen gleichzeitig auf die Schnittstellen zugreifen können um die Redundanz in den Anwendungen auszunutzen und somit Schnittstellen einzusparen.

Hierfür scheint eine Nachrichten-basierte Middleware mit Request-Reply- und Publish-Subscribe-Modell geeignet. Das Request-Reply-Modell erlaubt eine sichere Übertragung von Parametern während das Publish-Subscribe-Modell zeitgleich das asynchrone Versenden von zyklischen Zuständen an mehrere Client-Anwendungen gestattet.

Des Weiteren ist eine IDL (Interface Description Language) zur Beschreibung der Schnittstellen wünschenswert. Die Schnittstellen können somit an einer Stelle definiert und in verschiedenen Anwendungen verwendet werden. Änderungen der Schnittstellendefinition können direkt in die Anwendungen übertragen werden.

Ein Naming-Service ermöglicht, mit der Verwaltung von Port-Nummern und IP-Adressen, einen dynamischen Verbindungsaufbau. Hierdurch wird der Austausch einzelner Komponenten zur Laufzeit vereinfacht. Die Robustheit und Flexibilität des Gesamtsystems werden damit maßgeblich gesteigert.

Ein weiterer wichtiger Aspekt ist ein C/C++-Interface damit eine Anbindung an Simulink mit den S-Functions möglich ist. Zudem wird eine Middleware benötigt, die aus einem kleinen schlanken Grundkern besteht und unter Linux und QNX funktionsfähig ist oder sich zumindest leicht portieren lässt. Auch sollte die Lösung keine hohe Komplexität aufweisen um den Einarbeitungsaufwand möglichst gering zu halten.

Optional sind Debuggingtools zur Überwachung des Gesamtsystems und die Anbindung an verschiedene Programmiersprachen und Plattformen um flexible Client-Anwendungen zu entwickeln. Außerdem sollte der Quellcode des Projekts zur Verfügung stehen und das Projekt aktiv weiterentwickelt werden.

Zur Übersichtlichkeit werden alle Auswahlkriterien noch einmal in der folgenden Tabelle zusammengefasst:

Kriterium	Beschreibung
Request-Reply	Sichere Übertragung von Parametern
Publish-Subscribe	Asynchrones Versenden von zyklischen Zuständen
IDL	Zentrale Definition der Schnittstellenbeschreibungen
Naming-Service	Verwaltung von Port-Nummern und IP-Adressen
C/C++-Schnittstelle	Anbindung an die Simulink S-Functions und C++ Anwendungen
Linux und QNX Kompatibilität	Einsatz auf den Linux und QNX Echtzeitsystemen für den Simulink Simulationsmodus, die Echtzeitanwendung und die Client-Anwendungen
Komplexität	Komplexität des Grundkerns
Dokumentation	Dokumentation des Systems und nützliche Beispielanwendungen
Weiterentwicklung	Aktive Weiterentwicklung des Systems und Fehlerbehebung
Tools	Optionale Tools zum Debuggen des Gesamtsystems
Client Schnittstellen	Schnittstellen zu anderen Programmiersprachen wie z.B. Python oder Java für Android
Integrierbarkeit	Integrierbarkeit in die vorhandene Architektur der Simulink Regelungsanwendung

Tabelle 3.1: Übersicht der Auswahlkriterien für eine Middleware-Lösung.

3.3 Existierende Middleware-Lösungen

Nach den zuvor aufgestellten Kriterien soll nun eine Middleware-Lösung ausgewählt werden. Es wurden die erfolgversprechendsten Lösungen mit objektorientierten, Funktionsaufruf-orientierten und Nachrichten-orientierten Hintergrund untersucht. Darüber hinaus werden Middleware-Lösungen mit speziellem Fokus auf die Robotik analysiert. Anwendungsorientierte Lösungen wie J2EE oder das .NET Framework werden nicht weiter untersucht. Diese Lösungen sind für die Zielanwendung zu umfangreich und bieten keine C/C++ Schnittstelle.

3.3.1 Objektorientierte Middleware-Lösungen

Unter den freien CORBA Implementierungen haben sich ACE/TAO [23] und omniORB [24] am meisten verbreitet. Wobei ACE/TAO die vollständigere Implementierung ist, omniORB aber sehr viel schlanker. CORBA bietet einen Naming-Service und eine IDL zum spezifizieren der Schnittstellen. Auch bietet CORBA zwar mit der Erweiterung des CORBA Components Model (CCM) durch das Event Service basierte Push-Pull-Modell eine Art Publish-Subscribe-Kommunikation. Allerdings ist CORBA ein komplexer Standard, der eine lange Einarbeitungszeit benötigt.

Neben CORBA gibt es mit ICE (Internet Communications Engine) [25] noch eine weitere verbreitete objektorientierte Middleware. ICE wird von der Firma ZeroC als Open-Source-Projekt entwickelt. Sie ist ebenso umfangreich wie CORBA, allerdings schlanker und übersichtlicher gestaltet. ICE ist auf vielen Programmiersprachen und Plattformen einsetzbar. Ähnlich wie CORBA, bietet auch ICE eine IDL die Slice genannt wird, mit der die

Schnittstellen definiert werden. ICE liefert eine Vielzahl an Tools und Diensten, um verteilte Systeme aufzubauen und zu warten. Dazu gehören ein Naming Service, Tools zum Verwalten von Anwendungen, Anwendungsserver und ein Dienst zum Verteilen von Objekten nach dem Publish-Subscribe-Modell. ICE basiert, im Gegensatz zu CORBA, nicht auf einem großen schwerfälligen Standard.

ICE und die CORBA Implementierungen bieten eine IDL, Request-Reply- und Publish-Subscribe-Kommunikation. Allerdings sind diese Frameworks im Vergleich zu anderen Lösungen sehr umfangreich und schlecht in die bestehende Architektur zu integrieren.

3.3.2 RPC-Lösungen

Unter den RPC-Lösungen ist das bereits eingesetzte ONC RPC (Open Network Computing Remote Procedure Call) [26], das oft auch Sun RPC genannt wird, eine verbreitete Variante. SUN RPC bietet eine IDL, mit der sich die Schnittstellen beschreiben lassen, und einen dazugehörigen IDL Compiler RPCGEN. Mit dem RPCGEN werden die Server und Client Stubs³ in C generiert, die zuvor in den IDL Dateien definiert wurden. Die generierten Stubs werden in das Client und Server Programm eingebunden und können ähnlich wie Funktionsaufrufe verwendet werden.

Weiterhin gibt es XML-RPC [27] und SOAP (ursprünglich Simple Object Access Protocol) [28] unter den bekannten RPC Lösungen. Beide Systeme setzen auf eine Kombination aus XML (Extensible Markup Language) und HTTP (Hypertext Transfer Protocol).

Ein Nachteil der RPC-Lösungen ist das Fehlen eines Naming-Services für die Abstraktion der IP-Adressen und Ports. SUN-RPC und XML-RPC sind hauptsächlich für einfache Client Server Kommunikationen gedacht und somit für die geplante Anwendung nicht flexibel genug.

3.3.3 Messaging Bibliotheken

Die beiden Messaging Bibliotheken YAMI4 [29] und zeroMQ [30] sind Netzbibliothek mit wenigen Abhängigkeiten zu anderen Bibliotheken. Mit ihnen ist es möglich, hochperformante verteilte Systeme zu erstellen. Neben einfachen asynchronen Verbindungen lassen sich mit ihnen auch Kommunikationen nach den Request-Reply- und Publish-Subscribe-Modellen aufbauen. Zudem wird ein einfacher Nameing-Service angeboten.

Anders als die RPC- und objektorientierten Lösungen bieten sie keine eigene IDL. Die Daten werden über Set- und Get-Schnittstellen in die Nachrichtenpakete geschrieben bzw. gelesen und dann versendet. Dies erschwert die Erstellung der Schnittstellen und die Konsistenzerhaltung zwischen den Anwendungen.

³ Stubs (Stummel) sind die Interfaces, die dem Anwender für den Aufruf der entfernten Methoden oder Funktionen zur Verfügung gestellt werden.

3.3.4 Middleware für die Robotik

Ein großes Problem bei der Standardisierung einer einheitlichen Middleware-Lösung für die Robotik sind die großen Unterschiede zwischen den verschiedenen Robotern. In robotischen Systemen wird die verschiedenste Hardware eingesetzt. Angefangen bei 8Bit Mikrocontrollern bis hin zu mittlerweile mehreren leistungsfähigen Workstations, wie zum Beispiel beim PR2 [31]. Zudem ist es schwierig, ein geeignetes Abstraktionsmodell für die Vielfalt an Sensoren und Aktoren zu entwickeln. Eine Übersicht an allgemeinen Problemen, die bewältigt werden müssen, um eine Middleware-Lösung für die Robotik zu erstellen, wird in den Arbeiten von [32] und [33] gegeben.

In den folgenden Teilen dieser Arbeit werden die vielversprechendsten Middleware-Lösungen mit speziellem Bezug auf Anwendungen in der Robotik vorgestellt, welche weiter untersucht worden sind.

3.3.4.1 aRDnet

aRDnet ist die aktuell verwendete Lösung für die Echtzeitkommunikation zwischen den Simulink-Modellen. Teilweise wird diese Lösung auch für die Anbindung von Benutzerschnittstellen verwendet. Allerdings wird für die Datenübertragung eine UDP-Verbindung verwendet, bei der nicht sichergestellt werden kann, ob die Daten sicher übertragen werden. Zudem bietet aRDnet keinen Naming-Service und erfüllt damit nicht die geforderten Kriterien.

3.3.4.2 FAMOUSO

FAMOUSO (Family of Adaptive Middleware for autonomOUS Sentient Objects) [34] ist eine Middleware dessen Ziel es ist, möglichst ressourcensparend zu sein. Dies ermöglicht den Einsatz auf einer breiten Auswahl an Plattformen. Angefangen bei 8Bit Mikrocontrollern bis hin zu x86 Workstations. Für diesen Zweck ist ein eigenes Protokoll zur Übertragung entwickelt worden. Die Hauptfunktionalität ist in C++ geschrieben. Darüber hinaus existieren Schnittstellen zu Python, Java, Simulink und Labview.

Die Kommunikation der Komponenten kann über verschiedene Medien stattfinden. Dazu gehören der CAN-Bus (Controller Area Network), Ethernet und kabellose Verbindungen. Es wird kein Naming-Service Angeboten und für die Datenübertragung mit Ethernet wird ein UDP-Multicast eingesetzt. Es wird nur die asynchrone Übertragung nach dem Publish-Subscribe-Modell unterstützt, wodurch nicht sichergestellt werden kann, ob die Daten fehlerfrei übertragen werden. [35]

3.3.4.3 OpenRTM-aist

Mit RT-Middleware (Robotics Technology Middleware) [36] wird das Ziel verfolgt, eine Middleware zu spezifizieren, die das Erstellen von Robotiksystemen mit Netzwerkfunktionalität vereinfacht. In diesem System sollen Komponenten nach ihren

Funktionen in sogenannte RT-Components (RTC) [37] aufgeteilt werden. Wodurch Module entstehen, die sich zu komplexen Robotik-Systemen verbinden lassen.

Die Idee dazu ist am japanischen National Institute of Advanced Industrial Science and Technology (AIST) entstanden. Mittlerweile wird mit Hilfe der Object Management Group ein Standard für die RT-Components entwickelt. Eine erste RTC Spezifikation [38] ist bereits erschienen. Die Spezifikation ist, wie CORBA, nicht an eine bestimmte Plattform oder Programmiersprache gebunden. Dazu ist am AIST eine Referenzimplementierung OpenRTMAIST [39] entwickelt worden, die auf der CORBA Implementierung omniORB [24] aufbaut und Schnittstellen für C++, Java und Python [40] bietet. Das System ist durch das strikte Komponentenmodell nicht flexibel genug. Darüber hinaus stellt CORBA als Kommunikationsframework eine komplexe Abhängigkeit dar.

3.3.4.4 Orca

Orca [41] ist eine komponentenbasierte Middleware Lösung für die Anwendung in der Robotik. Es nutzt als einzige bekannte Lösung ICE für die Kommunikationsinfrastruktur und verwendet die meisten Dienste und Tools, die in ICE enthalten sind. Zudem bietet es Tools für die Visualisierung, Logging und zum Debuggen. Orca wird allerdings nicht mehr aktiv weiterentwickelt. [42]

3.3.4.5 Orocos

Orocos (Open Robot Control Software) [43] ist ein freies modulares Framework für die Erstellung von Roboter- und Regelungsanwendungen. Das Framework besteht aus vier C++ Bibliotheken: Real-Time Toolkit, Kinematics, Dynamics und Bayesian Filtering. Wobei das Real-Time Toolkit (RTT) ein Komponentenmodell zum Erstellen der Applikation bereit stellt. Für die Netzwerkkommunikation in RTT kann zwischen den beiden CORBA Implementierungen ACE/TAO und omniORB gewählt werden.

OROCOS ist ein System für verteilte Regelungsanwendungen und gibt ein Komponentenmodell für die einzelnen Regler vor. Der Ansatz würde eine Umkonfiguration der Regelungsanwendungen erfordern, was vermieden werden soll. Zudem stellt das verwendete Kommunikationsframework eine komplexe Abhängigkeit dar.

3.3.4.6 ROS

ROS (Robot Operating System) [44] ist eine Art Meta-Betriebssystem für Roboter. Die Entwicklung begann 2007 am Stanford Artificial Intelligence Laboratory und wird seit 2008 hauptsächlich von der Firma Willowgarage als Open-Source-Projekt weitergeführt.

Der Fokus dieses Systems liegt bei der Wiederverwendbarkeit von lose gekoppelten Komponenten, Nodes genannt. Wobei es in ROS kein striktes Komponentenmodell mit Abstraktionen wie in anderen Frameworks gibt und ein Node einem Prozess entspricht. Die Nachrichten für die Kommunikation werden in einer IDL erstellt, mit C ähnlicher Syntax.

Namen von Nodes können zum Programmstart modifiziert werden, wodurch das System sehr flexibel ist. Es kann zum Beispiel leicht zwischen einem Publisher und einem Subscriber ein Programm eingefügt werden. Dadurch können Daten, die auf dem Topic zwischen den beiden Kommunikationspartnern versendet werden, bearbeitet oder gefiltert werden, ohne die Implementierung in den Endpunkten ändern zu müssen.

Der Kern des Kommunikationsframeworks ist eine in C++ geschriebene Bibliothek, zu der es auch Schnittstellen zu Python gibt. Neben den Funktionen von Nachrichten basierten Middleware-Lösungen, bietet ROS zusätzlich für Middleware unübliche Funktionen wie eine Paketverwaltung, Debugging-, Logging- und Visualisierungstools. ROS verwendet für die Koordination der einzelnen Klienten und die Namensauflösung einen zentralen ROS-Master. Die Datenübertragung selbst geschieht aber nach einem Peer-to-Peer System mit einem eigenen Protokoll, das TCP/IP oder UDP Verbindungen verwendet. Allerdings unterstützt ROS die QNX Plattform nicht. [45]

Inzwischen wird es von sehr vielen Forschergruppen verwendet und entwickelt sich sehr schnell weiter. Durch das enthaltende Paketsystem wird der Austausch von Komponenten zwischen den Entwicklern vereinfacht und es sind mittlerweile über 1000 Pakete für verschiedenste Roboterkomponenten entstanden [46]. Neben vielen Komponenten-Treibern sind zudem einige Umsetzungen zwischen ROS und anderen Frameworks entwickelt worden.

3.4 Diskussion

ICE und die CORBA Implementierungen ACE/TAO und omniORB sind generische Middleware-Lösungen mit denen der Aufbau von Kommunikationsinfrastrukturen vereinfacht wird. Wobei ICE die schlankere und übersichtlichere Lösung ist. Allerdings sind alle drei Lösungen schlecht in die bestehende Architektur integrierbar.

Die RPC-Lösungen und Messaging-Bibliotheken sind zwar sehr schlank und einfach zu handhaben, bieten aber keine IDL, mit der Schnittstellen definiert werden können.

FAMOUSO bietet keine Request-Reply-Kommunikation und Orca wird nicht mehr aktiv weiterentwickelt.

Mit OpenRTM-aist existiert eine Lösung, die den RTC Standard implementiert. Wie viele andere Lösungen, setzt auch OpenRTM-aist CORBA als Middleware ein und gibt ein Komponentenmodell vor.

Das Orocos Framework ist für die Erstellung von modularen verteilten Reglerkomponenten in einer Echtzeitumgebung ausgelegt. In Orocos wird ein Regler mit samt Parametern als eine Komponente betrachtet. Der Einsatz dieses Frameworks hat eine Umstrukturierung der Architektur der Regelungsanwendung zur Folge und erfüllt damit nicht geforderten Kriterien.

In dieser Arbeit wurde sich für ROS als Middleware entschieden. ROS erfüllt die meisten Voraussetzungen und bietet damit für viele Probleme die beste Lösungsmöglichkeit. Probleme, die nicht mit der Middleware gelöst werden, werden im Konzept behandelt. Erste Tests von ROS wurden unter Linux durchgeführt. Dabei wurden die für die Kommunikation zuständigen Bibliotheken untersucht und in einem nächsten Schritt auf QNX 6.5 kompiliert.

Eine genauere Beschreibung der Portierung der ROS-Bibliotheken auf QNX 6.5 wird in Kapitel 5.4 beschrieben.

Die Ergebnisse der Untersuchungen werden noch einmal in Tabelle 3.2 zusammengefasst. Das Auswahlkriterium C/C++-Schnittstelle wird nicht mehr extra aufgeführt. Es wurden nur Lösungen untersucht, die diese Voraussetzung erfüllen.

Lösung	Request-Reply	Publish-Subscribe	IDL	Naming-Service	Linux und QNX Kompatibilität	Komplexität	Dokumentation	Weiterentwicklung	Tools	Client Schnittstellen	Integrierbarkeit	Summe
OmniORB	+	+	+	+	0	-	-	-	-	+	-	0
ACE/TAO	+	+	+	+	0	-	-	-	-	-	-	-2
ICE	+	+	+	+	0	-	0	+	+	+	-	5
SUN-RPC	+	-	+	-	+	+	0	-	-	-	+	0
XML-RPC	+	-	-	-	+	+	+	-	-	+	+	1
YAMI4	+	+	-	+	+	+	+	+	-	+	+	7
zeroMQ	+	+	-	+	+	+	+	+	-	+	+	7
aRDnet	-	+	-	-	+	+	0	-	-	0	+	-1
FAMOUSO	-	+	-	-	-	+	-	-	-	-	-	-7
OpenRTM-aist	+	+	+	0	-	-	0	-	-	+	-	-1
Orca	+	+	+	+	+	-	-	-	-	-	-	-1
Orocos	+	0	+	+	+	-	-	+	+	-	-	2
ROS	+	+	+	+	0	0	+	+	+	+	+	9

Tabelle 3.2: Bewertung der untersuchten Middleware-Lösungen nach den gewählten Auswahlkriterien. Die Summe ergibt sich aus den positiven (+), neutralen (0) und negativen Eigenschaften (-) der Middleware-Lösungen.

4 Konzept für die Anbindung von Nutzerschnittstellen

In diesem Kapitel wird nun ein Konzept einer verteilten Architektur für die Anbindung von Benutzerschnittstellen an eine verteilte Regelungsanwendung vorgestellt. Das Konzept wird generisch entwickelt und kann somit auf die in Kapitel 1 vorgestellte Architektur übertragen werden. Abschließend wird das Konzept konkret auf einen Teil des Gesamtsystems angewendet.

4.1 Konzept

Mit den aus Kapitel 2 gewonnenen Erkenntnissen über die aktuell vorhandenen Probleme in der Architektur und der in Kapitel 3 ausgewählten Middleware, wird nun ein Lösungskonzept vorgestellt. Damit soll die Anbindung von redundanten Benutzerschnittstellen an die Simulink-Modelle vereinfacht werden, ohne die vorhandene Architektur des Gesamtsystems ändern zu müssen. Mit der in ROS enthaltenen Nachrichten-basierten Middleware, stehen hierfür zwei Verbindungsmodelle zur Verfügung: Zum einen das Publish-Subscribe-Modell für asynchron übertragene, zyklische Daten und zum anderen das Request-Reply-Modell für synchrone Funktionsaufrufe.

Bei der Anbindung verschiedener Benutzerschnittstellen an ein verteiltes System verschiedene Dinge zu beachten: Zunächst müssen die Parameter und Befehle des Benutzers in das System übertragen werden. Anschließend muss die Antwort des Systems zurück an das Benutzerinterface übertragen werden, damit der Nutzer die Reaktion sieht. Die Antwort des Systems beinhaltet also den aktuellen Zustand des Systems. Der Wunsch, einen Systemzustand zu ändern, kann unter Umständen gar nicht oder nicht sofort ausgeführt werden. Ebenso kann sich der Zustand unter Umständen, zum Beispiel durch das Auftreten von Fehlern, von alleine ändern. Um diese Verbindung zwischen Benutzerinterface und Systemzustand konsistent zu halten, gibt es verschiedene Mechanismen, auf die erst später eingegangen wird.

Verschärft wird diese Problematik, wenn mehrere Instanzen der gleichen Benutzerschnittstelle gestartet werden. Allerdings sollte in einem verteilten System auch möglich sein, diese Instanzen auf verschiedenen getrennten Systemen zu starten. In diesem Fall müssen die Systemzustände mit mehreren Programmen synchronisiert werden. Zudem muss das System richtig reagieren, falls Befehle von verschiedenen Benutzern gleichzeitig oder kurz aufeinander abgesendet werden.

4.1.1 Übertragung von Parametern und Befehlen an ein Programm

Für die Übertragung von Parametern und Befehlen an ein Programm gibt es mit der gewählten Nachrichten-orientierten Netzwerkkommunikation verschiedene Lösungsansätze. Eine einfache Möglichkeit ist es, das Request-Reply-Modell zu verwenden (siehe Abbildung 4.1). Die Eingaben des Benutzers werden so mit einem RPC-Aufruf an das entfernte

Programm gesendet. Die Antwort wird mit dem Rückgabewert an das aufrufende Programm zurückgeliefert und der Zustand des Systems mit dem Rückgabewert übertragen.

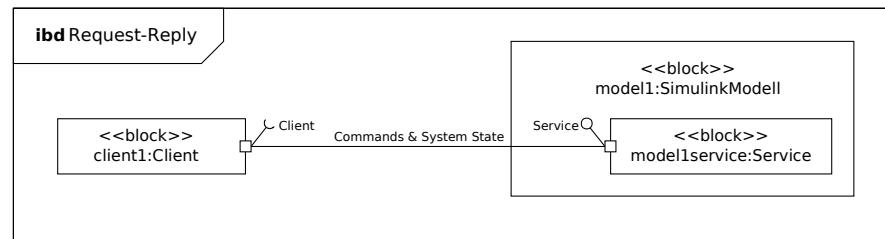


Abbildung 4.1: Kommunikation auf RPC Basis.

Dieser Mechanismus reicht aber nicht aus um den Zustand des Systems mit der Benutzerschnittstelle konsistent zu halten. Eine Lösung dafür ist das Pollen des Zustands durch das Userinterface mit RPC-Aufrufen. Dies bringt aber keinen Vorteil gegenüber der aktuell verwendeten Architektur.

Die Parameter und Befehle des Benutzers könnten auch mit dem Publish-Subscribe-Modell in das System übertragen werden (siehe Abbildung 4.2). Dazu wird jeweils ein Publisher und ein Subscriber im Benutzerprogramm und im System integriert. Daten werden vom Publisher des Benutzerprogramms an den Subscriber des Systems gesendet. Dort werden die Parameter gesetzt und der aktuelle Zustand des Systems vom dem Publisher des Systems zurück an das Userinterface gesendet.

Durch diesen Lösungsansatz wird zwar die Übertragung des Systemzustands eleganter gelöst, es treten allerdings andere Probleme auf. Zum einem ist die Publish-Subscribe-Übertragung lose gekoppelt, was bedeutet, dass der Benutzer keine Information erhält ob und wann sein Befehl am Empfänger angekommen ist. Außerdem veröffentlicht ein Publisher die Daten unabhängig davon, wie viele Subscriber auf seinen Topic angemeldet sind. Ist kein Subscriber angemeldet, verfallen die Nachrichten ohne Rückmeldung an den User.

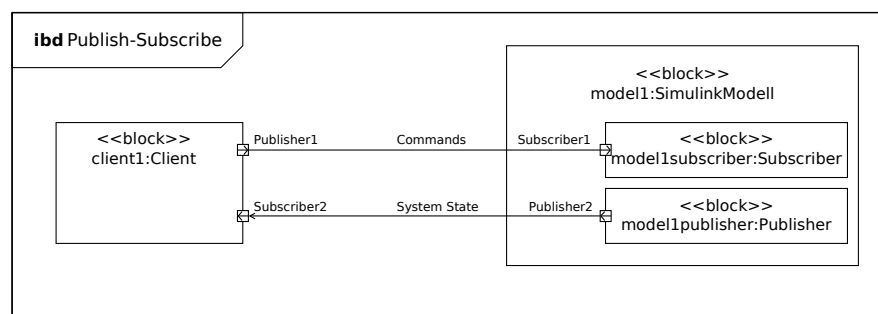


Abbildung 4.2: Kommunikation auf Publish-Subscribe Basis.

Ein dritter Ansatz ist eine Mischung aus dem Publish-Subscribe-Modell und dem Request-Reply-Modell. Parameter und Befehle werden mit einem RPC-Aufruf in das System übertragen (siehe Abbildung 4.3). Die synchrone Verbindung hat eine enge Koppelung und es

wird sichergestellt, dass die Daten im System übermittelt werden. Der Zeitpunkt, an welchem die Übertragung abgeschlossen ist, kann dadurch in Erfahrung gebracht werden.

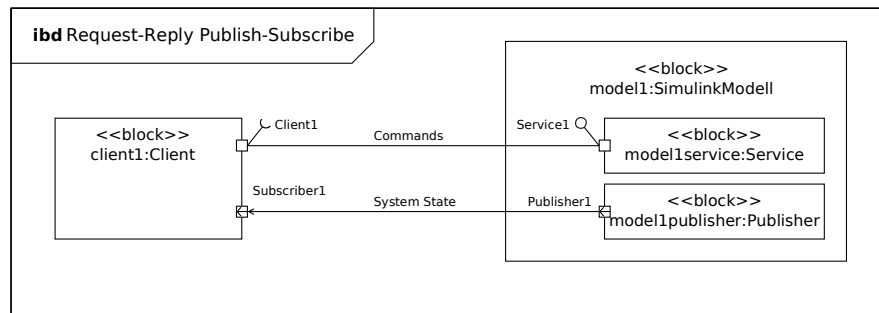


Abbildung 4.3: Kommunikation mit Request-Reply und Publish-Subscribe.

Um den Systemzustand an das Benutzerinterface zu senden, wird im System ein Publisher und im Userinterface ein Subscriber integriert. Damit kann der aktuelle Systemzustand zyklisch im Userinterface aktualisiert werden.

Unter den drei Lösungsansätzen ist der letzte der erfolgversprechendste. Hier werden Daten mit einem RPC-Aufruf sicher in das System übertragen und die Systemzustände regelmäßig an das entfernte Benutzerprogramm gesendet. Allerdings gibt es bei allen Lösungen ein Problem bei der Übertragung von Befehlen:

Beim Senden von Kommandos an die RPC-Schnittstelle können mehrere Daten transferiert werden. Ein Aufruf muss immer alle Daten enthalten. Um also nicht alle Befehle im System bei jedem Aufruf zu überschreiben, muss es einen Mechanismus geben mit dem nur Werte gesetzt werden, die gesetzt werden sollen. Eine Möglichkeit ist das Mitsenden von zusätzlichen Flags damit im Service entschieden werden kann, welche Werte gesetzt werden. Durch dieses System wird aber ein aufbauendes Protokoll eingeführt. Außerdem können die Schnittstellen so klein wie möglich gehalten werden, so dass nur einzelne Daten übertragen werden. Dies erzeugt aber eine große Menge an Schnittstellen und erhöht somit den Implementierungsaufwand.

Eine weitere Lösung ist die Abstimmung der beiden Schnittstellen aufeinander. So kann der vom Clienten gesendete Befehl einen Teil des empfangenden Systemzustands enthalten. Damit werden im System nicht alle Werte überschrieben (siehe Abbildung 4.4).

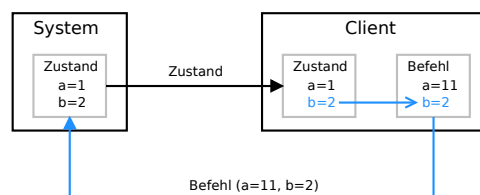


Abbildung 4.4: Mitsenden von einem Teil des letzten Zustands. Der Wert a wird vom Clienten neu gesetzt, der Wert b wird aus dem zyklisch empfangenden Systemzustand übernommen.

Bei der Verwendung von zwei gleichen Clients kann dieses System allerdings zu Problemen führen. Will der erste Client einen anderen Wert setzen als der zweite, und werden diese Aufrufe nahezu gleichzeitig abgesendet, kann nicht sicher vorherbestimmt werden, welchen Zustand das System nach dem Aufruf aufweist (siehe Abbildung 4.5).

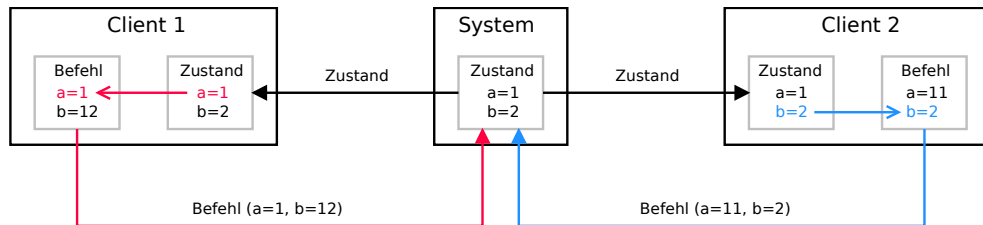


Abbildung 4.5: Problem beim Mitsenden des letzten Zustands mit zwei Clients.

Um dieses Problem zu Lösen, wird den Zustandsdaten, die zyklisch vom System an die Clients übertragen werden, ein Zeitstempel mit der Simulink-Modellzeit angehängt. Dieser Zeitstempel wird von den Clients nicht ausgewertet. Er wird weiter mit der Anfrage an das System gesendet. Der Kommunikationsblock im System bearbeitet die Anfrage nur, wenn der mitgesendete Zeitstempel neuer als der zuletzt bearbeitete Befehl ist. So wird bei zwei gleichzeitig abgesendeten Befehlen von verschiedenen Clients, die den selben Zeitstempel des Systemzustands mitsenden, nur der als erstes eingetroffene bearbeitet. Der zweite wird durch das System abgelehnt und ein Fehlercode mit der Response an den Clients gesendet.

Treffen die Anfragen in der zeitlich richtigen Reihenfolge beim Service ein, kann es trotzdem passieren, dass die Anfragen die mitgesendet werden, auf dem gleichen Zustand basieren und somit eine Inkonsistenz auslösen. Diese Inkonsistenz wird ausgelöst, falls ein Service eine Anfrage annimmt, in das Regelungsmodell weiterleitet, die Response absendet und einen neuen Aufruf annimmt bevor der Publisher den neuen Systemzustand veröffentlichen konnte. Eine Lösungsmöglichkeit die zur Lösung dieses Problems führt, wird erreicht indem der Service die Response erst versendet, wenn das System die Daten verarbeitet hat und der Publisher neue Zustände verteilt hat. Somit wird sichergestellt das Anfragen immer einen gültigen Systemzustand haben. Eine Lösung hierfür wird in Kapitel 4.1.4 durch die Weiterleitung des Request-Reply-Handshakes in das Modell mit dem Enable-Ready-Mechanismus gegeben.

Voraussetzung für diese Funktionen ist, dass die eintreffenden Requests im System nicht gleichzeitig bearbeitet werden. Hierzu könnten die Anfragen nach den Absendezeitpunkten sortiert werden. Dies würde eine Synchronisierung der Systemzeiten zwischen den Rechnern erfordern, welche nicht Teil dieser Arbeit ist. Eine einfachere Lösung ist die Verarbeitung von nur einer Anfrage von dem Service im System. Wenn sich ein Service in einem Aufruf befindet, werden weitere Serviceanfragen mit einem Fehlercode abgelehnt. Für diese Lösung werden die Anfragen nach ihrem Eintreffzeitpunkt bearbeitet, was für eine Anbindung von Benutzerschnittstellen ausreichend ist.

4.1.2 Zwei Programme mit sich überschneidenden Schnittstellen

Neben zwei Instanzen eines Programms soll zudem noch die Möglichkeit bestehen, verschiedene Programme mit überschneidenden Schnittstellen zu erzeugen. Dabei kann ein Programm auf mehr Teile des Systems zugreifen als ein anderes.

Beispiele im MiroSurge-System sind das TMCD und das Mirs-GUI. Mit dem Mirs-GUI kann die Hardware des MIRO-Roboters und des MICA Werkzeugs gesteuert werden und darüber hinaus die Regler des Roboters umgeschaltet werden. Das TMCD Programm hat nur Zugriff auf einen Teil des Systems. Mit diesem kann der Reglermodus gewechselt und der Roboter bewegt werden.

Außerdem sollte auch die Möglichkeit geboten werden, Clienten zu erstellen, die nur Zustandsdaten empfangen. Ein Beispiel wäre ein 3D-Viewer, der die Roboterposen darstellt.

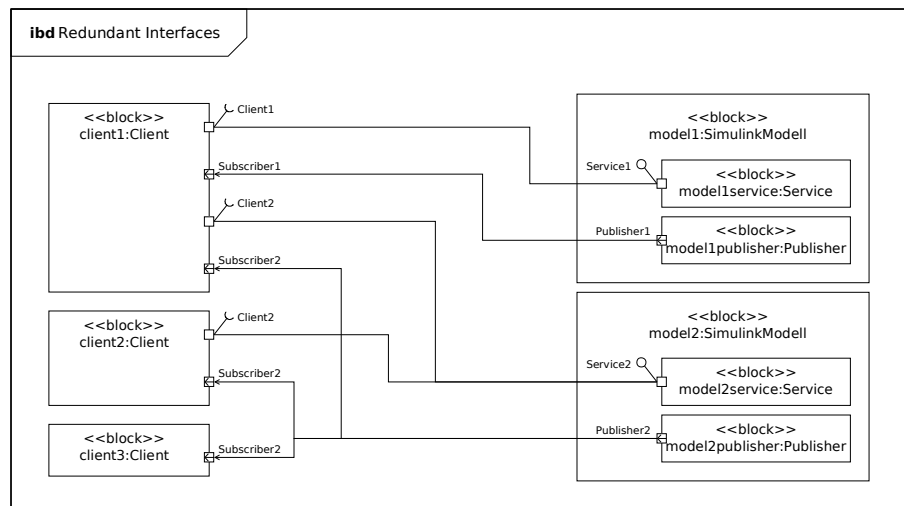


Abbildung 4.6: Ausnutzung der redundanten Schnittstellen.

Ein beispielhafter Aufbau ist in Abbildung 4.6 gezeigt. Hier werden drei Clienten, verwendet die unterschiedlich auf das verteilte Simulink-Modell zugreifen. Der erste Client hat alle Schnittstellen des Systems integriert. Der zweite Client integriert nur die Schnittstellen aus dem zweiten Simulink-Modell, wobei er Kommandos an das System senden kann. Der letzte Client kann nur auf die zyklisch aktualisierten Zustände aus dem zweiten Simulink-Modell zugreifen.

Für eine solche Anwendung sollten möglichst kleine Schnittstellen gewählt werden, um die Flexibilität zu steigern und den Implementierungsaufwand gering zu halten. So können verschiedene Clienten nur solche Teile der Schnittstellen implementieren, die sich mit anderen Clienten überschneiden können.

4.1.3 Exklusiver Zugriff

Manche Anwendungen benötigen Zugriffe auf verschiedene Systemteile. Ein Beispiel ist der Werkzeugwechsel am Roboter. Hier sind zwei Komponenten des MiroSurge-Systems involviert. Der MIRO-Roboter und das MICA-Werkzeug. Die Modelle hierfür werden auf zwei unterschiedlichen Computern berechnet. Zum Wechsel muss der Roboter in einen sicheren Zustand gefahren und gestoppt werden. Auch das Werkzeug muss in die Startposition gefahren und deaktiviert werden. Danach ist der Anwender aufgefordert, am Roboter das Werkzeug zu wechseln. Abschließend muss der Werkzeugwechsel durch den Benutzer bestätigt werden.

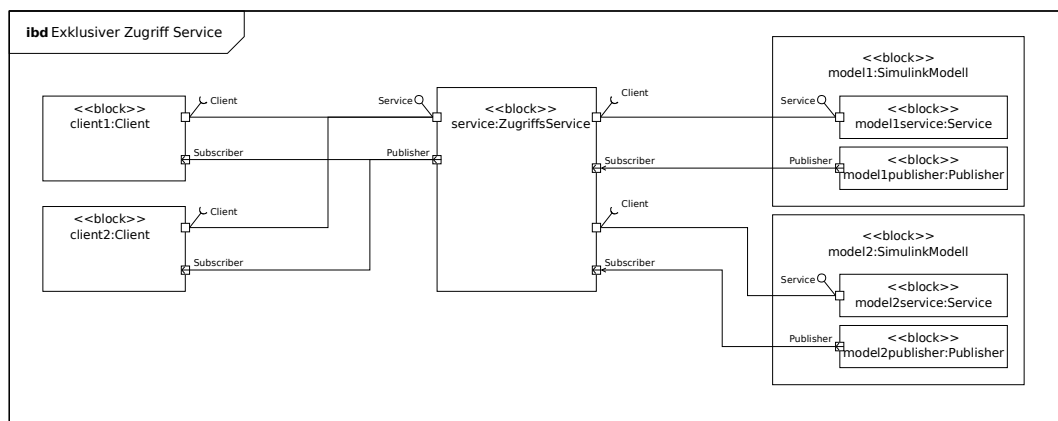


Abbildung 4.7: Modell mit einem Service für die Zugriffsverwaltung.

Ein Ablauf wie der Werkzeugwechsel wirft zwei Probleme auf: Zum einen sollte nur ein Client zur Zeit einen Werkzeugwechsel durchführen können, damit der Prozess nicht durch andere Aufrufe unterbrochen wird. Zum anderen treten bei dem Clienten Aufrufe an verschiedene Teile des Systems auf. Falls der Werkzeugwechsel in verschiedenen Clienten implementiert wird, bedarf es einer Reihe von Aufrufen, die in jedem Clienten enthalten sein müssen. Ein Ansatz ist eine Art Service, der zwischen den Systemteilen und den Clienten sitzt (Abbildung 4.7). Durch das Einführen einer Zwischenschicht kann ein exklusiver Zugriff gesteuert werden, indem der Service nur einen Clienten zur Zeit zulässt. Zudem sind in dem Service die Funktionsaufrufe an die verschiedenen Teile des Systems integriert, wodurch der Client nur noch zu dem Service verbinden muss.

4.1.4 Design der Simulink-Blöcke

Damit die vorgestellten Lösungen in ein Simulink-Modell integriert werden können, müssen drei verschiedene Blöcke entworfen werden. Der wichtigste Block ist der, der die RPC-Aufrufe entgegen nimmt. Dazu kommt ein Block, der Daten aus dem Modell mit einem Publisher versendet. Des Weiteren wird ein Subscriber-Block zum Empfangen von Daten entworfen. Dieser wird benötigt, um zum Beispiel die zyklischen Daten der SpaceMouse in ein Modell zu senden.

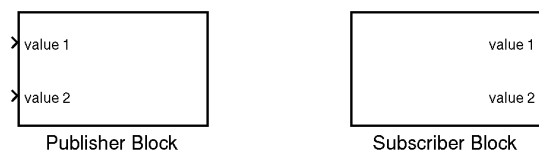


Abbildung 4.8: Publish- und Subscribe-Blöcke für Simulink.

Die Publish- und Subscribe-Blöcke sind einfach gehalten (siehe Abbildung 4.8). Diese Blöcke besitzen keine besondere Protokollimplementierung. Ein Publisher bekommt seine Werte zum Versenden als Eingänge und ein Subscriber besitzt nur Ausgänge, an denen die empfangenen Werte an das Modell übergeben werden.

Bei dem Entwurf des Request-Reply-Blocks gab es die gleiche Idee für das Design. Wenn der entsprechende Block nach dem Berechnungsmodell in Simulink an der Reihe ist, werden die empfangenen Werte in das Modell geschrieben und der RPC-Aufruf beendet. Dabei wird aber der Handshake, der mit dem RPC-Aufruf erreicht werden soll, nicht weiter in das Simulink-Modell geleitet und es können die in Kapitel 4.1.1 beschriebenen Konsistenzprobleme auftreten.

Um den Handshake des RPC-Aufrufs in das Modell zu übertragen und in der Modelllogik zu verarbeiten, wurde der Block erweitert. Aus diesem Grund fand eine Erweiterung des RPC-Blocks, mit den beiden Ports Enable und Ready, statt (siehe Abbildung 4.9).

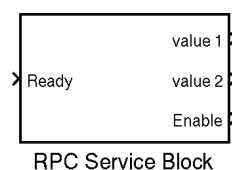


Abbildung 4.9: Request-Reply-Block für Simulink.

Durch diese Erweiterung kann auf einfache Weise von der Modelllogik entschieden werden, wann der RPC-Aufruf ausgeführt worden ist. Dazu wird beim Eintreffen eines Request am Block das Enable-Signal auf Eins gesetzt und die empfangenen Werte auf die Ausgänge des Blocks geschrieben (siehe Abbildung 4.10). Das Enable-Signal signalisiert dem Modell, dass neue gültige Daten vorhanden sind. Für die Rückgabe kann die Modelllogik den Ready-

Eingang des Empfangsblocks auf Eins setzen. Damit wird der RPC-Aufruf beendet und die Response an den Client gesendet, der den Aufruf durchgeführt hat. Zudem wird der Enable Ausgang wieder auf Null gesetzt, um die Ungültigkeit der Ausgangswerte des Blocks zu signalisieren.

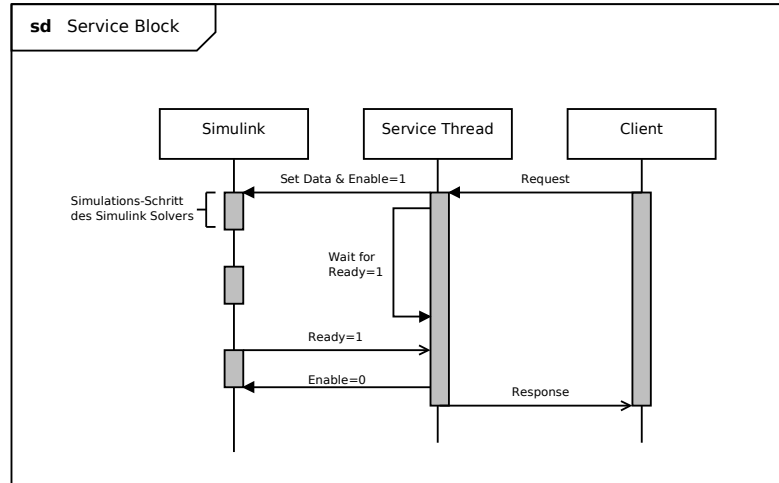


Abbildung 4.10: Zeitlicher Ablauf beim Aufruf des RPC-Blocks mit dem Enable/Ready Protokoll.

Falls es bei einem RPC-Aufruf Probleme gibt und das Modell den Ready-Eingang des Blocks nicht auf Eins setzt, kehrt der Block nicht aus dem Aufruf zurück. Damit dieses Problem in der Praxis nicht auftritt, wurde ein Mechanismus in Form eines einfachen Timers hinzugefügt (siehe Abbildung 4.11). Falls das Modell nach einer bestimmten Anzahl von Simulink-Rechenzyklen den Ready-Eingang nicht auf Eins setzt, verlässt der Block den Aufruf von allein. In diesem Fall wird mit der Response ein Fehler-Code mitgesendet.

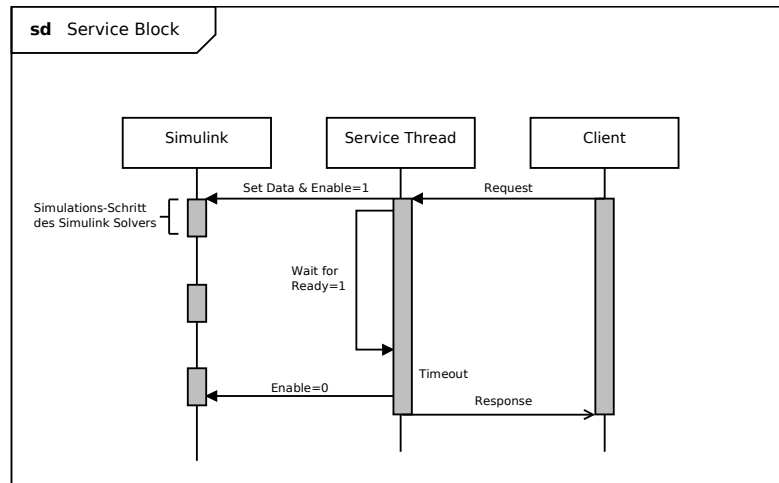


Abbildung 4.11: Zeitlicher Ablauf beim Aufruf des RPC-Blocks bei einem Timeout.

Die Rückgabe des Systemzustands an das jeweilige Benutzerprogramm wird durch einen auf den RPC-Block zugeschnittenen Publisher durchgeführt. In den ersten Entwürfen der RPC-Blöcke war angedacht, den ersten Systemzustand mit der Response des Blocks an das

Benutzerprogramm zu übertragen. Dieser Ansatz ist nicht weiter verfolgt worden, da er mit dem Publisher redundant ist und die Simulink-Modelle durch die vielen zusätzlichen Leitungen unübersichtlich geworden wäre. Zudem wäre der Implementierungsaufwand in den Benutzerprogrammen gestiegen.

4.1.5 Codegenerator

In dieser Arbeit wurde ROS als Middleware ausgewählt. ROS bietet eine einfache IDL, mit der Nachrichten für die Request-Reply-Services und die Publisher und Subscriber definiert werden. Der IDL-Compiler, der im ROS-Buildsystem enthalten ist, generiert Code für C/C++ und Python. Diese Idee wird an dieser Stelle mit einem Codegenerator weitergeführt, der direkt aus den Nachrichtendefinitionen und dem generierten C/C++ Code die Simulink S-Functions für die Blöcke erzeugt.

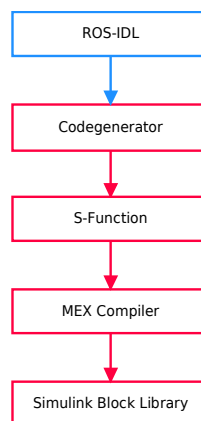


Abbildung 4.12: Prinzipieller Aufbau des Codegenerators für die Erzeugung der S-Functions und der Simulink-Block-Bibliothek.

Der prinzipielle Aufbau des Codegenerators ist in Abbildung 4.12 gezeigt. Aus den Definitionen der Nachrichten soll der C-Code für die S-Functions generiert werden. Des Weiteren sollen die S-Functions für die Simulink-Blöcke mit dem Matlab MEX-Compiler compiliert und zu einer Simulink-Block-Bibliothek zusammengefasst werden. Die automatische Erzeugung der Block-Bibliothek ermöglicht den Anwendern eine komfortable Erstellung der Schnittstellen für verschiedene Anwendungen.

4.2 Validierung des Konzepts mittels einer Beispielanwendung

Bei der Entwicklung des Konzeptes zur Anbindung von Nutzerschnittstellen an die verteilte Regelungsanwendung ist versucht worden, dieses Konzept allgemeingültig zu erstellen. Dadurch kann das Konzept mit ähnlichen Nachrichten-orientierten Middleware-Lösungen realisiert werden. In diesem Abschnitt der Arbeit wird nun das zuvor erarbeitete Konzept auf einen Teil des MiroSurge-Projekts angewendet. Dieser Ausschnitt wird für die Einarmapplikation mit dem MIRO-Roboter verwendet. Mögliche Szenarien sind hierbei die

Steuerung oder das Umschalten der Regler eines Roboterarms zu Testzwecken. Diesen Anwendungsfall betreffen eine überschaubare Anzahl an Anwendungen, die miteinander interagieren müssen. Dazu gehören ein MIRO-Roboter, der durch die MIRO-HAL gesteuert wird, das dazugehörige Regelungsmodell, eine GUI und ein TMCD.

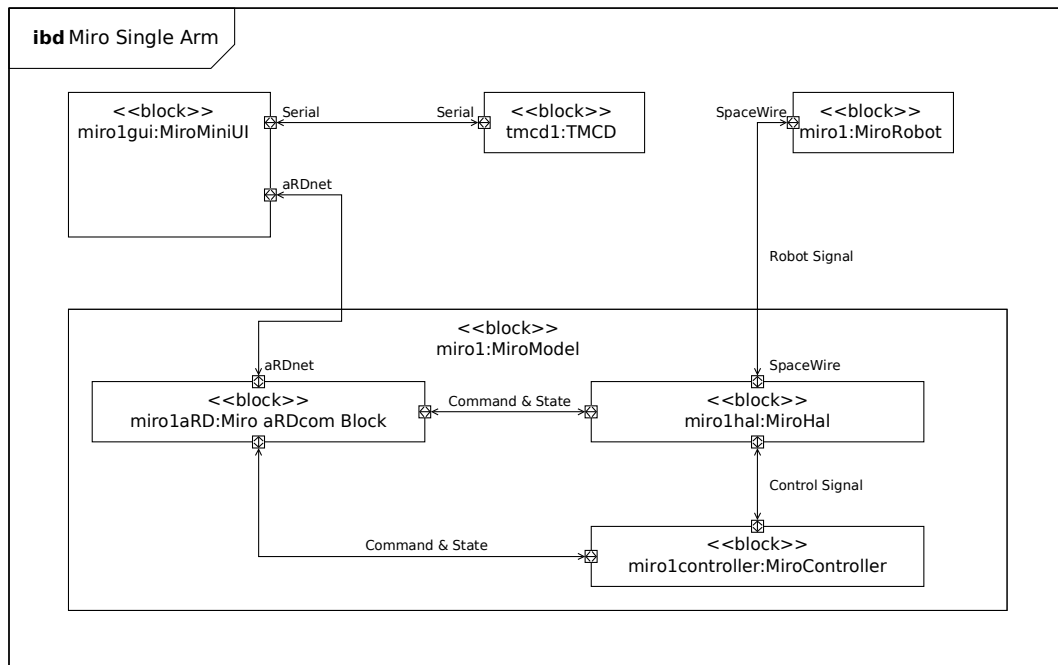


Abbildung 4.13: Architektur der MIRO Einarmapplikation.

In Abbildung 4.13 ist eine Übersicht der Einarmapplikation gezeigt. Sie besteht aus dem MIRO-Roboter, der mit einer SpaceWire-Verbindung an das Reglermodell angeschlossen ist. Das Modell für die Regelung ist ähnlich dem in der gesamten MiroSurge-Applikation.

Zudem gibt es eine GUI-Anwendung, MiroMiniUi genannt, die auf einem Linux System ausgeführt werden kann. Mit ihr besteht die Möglichkeit, die Regler umzuschalten und mit der HAL die Hardware des Roboters zu starten und zu stoppen. Außerdem wird das TMCD direkt von der Anwendung aus angesprochen. Die programmierbaren Tasten des TMCDs wurden so konfiguriert, dass sich auch mit ihnen die Regler umschalten lassen. Das TMCD besitzt also einen Teil der Funktionalität der GUI-Anwendung.

Zu den drei bekannten Reglermodi zur Steuerung des Roboters wurde in dieser Anwendung noch eine weitere Möglichkeit hinzugefügt. Mit der SpaceMouse des TMCDs ist es möglich, den TCP (Tool Center Point) des Roboters direkt zu verschieben und somit den Roboter direkt mit der SpaceMouse zu steuern. Dazu wird der Regler um das „VelocityInterface“ erweitert. Die Daten der SpaceMouse werden direkt in Geschwindigkeitsvektoren umgerechnet und an den Regler weitergeleitet.

Wie in der Abbildung 4.13 zu erkennen ist, wird für die Kommunikation mit dem Reglermodell eine aRDnet Verbindung verwendet. Das größte Problem bei der Verwendung von aRDnet an dieser Stelle ist, dass für die Datenübertragung eine verbindungslose UDP-

Kommunikation verwendet wird. Dadurch kann nicht sichergestellt werden, dass die Daten übertragen werden.

Auch ist zu erkennen, dass das TMCD nicht direkt mit dem Simulink-Modell verbunden ist, sondern die Daten über die aRDnet-Verbindung der GUI mitgesendet werden. Da über die aRDnet Schnittstelle alle Daten übertragen werden, ist die Schnittstelle sehr breit. Für die Überarbeitung der Anwendung wurde die aRDnet Schnittstelle zum Modell in schmalere Schnittstellen aufgeteilt. Eine Schnittstelle für die HAL und eine für den Regler (siehe Abbildung 4.14). Die Ansteuerung des TMCDs wurde von der GUI-Anwendung in eine eigene Anwendung verschoben. Es wurden jeweils ein RPC-Service Block und ein Publisher für die zwei Schnittstellen eingefügt. Für die Geschwindigkeit der SpaceMouse wurden nur Publisher- und Subscriber-Blöcke vorgesehen. Wobei die Geschwindigkeit nur von der TMCD-Anwendung an das Modell gesendet werden kann. In der GUI-Anwendung wird die aktuelle Geschwindigkeit des Roboters lediglich dargestellt. Die Aufteilung wurde nach dem zuvor vorgestellten Konzept durchgeführt.

Die neue Aufteilung ermöglicht die Verwendung des TMCDs und der GUI-Anwendung auf zwei unterschiedlichen Rechnern. Zudem ist es möglich, mehrere Instanzen der GUI-Anwendung auf verschiedenen Rechnern zu starten. Auch sollte die Aufteilung der breiten aRDnet-Schnittstelle in kleinere universellere Schnittstellen für die ROS-Simulink-Blöcke mit Hilfe des geplanten Codegenerators gelöst werden.

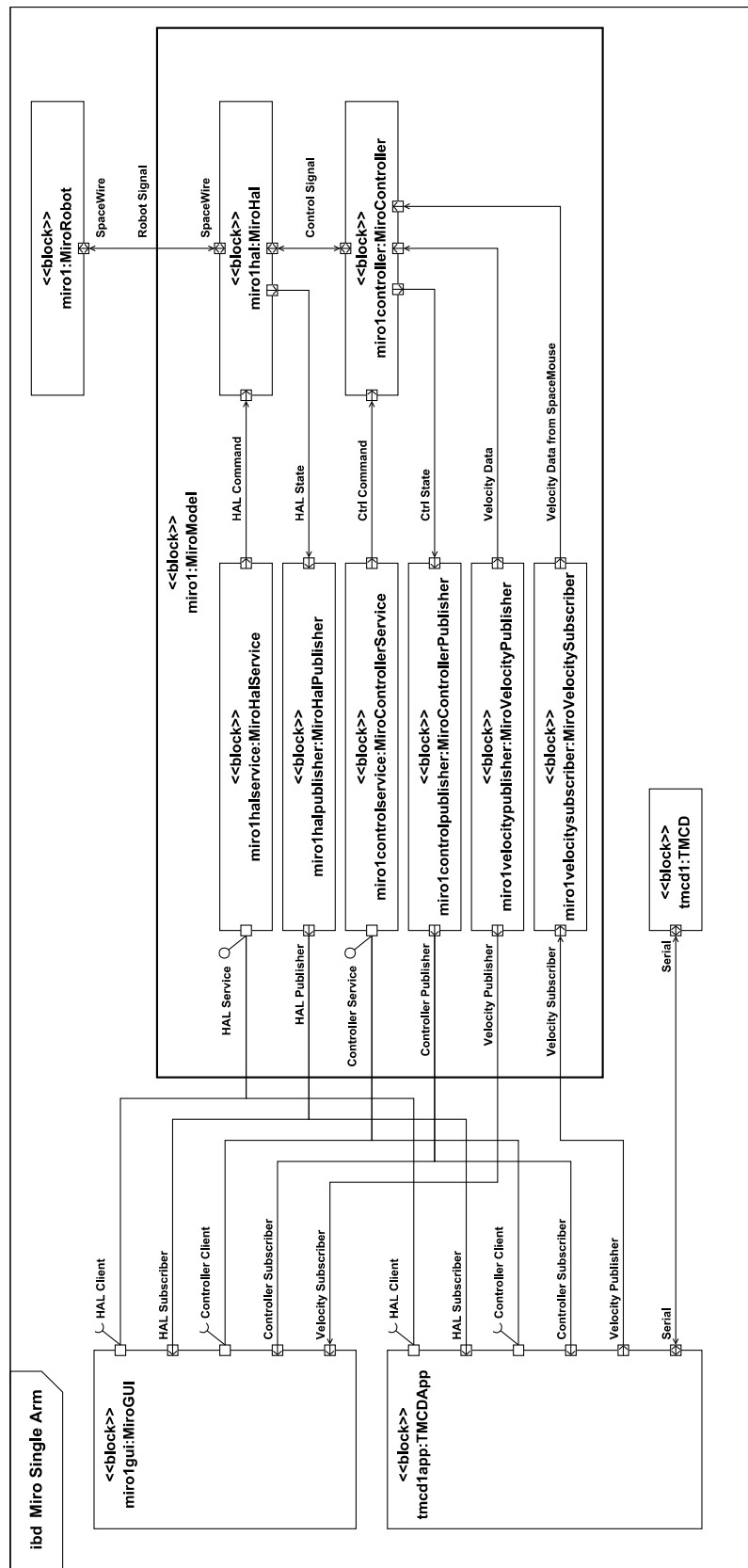


Abbildung 4.14: Überarbeitete Einarmapplikation nach dem vorgestellten Konzept.

5 Implementierung der ROS-Simulink-Schnittstelle

In diesem Kapitel wird die Implementierung der im Konzept vorgestellten Kommunikationsblöcke für Simulink beschrieben. Es wird die Implementierung der drei zuvor erläuterten Schnittstellen Publisher, Subscriber und Service zwischen ROS und Simulink vorgestellt. Die Umsetzung wird mit der C++-Bibliothek roscpp und den Simulink S-Functions gelöst. Abschließend wird auf den Aufbau des Codegenerators für die automatische Erzeugung der Simulink-Blöcke eingegangen. Dazu wird zunächst im Detail auf ROS und die Simulink S-Functions eingegangen.

5.1 ROS Pakete und IDL

ROS ist ein modular aufgebautes System. Es besteht aus einem Paketmanagementsystem, welches einzelne ROS Komponenten in sogenannte Packages und Stacks aufteilt. Stacks bilden eine übergeordnete Ebene und können mehrere Pakete enthalten. Pakete enthalten Bibliotheken, Tools, Anwendungen oder Definitionen für Nachrichten.

In ROS wird eine einfache Schnittstellenbeschreibungssprache verwendet. Mit ihr werden die C++ und Python Schnittstellen für die Publisher, Subscriber und Services erzeugt. Es wird zwischen msg-Dateien und srv-Dateien unterschieden. Messages für Publisher und Subscriber werden mit msg-Dateien beschrieben, dagegen beschreiben srv-Dateien Schnittstellen zu ROS Services. Beide Nachrichtendefinitionen sind grundsätzlich gleich aufgebaut, wobei sich srv-Beschreibungen nur durch einen Antwortteil in der Nachricht unterscheiden (siehe Listing 5.1 und Listing 5.2).

```
int64 A
int64 B
```

Listing 5.1: Einfache msg-Beschreibung für Publisher und Subscriber. Die Nachricht besteht aus A und B.

```
int64 A
int64 B
---
int64 Sum
```

Listing 5.2: Einfache srv-Beschreibung. Der Request besteht aus A und B. Sum ist die Response.

Die Nachrichten werden in einer C-ähnlichen Syntax beschrieben und unterstützen folgende Datentypen:

- bool
- int8, int16, int32, int64 (plus uint*)
- float32, float64
- string
- time, duration
- array[] (variable Größe) und array[Size] (feste Größe)

Zudem ist es möglich, msg-Dateien aus anderen Paketen einzubinden und somit komplexe verschachtelte Datenstrukturen für Nachrichten zu erzeugen. ROS selbst liefert in verschiedenen Packages vordefinierte Standard-Nachrichten. Diese Nachrichten werden in einfachen Textdateien beschrieben und in ROS-Paketen abgelegt. Im ROS-Buildsystem sind Skripte enthalten, die aus den Nachrichtendefinitionen die Schnittstellen für die verschiedenen Programmiersprachen erzeugen. Die generierten C und Python Dateien stellen Service- und Message-Klassen für die Nutzung in Anwendungen zur Verfügung.

5.2 ROS Topics

In ROS werden Services, Publisher und Subscriber in Topics geordnet. Die Topics entsprechen logischen Adressen von ROS-Komponenten zur Laufzeit. Die Topics werden in ROS wie Unix Dateisystempfade geordnet. Sie bestehen aus Strings, die durch Slashes getrennt werden.

Durch die Topic-Namen sind zum Beispiel Strukturen wie „/miro1/controller_service“ und „/miro1/controller_updates“ möglich. Durch diese baumartige Struktur lassen sich die Namen der Komponenten logisch einteilen und auf die Simulink-Modelle verteilen.

5.3 ROS C++-Schnittstelle

Die roscpp Implementierung ist eine C++ Schnittstelle zu ROS. Mit ihr besteht die Möglichkeit, Nodes, Publisher, Subscriber und Services zu erstellen. Unter den verschiedenen ROS Implementierungen ist roscpp am besten für die Integration in Simulink geeignet.

Jedes Node entspricht hierbei einem Prozess und somit einer unabhängig ausführbaren Datei. Bei Initialisierung der Anwendung wird ein NodeHandle erzeugt. Alle darauf folgenden Zugriffe zum Erzeugen von Publishern, Subscribern und Services werden über dieses NodeHandle durchgeführt. In Listing 5.3 wird ein Beispiel der Initialisierung eines einfachen Nodes mit einem Publisher gezeigt. In diesem Beispiel versendet der Publisher einen String als Nachricht.

```

#include "ros/ros.h"
#include "std_msgs/String.h"

int main(int argc, char **argv)
{
    ros::init(argc, argv, "my_publisher");
    ros::NodeHandle n;
    ros::Publisher pub = n.advertise<std_msgs::String>("my_topic", 1000);

    std_msgs::String msg;
    msg.data = "hello";
    pub.publish(msg);
    return 0;
}

```

Listing 5.3: Die Main-Funktion eines einfachen ROS Node mit Publisher.

Die ROS `init()` Funktion nimmt den Nodennamen und Kommandozeilenparameter entgegen und erlaubt damit eine Umkonfigurierung des Nodesnamens und anderer Parameter [47] zum Programmstart. In ROS besteht keine Möglichkeit, Nodennamen doppelt zu verwenden. Beim Erstellen des ersten `NodeHandle`s wird das Node vollständig initialisiert und am ROS Master angemeldet. Zusätzlich werden die ROS internen Kommunikationsthreads gestartet⁴.

Der im Beispiel gezeigte Publisher ist sehr simpel gehalten. Ein wenig komplexer wird die Implementierung dagegen wenn ein Subscriber oder ein Service genutzt wird. Hierfür werden Callback-Funktionen benötigt um die eintreffenden Daten und Anfragen zu verarbeiten.

```

#include "ros/ros.h"
#include "std_msgs/String.h"

void Callback(const std_msgs::String::ConstPtr& msg)
{
    cout << msg->data.c_str() << endl;
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "my_subscriber");
    ros::NodeHandle n;
    ros::Subscriber sub = n.subscribe("my_topic", 1000, Callback);

    ros::spin();
    return 0;
}

```

Listing 5.4: Ein einfaches ROS-Node mit Subscriber und dazugehöriger Callback-Funktion.

In Listing 5.4 ist eine Implementierung eines einfachen Subscribers zu sehen. Bei der Registrierung des Subscribers am `NodeHandle` wird die zu verwendende Callback-Funktion übergeben, die beim Eintreffen einer Nachricht abgearbeitet wird. Dies geschieht in diesem Fall nur dann, wenn die Funktion `spin()` aufgerufen wird. Das „Spinning“ spielt in ROS eine

⁴ Für die Kommunikation der ROS-Nodes und des ROS-Masters werden für jedes Node 5 Threads gestartet (TopicManager, ServiceManager, ConnectionManager, PollManager, XMLRPCManager).

wichtige Rolle. Mit ihm werden die Callback-Funktionen erst ausgeführt. Beim Eintreffen einer Nachricht wird diese von den ROS Kommunikationsthreads, in eine Callback-Queue eingehängt und erst durch den Aufruf einer Spinning-Funktion abgearbeitet. Das Spinning kann Single-threaded synchron im selben Thread des Hauptprogramms durch den Aufruf von `spin()` oder Multi-threaded asynchron in Hintergrund-Threads mit dem `AsyncSpinner` durchgeführt werden. ROS bietet dafür verschiedene Lösungsmöglichkeiten an. Zudem können zusätzliche Callback-Queues für jeden Subscriber und Service erzeugt werden [48].

Ein Service unterscheidet sich in der Implementierung durch eine erweiterte Callback-Funktion die eine Response besitzt. Ein einfaches Beispiel eines Services wird in Listing 5.5 gezeigt.

```
#include "ros/ros.h"
#include "my_package/test.h"

bool ServiceCallback( my_package::test::Request &req,
                     my_package::test::Response &res )
{
    res.result = req.a + req.b;
    return true;
}

int main(
in(int argc, char **argv)
{
    ros::init(argc, argv, "my_server");
    ros::NodeHandle n;
    ros::ServiceServer service = n.advertiseService("my_service",
ServiceCallback);

    ros::spin();
    return 0;
}
```

Listing 5.5: Ein einfaches ROS-Node mit Service und dazugehöriger Callback-Funktion.

5.4 Portierung der ROS-Bibliotheken auf QNX 6.5

Erste Tests von ROS wurden auf den OpenSuse Instituts-Rechnern (Suse Linux Enterprise Desktop 11 SP1) durchgeführt. Auf den OpenSuse Systemen wurde eine volle ROS-Desktop Installation durchgeführt, damit die graphischen und Konsolen-Tools zur Verfügung stehen. [49]

Damit ROS auch auf dem Betriebssystem QNX 6.5 verwendet werden kann, müssen die Kommunikations-Bibliotheken von ROS für QNX kompiliert werden. Die eigentliche Middleware, die in ROS enthalten ist, befindet sich im „ros_comm“-Stack [50] und besteht aus fünf wichtigen Bibliotheken. Die ROS Bibliotheken haben Abhängigkeiten zu den Boost C++ Bibliotheken [51]. Zudem basiert der Logger der in ROS enthalten ist auf dem Apache-Logging-Service `log4cxx` [52], der selbst die Apache Portable Runtime [53] benötigt. In Tabelle 5.1 werden alle verwendeten Bibliotheken aufgelistet. Die Apache Portable Runtime

Bibliotheken und die Expat Bibliothek müssen nicht selbst kompiliert werden. Es können vorkompilierte Pakete verwendet werden⁵.

Bibliothek	Herkunft	Version
libboost_filesystem	Boost C++ Libraries	1.40
libboost_regex	Boost C++ Libraries	1.40
libboost_serialization	Boost C++ Libraries	1.40
libboost_signals	Boost C++ Libraries	1.40
libboost_system	Boost C++ Libraries	1.40
libboost_thread	Boost C++ Libraries	1.40
libapr-1	Apache Portable Runtime	1.4.2
libapr-util-1	Apache Portable Runtime	1.3.10
libexpat	The Expat XML Parser	2.0.1
liblog4cxx	Apache Logging Projekt	0.10.0 (SVN 4.4.2011)
libros	ROS Projekt (ros_comm Stack)	1.4.5 (Diamondback)
libxmlrpc	ROS Projekt (ros_comm Stack)	1.4.5 (Diamondback)
librosconcole	ROS Projekt (ros_comm Stack)	1.4.5 (Diamondback)
librostime	ROS Projekt (ros_comm Stack)	1.4.5 (Diamondback)
libroscpp_serialize	ROS Projekt (ros_comm Stack)	1.4.5 (Diamondback)

Tabelle 5.1: Für die ROS-Kommunikation unter QNX 6.5 benötigte Bibliotheken.

5.5 Simulink S-Functions

In Simulink können benutzerdefinierte Blöcke in Form von S-Functions (System Functions) erstellt werden. S-Functions können in verschiedenen Programmiersprachen erstellt werden. Für diese Arbeit ist nur die Schnittstelle zu C/C++ relevant. In C geschriebene S-Functions kann beliebiger C-Code integriert werden. Für die Kompilierung der S-Functions wird das mit Matlab/Simulink mitgelieferte Programm „mex“ verwendet. Dieses Programm ruft den unter dem jeweiligen System installierten C-Compiler auf und compiliert die erstellte S-Function zu einer Mex-Datei. Diese entsprechen dynamisch ladbaren Bibliotheken, die von Simulink ausgeführt werden können. Sie werden auch vom Realtime-Workshop für die Kompilierung von Programmen für die echtzeitfähigen Zielplattformen verwendet.

In Simulink wird ein synchrones Berechnungsmodell genutzt. Die Zeitschritte und Raten, in der die Blöcke abgearbeitet werden, werden durch den Simulink-Solver vorgegeben. Dies spiegelt sich auch in der Art wider, in der die S-Functions aufgebaut sind. Sie bestehen aus einer Reihe von Callback-Funktionen, die Simulink zu bestimmten Zeitpunkten aufruft. So gibt es Funktionen für die Initialisierung der Ein- und Ausgänge, Berechnungen und das Beenden eines Blocks (siehe Abbildung 5.1).

⁵ Die vorkompilierten Pakete konnten dem PKGSRC-Projekt, entnommen werden.
http://ftp.netbsd.org/pub/pkgsrc/packages/QNX/i386/6.5.0_head_20101224/All/

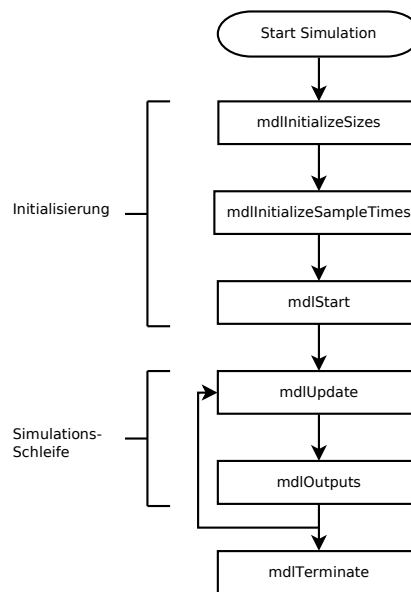


Abbildung 5.1: Ablauf der S-Function Funktionsaufrufe von Simulink zur Initialisierung und Simulation. [54]

Zur Initialisierung des Blocks werden `mdlInitializeSizes`, `mdlInitialSampleTimes` und `mdlStart` aufgerufen. Während der Simulationsschleife werden mit dem Aufruf von `mdlUpdate` die Eingänge und der interne Zustand des Blocks aktualisiert. Mit `mdlOutputs` werden die Berechnungen durchgeführt, die zur Aktualisierung der Ausgänge nötig sind. Aufräumarbeiten beim Beenden der Simulation werden in der `mdlTerminate` Funktion durchgeführt. [54]

Zudem gibt es das sogenannte „SimStruct“, in dem der aktuelle Zustand des Blocks gespeichert ist. Mit diesem Datenobjekt kann zudem auf verschiedene Einstellungen des aktuellen Simulink-Modells zugegriffen werden. In dieser Struktur werden auch die Einstellungen, wie die Ein- und Ausgänge, sowie die Abtaste (Sample Time) des Blocks gespeichert. Im SimStruct können auch Zeiger von Objekten, die vom Benutzer erstellt wurden, abgelegt werden, damit diese persistent bleiben. [55]

Weiter kann in den S-Functions zur Laufzeit erkannt werden, ob Simulink im Simulationsmodus, auf einem Real-Time-Workshop Zielsystem oder im External-Mode ausgeführt wird.

5.6 Aufbau der Simulink-Blöcke

In diesem Teil der Arbeit wird die Implementierung der in Kapitel 4.1.4 erarbeiteten Simulink-Blöcke erläutert. Bei der Integration der ROS Publisher, Subscriber und Services in die Simulink S-Functions müssen zu Anfang noch allgemeine Probleme gelöst werden, die alle implementierten Blöcke betreffen. Dies betrifft die Initialisierung des ROS-Nodes in Simulink und die Synchronisierung zwischen den ROS-Callback- und Simulink-Funktionen. Es muss sichergestellt werden, dass die Initialisierung des Nodes zum Modellstart geschieht.

Zudem dürfen die Callback-Funktionen nicht das synchrone Berechnungsmodell von Simulink zerstören damit, die geforderte Echtzeitfähigkeit erhalten bleibt. Anschließend wird auf die einzelnen Blöcke eingegangen.

5.6.1 Initialisierung der Simulink S-Functions

Die Initialisierung setzt sich für alle Blöcke aus den S-Function-Aufrufen von `mdlInitializeSizes`, `mdlInitializeSampleTimes` und `mdlStart` zusammen. In den folgenden drei Abschnitten werden die Initialisierungsaufgaben, die alle drei implementierten Simulink-Blöcke betreffen, erläutert.

Zu Beginn der Simulation werden für alle Blöcke in der `mdlInitializeSizes` die Ein- und Ausgänge mit den dazugehörigen Datentypen definiert. Simulink unterstützt nicht alle Datentypen, die in ROS definierbar sind. Strings und Arrays mit dynamischer Länge können zum Beispiel nicht in Simulink überführt werden. Verschachtelte Strukturen, die durch die Einbindung anderer Msg-Dateien erzeugbar sind, müssen für die Darstellung in Simulink-Blöcken abgeflacht werden. Das bedeutet, dass einzelne Member-Variablen der generierten Nachrichten-Klassen direkt zugreifbar gemacht werden müssen.

In Simulink-Modellen können Blöcke unterschiedliche Abtastraten haben. Die Abtastraten werden in der `mdlInitializeSampleTimes`-Funktion der Blöcke definiert. Service- und Subscriber Blöcke können mit der Modellabtastrate aufgerufen werden. Publisher-Blöcke erzeugen einen zu großen Datendurchsatz, wenn sie mit einer hohen Modellabtastrate senden. Daher werden sie mit Rate-Transition-Blöcken gedrosselt. Rate-Transition-Blöcke reduzieren die Abtastraten von bestimmten Blöcken, indem sie nur jeden n-ten Wert an den Block weitergeben. Subscriber-Blöcke besitzen eine Einstellmöglichkeit um die Rate zu definieren, mit der sie aufgerufen werden.

Die Initialisierung des ROS Nodes wird durch den Konstruktor der Klasse `rosNodeInstance` durchgeführt. Jeder Simulink-Block, der die ROS-Schnittstelle integriert, erstellt in der `mdlStart`-Funktion eine Instanz der Klasse. Der erste Simulink-Block im Modell übernimmt die Initialisierung des ROS Nodes und erstellt die Threads für die Abarbeitung der Callback-Aufrufe, die durch eintreffende Nachrichten aufgerufen werden. Die ROS-Bibliothek lässt sich nur einmal pro Prozess instanziiieren. Dadurch kann abgefragt werden, ob die ROS-Initialisierung schon durchgeführt wurde. Dies gilt für die Verwendung im Real-Time-Workshop sowie in der Simulation.

5.6.2 Initialisierung der S-Functions für den External-Mode

Wird das mit dem Real-Time-Workshop erstellte Modell auf dem Echtzeit-System ausgeführt, kann eine Verbindung mit dem External-Mode von Simulink zu diesem System erzeugt werden. So lassen sich Daten des Systems in der Simulink-Umgebung anzeigen. In diesem Fall wird das selbe Modell verwendet. Es wird aber in der Simulink-Benutzeroberfläche vorher auf den External-Mode gewechselt. Da das Modell im Modelleditor in diesem Fall nur für die Darstellung und nicht für Berechnungsaufgaben verwendet wird, müssen die Blöcke

anders als im normalen Simulations-Modus initialisiert werden. Im External-Mode wird daher in der mdlStart-Funktion kein ROS-Node erstellt und alle Funktionalität der Blöcke bis auf die Initialisierung der Blockeingänge und -ausgänge deaktiviert.

5.6.3 Bearbeitung eintreffender Nachrichten

Die Abarbeitung der, durch eintreffende Nachrichten ausgelösten, Callback-Funktion, wird durch den AsyncSpinner von ROS durchgeführt. Mit diesem Spinner kann eine definierte Anzahl an Threads gestartet werden, die parallel zum Simulink-Modell laufen.

Wenn eine Nachricht, für einen ROS-Subscriber oder ROS-Service, eintrifft kann diese sofort bearbeitet werden, wenn dies in einem anderen Thread geschieht. Allerdings dürfen die empfangenden Werte erst an die Blockausgänge geschrieben werden, wenn ein Aufruf des Blocks nach dem Berechnungsmodell stattfindet. Dies entspricht einem Aufruf der mdlOutputs Funktion. Dieser Funktionsaufruf sollte so kurz wie möglich ausfallen damit das Modell nicht blockiert wird. Dies bedeutet, dass die empfangenen Werte zwischengepuffert werden müssen. Außerdem muss ein Zugriffsschutz implementiert werden damit die Synchronisation zwischen Callback-Thread und Simulink-Block funktioniert.

Für jede in der ROS-IDL erstellte Nachricht, wird von dem IDL-Compiler im ROS-Buildsystem eine Klasse generiert. Damit für die Pufferung zwischen Callback-Thread und Simulink keine Datenstrukturen erstellt werden müssen, werden die generierten Message-Klassen hierzu verwendet. Zu diesem Zweck wurde die Template-Klasse rosRequestBuffer erstellt, welche mit den unterschiedlichen generierten Message-Klassen umgehen kann. Zudem bietet diese Klasse für die Synchronisation Methoden, die den Zugriffsschutz nach dem Erzeuger-Verbraucher-Muster regeln.

5.6.4 Aufbau des Subscriber-Blocks

Der Subscriber-Block empfängt Nachrichten und setzt diese in Simulink-Signale um. Die Nachrichten werden von den ROS-Kommunikations-Threads empfangen und in eine Callback-Liste eingehängt. Die Callbacks werden durch die mit dem asynchronen Spinner gestarteten Callback-Threads abgearbeitet. Der Block darf die empfangenen Werte erst an das Modell übertragen, wenn er nach dem Simulink-Berechnungsmodell an der Reihe ist. Daher werden die Daten mit Hilfe der Template-Klasse zwischen gepuffert und erst in der mdlOutputs-Funktion des Blocks an die Ausgänge geschrieben. In Abbildung 5.2 ist die Synchronisation, zwischen der Client-Anwendung und dem Subscriber-Block, nach dem Erzeuger-Verbraucher-Muster gezeigt. Der Client veröffentlicht eine Nachricht. Der Subscriber-Thread verarbeitet die Nachrichten in einer Callback-Funktion und kopiert sie in den Puffer. Beim Aufruf der mdlOutputs-Funktion des Simulink-Blocks wird der Puffer gelesen und die Werte an die Blockausgänge geschrieben.

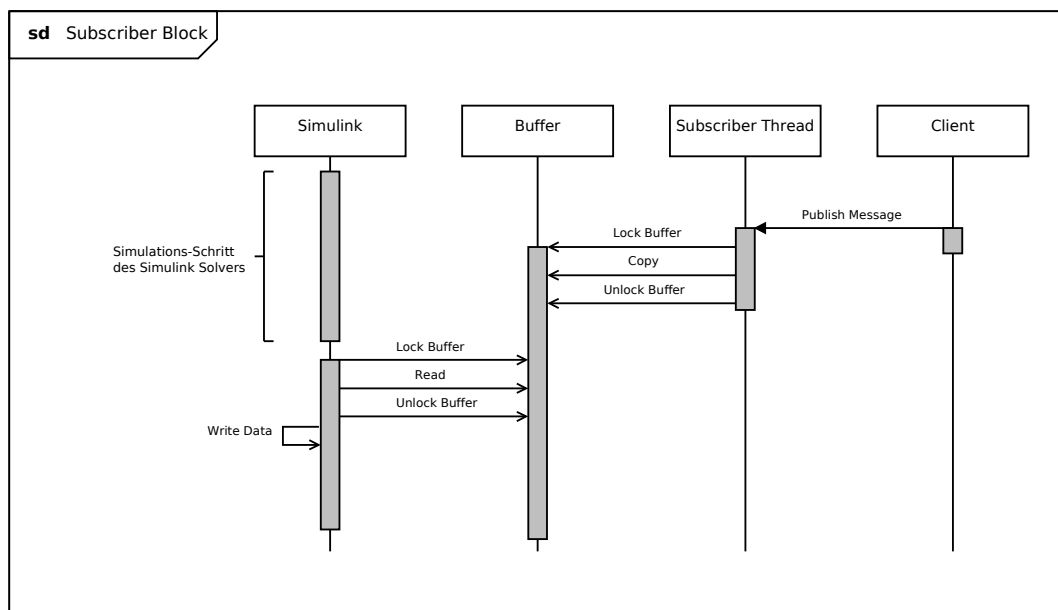


Abbildung 5.2: Synchronisation zwischen Subscriber-Thread und Simulink.

5.6.5 Aufbau des Service-Blocks

Der Service-Block stellt einen Request-Response-Service bereit. Mit ihm ist es möglich, Benutzerbefehle sicher in ein Simulink-Modell zu senden. Die Übertragung in das Modell kann vom Modell selbst bestätigt werden. Dafür wurde der in Kapitel 4.1.4 beschriebene Enable-Ready-Mechanismus implementiert. Der Client wird so lange blockiert, bis das Modell die Response absendet. Auch hierfür wurde für die Pufferung und Synchronisierung die Template-Klasse verwendet.

In Abbildung 5.3 wird der Synchronisationsmechanismus erläutert. Der asynchrone Service-Thread verarbeitet die Anfragen, kopiert die empfangenen Daten in den Buffer und wartet auf das Ready-Signal aus Simulink. Der Simulink-Block liest in der mdlOutputs-Funktion den Buffer, schreibt die Daten an die Ausgänge des Blocks und setzt den Enable-Ausgang. In der mdlUpdate-Funktion wird der Ready-Eingang des Blocks in den darauf folgenden Modelltakten abgefragt. Wird er auf eins gesetzt oder läuft ein Timeout ab, wird ein Signal an den Service-Thread gesendet um ihn zu entsperren und die Response an den Clienten zu senden. Der Timeout wird durch einen Parameter für die S-Funktion von Simulink übergeben.

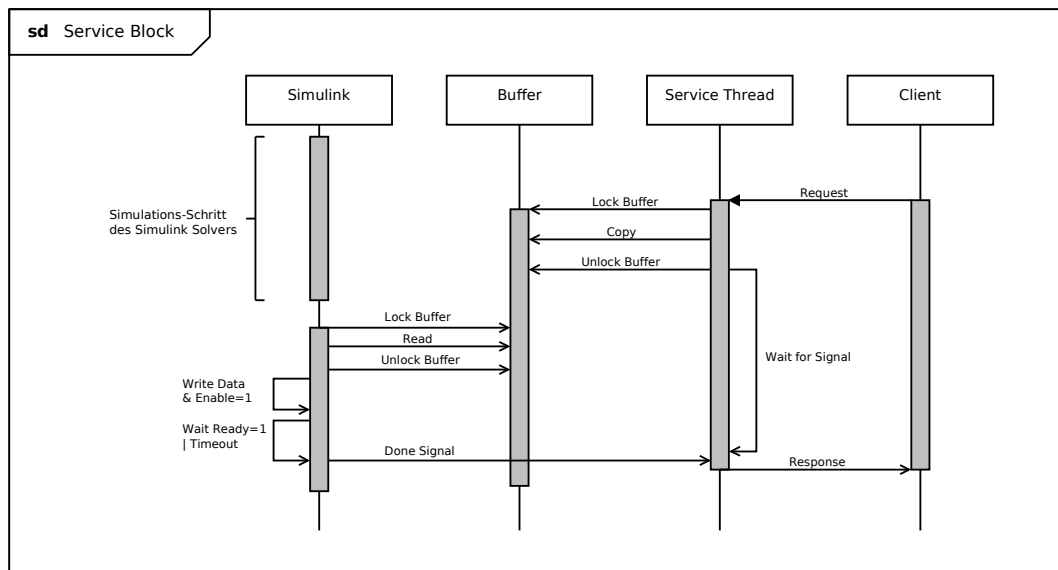


Abbildung 5.3: Synchronisation zwischen Service-Thread und Simulink.

Um das in Kapitel 4.1.1 eingeführte Zeitstempelverfahren anwenden zu können, müssen in der Schnittstellenbeschreibung für einen Service die zwei Variablen „float64 ros2simulink_timestamp“, und „int32 ros2simulink_error“ definiert sein.

Der Timestamp muss beim Service als Request-Parameter angegeben werden, damit dieser überprüfen kann, ob ein Client keine veralteten Zustandsinformationen aus dem Modell benutzt. Damit ein Aufruf aus einer Client-Anwendung erfolgreich ausgeführt wird, muss entweder ein aktueller Zeitstempel oder der Wert „-1“ mitgesendet werden. Letzteres ist nützlich um zum Beispiel Aufrufe mit dem ROS Konsolen-Tool „rosservice“ durchzuführen.

Der „int32 ros2simulink_error“ ist die Response des Service-Blocks und wird benötigt um den Error-Code des Service-Blocks an einen Clienten zu schicken. Dieser kann so überprüfen, ob der Aufruf erfolgreich war oder einen möglichen Fehler identifizieren. Die Rückgabewerte des Service-Blocks an den Clienten sind in Tabelle 5.2 aufgelistet:

Fehler-Code	Bedeutung
0	Aufruf erfolgreich
-1	Timeout durch nicht Eintreffen des Ready-Signals am Block ausgelöst
-2	Service-Aufruf auf Grund eines alten Zeitstempels nicht akzeptiert
-3	Der Service wird von einem anderen Clienten verwendet

Tabelle 5.2: Mit ros2simulink_error zugegebene Fehler-Codes des Service-Blocks.

5.6.6 Aufbau des Publisher-Blocks

Bei einem Publisher, welcher Daten aus einem Simulink-Modell heraus sendet, sieht die Umsetzung ein wenig einfacher aus. Ein Publisher benötigt keine Callback-Funktion und er

besitzt nur Eingänge, an denen die zu sendenden Daten anliegen. Eingangswerte werden in Simulink im Berechnungsablauf durch den Aufruf von `mdlUpdate` in den Blockzustand übernommen. In dieser Funktion werden die Daten in eine ROS-Message kopiert und versendet. Damit der für den Service-Block nötige Zeitstempel durch den Publisher mitgesendet wird, muss die Variable „double `ros2simulink_timestamp`“ in der Nachricht definiert sein. Der Codegenerator fügt die nötige Funktion in die S-Function ein, damit der Zeitstempel in die Nachricht kopiert wird.

5.7 Aufbau des Codegenerators

Mit den entworfenen Simulink-Blöcken konnte gezeigt werden, dass eine Anbindung der ROS-Middleware an Simulink möglich ist. Die händische Erstellung der einzelnen Blöcke für unterschiedliche Aufgaben ist allerdings zeitaufwändig. Für jeden Block muss eine S-Function geschrieben, kompiliert und in Simulink eingebunden werden. Um den Middleware-Gedanken weiterzuführen und die Arbeitsschritte zu automatisieren, wurde ein Codegenerator zur automatischen Erstellung der Simulink-Blöcke aus den Message-Definitionen von ROS implementiert.

Der entwickelte Codegenerator wurde in Python geschrieben, da ROS selbst für die Generierung der Schnittstellen Python-Skripte nutzt. Mit der Python-Bibliothek für ROS gibt es eine komfortable Schnittstelle, um auf die Infrastruktur und das Paketsystem zuzugreifen.

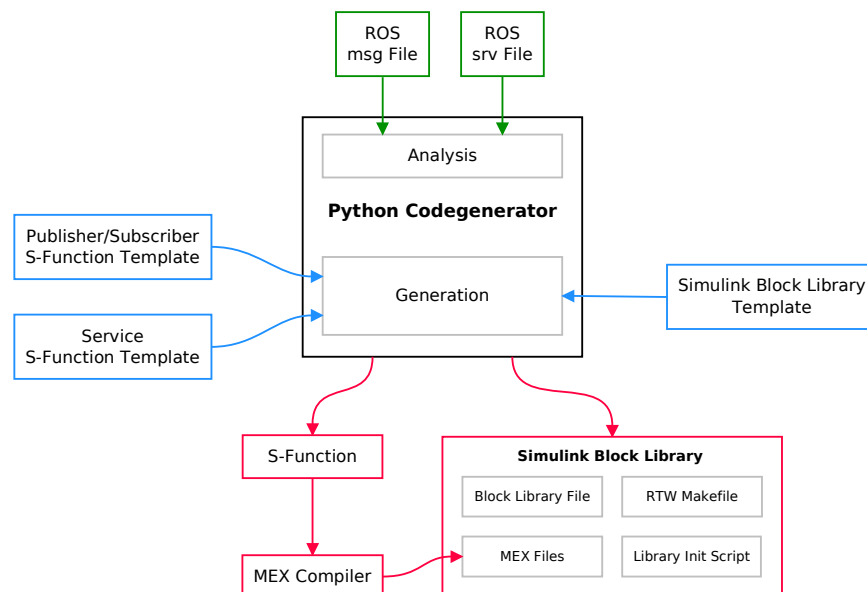


Abbildung 5.4: Aufbau des implementierten Codegenerators zum Erzeugen der Simulink-Blöcke aus der ROS-IDL.

Der Codegenerator liest die ROS-Message Definitionen ein und analysiert diese mit Hilfe der `roslib` Python-Bibliothek [56]. Mit dieser Bibliothek ist es möglich, die Nachrichtendefinitionen zu parsen. Die ausgelesenen Nachrichteninformationen enthalten die Datentypen, Namen und Informationen, ob es sich um ein Array handelt. Da in Simulink nicht

alle Datentypen vorhanden sind, die in ROS definiert werden können, müssen folgende Konvertierungen vorgenommen werden:

ROS Datentyp	Simulink Datentyp	ROS Datentyp	Simulink Datentyp
bool	SS_BOOLEAN	int64	SS_DOUBLE
int8	SS_INT8	uint64	SS_DOUBLE
uint8	SS_UINT8	float32	SS_SINGLE
int16	SS_INT16	float64	SS_DOUBLE
uint16	SS_UINT16	string	-
int32	SS_INT32	time	-
uint32	SS_UINT32	duration	-

Tabelle 5.3: Konvertierung der Datentypen zwischen ROS und Simulink.

Die Datentypen string, time und duration, welche nicht konvertiert werden können, werden ausgeblendet und es wird eine Warnung durch den Codegenerator ausgegeben. Außerdem unterstützt Simulink keine 64bit Integer Datentypen. Diese werden nach double konvertiert und es wird ebenfalls eine Warnung angezeigt.

Für den Aufruf eines Service wird in der Client-Anwendung ein aktueller Zustand des Systems mit dazugehörigem Zeitstempel benötigt. Damit der Zeitstempel vom Publisher versendet wird muss in die Nachrichtendefinition für den Publisher die Variable „float64 ros2simulink_timestamp“, eingetragen werden. Der Codegenerator fügt dadurch automatisch die Funktionalität zum Senden der aktuellen Simulink-Zeit in die S-Function ein. Diese wird durch den Publisher mitgesendet und kann in den Client-Anwendungen für Service-Aufrufe verwendet werden.

Neben den Standarddatentypen können Nachrichten aus anderen ROS-Paketen in den Nachrichtenbeschreibungen verwendet werden. Sie werden durch den Codegenerator rekursiv bis auf die unterstützten Standarddatentypen aufgelöst. Außerdem ist es in der ROS-IDL möglich, aus allen Datentypen mehrfach verschachtelte Arrays zu erzeugen. Verschachtelte Strukturen sind nicht ohne weiteres an den Blöcken möglich. Dies bedeutet, dass Datenstrukturen bis auf eine Array Ebene aufgelöst werden müssen. Der Codegenerator enthält Funktionen, die die Nachrichtendefinitionen auflösen und eine abgeflachte Struktur erzeugen, welche an einem Simulink-Block genutzt werden kann.

Aus den aufgelösten Datenstrukturen werden in einem nächsten Schritt die S-Functions für die Blöcke erstellt. Die Erstellung der S-Functions wird mit String-Templates gelöst. Dazu werden die in der Python String-Klasse enthaltenen Template Methoden verwendet [57]. Die Templates für die Simulink Blöcke werden nach den Beschreibungen aus dem vorherigen Teil der Arbeit erzeugt. In die Templates werden die Ports und die Datentransferoperationen für die S-Functions aus den aufgelösten Nachrichtenstrukturen eingefügt. Nach der Erstellung der S-Funktionen werden diese mit dem Matlab MEX-Compiler zu MEX-Files kompiliert. Der Codegenerator erzeugt für jede Nachrichtendefinition eine S-Function. Für srv-Dateien wird eine S-Function mit Service-Funktionalität und für msg-Dateien eine S-Function mit Publish und Subscribe Funktionalität generiert.

Die S-Funktionen können nicht direkt in ein Simulink-Modell eingefügt werden. Sie werden über einen speziellen S-Funktion-Block eingebunden. In diesem Block werden der Name der S-Function und die Funktionsparameter eingetragen. Zudem können für Blöcke Masken erstellt werden, mit denen Beschriftungen und Logos in Blöcke eingefügt werden können. Die Blöcke mit den Masken können in einer Block-Bibliothek gespeichert und mehrfach verwendet werden.

Die erzeugten Simulink-Blöcke erhalten alle einen Parameter, der den Topic des Blocks definiert. Der Topic-Name wird vom Codegenerator automatisch nach dem Muster „/ROS_Nodename/ROS_Paketname/Message_Name“ generiert. Da jedes Simulink-Modell einem ROS-Node entspricht, wird für den Node-Namen im Topic eine Matlab-Variable („ros_nodename“) eingesetzt. Dies ermöglicht das Festlegen des Node-Namen in den Modelleinstellungen für das gesamte Simulink-Modell. Für die Nachrichtendefinition „MiroHalTopic.msg“ aus dem Paket „miro_msgs“ wird beispielsweise der Topic „/ros_nodename/miro_msgs/MiroHalTopic“ generiert.



Abbildung 5.5: Logos für die ROS Simulink-Blöcke.

Der implementierte Codegenerator erzeugt für jedes ROS-Paket eine Simulink-Block-Bibliothek und speichert diese zusammen mit den erzeugten S-Functions und Logos (siehe Abbildung 5.5) in einem Ordner ab. Zusätzlich wird ein Matlab-Skript zur Initialisierung der Block-Bibliothek und ein Makefile für den Real-Time-Workshop erstellt.

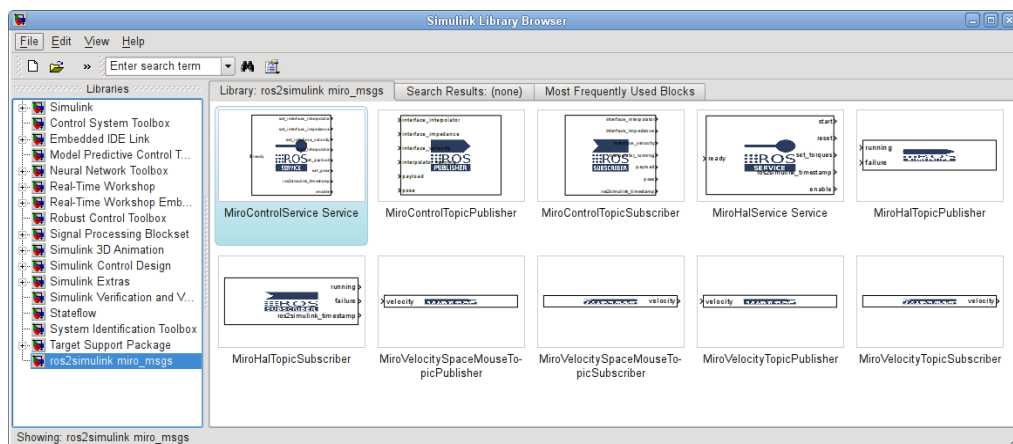


Abbildung 5.6: Simulink Library Browser mit der durch den Codegenerator erzeugten Blockbibliothek

Die erzeugte Block-Bibliothek wird in Matlab durch das generierte Skript vom Anwender in den Simulink-Library-Browser geladen und kann in eigenen Modellen genutzt werden (siehe Abbildung 5.6). Im nächsten Kapitel der Arbeit werden die mit dem Codegenerator erstellten Simulink-Blöcke weiter untersucht und in die Einarmapplikation aus Kapitel 4.2 integriert.

6 Analyse und Test der Implementierung

Dieser Teil der Arbeit beschäftigt sich mit der Analyse der, durch den Codegenerator erzeugten, Simulink-Blöcke und dem dahinter stehenden Konzept. Dazu werden Zeitmessungen der Blöcke durchgeführt. Abschließend wird das entwickelte Konzept auf die in Kapitel 4.2 vorgestellte Beispielanwendung übertragen und getestet.

6.1 Zeitmessungen der Simulink-Blöcke

Damit sichergestellt werden kann, dass die in dieser Arbeit erstellten Kommunikationsblöcke den Echtzeitkontext der Simulink-Modelle nicht negativ beeinflussen, wurden Messungen auf einem Echtzeit-System durchgeführt. Für die Messungen stand ein QNX Neutrino 6.5 System mit einem Intel Core 2 Duo Prozessor (E6600, 2.4GHz, 4MB L2 Cache) mit 2GB RAM zur Verfügung. Das System zum Aufzeichnen der Messungen besteht aus einem Intel Core 2 Quad Prozessor (Q9550, 2.83GHz, 12MB L2 Cache) mit 3GB RAM. Als Betriebssystem wird Suse Linux Enterprise Desktop (Kernel Version 2.6.32.43) verwendet. Die Systeme sind mit einem 1Gbit/s Ethernet über das Institutsnetzwerk verbunden.

Die Überprüfung der erstellten Simulink-Blöcke erfolgt einzeln. Bei allen Blöcken wird die Zeit gemessen, die das Modell für den Aufruf des Simulink-Blocks benötigt. Es wird also die Zeit der mdlUpdate- und/oder mdlOutputs-Funktion gemessen. Die Programme werden mit Simulink modelliert und mit dem Real-Time-Workshop für das Zielsystem kompiliert.

Die Messungen werden mit einer Uhren-Klasse durchgeführt, die auf Prozessor-Takten basiert. Für die Messungen ist die Uhren-Klasse in die S-Functions der Blöcke integriert. Die für die Funktionen verwendete Zeit, wird an einem zusätzlich eingefügten Port der Blöcke ausgegeben. Der Zugriff auf die Modelle zur Aufnahme der Messdaten erfolgt über den External-Mode von Simulink.

Bei den Publish- und Subscribe-Blöcken wird für die Messungen eine Datenstruktur mit einer Größe von 992bit versendet. Der Request des Service-Blocks hat ebenfalls eine Größe von 992bit. Die Größe der Service-Response beträgt 32bit. Die ROS-Schnittstellenbeschreibungen befinden sich im Anhang (MiroControlTopic.msg und MiroControlService.srv).

Des Weiteren werden alle Modelle zur Messung mit einem Modelltakt von 1kHz ausgeführt. Dazu wird ein spezieller Clock-Simulink-Block verwendet, der den Modelltakt vorgibt. Der Aufruf des Subscribe- und Service-Blocks erfolgt mit der Modellrate von 1kHz. Ein Rate-Transition-Block legt die Senderate des Publishers fest.

6.1.1 Zeitmessungen des Publisher-Blocks

Der Publisher-Block veröffentlicht seine Daten mit jeder Ausführung der mdlUpdate-Funktion. Die mdlOutputs-Funktion wird nur für die Messung zur Ausgabe der gemessenen Zeit verwendet.

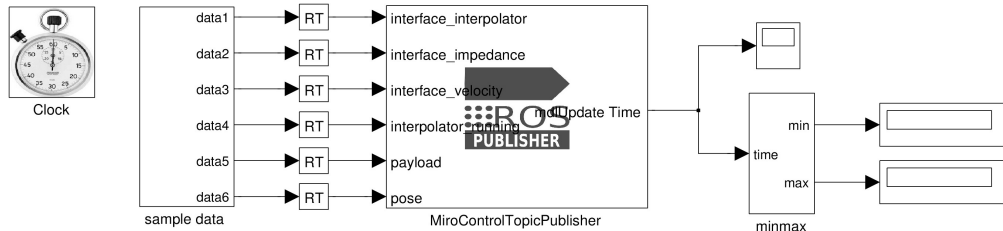


Abbildung 6.1: Simulink-Modell für die Messung des Publisher-Blocks.

In Abbildung 6.1 ist das für die Messung verwendete Simulink-Modell abgebildet. Die Clock gibt den Modelltakt vor. Der Sample-Data-Block dient zur Erzeugung von Daten, die durch Rate-Transition-Blöcke an den Publisher-Block zum Versenden weitergegeben werden. Am Ausgang des Publisher-Blocks ist zur Visualisierung und zur Aufzeichnung der Daten ein Simulink-Oszilloskop (Scope) und eine kleine Logik zum Bestimmen der minimalen und maximalen Zeit angebracht.

Mit dem Publisher-Block wurden Messungen mit 50Hz durchgeführt. Dafür begrenzt die Rate-Transition die Daten auf 50Hz. Ein Service-Aufruf aus einer Client-Anwendung benötigt ca. 100-200ms. In dieser Zeit werden bei einer Rate von 50Hz ca. 5-10 neue Zustände mit den dazugehörigen Zeitstempeln durch den Publisher an die Client-Anwendung übertragen. Aus diesem Grund reicht die gewählte Datenrate für die Aktualisierung der Benutzerinterfaces und die Synchronisation mit dem Service aus. Eine höhere Senderate steigert die Netzwerklast und bietet für die Aktualisierung einer GUI keinen Vorteil.

In Abbildung 6.2 wird das Ergebnis der Messung des Publisher-Blocks dargestellt. Bei dieser Messung ist kein Subscriber zum Empfangen der Daten auf dem Topic des Publishers angemeldet. Daher ist der Mittelwert der mdlUpdate-Funktion mit ca. 2,4 μ s deutlich geringer als bei der Messung mit angemeldetem Subscriber (siehe Abbildung 6.3). Hier beträgt der Mittelwert ca. 14,2 μ s.

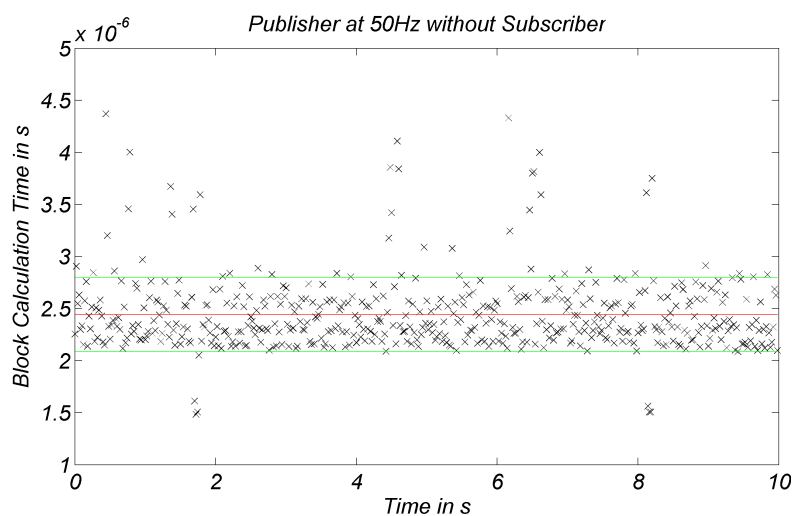


Abbildung 6.2: Messung des Publisher-Blocks bei einer Rate mit 50Hz, ohne Subscriber. Der Mittelwert ist rot und die Standardabweichung grün dargestellt (Mittelwert $2,4\mu\text{s}$, Maximalwert $4,4\mu\text{s}$).

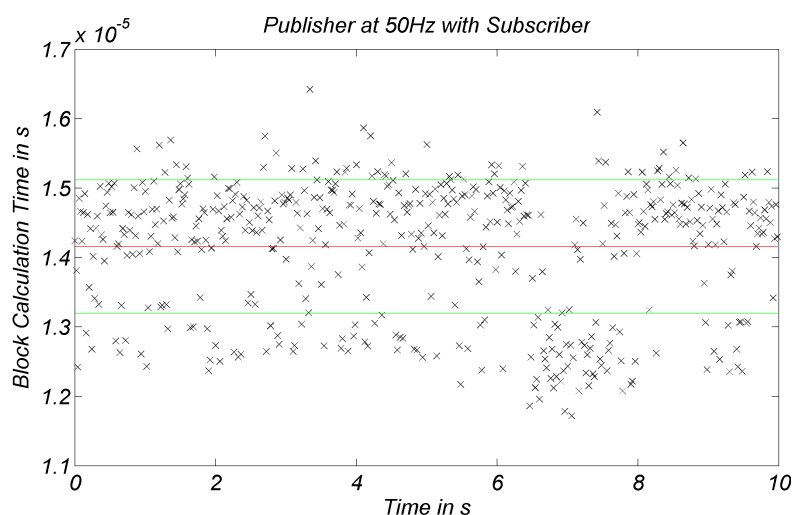


Abbildung 6.3: Messung des Publisher-Blocks bei einer Rate mit 50Hz und einem Subscriber (Mittelwert $14,2\mu\text{s}$, Maximalwert $16,4\mu\text{s}$).

Die höhere Bearbeitungszeit der `mdlUpdate`-Funktion bei einem Publisher mit einem angemeldeten Subscriber ist darauf zurückzuführen, dass in diesem Fall eine Serialisierung der Daten für die Netzwerkübertragung stattfindet [58]. Ist kein Subscriber angemeldet oder ein Subscriber im selben Prozess, findet keine Serialisierung statt. Als Subscriber wurde bei dieser Messung das ROS-Tool „`rostopic`“ verwendet. Mit diesem Tool lassen sich die Daten eines ROS-Topics und die Senderate des Publishers direkt auf einer Konsole anzeigen.

6.1.2 Zeitmessungen des Subscriber-Blocks

Der Subscriber-Block empfängt die Daten mit Hilfe einer Callback-Funktion für den ROS-Subscriber. Die Callback-Funktion kopiert die empfangenen Daten in einen Puffer und teilt dem Block mit, dass neue Daten zum Ausgeben eingetroffen sind. In der mdlOutputs-Funktion werden die Daten aus dem Puffer gelesen und an die Ports des Blocks geschrieben. Die Callback-Funktion wird in einem zusätzlichen Thread ausgeführt dadurch ist nur die Messung der mdlOutputs-Funktion erforderlich. Der Messaufbau ist analog zu dem des Publishers aufgebaut und das Modell wird ebenfalls mit einer Rate von 1kHz ausgeführt (siehe Abbildung 6.4).

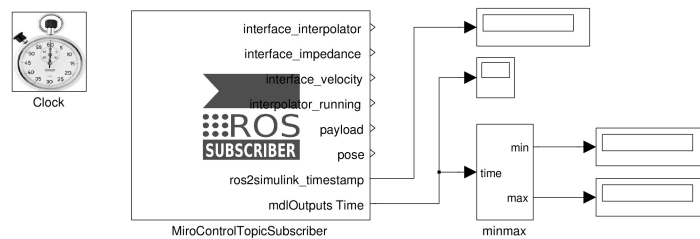


Abbildung 6.4: Simulink-Modell für die Messung des Subscriber-Blocks.

In Abbildung 6.5 ist das Ergebnis der Messung für einen Subscriber ohne einen sendenden Publisher gezeigt. Der Mittelwert der Messung beträgt ca. 140ns. Die größte Abweichung liegt bei ca. 280ns. Empfängt der Subscriber keine Daten, wird der Empfangspuffer nicht ausgelesen und die Pufferdaten nicht an die Ausgänge des Blocks geschrieben.

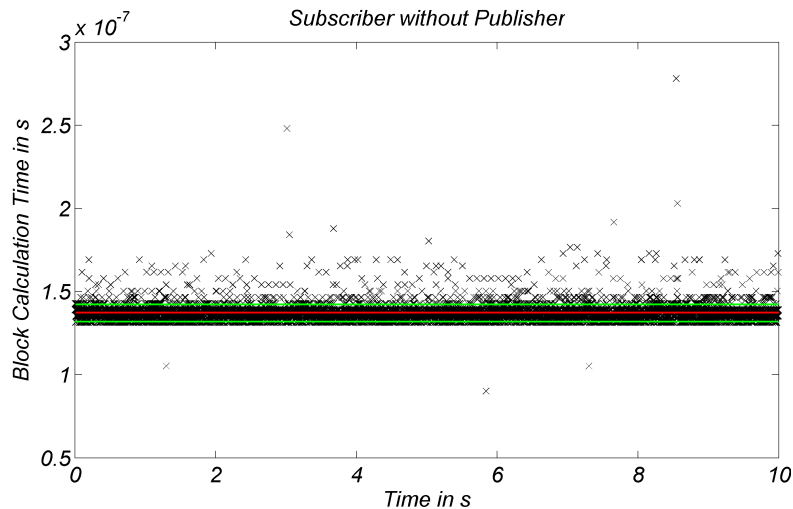


Abbildung 6.5: Messung des Subscriber-Blocks ohne Publisher (Mittelwert 140ns, Maximalwert 280ns).

In Abbildung 6.6 ist die Messung eines Subscribers, zu dem Daten mit einer Rate von 50Hz gesendet werden, abgebildet. Die Daten werden für diese Messung mit dem ROS-Tool

„rostopic“ versendet. In diesem Fall werden die empfangenen Werte in der mdlOutputs-Funktion an die Ausgänge des Blocks geschrieben. Beim Eintreffen von Daten lässt sich ein Anstieg der Bearbeitungszeit auf ca. 600ns erkennen.

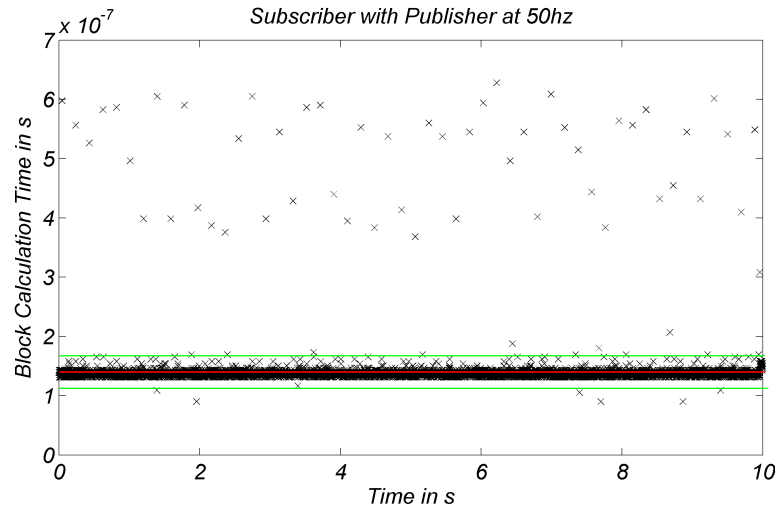


Abbildung 6.6: Messung des Subscriber-Blocks mit einem Publisher bei einer Rate von 50Hz (Mittelwert 140ns, Maximalwert 630ns).

Der geringe Anstieg lässt sich dadurch erklären, dass der Subscriber-Block die Deserialisierung der Daten nicht selbst durchführen muss. Die Deserialisierung findet in einem separaten Thread statt, der vom ROS-Spinner bei der Initialisierung der Blöcke gestartet wird. Der Subscriber-Block liest in der mdlOutputs-Funktion nur den Puffer aus und schreibt die Daten an die Blockausgänge.

6.1.3 Zeitmessungen des Service-Blocks

Der Service-Block bearbeitet die empfangenen Daten, wie auch der Subscriber-Block, mit Hilfe eines durch den ROS-Spinner gestarteten Threads. Die Callback-Funktion schreibt die empfangenen Daten in einen Puffer und in der mdlOutputs-Funktion werden die Werte an die Blockausgänge geschrieben. Durch den in Kapitel 4.1.4 beschriebenen Enable-Ready-Mechanismus wird zudem mit der mdlUpdate-Funktion der Enable-Port des Blocks abgefragt.

Bei diesem Block wird also die Zeit, die Simulink für die mdlUpdate- und mdlOutputs-Funktion aufwendet, gemessen. Die Zeiten werden addiert und an einem für die Messungen angebrachten Ausgang am Block ausgegeben. Auch hier ist das Modell für den Messaufbau ähnlich dem des Subscribers und wird mit 1kHz ausgeführt (siehe Abbildung 6.7).

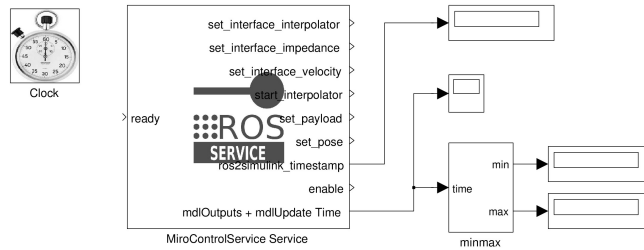


Abbildung 6.7: Simulink-Modell für die Messung des Service-Blocks.

In Abbildung 6.8 ist eine Zeitmessung des Service-Blocks mit Aufrufen durch einen Clienten zu sehen. Treffen keine Anfragen beim Service ein, werden die mdlOutputs- und mdlUpdate-Funktionen sofort wieder verlassen. Die Service-Aufrufe lassen sich in den erhöhten Messpunkten wieder erkennen.

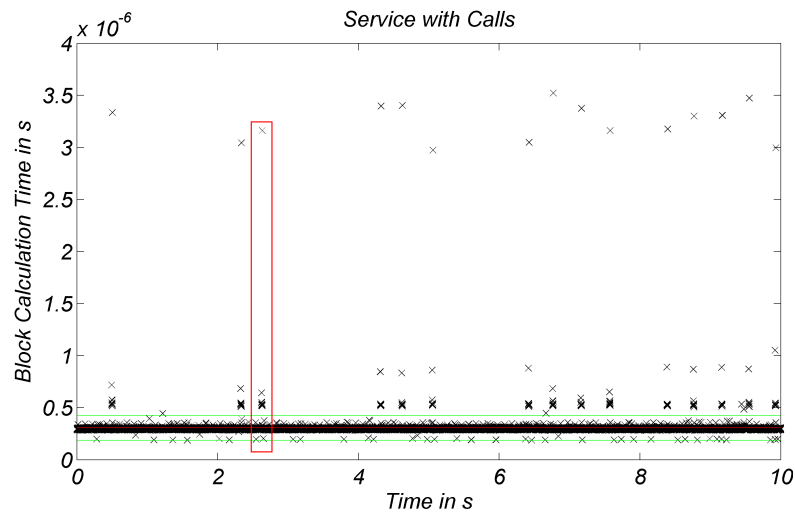


Abbildung 6.8: Messung des Service-Blocks mit Aufrufen durch einen Clienten. Der dritte Aufruf ist rot markiert. (Insgesamt 15 Aufrufe)

In Abbildung 6.9 ist eine Vergrößerung des dritten Service-Aufrufs aus Abbildung 6.8 abgebildet, die einen einzelnen Serviceaufruf zeigt. Beim ersten erhöhten Messpunkt werden die Daten aus dem Puffer an die Ausgänge des Blocks geschrieben. Danach wird in der mdlUpdate-Funktion auf das Ready-Signal gewartet. Nach zehn Aufrufen des Blocks durch das Simulink-Modell wird der Timeout ausgelöst und ein Signal an den Callback-Thread gesendet. Das Signal entsperrt den Callback-Thread und löst das Senden der Response an die Client-Anwendung aus.

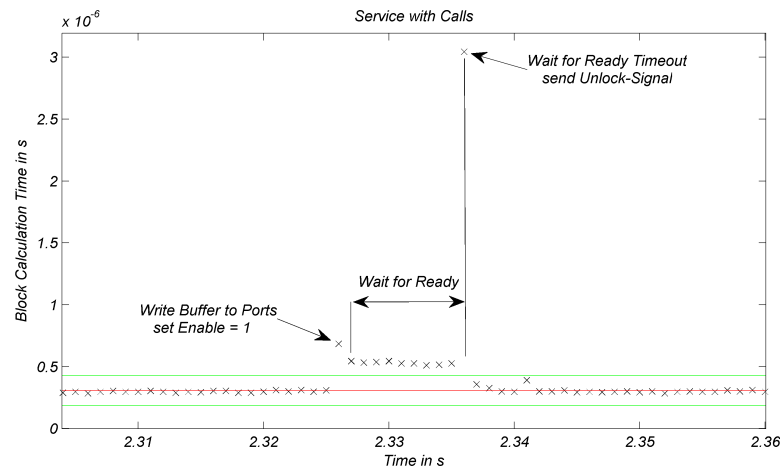


Abbildung 6.9: Vergrößerter Ausschnitt des Service-Aufrufs aus Abbildung 6.8.

Die meiste Zeit wird durch das Senden des Signals zum Endsperren des Callback-Threads beansprucht. Hierfür wurde eine Kondition-Variable verwendet. Werden keine Anfragen vom Service verarbeitet, liegt die Berechnungszeit bei ca. 300ns. Dies entspricht ungefähr der doppelten Zeit des Subscriber-Blocks, wenn dieser keine Daten verarbeiten muss. Die erhöhte Zeit gegenüber dem Subscriber im Leerlauf lässt sich dadurch erklären, dass beim Service-Block zwei Funktionen (mdlOutputs und mdlUpdate) gemessen werden.

6.1.4 Diskussion

Die entworfenen Simulink-Blöcke sind für die Anbindung von Nutzerschnittstellen entworfen worden. Die Datenraten für die Messungen wurden dementsprechend gewählt. Die durchgeführten Messungen zeigen in ihren Ergebnissen ein deterministisches Verhalten. Die meiste Zeit wird im Publisher-Block verwendet, da die versendeten Daten direkt im Thread des laufenden Modells serialisiert werden.

Um die Grenzen des Systems zu bestimmen, werden weitere Messungen mit den Publisher-, Subscriber- und Service-Blöcken durchgeführt. Dazu wird die Datenrate bei den Publisher- und Subscriber-Blöcken für die Messungen auf 1kHz erhöht.

Die Messung des Publishers bei einer Senderate von 1kHz zeigt nur eine leichte Erhöhung des Mittelwertes von der 50Hz Messung mit ca. 14µs auf ca. 15µs (siehe Abbildung 6.10). Allerdings fällt bei dieser Messung ein stark abweichender Wert mit ca. 25µs bei Sekunde 6,5 auf.

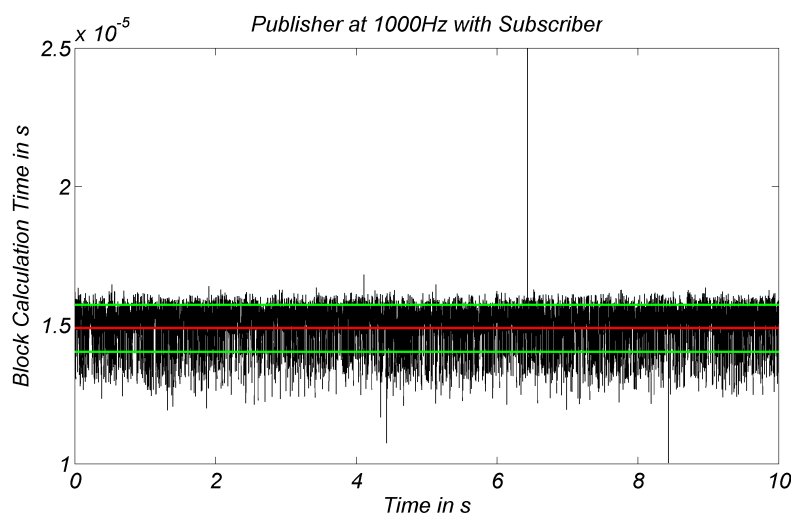


Abbildung 6.10: Messung des Publisher-Blocks bei einer Rate mit 1kHz und einem angemeldeten Subscriber (Mittelwert $15\mu\text{s}$, Maximalwert $25\mu\text{s}$).

Auch bei der Messung des Subscriber-Blocks, zu dem Daten mit einer Rate von 1kHz gesendet werden, sind ähnliche Peaks zu beobachten (siehe Abbildung 6.11). Die Abweichungen betragen hier ca. $11\mu\text{s}$.

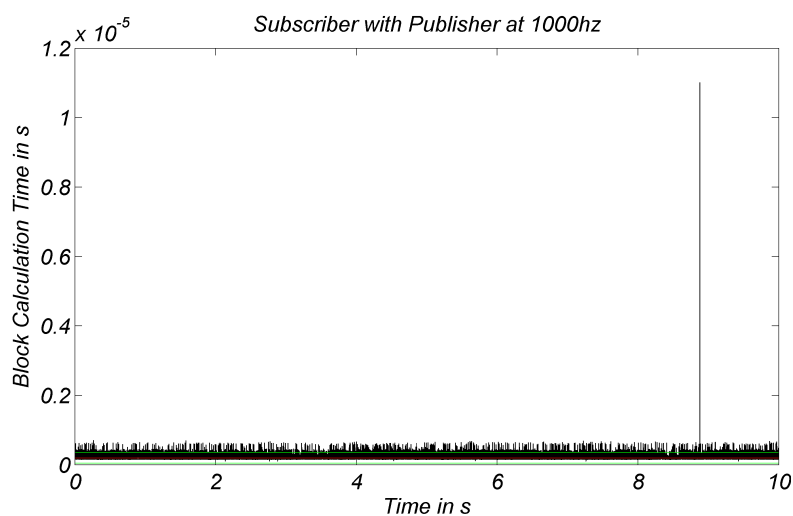


Abbildung 6.11: Messung des Subscriber-Blocks mit einem sendenden Publisher bei einer Rate von 1kHz (Mittelwert 190ns , Maximalwert $11\mu\text{s}$).

Um zu überprüfen, ob die Abweichungen auch beim Service-Block auftreten, wurden für diesen ebenfalls weitere Messungen aufgezeichnet. In Abbildung 6.12 ist ein Ausschnitt einer längeren Messung des Service-Blocks ohne Client-Aufrufe zu sehen. Der Ausschnitt zeigt ebenso einen Peak mit ca. $11\mu\text{s}$.

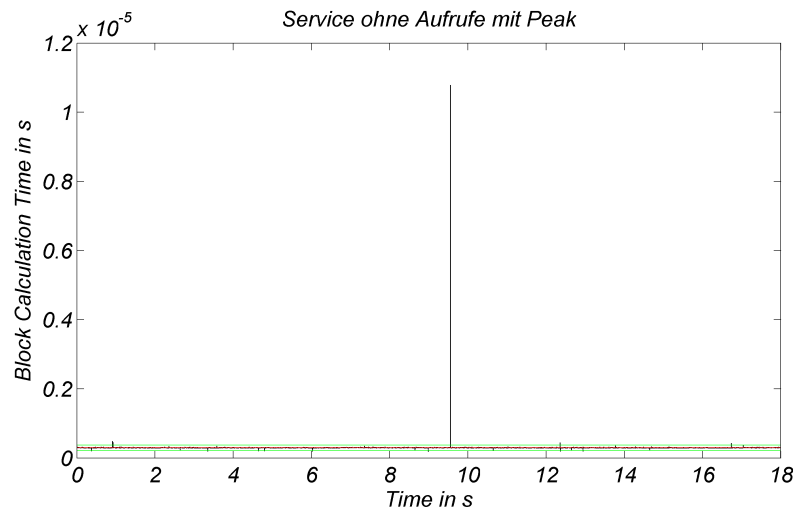


Abbildung 6.12: Messung des Service-Blocks ohne Aufrufe (Mittelwert 296ns, Maximalwert 10,8 μ s).

Um dem Problem auf den Grund zu gehen, wurde ein S-Function-Block erstellt, der nur die Zeit der mdlOutputs-Funktion misst. Eine Last wird durch eine For-Schleife simuliert, die so gewählt ist, dass sie den realen Blöcken ähnelt. Die Messungen ergeben einen Mittelwert von ca. 680ns (siehe Abbildung 6.13). Bei dieser Messung treten ebenso sporadisch Abweichungen auf. Die Abweichung liegt hier bei ca. 11 μ s.

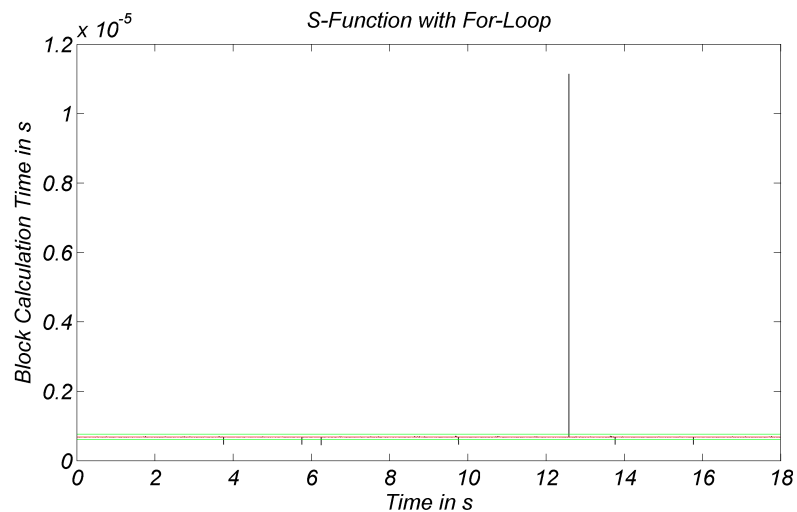


Abbildung 6.13: Messung der S-Function mit For-Schleife (Mittelwert 680ns, Maximalwert 11 μ s).

Alle Messungen der Blöcke zeigen Abweichungen in der Berechnungszeit von ca. 10-15 μ s. Dass der Ursprung der Abweichungen nicht in den ROS-Bibliotheken liegt, wird mit der Messung der S-Function, die nur eine For-Schleife enthält, gezeigt. Alle Programme wurden unter QNX 6.5 mit einer erhöhten Priorität (200) ausgeführt. Allerdings wurde für alle Messungen der External-Mode von Simulink zur Aufzeichnung und Übertragung der

Messwerte verwendet. Ein Anhaltspunkt für die Abweichungen ist also die Verwendung des External-Modes von Simulink. Zudem ist der QNX-Rechner mit dem Institutsnetzwerk verbunden. Dies lässt auch vermuten, dass das System auf eingehenden Netzwerkverkehr reagieren muss. Somit können die Peaks von Interrupt-Behandlungen ausgelöst werden, die den Simulink Prozess unterbrechen und einen Kontextwechsel verursachen.

Damit sichergestellt werden kann, dass das entwickelte System in den Regelungssystemen einsetzbar ist, wurden zusätzlich Langzeitmessungen von ca. einer halben Stunde durchgeführt. Bei diesen Messungen wurde die maximale Zeit, die der jeweilige Block im Modell benötigt, ermittelt. Die Ergebnisse sind in Tabelle 6.1 dargestellt. Die Senderate wurde beim Publisher und Subscriber jeweils auf 50Hz gesetzt um einen normalen Einsatz zu simulieren. Der Service-Block wurde sporadisch durch eine GUI aufgerufen.

Simulink-Block	Maximale Zeit in μ s
Publisher	29,96
Subscriber	11,46
Service	10.96

Tabelle 6.1: Maximale Ausführungszeit der Blöcke (mdlUpdate und/oder mdlOutputs) über den Zeitraum von ca. einer halben Stunde.

Die Ergebnisse weichen nur gering von denen aus den Kurzzeitmessungen ab. Es kann also davon ausgegangen werden, dass diese Werte unter der Testkonfiguration die Maximalwerte sind.

Die Simulink-Blöcke sind für die Anbindung von Nutzerschnittstellen entwickelt worden was bedeutet, dass die Senderaten bei einer Größenordnung von 1-50Hz liegen sollten und für die Benutzerinterfaces keine großen Datenmengen nötig sind. Aus diesem Grund können die Blöcke in Simulink in eine langsamere Rate gesetzt werden. Die Blöcke werden dadurch nicht so häufig aufgerufen und zudem durch Simulink in einen Thread verlagert. Somit wirken sich die Schwankungen nicht auf den Haupt-Thread des Modells aus.

Abschließend kann gesagt werden, dass die entwickelten Simulink-Blöcke unter der getesteten Konfiguration die Echtzeitfähigkeit des Systems einhalten, solange die Zeiten aus den Langzeitmessungen nicht überschritten werden.

6.2 Integration der Schnittstellen in die MIRO-Einarmapplikation

Im vorherigen Teil des Kapitels wurde gezeigt, dass die entwickelten Simulink-Blöcke zur Anbindung an ROS die Echtzeiteigenschaften nicht beeinträchtigen. Mit Hilfe des entwickelten Codegenerators zur automatischen Erzeugung der Simulink-Blöcke wird in diesem Teil der Arbeit die Implementierung des in Kapitel 4.2 erarbeiteten Konzepts für die Einarmapplikation beschrieben.

6.2.1 Entwurf der Schnittstellen für die MIRO-Einarmapplikation

In Kapitel 4.2 wurden die Schnittstellen des ursprünglichen aRDnet-Kommunikations-Blocks in kleinere Schnittstellen aufgeteilt. Die genaue Definition der Schnittstellen wird an dieser Stelle der Arbeit weiter erläutert. In Abbildung 6.14 wird der aRDnet-Kommunikations-Block der Einarmapplikation gezeigt. An diesem Beispiel lässt sich gut erkennen, dass bei der Verwendung von kleineren Schnittstellen weniger Signale durch das gesamte Modell zu einem zentralen Kommunikationsblock geleitet werden müssen.



Abbildung 6.14: aRDnet-Kommunikationsblock für die MIRO Einarmapplikation.

Mit dem aRDnet-Kommunikationsblock werden, der Regler und die HAL des MIRO-Modells für die Einarmapplikation gesteuert. Aus diesem Grund wird die Kommunikationsschnittstelle in zwei Hauptteile gespalten, die jeweils ein Paar aus Publisher und Subscriber bilden. Zudem

ist für die Übertragung der Geschwindigkeitsvorgabe in das Modell, durch die SpaceMouse des TMCDs, ein Subscriber und zur Ausgabe der aktuellen Geschwindigkeit aus dem Modell ein Publisher definiert. Die Geschwindigkeit wird nicht mit einem Service in das Modell übertragen, da hierfür aus Sicherheitsgründen eine kontinuierliche Datenquelle erforderlich ist. Serviceaufrufe sind dafür zu langsam. Aus der Aufteilung des aRDnet Blocks ergeben sich fünf Schnittstellenbeschreibungen für die neuen ROS basierten Schnittstellen. Eine Übersicht der erstellten Schnittstellen ist in Tabelle 6.2 dargestellt. Die vollständigen Beschreibungsdateien befinden sich im Anhang.

Datei	Beschreibung
MiroHalService.srv	Service Beschreibung zur Steuerung des HAL-Blocks für den MIRO-Roboter
MiroHalTopic.msg	Publisher und Subscriber Beschreibung für die Ausgabe des HAL-Zustands
MiroControlService.srv	Service Beschreibung für die Steuerung des MIRO-Reglers
MiroControlTopic.msg	Publisher und Subscriber Beschreibung für die Ausgabe des Regler-Zustands
MiroVelocityTopic.msg	Publisher und Subscriber Beschreibung für die Eingabe der Geschwindigkeit durch die SpaceMouse und die Ausgabe der aktuellen Geschwindigkeit des Roboters

Tabelle 6.2: Dateien für die Schnittstellendefinitionen der MIRO-Einarmapplikation.

Die Datentypen der Schnittstellen wurden weitgehend direkt aus dem aRDnet-Block übernommen. Für die Nachrichten-Definitionen wurde das „miro_msgs“ ROS-Paket erstellt. Mit dem im ROS-Buildsystem integrierten Codegenerator werden aus den erstellten Nachrichten-Definitionen die C++ und Python-Klassen generiert. Mit dem in dieser Arbeit entwickelten Codegenerator werden die Simulink-Blöcke mit ROS-Anbindung erzeugt. Für jede srv-Datei wird ein Service-Block und für jede msg-Datei ein Publisher- und Subscriber-Block generiert. Die generierten Blöcke sind in Abbildung 6.15 abgebildet.

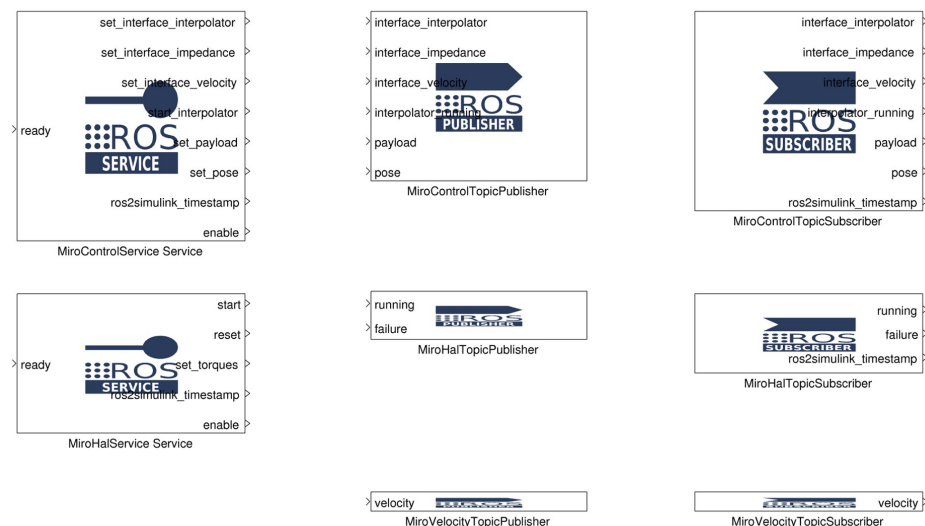


Abbildung 6.15: Mit dem Codegenerator erstellte Simulink-Blöcke.

Der Codegenerator erzeugt automatisch die Topic-Namen wie in Kapitel 5.7 beschrieben und trägt diese als Parameter in die Simulink-Blöcke ein. Für die hier erstellten Blöcke wurden folgende Topic-Namen erstellt:

- Service-Topics
 - `/ros_nodename/miro_msgs/MiroControlService`
 - `/ros_nodename/miro_msgs/MiroHalService`
- Publish-Subscribe-Topics
 - `/ros_nodename/miro_msgs/MiroControlTopic`
 - `/ros_nodename/miro_msgs/MiroHalTopic`
 - `/ros_nodename/miro_msgs/MiroVelocityTopic`

Der ROS-Node-Name kann für ein Simulink-Modell mit einer Matlab-Variable gesetzt werden. Es können mehrere Modelle mit den gleichen Schnittstellen erstellt und parallel ausgeführt werden. Die Modelle müssen sich allerdings in den Nodennamen unterscheiden. Clienten können sich dann anhand dieser Namen mit den Modellen verbinden, da die Struktur der Topicnamen innerhalb eines Modells transparent ist. Die Flexibilität des Systems wird hierdurch gesteigert.

6.2.2 Integration der Simulink-Blöcke in das Regelungsmodell

Im vorherigen Teil der Arbeit wurden die ROS-Simulink-Blöcke für die Einarmapplikation definiert und erstellt. An dieser Stelle der Arbeit wird gezeigt, wie die Schnittstellen in dem Simulink-Modell für die Einarmapplikation integriert werden und welche Vorteile sie bieten.

Der aRdnet-Kommunikationsblock in der ursprünglichen Version der Einarmapplikation liegt auf der obersten Ebene des Simulink-Modells. Der Regler befindet sich eine Ebene und der HAL-Block des MIRO-Roboters zwei Ebenen darunter (siehe Abbildung 6.16).

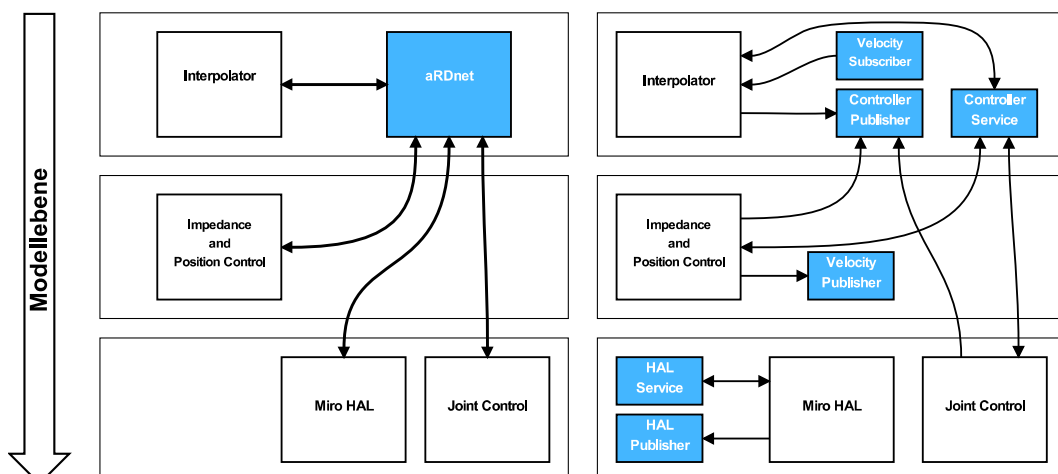


Abbildung 6.16: Vereinfachte Darstellung der Modellebenen in der Einarmapplikation. Links die alten und rechts die neuen Schnittstellen.

In der ursprünglichen Einarmapplikation existiert nur ein Kommunikationsblock mit einer breiten Schnittstelle, die auf alle Teile des Modells zugreift. Aus diesem Grund werden zum Beispiel die Signale zur Steuerung der MIRO-HAL durch alle Ebenen bis zum Kommunikationsblock geleitet. Dazu findet entweder eine Erweiterung der Subsysteme um zusätzliche Ports statt oder es werden Goto-Blöcke verwendet, mit denen Signale ohne Leitungen über Links weitergegeben werden können. Die Regelungssysteme werden dadurch unübersichtlich und sind schlecht zu warten.

Durch die Aufspaltung der Schnittstellen in kleinere ROS-Blöcke kann ein Teil der Blöcke direkt dort platziert werden, wo sie benötigt werden. Beispielsweise wurden die Kommunikations-Blöcke für die MIRO-HAL in der selben Ebene platziert, in der sich auch der HAL-Block befindet. Dadurch werden weniger Signalleitungen durch das gesamte Modell geführt, was wiederum eine Steigerung der Übersichtlichkeit zur Folge hat.

Das Modell wird mit einer Rate von 1kHz berechnet. Die Subscriber- und Service-Blöcke werden im Modelltakt ausgeführt. Die Publisher-Blöcke werden mit Hilfe von Rate-Transission-Blöcken auf 50Hz gedrosselt.

Der Codegenerator erzeugt die Simulink-Blöcke automatisch aus den definierten Schnittstellen. Die Schnittstellen werden für alle Anwendungen mit der ROS-IDL definiert und können direkt in Client-Anwendungen genutzt werden.

6.2.3 Client-Anwendungen für die MIRO-Einarmapplikation

Mit der Erzeugung der Schnittstellen und der Integration der generierten Simulink-Blöcke in das Regelungsmodell wurde ein ROS-Node erzeugt. Auf dieses Node kann nun mit verschiedenen Clienten zugegriffen werden, indem die erzeugten Schnittstellen genutzt werden. Für die Beispielanwendung wurden zwei Client-Anwendungen nach dem in Kapitel 4.2 entwickelten Konzept implementiert. Dazu gehören eine GUI-Anwendung und eine Anwendung für das TMCD.

Die MIRO-Singlearm-GUI dient zur Steuerung der Regelungsanwendung (siehe Abbildung 6.17). Zudem wird der aktuelle Zustand der Regelungsanwendung in das GUI übertragen und angezeigt. Benutzerbefehle werden mit Serviceaufrufen in das Simulink-Modell übertragen und der Empfang der Zustände erfolgt durch Subscriber. In der Anwendung sind Service-Clients für den MIRO-HAL-Block und den MIRO-Control-Block integriert worden. Zudem sind Subscriber für die HAL, den Controller und die aktuelle Robotergeschwindigkeit integriert.



Abbildung 6.17: MIRO Singlearm GUI mit ROS-Anbindung.

Als zweite Client-Anwendungen wurde eine Anwendung für das TMCD mit den erzeugten Schnittstellen erstellt. Die Anwendung integriert, wie auch die GUI-Anwendung, die Schnittstellen zur HAL und zum Regler (siehe Abbildung 6.18).

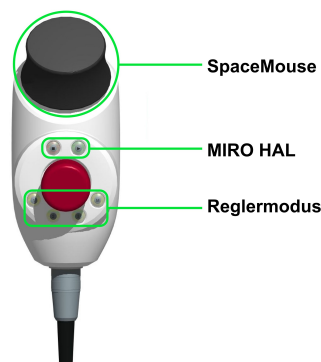


Abbildung 6.18: Belegung des TMCDs für die Client-Anwendung

Jedoch sind durch die begrenzte Anzahl an Eingabeknöpfen nicht alle Funktionen abrufbar. Mit der TMCD-Anwendung ist es möglich, die MIRO-HAL zu Starten und zu Stoppen. Außerdem kann der Reglermodus umgeschaltet werden. Weiter wurde ein Publisher zum Übertragen der Daten der SpaceMouse in das Modell integriert. Er sendet Geschwindigkeitsvorgaben, wenn sich der Regler im Velocity-Modus befindet. Der Zustand der HAL und der gewählte Reglermodus werden mit Hilfe der LEDs am TMCD dargestellt.

6.2.4 Test der überarbeiteten MIRO-Einarmapplikation

An dieser Stelle soll nun anhand eines Tests gezeigt werden, welche Vorteile das entwickelte Konzept mit ROS als ausgewählte Middleware bietet. Auch für diesen Test stand, wie schon bei den Zeitmessungen der Kommunikationsblöcke, ein QNX Neutrino 6.5 System mit einem Intel Core 2 Duo Prozessor (E6600, 2.4GHz, 4MB L2 Cache) mit 2GB RAM zur Verfügung. Darüber hinaus wurde ein Linux Rechner mit Suse Linux Enterprise Desktop (Kernel Version 2.6.32.43) und einem Intel Core 2 Quad Prozessor (Q9550, 2.83GHz, 12MB L2 Cache) mit 3GB RAM verwendet. Die Systeme sind mit einem 1Gbit Ethernet über das Institutsnetzwerk verbunden.

Das MIRO-Simulink-Modell für die Einarmapplikation wird für diesen Aufbau mit dem Real-Time-Workshop für QNX 6.5 kompiliert und auf dem QNX Rechner ausgeführt. In diesen Versuchen wird das Simulink-Modell ohne angeschlossene Roboterhardware ausgeführt. Ein Simulink-Block dient zur Simulation der Roboterhardware. Der ROS-Nodename für das Simulink-Modell lautet „/miro1“. Auf einem Linux Rechner läuft der ROS-Master, zwei Instanzen der GUI-Anwendung und ein TMCD-Client (siehe Abbildung 6.19).

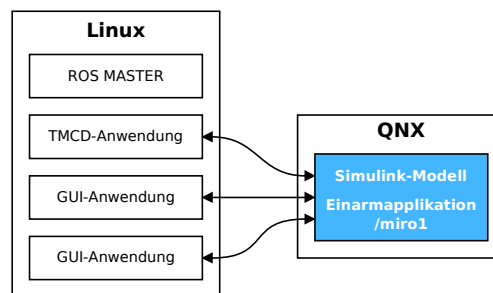


Abbildung 6.19: Systemkonfiguration mit mehreren Client-Anwendungen auf einem Linux-System und dem Simulink-Modell auf einem QNX-Echtzeitsystem.

Damit sich das Node des Simulink-Modells auf dem QNX Rechner mit dem ROS-Master auf dem Linux-Rechner verbindet, muss die ROS-Master-URI (Uniform Resource Identifier) in Form einer Umgebungsvariable⁶ auf dem QNX-Rechner gesetzt werden. Nachdem dies durchgeführt wurde, kann das Simulink-Modell auf dem QNX-System gestartet werden und ist mit Hilfe der ROS-Tools und den erstellten Client-Anwendungen auffindbar. Mit der in dieser Arbeit erstellten GUI-Anwendung kann das Simulink-Modell unter dem ROS-Nodnamen „/miro1“ ausgewählt und eine Verbindung aufgebaut werden. In der TMCD-Anwendung wird der Nodename mit einem Kommandozeilenargument übergeben. Nachdem die Client-Anwendungen gestartet wurden und sie sich zum Simulink-Modell verbunden haben, erhalten sie Updates des Systemzustands und sind einsatzbereit.

Eine Übersicht über das aktuell laufende System wird mit dem Konsolen-Tool „rostopic“ und dem graphischen Tool „rxgraph“ dargestellt. In Abbildung 6.20 wird der Systemaufbau mit „rxgraph“ aufgezeigt.

⁶ Der ROS-Master wird in folgender Form als Umgebungsvariable gesetzt:
ROS_MASTER_URI=http://Hostname:Portnummer/

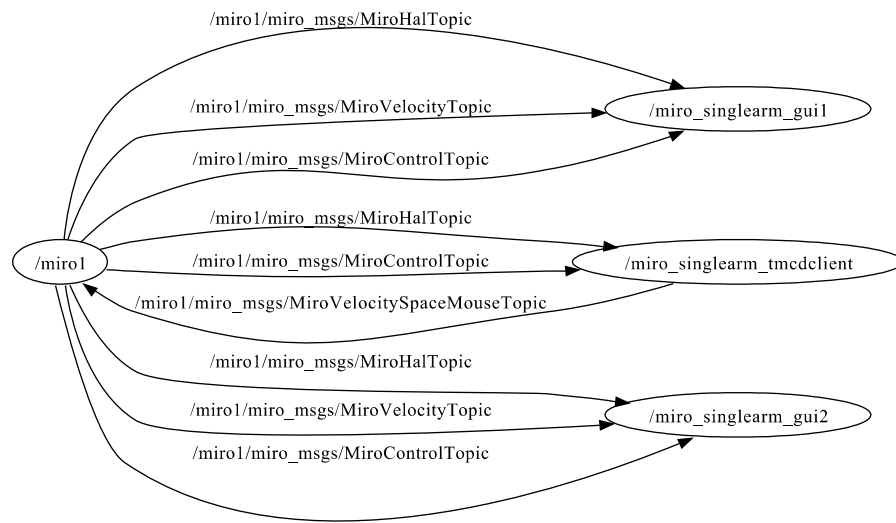


Abbildung 6.20: Konfiguration der MIRO Einarmapplikation mit einem MIRO-Modell, zwei GUI-Anwendungen und TMCD-Clients dargestellt mit dem ROS-Tool rxgraph.

Zudem ist es mit ROS-Konsolen-Tools möglich, direkt auf einer Konsole Topic-Inhalte auszugeben oder zu versenden und Service-Aufrufe auszuführen. Dies bietet einen großen Vorteil beim Debuggen des Systems und der Schnittstellen gegenüber dem alten System.

In einem zweiten Versuch wird ein weiteres Simulink-Modell im Simulationsmodus gestartet (siehe Abbildung 6.21). Der Systemaufbau ermöglicht ein komfortables Auswählen der gestarteten Modelle in den Client-Anwendungen. Einzig die ROS-Nodennamen der Modelle müssen unterschiedlich sein.

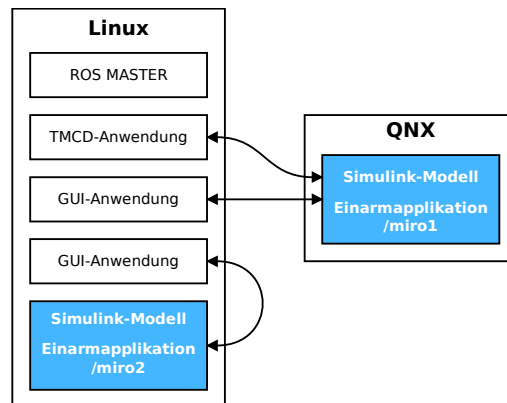


Abbildung 6.21: System mit mehreren Client-Anwendungen und einem Simulink-Modell auf einem Linux-System und einem zusätzlichen Modell auf einem QNX-Echtzeitsystem.

In Abbildung 6.22 wird ein laufendes System mit zwei Simulink-Modellen gezeigt. Das erste wird in Simulink im Simulations-Modus und das zweite auf dem QNX-Echtzeitsystem

ausgeführt. Mit den Client-Anwendungen kann flexibel zwischen den laufenden Modellen gewechselt werden.

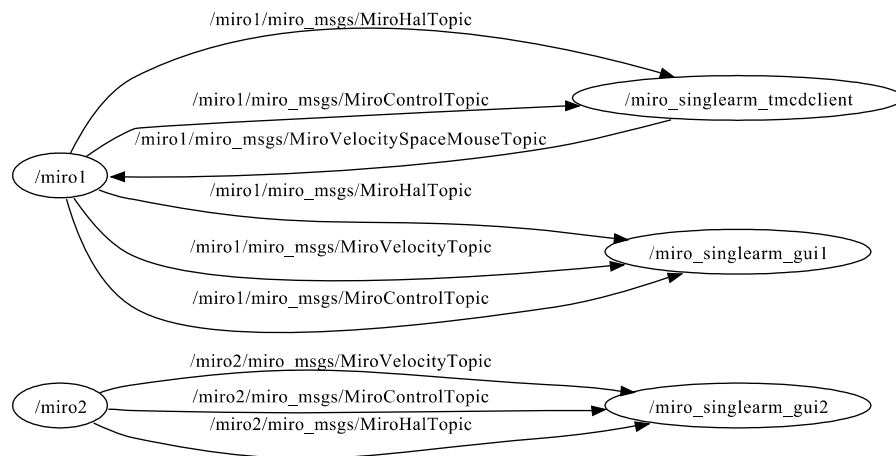


Abbildung 6.22: Konfiguration der MIRO Einarmapplikation mit dem ROS-Tool rxgraph dargestellt. Zwei laufende Modelle und drei verbundene Client-Anwendung.

6.2.5 Ergebnisse

Das entwickelte Konzept hat sich als flexibel erwiesen. Es können mehrere Client-Anwendungen und Modelle auf verschiedenen Rechnern gestartet werden. Einzig die ROS-Nodenamen der Anwendungen müssen unterschiedlich sein und der ROS-Master muss zuvor auf jedem beteiligten Rechner angegeben werden. Die Verwendung von universellen Schnittstellen in den beiden verschiedenen Client-Anwendungen ermöglicht einen einheitlichen Zugriff auf das Simulink-Modell. Die Mechanismen zur Synchronisation erhalten die Konsistenz zwischen den Client-Anwendungen und somit den sicheren Betrieb des Systems. Die Simulink-Modelle können im Simulations-Modus auf einem Linux-System oder auf einem QNX-Echtzeitsystem ausgeführt werden. Mit den Client-Anwendungen besteht die Möglichkeit, transparent Verbindungen zu den Modellen aufzubauen und auch zwischen diesen zu wechseln.

Die Aufteilung des ursprünglichen aRDnet-Kommunikations-Blocks in die kleineren ROS-Blöcke erlaubt eine Integration der Blöcke in der Modellebene, in der die Kommunikationsschnittstellen benötigt werden. Die Modelle werden hierdurch übersichtlicher und es müssen nicht alle benötigten Leitungen durch mehrere Modellebenen zu einem Kommunikations-Block geleitet werden.

Des Weiteren ist die Robustheit des Systems gesteigert worden. Fällt ein Modell aus, müssen die Clients nicht neu gestartet werden. Es reicht ein Neustart des Simulink-Modells. Die Client-Anwendungen nehmen die Verbindung automatisch wieder auf. Dies gilt auch im Falle eines Absturzes einer Client-Anwendung. Die einzige Schwachstelle ist der zentrale ROS-Master für die Verwaltung der Nodes. Stürzt der ROS-Master ab müssen alle Nodes neugestartet werden.

7 Weiterführende Arbeiten und Ausblick

Mit dem Ausblick als letztes Kapitel dieser Arbeit soll ein Überblick über die weiterführenden Arbeiten im Zusammenhang mit dem entwickelten Konzept zur Anbindung von Nutzerschnittstellen an die verteilte Regelungsanwendung gegeben werden. Das Kapitel wird in weiterführende Arbeiten, welche Verbesserungsmöglichkeiten und Erweiterungen beinhalten, sowie in einen Ausblick für langfristige Ziele unterteilt.

7.1 Weiterführende Arbeiten

Die ROS-IDL erlaubt die Definition von komplexen mehrfach verschachtelten Datenstrukturen. Der in dieser Arbeit implementierte Codegenerator kann diese Strukturen in Simulink-Blöcken darstellen. Die Strukturen werden allerdings bis auf ihre Basisdatentypen aufgelöst. Eine Verbesserungsmöglichkeit besteht darin, diese Datenstrukturen an den Blöcken als Simulink-Bus-Objekte darzustellen. Bus-Objekte bieten die Möglichkeit Signalbusse zu erstellen, die komplexe Datenstrukturen direkt in einer Signalleitung bündeln. Damit würde die Übersichtlichkeit in den Simulink-Modellen weiter gesteigert werden.

Der ROS-Master enthält einen Parameterserver mit dem, ähnlich wie in einer Datenbank, systemweit Parameter ablegbar und in allen Nodes abrufbar sind. Die im Parameterserver abgelegten Daten können in Konfigurationsdateien gesichert und bei Bedarf wieder geladen werden. Es würde sich anbieten, Simulink-Blöcke zu erstellen, die Einstellungsdaten aus dem Parameter Server abfragen. So könnten Standardeinstellungen für ganze Systemkonfigurationen abgelegt werden, die beim Start des Systems automatisch abgerufen werden.

7.2 Ausblick

Mit der Erzeugung der schmalen universellen Schnittstellen und der Platzierung der generierten Kommunikations-Blöcke in den Simulink-Modellen an den geeigneten Positionen wurde gezeigt, dass die Übersichtlichkeit der Modelle gesteigert wird. Dies lässt sich besonders gut an den Simulink-Blöcken erkennen, die für die Kommunikation zur HAL des MIROs zuständig sind. Ein weiterer Schritt zu einer aufgeräumteren Architektur wäre die Integration der HAL-Schnittstellen direkt in die HAL-Software ohne Umweg über Simulink. Durch die Transparenz der ROS-Middleware können die Schnittstellen in HAL verlagert werden. Client-Anwendungen können bei Einhaltung der vorhandenen Schnittstellenbeschreibungen ohne Modifikationen weiter verwendet werden.

Zudem ist während dieser Arbeit eine offizielle ROS Implementierung in Java erschienen, die mit der nächsten ROS-Version veröffentlicht wird und kompatibel mit Android-Geräten ist [59]. Hiermit werden sich mobile Enbgeräte, wie etwa Tablets, direkt als Clienten einsetzen lassen. Es können die selben Schnittstellen verwendet werden, die auch von den anderen Client-Anwendungen genutzt werden. Somit würden für das System neue Eingabeoptionen mit geringem Aufwand entstehen.

Darüber hinaus haben die Messungen mit den generierten Simulink-Blöcken gezeigt, dass die Echtzeitfähigkeit auch bei hohen Datenraten erhalten bleibt. Infolgedessen würde es sich anbieten, ROS für die Übertragung von Echtzeitsignalen zu verwenden. Dadurch könnte eine Kommunikationsplattform für das gesamte System genutzt werden.

8 Zusammenfassung

In dieser Arbeit wurde ein Konzept für die Anbindung von Benutzerschnittstellen an die Simulink-Modelle der verteilten Regelungsanwendung des MiroSurge-Projekts entwickelt und an einem Beispiel implementiert. Als Grundlage für die Implementierung wurde ROS als Middleware-Lösung verwendet. Das Konzept ermöglicht die Anbindung von redundanten Benutzerschnittstellen an die Simulink-Modelle. Durch die Erzeugung von universellen Schnittstellen ist es hierbei möglich, mit verschiedenen Benutzerschnittstellen gleiche Funktionen auszuführen. Es wurden zudem Mechanismen verwendet um verschiedene Instanzen der Benutzerinterfaces synchron zu halten.

Der entwickelte Codegenerator dient zur Erstellung der Simulink-Blöcke direkt aus der ROS-IDL. Somit werden die Schnittstellen für alle Anwendungen an einer zentralen Stelle definiert und können in den Client-Anwendung genutzt werden. Zudem wird der Anwender durch den Codegenerator dabei unterstützt, schmale universelle Schnittstellen zu erstellen. Diese spiegeln sich dann in kleineren Simulink-Blöcken wider, welche sich in den Modellen in der Ebene integrieren lassen, in der sie auch benötigt werden. Dies steigert die Übersichtlichkeit und vereinfacht die Wartung der Simulink-Modelle.

Die Integration und der Test der MIRO-Einarmapplikation als Beispielanwendung haben gezeigt, dass das entwickelte Konzept die Flexibilität und die Robustheit des Systems erhöht. Die Verwendung der ROS-Simulink-Blöcke beeinflusst die Simulink-Modelle nur geringfügig. Dies wurde durch Zeit-Messungen der implementierten ROS-Simulink-Blöcke überprüft. Die Simulink-Modelle können mit dem System transparent auf dem QNX Echtzeitsystem oder im Simulations-Modus unter Linux verwendet werden. Darüber hinaus können alle Anwendungen auf Rechnern gestartet werden, die eine Verbindung zum ROS-Master besitzen. Die ROS-Middleware übernimmt mit dem Naming-Service die Verwaltung der IP-Adressen und Port-Nummern für alle Clienten und Modelle. Der Anwender kümmert sich nur um die Vergabe von Node- und Topic-Namen. Das implementierte System weist außerdem eine hohe Robustheit auf. Clienten und Modelle können abgeschaltet werden, ohne dass andere Komponenten abstürzen. Verbindungen werden beim Neustart der Programme automatisch wieder aufgebaut. Zusammenfassend gesagt, vereinfachen das erstellte Konzept und der Einsatz von ROS als Middleware die Anbindung von Nutzerschnittstellen an die verteilten Simulink-Modelle.

Literaturverzeichnis

- [1] Hagn U, Konietschke R, Tobergte A, Nickl M, Jörg S, Kübler B, u. a. DLR MiroSurge: a versatile system for research in endoscopic telesurgery. *Int J CARS* 2009;5:183-193. doi:10.1007/s11548-009-0372-4
- [2] Hagn U, Nickl M, Jörg S, Tobergte A, Kübler B, Passig G, u. a. DLR MIROSURGE—towards versatility in surgical robotics. *Proceedings of CURAC* 2008;9:143–146.
- [3] Tobergte A, Passig G, Kuebler B, Seibold U, Hagn UA, Fröhlich FA, u. a. MiroSurge-Advanced User Interaction Modalities in Minimally Invasive Robotic Surgery. *Presence: Teleoperators and Virtual Environments* 2010;19:400–414.
- [4] Hagn U, Nickl M, Jörg S, Passig G, Bahls T, Nothhelfer A, u. a. The DLR MIRO: a versatile lightweight robot for surgical applications. *Industrial Robot: An International Journal* 2008;35:324–336.
- [5] Thielmann S, Seibold U, Haslinger R, Passig G, Bahls T, Jörg S, u. a. MICA-A new generation of versatile instruments in robotic surgery. In: *Intelligent Robots and Systems (IROS)*, 2010 IEEE/RSJ International Conference on. o. J. S. 871–878.
- [6] Simulink - Simulation and Model-Based Design. o. J. Available: <http://www.mathworks.com/products/simulink/>. Zugegriffen 2 Juli 2011.
- [7] Simulink Coder - MATLAB. o. J. Available: <http://www.mathworks.com/products/simulink-coder/index.html>. Zugegriffen 10 Juli 2011.
- [8] QNX Neutrino RTOS. o. J. Available: <http://www.qnx.com/products/neutrino-rtos/neutrino-rtos.html>. Zugegriffen 2 Juli 2011.
- [9] Bäuml B, Hirzinger G. Agile robot development (aRD): A pragmatic approach to robotic software. In: *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2006.
- [10] BiSS Homepage. o. J. Available: <http://www.biss-interface.com/>. Zugegriffen 10 Juli 2011.
- [11] SpaceWire - Home. o. J. Available: <http://spacewire.esa.int/content/Home/HomeIntro.php>. Zugegriffen 10 Juli 2011.
- [12] MathWorks Deutschland - Durchführen von Simulationen - Simulink. o. J. Available: <http://www.mathworks.de/products/simulink/description4.html>. Zugegriffen 24 Aug 2011.
- [13] Nickl M, Jörg S, Hirzinger G. The Virtual Path: The Domain Model for the Design of the MIRO Surgical Robotic System. In: *9th International IFAC Symposium on Robot Control, IFAC*. Gifu, Japan: <http://www.ifac-papersonline.net>. 2009. S. 97–103.

- [14] Force Dimension - Products - omega.7. o. J. Available: <http://www.forcedimension.com/sigma7-overview>. Zugriffen 12 Juli 2011.
- [15] Qt — Qt - A cross-platform application and UI framework. o. J. Available: <http://qt.nokia.com/products/>. Zugriffen 12 Juli 2011.
- [16] Bakken DE. Middleware. Encyclopedia of Distributed Computing 2002. Available: www.dia.uniroma3.it/~cabibbo/ids/altrui/middleware-bakken.pdf.
- [17] Pinus H. Middleware: Past and present a comparison. consultado en febrero de 2006. Available: <http://userpages.umbc.edu/~dgorin1/451/middleware/middleware.pdf>.
- [18] Hurwitz J. Sorting out middleware. DBMS 1998;11:10–12.
- [19] Object Management Group. o. J. Available: <http://www.omg.org/>. Zugriffen 9 Juli 2011.
- [20] CORBA FAQ. o. J. Available: <http://www.omg.org/gettingstarted/corbafaq.htm>. Zugriffen 9 Juli 2011.
- [21] Java EE 6 Technologies. o. J. Available: <http://www.oracle.com/technetwork/java/javaee/tech/index.html>. Zugriffen 24 Aug 2011.
- [22] .NET Framework. o. J. Available: <http://www.microsoft.com/germany/net/net-framework.aspx>. Zugriffen 24 Aug 2011.
- [23] Real-time CORBA with TAO (The ACE ORB). o. J. Available: <http://www1.cse.wustl.edu/~schmidt/TAO.html>. Zugriffen 19 Juli 2011.
- [24] omniORB. o. J. Available: <http://omniorb.sourceforge.net/>. Zugriffen 19 Juli 2011.
- [25] ZeroC - The Internet Communications Engine (Ice). o. J. Available: <http://www.zeroc.com/ice.html>. Zugriffen 19 Juli 2011.
- [26] Open Network Computing Remote Procedure Call - Wikipedia, the free encyclopedia. o. J. Available: http://en.wikipedia.org/wiki/ONC_RPC. Zugriffen 19 Juli 2011.
- [27] XML-RPC Home Page. o. J. Available: <http://www.xmlrpc.com/>. Zugriffen 18 Juli 2011.
- [28] SOAP Specifications. o. J. Available: <http://www.w3.org/TR/soap/>. Zugriffen 18 Juli 2011.
- [29] Inspirel - YAMI4. o. J. Available: <http://www.inspirel.com/yami4/>. Zugriffen 18 Juli 2011.
- [30] The Intelligent Transport Layer - zeromq. o. J. Available: <http://www.zeromq.org/>. Zugriffen 18 Juli 2011.
- [31] Hardware Specs | Willow Garage. o. J. Available: <http://www.willowgarage.com/pages/pr2/specs>. Zugriffen 19 Juli 2011.

- [32] Smart W. Is a common middleware for robotics possible? In: Proceedings of the IROS 2007 workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware. Citeseer.2007.
- [33] Mohamed N, Al-Jaroodi J, Jawhar I. Middleware for robotics: A survey. In: Robotics, Automation and Mechatronics, 2008 IEEE Conference on.o. J. S. 736–742.
- [34] FAMOUSO -- Family of Adaptive Middleware for autonomOUS Sentient Objects. o. J. Available: <http://famouso.sourceforge.net/>. Zugegriffen 21 Juli 2011.
- [35] Schulze M. FAMOUSO—Eine adaptierbare Publish/Subscribe Middleware für ressourcenbeschränkte Systeme. Electronic Communications of the EASST 2009;17.
- [36] Ando N, Suehiro T, Kitagaki K, Kotoku T, Yoon WK. RT-middleware: distributed component middleware for RT (robot technology). In: Intelligent Robots and Systems, 2005.(IROS 2005). 2005 IEEE/RSJ International Conference on.2005. S. 3933–3938.
- [37] Ando N, Suehiro T, Kitagaki K, Kotoku T, Yoon WK. RT-Component Object Model in RT-Middleware&# 8212; Distributed Component Middleware for RT (Robot Technology). In: Computational Intelligence in Robotics and Automation, 2005. CIRA 2005. Proceedings. 2005 IEEE International Symposium on.2005. S. 457–462.
- [38] RTC 1.0. o. J. Available: <http://www.omg.org/spec/RTC/1.0/>. Zugegriffen 19 Juli 2011.
- [39] OpenRTM-aist official website | OpenRTM-aist. o. J. Available: <http://www.openrtm.org/openrtm/en>. Zugegriffen 19 Juli 2011.
- [40] Python Programming Language – Official Website. o. J. Available: <http://www.python.org/>. Zugegriffen 31 Juli 2011.
- [41] Orca Robotics. o. J. Available: <http://orca-robotics.sourceforge.net/index.html>. Zugegriffen 19 Juli 2011.
- [42] Makarenko A, Brooks A, Kaupp T. Orca: Components for robotics. In: International Conference on Intelligent Robots and Systems (IROS).2006. S. 163–168.
- [43] The Orocos Project | Smarter control in robotics & automation! o. J. Available: <http://www.orocos.org/>. Zugegriffen 20 Juli 2011.
- [44] Documentation - ROS Wiki. o. J. Available: <http://www.ros.org/wiki/>. Zugegriffen 20 Juli 2011.
- [45] Quigley M, Gerkey B, Conley K, Faust J, Foote T, Leibs J, u. a. ROS: an open-source Robot Operating System. In: ICRA Workshop on Open Source Software.2009.
- [46] 1000+ ROS Packages - ROS robotics news. o. J. Available: <http://www.ros.org/news/2010/04/1000-ros-packages.html>. Zugegriffen 19 Juli 2011.
- [47] Nodes - ROS Wiki. o. J. Available: http://www.ros.org/wiki/Nodes#Remapping_Arguments.Node_parameter_assignment. Zugegriffen 27 Juli 2011.

- [48] roscpp/Overview/Callbacks and Spinning - ROS Wiki. o. J. Available: <http://www.ros.org/wiki/roscpp/Overview/Callbacks%20and%20Spinning>. Zugegriffen 27 Juli 2011.
- [49] rosjava/Overview/Build - ROS Wiki. o. J. Available: <http://www.ros.org/wiki/rosjava/Overview/Build>. Zugegriffen 25 Aug 2011.
- [50] ros_comm - ROS Wiki. o. J. Available: http://www.ros.org/wiki/ros_comm. Zugegriffen 22 Aug 2011.
- [51] Boost C++ Libraries. o. J. Available: <http://www.boost.org/>. Zugegriffen 22 Aug 2011.
- [52] log4cxx - Short introduction to Apache log4cxx. o. J. Available: <http://logging.apache.org/log4cxx/>. Zugegriffen 22 Aug 2011.
- [53] Welcome! - The Apache Portable Runtime Project. o. J. Available: <http://apr.apache.org/>. Zugegriffen 22 Aug 2011.
- [54] How the Simulink Engine Interacts with C S-Functions :: Writing S-Functions in C (Simulink®). o. J. Available: <http://www.mathworks.de/help/toolbox/simulink/sfg/f8-37326.html>. Zugegriffen 24 Aug 2011.
- [55] Making C++ Objects Persistent :: Creating C++ S-Functions (Simulink®). o. J. Available: <http://www.mathworks.com/help/toolbox/simulink/sfg/f3-88429.html>. Zugegriffen 8 Aug 2011.
- [56] roslib. o. J. Available: <http://www.ros.org/doc/api/roslib/html/python/>. Zugegriffen 8 Aug 2011.
- [57] 7.1. string — Common string operations — Python v2.7.2 documentation. o. J. Available: <http://docs.python.org/library/string.html#template-strings>. Zugegriffen 8 Aug 2011.
- [58] roscpp/Overview/Publishers and Subscribers - ROS Wiki. o. J. Available: http://www.ros.org/wiki/roscpp/Overview/Publishers%20and%20Subscribers#Intraprocess_Publishing. Zugegriffen 10 Aug 2011.
- [59] rosjava - An implementation of ROS in pure Java with Android support. - Google Project Hosting. o. J. Available: <http://code.google.com/p/rosjava/>. Zugegriffen 17 Aug 2011.

Abbildungsverzeichnis

Abbildung 1.1:	DLR MiroSurge-System mit künstlichem Herzen.....	1
Abbildung 1.2:	Eingabestation des DLR MiroSurge-Szenarios mit 3D-Monitor und haptischen Eingabegeräten.....	2
Abbildung 1.3:	Architektur des verteilten MiroSurge-Regelungssystems mit den verschiedenen Kommunikationsarten.....	3
Abbildung 2.1:	Kommunikationsebenen der MIRO Hardware.....	7
Abbildung 2.2:	Der virtuelle Pfad beginnt in der physikalischen Welt beim Sensor und endet beim Aktor. [13].....	9
Abbildung 2.3:	Aktor Zustandsmaschine. [13].....	9
Abbildung 2.4:	Haptisches Eingabegerät Sigma7.....	11
Abbildung 2.5:	TMCD, Notausschalter mit konfigurierbaren Tasten und SpaceMouse.....	11
Abbildung 2.6:	Mirs-GUI Anwendung für die Steuerung des MiroSurge-Systems.....	12
Abbildung 3.1:	Middleware im Zusammenhang eines verteilten Systems. [16].....	15
Abbildung 4.1:	Kommunikation auf RPC Basis.....	25
Abbildung 4.2:	Kommunikation auf Publish-Subscribe Basis.....	25
Abbildung 4.3:	Kommunikation mit Request-Reply und Publish-Subscribe.....	26
Abbildung 4.4:	Mitsenden von einem Teil des letzten Zustands. Der Wert a wird vom Clienten neu gesetzt, der Wert b wird aus dem zyklisch empfangenden Systemzustand übernommen.....	26
Abbildung 4.5:	Problem beim mitsenden des letzten Zustands mit zwei Clienten.....	27
Abbildung 4.6:	Ausnutzung der redundanten Schnittstellen.....	28
Abbildung 4.7:	Modell mit einem Service für die Zugriffsverwaltung.....	29
Abbildung 4.8:	Publish- und Subscribe-Blöcke für Simulink.....	30
Abbildung 4.9:	Request-Reply-Block für Simulink.....	30
Abbildung 4.10:	Zeitlicher Ablauf beim Aufruf des RPC-Blocks mit dem Enable/Ready Protokoll.....	31
Abbildung 4.11:	Zeitlicher Ablauf beim Aufruf des RPC-Blocks bei einem Timeout.....	31
Abbildung 4.12:	Prinzipieller Aufbau des Codegenerators für die Erzeugung der S-Functions und der Simulink-Block-Bibliothek.....	32
Abbildung 4.13:	Architektur der MIRO Einarmapplikation.....	33
Abbildung 4.14:	Überarbeitete Einarmapplikation nach dem vorgestellten Konzept.....	35
Abbildung 5.1:	Ablauf der S-Function Funktionsaufrufe von Simulink zur Initialisierung und Simulation. [54].....	41
Abbildung 5.2:	Synchronisation zwischen Subscriber-Thread und Simulink.....	44
Abbildung 5.3:	Synchronisation zwischen Service-Thread und Simulink.....	45

Abbildung 5.4:	Aufbau des implementierten Codegenerators zum Erzeugen der Simulink-Blöcke aus der ROS-IDL.....	46
Abbildung 5.5:	Logos für die ROS Simulink-Blöcke.....	48
Abbildung 5.6:	Simulink Library Browser mit der durch den Codegenerator erzeugten Blockbibliothek.....	48
Abbildung 6.1:	Simulink-Modell für die Messung des Publisher-Blocks.....	50
Abbildung 6.2:	Messung des Publisher-Blocks bei einer Rate mit 50Hz, ohne Subscriber. Der Mittelwert ist rot und die Standardabweichung grün dargestellt (Mittelwert 2,4µs, Maximalwert 4,4µs).....	51
Abbildung 6.3:	Messung des Publisher-Blocks bei einer Rate mit 50Hz und einem Subscriber (Mittelwert 14,2µs, Maximalwert 16,4µs).....	51
Abbildung 6.4:	Simulink-Modell für die Messung des Subscriber-Blocks.....	52
Abbildung 6.5:	Messung des Subscriber-Blocks ohne Publisher (Mittelwert 140ns, Maximalwert 280ns).....	52
Abbildung 6.6:	Messung des Subscriber-Blocks mit einem Publisher bei einer Rate von 50Hz (Mittelwert 140ns, Maximalwert 630ns).....	53
Abbildung 6.7:	Simulink-Modell für die Messung des Service-Blocks.....	54
Abbildung 6.8:	Messung des Service-Blocks mit Aufrufen durch einen Clienten. Der dritte Aufruf ist rot markiert. (Insgesamt 15 Aufrufe).....	54
Abbildung 6.9:	Vergrößerter Ausschnitt des Service-Aufrufs aus Abbildung 6.8.....	55
Abbildung 6.10:	Messung des Publisher-Blocks bei einer Rate mit 1kHz und einem angemeldeten Subscriber (Mittelwert 15µs, Maximalwert 25µs).....	56
Abbildung 6.11:	Messung des Subscriber-Blocks mit einem sendenden Publisher bei einer Rate von 1kHz (Mittelwert 190ns, Maximalwert 11µs).....	56
Abbildung 6.12:	Messung des Service-Blocks ohne Aufrufe (Mittelwert 296ns, Maximalwert 10,8µs).....	57
Abbildung 6.13:	Messung der S-Function mit For-Schleife (Mittelwert 680ns, Maximalwert 11µs).....	57
Abbildung 6.14:	aRDnet-Kommunikationsblock für die MIRO Einarmapplikation.....	59
Abbildung 6.15:	Mit dem Codegenerator erstellte Simulink-Blöcke.....	60
Abbildung 6.16:	Vereinfachte Darstellung der Modellebenen in der Einarmapplikation. Links die alten und rechts die neuen Schnittstellen.....	61
Abbildung 6.17:	MIRO Singlearm GUI mit ROS-Anbindung.....	63
Abbildung 6.18:	Belegung des TMCDs für die Client-Anwendung.....	63
Abbildung 6.19:	Systemkonfiguration mit mehreren Client-Anwendungen auf einem Linux-System und dem Simulink-Modell auf einem QNX-Echtzeitsystem.....	64
Abbildung 6.20:	Konfiguration der MIRO Einarmapplikation mit einem MIRO-Modell, zwei GUI-Anwendungen und TMCD-Clienten dargestellt mit dem ROS-Tool rxgraph.....	65

Abbildung 6.21: System mit mehreren Client-Anwendungen und einem Simlink-Modell auf einem Linux-System und einem zusätzlichen Modell auf einem QNX-Echtzeitsystem.....	65
Abbildung 6.22: Konfiguration der MIRO Einarmapplikation mit dem ROS-Tool rxgraph dargestellt. Zwei laufende Modelle und drei verbundene Client-Anwendung.....	66

Tabellenverzeichnis

Tabelle 2.1: Übersicht der Benutzerschnittstellen mit der verwendeten Kommunikationsart.....	13
Tabelle 3.1: Übersicht der Auswahlkriterien für eine Middleware-Lösung.....	18
Tabelle 3.2: Bewertung der untersuchten Middleware-Lösungen nach den gewählten Auswahlkriterien. Die Summe ergibt sich aus den positiven (+), neutralen (0) und negativen Eigenschaften (-) der Middleware-Lösungen.....	23
Tabelle 5.1: Für die ROS-Kommunikation unter QNX 6.5 benötigte Bibliotheken.....	40
Tabelle 5.2: Mit ros2simulink_error zugegebene Fehler-Codes des Service-Blocks.....	45
Tabelle 5.3: Konvertierung der Datentypen zwischen ROS und Simulink.....	47
Tabelle 6.1: Maximale Ausführungszeit der Blöcke (mdlUpdate und/oder mdlOutputs) über den Zeitraum von ca. einer halben Stunde.....	58
Tabelle 6.2: Dateien für die Schnittstellendefinitionen der MIRO-Einarmapplikation..	60

Anhang

Schnittstellenbeschreibungen der Miro-Einarmapplikation

```
bool start
bool reset
bool set_torques
float64 ros2simulink_timestamp
---
int32 ros2simulink_error
```

Listing 1: MiroHalService.srv

```
bool running
bool failure
float64 ros2simulink_timestamp
```

Listing 2: MiroHalTopic.msg

```
bool set_interface_interpolator
bool set_interface_impedance
bool set_interface_velocity
bool start_interpolator
float64[7] set_payload
float64[7] set_pose
float64 ros2simulink_timestamp
---
int32 ros2simulink_error
```

Listing 3: MiroControlService.srv

```
bool interface_interpolator
bool interface_impedance
bool interface_velocity
bool interpolator_running
float64[7] payload
float64[7] pose
float64 ros2simulink_timestamp
```

Listing 4: MiroControlTopic.msg

```
float64[6] velocity
```

Listing 5: MiroVelocityTopic.msg