FH Vorarlberg
University of Applied Sciences

Systemarchitektur eines Sensor/Aktor-Knotens für dezentralisierbare Aufgabenverteilung

Bachelorarbeit
zur Erlangung des akademischen Grades

Bachelor of Science

Fachhochschule Vorarlberg
Mechatronik

Betreut von
Prof. (FH) Dipl.-Ing. Horatiu O. Pilsan

Vorgelegt von
Nicolai Schwartze

Dornbirn, 3. Juni 2018

Kurzreferat

Systemarchitektur eines Sensor/Aktor-Knotens für dezentralisierbare Aufgabenverteilung

Das Ziel dieser Bachelorarbeit ist es, eine Systemarchitektur zu entwickeln und zu beschreiben, die Tasks von einer ausgelasteten CPU auf einen anderen Knoten im Netzwerk verlagert. Dadurch sollen eine Überlast eines Knotens verhindert und gleichzeitig brachliegende Ressourcen ausgenützt werden.

Der theoretische Teil beinhaltet das Erarbeiten der vollständigen Systemarchitektur und deren Darlegung in UML. Dafür werden bestimmte Standardfunktionalitäten festgelegt, an denen die Architektur theoretisch validiert wird.

Um einen technischen „proof of concept“ umsetzen zu können, werden Teile der Architektur ausprogrammiert. Dafür werden zwei BeagleBone Blacks verwendet. Die Programmierung erfolgt in C bzw. C++. Damit die Knoten leicht in ein bereits bestehendes Netzwerk eingefügt werden können, wird für die Netzwerkkommunikation der offene Standard OPC UA verwendet.

Schließlich wird das System unter verschiedenen Konfigurationen getestet und kritisch analysiert. Dabei werden sowohl Stärken als auch Schwächen dargelegt und mögliche Optimierungspotentiale hervorgehoben.

Abstract

System Architecture of a Sensor/Actuator-Node for decentralized Task-Management

The objective of this Bachelor Thesis is to develop and describe a system architecture that would allow a stressed CPU to move its tasks to another node on the network. Therefore, an overload on one CPU could be prevented while using idle resources on the network.

The theoretical part of this thesis includes the development of a complete system architecture. The software components are visualized in UML. Certain standard functionalities are defined to validate the theoretical model.

In order to perform a technical proof of concept, the necessary parts of the architecture are programmed on two BeagleBone Blacks. The programs are created in C and C++. For including the nodes into an already existing network the open standard of OPC UA is used.

Finally, the system is tested under various configurations and the results are critically analyzed. Further, the strengths and weaknesses that are leading to optimization potential are highlighted.

Inhaltsverzeichnis

Darstellungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

Abkürzung	Bedeutung
ASIC	Application-Specific Integrated Circuit
CPU	Central Processing Unit
CSV	Comma-Separated Values
DRAM	Dynamic Random Access Memory
GUI	Graphical User Interface
GIO	General Input Output
Hz	Hertz, Schwingungen pro Sekunde
IDE	Integrated Development Environment
IoT	Internet of Things
IPC	Interprocess Communication
LB	Load Balancing
MAC (Adresse)	Media Access Control (Adresse)
MB	MegaByte
PC	Personal Computer
PID	Process Identification Number
PLC	Programmable Logic Controller
PPID	Parent Process Identification Number
PRU	Programmable Real Time Unit
PWM	Pulsweiten moduliertes (Signal)
P2P	Peer-to-Peer
RT	Real Time (Echtzeit)
RTOS	Real Time Operating System
SCADA	Supervisory Control And Data Acquisition
semid	Semaphore Identifier
shmid	Shared Memory Identifier
SPI	Serial Peripheral Interface
SPS	Speicherprogrammierbare Steuerung
TSN	Time Sensitive Networking
UART	Universal Asynchronous Receiver Transmitter
UML	Unified Modeling Language
USB	Universal Serial Bus

Tabelle 1: Abkürzungsverzeichnis

1. Einleitung

Die Bachelorarbeit „Zweistufige Kommunikation mit einem multifunktionalen ASIC über UART und SPI“ von Klaus Jäger wurde ebenfalls bei Bachmann electronic GmbH durchgeführt. Klaus Jäger hat die Hardwarebasis für den aktiven Netzwerkteilnehmer (Knoten) gelegt. Dieser besteht aus einem Minicomputer (BeagleBone Black) und einem GIO212. Das GIO Interface kann bereits mittels eines Konfigurators am PC eingestellt und verwendet werden.

In dieser Arbeit wird die Systemarchitektur softwaretechnisch auf dem Knoten implementiert. Als Betriebssystem wird die Linux Distribution Debian verwendet.

1.1 Motivation

Im Jahre 1965 formulierte der Intel Mitbegründer Gordon Moore das nach ihm benannte „Gesetz“: Moore’s Law. Die heute eher als Vermutung abgestempelte Aussage besagt, dass die Integrationsdichte, also die Anzahl der Transistoren auf einem Computerchip, sich etwa alle zwei Jahre verdoppelt. Daran gekoppelt ist die kontinuierliche Leistungssteigerung von Prozessoren. Bis jetzt hat sich diese Aussage als wahr erwiesen, es gibt jedoch starke Anzeichen dafür, dass in naher Zukunft die Integrationsdichte nicht weiter erhöht werden kann.

Auch der altbewährten Von Neumann Architektur sind Grenzen gesetzt. Der sogenannte Von Neumann Flaschenhals (*englisch: bottle-neck*) beschreibt die Abhängigkeit der Rechenleistung von der Zugriffsgeschwindigkeit auf den Hauptspeicher. Diese Zugriffsgeschwindigkeit ist insbesondere bei DRAM sehr beschränkt.

All diese Probleme deuten darauf hin, dass der Rechenleistung von Computern eine theoretische Grenze gesetzt ist. Trotzdem werden die anstehenden Aufgaben immer komplexer und die Datenmengen immer größer. Um diesen Widerspruch zu lösen, werden neue Wege gesucht, mit den steigenden Anforderungen Schritt zu halten.

Neben der Forschung an neuen Trägermaterialien für Transistoren oder der Entwicklung von Quantencomputern ist auch die Parallelisierung von Aufgaben und deren dynamische Verteilung auf mehrere Maschinen eine mögliche Lösung für dieses Problem. Dabei wird versucht, Leerlaufzeiten von Prozessoren optimal auszunutzen.

[Vgl. Bauke, 2006, S. 5 - 8]

1.2 Aufgabenstellung

Die Aufgabenstellung der Bachelorarbeit ist es, vordefinierte Load Balancing Tasks in einem Netzwerk aus Sensor/Aktor Knoten dynamisch auf verschiedene CPUs aufzuteilen. Die Auslagerung erfolgt nach taskspezifischen Kriterien sowie der Auslastung des Knotens selbst. Die Auslagerung soll transparent und nachvollziehbar erfolgen. Zu diesem Zweck wird die Softwarearchitektur des Knotens entwickelt und modelliert.

Eine besondere Schwierigkeit ist dabei die Vereinigung von zeitkritischer Aufgabenverteilung, dynamischer Netzwerkkommunikation und Echtzeitbearbeitung der Ein- und Ausgänge.

Damit die Portierung des Softwaregerüstes auf andere Plattformen zu einem späteren Zeitpunkt gewährleistet werden kann, soll die Architektur möglichst abstrakt beschrieben werden. Dank der Verwendung von offenen Standards, wie zum Beispiel OPC UA, soll die Integration der Systemarchitektur in bereits existierende Umgebungen erleichtert werden.

Die Integration des GIO212 wird zunächst hintangestellt, da der Arbeitsumfang dafür zu groß wäre. Um dennoch Inputs und Outputs schalten zu können, wird die I/O-Struktur des BeagleBone Black verwendet.

1.3 Zielsetzung

Das Ziel dieser Arbeit ist es, eine einfache Form der Taskverlagerung auf andere CPUs im Netzwerk zu bewerkstelligen. Dafür werden zwei BeagleBone Blacks in einem Peer-to-Peer Netzwerk miteinander verbunden. Ein Knoten generiert dabei die CPU-Last und muss Tasks auf den anderen Knoten auslagern. Der zweite Knoten ist nicht ausgelastet und kann theoretisch jeden Task annehmen.

Bei den Tasks handelt es sich um verschiedene vorgefertigte Programme, deren Eigenschaften und Charakteristiken schon vor der Ausführung bekannt sind. Diese Tasks simulieren verschiedene, oft angewendete Aufgaben, wie einen Datenbankzugriff oder eine Regelung.

Der implementierte Load Balancer kontrolliert die Auslastung des CPU. Sobald eine konfigurierbare Auslastungsgrenze überschritten wird, lagert dieser die Tasks auf die zweite CPU aus. Die Auslagerung soll anhand einer festgelegten priorisierten Reihenfolge nachvollzogen werden können.

2. Stand der Vorarbeit

Wie bereits erwähnt, basiert diese Bachelorarbeit auf der Vorentwicklung von Klaus Jäger in „Zweistufige Kommunikation mit einem multifunktionalen ASIC über UART und SPI“. Im Folgenden werden die aktuellen Funktionalitäten des Knotens zusammengefasst.

2.1 Vorhandene Hardware

Als Interface zur realen Welt wurde ein modifiziertes GIO212 verwendet. Hier können insgesamt zwei Kanäle beliebig konfiguriert werden. Folgende Funktionen stehen zur Verfügung:

- Digitaler Spannungs- Aus/Eingang
- Analoger Spannungs- Aus/Eingang
- Analoger Strom- Aus/Eingang
- Pulsweiten modulierte Signale
- 32 Bit Zähler
- Anschluss für Thermoelemente
- Platin Temperaturfühler

Zusätzlich können einfache Filter auf diese Signale angewendet werden. Dazu zählen sowohl Spikefilter als auch Filter zur Bandbreitenbegrenzung.

Um Fehler zu vermeiden, besteht die Möglichkeit, den aktuellen Status des Kanals abzurufen. Im sogenannten Mischmode können auf einem Kanal zwei Signale übertragen werden, dies ist aber nicht mit jeder beliebigen Konfiguration möglich.

Weiters kann zu Wartungszwecken die interne Chiptemperatur abgerufen und überwacht werden. [Vgl. Jäger, 2017, S. 4 - 5]

2.2 Vorhandene Software

Um die genannten Funktionen der Hardware nutzen zu können, muss das modifizierte GIO212 entsprechend konfiguriert werden. Dies geschieht in der ersten Stufe über die SPI Schnittstelle des BeagleBone Black Minicomputers. Die zweite Stufe ist eine UART zu USB Kommunikation mit dem PC. Auf dem PC läuft eine C# GUI Anwendung, mit welcher sich die Kanäle konfigurieren lassen. An diesem Konfigurator lassen sich nicht nur die einzelnen Kanäle konfigurieren, sondern auch die aktuellen Ein- und Ausgabewerte überwachen. Ein C-Programm auf dem Minicomputer übersetzt die Befehle zwischen der UART und der SPI Schnittstelle. [Vgl. Jäger, 2017, S. 38 - 41]

3. Stand der Technik

In diesem Kapitel werden bereits existierende Komponenten und Konzepte erklärt, auf die bei der Implementierung zurückgegriffen wird.

3.1 BeagleBone Black

Damit sich die Hardwaregrundlage nicht ändert und auf den Erfahrungen der Vorarbeit von Klaus Jäger aufgebaut werden kann, wird auch in dieser Arbeit wieder das BeagleBone Black verwendet. Dies ist ein Einplatinencomputer, für den mehrere verschiedene Betriebssysteme angeboten werden. Auf der Webseite beagleboard.org wird jedoch die Linux Distribution Debian empfohlen. Das Layout ist frei verfügbar, somit ist der Computer besonders für Open Source Projekte geeignet.

Programmiert wird das BeagleBone Black über die bereits installierte Cloud9 IDE. Dazu wird das Board mit einem PC per USB verbunden. Anschließend kann die IDE im Browser des PCs geöffnet werden. Dafür verbindet sich der PC zum Port 3000 des BeagleBone.

Hardwaretechnisch ist das BeagleBone Black mit einem Cortex-A8 ausgerüstet, welcher mit einer Taktfrequenz von 1000 MHz operiert. Die Größe des Hauptspeichers beträgt 512 MB. Neben dem normalen Prozessor besitzt das BeagleBone Black auch zwei Programmable Real Time Units mit einer Taktfrequenz von 200 MHz. Die Programmierung der PRUs erfolgt durch eine Reihe von Registern. Dadurch wird der Umweg über den Kernel des Betriebssystems vermieden. Diese Register sind für die Nutzenden über einen Treiber zugänglich gemacht. Dadurch sind die PRUs besonders für deterministische Echtzeitaufgaben geeignet. Wegen dieser sehr hardwarenahen Ausführung müssen die PRUs allerdings in Assemblersprache programmiert werden. [Vgl. Jäger, 2017 S. 2 - 3]

3.2 Betriebssystem

Auf dem BeagleBone Black ist aktuell die Version 9.3 2018-03-05 (IoT) von Debian (Linux) installiert. Im Gegensatz zur LXQT Version verfügt dieses Betriebssystem über keinen graphischen Desktop, wodurch Rechenleistung gespart wird. Mit Linux werden grundlegende Eigenschaften wie Netzwerkkommunikation, Interprozesskommunikation, Compiler, Filesystem, Multiprocessing/Multithreading, Schnittstelleninterface sowie alle weiteren Hardwareinteraktionen unterstützt.

Obwohl Linux im Allgemeinen gewisse Real Time Funktionalitäten bereithält, ist es jedoch nur bedingt für Hard Real Time Aufgaben geeignet. Für Soft Real Time kann zwischen verschiedenen Schedulern gewechselt werden. Das bedeutet, dass der Standard-Scheduler von *sched_other* auf einen der Real Time Scheduler *sched_fifo* oder *sched_rr* (Round Robin) gesetzt wird.

sched_other: Dabei handelt es sich um den standardmäßig installierten Scheduler. Hier basiert das Scheduling auf Prioritäten und Zeitscheiben. Prozesse können altern und erhalten dadurch eine höhere Priorität, so wird „starvation“ von einzelnen Prozessen verhindert. Die CPU wird einem laufenden Prozess entzogen, wenn seine Zeitscheibe abgelaufen ist oder wenn ein Prozess auf eine Eingabe bzw. auf ein Interrupt wartet. Schließlich wird der Prozess aussortiert und muss warten, bis alle anderen Prozesse abgelaufen sind.

sched_fifo: Hier werden statische Prioritäten vergeben. Ein Prozess behält die CPU bis dieser seine Arbeit erledigt hat oder die CPU freiwillig selbst abgibt. Anschließend wird der Prozess sofort wieder rechenbereit und entsprechend seiner Priorität eingereiht.

sched_rr: Zusätzlich zu den Eigenschaften des FIFO Schedulings haben Prozesse, die das Round Robin Scheduling verwenden, eine Zeitscheibe, die ablaufen kann. Wenn mehrere Prozesse die gleiche Priorität haben, erfolgt innerhalb dieser Ordnung ein Round Robin Wechsel.

[Vgl. Achilles, 2006, S. 44 - 51]

Solange jedoch rechenbereite Real Time Prozesse vorhanden sind, kann kein anderer Prozess mit *sched_other* die CPU erlangen. Deswegen wird das Real-Time Throttling eingeführt. Dies wird im Wesentlichen durch zwei Variable bestimmt: Die gesamte Dauer, für die ein Prozess die CPU besitzt, darf während einer Periode *rt_period* die Laufzeit *rt_runtime* nicht überschreiten.

Mit dem sogenannten *preempt_rt* Patch kommt man der Hard Real Time schon etwas näher. Dadurch werden die meisten Funktionen des Kernels durch unterbrechbare Funktionen ersetzt. Dies verkürzt die Latenzzeit, also die Antwortzeit bei Interrupts, auf ein Minimum.
[Vgl. linuxrealtime.org, 2015]

Die Priorität in Linux wird in eine statische und eine dynamische Priorität unterteilt. Die statischen Prioritäten reichen von 0 bis 99 und können von den Nutzenden nicht verändert werden. Die dynamische Priorität, auch Niceness oder Nice-Value genannt, ist veränderbar. Diese reicht von +20 bis -20 wobei standardgemäß 0 verwendet wird. Der Nice-Value kann während der Laufzeit des Programms mit der Funktion renice erhöht oder verringert werden. Je höher der Nice-Value ist, desto geringer ist die Priorität des Prozesses. Um die Priorität eines Prozesses zu erhöhen, muss der oder die Nutzende über Root-Rechte oder Superuser-Rechte verfügen.

[Vgl. Plötner, Wendzel, 2012, Kap. 26.5]

3.3 Cluster Computing

Unter Cluster Computing versteht man das Lösen von informationstechnischen Problemstellungen mit Hilfe von mehreren Teilnehmern eines Netzwerks. Diese Teilnehmer werden als Knoten (*englisch: node*) bezeichnet. Beispiele für Knoten wären PCs, IoT Geräte oder sogar Speicherprogrammierbare Steuerungen. Ähnlich wie beim parallelen Programmieren, bei welchem eine Aufgabe mittels Threads oder Prozessen auf mehrere CPUs innerhalb einer Maschine aufgeteilt wird, werden beim Cluster Computing die Ressourcen von mehreren Maschinen geteilt.

Den ersten Schritt in Richtung parallele Berechnung wagte Gene Amdahl 1967, als er zum ersten Mal den mathematischen Zusammenhang zwischen der Parallelisierung von Aufgaben und deren Laufzeit auf mehreren Prozessoren beschrieb. Dieses Gesetz (Gleichung 1) wurde später unter dem Namen Amdahls Gesetz *(englisch: Amdahl's Law)* bekannt.

$$T(p) = \sigma + \frac{\pi}{p} \tag{1}$$

Dabei ist p die Anzahl der Prozessoren. Die Laufzeit T ist die Summe aus der nicht parallelisierten Laufzeit σ und der Laufzeit der parallelisierten Anweisungen π. Da π auf die Anzahl der Prozessoren aufgeteilt wird, muss noch durch p dividiert werden.

Der erste richtige Cluster Computer wurde aber erst 1994 von der NASA unter dem sogenannten Beowulf Project umgesetzt. Da bei diesem Projekt sowohl Standard Hardware als auch freie Software verwendet wurden, gilt es noch heute als Vorbild für moderne verteilte Architekturen.

Jedoch gehen Cluster Computer auch mit einer Vielzahl von Problemen einher, die es zu überwinden gilt. Wie bereits 1967 von Amdahl angesprochen, eignen sich nicht alle Algorithmen zur Parallelisierung. Besonders Funktionen, die stark von anderen Ergebnissen abhängig sind, werden oft besser sequentiell abgearbeitet. Ansonsten werden komplizierte Kommunikationswege benötigt. Insbesondere bei einem Netzwerk aus mehreren Maschinen kann dies zu einem ungewünscht großen Overhead führen.

Wenn mehrere Prozesse die gleiche Aufgabe erfüllen, kann es zum sogenannten Synchronisationsproblem führen. Dabei dürfen bestimme Teile des Algorithmus nur in der richtigen Reihenfolge erledigt werden. Wenn diese Reihenfolge verletzt wird, kann es zur sogenannten Race Condition kommen. Dabei hängt das absolute Ergebnis eines Programms davon ab, in welcher Reihenfolge verschiedene parallele Programme ausgeführt werden. Diese Reihenfolge kann mit verschiedenen Synchronisationsmitteln festgelegt werden, wie zum Beispiel Semaphoren oder MUTEX.

Ein weiterer wichtiger Punkt ist die Granularität der Aufgabe. Dabei gilt es herauszufinden, wie viele Prozesse zur Ausführung einer Aufgabe wirklich geeignet sind. Wenn es zu wenig sind, haben die einzelnen Prozesse zu viel zu tun, wenn es zu viele sind, überwiegt der Overhead der Kommunikation.

Nicht zu verwechseln ist das Cluster Computing mit dem Grid Computing, bei welchem es sich um die kollaborative Zusammenarbeit über das Internet handelt. Der vermutlich größte Unterschied zwischen den zwei Verfahren ist, dass das Netzwerk beim Grid Computing aus heterogenen Knoten aufgebaut ist. Dadurch muss die Software systemunabhängig aufgebaut und umgesetzt werden. Zudem gehören die Knoten ganz unterschiedlichen Personen oder Organisationen. Um Manipulationen vermeiden zu können, muss zusätzlich ein Sicherheitskonzept eingebaut werden. Hingegen beim Cluster Computing gehören alle Knoten einer Person oder Organisation. Wenn ausschließlich homogene Knoten verwendet werden, wird die Implementierung vereinfacht.
[Vgl. Bauke, 2006 S. 10 - 16]

3.4 OPC Unified Architecture

OPC Unified Architecture ist der Nachfolger des Kommunikationsprotokolls OPC Classic und ist im internationalen Standard IEC 62541 festgelegt. OPC UA befindet sich bereits seit Beginn 2010 in andauernder Entwicklung. Neben vielen kostenpflichtigen Versionen existieren auch einige Open Source Varianten - eine davon der C/C++ Stack von open62541. open62541 wird unter anderem vom Fraunhofer Institut und der RWTH Aachen entwickelt und ist unter der Mozilla Public Licence veröffentlicht. Im Folgenden wird OPC UA, im speziellen open62541, mit den wichtigsten Funktionalitäten beschrieben.

OPC UA ist als Client/Server Kommunikation angedacht worden. Es gibt jedoch auch schon Implementierungen, die eine Publish/Subscribe Kommunikationsarchitektur unterstützen. open62541 beinhaltet nur die Client/Server Variante. Die Kommunikation kann auf verschiedenen Protokollen aufgebaut werden, jedoch wird wegen des robusten Protokolls meistens TCP/IP verwendet. Da manche Anfragen zur Bearbeitung länger dauern als andere, muss die Antwort des Servers nicht in chronologischer Reihenfolge ablaufen.

Das Prinzip von OPC UA ist auf sogenannten Service Sets aufgebaut. Diese Sets beinhalten Funktionen, die aufgerufen werden können und dann das gewünschte Ergebnis bereitstellen. Diese Funktionen verwenden eigens in UA spezifizierte Datentypen. Das ermöglicht eine uneingeschränkte Kommunikation zwischen verschiedenen Systemen.

Einer der wichtigsten Services ist das Discovery Service Set. Es beinhaltet Funktionen, die der Client verwenden kann, um nach dezidierten Servern zu suchen. Dabei können bestimmte Filter eingestellt werden. Diese Funktionen werden besonders beim ersten Einfügen eines Knotens in ein bereits existierendes Netzwerk verwendet.

Ein weiteres sehr wichtiges Service Set ist das Method Service Set. Es beinhaltet nur eine einzige Funktion: UA_Client_Service_call(). Mit dieser Funktion können andere Methoden in Objekten auf dem Server aufgerufen werden. Diese Objekte müssen vom

Programmierenden selbst erstellt werden. Dadurch können weitere Funktionalitäten eingerichtet und an individuelle Bedürfnisse angepasst werden.

Das Attribute Service Set stellt Funktionen zum Lesen (UA_Client_Service_read()) und Schreiben (UA_Client_Service_write()) von systeminternen Variablen zur Verfügung. Im IEC 62541 sind noch weitere Funktionen zum Abrufen von älteren Werten, also die Historie von Variablen, beschrieben. Diese Funktionen sind jedoch nicht im open62541 implementiert.

Mit dem MonitoredItem Service Set kann eine bestimmte Variable auf dem Server beobachtet werden. Dabei wird die Variable jedoch nicht durch „pollen“ abgefragt, sondern der Server benachrichtigt den Client mittels Event, wenn sich die Variable ändert. Dies ähnelt dem bereits angesprochenen Publish/Subscribe Modell, wird jedoch von der Norm anders benannt.

Eine weitere sehr wichtige Funktionalität von OPC UA ist die Informationssicherheit und Verschlüsselung. Wenn eine Verbindung zwischen einem Client und einem Server aufgebaut wird, geschieht dies zuerst in reinem TCP. Nach dem Handshake wird ein sicherer Kanal (SecureChannel) erstellt. Die weitere Kommunikation erfolgt über diesen Kanal. Der Kanal kann in verschiedenen Konfigurationen betrieben werden: None, Sign und SignAndEncrypt. Bei der Einstellung Sign wird sichergestellt, dass die Anfrage bzw. die Antwort auch wirklich vom richtigen Client bzw. Server kommt. Mit der Einstellung SignAndEncrypt wird die Kommunikation zusätzlich über einen asynchronen Verschlüsselungsalgorithmus verschlüsselt. Das SignAndEncrypt fehlt allerdings noch in der aktuellen Version 0.2 von open62541.

[Vgl. open62541.org, 2018]

3.5 Netzwerkarchitektur

Es lassen sich zwei grundverschiedene Konzepte zum Aufbau von Netzwerkarchitekturen unterscheiden, der Server-Client und der Peer-to-Peer Ansatz. Sie weisen maßgebliche Unterschiede in der Funktionalität auf. Aufgrund der Vor- und Nachteile ergeben sich weitere, beziehungsweise unterschiedliche Anforderungen an den Knoten.
[Vgl. Schill, Springer, 2012, S. 14]

3.5.1 Server-Client

In der Architektur aus Abbildung 1 übernimmt der Server die vom Client angeforderten Aufgaben. Dabei kann der Server auch Pakete aus dem an ihn angeschlossenen SCADA System erhalten oder an das System versenden. Größere Applikationen, die mehr Ressourcen benötigen (wie zum Beispiel ein Analysetool mit entsprechender Visualisierung oder eine HMI Applikation für User- Interaktive Steuerung), könnten ebenfalls auf dem Server installiert sein. [Vgl. Schill, Springer, 2012, S. 14 - 15]

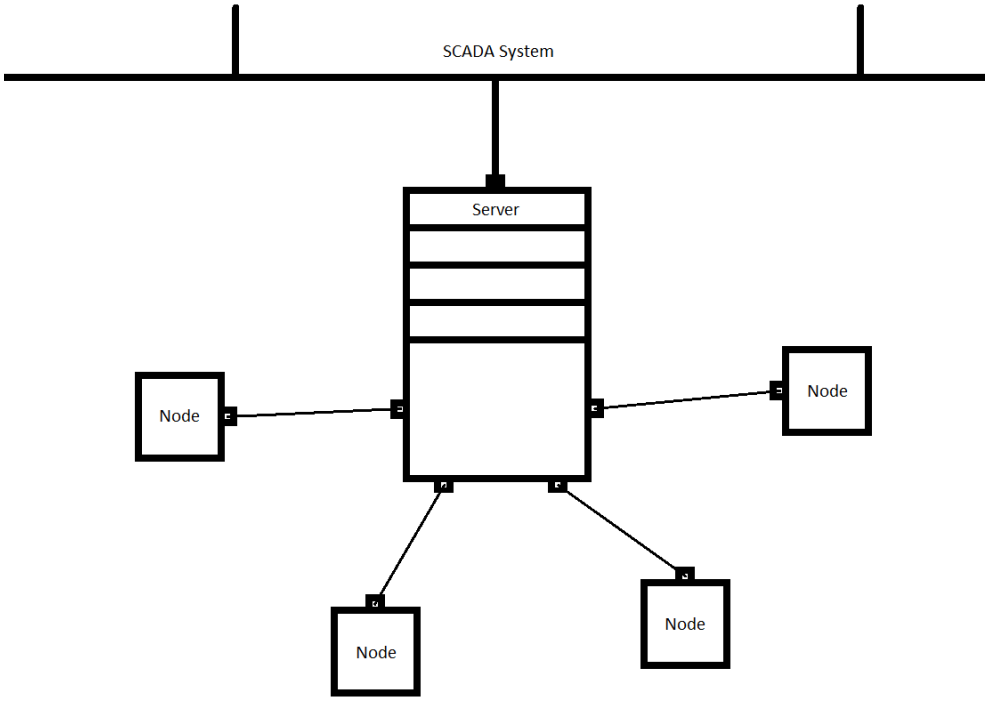


Abbildung 1: Server Client Architektur

Vorteile	Nachteile
Ressourcen- und datenintensive Anwendungen müssen nicht auf dem Knoten laufen: Geringere Hardwareanforderung an den Client	Großer und teurer Server wird benötigt
Rechenleistung für Switching Aufgaben wird vom Server übernommen	Ausfall des Servers bedeutet Ausfall des Netzes
	Schwer skalierbar: Neue Knoten müssen für den Server konfiguriert und eingerichtet werden

Tabelle 2: Vergleichstabelle Server Client Architektur

3.5.2 Peer-to-Peer

Bei dieser Architektur (Abbildung 2) werden alle Knoten über einen transparenten Switch miteinander verbunden. Um den Netzwerk-Traffic so klein wie möglich halten zu können, sollte der Switch mindestens auf Layer 3 des OSI Modells arbeiten. Der Switch ist außerdem mit einem übergeordneten SCADA System verbunden, aus welchem die Knoten ebenfalls Anfragen erhalten können. Damit jeder Knoten gleichberechtigt ist, werden auf allen Teilnehmern sowohl ein Server als auch ein Client installiert.
[Vgl. Schill, Springer, 2012, S. 36 - 39]

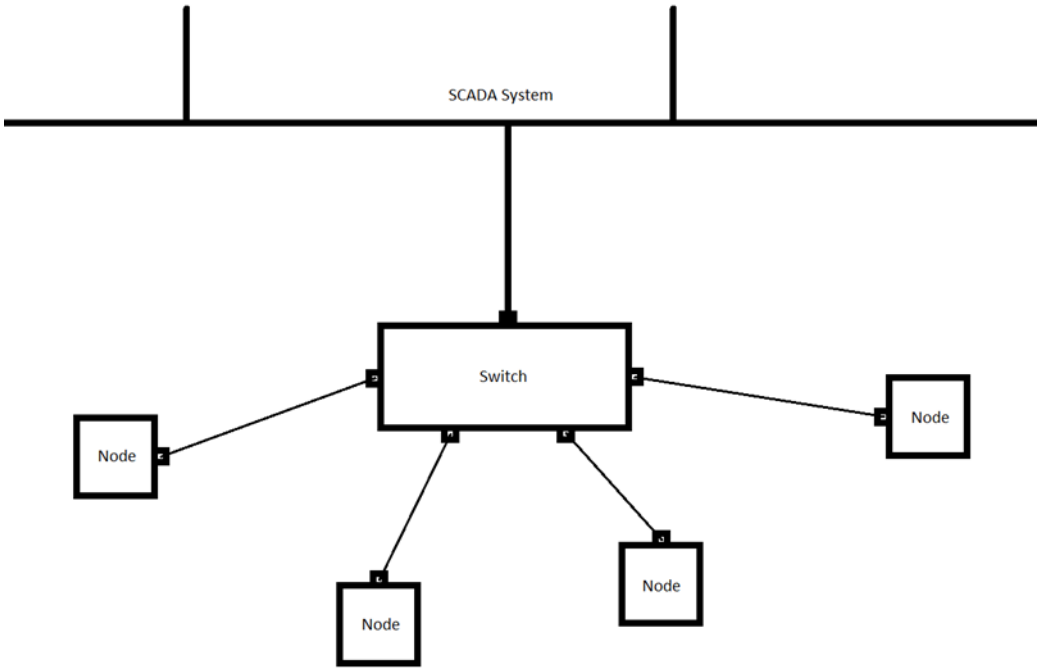


Abbildung 2: Peer-to-Peer Architektur

Vorteile	Nachteile
Leicht skalierbar: Neue Knoten können ohne großen Aufwand installiert werden	Unverzichtbare große Anwendungen müssen auf einem Knoten installiert werden: Größere Hardware Anforderungen
Gleichberechtigung aller Teilnehmer	Ausfall des Switches bedeutet Ausfall des Netzes

Tabelle 3: Vergleichstabelle Peer-to-Peer Architektur

Aufgrund der einfachen Skalierbarkeit und der Gleichberechtigung der Netzwerkteilnehmer ist die nachfolgende Arbeit für eine Peer-to-Peer Architektur entworfen worden.

4. Systemarchitektur

Im folgenden Kapitel wird das System, das Gegenstand dieser Arbeit ist, beschrieben. Zuerst wird ein grober Überblick über die Funktionalitäten des Systems gegeben, dann wird das System in kleinere Abschnitte unterteilt und detailliert erläutert. Zur besseren Veranschaulichung ist die Beschreibung an die Modelliersprache Unified Modeling Language (UML) angelehnt. Als Werkzeug zur Visualisierung wird Papyrus, ein Plugin für Eclipse, verwendet.

4.1 Use Case Diagram

Das Use Case Diagram in Abbildung 3 zeigt die Anwendung des Systems im laufenden Betrieb im Kontext mit den verschiedenen Benutzern. In diesem Beispiel besteht das Cluster aus zwei Knoten, die Anzahl der Netzwerkteilnehmer ist jedoch theoretisch nicht darauf beschränkt. Ebenfalls ist das System nicht an zwei Personen gebunden. Bevor das Netzwerk von einem oder mehreren Usern verwendet werden kann, muss der Systemadministrator die einzelnen Knoten einrichten. Sobald das System zum ersten Mal vom Administrator konfiguriert wird, kann der oder die Nutzende bestimmte Parameter der internen PLC bearbeiten. Die PLC interagiert mit den angeschlossenen IOs, an welche wiederum Sensoren oder Aktoren angeschlossen sind. Während des ganzen Prozesses kann der Knoten bestimmte Load Balancing Tasks an andere Knoten auslagern, oder auch Aufgaben von externen Knoten annehmen. Schließlich kann der Knoten wieder heruntergefahren werden.

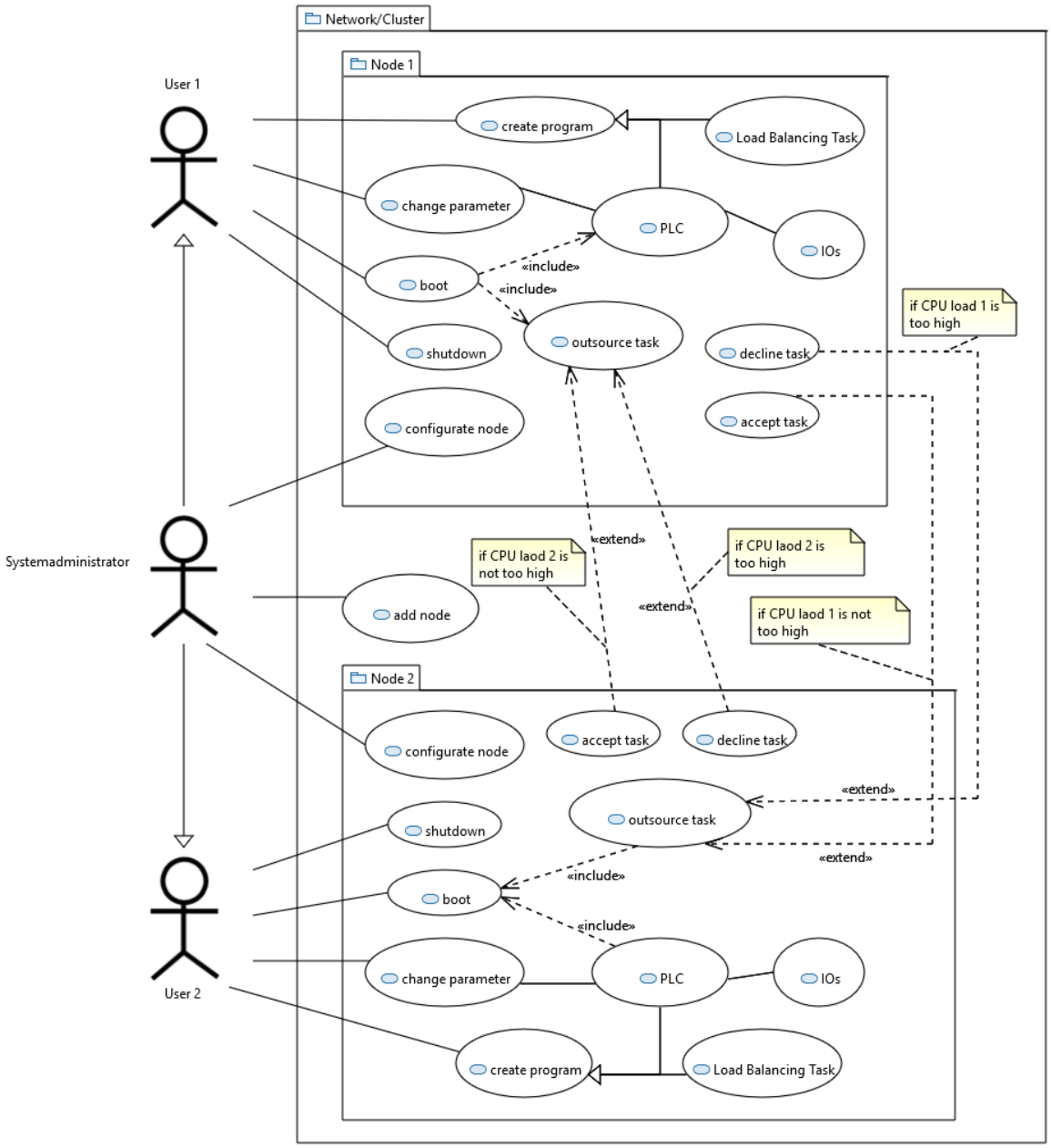


Abbildung 3: Use Case Diagram der Systemarchitektur im Kontext verschiedener User

4.2 Funktionelle Unterteilung

Wie in Abbildung 4 ersichtlich ist, sind vier große Hauptblöcke für die Funktionalität eines einzelnen Knotens verantwortlich.
Mit den Inputs und Outputs werden Sensoren und Aktoren bedient. Sie bilden somit die Schnittstelle zur realen Welt. Die IOs werden vom PLC Programm des Knotens gesteuert und ausgelesen. Durch die zyklische Abarbeitung des Programms, die durch die Laufzeitumgebung überwacht wird, wird eine deterministische Antwortzeit garantiert. Das PLC Programm kann asynchron zum Zyklus sogenannte Load Balancing Tasks aufrufen. Dabei handelt es sich vorwiegend um Aufgaben, die durch vergleichsweise lange Wartezeiten dominiert werden. Der Load Balancer verwaltet diese Tasks und kann bestimmte Aufgaben auf andere Knoten im Netzwerk auslagern. Die Auslagerung von Tasks geschieht, wenn die CPU des Knotens zu stark belastet ist. Zur Kommunikation zwischen den Knoten ist das Netzwerkinterface zuständig. Dies ist mit der bereits beschriebenen OPC UA Architektur aufgebaut.

Diese Hauptblöcke sind wiederum aus verschiedenen Softwareelementen zusammengesetzt. Jedes Element ist sowohl aus Variablen und Datenstrukturen, als auch aus eigenen Funktionen aufgebaut. Diese Elemente sind den in UML enthaltenen Klassen nachempfunden.

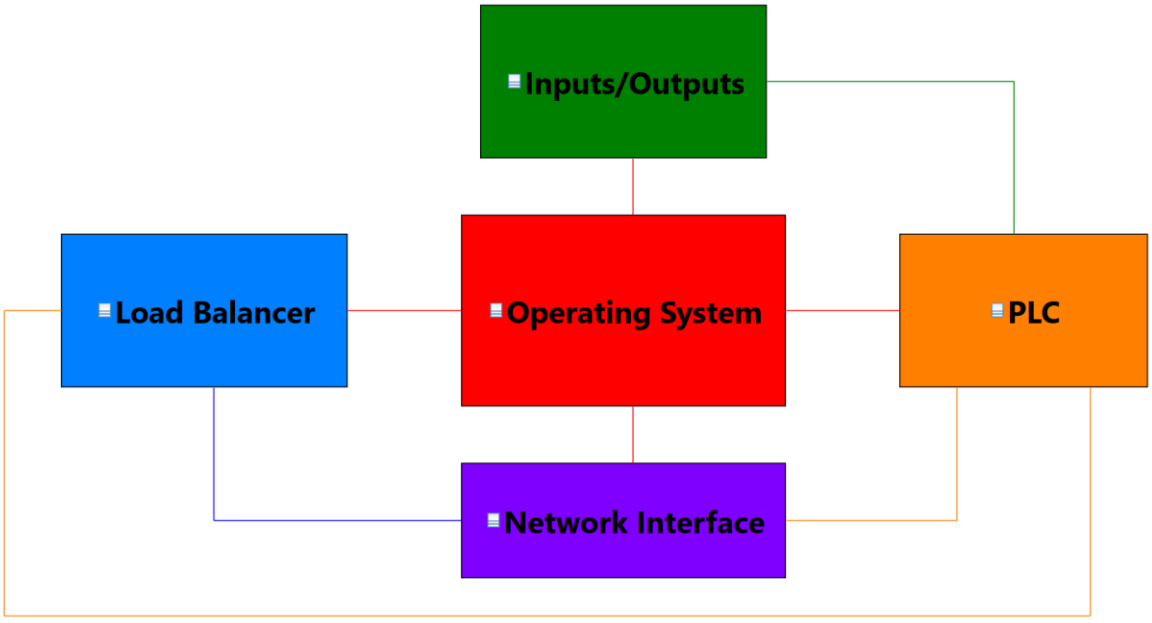


Abbildung 4: Unterteilung der Systemarchitektur

4.3 Class Diagram

Das an UML angelehnte „Class Diagram“ aus Abbildung 5 verbindet alle Softwareelemente zu einer vollständigen Systemarchitektur. Daraus wird ersichtlich, wie die Komponenten zusammenspielen und welche Softwareelemente voneinander abhängig sind.

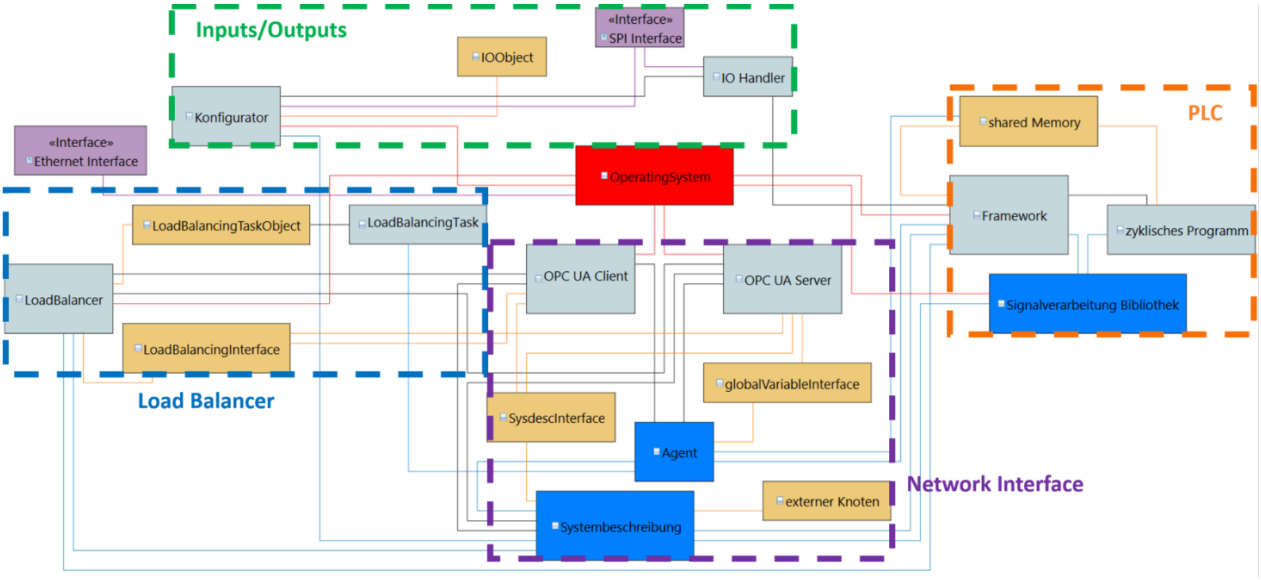


Abbildung 5: Class Diagram der gesamten Systemarchitektur

Die wichtigste Komponente der Systemarchitektur ist das Betriebssystem, welches in Rot markiert ist. Darauf aufgesetzt laufen alle weiteren Programme.

Die Schnittstellen des Knotens sind in violett gekennzeichnet. Das Ethernet Interface verbindet mehrere Knoten zu einem Netzwerk. Mit dem SPI Interface wird das GIO212 an den Knoten angeschlossen.

Bei den grauen Elementen handelt es sich um die eigentlichen Programme, die die konkreten Funktionalitäten des Knotens bereitstellen. Diese Programme greifen zum Teil auf die blauen Container zu. Container beinhalten Funktionen und Daten, die von den Programmen aufgerufen werden können - jedoch der Übersicht halber getrennt sind. In Gelb sind Elemente gekennzeichnet, die sich während der Laufzeit ständig ändern. Das sind insbesondere geteilte Hauptspeicher zur Interprozesskommunikation oder Objekte, die von den Programmen während der Laufzeit erstellt und bearbeitet werden.

Jede dieser Klassen ist aus eigenen Daten und Funktionen aufgebaut, welche in Anhang A aufgelistet sind.

4.4 Softwareelemente

Die zuvor unterteilten funktionellen Blöcke bestehen wiederum aus mehreren Softwareelementen. Diese Elemente bestehen wie die Objekte in UML aus Daten und Funktionen. Im weiteren Verlauf werden die genauen Aufgaben der einzelnen Elemente erläutert.

4.4.1 Inputs/Outputs

Konfigurator: Bevor das GIO212 benutzt werden kann, müssen die Kanäle konfiguriert werden. Dazu wird ein entsprechendes Bitmuster erzeugt und per SPI auf das GIO212 geladen. Dieser Bereich wurde bereits zum Teil von Klaus Jäger in seiner Arbeit behandelt. Die IO Kommunikation beinhaltet einen Konfigurator und einen IO Handler. Mit dem Konfigurator werden die IOs vor der Verwendung initialisiert. Diese Konfiguration soll über einen eigenen Port am Ethernet erfolgen. Somit kann durch einen einzigen Zugang jeder Knoten im Netzwerk konfiguriert werden.

IOObject: Das GIO212 wird über die SPI Schnittstelle an den Knoten angeschlossen. Der Konfigurator erstellt für jeden angeschlossenen GIO212 ein IOObject. Dieses Objekt verknüpft ein Chip Select der SPI Schnittstelle mit der entsprechenden Konfiguration der Hardware.

IO Handler: Der IO Handler liest die Ein- und Ausgabewerte des GIO212. Dafür müssen die erhaltenen Bitmuster interpretiert und ausgewertet werden. Anschließend vermittelt der IO Handler die Daten an die PLC. Umgekehrt übernimmt der IO Handler auch die Ausgabewerte der PLC und schreibt diese an das SPI Interface.

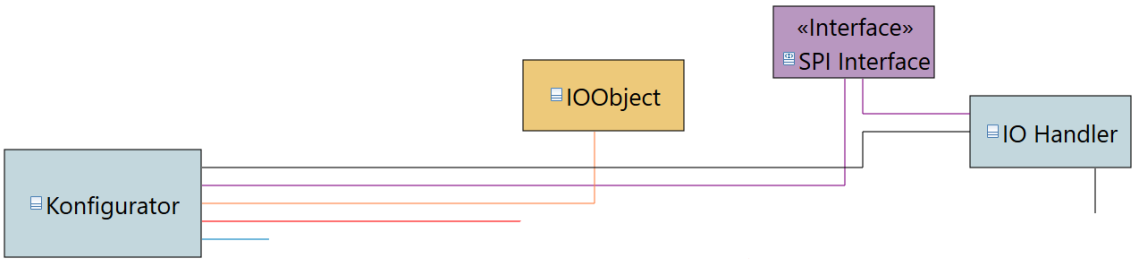


Abbildung 6: Softwareelemente des IO Blocks

4.4.2 Networkinterface

Das externe Auftreten des Knotens wird maßgeblich durch die folgenden Komponenten bestimmt (Abbildung 7). Damit alle Knoten untereinander gleichberechtigt sind, muss jeder über einen OPC UA Server und einen Client verfügen.

OPC UA Server:	Der Server stellt die globalen Variablen des Knotens für andere Knoten im Netzwerk zur Verfügung und präsentiert ihn somit nach außen. Diese Daten können dabei jederzeit verändert werden.
OPC UA Client:	Der Client kann die Daten eines Servers lesen oder schreiben. Dadurch können unterschiedliche Informationen übergeben werden. Dies könnten Normal Tasks oder Load Balancing Tasks sein.
Agent:	Der Agent übersetzt die bereitgestellten Variablen des Programms in die für den Server verständlichen OPC UA Datentypen.
globalVariableInt.:	Bei jeder Veränderung der Daten im globalVariableInterface werden diese dem Framework - bzw. dem PLC Programm - für den nächsten Durchlauf präsentiert. Bei einer Anfrage nach einem sogenannten „Normal Task“ konsultiert der Agent die Systembeschreibung und überprüft, ob die Anfrage zulässig ist. Normal Tasks sind das Lesen und Schreiben von Variablen oder Ein- und Ausgabewerten.
Sysdesc:	Die Systembeschreibung, an anderen Stellen auch Sysdesc (*englische Abkürzung* für system description) genannt, beschreibt die volle Funktionalität des Knotens. Dazu zählen die angeschlossenen und konfigurierten GIO212s, die Signalverarbeitungsbibliothek und das zyklische Programm selbst. Für jeden Knoten im Netzwerk wird eine Sysdesc erstellt und in einem externen Knoten-Objekt gespeichert.
Externer Knoten:	Dieses Objekt wird als Container für die IP Adresse und die externe Systembeschreibung verwendet.
SysdescInterface:	Das SysdescInterface ist ein Shared Memory zur Interprozesskommunikation und vermittelt die Daten zwischen der Systembeschreibung und dem OPC UA Server oder Client.

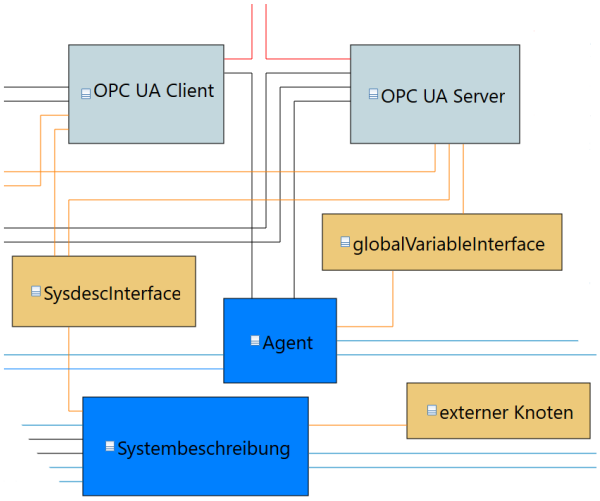


Abbildung 7: Softwareelemente des Netzwerkinterfaces

4.4.3 Load Balancer

Load Balancer: Der Load Balancer ruft die Load Balancing Tasks, die vom internen Programm oder von externen Knoten angefordert werden, auf. Außerdem entscheidet er, welche Tasks auf andere Knoten verschoben werden müssen. Wenn ein Task ausgelagert wird, muss zuerst in der Systembeschreibung ein anderer Knoten gesucht und ausgewählt werden.

Task Object: Für jeden gestarteten Load Balancing Task wird ein LoadBalancingTaskObject erzeugt. Dieses Objekt beschreibt, um welchen Task es sich handelt und wo dieser Task ausgeführt wird. Zusätzlich werden die aktuelle Priorität und die Prioritätsklasse aufgelistet.

LoadBalancingInt.: Zur Kommunikation von Load Balancer und dem OPC UA Server oder Client dient das LoadBalancingInterface. Wenn ein Task auf einen anderen Knoten ausgelagert werden muss oder ein externer Task angenommen wird, wird ein entsprechender Vermerk auf diesem Interface hinterlassen. Auch zur Kommunikation von externen oder internen Ergebnissen wird dieses Interface verwendet.

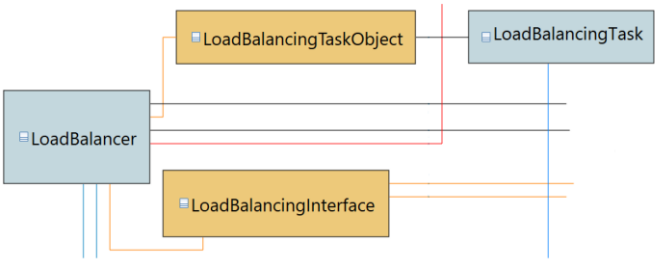


Abbildung 8: Softwareelemente des Load Balancer

4.4.4 PLC

PLC: Beim internen zyklischen Programm handelt es sich, neben den Load Balancing Tasks, um die eigentliche programmierbare Funktion des Knotens. Im Idealfall erfüllt das zyklische Programm eine konstante Laufzeit, ähnlich wie die Laufzeit einer PLC. Dafür wird eine Laufzeitumgebung benötigt, das wird in diesem Fall durch das Framework dargestellt. Asynchron zu diesem Zyklus können hier bestimmte Load Balancing Tasks aufgerufen werden.

Framework: Das Framework überwacht dabei die Zykluszeit des Programms, vermittelt die Ein- und Ausgänge mit dem IO Handler und sendet Anfragen für Load Balancing Tasks an den Load-Balancer. Durch einen eigenen Port am Ethernet kann das PLC Programm anwendungsspezifisch angepasst werden.

shared Memory: Mit dem shared Memory zwischen dem Programm und dem Framework werden Befehle und Daten ausgetauscht.

SignalLib: Die Signalverarbeitungsbibliothek (auch englisch SignalLib) beinhaltet Standardfunktionen, die das zyklische Programm verwenden kann. Der Programmierer oder die Programmiererin soll sich bei der Entwicklung des Programms komplett auf die SignalLib stützen können. Da einige Funktionen der SignalLib auch von der Hardware abhängig sein könnten, soll die Bibliothek nach Belieben aktualisiert werden können. Dies geschieht ebenfalls über einen eigenen Netzwerkport.

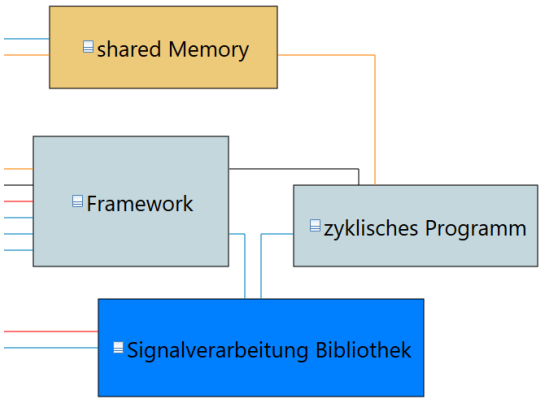


Abbildung 9: Softwareelemente der PLC

4.5 Auslagerung von Load Balancing Tasks

Der Load Balancer muss für jeden aufzurufenden Load Balancing Task entscheiden, ob dieser auf einen anderen Knoten ausgelagert wird oder ob der Task auf der internen CPU gestartet wird. Dies wird anhand von zwei Kriterien entschieden: der aktuellen Auslastung der CPU und der Spezialisierung des Tasks auf einen bestimmten Knoten. Diese Entscheidung trifft jeder Knoten, auch wenn er einen externen Task entgegennimmt. Zur Implementierung wird davon ausgegangen, dass auf jedem Knoten alle Load Balancing Tasks vorhanden sind, so dass lediglich die Task-Parameter übergeben werden müssen.

4.5.1 Auslastung des Knotens

Die Bestimmung der CPU-Auslastung des Knotens erfolgt mit den vom Betriebssystem bereitgestellten Daten. Linux beschreibt die Auslastung der CPU sowohl in Prozent als auch mit dem sogenannten Load Average. Diese Kennzahlen werden durch alle Prozesse bestimmt, die sich aktuell im Status Running befinden. Wenn ein System nicht voll ausgelastet ist, kann es sein, dass Linux Prozesse startet, die das System wieder aufräumen. Ein Beispiel dafür ist das Defragmentieren der Festplatte. Diese Prozesse werden mit einer sehr niedrigen Priorität gestartet und können von fast allen anderen Prozessen unterbrochen werden. Dennoch steigt die Auslastung der CPU gegen 100% und der Load Balancer würde jeden neuen Task auslagern. Um einer solchen Situation vorzubeugen, wird in dieser Implementierung die Knoten-Auslastung mit eigenen Mitteln erhoben.

Dafür werden alle Prozesse unter einer konfigurierbaren Priorität für die Berechnung vernachlässigt. Schließlich wird die prozentuale Auslastung aller verbliebenen Prozesse addiert. Auf der Basis dieses Wertes entscheidet der Load Balancer, ob bestimmte Prozesse ausgelagert werden. Laut dem Anwenderhandbuch von Bachmann electronic GmbH sollte die Auslastung einer Steuerung 70% nicht überschreiten. Damit ist noch genügend Reserve vorhanden, um einen transient overload zuzulassen. Bevor der Load Balancer nach einer Überlast interne Tasks wieder zulässt, muss die Auslastung unter einen bestimmten Wert sinken. Mit dieser Hysterese werden Schwingungen im Auslagerungsverfahren vermieden.

Die Grenzen für die minimale Priorität und die maximale Auslastung werden konfigurierbar gestaltet. Dadurch bleiben die Anwendungen systemunabhängig. Als Standardwerte wird eine Priorität von 20 und eine Auslastung von 70% vorgegeben.

4.5.2 Knotenpriorität

Die Knotenpriorität, an anderen Stellen auch mit node priority bezeichnet, ist ein Maß dafür, wie stark ein Task an einen speziellen Knoten gebunden ist. Basierend auf diesem Parameter wird eine Reihenfolge erstellt, die leicht und schwer auszulagernde Tasks sortiert. Zusätzlich zur Reihenfolge werden für alle Parameter statische konfigurierbare Grenzen definiert. Wenn diese Grenzen überschritten beziehungsweise unterschritten werden, wird ein Task nicht ausgelagert.

Die Knotenpriorität wird mithilfe der drei Aspekte Bandbreite der Interprozesskommunikation (IPC), Netzwerkbandbreite und CPU Auslastung errechnet.

K1 IPC Bandbreite (Daten/Zeit):	Dieser Wert beschreibt, wie stark Tasks auf einem Knoten wechselseitig voneinander abhängig sind. Dabei müssen auch zeitkritische Anforderungen beachtet werden. Dazu zählen zum Beispiel IO Operationen. Zur Kommunikation werden Shared Memories verwendet. Diese sind im Vergleich zu Pipes oder Message Queues schneller und generieren einen geringeren Overhead. Es werden Semaphoren zur Mutual Exclusion benötigt.
K2 Netzwerkbandbreite (Daten/Zeit):	Dieser Parameter beschreibt, wie viele Daten für die vollständige Funktion von diesem Task von einem Knoten auf einen anderen transportiert werden müssen. Dies ist im Wesentlichen durch die Größe des Inputs und des Outputs bestimmt, aber auch Daten die während der Laufzeit kopiert werden müssen, sind in diesem Parameter berücksichtigt.
K3 CPU Intensität (Auslastung/Laufzeit):	Dabei werden kleine und schnelle Aufgaben eher auf dem Knoten behalten, lange und CPU-lastige Tasks werden eher ausgelagert.

Die Bestimmung der tatsächlichen Knotenpriorität kann auf drei Ebenen erfolgen.

Bei der *statischen* Analyse durchsucht ein Parser den Programmcode nach IPC und Netzwerkaufrufen. Weiters bestimmt er die maximale Laufzeit sowie die Größe der benötigten Daten. Daraus kann dann die maximal benötigte IPC Bandbreite errechnet werden.

Die *dynamische* Analyse erfolgt während der Laufzeit. Hier wird bestimmt, wie viel IPC Daten versendet werden, wie viel Netzwerkkommunikation in Anspruch genommen wird und wie stark die CPU Auslastung ist. Diese Daten müssen über einen definierten

Zeitraum betrachtet werden, jedoch kann damit nicht bestimmt werden, wie sich dieser Task in der Zukunft verhält.

Eine weitere Form ist die Kombination aus *statisch und dynamisch*: Mit der statischen Methode werden die maximal möglichen Werte bestimmt. Im Vergleich dazu werden dynamisch die tatsächlichen Werte erhoben. Somit kann eine ungefähre Abschätzung getroffen werden, wie sich der Task verhalten wird.

Die somit bestimmten Daten werden in eigene Parameter K1, K2 und K3 umgerechnet. Diese Parameter ordnen sich auf einer numerischen Skala von 0 bis 9 an, wobei 9 der oberen Grenze und 0 der unteren Grenze entspricht. Die obere und die untere Grenze sind von der verwendeten Hardware abhängig und müssen bei jeder neuen Implementierung beachtet werden.

Die Reihenfolge der tatsächlichen Knotenpriorität wird durch die Formel in Gleichung 2 bestimmt.

$$NP = \frac{K1 + K2 + K3}{3} \tag{2}$$

Zur Implementierung ist die Knotenpriorität für jeden Task von Anfang an bekannt.

4.6 Activity Diagrams

Damit das System bestimmte Aufgaben erfüllen kann, sind einige Funktionen eine grundlegende Voraussetzung. Diese so genannten „Standard Szenarios“ werden in den folgenden Kapiteln vorgestellt und beschrieben. Mit Hilfe von Activity Diagrams soll die eigentliche Funktionalität umgesetzt werden.

4.6.1 Konfiguration der Hardware

Die Konfiguration der Hardware (Abbildung 10) erfolgt über einen eigenen Port. Diesen muss der Konfigurator zunächst mit readCommIN() auslesen. Damit während des Konfigurationsprozesses nicht auf die Hardware zugegriffen wird und gegebenenfalls falsche Werte ausgelesen werden, wird die Variable configMode gesetzt. Aus den angekommenen Paketen wird mit updateConfig() die HWFunctionList erstellt. In dieser Liste sind die aktuellen Konfigurationen vermerkt. Mit writeSPI() wird das dazugehörige Bitmuster erzeugt und an das GIO212 versendet. Die Konfiguration wird über das IOObject mit einem bestimmten GIO212 verknüpft. Schließlich wird durch die Funktion messageSysdesc() der Systembeschreibung eine Änderung der Hardware mitgeteilt und der configMode wieder zurückgesetzt.

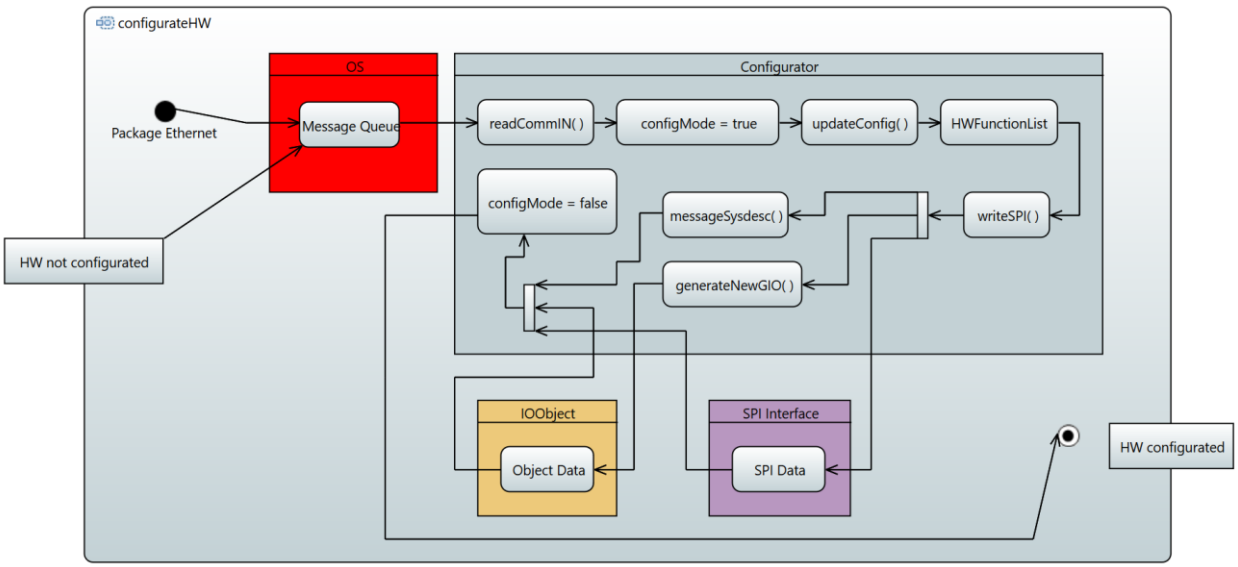


Abbildung 10: Activity Diagram Konfiguration der Hardware

4.6.2 Installation des Programms

Zur Installation des internen zyklischen Programms (Abbildung 11) wird ebenfalls ein eigener Port verwendet. Dieser wird vom Framework mit readCommIN() ausgelesen. Mit der Variable programmingMode wird dem System mitgeteilt, dass gerade ein neues Programm installiert wird. Durch updateProgram() werden die Pakete interpretiert und das vollständige Programm wird auf dem Filesystem abgespeichert. Anschließend muss der Systembeschreibung mitgeteilt werden, dass ein neues Programm installiert wurde. Durch das Zurücksetzen des programmingMode wird das neue zyklische Programm zur Verwendung freigegeben.

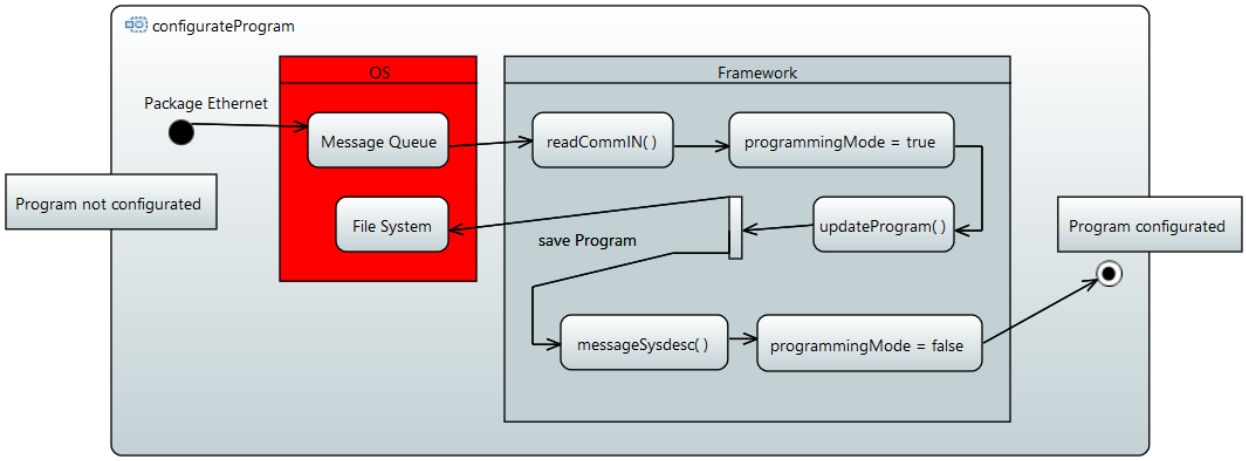


Abbildung 11: Activity Diagram Installation des Programms

4.6.3 Konfiguration der Signalbibliothek

Zur Installation einer neuen Signalbibliothek (Abbildung 12) wird der eigene Port mit readCommIN() ausgelesen. Durch Setzen der booleschen Variable updateMode wird ein unkontrollierter Zugriff auf die alte Signalbibliothek verhindert. Mit der Funktion selfUpdate() wird das FunctionsArray erzeugt, welches die installierten Funktionen dem System präsentiert. messageSysdesc() signalisiert der Systembeschreibung eine neue Bibliothek, woraufhin eine neue interne Systembeschreibung erstellt wird. Nach dem Zurücksetzen von updateMode ist die Konfiguration abgeschlossen.

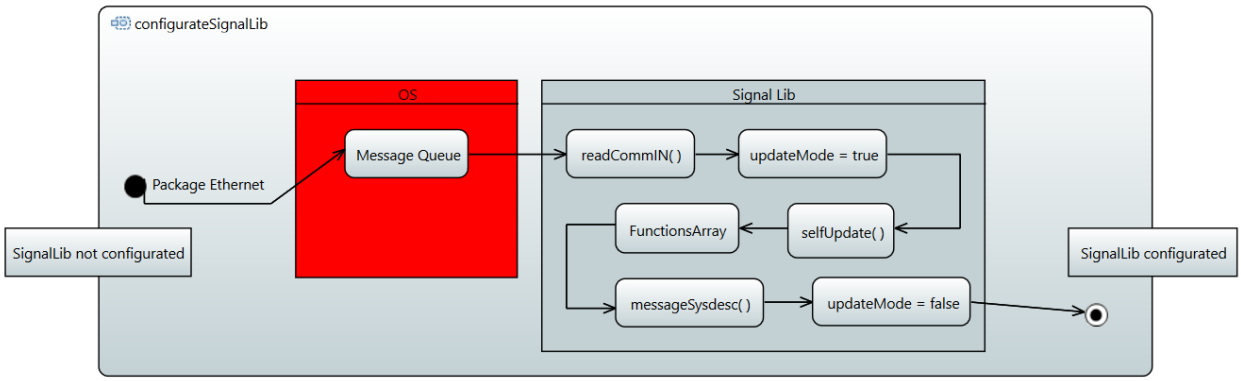


Abbildung 12: Activity Diagram Konfiguration der Signalbibliothek

4.6.4 Installation von Load Balancing Tasks

Die Installation eines neuen Load Balancing Tasks (Abbildung 13) beginnt mit einem Paket am Port des Load Balancer. Dieses Paket wird mit readCommIN() gelesen. Damit bereits installierte und neue Tasks nicht kollidieren, wird die Variable updateMode gesetzt. Durch updateLoadBalancingTask() werden die Pakete interpretiert und im LoadBalancingTaskArray dargestellt. Die Systembeschreibung wird mit messageSysdesc() über die neuen Tasks informiert. Schließlich wird die Variable updateMode zurückgesetzt und der Load Balancer wieder freigegeben.

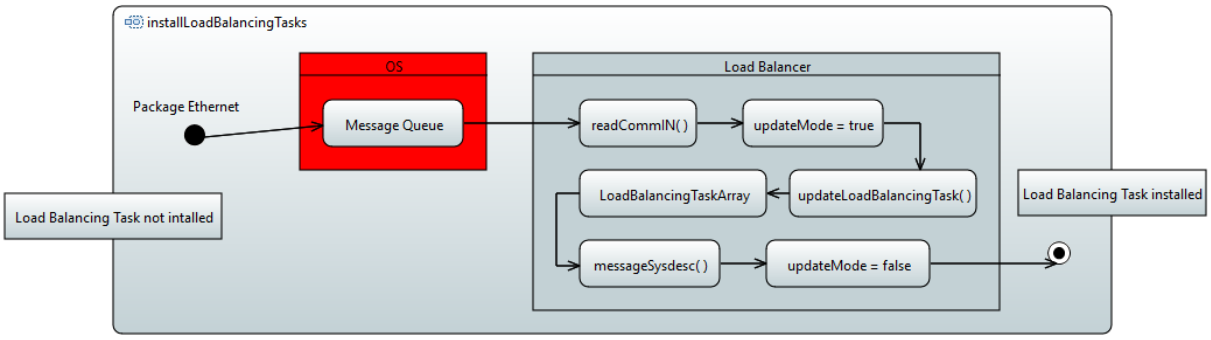


Abbildung 13: Activity Diagram Installation von Load Balancing Tasks

4.6.5 Normalen Task annehmen

Das Annehmen von „normalen Tasks“ wird in der Abbildung 14 dargelegt. Der OPC UA Server interpretiert die ankommenden Befehle und schreibt in (beziehungsweise liest) das Global Variable Interface. Damit sichergestellt ist, dass die geschriebenen Werte auch die Kriterien für eine weitere Verarbeitung durch das zyklische Programm erfüllen, überprüft der Agent mit checkInternalNormalTask(), ob die Systembeschreibung diese Werte erlaubt. Sind die Werte nicht erlaubt, wird die Funktion generateSysdescError() aufgerufen und der Herkunftsknoten wird benachrichtigt. Ist die Operation valide, wird das Global Variable Interface mit den Funktionen writeGlobalVariables() und writeGlobalOutputs() ins „shared Memory“ zwischen Framework und Programm kopiert. Schließlich wird das Programm aufgerufen und die Variablen, Inputs und Outputs für die interne Logik verwendet. Da während dieses Prozesses die Daten verändert werden können, muss das Global Variable Interface durch die Funktionen getGlobalVariables(), getGlobalInputs() und getGlobalOutputs() auf den neuesten Stand gebracht werden.

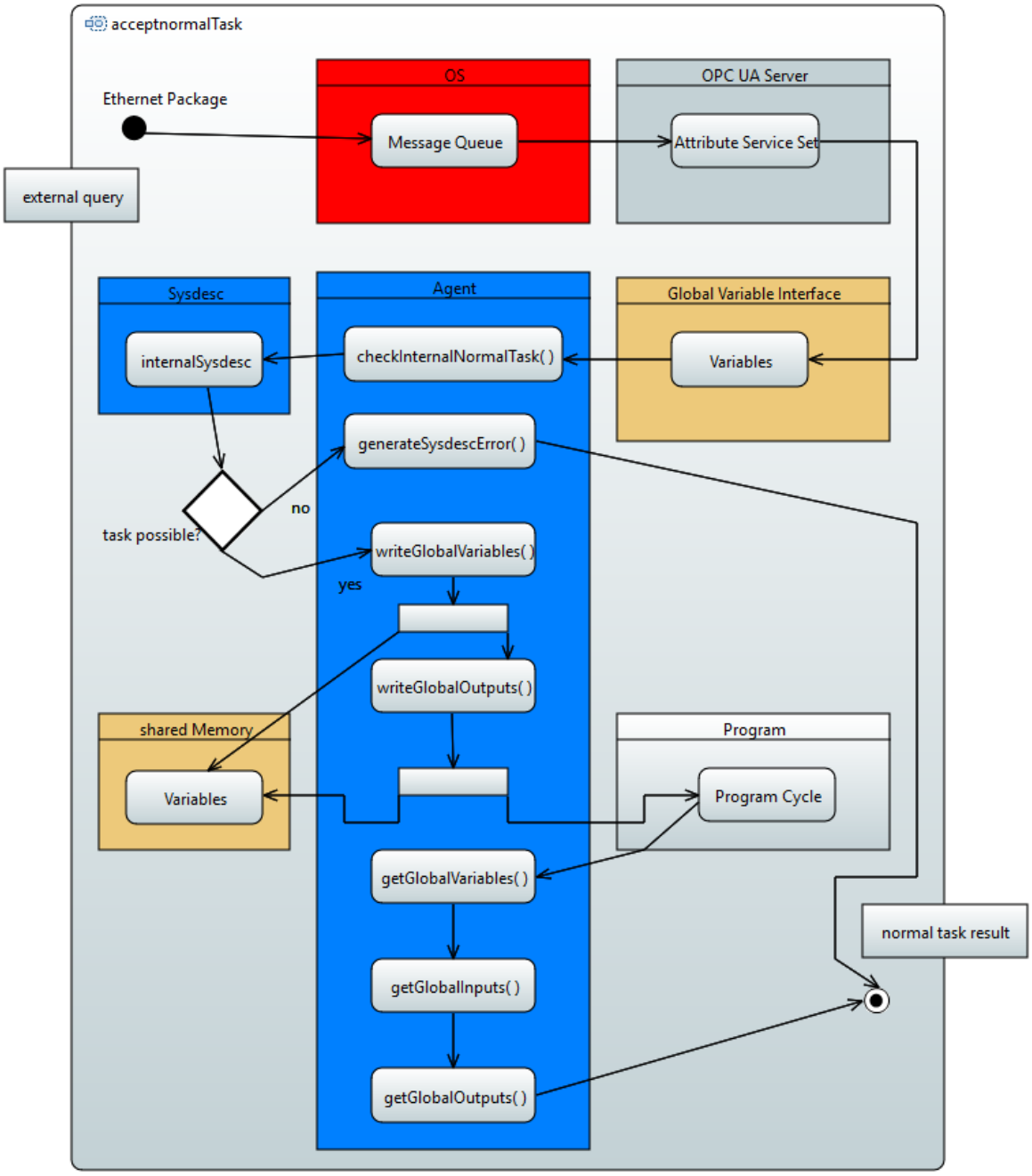


Abbildung 14: Activity Diagram Normalen Task annehmen

4.6.6 Normalen Task versenden

Das Versenden von „normalen Tasks“ wird in Abbildung 15 erläutert. Zunächst benötigt das interne zyklische Programm einen Input, Output oder eine Variable von einem anderen Knoten, um die aktuelle Berechnung durchführen zu können. Dies muss zunächst dem Framework über das „shared Memory“ mitgeteilt werden. Die Funktion readSharedRegister() des Frameworks liest diese Daten aus und interpretiert sie. Mit generateExternalNormalTask() wird dem Agenten mitgeteilt, welche Variablen von welchem Knoten geschrieben oder gelesen werden sollen. Der Agent durchsucht

zunächst die Systembeschreibung nach dem angeforderten Knoten und überprüft, ob diese die angeforderten Variablen besitzt. Falls die Operation nicht durchgeführt werden kann, wird dies mit generateSysdescError() dem Programm mitgeteilt. Wenn der Task der Systembeschreibung nicht widerspricht, wird durch forwardNormalTask() die Aufgabe dem OPC UA Client mitgeteilt. Die Antwort des Clients wird über die Funktion forwardNormalTaskResult() an das Framework weitergeleitet. Von da wird dann die Antwort über das „shared Memory“ an das interne zyklische Programm weitergegeben.

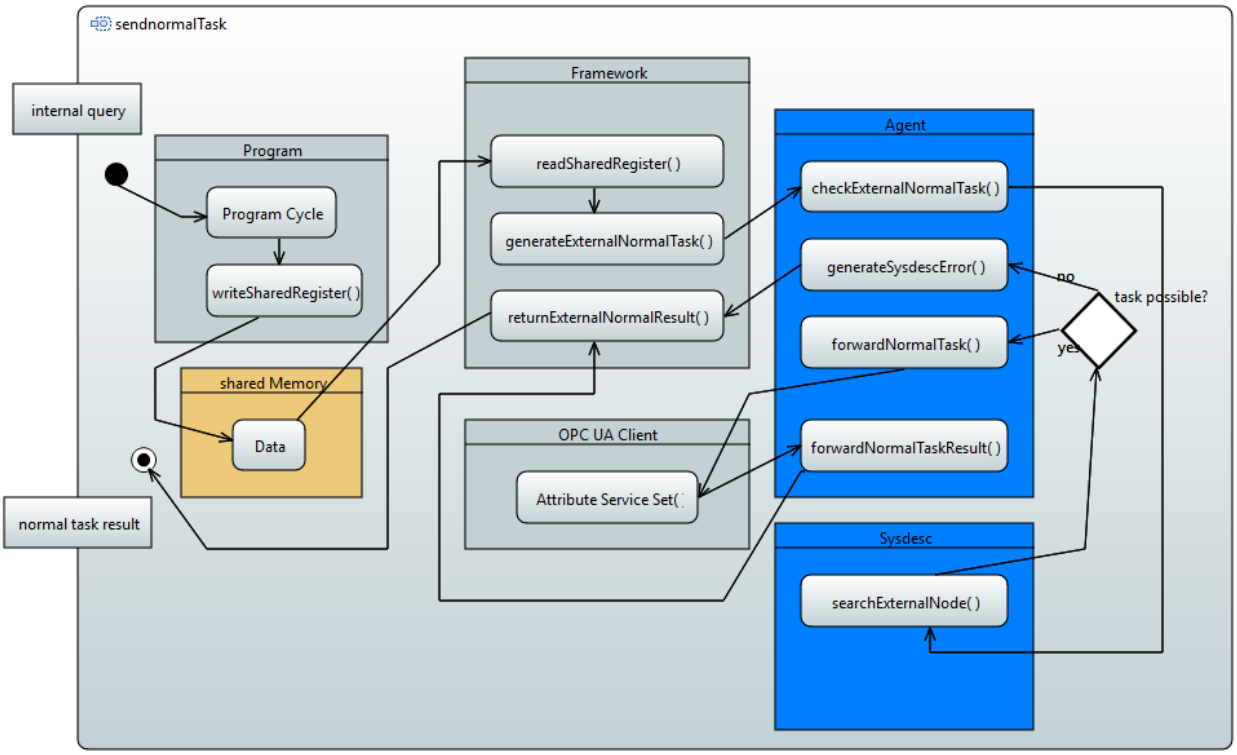


Abbildung 15: Activity Diagram Normalen Task versenden

4.6.7 Interne Systembeschreibung generieren

Die Systembeschreibung (Abbildung 16) wird insbesondere bei vier Ereignissen neu erstellt: Bei einer Änderung der Hardware-Konfiguration, wenn die Signalverarbeitungsbibliothek überarbeitet wird, wenn ein neues Programm installiert wird oder wenn neue Load Balancing Tasks installiert werden. Jedes dieser Ereignisse erzeugt ein Event und teilt so der Systembeschreibung eine aktuelle Änderung mit. Durch readSignalLib() wird die Signalverarbeitungsbibliothek gelesen und abgespeichert. Mit readHWconfig() liest die Systembeschreibung die HWFunctions Liste des Konfigurators. Mit interpreteProgram() wird das interne Programm nach globalen Variablen durchsucht, die im Zuge eines Normalen Tasks bearbeitet werden können. Durch readLoadBalancingTaskArray() werden neue Load Balancing Tasks in die Systembeschreibung eingebettet. Die Funktion writeSysdescInterface() beschreibt das SysdescInterface. Über das InternalSysdescInterface und publishInternalSysdesc kann

schließlich dem OPC UA Server mitgeteilt werden, dass eine neue Systembeschreibung erstellt wurde.

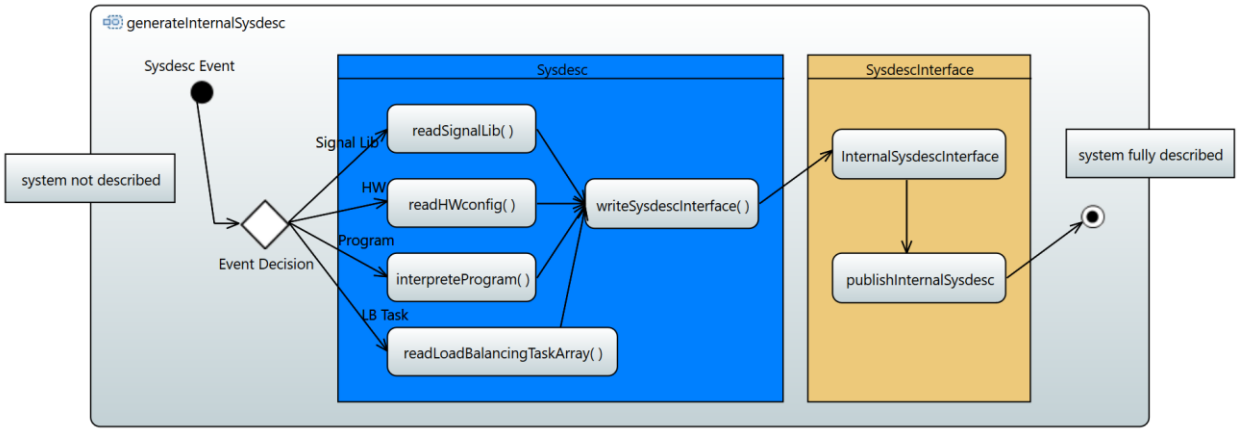


Abbildung 16: Activity Diagram Interne Systembeschreibung generieren

4.6.8 Externe Systembeschreibung generieren

Der OPC UA Client (Abbildung 17) findet einen neuen Server im Netzwerk, welcher eine Systembeschreibung bereitstellt. Diese wird mit der Funktion writeSysdescInterface() in das ExternalSysdescInterface kopiert. Von dort können die Daten von der Systembeschreibung mit readSysdescInterface() ausgelesen und abgespeichert werden. Die Funktion generateExternalNode() erstellt ein External Node Object, welches die IP Adresse und die dazugehörige Systembeschreibung vereint.

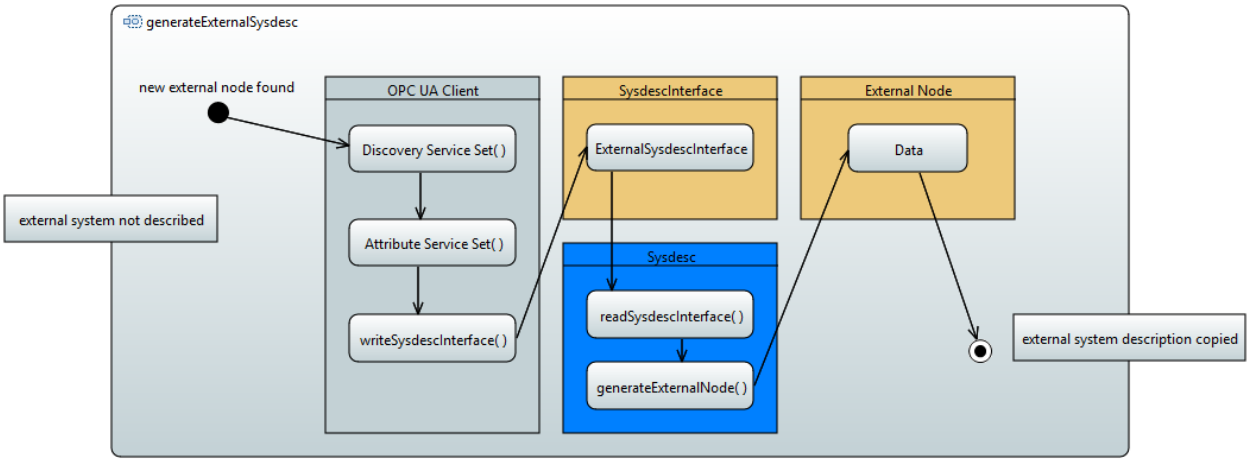


Abbildung 17: Activity Diagram Externe Systembeschreibung generieren

4.6.9 Load Balancing Task annehmen

Das Versenden von Load Balancing Tasks wird in den Abbildungen 18 und 19 erklärt. Der OPC UA Server bekommt eine Anfrage von einem anderen Knoten für die Ausführung eines Load Balancing Tasks. Der Server gibt die Anfrage mittels der Funktion writeLoadBalancingTaskInterface() an das externalTaskInterface weiter. Von dort wird es vom Load Balancer mit der Funktion readLoadBalancingInterface() gelesen. Zunächst wird mittels der Systembeschreibung überprüft, ob der gewünschte Task auf dem Knoten installiert ist. Wenn die Ausführung des Tasks möglich ist, also die Systemauslastung nicht zu hoch und der Task installiert ist, kann die Aufgabe bearbeitet werden. Andernfalls wird mit generateSysdescError() über das externalAnswerInterface eine entsprechende Meldung an den Auftraggeber gesendet. Ist die Ausführung möglich, wird zunächst mit generateLoadBalancingTaskObject() ein Objekt erzeugt. Dieses Objekt enthält alle Informationen, die für die Bearbeitung benötigt werden. Über setLoadBalancingTaskPriorityClass() wird dem Task eine Prioritätsklasse zugeteilt. Innerhalb dieser Klasse kann die Priorität dynamisch verändert werden. Mit callLoadBalancingTask() wird schließlich der eigentliche Task ausgeführt. Nach Beendigung der Aufgabe wird das erzeugte Objekt wieder gelöscht und die Antwort wird über forwardLoadBalancingTaskResult() und writeLoadBalancingInterface() an das externalAnswerInterface weitergegeben. Von dort stellt der Server die Antwort für den ursprünglichen Knoten zur Verfügung.

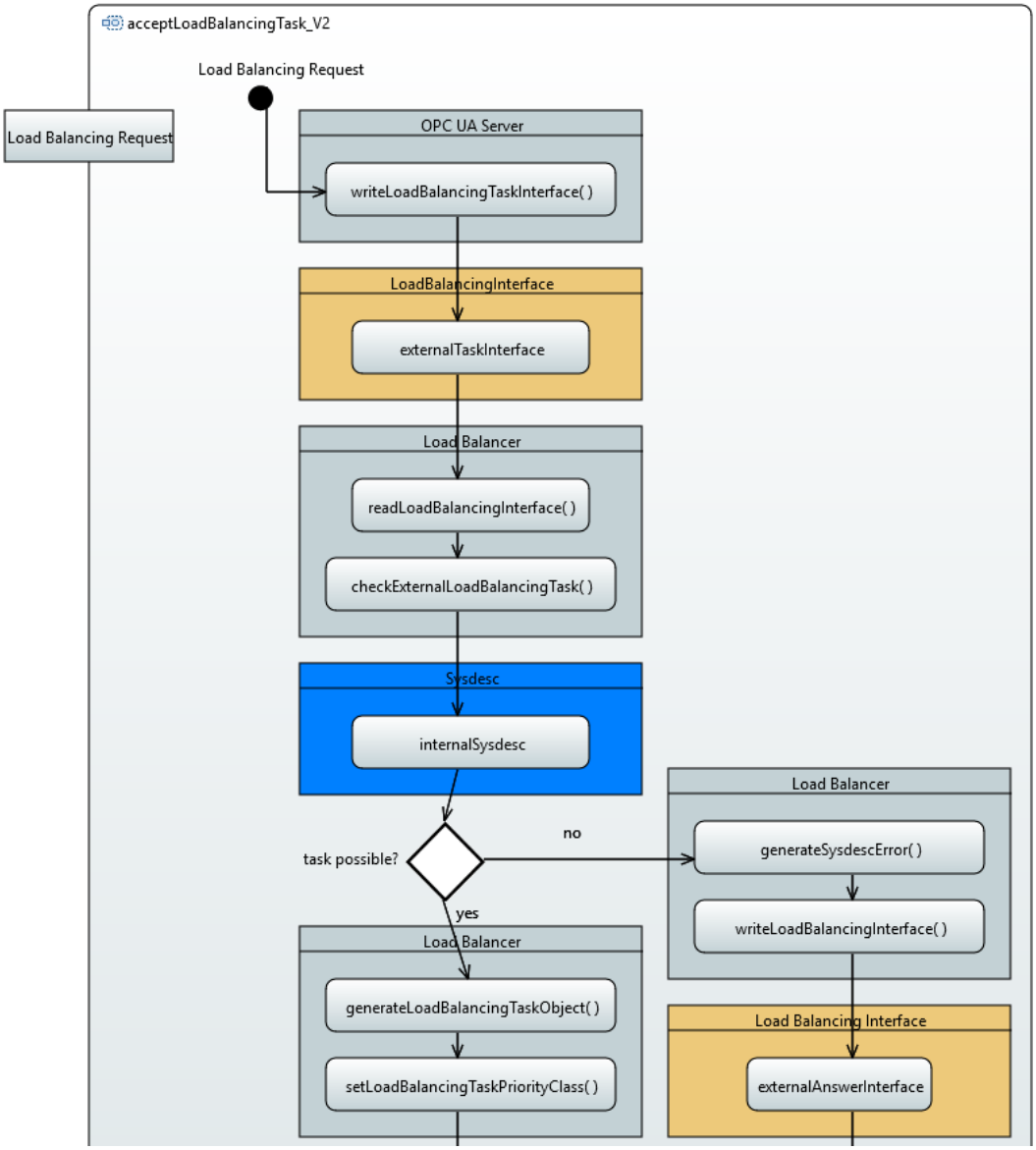


Abbildung 18: Activity Diagram Load Balancing Task annehmen Teil 1

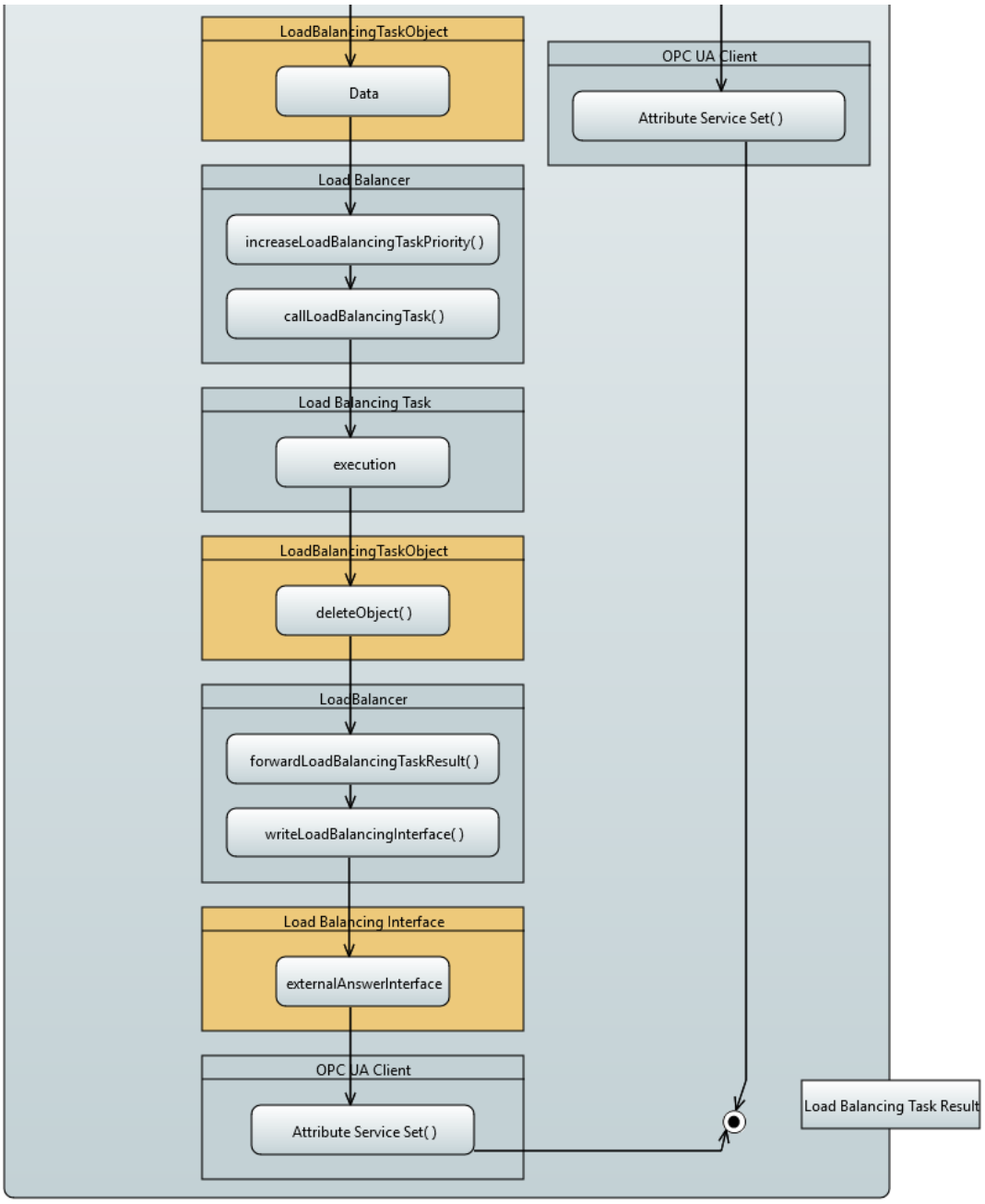


Abbildung 19: Activity Diagram Load Balancing Task annehmen Teil 2

4.6.10 Load Balancing Task versenden

Das Annehmen von Load Balancing Tasks wird in den Abbildungen 20 und 21 dargelegt. Zuerst muss das zyklische Programm einen Load Balancing Task aufrufen. Dafür wird ein entsprechender Befehl über das shared Register an das Framework übergeben. Dieses interpretiert den Befehl und schickt die Anfrage weiter an den Load-Balancer. Der Load Balancer berechnet zuerst die Knoten Priorität mit calcNodePriority(). Mit calcSystemLoad() sind nun beide Parameter zur Auslagerungsentscheidung bekannt. Zunächst wird das Load Balancing Task Object erzeugt und die Linux Priorität mit setLoadBalancingTaskPriorityClass() festgelegt. Bevor der Task gestartet wird, muss entschieden werden, ob der Task ausgelagert werden muss.

Wird der Task ausgelagert, muss ein geeigneter Knoten mit searchExternalNode() aus der Systembeschreibung gesucht werden. Ist ein Knoten gefunden, beschreibt der Load Balancer das Load Balancing Interface. Dies wird vom Client gelesen und der Task wird an den ausgewählten Server gesendet. Sobald der Task fertig ist, schreibt der Client das Ergebnis mit writeLoadBalancingTaskInterface() an das internalAnswerInterface. Der Load Balancer liest wiederum das Ergebnis aus dem Interface und schickt über forwardLoadBalancingTaskResult() die Antwort ans Framework.

Wird der Task intern abgehandelt, muss die Systembeschreibung nicht durchsucht werden. Stattdessen startet der Load Balancer den Task mit callLoadBalancingTask(). Wird dieser Task eine bestimmte Zeit lang nicht ausgeführt, kann die Linux Priorität mit increaseLoadBalancingTaskPriority() erhöht werden. Sobald der Task ausgeführt ist, wird ebenfalls über forwardLoadBalancingTaskResult() das Ergebnis ans Framework zurückgegeben. Im gleichen Zug muss noch das Load Balancing Task Object gelöscht werden. Das Framework gibt schließlich über writeSharedRegister() das Ergebnis ins „shared Memory“, von wo aus das zyklische Programm auf die Antwort zugreifen kann.

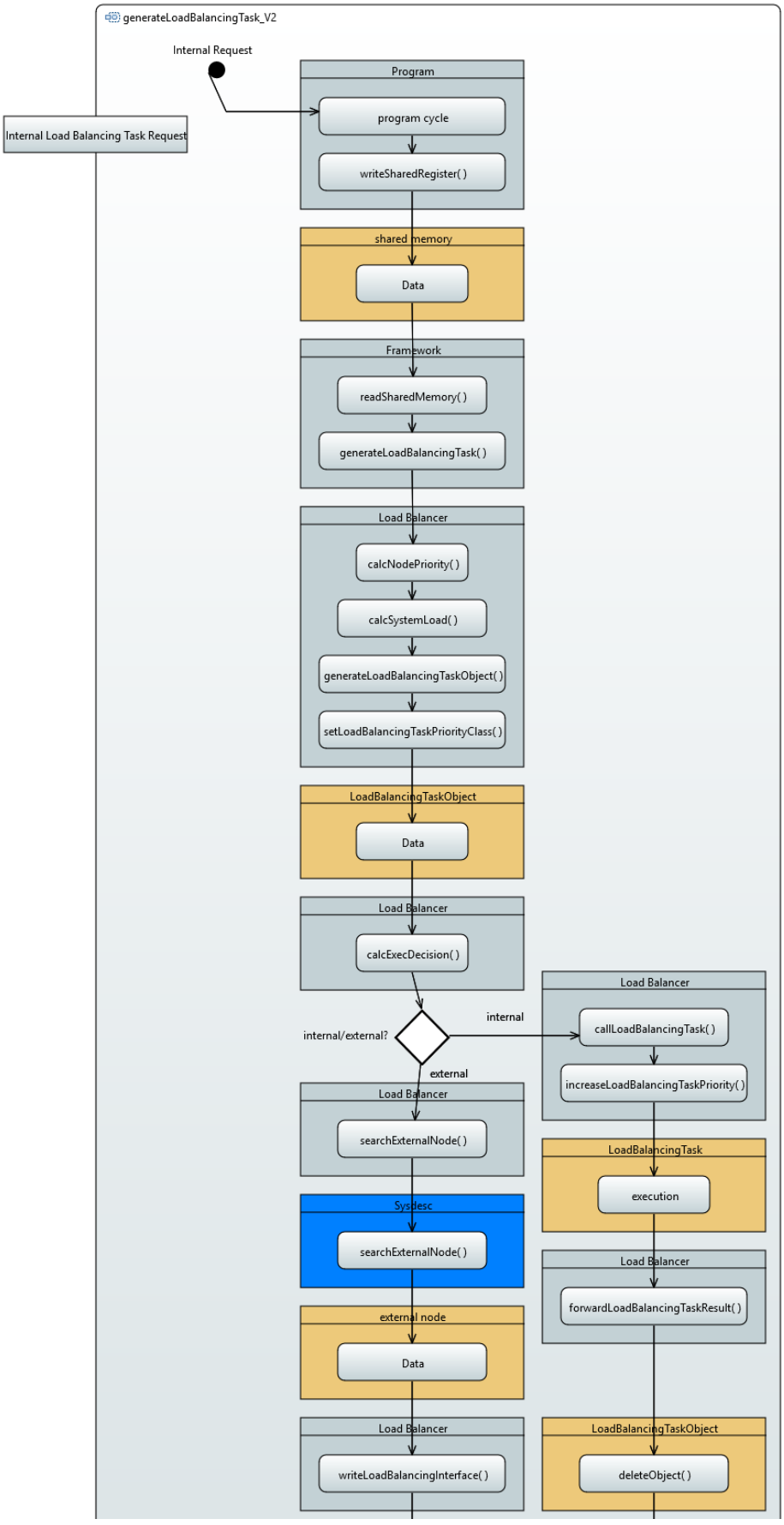


Abbildung 20: Activity Diagram Load Balancing Task versenden Teil 1

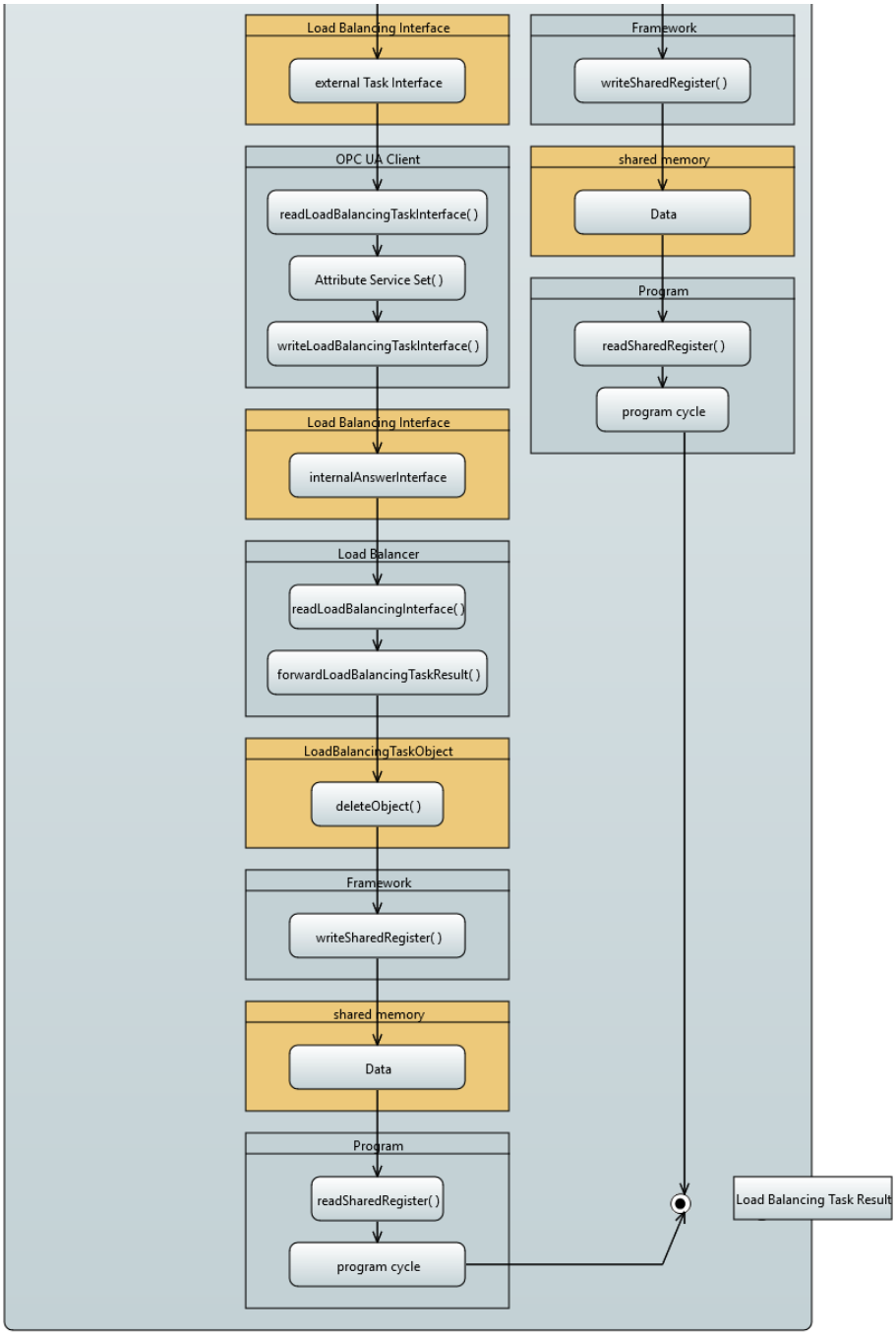


Abbildung 21: Activity Diagram Load Balancing Task versenden Teil 2

4.6.11 Laufzeitumgebung und Programmablauf

Der zyklische Programmablauf wird in den Abbildungen 22 und 23 verdeutlicht. Die Laufzeitumgebung beinhaltet das zyklische Programm, welches in einer Programmiersprache der Norm EN 61131 Teil 3 entwickelt wurde. Der Beginn des ersten Zyklus startet im Framework. Das Framework startet einen Timer, der zur Überwachung der Laufzeit dient. Zunächst werden die Inputs aktualisiert und über das „shared Memory“ an das zyklische Programm weitergegeben. Nach dem Beschreiben des geteilten Registers wird mit invokeCyclicProgram() das zyklische Programm gestartet. Dieses liest zunächst das geteilte Register aus und interpretiert die darin enthaltenen Befehle. Nach der Ausführung des Programms wird das Register wieder beschrieben. Dadurch werden dem Framework Informationen über den Ablauf des Programms mitgeteilt und dieses kann gewünschte Load Balancing Tasks aufrufen. Bevor die Ausgabewerte an den IO Handler weitergegeben werden, muss der Agent die von außen an ihn herangetragenen Werte vom globalVariableInterface ins „shared Memory“ kopieren. Als nächstes werden alle geteilten Inputs, Outputs und Variablen ins globalVariableInterface kopiert. Damit hat der Agent seine Aufgaben erledigt. Das Framework liest das geteilte Register und führt die darin enthaltenen Befehle aus. Schließlich müssen die geteilten Outputs gesetzt werden. Dazu liest die Funktion readSharedOutputVariables() die Outputs aus dem „shared Memory“ und gibt diese über die Funktion writeOutputToIOHandler() an den Output Buffer des IO Handler weiter. Dieser generiert den benötigten Bitstring und beschreibt das GIO212 über die SPI Schnittstelle. Mit readSPI() werden die Inputs gelesen und über intrepreteBitString() werden die erhaltenen Informationen interpretiert. Schließlich schreibt writeInputs() die Variablen an den Buffer, von wo diese im nächsten Zyklus vom Framework wieder abgeholt werden können. Bevor der folgende Zyklus von neuem startet, werden der Timer gestoppt und die Zykluszeit überprüft.

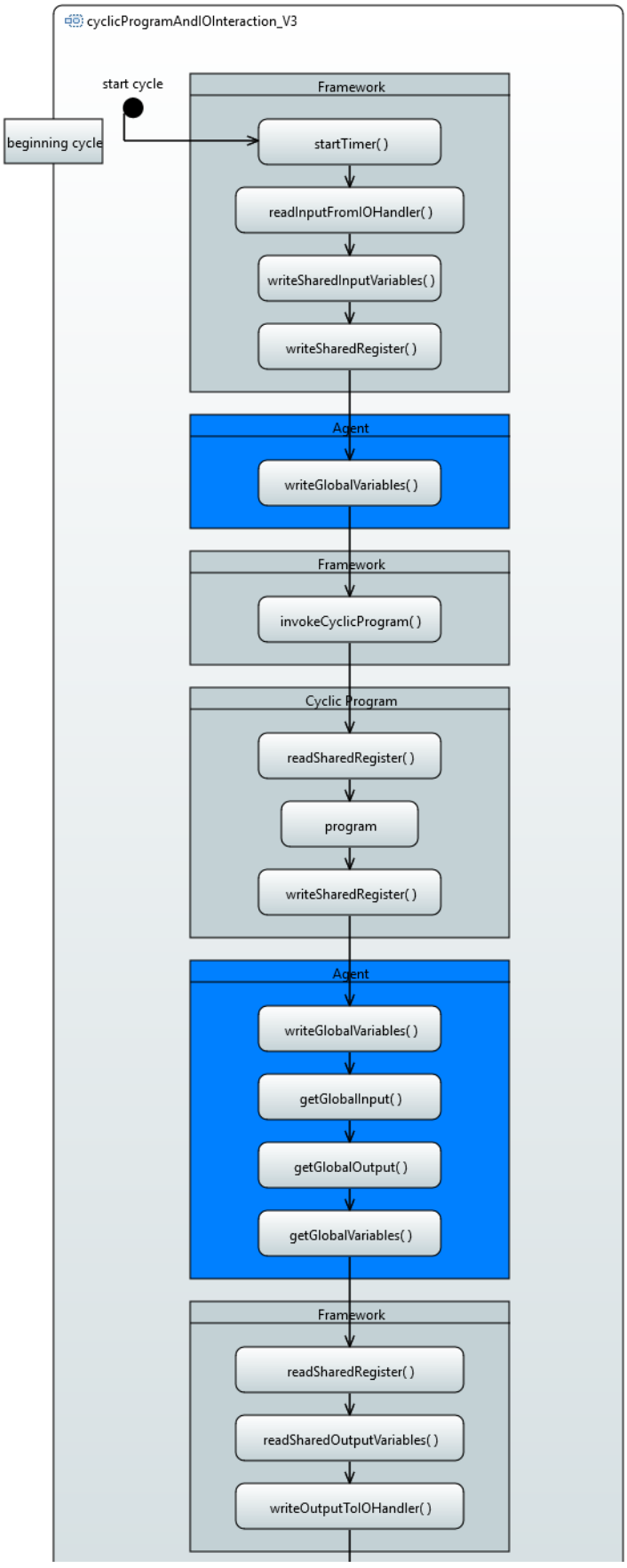


Abbildung 22: Activity Diagram Laufzeitumgebung und Programmablauf Teil 1

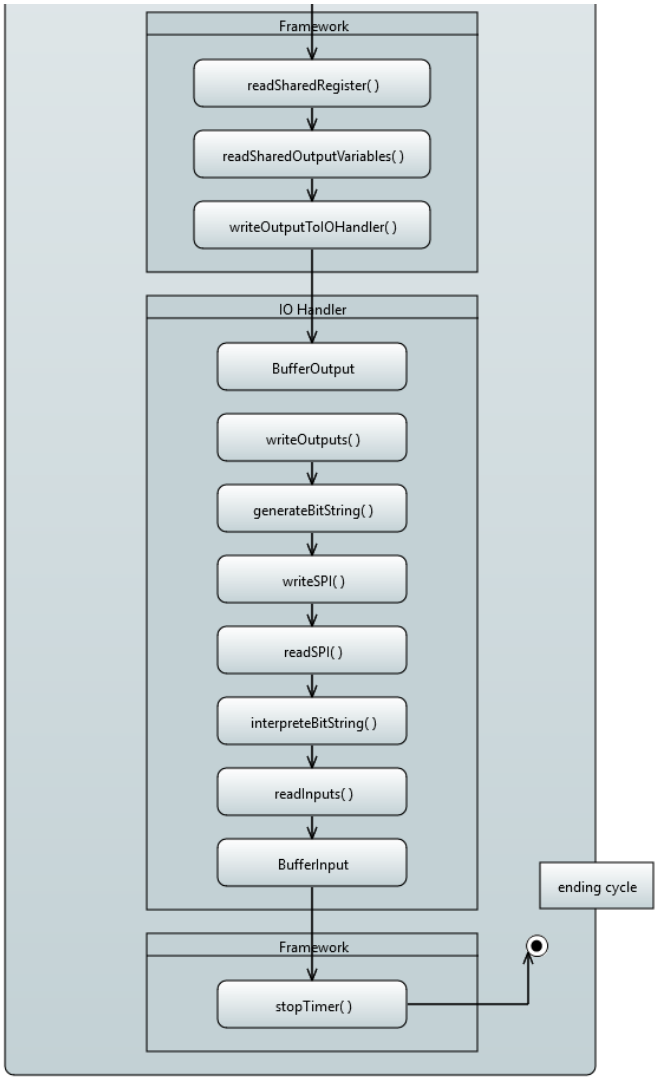


Abbildung 23: Activity Diagram Laufzeitumgebung und Programmablauf Teil 2

4.6.12 Booten

Während des Bootvorgangs (Abbildung 24) eines Knotens müssen sowohl die Hardwarekonfiguration als auch die installierte Signalbibliothek wieder erstellt werden. Dafür werden jeweils die Funktionen retrieveHWConfig() beziehungsweise retrieveSignalLib() verwendet. Schließlich werden alle weiteren Programme gestartet.

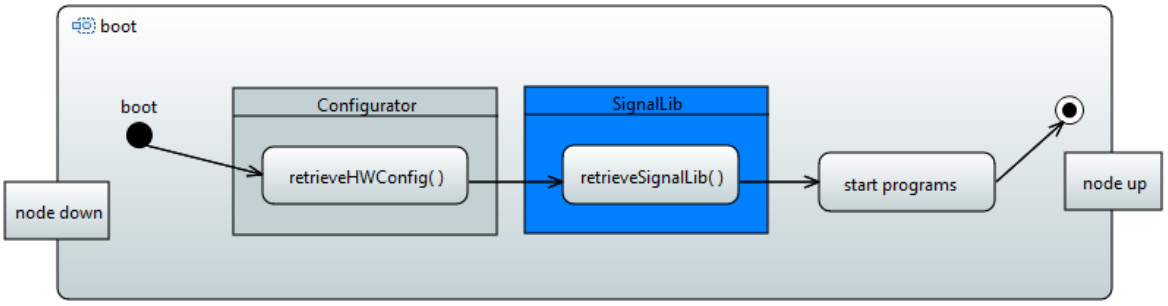


Abbildung 24: Activity Diagram Booten

4.6.13 Herunterfahren

Beim Herunterfahren (Abbildung 25) eines Knotens müssen alle Programme ordnungsgemäß beendet werden, damit eine fehlerfreie Funktionsweise der restlichen Knoten gesichert ist. So muss der Load Balancer für jeden extern ausgeführten Load Balancing Task eine entsprechende Errornachricht an den Auftraggeber versenden. Mit der Funktion shutdownError() wird die Nachricht generiert, worauf diese über das externalAnswerInterface des Load Balancing Interface an den OPC UA Server weitergegegeben wird. Sobald dies geschehen ist, kann das Load Balancing Task Object gelöscht werden. Damit beim nächsten Booten die Hardwarekonfiguration und die installierte Signalbibliothek wieder vorhanden sind, müssen diese mittels saveHWConfig() und saveSignalLib() auf dem Filesystem gespeichert werden. Zudem werden die external Node Objects von der Systembeschreibung freigegeben.

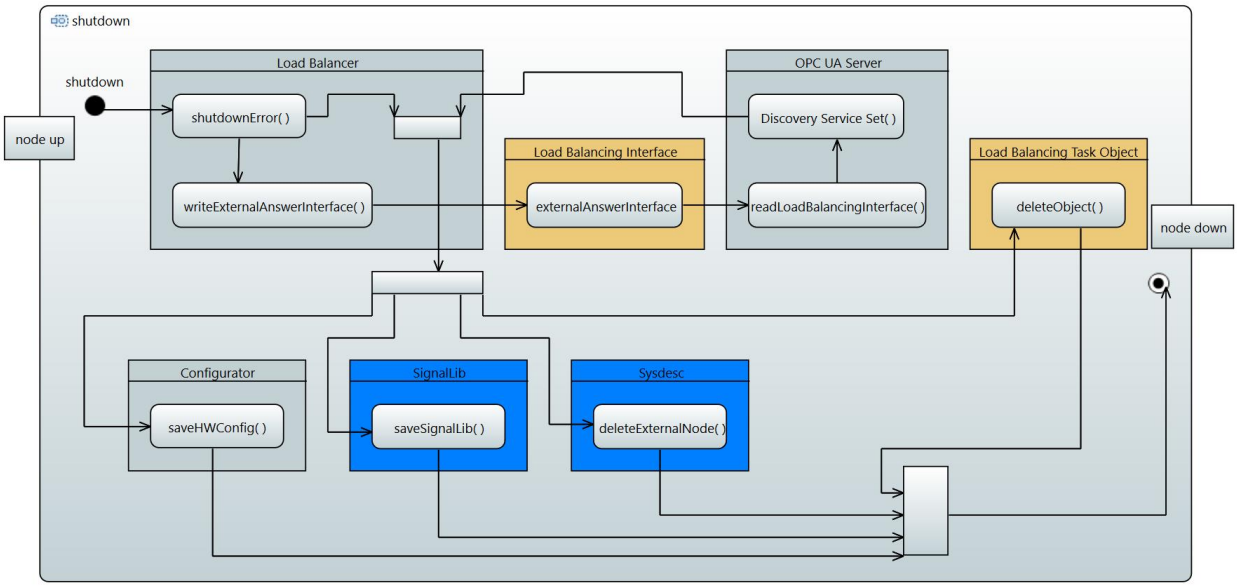


Abbildung 25: Activity Diagram Herunterfahren

4.7 Erweiterungsmöglichkeiten des Systems

Die aktuelle Entwicklung des Systems hat Einschränkungen. Unter anderem ist es nicht vorgesehen, die Hardware-Konfiguration oder die Signalverarbeitungsbibliothek während der Laufzeit zu verändern, das würde den Rahmen dieser Arbeit sprengen.

Der Programmablauf des internen Programms soll zyklisch erfolgen, also an die Norm EN 61131 angelehnt sein. Ein eventbasiertes Vorgehen, wie in der EN 61499 beschrieben, ist nicht umgesetzt worden, da hierbei der Ablauf nur sehr schwer deterministisch zu gestalten wäre.

Linux ist in der verwendeten Standardkonfiguration kein Real Time Operating System. Obwohl verschiedene Scheduler betrachtet werden, wird bei der folgenden Implementierung lediglich der sched_other verwendet. Außerdem wird kein preempt_rt Patch installiert.

Mit der aktuellen Architektur kann der OPC UA Client Variable auf dem Server lesen und bearbeiten. So werden die Informationen von einem Knoten zu einem anderen Knoten transportiert. OPC UA bietet aber auch eine Art von Remote Procedure Call im Method Service Set an. Diese Möglichkeit ist nicht in die Systemarchitektur mit eingebunden worden.

5. Implementierung

Dieses Kapitel beschreibt die Implementierung des Systems auf zwei BeagleBone Blacks. Aufgrund der Vorkenntnisse in diesen Sprachen wird zur Programmierung C bzw. C++ verwendet.

5.1 Erweiterungsmöglichkeiten der Implementierung

Da für die vollständige Programmierung des beschriebenen Systems die Zeit nicht reicht, wird die Implementierung auf die wichtigsten Funktionalitäten beschränkt, sodass dennoch ein „proof of concept“ erstellt werden kann.

Im Endausbau der Applikation müssen nicht auf allen Knoten alle Load Balancing Tasks installiert sein. Stattdessen werden mit der Systembeschreibung (Sysdesc) Mechanismen implementiert, die es den Knoten erlauben, Tasks untereinander auszutauschen. Dafür übergibt der ausgelastete Knoten dem empfangenden Knoten sowohl den Task als auch dessen Parameter. Die Implementierung dieser Funktionalität würde den Rahmen dieser Arbeit sprengen. Stattdessen sind auf allen Knoten alle Tasks bereits installiert. Somit muss nur der gewünschte Task ausgewählt werden.

Obwohl die vorangegangenen Kapitel mehrere Möglichkeiten zur Bestimmung der Knotenpriorität enthalten, sind in der Implementierung die Werte für jeden Parameter fest vorgegeben. Dies ähnelt der beschriebenen statischen Methode, jedoch werden die Werte nicht automatisch bestimmt, sondern vor der Laufzeit manuell festgelegt. Langfristig könnte die Applikation um die beschriebenen komplexeren Methoden erweitert werden.

Da der zeitliche Aufwand der Integration des GIO212 zu groß wäre, wird die Arbeit von Klaus Jäger nicht eingebunden. Um dennoch Inputs und Outputs bedienen zu können, werden die internen GPIOs des BeagleBones verwendet.

5.2 Erstellte Bibliotheken

Einige erstellte Bibliotheken enthalten Funktionen und Klassen, die für die Funktionalität des Systems unerlässlich sind. Dazu zählen die Interaktion mit den BeagleBone internen GPIOs, die Interprozesskommunikation mittels Shared Memory und Semaphoren sowie das Erzeugen und Beenden von Child-Prozessen. Um die Wiederverwendbarkeit zu gewährleisten, werden diese Funktionalitäten zentral in Header und Source File zusammengefasst. Deren Aufbau und Funktionalität wird in den folgenden Kapiteln dargelegt.

5.2.1 gpiolib

Die Interaktion zwischen den internen GPIOs und den Nutzenden des BeagleBones erfolgt über entsprechende Dateien in /sys/class/gpio. Um den Zugriff zu erleichtern, wird die gpiolib erstellt. Diese besteht im Wesentlichen aus vier Funktionen zum Initialisieren, Lesen, Schreiben und Löschen eines GPIOs.

GPIO initialisieren: An die Funktion openGPIO wird eine Pin-Nummer und die Richtung (I/O) des Pins übergeben. Falls diese Pin-Nummer nicht existiert oder die Richtung nicht korrekt angegeben wird, retourniert die Funktion -1, wodurch ein Fehler signalisiert wird. Die Pin-Nummer muss vom Datenformat Integer in Charakter konvertiert werden, damit das entsprechende File geöffnet werden kann. Schließlich wird das File mit der entsprechenden Bezeichnung „out“ oder „in“ beschrieben und wieder geschlossen.

GPIO lesen: Wurde ein Pin zuvor mit „in“ initialisiert, fungiert der Pin als Input und kann gelesen werden. An die Funktion digitalRead wird eine Pin-Nummer im Integer Datenformat übergeben. Diese muss zuerst in einen Charakter konvertiert werden, womit dann die richtige Datei geöffnet werden kann. Daraus wird schließlich der eigentliche Wert gelesen und als _Bool retourniert.

GPIO schreiben: Wurde ein Pin zuvor mit „out“ initialisiert, kann der Pin wie ein Output beschrieben werden. Die Funktion digitalWrite übernimmt die Pin-Nummer als Integer und einen entsprechenden Output-Wert mit dem Datenformat _Bool. Die Pin-Nummer wird zuerst wieder in einen Charakter umgewandelt. Schließlich wird der Output-Wert in das value-File geschrieben und anschließend geschlossen.

GPIO löschen: Damit die Pins beim nächsten Aufruf wieder neu initialisiert werden können, muss die alte Konfiguration gelöscht werden. Dabei wird an die Funktion closeGPIO wieder eine Pin-Nummer übergeben. Nach der Konvertierung in einen Charakter kann die Pin-Nummer in das unexport-File geschrieben werden. Dadurch wird die Zuordnung gelöscht.

5.2.2 ipclib

Linux stellt bereits alle Funktionen zur Interprozesskommunikation zur Verfügung, welche jedoch kompliziert in der Anwendung sind. Die ipclib besteht zum Großteil aus Wrapper-Funktionen, die die Konfigurationen der bereitgestellten Funktionen vereinfachen und schließlich aufrufen. Folgende Funktionen werden in der ipclib bereitgestellt:

Semaphore initialisieren: Mit der Funktion semaphore_init wird eine Semaphore aus einem Key generiert. Zunächst wird überprüft, ob die Semaphore bereits existiert. Sofern das nicht der Fall ist, wird diese Semaphore erzeugt.

	Die Berechtigungen werden so eingestellt, dass jede Anwendung auf die Semaphore zugreifen darf. Schließlich wird die Semaphore mit 1 initialisiert. Der Rückgabewert dieser Funktion ist der Semaphore Identifier.
Semaphore verwenden:	Zur Verwendung der Semaphore wird die Funktion semaphore_operation aufgerufen. Um den Funktionsaufruf leserlicher zu gestalten, werden zwei defines LOCK und UNLOCK im Headerfile festgelegt.
Semaphore freigeben:	Sobald die Semaphore nicht mehr benötigt wird, sollte sie mit der Funktion semaphore_destroy freigegeben werden. Als Eingangsparameter benötigt die Funktion die semid. Nach dem Funktionsaufruf ist die Semaphore auch in anderen Prozessen nicht mehr verfügbar.
Shared Memory initialisieren:	Um ein neues Shared-Memory-Segment zu erstellen, wird die Funktion sharedmem_init aufgerufen. Dabei wird sowohl ein eindeutiger Key als auch die Größe in Bytes übergeben. Zunächst wird getestet, ob ein entsprechendes Segment bereits existiert. Wenn dies nicht der Fall ist, wird das Shared Memory mit den übergebenen Angaben erstellt. Der Zugriff ist für alle Benutzer und Benutzerinnen berechtigt. Der Rückgabewert der Funktion ist der Shared Memory Identifier (shmid).
Shared Memory anbinden:	Bevor das erstellte Shared-Memory-Segment verwendet werden kann, muss es im Hauptspeicher des Programms mit der Funktion sharedmem_attach angebunden werden. Dies erfolgt durch den retournierten Pointer, welcher auf die erste Stelle des Segments zeigt. Als Übergabewert zur Identifikation des Shared Memorys dient die bereits erstellte shmid.
Shared Memory ablösen:	Bevor das Programm beendet werden darf, muss das Shared-Memory vom Hauptspeicher mit der Funktion sharedmem_detach abgelöst werden. Dadurch wird sichergestellt, dass im anschließenden Programmverlauf keine Daten auf dem geteilten Speicher verändert werden können.
Shared Memory freigeben:	Wenn die Funktion sharedmem_destroy aufgerufen wird, gibt diese das Shared Memory frei. Dabei ist zu beachten, dass anschließend auch die anderen Prozesse und Services nicht mehr auf den Speicher zugreifen können.

Load Balancing Task aufrufen:	Mit der Funktion callTask werden die Load Balancing Tasks aufgerufen. Dafür wird zunächst mit fork das gesamte Programm kopiert und ein neuer Child-Process erstellt. In diesem Prozess wird mit execv der Load Balancing Task aufgerufen. Der Rückgabewert der Funktion im Parent-Process ist die PID des neu erstellten Prozesses.
PID überprüfen:	Die Funktion checkPID überprüft, ob eine PID verwendet wird oder ob der dazugehörige Prozess bereits beendet ist. Dafür wird die Linuxfunktion kill(0) aufgerufen. Ist die PID nicht in Verwendung, generiert das Betriebssystem die Errornummer 3.
Zombie löschen:	In der Funktion deleteZombie wird waitpid für eine übergebene PID aufgerufen. Damit wird der Rückgabewert des Childs abgefragt und der Prozess beendet. So werden Prozesse im Zombie Status auf dem Betriebssystem verhindert. Wenn nicht jeder beendete Prozess einzeln gelöscht werden soll, kann durch die Zeile signal(SIGCHLD, SIG_IGN) jedes Signal eines Child-Prozesses aktiv ignoriert werden. Somit werden die fertigen Programme dem Init-Process zugeordnet, welcher diese löscht. Ein Vorteil ist, dass das eigene Programm kleiner und schneller wird. Im Gegenzug dafür kann kein Einblick auf einen möglichen Errorcode des Child-Process genommen werden.

5.2.3 lblib

Diese C++ Bibliothek beinhaltet einige oft verwendete Funktionen des Loadbalancers. Der Loadbalancer behält den Überblick über alle aufgerufenen Tasks mit Hilfe eines nach der PID sortierten Vektors aus Objekten. Die dazugehörige Klasse ist ebenfalls im Header-File dieser Bibliothek hinterlegt.

Class LoadBalancingTask:		
	Attribute:	
		int taskNumber: Eine Nummer, die jeden Task identifizieren kann. Da 7 Load Balancing Tasks vorhanden sind, reicht die Nummer von 0 bis 6. -1 bedeutet „nicht initialisiert“.
		int internal: Eine Nummer, die darlegt, ob ein Task intern oder extern ausgeführt wird. -1 bedeutet „nicht initialisiert“.
		int PID: Die PID, die dem Task von Linux vergeben wurde. Hier steht -1 für die Ausführung von externen Tasks, -10 ist der Standardwert für nicht initialisierte Werte.
		timeval timeStamp: Bei der Konstruktion des Objektes wird der aktuelle Zeitstempel gespeichert.
		int linuxNicenessValue: Zeigt die aktuelle Priorität des Tasks unter Linux anhand des Niceness Wertes im Userspace an.
		int linuxPriorityClass: Tasks werden bestimmte Prioritätsklassen zugewiesen, innerhalb welcher die Linux Priorität erhöht werden kann. Es existieren 8 Klassen zu je 5 Prioritäten – wobei die höchst priorisierte Klasse nicht für Load Balancing Tasks freigegeben wird, sondern ausschließlich systemkritischen Programmen vorbehalten ist.
		int nodePriority: Die Knotenpriorität des Tasks wird ebenfalls in der Klasse abgelegt.
	Public Funktionen:	
		int getTaskNumber(void): Diese Funktion gibt die Tasknummer des Objektes in Form eines Integer zurück.
		int getInternal(void): Der Aufruf dieser Funktion retourniert die Variable internal des ausgewählten Objektes.
		int getPID(void): Diese Funktion liest die gespeicherte PID des Objektes aus.

		double getTimeStamp(void): Die Funktion berechnet die Prozessorzeit zum Zeitpunkt der Erstellung in Sekunden und gibt diese Zeit im double Format zurück.
		double lifeTime(void): Die Funktion berechnet die Differenz zwischen der Zeit der Erstellung und der aktuellen Zeit. Das Ergebnis ist das Alter des Objektes zum Zeitpunkt des Funktionsaufrufes.
		int increaseLinuxPriority(void): Innerhalb der angegebenen Klasse kann mit dieser Funktion die Linux-Priorität erhöht werden. Ist die Erhöhung erfolgreich, retourniert die Funktion einen Integer Wert von 1, andernfalls -1. Um diese Funktion verwenden zu können, muss das Programm mit Root-Rechten aufgerufen werden.

Tabelle 4: Load Balancing Class

Knotenpriorität bestimmen:	Die Parameter K1, K2 und K3 für jeden Task sind in dem File NP.csv abgespeichert. Die Funktion readNP liest das File an der gewünschten Stelle aus und retourniert die Knotenpriorität eines Tasks.
CPU-Last lesen:	Die Funktion readLoad bestimmt die CPU Auslastung des Knotens zur Zeit des Aufrufes. Dafür wird mit einem Systemcall der Output des Kommandozeilenprogrammes ps in einem File gespeichert. Das File wird durchsucht und alle CPU Auslastungen, die höher als die spezifizierte Priorität sind, werden zusammengezählt.
CPU-Last lesen – erweitert:	In readLoadIM wird die Funktion readLoad mehrmals aufgerufen und der Mittelwert gebildet. Dadurch sind Ausreißer in der CPU Auslastung leichter zu ignorieren.
Knoten erkennen:	Manche Load Balancing Tasks müssen für deren Funktion wissen, ob sie intern oder extern ausgeführt werden. Die Funktion amIInternally bestimmt die MAC Adresse des Ethernet-Ports des Knotens und vergleicht diese mit einer zuvor spezifizierten MAC Adresse. Wenn diese übereinstimmen, ist der Task intern und die Funktion retourniert 1.

5.3 Zyklisches Programm

Das interne zyklische Programm wird zur Implementierung auf dem BeagleBone mit einem einfachen C Programm erstellt. Die grundlegenden Charakteristiken einer PLC, wie der zyklische Ablauf und die feste Zykluszeit, werden dabei gewahrt.
Zuerst werden alle Inputs bestimmt und in das Hauptprogramm kopiert. Nach dem Ablauf des Hauptprogramms, welches von den Nutzenden verändert werden kann, werden alle Daten nach außen geschrieben. Hierbei werden auch die userspezifischen Load Balancing Tasks aufgerufen. Um die konstante Ablaufzeit festzulegen, geht das Programm für die verbleibende Zeit mit der Funktion sleep() in den Sleep-Zustand.

Zur Kommunikation mit anderen Prozessen besitzt die PLC zwei geteilte Speicher: Das „shared Memory“ und das „globalVariableInterface“. Der Load Balancer greift auf das „shared Memory“ zu. Hier werden die aufzurufenden Load Balancing Tasks vermerkt. Das „globalVariableInterface“ speichert die freigegebenen Variablen, Inputs und Outputs und stellt diese dem OPC UA Server zur Verfügung. Jede Operation auf einem der geteilten Speicher ist durch die dazugehörige Semaphore geschützt.

Insgesamt stehen drei Variablen zur Verfügung:

- eine boolsche Variable mit dem Namen globalVariableTest
- ein digitaler Input auf dem Pin 47 des BeagleBoards
- sowie ein digitaler Output auf dem Pin 45.

Diese Variablen können von einem anderen Knoten im Netzwerk verändert werden.

Zur Beendigung des Programms wird das in Linux übliche Signal SIGINT an das Programm gesendet. Dies geschieht in der Konsole mit der Tastenkombination ctrl + c. Dieses Signal wird vom Programm abgefangen. Bevor der Prozess endgültig beendet wird, werden die erstellten geteilten Speicher sowie die verwendeten Semaphoren gelöscht.

5.4 Loadbalancer

Der Loadbalancer erfüllt eine der Kernaufgaben des gesamten Systems. Zyklisch misst er die CPU Auslastung und entscheidet, ob und in welcher Reihenfolge die Tasks ausgelagert werden.

5.4.1 Test Aufgaben

Damit ein „proof of concept“ zur Auslagerung von Tasks durchgeführt werden kann, werden einige Test-Tasks formuliert und programmiert. Die Parameter K1, K2 und K3 dieser Tasks sind unterschiedlich, wodurch sich eine priorisierte Reihenfolge ergibt. Die Beispiele sind an das mögliche Einsatzgebiet der Gebäudeautomation angelehnt:

1. Datenbankzugriff: Der Fingerabdruck-Scanner an der Haustür erlaubt den Zutritt nur für Personen, die in einer Datenbank abgespeichert sind.
2. Webseitenaufruf: Die Heizung des Warmwasserspeichers erfolgt an sonnigen Tagen durch die Solaranlage, an regnerischen Tagen durch die Gasheizung. Der Wetterbericht für den nächsten Tag wird von einer Webseite geladen.
3. Bildverarbeitung: Ähnlich dem Fingerabdruck kann der Zugang durch Gesichtserkennung erlaubt oder verwehrt werden.
4. Mittelwertberechnung: Über einen Monat wird die Temperatur aufgenommen und abgespeichert. Am Ende des Monats wird die mittlere Temperatur berechnet. Dieser Wert kann zum Beispiel im folgenden Jahr in die Heizungsregelung einfließen.
5. Email Einbruchsmelder: An den Fenstern sind Sensoren angebracht. Diese lösen aus, wenn das Fenster beschädigt oder von außen geöffnet wird. Daraufhin wird eine Email oder SMS mit entsprechendem Inhalt an die Besitzer gesendet.
6. User Eingabe: Der oder die Nutzende kann über einen Webserver bestimmte Parameter des Gebäudes auslesen und bestimmen. Diese Parameter könnten Beleuchtung oder Raumtemperatur sein.
7. GPIO Task: Dieser Task sollte so stark auf dem Knoten integriert sein, dass eine Auslagerung nur in Extremfällen vorgenommen wird. Ein Beispiel könnte das Schalten von Außenbeleuchtung bei einer Bewegung sein.

Die programmierten Load Balancing Tasks imitieren nur das Verhalten der einzelnen Aufgaben. Mit langen Schleifen wird die CPU belastet, mit sleep werden Wartezeiten simuliert, wie zum Beispiel das Warten auf die Antwort eines Kommunikationspartners.

Die folgende Tabelle 4 beschreibt den Inhalt des Files NP.csv. Jeder Task-Nummer werden die Parameter K1, K2 und K3 gegenübergestellt. Die Berechnung der tatsächlichen Knotenpriorität erfolgt im Programm mit der Funktion readNP. Die Parameter werden vor der Ausführung der Tasks festgelegt und das File ist zur Laufzeit verfügbar.

Task Nummer	Name	K1	K2	K3	NP
0	databaseAccess	2	5	2	3
1	websiteCall	1	2	3	2
2	imageProcessing	5	9	1	5
3	averageCalc	6	7	5	6
4	sendEmail	4	5	2	3,666
5	userInput	3	4	6	4,333
6	blinky	8	9	4	7

Tabelle 5: Knotenpriorität aufgeschlüsselt in die Parameter K1, K2 und K3

5.4.2 Auslastungsbestimmung

Mit dem Befehl system("ps -eo pid,uid,priority,ni,pcpu,comm --sort -pcpu > loadFile") der Funktion readCPULoad(minPriority) wird das Kommandozeilenprogramm ps aufgerufen. Der Aufruf sortiert die Ausgabe nach der Spalte %CPU und speichert den Output in der lokalen Datei loadFile ab. Anhand des Headers wird die Position von Priorität und %CPU bestimmt und jede Zeile des Files nach den Werten durchsucht. Zur Gesamtauslastung des Systems zählen die Teilauslastungen der Programme, welche eine Mindestpriorität von minPriority haben.

Die Spalte %CPU von ps stellt die Zeit, in der ein Prozess die CPU verwendet, in Relation zu der Zeit, in der sich der Prozess im Status Running (R) befindet. Da auch das Betriebssystem Zeit auf der CPU verbringt, ist es sehr selten, dass die Summe aller Auslastungen genau 100% ergibt.

[Vgl. man7.org, 2018]

Diese Form der Auslastungsbestimmung ist die einzige Darstellungsweise, in der die Auslastung von Prozessen mit der Linux-Priorität verknüpft ist.

Da die einfache Auslastungsmessung zu großen Sprüngen der CPU-Last führt, wird die Messung mehrmals vorgenommen. Hierfür existieren zwei Versionen der Funktion readLoadIM, wobei der Mittelwert über 5 oder 10 Aufrufe gebildet wird. Für jeden Aufruf wird ein neues loadFile erzeugt und durchsucht. Diese zwei Messungen führen zu unterschiedlichen Eigenschaften des Load Balancers, welche im Kapitel 6 (Ergebnis) genauer erläutert werden.

5.4.3 Auslagerung

Bevor neue Tasks ausgeführt werden, wird anhand der CPU Auslastung und der Knotenpriorität entschieden, welche Tasks ausgelagert werden.

Der Prozess der Auslagerung beginnt ab einer Auslastung von 60%. Ab 61% werden Tasks mit einer Knotenpriorität von 1 ausgelagert. Mit jedem neuen ganzzahligen Prozentpunkt werden Tasks mit einer höheren Knotenpriorität ausgelagert, bis schließlich

bei 69% alle neuen Tasks auf eine andere CPU verschoben werden. Somit sollte eine theoretische Grenze von 70% CPU-Last nicht überschritten werden.

Die Auslagerung für jeden Task wird beibehalten, bis die Last unter die 50% Hysterese-Grenze fällt, dann werden wieder alle Tasks angenommen und der Auslagerungsprozess wird beendet. Dadurch wird Schwingungen im Auslagerungsverfahren vorgebeugt.

Die Auslagerungsgrenze kann mit dem define OUTSOURCING_BORDER auf die Hardware zugeschnitten werden. Sowohl die Hysterese-Grenze als auch die theoretische Obergrenze werden automatisch mit +/- 10% angepasst.

5.4.4 Task Administration

Für jeden aufgerufenen Task wird ein neues Objekt der Klasse LoadBalancingTask aus dem Header-File der lblib erstellt. Da die Objekte zur Laufzeit erstellt werden, ist eine Form der dynamischen Speicherallokation notwendig. Dafür wird die C++ Bibliothek <vector> verwendet. Diese Bibliothek ermöglicht eine einfache, dynamische Speicherverwaltung von Vektoren.

Es werden zwei Vektoren von Objekten erzeugt, wobei die internen und externen Tasks voneinander getrennt werden. Beim Aufruf eines neuen Tasks wird der Vektor um ein Element erweitert. Dadurch sind die Elemente automatisch nach der PID bzw. dem Alter sortiert.

Das Löschen der Objekte erfolgt nach einem einfachen Prinzip:
Es wird für jeden internen Task überprüft, ob die PID noch in Linux existiert. Wenn die PID nicht mehr vorhanden ist, wird das entsprechende Objekt entfernt. Beim Entfernen von externen Tasks wird nach der ältesten passenden Tasknummer gesucht und das gefundene Objekt wird gelöscht.

5.5 OPC UA

Da eine dezentralisierte, gleichberechtigte Architektur angewendet wird, verfügt jeder Knoten sowohl über einen Server als auch über einen Client. Der Server stellt Variablen zur Verfügung, welche von einem anderen Client im Netzwerk verändert werden können.

Standardgemäß startet Linux einen DHCP Server, der bei Anschluss eines Netzwerks dem Ethernet Port automatisch eine neue IP Adresse zuweist. Da der Client eine fest codierte IP Adresse des Servers benötigt, muss der DHCP Server ausgeschaltet werden. Die statisch festgelegten IP Adressen sind 10.0.0.41 für den Knoten, der die Überlast annimmt und 10.0.0.42 für den Knoten, der die Last generiert. Im weiteren Verlauf werden die Knoten ip41 und ip42 genannt.

5.5.1 Server

Server ip41:

Der Server auf dem Knoten ip41 empfängt die ausgelagerten Tasks und schreibt diese an das Load Balancing Interface des Load Balancers. Die Variablen sind vom Typ Integer und haben den durchnummerierten Namen „tasks Server X“ wobei X der Tasknummer von 0 bis 6 entspricht. Um die Lesbarkeit des Programmes zu steigern, wird als Node ID der gleiche String wie beim Namen verwendet.
Da die erstellte Sitzung zwischen Server und Client nach zu langer Inaktivität geschlossen wird, wird zusätzlich eine sogenannte heartBeat Variable dem Server hinzugefügt, welche sich jede Sekunde ändert.

Server ip42:

Der Server auf dem Knoten der Lastgenerierung ist größer als der des ip41, da zusätzliche Variablen für das „Global Variable Interface“ des zyklischen Programmes hinzugefügt werden. Die Variablen für das Load Balancing sind „answer Server X“ wobei X wiederum für die Tasknummer steht. Die Variablen werden an das Load Balancing Answer Interface des Load Balancer geschrieben. Auch dieser Server verfügt über die heartBeat Variable, die die vorzeitige Schließung der Sitzung verhindert.

5.5.2 Client

Client ip41:

Dieser Client verbindet sich zum Server auf 10.0.0.42. Das Programm übermittelt die Antworten des Load Balancer auf den Server des Knotens ip42, dafür wird das Load Balancing Answer Interface ausgelesen. Zusätzlich zum Load Balancing Answer Interface verfügt der Client über das Global Variable Interface, welches die globalen Variablen des zyklischen Programmes enthält und für die Funktion des ausgelagerten GPIO Tasks verantwortlich ist. Zusätzlich wird die Variable heartBeat des Servers kontinuierlich gelesen, damit die Sitzung nicht nach zu langer Inaktivität geschlossen wird.

Client ip42:

Dieser Client verbindet sich zum Server auf 10.0.0.41. Das Programm schreibt die Variablen und somit die extern auszuführenden Tasks des Load Balancing Interface an den Server des zweiten Knotens. Auch hier wird die Variable heartBeat des Servers zyklisch gelesen, um den Sitzungsabbruch zu verhindern.

6. Ergebnis

Um den Ablauf des Systems und die allgemeine Funktionstauglichkeit darzulegen, wird die prozentuale CPU-Last auf den beiden Knoten in einer .csv Datei aufgezeichnet und geplottet.

Damit die Lastgrenze erreicht wird und es zum Auslagerungsprozess kommt, steigert das zyklische Programm die Last. Dafür werden die Zeiträume zwischen den Aufrufen von neuen Load Balancing Tasks verkürzt, bis schließlich in jedem Zyklus, also alle 5 Sekunden, neue Instanzen von allen Tasks aufgerufen werden. Dies schließt den GPIO Task aus, da mehrere Operationen auf den gleichen IO Pins nicht zielführend sind. Das „forken“ neuer Tasks ist in den folgenden Abbildungen in der grünen Kurve dargestellt.

Ein Zyklus des Load Balancer besteht aus CPU Last messen, die generierten Tasks aufrufen und schließlich die Objekte der Load Balancing Tasks verwalten. Die Einstellungen des Load Balancer sind eine Zykluszeit von 5 Sekunden sowie die Auslastungsmessung über einen Mittelwert von 10 Iterationen. Die Zykluszeit des zyklischen Programmes liegt ebenfalls bei 5 Sekunden und die maximale Lastgenerierung ist auf 50 eingestellt. Nach 500 Sekunden wird die Lastgenerierung ausgeschaltet und gewartet, bis beide Knoten alle Prozesse abgearbeitet haben. Mit dieser Konfiguration wird der CPU-Last Verlauf in Abbildung 26 generiert.

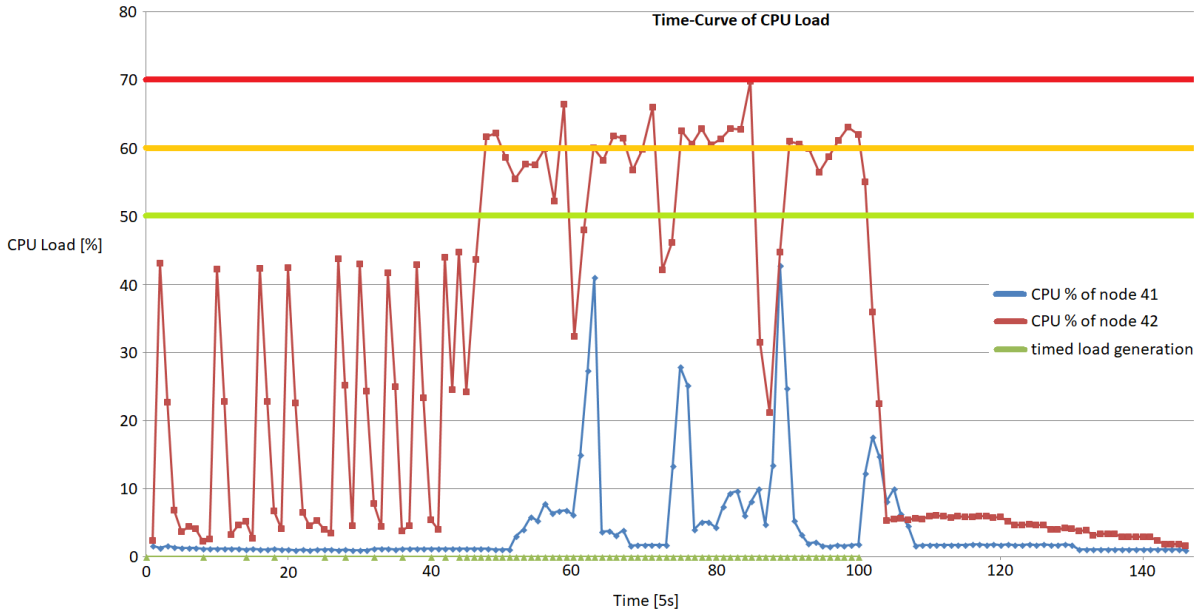


Abbildung 26: CPU-Last Verlauf a)

Zwischen Punkt 46 und 100 kann der Load Balancer auf ip42 die vorgebebenen 5 Sekunden nicht einhalten. Dadurch werden weniger Punkte geloggt und deren zeitlicher Abstand ist größer als der der Vergleichsmessung auf dem Knoten ip41. Um die zeitliche Kohärenz und die Vergleichbarkeit wiederherzustellen, sind die Abstände zwischen 46 und 100 um den Faktor 1,375 gegenüber der Vergleichskurve skaliert.

Bei der zweiten Messung werden die Einstellungen des Load Balancer geändert. Es wird die Auslastungsmessung mit einem Mittelwert über 5 Iterationen verwendet. Alle anderen Einstellungen bleiben erhalten: Die Zykluszeit des Load Balancer und des zyklischen Programmes werden bei 5 Sekunden belassen. Die maximale Lastgenerierung liegt wieder bei 50 und wird ebenfalls nach 500 Sekunden ausgeschaltet. Mit dieser Konfiguration wird der CPU-Last Verlauf in Abbildung 27 generiert.

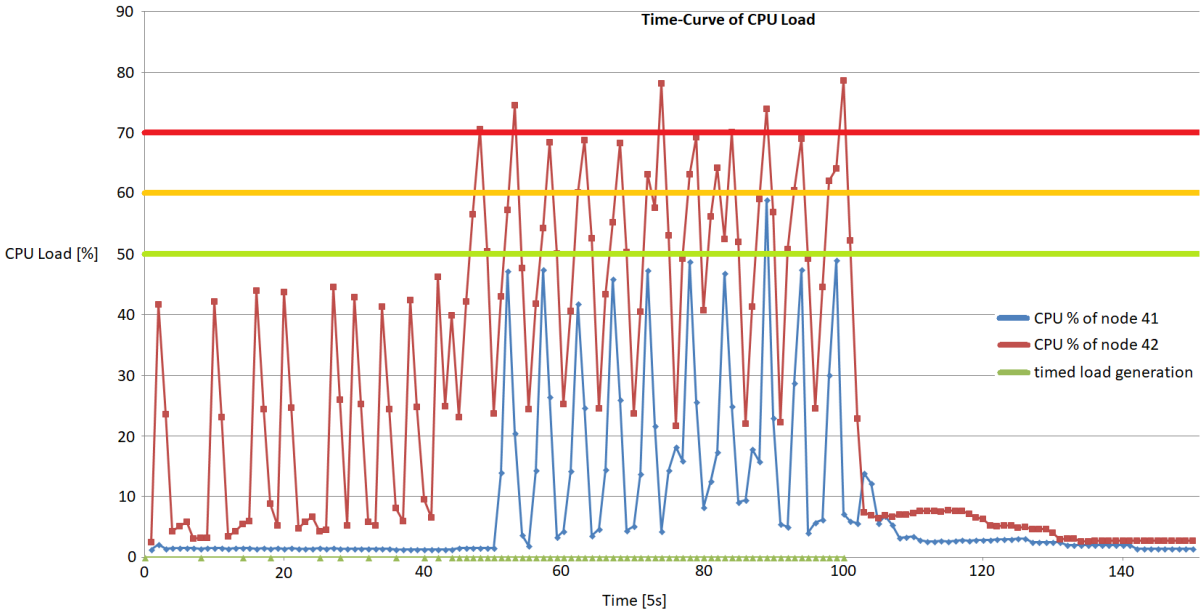


Abbildung 27: CPU-Last Verlauf b)

Weil die kürzere Auslastungsmessung verwendet wird, welche nur 5 Files durchsucht, wird der gesamte Zyklus des Load Balancers kürzer. Bei der zweiten Messung wird die 5-Sekunden-Zykluszeit des Load Balancer nicht verletzt. Deshalb ist keine Skalierung der Punkte notwendig.

In Anhang B und Anhang C sind beide Abbildungen zur besseren Lesbarkeit nochmals größer abgebildet.

7. Diskussion

In beiden Abbildungen ist erkennbar, wie die Abstände der Spitzen der CPU Auslastung bis zum Punkt 50 immer kleiner werden. Dieser Einschwingvorgang endet um den Punkt 50, ab dort steigt der Mittelwert der CPU-Last des Knotens ip42 stark an. Zudem beginnt kurz danach die CPU-Last des Knoten ip41 zu steigen, als dieser ausgelagerte Tasks annimmt. Die CPU-Lasten korrespondieren bis zum Punkt 100, dann wird die Lastgenerierung ausgeschaltet und es werden nur noch die bereits gestarteten Tasks abgearbeitet.

Die Konfiguration a), mit der CPU Mittelwertbildung über 10 Files, hat eine größere Zeitkonstante. Dadurch ist das System weniger dynamisch und die Spitzen in der Lastmessung sind geringer. Somit ist die Last leichter zu kontrollieren, erkennbar daran, dass die theoretische Obergrenze von 70% nicht überschritten wird.

Allerdings wird dadurch der Algorithmus und somit die Laufzeit des Load Balancer länger, wodurch die Zykluszeit nicht eingehalten werden kann. An dieser Stelle besteht noch Verbesserungspotential.

Die Konfiguration b) mit der CPU Mittelwertbildung über 5 Iterationen hat eine geringere Zeitkonstante. Dadurch ist das System dynamischer, was an den hohen Spitzen der Messung sichtbar ist. Die Last ist aufgrund der Sprünge schwieriger zu kontrollieren. Aus diesem Grund wird die Grenze von 70% an einigen Stellen verletzt.

Im Vergleich zur Konfiguration a) ist der Algorithmus schneller. Der Funktionsaufruf der Auslastungsmessung, welcher sich im Zyklus des Load Balancer befindet, durchsucht und interpretiert das „loadFile“ nur 5-mal. Da dieser Vorgang sehr aufwändig ist, wird somit Zeit eingespart, wodurch die 5 Sekunden der Zykluszeit eingehalten werden können. Allerdings wird dadurch die Genauigkeit und Kontrollierbarkeit des Systems verschlechtert.

Im Allgemeinen deutet das darauf hin, dass die Laufzeit des Algorithmus des Load Balancer zu lang ist und verbessert werden sollte. Dabei darf jedoch die Mittelwertbildung zur Auslastungsmessung nicht verkürzt werden. Stattdessen wäre eine laufzeittechnische Optimierung der Vektoroperationen sinnvoll. Der Suchalgorithmus zum Löschen der Elemente in der internen und externen Liste durchsucht jedes Element einzeln. Da es sich um eine sortierte Liste handelt, könnten binäre Suchverfahren angewendet werden.

Im Normalbetrieb, wenn keine Aufzeichnungen mit fester Zykluszeit notwendig sind, ist die Konfiguration a) der Auslastungsmessung bestens geeignet. Wenn im Rahmen von Echtzeitanforderungen die Zykluszeit des Load Balancer beachtet werden muss, wird eine Alternative benötigt. Eine Möglichkeit wäre das Aufspalten des Load Balancer Algorithmus und der Auslastungsmessung in zwei verschiedene Threads. Diese laufen parallel, wobei die Auslastung über eine gemeinsame Variable geteilt wird.

Dennoch könnten neue oder andere Wege der Auslastungsbestimmung erforscht werden. Da die Messung der CPU-Last ebenfalls auf der CPU ausgeführt wird, wird dadurch die Last selbst beeinflusst. Besser und präziser wäre es, wenn die Messung das Ergebnis so wenig wie möglich beeinflussen würde. Dabei gilt grundsätzlich, je hardwarenaher, desto geringer ist der Fehler - am besten wäre eine rein hardwaretechnische Lösung, zum Beispiel über das Beobachten bestimmter Pins an der CPU. Ein anderer Weg der Lastmessung über die Bestimmung der Temperatur der CPU ist verworfen worden, da hierbei keine Möglichkeit besteht, zwischen der Priorität von Prozessen differenzieren zu können.

Abseits der konfigurierbaren Auslastungsgrenze könnte je nach Hardware und Anwendung eine andere Definition von Auslastung notwendig sein. So hängt die Prozessorzeit eines Programmes auch von der Zugriffszeit auf Speicherelemente ab, siehe Kapitel 1.1. (Von Neumann Flaschenhals).

8. Zusammenfassung

In dieser Arbeit wird ein System für dezentralisierte Aufgabenverteilung unter mehreren Knoten im Netzwerk beschrieben, an Hand von festgelegten Funktionalitäten modelliert und Teile dieser Architektur ausprogrammiert. Ein Knoten besteht zumindest aus einer PLC, einem Load Balancer, einem Netzwerk Interface und einem IO Interface.

Der oder die Nutzende kann die Programme der PLC und des Load Balancing Tasks verändern. Die Load Balancing Tasks werden vom Programm der PLC aufgerufen und asynchron zum Zyklus gestartet. Sobald diese Programme fertig sind, retournieren sie das Ergebnis an die PLC. Das Besondere an diesen Tasks ist, dass sie auf anderen Knoten im Netzwerk ausgeführt werden können, wenn die CPU-Last des Knotens eine konfigurierbare Grenze überschreitet. Dadurch werden brachliegende CPUs im Netzwerk sinnvoll genützt.

Um die Integrierbarkeit in bereits existierende Netze zu vereinfachen, wird für die Netzwerkkommunikation der offene Standard OPC UA verwendet. Da die Knoten gleichberechtigt im Netzwerk sein sollen, benötigt jeder Knoten sowohl einen OPC UA Server als auch einen Client.

Besonderen Wert legt die Arbeit auf die Auslastungsmessung, da diese die fundamentale Eigenschaft des Load Balancers ist. Die Lösung ist eine nach Prozessen aufgeteilte Auslastungsmessung in Kombination mit der Priorität der Tasks. Ein weiterer wichtiger Aspekt der Arbeit ist die Reihenfolge, in der Load Balancing Tasks von einem Knoten auf einen anderen ausgelagert werden. Hierfür werden drei Eigenschaften eines Programms betrachtet: Netzwerkbandbreite, IPC Bandbreite sowie CPU-Last/Laufzeit. Anhand dieser Parameter wird eine priorisierte Reihenfolge, genannt node priority, erstellt, die beschreibt, wie leicht ein Task auszulagern ist.

Im praktischen Teil der Arbeit wird die Systemarchitektur zum Teil auf BeagleBone Blacks umgesetzt. Aufgrund der gesammelten Erfahrung aus der Vorarbeit von Klaus Jäger und der niedrigen Einstiegsschwelle werden diese Entwicklungsboards gewählt.

Um die Weiterentwicklung des Systems zu vereinfachen, werden so viele Funktionen wie möglich in Header und Source Files aufgeteilt. Dies beinhaltet die gpiolib für die internen GPIOs des BeagleBones, die ipclib für Interprozesskommunikation mit Semaphoren und geteiltem Speicher sowie die lblib für Load Balancing. Zur Implementierung der Netzwerkkommunikation wird der opensource Stack open62541 zu OPC UA verwendet. Dieser Stack wird mit einer einfachen Source und Header Datei vertrieben und kann ohne Lizenz installiert werden. Um das System zu testen, werden zuvor sieben einfache Test-Tasks erstellt, die das Verhalten von richtigen Tasks simulieren sollen. Diese Tasks werden schließlich vom Load Balancer intern oder extern aufgerufen.

Die CPU-Last wird über den Zeitverlauf aufgezeichnet, wodurch die allgemeine Funktionstauglichkeit nachgewiesen werden kann. Die Aufzeichnung wird bei verschiedenen Einstellungen des Load Balancer wiederholt und das Ergebnis kritisch analysiert. Bei einer längeren Auslastungsmessung ist das System weniger dynamisch und kann leichter kontrolliert werden. Dafür ist die Messung umfangreicher und stellt höhere Anforderungen an die Laufzeit. Zudem beeinflusst die Auslastungsmessung das Ergebnis. Um diese Unschärfe der Lastmessung zu beseitigen, muss das Messverfahren grundsätzlich überdacht werden.

9. Ausblick

Abseits der genannten Verbesserungsmöglichkeiten gibt es noch zusätzliche Punkte für eine mögliche Weiterentwicklung.

Dazu zählt das Auslagern von Tasks, ohne dass alle Aufgaben auf allen Knoten installiert sind. Dafür müsste die Aufgabe an sich zerlegt und über das Netzwerk transportiert werden. Eine Möglichkeit wäre das Versenden des Quellcodes, dann müsste allerdings das File auf dem anderen Knoten kompiliert werden. Die Generierung dieses Overheads gilt es abzuwägen.

Ein weiteres Feature könnte das „Slicing“ von Load Balancing Tasks sein. Dabei werden Tasks bei zu hoher Auslastung während der Laufzeit unterbrochen. Auf einem anderen Knoten wird der Task dort fortgesetzt, wo die erste CPU beendet hat. Somit werden die Load Balancing Tasks in kleinere Arbeitsschritte unterteilt und die CPU-Last kann noch genauer verteilt werden.

Die Knotenprioritätsbestimmung nach dem statischen, dynamischen und kombinierten Verfahren würde die Auslagerung der Tasks noch feiner gestalten. Diese Verfahren beruhen anstatt auf Abschätzungen, auf tatsächlich gemessenen Eigenschaften der Load Balancing Tasks.

Mit anderer Hardware sowie mit einem anderen Betriebssystem könnte man weiter die Echtzeiteigenschaften des Systems untersuchen. Besonders das zyklische Programm, dessen Echtzeitzyklus bis jetzt nur auf der Verwendung der sleep Funktion beruht, könnte zu einer echten PLC mit Laufzeitumgebung entwickelt werden. Aber auch Echtzeitanforderungen an die Load Balancing Tasks oder die Netzwerkkommunikation (OPC UA für TSN) könnten in einer nachfolgenden Arbeit betrachtet werden.

Literaturverzeichnis

Achilles, Albrecht (2006): Betriebssysteme. 1. Auflage. Berlin Heidelberg: Springer-Verlag.

Bauke, Heiko; Mertens, Stephan (2006): Cluster Computing. Praktische Einführung in das Hochleistungsrechnen auf Linux Cluster. 1. Auflage. Berlin Heidelberg: Springer Verlag.

Jäger, Klaus (2017): Zweistufige Kommunikation mit einem multifunktionalen ASIC über UART und SPI. Bachelor Thesis an der Fachhochschule Vorarlberg. Dornbirn.

linuxrealtime.org (2015): Basic Linux form a Real-Time Perspective. Online im Internet: http://linuxrealtime.org/index.php/Basic_Linux_from_a_Real-Time_Perspective (Zugriff am 13.02.2018)

man7.org (2018): Linux man-page for user command ps. Online im Internet: http://man7.org/linux/man-pages/man1/ps.1.html (Zugriff am 31.5.2018)

open62541.org (2018): open62541 OPC UA stack documentation. Online im Internet: https://open62541.org/doc/current/index.html (Zugriff am 20.02.2018)

Plötner, Johannes; Wendzel, Steffen (2012): Linux. Das umfassende Handbuch. 5. Auflage. Bonn: Rheinwerk Verlag. Online im Internet: http://openbook.rheinwerk-verlag.de/linux/index.html (Zugriff am 31. 5. 2018)

Schill, Alexander; Springer, Thomas (2012): Verteilte Systeme. Grundlagen und Basistechnologien. 2. Auflage. Berlin Heidelberg: Springer-Verlag.

Wolf, Jürgen (2009): C von A bis Z. Das umfassende Handbuch. 3. Auflage. Bonn: Rheinwerk Verlag. Online im Internet: http://openbook.rheinwerk-verlag.de/c_von_a_bis_z/index.htm#_top (Zugriff am 23.3.2018)

Anhang A: Elemente der Systemarchitektur

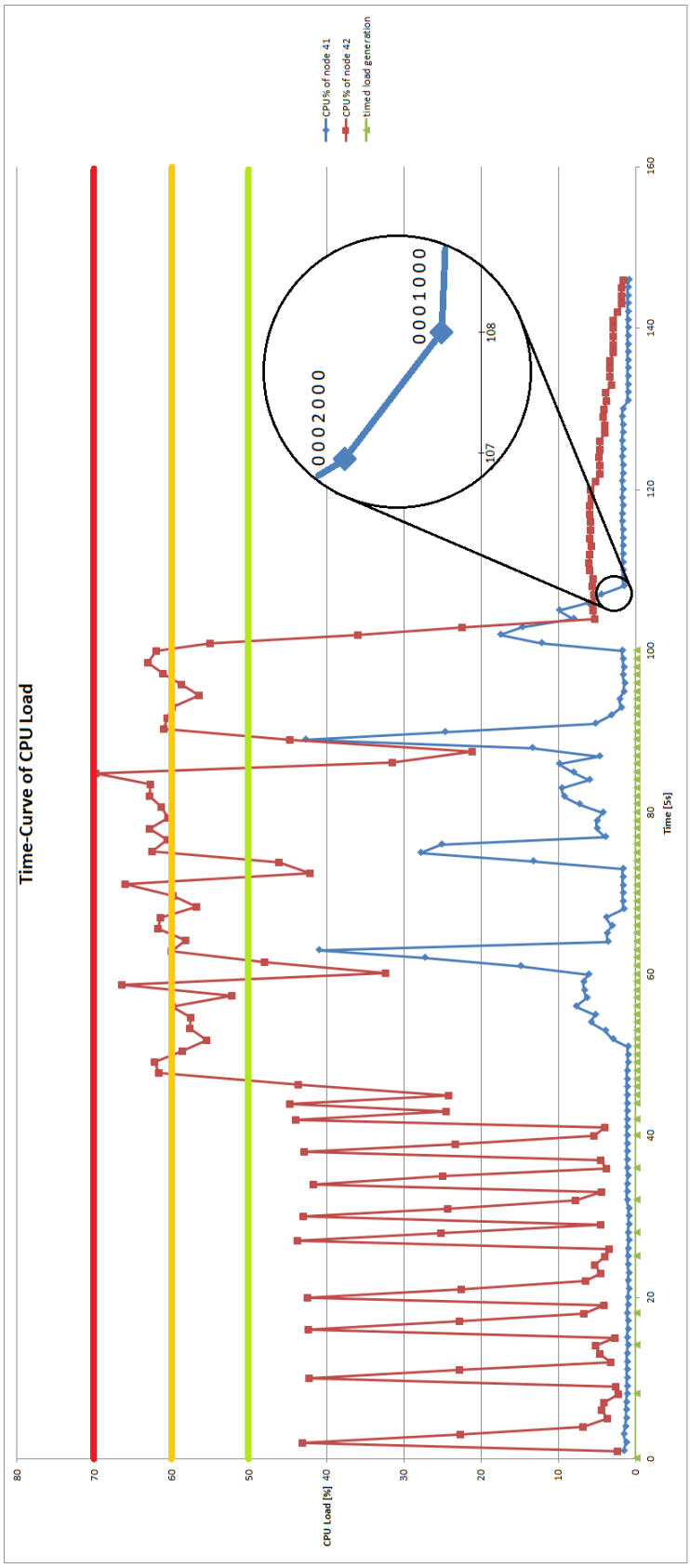
Konfigurator	
Daten	Funktionen
configMode	writeSPI()
HWFunctions	readCommIN()
IOchipSelect	generateNewGIO()
	mesageSysdesc()
	saveHWConfig()
	retrieveHWConfig()
IO Handler	
Daten	Funktionen
Konfigurator.configMode	writeSPI()
BufferInput	readSPI()
BufferOutput	writeOutputs()
Konfigurator.HWFunctions	readInputs()
Konfigurator.IOChipselect	interpreteBitString()
	generateBitString()
IOObject	
Daten	Funktionen
chipSelect	
ArrayAddress	
Framework	
Daten	Funktionen
Konfigurator.HWFunctions	returnExternalNormalResult()
actualTimeCycle	writeOutputToIOHandler()
uncompiledCode	readInputFromIOHandler()
programmingMode	calcCurrentRuntime()
Signalverarbeitung_Bibliothek.FunctionsArray	generateExternalNormalTask()
	generateLoadBalancingTask()
	updateProgram()
	readCommIN()
	messageSysdesc()
	writeSharedInputVariables()
	readSahredOutputVariables()
	setTimer()
	stopTimer()
	invokeCyclicProgram()
	writeSharedRegister()
	readSharedRegister()

shared Memory	
Daten	Funktionen
Input Variables	
Output Variables	
Variables	
Register	
Zyklisches Programm	
Daten	Funktionen
internalVariables	callSignalFunctions()
	writeSharedRegister()
	readSharedRegister()
Signalverarbeitung Bibliothek	
Daten	Funktionen
updateMode	SignalLib.o()
FunctionsArray	SignalLibExternal.o()
resultBuffer	readCommIN()
	writeCommOUT()
	selfUpdate()
	messageSysdesc()
	saveSignalLib()
	checkDatatype()
LoadBalancer	
Daten	Funktionen
SystemCPULoad	readCPULoad()
RealTimeCycle	calcRunTimeDifference()
Framwework.actualTimeCycle	internalBalancingQuery()
laodBalancingTaskQueue	searchExternalNode()
taskID-IPnum	generateLoadTaskObject()
RejectionPropability	forwardLoadBalancingTaskResult()
NodePriority	checkExternalLoadBalancingTask()
LoadBalancingTaskArray	increaseLoadBalancingTaskPriority()
updateMode	setLoadBalancingTaskPriorityClass()
	writeLoadBalancingInterface()
	readLoadBalancingInterface()
	generateSysdescError()
	shutdownError()
	calcSystemLoad()
	calcNodePriority()
	callLoadBalancingTask()
	calcExecDecision()
	readCommIN()
	messageSysdesc()
	updateLoadBalancingTask()

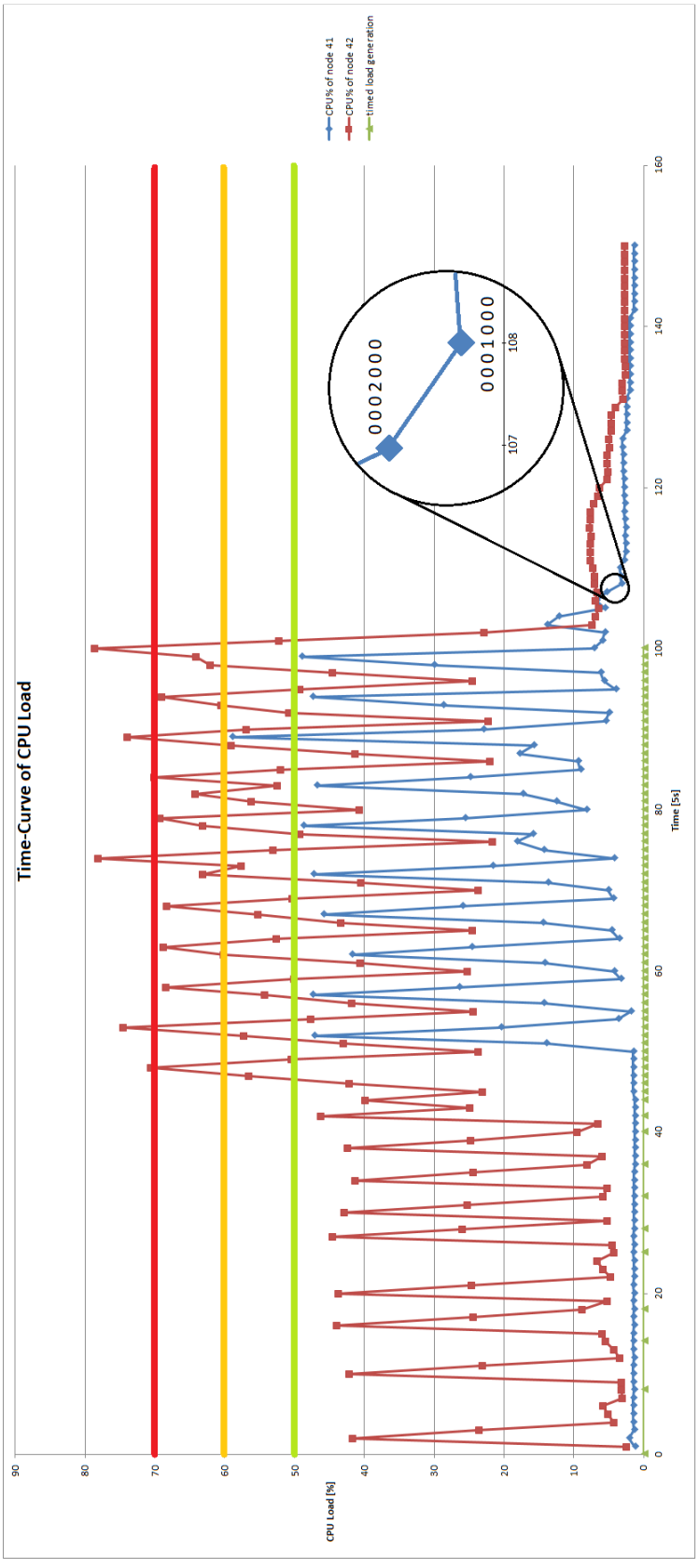
LoadBalancingTaskObject	
Daten	Funktionen
TimeToLive	getTimeToLive()
NodeIP	getNodeIP()
PointerToProgramList	writeNodeIP()
Internal/External	deleteObject()
priorityClass	
priority	
LoadBalancingInterface	
Daten	Funktionen
externalTaskInterface	
internalTaskInterface	
externalAnswerInterface	
internalAnswerInterface	
OPC UA Client	
Daten	Funktionen
OPC UA Datatypes	Subscription Service Set()
	Monitored Item Service Set()
	Attribute Service Set()
	View Service Set()
	Node Management Service Set()
	Session Service Set()
	Secure Channel Service Set()
	writeLoadBalancingTaskInterface()
	readLoadBalancingTaskInterface()
	writeSysdescInterface()
	readSysdescInterface()
OPC UA Server	
Daten	Funktionen
OPC UA Datatypes	Subscription Service Set()
	Monitored Item Service Set()
	Attribute Service Set()
	View Service Set()
	Node Management Service Set()
	Session Service Set()
	Secure Channel Service Set()
	writeLoadBalancingTaskInterface()
	readLoadBalancingTaskInterface()
	writeSysdescInterface()
	readSysdescInterface()

Agent	
Daten	Funktionen
	forwardNormalTask()
	checkInternalNormalTask()
	sendInternalSysdesc()
	checkExternalNormalTask()
	generateSysedescError()
	getGlaoallInput()
	getGlobalOutput()
	getGlobalVariables()
	writeGlobalOutputs()
	forwardNormalTaskResult()
globalVariableInterface	
Daten	Funktionen
global Inputs	
global Outputs	
globalVariables	
Systembeschreibung	
Daten	Funktionen
Konfigurator.HWFunctions	readHWConfig()
Signalverabeitung_Bibliothek.FunctionsArray	readSignalLib()
Framework.uncompiledCode	interpreteProgram()
internalSysdesc	generateExternalNode()
LoadBalancer.LoadBalancingTaskArray	searchExternalNode()
	deleteExternalNode()
	readSysdescInterface()
	writeSysdescInterface()
	readLoadBalancingTaskArray()
Externer Knoten	
Daten	Funktionen
IP_Address	
externalSysdesc	
SysdescInterface	
Daten	Funktionen
InternalSysdescInterface	
ExternalSysdescInterface	
publishInternalSysdesc	

Anhang B: CPU-Last Verlauf a)



Anhang C: CPU-Last Verlauf b)



Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich vorliegende Bachelorarbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Stellen sind als solche kenntlich gemacht. Die Arbeit wurde bisher weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Dornbirn, am 3. Juni 2018 Nicolai Schwartze