TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Realzeit-Computersysteme

Virtualisierte, fehlertolerante Systemplattform für automotive Systeme

Sebastian Martin Drössler

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor–Ingenieurs

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr.-Ing. E. Steinbach
Prüfer der Dissertation:	1. Univ.-Prof. Dr.-Ing. G. Färber, em.
	2. Univ.-Prof. Dr. sc. techn. A. Herkersdorf

Die Dissertation wurde am 22.06.2010 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 03.12.2010 angenommen.

Virtualisierte, fehlertolerante Systemplattform für automotive Systeme

Sebastian Martin Drössler

Danksagung

Diese Dissertation entstand im Rahmen meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Realzeit–Computersysteme der Technischen Universität München.

Einige der Ergebnisse entstanden im Forschungsprojekt FORBIAS, gefördert durch die BFS (Bayerische Forschungsstiftung), sowie im Kooperationsprojekt IT_Motive 2020[1)] in Zusammenarbeit mit der BMW Group und weiteren Lehrstühlen der TU München.

Meinen Kollegen am Lehrstuhl und im Projekt IT_Motive 2020 möchte ich für die zahlreichen fruchtbaren Diskussionen danken. Mein besonderer Dank gilt Prof. Färber, der mir diese Dissertation ermöglicht hat und mir immer mit Rat und Tat zur Seite stand.

Nicht zuletzt möchte ich mich bei meiner Familie und meinen Freunden für die moralische Unterstützung bedanken.

München, im Juni 2010.

[1)] Das Forschungsprojekt IT_Motive 2020 wurde im Rahmen der CAR@TUM Kooperation der BMW Group und der TU München durchgeführt.

Für Jonas.

Inhaltsverzeichnis

Abkürzungen und Symbole

BF(D)	Best Fit (Decreasing)
CFI	Canonical Format Indicator
COTS	Commercial Off The Shelf
FAS	Fahrerassistenzsysteme
FCFS	First Come First Serve (Abarbeitungsreihenfolge gemäß des Eintreffens)
FCS	Frame Check Sequence (Kontrollfeld mit Prüfsumme eines Ethernet Pakets)
FF(D)	First Fit (Decreasing)
LAN	Local Area Network
LSB	Least Significant Bit (Bit mit niedrigstem Stellenwert)
MAC	Media Access Control
MMU	Memory Management Unit (Speicherverwaltungseinheit)
MSB	Most Significant Bit (Höchstwertiges Bit)
PAD	Füllfeld
PTP	Precision Time Protocol (IEEE 1588)
RT/RZ	Echtzeit
RTC	Real-Time Clock (Echtzeituhr)
SFD	Start Frame Delimiter (Kennzeichnung des Beginns eines Ethernet Pakets)
TM	Timed Message
TTE	Time Triggered Ethernet
VLAN	Virtual Local Area Network
VMDq	Virtual Machine Device Queue
VOQ	Virtual Output Queue (virtuelle Ausgangswarteschlange)
WF(D)	Worst Fit (Decreasing)
c	Verarbeitungszeit (relativ zur Prozessorkapazität U)
d	relative Deadline
p	Periode
T	Task
u	relative Auslastung (Utilization) einer Task T_i
U	Prozessorkapazität
ε	maximaler Scheduling-Fehler
η	maximale Abweichung der lokalen Uhr zu einem beliebigen Zeitpunkt
ρ	Dichte (Density)
τ	Taskset

Zusammenfassung

Die meisten Innovationen im Automobil finden heute im Bereich der Elektronik statt. Die zunehmende Vielfalt, Heterogenität, Vernetzung und Komplexität der neuen Funktionen lassen sich jedoch kaum mehr mit den heutigen Systemstrukturen realisieren. In dieser Arbeit wird eine ereignisgesteuerte Systemplattform für zukünftige Automobile vorgestellt, die den neuen Herausforderungen gewachsen ist.

Basierend auf einer Analyse der Anforderungen an eine zukünftige Systemplattform werden die wesentlichen Aspekte herausgegriffen und in einem Architekturvorschlag umgesetzt. Die bisherige starre Zuordnung von Funktionen zu Steuergeräten wird dabei aufgehoben. Sensoren liefern lediglich geringfügig aufbereitete Daten, die Funktionen werden von mehreren zentralisierten, leistungsstarken Recheneinheiten bereitgestellt (Cluster-Ansatz). Die einzelnen Komponenten sind über ein homogenes Kommunikationsnetz miteinander verbunden.

Damit die verfügbare Rechenleistung optimal ausgenutzt wird, müssen sowohl sicherheitskritische Anwendungen mit harten Zeitbedingungen, z. B. aus dem Bereich der aktiven Fahrerassistenzsysteme, als auch Anwendungen ohne Zeitanforderungen auf den zentralen Recheneinheiten platziert werden. Dies erfordert eine strikte gegenseitige Abschottung sowohl der Anwendungen auf den Recheneinheiten als auch der Nachrichten im Kommunikationsnetz. Im vorliegenden System wird diese durch den Einsatz von Virtualisierungslösungen erreicht. Des Weiteren werden verschiedene Lösungen der Anbindung der Recheneinheiten an das Kommunikationsnetz bzgl. ihres zeitlichen Verhaltens verglichen. Für das Kommunikationsnetz wird ein Protokoll vorgestellt, das auf Standard-Ethernet basiert und mit priorisierten Paketklassen arbeitet.

Komplexe verteilte Systeme dieser Art bedürfen einer tiefergehenden Betrachtung des zeitlichen Verhaltens. In dieser Arbeit werden sog. „Timed Messages“ verwendet, um für einen konsistenten Gesamtsystemzustand zu sorgen. Im Speziellen werden diese für den Einsatz in der virtualisierten Umgebung angepasst. Des Weiteren werden Einfachfehler untersucht, insbesondere wird auf das Verhalten fehlertoleranter TMR-Systeme (engl. „Triple Modular Redundancy“) und auf die Wiederherstellung eines fehlerfreien Systemzustands eingegangen.

Vor allem die Anforderungen an Fehlertoleranz und Echtzeit erfordern spezielle Methoden der Verteilung der Anwendungen auf die Recheneinheiten (Allokation). Zum einen müssen die Task-Replikate eines TMR-Systems auf unterschiedlichen Recheneinheiten untergebracht werden, wobei auch die Pakete über disjunkte Pfade zu versenden sind. Zum anderen muss nach einem Ausfall eine schnelle und ressourcenschonende Rekonfiguration des Systems ermöglicht werden. Hierfür werden Heuristiken erarbeitet, die sowohl den fehlerfreien Fall als auch mögliche Fehlerfälle betrachten und entsprechende Konfigurationen offline erzeugen.

Abstract

Most innovations in today's cars happen to be in the embedded electronics. However, common component-based system architectures, i.e., electronic control units (ECUs) connected by different automotive busses, are unable to deal with the increasing complexity and heterogeneity of many modern applications. To overcome this problem, a new event-triggered system platform is presented in this thesis.

The proposed system architecture results from analyzing current and future requirements on embedded automotive electronics. In prevailing system architectures, function and ECU are seen as an inseparable unit. For future systems, we recommend detaching the function from the ECUs, i.e., sensors only provide raw or slightly preprocessed data, whereas the software implementing the functionality is moved towards a cluster of uniform high-capacity control units that are connected over switched Ethernet.

In order to highly utilize the control units, functions with safety and real-time requirements are allocated on the same control unit as functions with soft or even without real-time demands. To achieve strict isolation between these functions, we use virtualization of network and computing resources. Particularly, in this thesis, different possibilities of connecting the control units to the network are examined with respect to the timing characteristics in the virtual environment. Further, we propose a real-time protocol for switched Ethernet introducing fixed priorities and criticality classes.

A holistic technique is presented to analyze the system's real-time behavior. Timed Messages are adapted to the virtual environment and, hence, used to provide a consistent system state at any point in time. Additionally, the real-time behavior of replicated tasks is studied, e.g., for their use in a triple modular redundancy (TMR) system so as to provide fault-tolerance.

To perform an optimized allocation of functions onto control units, different algorithms considering real-time and fault-tolerance requirements are introduced and studied in detail. Clearly, replicated tasks of the TMR system must be assigned to different ECUs. In Addition, the communication network must meet redundancy demands, i.e., providing node-disjoint paths. In case of a partial failure, the system must be reconfigured to regain full redundancy. How to perform this reconfiguration, i.e., the relocation of functions, is taken into account during the initial allocation process which results in a highly efficient recovery.

1 Einleitung

Der steigender Vernetzungsgrad von Funktionen im Automobil stellt den Bereich der Elektrik bzw. Elektronik vor eine große Herausforderung. Die einst starre Zuordnung von Funktion zu Steuergerät ist bereits heute nicht mehr möglich. Vor allem bei den aktiven Fahrerassistenzsystemen (FAS) wird dies besonders deutlich: Verschiedenste Sensoren, die Eigenzustand und Umfeld des Fahrzeugs erfassen, liefern Eingangsdaten für unterschiedliche FAS-Funktionen. Zum Beispiel werden Querbeschleunigung, Lenkwinkel und Objekthypothesen von Radar, Lidar oder Video sowohl für einen Spurhalte-Assistenten als auch für Funktionen zur Unfallfolgenminderung benötigt. Ausgangsseitig müssen aktive FAS Zugriff auf Aktoren besitzen, die über das gesamte Fahrzeug verteilt sein können, wie z. B. Bremsen, Lenkwinkelsteller oder aktive Stoßdämpfer.

Heute werden zwei Lösungsmöglichkeiten umgesetzt. Entweder stellen Steuergeräte eine Schnittstelle bereit, worüber andere Steuergeräte über einen Bus auf die angeschlossenen Sensoren oder Aktoren zugreifen können. Die verbreitetere Variante ist jedoch, dass Steuergeräte ihre Sensordaten über einen oder mehrere Busse an alle anderen verschicken. Jedes Steuergerät kann dann die benötigten Informationen mitlesen – nicht benötigte Informationen werden ignoriert.

Oftmals besitzen die verteilten Steuergeräte keine Kenntnis voneinander oder sind an unterschiedlichen Bussen angeschlossen, so dass die Nachrichten über Gateways weitergereicht werden müssen. Ohne weitere Maßnahmen kann es schlimmstenfalls zu konkurrierenden Zugriffen und dadurch zu inkonsistenten Zuständen kommen. Beispielsweise kann eine Geschwindigkeitsregelanlage das Kommando zum Beschleunigen erteilen, während die FAS-Funktion „automatische Notbremse“ das Kommando zur Vollbremsung verschickt.

Daher ist ein Lösungsansatz notwendig, der die bestehende Problematik entschärft. Im Projekt IT_Motive 2020[1)] wurden die Kernthemen untersucht und ein zukunftssicheres Konzept erstellt. In dieser Arbeit werden Teile der Ergebnisse hieraus beschrieben. Ein wichtiger Teilaspekt umfasst die Virtualisierung der Ressourcen, wie sie aus der IT-Welt, z. B. bei Serverfarmen, bekannt sind. Seit kurzem hält die Virtualisierung auch verstärkt in den Bereich eingebetteter Systeme Einzug. Hiermit ergeben sich zwei große Vorteile: Einerseits können Applikationen weitgehend Plattform-unabhängig gestaltet werden, andererseits besteht die Möglichkeit, Applikationen dynamisch (zur Laufzeit) auf andere Steuergeräte zu verschieben.

Ein solches dynamisches System lässt sich effizienter nutzen. Während des Betriebs können nicht benötigte Ressourcen (Steuergeräte) abgeschaltet werden, indem die relevanten Funktio-

[1)] Forschungsprojekt im Rahmen der CAR@TUM Kooperation der Technischen Universität München und der BMW Group

nen auf einige wenige Steuergeräte konzentriert werden. Im Fehlerfall oder bei Überlastung einzelner Steuergeräte dienen die übrigen als Reserve. Dadurch lässt sich die Verfügbarkeit erhöhen. Viel wichtiger ist jedoch, dass auf diese Weise sicherheitskritische Funktionen mit geringen zusätzlichen Ressourcen umgesetzt werden können.

1.1 Stand der Technik

Diese Arbeit umspannt mehrere Gebiete. Aus diesen werden im folgenden jeweils nur relevante Teilaspekte dargestellt.

1.1.1 Systemplattform und Systemarchitektur

AUTOSAR [51] hebt erstmals die starre Zuordnung von Funktionen zu Steuergeräten auf. Es bietet eine Infrastruktur an, die es ermöglicht, Funktionen weitgehend unabhängig von der Hardware zu entwickeln. Es werden standardisierte Schnittstellen zur Verfügung gestellt, die einen konsistenten Zugriff auf die Fahrzeug-Zustandsdaten ermöglichen. Obwohl durch das AUTOSAR Betriebssystem die Möglichkeit zur Abschottung von Applikationen gegeneinander besteht, wird im praktischen Einsatz an der Domänen-orientierten Struktur festgehalten, d. h. Applikationen werden nur auf Steuergeräte innerhalb einer Fahrzeug-Domäne (Komfort, Fahrwerk, etc.) verteilt.

Die vorgeschlagenen Architekturen der Forschungsprojekte KogniMobil [46] und FORBIAS [47] zielen eher auf prototypische Umsetzungen ab, wobei sich einige Aspekte durch Anpassung auf Anforderungen hinsichtlich Energie- und Ressourcenbedarf auch für den Einsatz im Serienfahrzeug eignen. BASEMENT [50][49] und die auf Verlässlichkeit optimierte Umsetzung DACAPO [26][90] bieten ein Gesamtkonzept an, bei dem sich sicherheitskritische und QoS-Anwendungen die verfügbaren Ressourcen teilen – dies gilt sowohl für Rechenknoten als auch für das Kommunikationsnetz. SEIS [38] untersucht die Eignung des Internet-Protokolls (IP) als Standard Kommunikationsprotokoll im Fahrzeug und geht dabei insbesondere auf Sicherheitsaspekte ein.

Neben den genannten gibt es zahlreiche weitere universitäre und auch industrielle Forschungsprojekte, die sich mit diesem Thema auseinandersetzen. BASEMENT/DACAPO und MARS [56] stehen hier als Vetreter der wenigen Projekte, die eine gesamtheitliche Architekturbetrachtung durchführen. Die übrigen genannten Projekte stammen aus dem näheren Forschungsumfeld von IT_Motive 2020.

Die vorliegende Arbeit verfolgt eine Domänen-übergreifende HW/SW-Architektur im Automobil. Hinzu kommt der Einsatz von Virtualisierungstechniken zur Abschottung der sicherheitskritischen Funktionen. Zudem werden Mechanismen zur Rekonfiguration des Systems während des Betriebs bereitgestellt, so dass ein beliebiger Ausfall nicht zum Verlust einzelner Funktionen führt. Weitere Merkmale sind der direkte Anschluss von intelligenten Sensoren und Aktoren am

Kommunikationsnetz sowie die Möglichkeit, einfache Sensoren und Aktoren über ein Gateway an das Kommunikationsnetz anzuschließen.

1.1.2 Virtualisierungslösungen und Mikrokernel

Virtualisierungslösungen aus der IT-Welt sind auf hohe Verfügbarkeit und hohen Durchsatz ausgelegt. Zudem sollen die vorhandenen Ressourcen möglichst gerecht unter den Benutzern aufgeteilt werden. Beispielsweise sind openMosix [78] und OpenSSI [79] Clusterlösungen, die ein sog. „Single System Image“ bereitstellen und auf Rechenleistung, nicht aber auf Zuverlässigkeit optimiert sind. Als Beowulf Cluster (siehe z. B. [91]) werden Cluster bezeichnet, die ebenfalls eine hohe Rechenleistung zum Ziel haben und i. d. R. aus Standard PCs und Betriebssystemen (meist: Linux) zusammengestellt sind. Auf Grund der steigenden Leistung eingebetteter Systeme halten solche Cluster in den Bereich eingebetteter Systeme Einzug. [41][72][71] beschreiben den Einsatz eines Beowulf Clusters in einem realen Satelliten. XEN [8] sowie das Container-Konzept [95] von Sun Microsystems dienen der Servervirtualisierung. Alle diese Lösungen sind jedoch nicht echtzeitfähig im unteren Millisekunden-Bereich und somit für echtzeitkritische Anwendungen mit kurzen Reaktionszeiten nicht geeignet.

Für eingebettete Systeme stehen mittlerweile mehrere Virtualisierungslösungen zur Verfügung, die echtzeitfähig sind. Oft sind diese zertifizierbar, so dass sie bereits in der Luftfahrt eingesetzt werden. Einige kommerziell erhältliche Produkte sind (ohne Anspruch auf Vollständigkeit)

- PikeOS (SysGo)
- Trango (VmWare)
- COQOS (Opensynergy)
- L4 in verschiedenen Ausprägungen, z. B. L4ka (Universität Karlsruhe), DROPS (TU Dresden) oder OKL4 (Open Kernel Labs)
- VirtualLogix VLX (VirtualLogix)
- VXworks (Wind River)
- QNX Neutrino RTOS (QNX Software Systems)
- Rubus (Arcticus Systems)

In der vorliegenden Arbeit wird die freie Virtualisierungslösung XEN als Basis für die konzeptionellen Überlegungen sowie für die prototypische Umsetzung verwendet. Die Betrachtungen sind jedoch auf andere Virtualisierungslösungen übertragbar. Der von XEN mitgelieferte Scheduler ist nicht echtzeitfähig. Zudem erlaubt die Konfiguration der Treiberdomäne keine echtzeitfähige Kommunikation über das Netzwerk. Deshalb werden an der Domänenkonfiguration und am Scheduler selbst Änderungen durchgeführt, um das Echtzeitverhalten wesentlich zu verbessern.

1.1.3 Redundanzanforderungen für sicherheitskritische Funktionen

Redundanz kann zur Steigerung der Fehlertoleranz sowie der Zuverlässigkeit eingesetzt werden. Beide Aspekte spielen im Automobil eine große Rolle, auch wenn sie zunächst vom Fahrer nicht als spürbarer Mehrwert wahrgenommen werden.

Für x-by-wire Funktionen wird beispielsweise eine Ausfallrate von $< 10^{-9}\frac{1}{h}$ vorgegeben, die weit unterhalb der Ausfallrate von Einzelkomponenten ($10^{-5}\frac{1}{h}$) liegt [96]. Daher wird es unvermeidlich sein, ein System mit redundanten Komponenten für diese Funktionen einzusetzen. In der vorliegenden Arbeit wird hauptsächlich *statische Redundanz* betrachtet. Hierbei sind mehrere Systeme gleichzeitig aktiv und überwachen sich gegenseitig (z. B. 2-aus-3 Systeme). Einsatzgebiete sind sicherheitskritische Funktionen aller ASIL Stufen (A-D). *Fremdgenutzte Redundanz* stellt ihre Ressourcen ganz oder teilweise für andere Applikationen zur Verfügung und wird nur im Notfall aktiv. Letztere Variante kann auch für Funktionen eingesetzt werden, auf die ggf. für einige Sekunden verzichtet werden kann; i. d. R. sind dies Funktionen der Verletzungsklasse S0, deren Ausfall keine Verletzung von Personen nach sich zieht (z. B. Infotainment).

1.1.4 Sicherheitsintegrität für Automobile (ASIL)

In der Norm IEC 61508 werden 4 Sicherheits-Integritätsstufen (engl.: *Safety Integrity Level*, SIL) festgelegt. Diese Einstufung wird in der ASIL-Spezifikation (ISO 26262)[5] für das Automobil angepasst[2)]. Abhängig vom Ausmaß des Schadens, der bei einem Ausfall einer Funktion eintreten kann, wird jede Funktion in eine Sicherheitsanforderungsstufe von A (niedrigste) bis D (höchste Sicherheitsanforderungsstufe) eingruppiert. QM (Qualitätsmanagement) bedeutet, dass für diese Funktionen Standardmaßnahmen zur Qualitätssicherung (Software Engineering) ausreichend sind.

Welche ASIL Stufe eine Funktion besitzt, lässt sich aus Tabelle 1.1 ablesen. In die Einstufung gehen die Schwere der Verletzung (**S**), das Gefahrenpotenzial (**E**) sowie die Kontrollierbarkeit (**C**) des Fehlers ein. Diese Parameter sind in Anhang A dargestellt.

Aufspaltung zur Kritikalitätsreduktion

Die Spezifikation ISO 26262 erlaubt es, Funktionen der ASIL Stufen B bis D in unabhängige Funktionen geringerer Kritikalität zu unterteilen. Um zu erkennen, dass es sich um aufgespaltene Funktionen handelt, wird hinter die neue Stufe die ursprüngliche Stufe vermerkt. Weitere Aufspaltungen sind möglich, dabei verbleibt die Ursprungsstufe auf dem initialen Wert.

Beispielsweise kann eine Funktion mit ASIL D Einstufung in zwei (ASIL B(D), ASIL B(D)) oder auch drei Funktionen (ASIL A(D), ASIL A(D), ASIL B(D)) aufgespalten werden. Da-

[2)] Die Veröffentlichung als internationaler Standard wird für 2011 erwartet.

		C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	E1	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	E1	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

Tabelle 1.1: Zuordnung zu ASIL Stufen A-D

bei stellen zusätzliche Randbedingungen u. a. sicher, dass die ursprüngliche Sicherheitsstufe gewährleistet ist. Beispielsweise müssen zwei aufgeteilte Funktionen auf vollkommen unabhängigen Plattformen untergebracht sein.

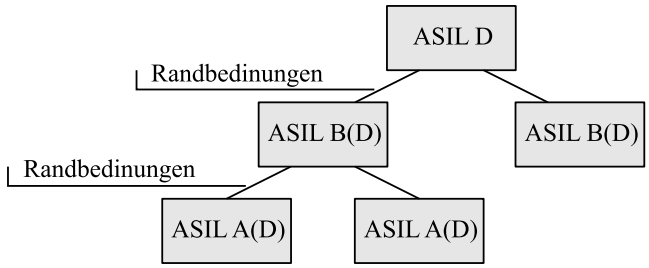


Bild 1.1: Aufspaltung in niedrigere ASIL Stufen gem. [5, Part 3].

1.2 Taskmodell und Zuteilungsverfahren

In der vorliegenden Arbeit wird von Taskreplikation und in den meisten Fällen von 2-aus-3 Systemen ausgegangen. Statt der Taskreplikation können auch verschiedene Implementierungen der gleichen Funktion verwendet werden, um die Ausfallsicherheit weiter zu erhöhen. Auf diversitäre Software wird jedoch nicht weiter eingegangen.

1.2.1 Taskmodell

Jede Task T_i wird einer Taskklasse entsprechend ihrem Redundanzgrad wie folgt zugeordnet. Die Replikate werden Cotasks genannt und mit $T_{i(tc)}$ bezeichnet. Mit tc werden die Cotasks einer Task mit höherer Taskklasse unterschieden.

Klasse 1: Keine Redundanz. Es existiert nur eine Instanz der Task T_i.

Klasse 2: Einfache Redundanz. Es existieren genau 2 aktive Instanzen der Task T_i. Ihre Cotasks werden mit $T_{i(1)}$ und $T_{i(2)}$ bezeichnet.

Klasse 3: 2-aus-3 System. Es existieren genau 3 aktive Instanzen der Task T_i. Ihre Cotasks werden mit $T_{i(1)}$, $T_{i(2)}$ und $T_{i(3)}$ bezeichnet.

Die Menge aller Cotasks wird als Tasksets τ bezeichnet. Eine Task T_i bzw. eine ihrer Cotasks $T_i(tc)$ wird durch die Periode p_i, die relative Deadline d_i und ihre Rechenzeit c_i bestimmt. Die Auslastung (Utilization) von T_i berechnet sich zu $u_i = \frac{c_i}{p_i}$. Die Dichte (Density) wird mit $D_i = \frac{c_i}{\min(p_i,d_i)}$ angegeben. Im vorliegenden System gilt immer $d_i \leq p_i$, deshalb wird im Folgenden immer $D_i = \frac{c_i}{d_i}$ gesetzt.

Eine Wirkkette W_i beschreibt die logische Kommunikationsbeziehung einer Task T_i, ausgehend von mindestens einer Quelle (Sensoren oder Recheneinheiten) über eine Verarbeitung (Recheneinheit) zu mindestens einer Senke (Aktoren oder Recheneinheiten). $P(W)$ beschreibt einen tatsächlichen Pfad (Menge aller durchlaufenen Knoten und Kanten), der für eine Wirkkette im Kommunikationsnetz verwendet wird.

1.2.2 Taskallokation

Die Taskallokation beschreibt die Zuteilung der Tasks eines Tasksets τ auf verfügbare Prozessoren bzw. hier gleichzusetzen mit Recheneinheiten. Das Ziel der Allokation ist meist, eine optimale Prozessorauslastung zu erreichen oder möglichst wenige Prozessoren für ein Taskset verwenden zu müssen.

Statische (offline) Allokation: Statische Allokationsverfahren legen zur Entwurfszeit fest, welche Ressourcen einer Task zugeordnet werden.

Dynamische (online) Allokation: Dynamische Allokationsverfahren sind in der Lage, die Tasks den Prozessoren zur Laufzeit zuzuteilen.

In der vorliegenden Arbeit wird nur die statische (offline) Allokation betrachtet.

Das Zuteilungsproblem ist NP hart, d. h. in polynomialer Zeit kann keine exakte Lösung des Problems gefunden werden. Es stehen zwei Lösungsmöglichkeiten zur Verfügung: Optimale und Heuristische Verfahren. In dieser Arbeit werden ausschließlich heuristische Verfahren für die Allokation verwendet.

Optimale Zuteilungsstrategien

Optimale Verfahren zeichnen sich dadurch aus, dass sie immer das globale Optimum finden. Da das Zuteilungsproblem NP-hart ist, stoßen diese Verfahren bei komplexeren Systemen schnell an ihre Grenzen, d. h. benötigen lange Rechenzeiten (Tage bis hin zu Jahren), um die Lösung zu finden. Solange sich das Modell linear beschreiben lässt, kann z. B. *integer linear programming* (ILP) zur Lösung herangezogen werden.

Heuristische Zuteilungsstrategien

Bei den Heuristiken wird mit Modellen gearbeitet, die das Problem abstrakter und i. d. R. ungenauer beschreiben und so eine Reduzierung der Komplexität erreichen. Das Ergebnis kann, muss aber nicht optimal sein.

Der meistverwendete Typ von Heuristiken ist *Bin Packing*. Bin packing beschreibt zunächst nur die Frage nach der bestmöglichen Ausnutzung vorhandener Behälter mit Gegenständen (z. B. Rucksackproblem). Angewandt auf das Zuteilungsproblem sind die Behälter die Prozessoren, auf die Tasks möglichst optimal verteilt werden müssen. Die Kostenfunktion beschränkt sich hier auf die Optimierung der Prozessornutzung. Optimal bedeutet hier, eine möglichst geringe Anzahl an Prozessoren für das Taskset zu verwenden. Man unterscheidet prinzipiell zwischen Online- und Offline-Verfahren zur Lösung des Zuteilungsproblems. Bei den Online-Verfahren werden die Tasks in beliebiger Reihenfolge auf die verfügbaren Prozessoren zugeteilt; zur Laufzeit eines Systems bedeutet dies, dass die Tasks in der Reihenfolge ihrer Erzeugung zugewiesen werden. Bei den Offline-Verfahren sind alle Tasks im Vorfeld bekannt und können daher vor der Zuteilung sortiert werden. [34] liefert eine gute Übersicht über Heuristiken für das Bin-Packing Problem. Die ausgewählten Verfahren sind deterministisch und besitzen eine polynomische Komplexität von $\mathcal{O}(n \log n)$:

First Fit (FF): FF ist ein Online-Verfahren. Die Tasks werden in zufälliger Reihenfolge auf einen Prozessor zugeteilt, solange dieser noch freie Kapazitäten besitzt. Falls nicht, wird ein weiterer Prozessor benötigt. FF sucht für jede weitere Task freie Ressourcen auf allen Prozessoren. Die Task wird auf dem ersten Prozessor, der diese aufnehmen kann, zugeteilt. Um die Suche zur Laufzeit zu beschleunigen, können Prozessoren, deren Auslastung einen Grenzwert überstiegen haben, von der Suche ausgeschlossen werden. Die maximal benötigte Anzahl an Prozessoren liegt höchstens um den Faktor 1,7 über der Anzahl bei einer optimalen Zuteilung.

First Fit Decreasing (FFD): FFD ist ein Offline-Verfahren. Zunächst werden die Tasks nach nicht aufsteigender Auslastung sortiert. Anschließend erfolgt die Verteilung nach FF. Offline-Verfahren wie FFD kommen grundsätzlich zu besseren Ergebnissen als Online-Verfahren. Die maximal benötigte Anzahl an Prozessoren liegt höchstens um den Faktor 1,25 über der Anzahl einer optimalen Zuteilung.

Worst Fit / Worst Fit Decreasing (WF/WFD): WF teilt eine Task dem Prozessor zu, auf den sie „am wenigsten passt“, d. h. auf dem noch am meisten Ressourcen zur Verfügung

stehen. Durch WF entsteht eine gleichmäßige Auslastung der Prozessoren. WF kann nur bei vorgegebener Anzahl von Prozessoren sinnvoll eingesetzt werden. Die Zuteilung kann wieder mit oder ohne vorsortiertem Taskset erfolgen.

Best Fit / Best Fit Decreasing (BF/BFD): BF teilt eine Task dem Prozessor zu, der nach der Zuteilung die geringsten freien Ressourcen besitzt. BFD liefert von den genannten die besten Zuteilungsergebnisse, d. h. die geringste Anzahl von Prozessoren.

Daneben gibt es viele weitere Heuristiken, die sich oftmals die Natur zum Vorbild nehmen. Im Folgenden werden nur einige der gebräuchlichsten Vertreter aufgezählt, da diese im weiteren Verlauf nicht weiter betrachtet werden.

„Hill Climbing“ geht von einem Ausgangszustand immer in die Richtung des nächsten, besseren Zustands – bewertet über eine Fitness-Funktion – bis keine Verbesserung mehr möglich ist [89][94]. „Simulated Annealing“ versucht das physikalische Verhalten von Molekülen bei der Abkühlung des Materials nachzubilden. Dabei lässt es im Gegensatz zu Hill Climbing auch leichte Verschlechterungen der Fitness-Funktion zu und vermeidet so lokale Optima [45].

Genetische Verfahren sind an die Theorie von Darwins bzw. Spencers angelehnt [15, 58]: „survival of the fittest“ (Überleben des am besten Angepassten). Erstmals wurde das Prinzip der genetischen Verfahren von Holland [53] auf technische Systeme angewendet. Genetische Verfahren sind stochastisch, d. h. sie liefern bei gleichen Eingangswerten nicht zwangsläufig das gleiche Ergebnis. Die Qualität des Ergebnisses ist stark davon abhängig, ob sich eine geeignete (stetige, monoton steigende oder fallende) Fitnessfunktion finden lässt.

Wegen des nicht deterministischen Verhaltens sollten genetische Verfahren nicht für die Berechnung zur Laufzeit verwendet werden, jedoch können sie für die Exploration des Ergebnisraums in der Designphase verwendet werden, um ggf. bessere Allokationen als die o. g. Algorithmen zu finden.

1.3 Aufbau der Arbeit

In Kapitel 2 wird die Systemarchitektur beschrieben. Ausgehend von den Herausforderungen in verschiedenen Bereichen des Produkt-Lebenszyklus eines Automobils werden die architektonischen Grundprinzipien abgeleitet, die schließlich in einem Architekturvorschlag zusammengeführt werden. Es folgen die Beschreibungen der einzelnen Hardwarekomponenten Kommunikationsnetz, Recheneinheiten, Sensoren und Aktoren sowie die Beschreibung der Komponenten der Basis-Software. Die Ergebnisse dieses Kapitels entstanden im Rahmen des Forschungsprojektes IT_Motive 2020.

Kapitel 3 beschreibt das zeitliche Verhalten des Systems. Zunächst wird allgemein auf die Synchronisierung des Systems und den Einfluss der virtuellen Umgebung eingegangen. Nach einer kurzen Betrachtung des Kommunikationsnetzes wird das Zeitverhalten der Recheneinheiten näher beleuchtet. Insbesondere wird der Einfluss verschiedener Schedulingverfahren sowie die

Verwendung und Partitionierung von mehreren Prozessoren für die virtuellen Domänen untersucht.

In Kapitel 4 wird auf die Fehlerbehandlung eingegangen. Zunächst werden die Auswirkungen von redundanter Verarbeitung auf das Zeitverhalten dargestellt. Darauf folgt eine Beschreibung möglicher Fehler sowie deren Erkennung und Lokalisierung. Das Kapitel schließt mit einer zeitlichen Betrachtung verschiedener Behebungsmaßnahmen.

Kapitel 5 befasst sich mit Allokationsmethoden der Echtzeit-Tasks im vorgestellten System sowohl für den fehlerfreien Fall als auch für den Fehlerfall. Es werden Heuristiken vorgestellt, die Redundanzanforderungen berücksichtigen und versuchen, möglichst geringe Kommunikationskosten zu erzeugen. Die verschiedenen Verfahren werden in Kapitel 6 simuliert, gegenübergestellt und bewertet.

Kapitel 7 fasst die Arbeit zusammen und zeigt Möglichkeiten auf, weiterführende Untersuchungen auf Grundlage der Ergebnisse dieser Arbeit durchzuführen.

2 Systemarchitektur

In diesem Kapitel werden die Herausforderungen, denen man in zukünftigen Automobilen gegenübersteht, aus verschiedenen Sichten dargestellt. Daraus erwachsen architektonische Grundprinzipien, die der Konzeption des Kommunikationsnetzes bzw. der Recheneinheiten zu Grunde liegen.

2.1 Herausforderungen

Im Folgenden werden die Herausforderungen, die sich bei der Konzeption für zukunftssichere Automobile ergeben, dargestellt und daraus erwachsende Anforderungen an die Architektur abgeleitet.

Um die Herausforderungen zu untergliedern, können die einzelnen Abschnitte des Produkt-Lebenszyklus betrachtet werden. Am Anfang steht immer die Design- und Implementierungsphase. Sie beinhaltet auch die Verifikation von Einzelkomponenten. Darauf folgt die Integrations- und Konfigurationsphase. Darin werden die einzelnen Systemkomponenten (Hardware und Software) zu einem Gesamtsystem zusammengefasst, das als Ganzes verifiziert, konfiguriert und ggf. parametrisiert wird. Danach wird das Fahrzeug ausgeliefert und nimmt seinen Betrieb auf. Zum Kundendienst und für anfallende Reparaturen muss das Fahrzeug in die Werkstatt gefahren werden – der Dauerbetrieb ist dadurch unterbrochen. Nach erfolgter Wartung oder Reparatur kann es wieder in Betrieb genommen werden. Wünscht der Kunde neue Funktionen, d. h. Erweiterungen in Hardware und/oder Software, muss eine Integration mit anschließend erneuter Konfiguration durchgeführt werden. Bild 2.1 zeigt die Abschnitte des Produkt-Lebenszyklus, die hinsichtlich ihrer Anforderungen im Folgenden genauer untersucht werden. Es ist zu beachten, dass sich die Anforderungen in den Abschnitten auf verschiedenen Abstraktionsschichten befinden, die sich aus den verschiedenen Sichten ergeben. In der Design- und Implementierungsphase sowie in der Integrations- und Konfigurationsphase kommen die Anforderungen hauptsächlich von Systemdesignern und Entwicklern. In der Phase Dauerbetrieb stammen sie von Kunden und in den Phasen Reparatur und Wartung sowie Nachrüstung kommen sie von Kunden und Werkstattpersonal.

2.1.1 Design- und Implementierungsphase

Bereits während der frühen Entwicklungsphase herrscht der Wunsch vor, einen möglichst hohen Anteil der bestehenden Hardware und Software wiederverwenden und leicht erweitern zu

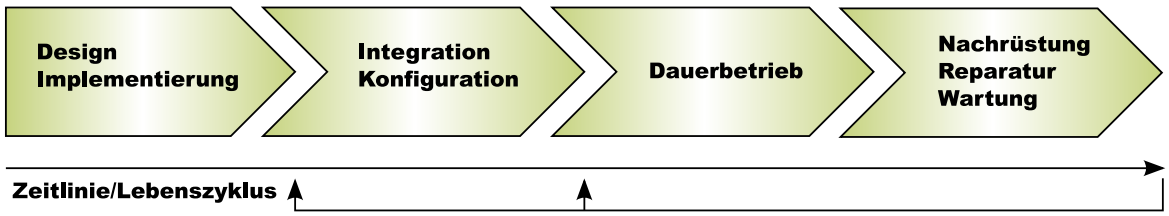


Bild 2.1: Relevante Abschnitte des Produkt-Lebenszyklus

können. Des Weiteren sollten Hardware und Software skalierbar, d. h. so gestaltet sein, dass sie in verschiedenen Baureihen, also vom Modell für den kleinen Geldbeutel bis zum Premiummodell (Variantenvielfalt), eingesetzt werden können. Speziell für die Applikationen ist es hilfreich, wenn alle Sensordaten an beliebiger Stelle im System gleichermaßen verfügbar sind und auf alle Aktoren zugegriffen werden kann. Hierbei muss immer auch sichergestellt sein, dass nur berechtigte Applikationen auf die entsprechenden Aktoren zugreifen können. Ebenso spielt die Komplexitätsbeherrschung eine große Rolle, da das Gesamtsystem verifizierbar sein muss; optimal wäre eine erste gute Abschätzung der Verifizierbarkeit des Systems schon in der Designphase.

2.1.2 Integrations- und Konfigurationsphase

In dieser Phase werden alle Einzelkomponenten zu einem Ganzen kombiniert und das Gesamtsystem konfiguriert und parametrisiert. Im Fahrzeug steht nur ein begrenzter Platz, der sog. Verbauraum, zur Verfügung, an dem überhaupt Hardware in Form von Recheneinheiten oder Kabeln verbaut werden kann. Hier ist der Wunsch gegeben, diesen Platz möglichst optimal zu nutzen, d. h. möglichst wenige Recheneinheiten zu verbauen und die Komplexität des Kabelstrangs möglichst gering zu halten. Werden allerdings nur wenige Recheneinheiten verbaut, muss es möglich sein, verschiedene Applikationen auf ein und derselben Hardware laufen lassen zu können. Da die Ausstattungen von verschiedenen Modellen oder Varianten stark variieren, kann das Mapping von Funktion auf Recheneinheit nicht starr vorgegeben sein. Dies bedeutet auch, dass für jede eingesetzte Konfiguration eine Verifikation des Gesamtsystems durchgeführt werden muss. Damit dies praktikabel bleibt, muss es Methoden geben, die die Komplexität der Verifikation gering halten.

Vor der Auslieferung an den Kunden wird das Gesamtsystem programmiert und parametrisiert. Bisher dauert dieser Vorgang mehrere Stunden. Wünschenswert ist eine drastische Reduzierung dieses Zeitaufwands.

2.1.3 Dauerbetrieb

Beim Dauerbetrieb stehen Kundennutzen und Kundenzufriedenheit im Vordergrund. Wünschenswert sind hier eine möglichst geringe Ausfallrate und, falls ein Teil des Systems ausfällt, eine

möglichst geringe Beeinträchtigung des täglichen Gebrauchs; d. h. keine Totalausfälle und kurze Standzeiten bzw. Zeiten, in denen sich das Fahrzeug in der Werkstatt befindet. Durch die steigende Anzahl an Fahrerassistenzsystemen im sicherheitsrelevanten Bereich, wie z. B. einem aktiven Spurhalteassistenten mit Abstandshaltung zum Vordermann, muss das E/E-System gerade in diesem Bereich eine klar definierte und verifizierbare Verfügbarkeit aufweisen. Hierzu müssen sich sicherheitskritische Funktionen in die ASIL-Stufen einsortieren lassen. Dem Kunden kann so eine hohe Sicherheit garantiert werden.

2.1.4 Reparatur und Wartung

Für die Werkstatt ist es ebenfalls von Vorteil, wenn die Aufenthaltszeiten des Fahrzeugs möglichst kurz sind. Ein guter Schritt in Richtung einheitlicher Diagnosemöglichkeit wurde bereits durch die Einführung des „Onboard-Diagnostic-System“ der zweiten Generation (OBD2 [76]) für alle Pkws ab Euro-3 Norm getan. Fahrzeugseitig ist darauf zu achten, dass sich die Diagnostizierbarkeit von Fehlern möglichst einfach und durchgängig gestaltet. Für eine effiziente Onboard-Fehlerdiagnose müssen insbesondere zusammenhängende Fehler oder Folgefehler erkannt und markiert werden.

2.1.5 Nachrüstung und Update

Betrachtet man die unterschiedlichen Produkt-Lebenszyklen eines Fahrzeugs und eines Consumer-Produkts, so fällt auf, dass manche im Fahrzeug integrierte Hardware- und Softwarekomponenten schon bei Auslieferung nicht mehr auf dem neuesten Stand sind. Deshalb muss schon beim Design der Architektur auf leichte Erweiterbarkeit und Aktualisierbarkeit geachtet werden. Hierbei sollte zwischen der Integrierbarkeit herstellereigener bzw. zertifizierter Produkte und Produkten von Drittanbietern unterschieden werden, da diese unterschiedliche Anforderungen an die Abschottung und Systemsicherheit stellen. Auch hier gilt es darauf zu achten, dass das Fahrzeug eine kurze Standzeit hat und die Nachrüstung oder Aktualisierung nicht zu komplex wird.

2.2 Architektonische Grundprinzipien

Neben den angeführten Wünschen und Anforderungen gibt es noch viele weitere, die sich aber alle auf die im Folgenden beschriebenen Grundprinzipien, die einer modernen und zukunftssicheren Architektur zugrunde liegen sollten, abbilden lassen.

2.2.1 Zentralisierung

Die gewachsenen Strukturen in der Automobilelektronik haben zu einer sehr heterogenen Hardwarelandschaft geführt. Es gibt für jede Funktion ein spezialisiertes Steuergerät mit direkt ange-

bundener Sensorik bzw. Aktorik; ebenso gibt es verschiedene Bussysteme, die speziell für ihren Einsatzbereich ausgelegt sind. Künftige Funktionen werden aber Daten mit unterschiedlichen Anforderungen an Zeit und Umfang benötigen. Des Weiteren werden Sensorwerte nicht mehr nur in einer Funktion in einem Steuergerät benötigt, vielmehr ist das Zusammenspiel verschiedener Sensoren für eine Vielzahl von Funktionen von Bedeutung. Dies lässt sich nur schwer auf die aktuellen Strukturen abbilden.

Logische Konsequenz ist, die Verarbeitung (die Funktion) von der Sensorik bzw. Aktorik zu entkoppeln und zu zentralisieren. Da diese Rechner nun keine direkte Ein-/Ausgabe besitzen, werden sie im Folgenden als *Recheneinheiten* bezeichnet. Es gibt einige Randbedingungen, die eine Lösung mit nur einer zentralen Recheneinheit ausschließen: Offensichtlich ist es nicht förderlich, einen Totalausfall des Fahrzeugs, eventuell sogar den Verlust der Kontrolle darüber zu riskieren, wenn diese eine Recheneinheit ausfällt. Auch lässt sich das Problem der Abwärme einfacher handhaben, wenn es mehrere Recheneinheiten in örtlich getrennten Bauräumen gibt, die sich die Aufgaben teilen.

2.2.2 Homogenisierung

Wenn die Rechenleistung wie in einem Cluster zentralisiert ist, muss die Hardware – und dies gilt sowohl für die Recheneinheiten als auch für die Bussysteme – gleichermaßen Echtzeitbedingungen im Bereich von wenigen Millisekunden erfüllen und mit größeren Datenmengen umgehen können, ohne dass Echtzeitbedingungen verletzt werden. Da diese Anforderungen für alle zentralisierten Recheneinheiten sowie dem verbindenden Bussystem gilt, können sowohl Recheneinheiten als auch das Bussystem vereinheitlicht werden.

Damit lassen sich einige, aber nicht alle aktuellen und zukünftigen Funktionen auf homogene Recheneinheiten bringen. Insbesondere für Funktionen mit Zeitanforderungen unterhalb weniger Millisekunden sind die Zeiten für den Kontextwechsel nicht mehr zu vernachlässigen. Es muss im Einzelfall entschieden werden, ob die effektiv nutzbare Systemleistung noch ausreicht oder ob manche Funktionen auf spezialisierter Hardware implementiert werden müssen.

2.2.3 Dynamisierung – Erweiterbarkeit und Rekonfigurierbarkeit

Die homogene, zentralisierte Struktur ermöglicht es, auf einfache Weise den Funktionsumfang zu erweitern. Zudem bietet sich die Möglichkeit, die vorhandene Rechenleistung im Falle eines Ausfalls sinnvoll (im Sinne einer „Graceful Degradation“), oder um Energie zu sparen, neu zuzuweisen.

Ein weiterer wichtiger Aspekt für zukünftige Funktionen wird die Erweiterbarkeit durch Applikationen von Drittanbietern sein, die ebenfalls auf den Recheneinheiten abgearbeitet werden müssen.

2.2.4 Virtualisierung

Jede der zentralen Recheneinheiten kann jede beliebige Aufgabe übernehmen, seien es echtzeitkritische bzw. sicherheitskritische oder daten- bzw. rechenintensive Funktionen mit weichen oder keinen Echtzeit-Anforderungen, verifizierte Software vom Hersteller oder beliebige Funktionen von Drittanbietern. Um allen Anforderungen gerecht zu werden, und um die Komplexität des Systems zu reduzieren, muss eine sichere Abschottung der sicherheitskritischen von den nicht-sicherheitskritischen Funktionen erfolgen. Am besten gelingt dies durch eine Virtualisierung der Ressourcen. Dies gilt wieder sowohl für die Ressourcen der Recheneinheiten (Rechenleistung, Speicher, Busanbindung, etc.) als auch für die des Kommunikationsnetzes.

Virtualisierung wird dabei eingesetzt, um

- die virtuellen Container auf beliebige physikalische Recheneinheiten zu platzieren,
- einem Container definierte Ressourcen zuzuweisen und
- Fehler einzudämmen (engl.: *Faultcontainment*).

2.2.5 Architekturvorschlag

Zusammengefasst bilden die architektonischen Grundprinzipien die Grundlage für eine zukunftssichere Architektur. Lässt sich eine Lösung für alle angegebenen Herausforderungen finden, erhält man eine robuste Architektur, mit der sich folgende Vorteile ergeben:

- verbesserte Datennähe
- bessere Ausnutzung der verfügbaren Rechenleistung
- einfache Erweiterbarkeit
- effizientes Fehlermanagement
- zentrale Systemkenntnis – wichtig für Diagnose und Fehlerbehandlung
- Reduzierung der Anzahl der ECUs
- Vereinfachung des Kabelbaums

Allerdings birgt der Architekturvorschlag auch einige Herausforderungen, die in den folgenden Kapiteln dieser Arbeit adressiert werden:

- Abschottung
- Verifikation
- Datenkonsistenz
- Synchronität
- optimale Zuteilung
- Verhalten im Fehlerfall (Rekonfiguration)

Die weiteren Abschnitte und Kapitel befassen sich nur mit den aus Echtzeit-Sicht interessanten Punkten; auf die Aspekte Abwärme, Energieeffizienz (Rekonfiguration zur Optimierung des Energiebedarfs) und Verbauraum wird nicht eingegangen.

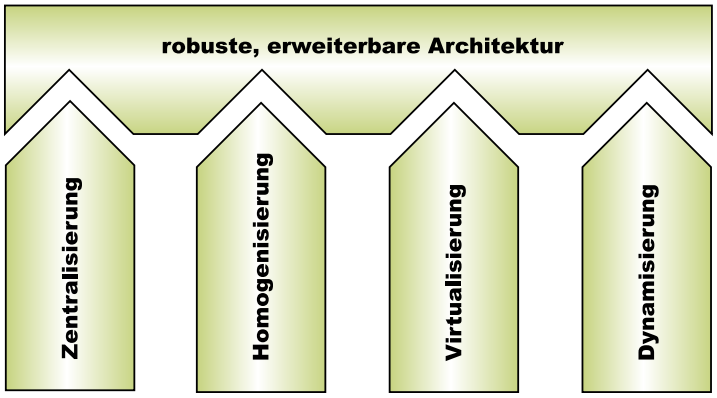


Bild 2.2: Architektonische Grundprinzipien

Im Folgenden wird auf die Umsetzung der vier allgemein beschriebenen architektonischen Grundprinzipien in eine reale Architektur eingegangen. Zunächst werden das Kommunikationsnetz und die Recheneinheiten von der erforderlichen Hardware bis einschließlich der Virtualisierungsschicht betrachtet.

Bild 2.3 zeigt die zentralisierte Struktur des Systems. Es besteht aus intelligenten Sensoren und Aktoren, die über ein homogenes Kommunikationsnetz mit den zentralen Recheneinheiten verbunden sind. Um Ausfallsicherheit für sicherheitskritische Funktionen bereitzustellen, müssen redundante Kommunikationswege vorhanden sein. Unter der Annahme, dass die Ausfallwahrscheinlichkeiten die Relation Sensor/Aktor $<$ Kommunikationsnetz/Switch $<$ Verarbeitungseinheit besitzen, müssen Sensoren für sicherheitskritische Funktionen, die nur einmal vorhanden sind, über redundante Pfade an das Kommunikationsnetz angeschlossen werden. Gleiches gilt für die Anbindung von sicherheitskritischen Aktoren. Wegen der Redundanz der Recheneinheiten müssen diese nicht redundant angebunden werden.

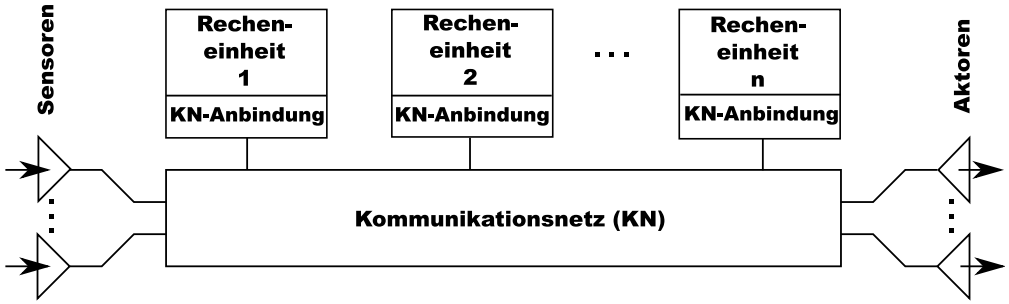


Bild 2.3: Grobstruktur des Systems

2.3 Kommunikationsnetz

Zentrale Aufgabe des Kommunikationsnetzes ist die deterministische Übertragung von sicherheitsrelevanten Daten. Zudem sollen auch Daten ohne zeitliche Schranke übertragen werden

können. Im Folgenden werden zunächst die Anforderungen an das Kommunikationsnetz aufgestellt. Nach einer Gegenüberstellung von Realisierungsalternativen wird schließlich ein eigenes Lösungskonzept vorgestellt.

2.3.1 Anforderungen

Deterministische Übertragung

Das Kommunikationsnetz muss sowohl Datenpakete mit wenigen Bytes als auch welche mit mehreren hundert Bytes handhaben können, um sowohl Sensordaten als auch Multimedia-Daten darüber verschicken zu können. Dabei sollte der Overhead durch das Protokoll möglichst gering sein. Die Übertragungsbandbreite richtet sich nach den Anforderungen aus dem Multimedia-Bereich (Video).

Zudem muss das Kommunikationsnetz in der Lage sein, bestimmte Nachrichten priorisiert zu behandeln und eine maximale Übertragungszeit für Echtzeit-Pakete zu garantieren.

Isolierung von Fehlern

Eine der wichtigsten Anforderungen für sicherheitskritische Systeme ist die Fehlerisolation. Zum einen muss sichergestellt sein, dass bei Fehlverhalten oder Zerstörung eines Teilsystems, z. B. durch einen Unfall, das restliche System intakt bleibt und dadurch nicht beeinträchtigt wird.

Zum anderen muss sichergestellt sein, dass das Fehlverhalten eines einzelnen Systems den fehlerfreien Zustand eines anderen Systems nicht korrumpiert.

Einheitliche Zeitbasis

Jeder einzelne Knoten des Gesamtsystems, also Recheneinheiten, Sensoren, Aktoren und das Kommunikationsnetz selbst muss eine einheitliche Systemzeit mit bekannter Präzision besitzen. Andernfalls kann es zu Dateninkonsistenzen kommen, die dazu führen, dass das Votieren einer Nachricht (bei redundanter Verarbeitung) fehlschlägt und fälschlicherweise ein Systemfehler angenommen wird.

2.3.2 Darstellung verschiedener Realisierungsalternativen

Die meisten der aktuell eingesetzten Fahrzeugbusse scheiden aufgrund ihrer zu geringen Bandbreite, z. B. CAN, oder der fehlenden Echtzeitfähigkeit, z. B. LIN, von vornherein aus.

MOST-Bus

Der MOST-Bus ist für Multimedia ausgelegt, hat aber die Nachteile, dass einerseits nur 64 Teilnehmer angeschlossen werden können und andererseits es ein fest vorgegebener Takt schwierig gestaltet, die unterschiedlichen Zyklen der Sensoren und Aktoren abzubilden.

FlexRay

FlexRay bietet im statischen Teil eine deterministische Übertragung durch Zeitmultiplexing, wie es z. B. auch bei TTP/C [99] oder TTCAN [43] verwendet wird. Auch bietet FlexRay die für die Anwendung für sicherheitskritische Funktionen geforderte Abschottung durch sog. Bus-Guardians. Allerdings ist das System ungeeignet für die Anforderungen im Multimedia-Bereich, da dort hohe Datenraten anfallen, diese aber nicht immer auftreten. Dies ist für den statischen Teil sehr ungünstig, da die reservierten Zeitschlitze dann ungenutzt bleiben. Auch ist FlexRay mit einer Bandbreite von derzeit maximal 10 Mbit/s für künftige videobasierte Fahrerassistenzsysteme und Multimedia-Anwendungen zu gering dimensioniert.

Real-Time-/Industrial-Ethernet

In den letzten Jahren hat sich Ethernet auch in der Automatisierungstechnik als echtzeitfähiges Kommunikationssystem etabliert. Die Zeitscheiben-orientierten Varianten EtherCAT [40] und Time-Triggered Ethernet (TTE) [54] eignen sich aufgrund ihrer Spezifikation am besten für den automotiven Bereich. Deshalb beschränkt sich die folgende Betrachtung auf diese zwei Lösungen.

Bei EtherCAT wird ein logischer Ring aufgebaut, wobei jeder Teilnehmer einen bestimmten Platz innerhalb eines Ethernet-Pakets zugewiesen bekommt. EtherCAT benötigt keinen speziellen Master. Die Slaves müssen jedoch einen speziellen EtherCAT-Controller besitzen, der die für den Knoten relevanten Daten aus dem Ethernet-Frame on-the-fly ausliest bzw. hinzufügt. Die Controller sind in Hardware implementiert, wodurch die Ausfallwahrscheinlichkeit sehr gering ist. Dieser kann als eine Art Bus-Guardian angesehen werden, so dass ein „babbling idiot“-Teilnehmer das restliche System nicht beeinflussen kann. Das EtherCAT Protokoll entspricht IEC 61508 SIL 3 [39]. Standard-Ethernet-Pakete werden azyklisch, d. h. zwischen den Echtzeit-Pakten, gesendet. Sie werden dabei durch einen EtherCAT-Frame getunnelt („Ethernet over EtherCAT“).

Bei TTE sind die Bus-Guardians in den Ports der Switches integriert. Auch hier erhält ein Teilnehmer einen bestimmten Zeitschlitz, in dem er senden darf. Falls mehrere Switches zwischen Sender und Empfänger liegen, werden die benötigten Ports gleichzeitig und nur für diese a priori festgelegte Übertragung durchgeschaltet. Jedes Datum muss in einem eigenen Ethernet-Paket übertragen werden, wodurch die effektive Datenmenge geringer ist als bei EtherCAT. Standard-Ethernet-Pakete werden unverändert durch das Netz geschickt und können von den Switches unterbrochen werden, falls ein Echtzeit-Paket zur Übertragung bereitsteht. Das abgebrochene

Paket wird danach erneut gesendet. Somit kann ein kurzes Delay mit geringem Jitter für die Echtzeit-Pakete garantiert werden.

Auch für den Multimedia-Bereich ist Ethernet hervorragend geeignet. Mit einer Übertragungsrate bei Fast Ethernet von 100 Mbit/s werden die Anforderungen für videobasierte Fahrerassistenzsysteme und Multimedia-Anwendungen erfüllt.

Gleichzeitig treibt die Consumer-Welt die Entwicklung von Ethernet weiter voran, so dass dieses Kommunikationsnetz mit den steigenden Anforderungen skalieren wird.

2.3.3 Auswahl

Ethernet bietet das größte Potenzial im Hinblick auf gleichzeitige Übertragung von Echtzeit-Daten mit garantierter maximaler Übertragungszeit und größeren Datenmengen, wie sie für Multimedia-Anwendungen auftreten. Durch Einführung eines eigenen Ethernet Typs und den Einsatz spezieller Switches ist es möglich, Echtzeit-Pakete priorisiert zu behandeln. Switches bieten zudem eine sehr gute Möglichkeit, durch Überwachung der Ports eine Fehlerausbreitung zu verhindern.

2.3.4 Anpassungen des Kommunikationsnetzes

Die Zeitsteuerung der o. g. Ethernetvarianten vereinfacht eine Verifikation des Systems und ist für Standard-Regelaufgaben prädestiniert. Jedoch leidet die Flexibilität unter der starren Einteilung in Zeitschlitze.

Zukünftig legen die großen Datenmengen aus den Anwendungen im Infotainment-Bereich die Bandbreite des Kommunikationsnetzes fest. Wenn zudem die Anforderungen an Delay und Jitter für sicherheitskritische Anwendungen nicht in den Bereich von wenigen Mikrosekunden eindringen, kann auch die Ereignis-basierte Natur des Ethernet beibehalten werden. Dennoch müssen einige Anpassungen erfolgen, um eine obere Schranke für die Übertragungszeit angeben zu können.

Verbindungstyp

Bereits die heutigen Fahrerassistenzfunktionen bedienen sich einer Vielzahl unterschiedlicher Sensoren, wobei ein einzelner Sensor nicht mehr nur einer Funktion zugeordnet ist, sondern als Eingang für mehrere Funktionen dienen kann. Dieser Trend wird sich weiter verstärken, so dass im Kommunikationsnetz neben (logischen) Punkt-zu-Punkt Verbindungen auch Mehrfachverbindungen (Multi- bzw. Broadcast) unterstützt werden müssen. Standard-Ethernet bietet Multi- bzw. Broadcast bereits auf MAC-Ebene an.

Im Folgenden wird von einer $1 : n$-Beziehung ausgegangen, d. h. ein Erzeuger von Daten generiert und verschickt Daten an mehrere Abnehmer. Die Empfänger müssen dabei die für sie interessanten Daten abonnieren bzw. abbestellen, wenn sie diese nicht mehr erhalten wollen.

Um einerseits den Aufwand am Erzeuger gering zu halten und andererseits das Kommunikationsnetz mit so wenig wie möglich Paketen zu belasten, sendet der Erzeuger nur an eine Multicast-Adresse. Das Routing zu den Abonnenten übernehmen die Switches des Kommunikationsnetzes. Für sicherheitskritische Funktionen sind die Wege bereits vorbestimmt und teilweise redundant vorhanden. Das Setzen der Routen erfolgt durch die Management-Software (siehe Abschnitt 2.6).

Protokoll-Overhead

Für Daten kleiner Größe, die alle mit derselben Frequenz verschickt werden, ist EtherCAT in puncto Protokoll-Overhead sehr gut aufgestellt. In einem verteilten System, in dem Daten unterschiedlicher Größe mit unterschiedlicher Frequenz auftreten, ist es jedoch sinnvoll, die Daten in einzelnen Paketen zu verschicken. Falls die Möglichkeit besteht, Daten von Sensoren zu sammeln, können sich diese natürlich auch ein Paket teilen.

Um dem Anspruch auf geringen Protokoll-Overhead gerecht zu werden, wird wie bei EtherCAT oder TTE auf MAC-Ebene gearbeitet. Bild 2.4 zeigt den Aufbau eines Standard-Ethernet Frames mit der Erweiterung um virtuelles LAN (VLAN). Ebenfalls eingezeichnet ist die in der Spezifikation festgelegte minimale Wartezeit zwischen zwei aufeinanderfolgenden Frames (engl.: *Inter Frame Gap* – IFG). Der Overhead für Ethernet Pakete mit VLAN-Unterstützung beträgt demnach 42 Bytes. Die Mindestlänge eines Ethernetframes (ohne Präambel und Start-Frame-Delimiter SFD) ist mit 64 Bytes spezifiziert, so dass bei weniger als 46 Datenbytes (ohne VLAN) bzw. 42 Datenbytes (mit VLAN) der Overhead entsprechend größer ausfällt.

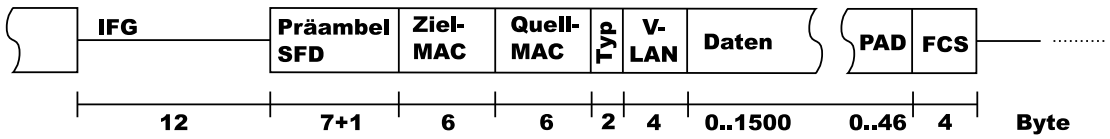


Bild 2.4: Aufbau Ethernet Frame

Separierung

Damit sichergestellt ist, dass die sicherheitskritischen Pakete, die üblicherweise zu den Anwendungen mit harten Echtzeitanforderungen gehören, den sonstigen Paketen vorgezogen werden, erfolgt eine Priorisierung der Pakete. Die Erweiterung des Ethernetstandards um VLAN erlaubt von Natur aus 3 Bit, also 8 Prioritätsstufen. Diese Anzahl ist jedoch nicht ausreichend, um jeder Anwendung bzw. jedem zugehörigen Paket eine eigene Priorität zuzuteilen. Deshalb wird für den folgenden Ansatz die Interpretation des VLAN-Feldes angepasst, ohne den Ethernetstandard dadurch zu verletzen.

Das Prioritätsfeld des VLAN-Tags dient zur Kategorisierung des Paketes. Die feinere Priorisierung innerhalb der Kategorien erfolgt dann über das VLAN-Id-Feld (siehe Bild 2.5).

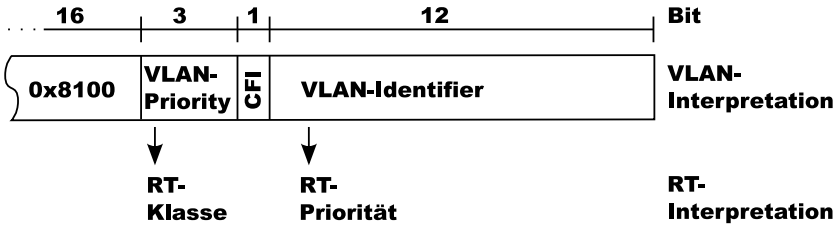


Bild 2.5: Echtzeit-Interpretation des VLAN-Tags

Eine zusätzliche Überwachungsmöglichkeit in den Switches besteht in der Verwendung von Algorithmen zur Begrenzung der Datenrate, wie beispielsweise „Leaky“ oder „Token Bucket“. Dabei wird jedem Paket-Typ a priori eine maximal erlaubte Rate zugewiesen. Der Token Bucket Algorithmus erweitert den Leaky Bucket Algorithmus dadurch, dass in definierbarem Rahmen eine teils geringere Datenrate durch gelegentliche höhere Datenraten ausgeglichen werden können, solange im Mittel die vorgegebene Datenrate eingehalten wird. Wird die mittlere Rate überschritten, meldet der Switch einen Fehler an das Diagnose- und Fehlermanagement[1)].

Mit dieser Anpassung lässt sich das Kommunikationsnetz theoretisch mit Standard-Komponenten aufbauen. Standard bedeutet in diesem Fall, dass es sich um Managed Switches handeln muss, die mit VLAN umgehen können müssen. Dadurch verliert man aber die Priorisierung innerhalb der Kategorien, wodurch sich die Vorhersagbarkeit dahingehend verschlechtert, dass auch die Pakete mit hoher Priorität denselben Verzögerungen ausgesetzt sind wie die mit niedrigen Prioritäten.

RT-Protokoll

Die acht VLAN-Prioritäten werden ähnlich der Quality of Service (QoS) Einteilung in Prioritäts-Klassen unterteilt. In die höchste Klasse fallen u. a. die Pakete für die Uhrensynchronisation. Dies garantiert geringstmöglichen Jitter der Pakete und damit maximale Synchronität der verteilten Uhren (System-Klasse). Drei weitere Klassen sind den harten Echtzeitaufgaben zugeordnet (HRT-Klasse). Die restlichen VLAN-Prioritäten können analog zu der QoS-Einteilung in Videoströme, Audioströme, normalen Verkehr und Hintergrundverkehr aufgeteilt werden (QoS-Klasse). Zu beachten ist, dass Ethernet-Pakete ohne VLAN-Markierung für systemfremde (third-party) Pakete verwendet werden. Diese werden als niederpriorste Pakete angesehen, die von allen VLAN-markierten Paketen unterbrochen werden können. Tabelle 2.1 zeigt die Zuordnung der Prioritäts-Klassen zu VLAN-Prioritäten.

Die drei HRT-Klassen werden wie folgt unterteilt: Die höchste VLAN-Priorität erhalten die Pakete für hochratige Sensordaten. Die darauffolgende wird für Daten aus Sensoren mit größeren Datenmengen, z. B. für Videodaten für Fahrerassistenzsysteme verwendet. Die dritte Klasse wird für Pakete zur Fehlerbehandlung freigehalten. Pakete aus den HRT-Klassen sind nicht unterbrechbar. Die Weiterleitung erfolgt ähnlich zum Cut-through-Verfahren. Im Unterschied

[1)] Für Erläuterungen zur Komponente „Diagnose- und Fehlermanagement“ siehe Abschnitt 2.6

Prioritäts-Klasse	VLAN-Priorität	unterbrechbar
System	7	nein
HRT	6–4	nein
QoS	3–0	ja
third-party	–	ja

Tabelle 2.1: Zuordnung der Prioritäts-Klassen zu VLAN-Prioritäten

dazu wird nicht bereits nach der MAC-Zieladresse sondern erst nach dem VLAN-Feld mit dem Einsortieren in den Sendepuffer (Warteschlange) des Ausgangsports begonnen.

Pakete aus darauffolgenden Klassen sind von Paketen aus den höheren Prioritäts-Klassen unterbrechbar. Untereinander findet wie zwischen den HRT-Klassen ein nicht-unterbrechbares, prioritätsbasiertes Scheduling Anwendung.

Innerhalb jeder Prioritäts-Klasse werden die VLAN-IDs als Prioritäten interpretiert. Das höchstwertige Bit (MSB) der Priorität sollte direkt auf das CFI-Feld folgen, da dadurch bei gleichzeitig ankommenden Paketen das Paket mit der höheren Priorität bevorzugt behandelt werden kann. Gleiche Prioritäten werden nach FCFS (engl.: *First Come First Serve*) bedient. Für die Einhaltung des Protokolls sind die Switches zuständig.

Zur Unterstützung der Votierung sollte im Datenteil des Pakets vor den eigentlichen Nutzdaten die Anzahl der Sender, die dieses Paket versenden, angegeben werden.

Switch – Hardware

Am besten eignen sich die sog. Crossbar-Switches, bei denen jeder Port mit jedem beliebigen anderen freien Port gleichzeitig verbunden werden kann. Von nicht-blockierender Verarbeitung spricht man, wenn jedes eintreffende Paket zu jeder beliebigen Zeit an den designierten Ausgangsport weitergeleitet werden kann, unabhängig davon, ob der Ausgangsport von früher eingetroffenen Pakete belegt ist. Einfache Switches besitzen lediglich eine Warteschlange pro Eingangsport, so dass Blockierungen auftreten können. Verwendet man Warteschlangen an den Ausgangsport, muss der Switch intern mit einer höheren Datenrate arbeiten als an den Ports nach außen hin zugelassen ist, um Blockierungen an der Ausgangswarteschlange zu vermeiden. Meist kommen deshalb virtuelle Ausgangs-Warteschlangen (engl.: „Virtual Output Queue“ – VOQ [64]) zum Einsatz. Dabei sind an jedem Eingangsport so viele Warteschlangen implementiert, wie es Ausgangsports gibt. Die Pakete werden entsprechend der Zieladresse und der dafür gesetzten Routen in die jeweiligen Warteschlangen einsortiert. Ankommende Pakete werden dem Arbiter des entsprechenden Ausgangsports über separate Signalleitungen gemeldet. Die Arbitrierungslogik steuert, aus welcher Warteschlange das nächste Paket an den Ausgangsport durchgereicht wird. Somit kann der Switch intern wie extern mit derselben Datenrate arbeiten, ohne dass es zu Blockierungen kommt.

Für die Einhaltung des RT-Protokolls müssen die VOQs nach Prioritäten sortiert sein. Existiert an den Eingangsports nur eine Warteschlange pro Ausgangsport, ergeben sich folgende Implikationen. Für das vorliegende RT-Protokoll muss zusätzlich vor der Sortierung der VOQs überprüft werden, ob das Paket, das gerade an den Ausgangsport weitergeleitet wird, unterbrochen werden darf. Die Arbitrierungslogik muss damit umgehen können, dass Pakete während der Durchleitung abgebrochen werden können. Dies kann entweder durch die Umsortierung der gerade durchgeschalteten Warteschlange passieren, oder durch ein neues Paket aus einer höheren Prioritäts-Klasse in einer Warteschlange eines inaktiven Eingangsports.

Ordnet man jedem Ausgangsport nicht nur eine Warteschlange zu, sondern genau so viele, wie es Prioritäts-Klassen gibt, reduziert sich der Aufwand für das Einsortieren in die VOQs, da innerhalb der Prioritäts-Klassen keine Unterbrechung stattfindet. Nur wenn gerade ein Paket an den Ausgangsport weitergeleitet wird, wird dieses bei der Einsortierung nicht berücksichtigt. Falls der Arbiter die Weiterleitung unterbricht, wird das abgebrochene Paket wie ein neues Paket behandelt und in die VOQ einsortiert. Das vorliegende System benötigt vier Warteschlangen: drei für die Prioritäts-Klassen und eine für systemfremde Pakete. Sobald ein Paket für die Weiterleitung bereit ist, sendet die Logik des Eingangsports eine Anfrage an den Arbiter des Ausgangsports. Ein ähnliches Verfahren wurde im Forschungsprojekt IT_Motive 2020 prototypisch implementiert [74][75].

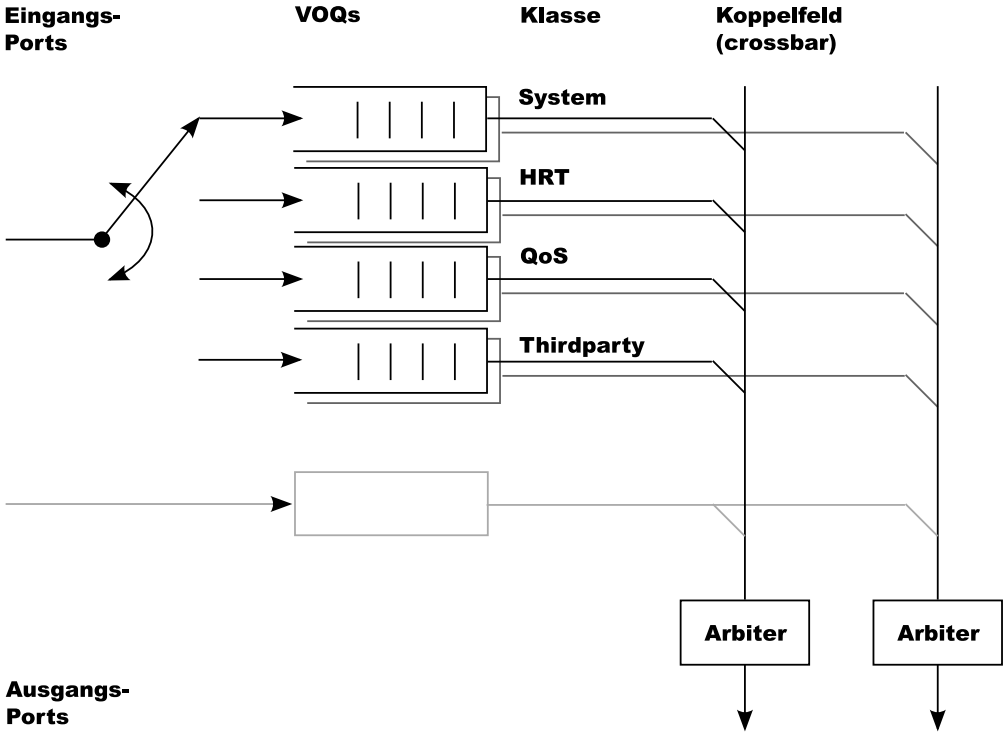


Bild 2.6: Aufbau des Crossbar-Switches mit Warteschlangen

Bild 2.6 zeigt den Aufbau des Switches beispielhaft anhand eines Eingangs- und zwei Ausgangsports. Die Logik des Arbiters lässt sich wie folgt in Pseudocode darstellen:

```
case wake_up_event
..from port_request:
  if (class_of_request == (System or HRT)) and
     (class_of_processed_packet < min(class_of_request,HRT))
    ; a new arbitration is required
    cancel current arbitration
  else
    ; do nothing - wait for port_is_free-event
    return
  endif
..from port_is_free:
    disconnect crossbar
endcase

for class in {System, HRT, QoS, Thirdparty}
  toforward = get_highest_prio(all_inputport_queues[class])
  if toforward != NULL
    connect toforward to outputqueue
    return
  endif
endfor

return
```

2.4 Recheneinheiten

Die Recheneinheiten sind für die Abarbeitung der Tasks im System zuständig. Das vorliegende System besteht aus einem Verbund von mehreren Recheneinheiten, die über das Kommunikationsnetz miteinander verbunden sind.

2.4.1 Anforderungen

Im Allgemeinen gelten für die Recheneinheiten ähnliche Anforderungen wie für das Kommunikationsnetz. Deshalb werden hier nur neue oder ergänzende Anforderungen aufgeführt.

Deterministische Verarbeitung

Auf den Recheneinheiten können sowohl sicherheitskritische als auch nicht-sicherheitskritische Applikationen ablaufen. Die meisten sicherheitskritischen Funktionen sind Regelungen, die einen oder mehrere Sensorwerte als Eingangsdaten benutzen, eine Berechnung durchführen und schließlich das Ergebnis entweder als eine Art virtueller Sensor anbieten oder an einen oder mehrere Aktoren weiterleiten. Um eine Ende-zu-Ende Deadline angeben zu können, müssen die Recheneinheiten eine deterministische Verarbeitung der sicherheitskritischen Funktionen sicherstellen.

Isolierung von Fehlern

Ein Fehler in einer Applikation darf sich nicht auf das Verhalten des Gesamtsystems auswirken. Insbesondere gilt dies für sicherheitskritische Anwendungen. Folgende Arten der Fehlerausbreitung sind zu verhindern.

Spezifikationsverletzung: Benötigt eine Applikation mehr Ressourcen (Prozessorleistung, Speicher, I/O: z. B. Kommunikationsverbindungen in Anzahl und Häufigkeit) als spezifiziert, dürfen andere Applikationen, die diese Ressourcen ebenfalls benutzen, nicht beeinträchtigt werden. Im schlimmsten Fall muss die fehlerhafte Applikation angehalten werden.

Liefert eine Applikation ein falsches Ergebnis, darf dieses nicht verwendet werden; eine Fehlerbehandlung muss eingeleitet werden. Im Folgenden werden falsche Ergebnisse durch Votierung für Tasks aus den Klassen 2 und 3 erkannt. Fehler von Tasks der Klasse 1 werden nicht betrachtet.

Liefert eine Applikation ein Ergebnis zu spät oder gar nicht, muss im Einzelfall entschieden werden, ob das Ergebnis noch verwendet werden kann, oder nicht. Für harte (hard) oder feste (firm) Echtzeit gilt jede Deadline-Verletzung als Fehler, der nicht toleriert wird. In diesem Fall muss eine Fehlerbehandlung eingeleitet werden.

Konsistentes System (Diagnose)

Zu jedem Zeitpunkt müssen alle Recheneinheiten, Sensoren, Aktoren und Switches – also alle Knoten im System – die gleiche Vorstellung über den für sie relevanten Teilbereich des Systems besitzen. Insbesondere müssen sich die intakten Knoten einig darüber sein, welcher Systemfehler vorliegt – sei es ein ausgefallener Knoten oder eine defekte Kommunikationsverbindung. Wäre dies nicht der Fall, kann es zu unterschiedlichen Fehler-Erholungsmaßnahmen kommen, die zu einem inkonsistenten Systemzustand führen.

Online Rekonfiguration

Im Fehlerfall müssen alle sicherheitskritischen Funktionen aufrecht erhalten werden. Dazu muss das Verhalten bei Rekonfiguration spezifiziert werden – z. B. wie lange die Funktion inaktiv sein darf – um daraus eine entsprechende Methode bei der Rekonfiguration für diese Funktion abzuleiten. Beispielsweise kann es notwendig sein, eine schnelle lokale Fehlererholungsmaßnahme zu initiieren und danach eine globale Systemüberprüfung und Rekonfiguration anzuschließen. Besonders hier wird deutlich, wie wichtig eine konsistente Systemvorstellung ist.

2.4.2 Realisierungsalternativen und Auswahl

Virtualisierung von Prozessor und Speicher

Die Hardware-Virtualisierung, oder Vollvirtualisierung, bietet die größte Flexibilität, hat jedoch einen hohen Overhead, wenn jede einzelne Hardware-Komponente emuliert werden muss. Zudem lassen sich aufgrund der vielen Schichten kaum mehr obere Schranken für eine Verarbeitung angeben, bzw. sind diese so pessimistisch, dass keine sinnvolle Nutzung der vorhandenen Ressourcen möglich ist.

Im Gegensatz dazu besitzt die Applikations-Virtualisierung einen geringeren Overhead und bietet gleichzeitig hohe Flexibilität. Da die virtuellen Umgebungen – meist Container oder Domänen genannt – innerhalb eines einzigen Betriebssystems laufen, ist es jedoch nahezu unmöglich, eine gegenseitige Abschottung unter allen Umständen zu garantieren. Ebenso muss im Falle einer Zertifizierung neben der eigentlichen sicherheitskritischen Applikation zusätzlich das gesamte zugrunde liegende Betriebssystem zertifiziert werden. Beispiele für kommerzielle Lösungen, die auf Applikations-Virtualisierung beruhen, sind Virtuozzo von Parallels oder das Container-Konzept von Sun Microsystems.

Mikro- oder Nano-Kerne (im Folgenden auch als Hypervisor bezeichnet) besitzen diese Nachteile nicht. Sie sind im einfachen Fall nur für die Weiterleitung von Hardware-Interrupts, die Prozessorzuteilung für virtuelle Domänen und die Speicherverwaltung zuständig. Sämtliche Treiber sind in den Userspace verlagert und werden entweder als native Task auf dem Kern (z. B. beim L4-Microkernel) oder innerhalb einer komplexeren Treiberdomäne (z. B. bei XEN) ausgeführt. Durch zusätzliche Hardware-Emulation können auf dem Kern auch modifizierte Betriebssysteme ausgeführt werden. Sicherheitskritische Echtzeit-Tasks können direkt auf dem Hypervisor aufsetzen. Der Overhead ist damit so gering wie bei einem „normalen“ Echtzeit-Betriebssystem. Die Mikro-/Nano-Kerne bestehen aus wenigen hundert Zeilen Software-Code, die Treiber können modular gestaltet werden. Ein auf Mikro-/Nano-Kern basierendes System lässt sich somit wesentlich einfacher zertifizieren.

Die Abschottung durch Speichervirtualisierung ist in allen genannten Typen durch eine „Memory Management Unit“ (MMU) gewährleistet.

Aufgrund der genannten Merkmale erfüllt ein System mit Hypervisor und Paravirtualisierung die Anforderungen am besten und wird im Folgenden zugrunde gelegt.

Virtualisierung der Kommunikationsnetzanbindung

Viele der Applikationen benötigen eine Verbindung zu Sensoren, Aktoren oder Applikationen auf anderen Recheneinheiten. Bei der vorliegenden Architektur bedienen sie sich eines einzigen Kommunikationsmittels. Auch hier muss eine Abschottung zwischen sicherheitskritischen und nicht-sicherheitskritischen Kommunikationen gewährleistet werden.

Eine Möglichkeit, die Kommunikation abzuwickeln, ist das Benutzen einer virtuellen Treiberdomäne bzw. Treibertask, über die jede Kommunikation läuft. Sie behandelt die Pakete auf gleiche Weise wie die Switches. Es stellt sich die Frage nach dem Scheduling dieser Domäne. Alle eingehenden Pakete müssen zunächst von der Treiberdomäne untersucht und anschließend die Ziel-Task oder -Domäne über ein vorliegendes Paket informiert werden. Meist wird das Paket mindestens einmal kopiert: von der Netzwerkkarte in den Kontext der Treiberdomäne und anschließend in den Kontext der Ziel-Task/-Domäne. Sowohl für Ein- als auch für Mehrprozessor-Systeme ist eine dynamische Priorisierung dieser Domäne notwendig. Da die Domäne von Tasks jeder Kategorie verwendet werden kann, besteht immer das Risiko einer Kompromittierung. Deshalb ist bei der Umsetzung besondere Sorgfalt notwendig, um Beeinflussungen durch nicht-sicherheitskritische Tasks auszuschließen.

Eine weitere Möglichkeit besteht darin, auf die kürzlich von Intel vorgestellte Technologie für virtuelle Netzwerkkarten (VMDq) zurückzugreifen. Dabei wird jeder virtuellen Task oder Domäne eine eigene (virtuelle) Netzwerkkarte zur Verfügung gestellt. Die Netzwerkkarte selbst entscheidet, für welche Ziel-Task/-Domäne das Paket bestimmt ist. Der Hypervisor setzt daraufhin lediglich ein Interrupt-Flag für die Ziel-Task/-Domäne und löst eine Neuberechnung des Schedules aus. Im günstigsten Fall befinden sich die Puffer der virtuellen Netzwerkkarten im jeweiligen Adressraum der Ziel-Task/-Domäne, so dass keine zusätzliche Kopieroperation durchgeführt werden muss. Aktuelle Realisierungen von virtuellen Netzwerkkarten verwenden intern einen Round-Robin Scheduler für die Warteschlangen der einzelnen Domänen. Für das vorliegende System muss das Scheduling jedoch dem der Switches angepasst werden.

Der Einfluss der Hardware-Virtualisierung sowie Schedulingüberlegungen werden in Abschnitt 3.3 genauer untersucht.

2.5 Sensoren und Aktoren

Beim vorliegenden System wird zwischen intelligenten und einfachen Sensoren bzw. Aktoren unterschieden. Intelligente Sensoren/Aktoren sind wie die Recheneinheiten an das Kommunikationsnetz angebunden. Einfache Sensoren/Aktoren werden über ein Gateway in das System eingekoppelt.

Das Gateway ist in den Switches integriert. Jede Art von Bussystem kann so mit dem System verbunden werden. Das Gateway ist für die einfachen Sensoren/Aktoren die letzte intelligente Instanz und damit zuständig für die Generierung von Zeitstempeln bzw. für die zeitlich korrekte Weiterleitung von Nachrichten sowie für die Übersetzung der Protokolle zwischen den Bussystemen.

Bei der Verwendung der Zeitstempel eines einfachen Sensors/Aktors muss der durch das Bussystem entstehende Jitter bzw. die maximal mögliche Verzögerung zwischen Sensor/Aktor und Gateway berücksichtigt werden.

Anforderungen und Umsetzung der intelligenten Sensoren und Aktoren sind aus Sicht des Kommunikationsnetzes mit denen der Recheneinheiten identisch, jedoch sind verschiedene Skalierungen möglich: Ein minimaler intelligenter Sensor oder Aktor muss lediglich über eine Ethernet-Schnittstelle verfügen und sich als Slave über PTP (engl.: *Precision Time Protocol* – IEEE 1588 [86]) synchronisieren lassen. Falls es sich um einen sicherheitskritischen Sensor handelt, müssen zusätzlich VLAN-Unterstützung und Systemsoftware (Slave) vorhanden sein. Auch eine Recheneinheit mit Ein-/Ausgabe-Funktion und virtuellen Domänen kann als Sensor/Aktor dienen.

2.6 Komponenten der Systemsoftware

Die Systemsoftware stellt die Funktionen zur Verwaltung des Systems sowohl im fehlerfreien Betrieb als auch im Fehlerfall bereit. Bild 2.7 zeigt die wichtigsten Funktionen im Überblick. Die einzelnen Softwarekomponenten sind mit „Component“ bezeichnet. Die Kennzeichnung „I“ weist darauf hin, dass es sich um Infrastruktur-Komponenten handelt.

Alle beschriebenen Master- und Management-Komponenten sowie die Datenbanken sind für einen korrekten Betrieb notwendig und deshalb redundant als 2-aus-3 Komponenten auszulegen. Die Slave-Komponenten befinden sich jeweils auf den Recheneinheiten und sind demnach nur einfach vorhanden.

Die Komponenten *Diagnose und Fehlermanagement* sowie der HRT-Teil des *Ressourcenmanagements* wurden vom Autor, die übrigen Komponenten von weiteren Teammitgliedern des Projekts IT_Motive 2020 prototypisch implementiert. Im Demonstrator wurden die Komponenten nicht redundant ausgelegt, jedoch konnte die Validität des Konzepts gezeigt werden.

Im folgenden werden die einzelnen Komponenten sowie deren Zusammenspiel erläutert.

2.6.1 Diagnose und Fehlermanagement

Die Komponente *Diagnose und Fehlermanagement* (Bild 2.8) besteht aus den Unterkomponenten Diagnoseslave, Diagnosemaster und Fehlermanagement. Diese sind zuständig, Fehler zu erkennen, zu lokalisieren und zu klassifizieren sowie im Anschluss geeignete Maßnahmen zur Eingrenzung und wenn möglich zur Behebung einzuleiten.

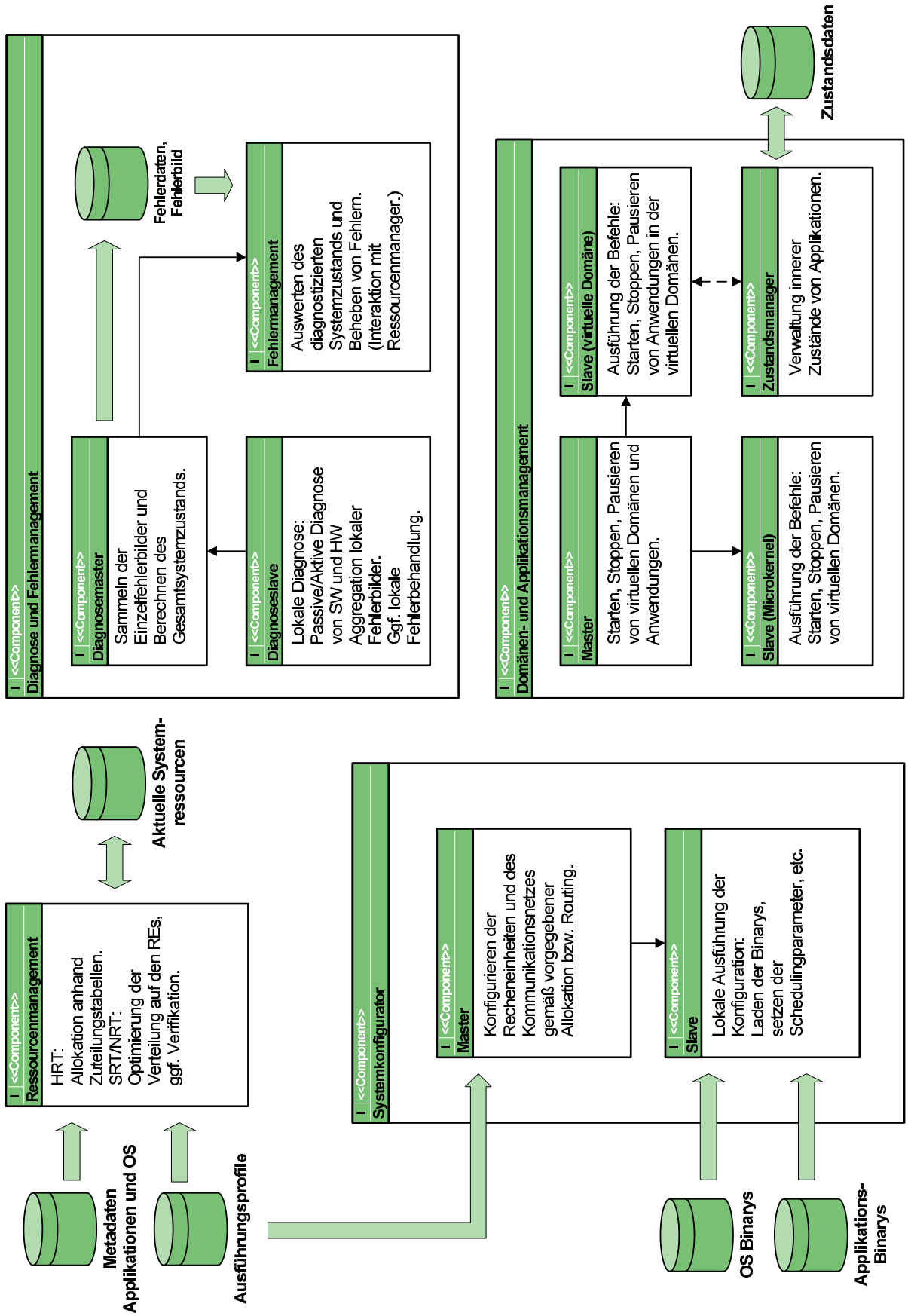


Bild 2.7: Überblick Systemsoftware

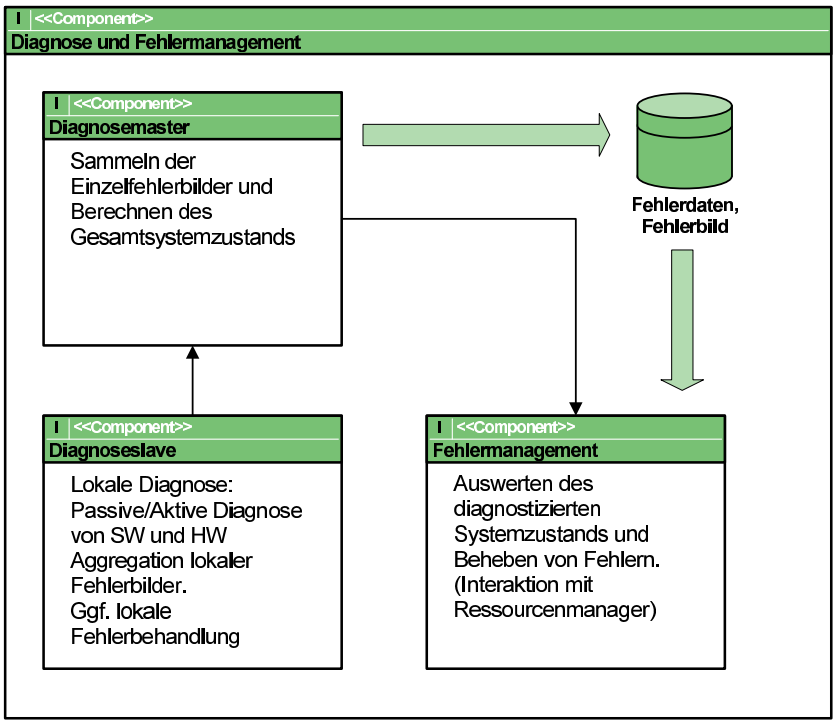


Bild 2.8: Systemsoftware: Diagnose und Fehlermanagement

Die Komponente *Diagnoseslave* ist mehrfach vorhanden. Die einzelnen Instanzen befinden sich an verschiedenen Stellen im System, um spezifische Informationen zu sammeln. Dadurch besitzen sie lediglich eine lokal beschränkte Sicht und dürfen nur in wenigen Ausnahmefällen die Entscheidungen zur Fehlerbehebung durchführen. Im folgenden werden nur die wichtigsten Slaves kurz erläutert. Einer der Slaves befindet sich auf unterster Ebene, im Hypervisor. Dort werden u. a. Informationen über Parameter der virtuellen Domänen (z. B. gesamte Prozessorauslastung, Speicherbedarf, I/O-Last), über den Zustand von Hardware-Komponenten (z. B. über das Anstoßen von Hardware-Selbsttests) und über das Einhalten der Zeitbedingungen (Schedule der virtuellen Domänen) aufgenommen. Ein weiterer Slave innerhalb der virtuellen Domäne sammelt Informationen über Prozessorlast (relativ zu den zugewiesenen Prozessorressourcen), Speicherauslastung und Scheduling der Tasks. Eine der wichtigsten Aufgaben erfüllen die Diagnoseslaves in den Treiberdomänen. Dort findet zusätzlich die Überprüfung auf Einhaltung von Zeitbedingungen der Nachrichten aus einem Verbund von Cotasks statt. Die Verarbeitungsdauer dieser Meldung hat direkten Einfluss auf die Reaktionszeit zur Rekonfiguration des Systems. In den Switches des Kommunikationsnetzes sind Diagnoseslaves platziert, die die Konnektivität aller Ports überwachen.

Der *Diagnosemaster* sammelt die Informationen der Diagnoseslaves und fügt sie zu einem einheitlichen Fehlerbild zusammen. Danach wird das gewonnene Fehlerbild in der Datenbank *Fehlerdaten und Fehlerbild* abgelegt und das Fehlermanagement informiert. Zudem übermittelt der Diagnosemaster die Informationen über verwendete Ressourcen an das Ressourcenmanagement. Auch der Diagnosemaster muss redundant mit gegenseitigem Abgleich arbeiten: Alle

Ergebnisse und ggf. Zwischenergebnisse werden zwischen den Instanzen des Diagnosemasters ausgetauscht und votiert.

Das *Fehlermanagement* berechnet den tatsächlich vorliegenden Fehler auf Basis der Daten aus der Datenbank „Fehlerdaten und Fehlerbild“(Fehlerlokalisation). In Abstimmung mit dem Ressourcenmanagement wird die Strategie zur Behebung des Fehlers mit den noch vorhandenen Ressourcen festgelegt und eingeleitet. Letzteres geschieht durch die Signalisierung an den Master des Systemkonfigurators.

2.6.2 Ressourcenmanagement

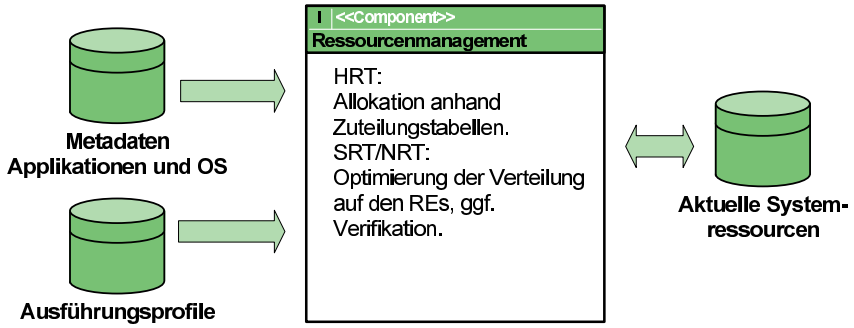


Bild 2.9: Systemsoftware: Ressourcenmanagement

Das Ressourcenmanagement (Bild 2.9) verwaltet die im System verfügbaren Hardware-Ressourcen, wie Recheneinheiten, Kommunikationswege, Prozessorleistung, Speicher, etc. Diese Informationen kommen von den Komponenten *Diagnosemaster* und *Fehlermanagement* und werden in der Datenbank *Systemressourcen* abgelegt.

Auf Basis dieser Informationen bestimmt das Ressourcenmanagement die Verteilung der Funktionen auf die Recheneinheiten. Für die sicherheitskritischen (HRT) Funktionen geschieht dies anhand zur Designzeit festgelegter Zuordnungstabellen, die in der Datenbank *Ausführungsprofile* abgelegt sind. Für weniger kritische Funktionen mit weichen oder keinen Deadlines kann die Zuteilung auch zur Laufzeit erfolgen. Die hierfür benötigten Eingangsdaten (z. B. benötigte Prozessorleistung, Speicher) stehen in der Datenbank *Metadaten für Applikationen und Betriebssysteme*.

2.6.3 Systemkonfigurator

Der Systemkonfigurator (Bild 2.10) ist eine reine ausführende Komponente. Deren Master erhält vom Fehlermanagement die Information, welche Allokation aus der Datenbank „Ausführungsprofil“ verwendet werden soll. Ebenso wird ihm die Verteilung der SRT und NRT Applikationen mitgeteilt. Das kombinierte Profil enthält den Gesamtsystemzustand. Der Master der Systemkonfiguration spaltet die Information für die einzelnen Komponenten auf und übergibt nur die relevanten Daten an die Slaves.

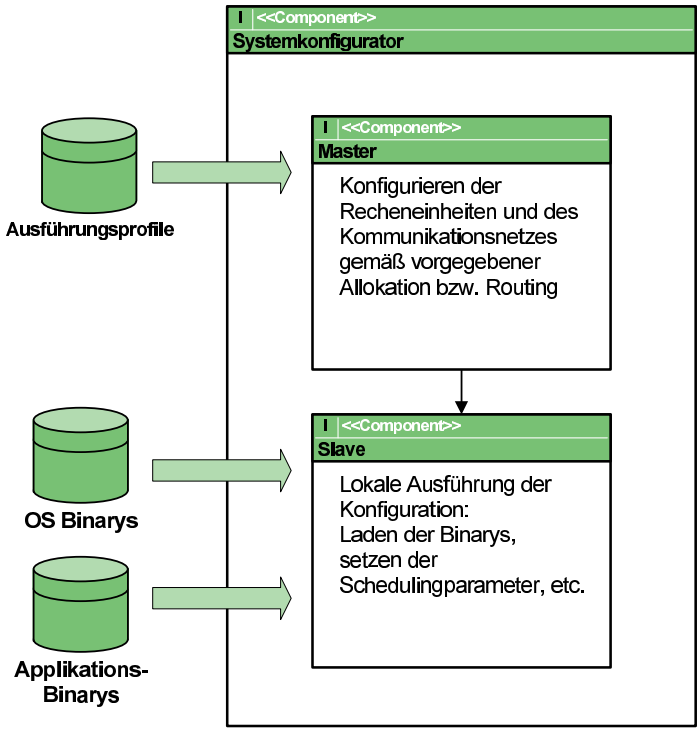


Bild 2.10: Systemsoftware: Systemkonfigurator

Jede Recheneinheit und jeder Switch besitzt eine Slave-Komponente des Systemkonfigurators. Bei den Switches ist er u. a. für das Setzen von Routing-Tabellen zuständig. Auf den Recheneinheiten befindet sich der Slave zum einen auf Ebene des Hypervisors: Gemäß der übermittelten Konfiguration holt er sich die Binär-Pakete der geforderten virtuellen Domänen und setzt hierfür Scheduling- und Ressourcen-Parameter. Zum anderen kann sich innerhalb einer virtuellen Domäne ein Slave befinden, der für die Konfiguration des jeweiligen Betriebssystems und das Laden der gewünschten Applikationen zuständig ist. Jedoch ist dieser nur dann erforderlich, wenn es sich um eine dynamisch konfigurierbare Domäne handelt, d. h. Applikationen entfernt und nachgeladen werden können. Für sicherheitskritische Anwendungen besteht oft eine statische Zuordnung zu einer speziellen Domäne, so dass hier die Slave-Komponente entfällt.

2.6.4 Domänen- und Applikationsmanagement

Das Domänen- und Applikationsmanagement (Bild 2.11) ist ebenfalls eine reine ausführende Komponente. Sie dient dazu, Domänen und Applikationen zu starten, zu stoppen oder anzuhalten. Die Signalisierung erhält die Master-Komponente vom Systemkonfigurator. In der realen Umsetzung können die Master-Komponenten des Systemkonfigurators und des Domänen- bzw. Applikationsmanagements zusammengefasst werden.

Die Slaves befinden sich auf den Recheneinheiten sowohl im Hypervisor als auch innerhalb der Domänen. Sie führen die Befehle des Masters zum Starten, Stoppen oder Pausieren der

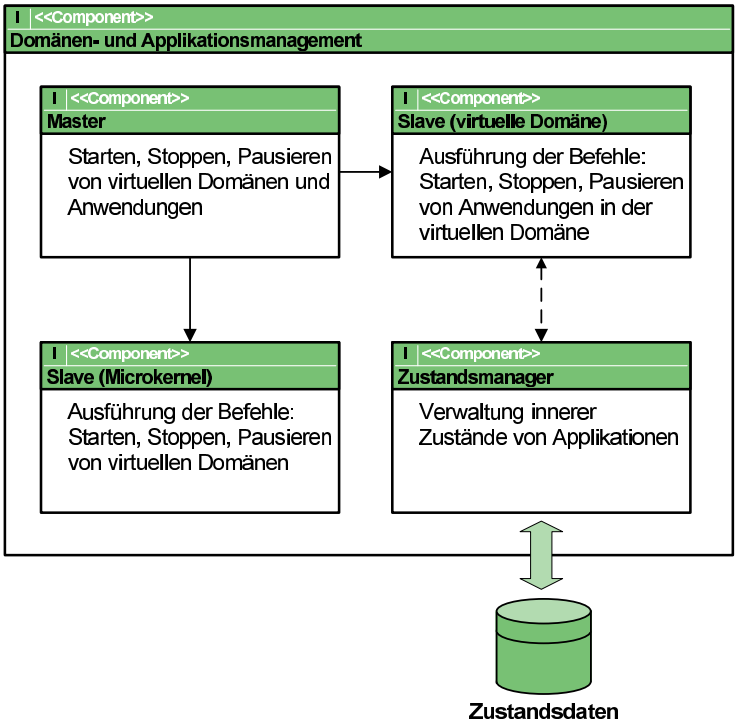


Bild 2.11: Systemsoftware: Domänen- und Applikationsmanagement

virtuellen Domänen oder der Applikationen darin aus. Switches besitzen kein Domänen-/Applikationsmanagement; sie werden über die Komponente Systemkonfigurator gesteuert.

2.6.5 Zeitsynchronisation

Die in der Arbeit beschriebenen Vorgehensweisen zur Synchronisation benötigen eine möglichst genaue Zeitsynchronisation zwischen den Recheneinheiten. In Bild 2.12 ist dargestellt, dass es sich nicht um gleichberechtigte Komponenten zur Zeitsynchronisierung handelt. Die Master befinden sich in der vorliegenden Architektur in den Switches. Die Slaves befinden sich in den Sensoren, Aktoren und ggf. auch in den Recheneinheiten, falls diese über eine eigene RTC verfügen und keinen Zugriff auf die RTC des Switches besitzen.

2.6.6 Abgrenzung

Die hier vorgestellten Software-Komponenten bilden nur jenes Subset der Basissoftware, welches für die Bereitstellung der in dieser Arbeit beschriebenen fehlertoleranten Funktionen (Applikationen) und die Rekonfiguration des Systems im Fehlerfall notwendig sind.

Beispielsweise sind die beschriebenen Komponenten auch in der Lage, zwischen Betriebsmodi (z. B. Stand, Fahrbetrieb, Werkstattmodus) umzuschalten. Jedoch werden hierfür zusätzliche Software-Komponenten benötigt, die eine solche Umschaltung initiieren.

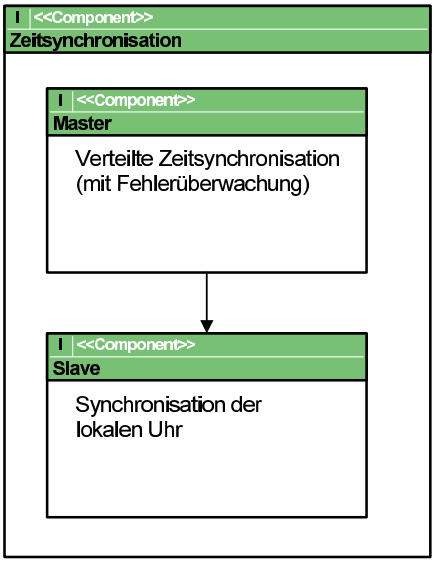


Bild 2.12: Systemsoftware: Zeitsynchronisation

3 Zeitverhalten des Systems

In diesem Kapitel wird das Zeitverhalten des beschriebenen Systems untersucht. Zeitliche Randbedingungen für ein synchrones und deterministisches System werden analysiert und Möglichkeiten zu deren Einhaltung für das vorliegende System angegeben. Die anschließenden Abschnitte konkretisieren die dabei auftretenden Variablen und untersuchen den Einfluss verschiedener Scheduling-Strategien auf die Antwortzeit der Recheneinheiten. Das Scheduling des Kommunikationsnetzes wird nicht variiert. Abschließend werden die Erkenntnisse der einzelnen Abschnitte zusammengefügt, um das zeitliche Verhalten des gesamten Systems im fehlerfreien Fall darzustellen. Das Verhalten im Falle eines Einfachfehlers wird im nächsten Kapitel beschrieben.

3.1 Synchronität

Es wird davon ausgegangen, dass die Sensoren und Aktoren, die direkt an das Kommunikationsnetz angeschlossen sind, als PTP[1)]-Slave konfiguriert sind und somit über eine exakte globale Zeit verfügen. Die Sensoren bzw. Aktoren, die über Gateways angeschlossen sind, besitzen diese i. d. R. nicht; der Zeitstempel wird erst durch das Gateway gesetzt.

Dennoch muss im gesamten System sichergestellt sein, dass zusammengehörige Berechnungen auf konsistenten Daten durchgeführt werden. Ohne zusätzliche Maßnahmen ist dies nicht gewährleistet. Durch unterschiedliche Verzögerungen im Kommunikationsnetz sowie durch unterschiedliche Schedules auf den Recheneinheiten können Daten zu einem bestimmten Zeitpunkt auf einer Recheneinheit bereits vorliegen, während auf einer anderen noch ältere Daten als aktuell angesehen werden.

Im vorliegenden System wird Fehlertoleranz durch aktive Replikation von Tasks bzw. Domänen mit anschließendem Vergleich der Ergebnisse erreicht. Gerade hierbei ist o. g. Anforderung, auf identischen Daten zu arbeiten, zwingend erforderlich.

3.1.1 Synchronisierungsvarianten

Determinismus der Replikate kann z. B. dadurch erreicht werden, dass 1) Ereignisse und Nachrichten überall im System in derselben Reihenfolge eintreffen und 2) jede (lokale) Scheduling-Entscheidung mit den anderen Schedulern im System abgeglichen wird. Um 1)

[1)] Engl.: „Precision Time Protocol“ (IEEE1588)

zu gewährleisten, wird ein Kommunikationsnetz mit entsprechendem Protokoll benötigt. Im vorliegenden Ethernet-basierten Kommunikationsnetz ist keine derartige Unterstützung vorhanden. Der zusätzliche Overhead, der durch 2) entsteht, ist nicht tragbar. Für jede Scheduling-Entscheidung fließen die zusätzlichen Latenzen durch den Nachrichtenaustausch über das Kommunikationsnetz und die durch die Bearbeitung des Protokolls selbst hinzukommenden Verzögerungen in die Reaktionszeit ein.

Eine weitere Möglichkeit besteht darin, das Scheduling offline festzulegen. Dadurch ist das System starr und unflexibel gegenüber zusätzlichen Tasks.

In [84] führt Poledna sog. *„Timed Messages“*, im Folgenden auch mit *TM* bezeichnet, ein. Durch das Setzen eines Gültigkeitszeitpunktes für jede Nachricht wird ein deterministisches Verhalten garantiert, unabhängig von lokalen Scheduling-Entscheidungen oder Verzögerungen durch das Kommunikationsnetz. Eine TM darf erst zu ihrem Gültigkeitszeitpunkt „aktiv“ werden, d. h. z. B. die Zieltask aktivieren. Weitere Voraussetzungen sind eine systemweit synchronisierte Uhr, ein deterministisches Kommunikationsnetz sowie eine deterministische Verarbeitung der Tasks.

Die letztgenannte Variante ist bzgl. des Kommunikationsoverheads sehr effizient und integriert sich am besten in das vorliegende System. Da sich die Betrachtung in [84] auf einzelne Nachrichten beschränkt, muss eine Erweiterung auf das Gesamtsystem erfolgen. Für den Einsatz in virtuellen Umgebungen müssen des Weiteren Anpassungen bei der Bestimmung des Gültigkeitszeitpunktes durchgeführt werden. Diese Aspekte werden in den folgenden Abschnitten dieses Kapitels näher beschrieben. Zunächst erfolgt eine allgemeine Betrachtung.

Die Idee der TM ist es, einen Gültigkeitszeitpunkt t_v zu finden, zu dem sichergestellt ist, dass die Nachricht überall im System bekannt ist. Dabei werden sowohl die maximale Abweichung der Uhren um $\pm\eta$ in Bezug auf die reale Zeit, als auch alle möglichen auftretenden Verzögerungen δ durch Kommunikation oder Scheduling-Entscheidungen ε berücksichtigt. Bild 3.1 gibt einen Überblick, aus welchen Zeiten sich der Gültigkeitszeitpunkt zusammensetzt. Beispielsweise ergibt sich für den Fall, dass zum (realen) Zeitpunkt $t_0 \pm \eta$ eine Nachricht mit der maximalen Übertragungsdauer δ_{WCTD} von REi nach REj verschickt wird: $t_v = t_0 + \delta_{\text{WCTD}} + 2\eta$.

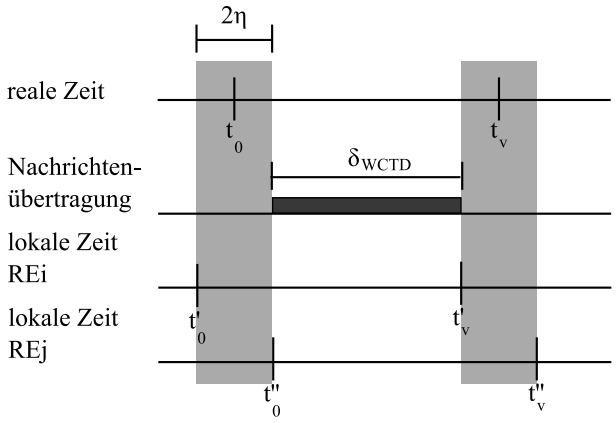


Bild 3.1: Timed Messages

3.1.2 Abweichung der lokalen Uhren

Die maximale Abweichung aller Uhren ist η. Sie hängt von der Systemstruktur, dem verwendeten Protokoll und seiner Implementierung (z. B. in Hardware oder in Software) ab. Für das vorliegende System wird das „*precision time protocol*“ (IEEE 1588 [1]), kurz PTP, verwendet. Laut Standard ist in Ethernet-basierten Systemen eine Genauigkeit im Sub-Mikrosekunden-Bereich möglich. Verschiedene Hardware-Hersteller bieten Ethernet-Transceiver an, die sowohl ein- bzw. ausgehende Pakete mit Zeitstempeln versehen, als auch das PTP Protokoll in Hardware implementieren. Laut Datenblatt erreicht beispielsweise der Transceiver-Baustein DP83640 von National Semiconductor eine Genauigkeit von $2\eta_{1:1}$=8 ns zwischen zwei PTP-Komponenten.

Die Berechnung von η für die hier verwendete Systemstruktur wird bestimmt durch die Kaskadierung der Switches und damit der PTP-Teilnehmer. Im schlechtesten Fall akkumuliert sich bei i aufeinander folgenden Komponenten auch die maximale Abweichung zu $\eta = i\eta_{1:1}$.

3.1.3 Vorverarbeitung in den Recheneinheiten

In einfachen Systemen kann eine Nachricht direkt zur Aktivierung einer Task herangezogen werden. Im vorliegenden System kann jedoch eine Vorverarbeitung der TM in den Recheneinheiten stattfinden, die vor dem Gültigkeitszeitpunkt t_v liegt[2)], zu dem die Daten von der Zielfunktion verwendet werden dürfen.

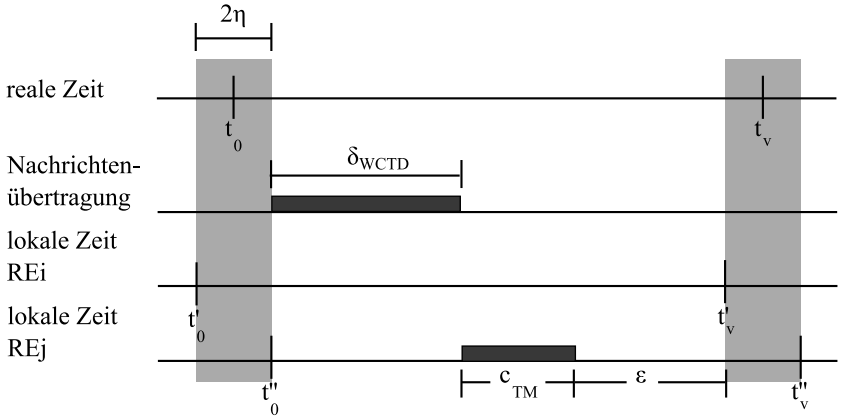


Bild 3.2: Validitätszeitpunkt mit Vorverarbeitung in den Recheneinheiten

Die virtuelle Domäne, in der die Vorverarbeitung stattfindet, unterliegt dem Scheduling auf der Recheneinheit. Die Zeit zwischen der frühestmöglichen und der spätestmöglichen Aktivierung der Domäne wird im Folgenden als maximaler Scheduling-Fehler ε bezeichnet. Die benötigte Rechenzeit für die Vorverarbeitung der TM sei c_{TM}. Daraus ergibt sich für t_v (siehe auch Bild 3.2):

$$t_v = t_0 + \delta_{\text{WCTD}} + c_{TM} + \varepsilon + 2\eta \tag{3.1}$$

[2)] Diese vorverarbeitende Task befindet sich innerhalb einer virtuellen Domäne. Deshalb wird im Folgenden auch vom Scheduling der Domänen gesprochen.

3.2 Kommunikationsnetz

Die Übertragungsdauer δ lässt sich in zwei Anteile aufspalten. Zum einen in die Verzögerungen durch Wartezeiten in den Switches, zum anderen in die Übertragung der Daten selbst. Sie hängt von verschiedenen Charakteristika des verwendeten Kommunikationsnetzes ab. Im Folgenden wird δ für das vorliegende Ethernet-basierte Kommunikationsnetz mit seinen Anpassungen bestimmt. Es wird nur auf die Weiterleitung bzw. die möglichen Verzögerungen der Nachrichten aus den höheren Prioritätsklassen, die den sicherheitskritischen Applikationen zuzuordnen sind, eingegangen.

3.2.1 Transport-Delay

Die Dauer der eigentlichen Übertragung eines Ethernet-Pakets mit b Bytes an Nutzdaten beträgt bei einer Bandbreite von B (in [bit/s]) ohne IFG, Präambel und SFD

$$\delta_{T*} = \frac{(\max(42, b) + 22) \cdot 8\,\text{bit}}{B}\,, \tag{3.2}$$

bzw. mit IFG, Präambel und SFD

$$\delta_T = \delta_{T*} + \frac{20 \cdot 8\,\text{bit}}{B}\,. \tag{3.3}$$

3.2.2 Verzögerungen in den Switches

Trifft ein Paket an einem Eingangsport eines Switches zur Weiterleitung an einen Ausgangsport ein, so gibt es drei mögliche Belegungen des Ausgangsports.

Freier Ausgangsport

Die Verzögerung bestimmt sich allein durch das (modifizierte) Cut-Through Verfahren; die Weiterleitung kann sofort nach der Auswertung der Zieladresse, d. h. nach dem 14. Byte (inkl. Präambel und SFD), beginnen:

$$\delta_F = \frac{14 \cdot 8\,\text{bit}}{B}\,.$$

Ausgangsport mit unterbrechbarem Paket belegt

In diesem Fall kann die Entscheidung, ob das gerade zu sendende Paket unterbrochen werden soll, erst nach Auswertung des VLAN-Tags erfolgen. Nach dem Abbruch muss das IFG abgewartet werden, bevor mit dem Senden des neuen Ethernet-Frames begonnen werden darf:

$$\delta_S = \delta_F + \frac{24 \cdot 8\,\text{bit}}{B} \,.$$

Ausgangsport mit nicht-unterbrechbarem Paket belegt

Blockierungszeiten treten bei gleichzeitigem Weiterleitungswunsch an einen Ausgangsport von ununterbrechbaren Paketen in den Klassen HRT und System auf (non-preemptive fixed priority scheduling). Für die folgenden Betrachtungen werden die Prioritätsklassen (System und HRT) aufgehoben und die Indizes i so vergeben, dass einerseits die Priorisierung innerhalb einer Prioritätsklasse beibehalten wird, andererseits die Indizes aller Nachrichten einer höheren Prioritätsklasse *kleiner* sind als die einer niedrigeren. Im Resultat sind die Prioritäten der Nachrichten anhand der Indizes erkennbar, wobei ein niedriger Index eine hohe Priorität bedeutet und umgekehrt.

Die maximale Verzögerung hängt von der Rate aller höherprioren Pakete ($1/p_j$ mit $j < i$) und deren Übertragungsdauern δ_{Tj} sowie der Verzögerung durch das größtmögliche Paket $\delta_{\max}$ aus der (ununterbrechbaren) HRT- bzw. System-Klasse ab. Die Bestimmung der maximalen Wartezeit δ_{Hi} erfolgt iterativ gemäß [60].

$$t^k = \delta_{\max} + \sum_{j=1}^{i-1} \left\lceil \frac{t^{k-1}}{p_j} \right\rceil \cdot \delta_{Tj} \tag{3.4}$$

mit $t^0 = 0$ und $k > 0$.

Die gesuchte Verzögerung ist gefunden, wenn sich zwischen zwei Iterationsschritten t^k nicht mehr ändert. Die maximale Verzögerung eines Pakets der Priorität i (δ_{Hi}) ist dann gleich t^k.

Für den Einsatz in online-Verfahren kann die Berechnungszeit von t^k mit Hilfe einer Abschätzung reduziert werden. Setzt man $t^k = t^{k-1}$ und löst die Aufrundungsoperation auf, ergibt sich:

$$t^k \leq \delta_{\max} + \sum_{j=1}^{i-1} \left(\frac{t^k}{p_j} + 1 \right) \cdot \delta_{Tj}$$

$$t^k \leq \delta_{\max} + t^k \sum_{j=1}^{i-1} \frac{\delta_{Tj}}{p_j} + \sum_{j=1}^{i-1} \delta_{Tj}$$

$$t^k \left(1 - \sum_{j=1}^{i-1} \frac{\delta_{Tj}}{p_j} \right) \leq \delta_{\max} + \sum_{j=1}^{i-1} \delta_{Tj}$$

$$\delta_{Hi} = t^k \leq \frac{\delta_{\max} + \sum_{j=1}^{i-1} \delta_{Tj}}{1 - \sum_{j=1}^{i-1} \frac{\delta_{Tj}}{p_j}} \tag{3.5}$$

3.2.3 Kaskadierte Switches

Für jeden durchlaufenen Switch s eines Pfades $P(W)$ einer Wirkkette W akkumulieren sich die Zeiten entsprechend. Zu beachten ist, dass auf unterschiedlichen Switches Nachrichten unterschiedlicher Priorität vorzufinden sind, d. h. dass δ_{Hi}^s für jeden Switch s unterschiedlich ist und getrennt berechnet werden muss.

$$\delta_{Hi} = \sum_{\forall s \in P(W)} \delta_{Hi}^s$$

Da $\delta_F < \delta_S < \delta_{\max}$ ist, und wegen der Verwendung des Cut-Through Verfahrens, ist das Transport-Delay in δ_{Hi} bereits enthalten. Für die maximale Ende-zu-Ende Übertragungszeit muss δ_T deshalb nur einmal berücksichtigt werden:

$$\delta_{\text{WCTD}} = \delta_{Hi} + \delta_T \tag{3.6}$$

Tatsächlich ist die Annahme, dass pro Switch das maximale Delay auftritt, etwas pessimistisch, da die Phasen der Nachrichten aus den vergangenen Berechnungen nicht berücksichtigt werden. In dieser Arbeit wird dennoch die pessimistische Schätzung verwendet. Das Hauptaugenmerk liegt auf der prinzipiellen formalen Darstellung des Zeitverhaltens und nicht auf der Optimierung einzelner Parameter.

3.3 Recheneinheiten

Das Zeitverhalten der Recheneinheiten ist maßgeblich durch den Scheduler und die Paketverarbeitung bestimmt. Im Folgenden wird daher der Einfluss verschiedener Scheduling-Strategien und Paketverarbeitungsmechanismen auf den maximalen Scheduling-Fehler

$$\varepsilon_{\text{Fall}}^{\text{Strategie}}$$

untersucht und bewertet. Der hochgestellte Index ist FP für Scheduling mit festen Prioritäten oder EDF für „earliest deadline first“. Der tiefgestellte Index bezeichnet die verschiedenen untersuchten Fälle. SC steht für Singlecore, MC für Multicore. Auf die Indizes wird verzichtet, wenn es sich um ein nicht näher spezifiziertes Zwischenergebnis handelt.

3.3.1 Schedulingtypen

Globales und lokales Scheduling

Prinzipiell stellt sich die Frage, ob globales oder lokales Scheduling zum Einsatz kommen soll. Globales Scheduling bedeutet, dass ein globaler Scheduler alle im System befindlichen Tasks auf die Rechnerknoten verteilt. Für einfache Systeme ist dies zwar effizienter und liefert kürzere Antwortzeiten [14]. Jedoch muss gewährleistet sein, dass eine beliebige Task zu jeder Zeit auf eine beliebige Recheneinheit verschoben werden kann. Mit der Größe der Applikationen steigt gleichzeitig auch der Aufwand für eine solche Migration. Im vorliegenden System laufen die Applikationen innerhalb virtueller Umgebungen, die nicht zwangsläufig auf allen Recheneinheiten verfügbar sind. In solchen Fällen müsste der Kontext der gesamten virtuellen Umgebung migriert werden. Bei der Migration einer virtuellen Maschine unter XEN wird – ohne Berücksichtigung der Vorbereitungsphasen – allein die Down-Time des Systems mit mindestens 50 ms angegeben [33]; eine obere Schranke kann nicht angegeben werden. Aus diesem Grund wird im Folgenden nur lokales Scheduling verwendet, bei dem der Hypervisor jeder Recheneinheit für die Zuteilung des Prozessors zu virtuellen Domänen zuständig ist.

Wie bei der Behandlung von Paketen im Kommunikationsnetz gesehen, gibt es eine Unterteilung in sieben Prioritäts-Klassen – von harter Echtzeit über weiche Echtzeit bis hin zu keinen Echtzeit-Anforderungen und third-party Kommunikation. Auf den Recheneinheiten spiegelt sich diese Aufteilung wider, wobei zusätzlich der Zugriff auf die gemeinsam genutzte Ressource (die physikalische Kommunikationsnetzanbindung wird im Folgenden auch als Netzwerkkarte bezeichnet) berücksichtigt werden muss.

Domänen-Scheduling

Direkt auf der Hardware ist der Hypervisor angesiedelt. Für gewöhnlich, und im Folgenden angenommen, sieht und verwaltet er (virtuelle) Domänen. Innerhalb der Domänen sind Tasks untergebracht, die durch einen Scheduler innerhalb der Domänen verwaltet werden. Im speziellen Fall der HRT-Applikationen besitzt eine Domäne genau eine Task.

Für die Domänen in den System- und HRT-Klassen werden folgende Parameter festgelegt: eine feste Periode p, mit der ein Ereignis erwartet wird, die Worst-Case Rechenzeit c, die innerhalb einer Periode verbraucht werden darf und eine relative Deadline d, innerhalb der eine Reaktion auf das Ereignis erwartet wird. Zusätzlich hilft ein Parameter $p_{\max}$, der den maximal erlaubten Abstand zwischen zwei Aktivierungen angibt, um auch auf ausbleibende Stimuli reagieren zu können. Die Verletzung eines Parameters führt zu einer Fehlermeldung und zur Unterbrechung der laufenden Domäne. Im Folgenden wird sowohl ein Scheduling mit festen Prioritäten als auch ein Deadline Scheduling (earliest deadline first – EDF) untersucht.

Für die Klassen SRT und NRT reicht bei heutigen Systemen i. d. R. eine Beschränkung auf einen prozentualen Anteil der Rechenleistung aus. Das Scheduling erfolgt dann z. B. über die faire Zuteilung des Prozessors zu allen Domänen gemäß vorbestimmter Gewichte.

3.3.2 Paketverarbeitung durch eine Treiberdomäne

Die Netzwerkkarte ist dem Kontext einer bestimmten Domäne, im Folgenden als Treiberdomäne bezeichnet, zugeordnet. Kommt ein Paket an der Netzwerkkarte an, löst diese einen Interrupt und damit einen Sprung in den Hypervisor aus. Dort wird der Interrupt erkannt und ein Ereignis-Flag für die Treiberdomäne gesetzt. Anschließend wird ein Rescheduling durchgeführt. Sobald die Treiberdomäne das nächste Mal den Prozessor zugeteilt bekommt, analysiert sie das Paket und entscheidet anhand der Zieladresse, an welche virtuelle Domäne es weitergeleitet werden muss. Dies teilt sie dem Hypervisor über einen Systemaufruf (Hypercall) mit. Der Hypervisor setzt seinerseits wieder ein Ereignis-Flag für die Zieldomäne und führt ein Rescheduling durch. Erst dann erreicht das Paket seinen Bestimmungsort. Die Verarbeitung von ausgehenden Paketen erfolgt in umgekehrter Richtung. Hier kommt zusätzlich noch eine Bestätigungssignalisierung durch die Netzwerkkarte hinzu, die wieder über die Treiberdomäne zur versendenden Domäne bzw. Task geleitet wird. Es ist gut zu erkennen, dass das Scheduling insbesondere der Treiberdomäne einen wesentlichen Einfluss auf die Latenz der Paketverarbeitung hat.

Zusätzlich sollte in der Treiberdomäne eine Überwachung der Paketraten durchgeführt werden, um bei Fehlverhalten eine entsprechende Fehlerbehandlung einzuleiten.

Im Folgenden wird angenommen, dass die maximale Rechenzeit der Treiberdomäne c_N maßgeblich durch die Einsortierung der Pakete in die Ausgangswarteschlange bestimmt ist. Erst wenn alle Pakete einsortiert sind, übermittelt die Treiberdomäne das höchstpriore Paket an die Netzwerkkarte. c_{Na} ist die Rechenzeit, die von der Treiberdomäne für die Bearbeitung der Bestätigung benötigt wird. Weiter wird angenommen, dass die maximale Rechenzeit der Treiberdomäne für eingehende Pakete gleich der der ausgehenden Pakete ist. Der Scheduling-Overhead O_v wird jeweils für die Aktivierung und Deaktivierung einer Domäne einmal angesetzt.

Für sich gesehen gilt für das Versenden eines Pakets:

$$t_N^{out} = c_N + c_{Na} + 4O_v \,, \tag{3.7}$$

und für das Empfangen eines Pakets:

$$t_N^{in} = c_N + 2O_v \,. \tag{3.8}$$

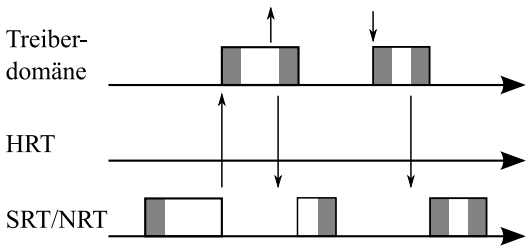


Bild 3.3: Prinzipieller Sendevorgang

Bild 3.3 zeigt den prinzipiellen Ablauf beim Senden einer Nachricht. Der zu berücksichtigende Overhead durch Kontextwechsel ist jeweils grau markiert.

Die dreistufige Paketverarbeitung erfordert es, dass drei innere Deadlines pro Recheneinheit und eine Ende-zu-Ende Deadline für jede Task einer Wirkkette vergeben werden. Hochgestellte Indizes definieren die Zugehörigkeit wie folgt: *in* für eingehende Paketverarbeitung, *out* für ausgehende Paketverarbeitung und *c* für die Phase der eigentlichen Berechnung. Die Deadline kann demnach immer kleiner als die Periode angenommen werden, so dass im Folgenden statt $\min(d_i, p_i)$ immer d_i geschrieben werden kann.

Des Weiteren wird die Zeit für die Verarbeitung der Bestätigung c_{Na} in die Berechnungszeit c_N integriert und die eventuell zusätzlich benötigten Kontextwechsel $2O_v$ vernachlässigt. Weiter wird angenommen, dass die Echtzeitdomäne erst mit dem nächsten für sie entreffenden Paket über das erfolgreiche Versenden ihres letzten Pakets benachrichtigt wird, so dass auch hier der Kontextwechsel entfallen und die Berechnungszeit c_{ia} in c_i integriert werden kann.

3.3.3 Scheduling mit festen Prioritäten

In vielen Systemen wird ein Scheduling mit festen Prioritäten eingesetzt. Im Folgenden wird das Verhalten der Paketverarbeitung auf Single- und Multicore-Systemen analysiert und bewertet.

Singlecore

Für eine Recheneinheit mit nur einem Core muss die Treiberdomäne eine Priorität besitzen, die höher ist als die der HRT-Klassen. Alle ein- und ausgehenden Pakete laufen über die Treiberdomäne, so dass durch eine hohe Paketrate aus den Klassen mit niedrigerer Priorität oder sogar aus der third-party Kommunikation die Möglichkeit einer Denial-of-Service Attacke besteht.

Für eingehende Pakete muss demnach bereits das Kommunikationsnetz dafür Sorge tragen, dass nur die spezifizierte Anzahl an Paketen pro Zeiteinheit an die Recheneinheit weitergegeben wird. Die relative Belastung des Prozessors durch die Treiberdomäne ergibt sich für eingehende Pakete in Abhängigkeit des minimal erlaubten Paketabstands $p_{\min}$ inklusive Scheduling-Overhead O_v und der maximalen Rechenzeit der Treiberdomäne c_N zu:

$$u_N = \frac{c_N + 2O_v}{p_{\min}} \tag{3.9}$$

Für den Fall, dass Paket-Bursts, also n Pakete direkt hintereinander, zugelassen werden, verzögert sich die Bearbeitung eines Pakets im Worst-Case um

$$\varepsilon_{\text{SC}}^{-} = (n-1) \cdot c_N \, . \tag{3.10}$$

Die Rate von ausgehenden Paketen kann nicht beschränkt werden. Dies ist auch nicht notwendig, da eine Domäne nur dann Pakete verschicken kann, wenn sie den Prozessor zugeteilt

bekommen hat. Es kann also genau ein Paket zum Versand bereitstehen, während ein Paket für eine höherpriore Domäne eintrifft. Das ausgehende Paket verzögert die Weiterleitung des eingehenden Pakets um c_N und unterbricht im schlechtesten Fall die höherpriore Domäne einmal, um die Bestätigung von der Netzwerkkarte entgegen zu nehmen.

Insgesamt muss folgende Verzögerung einbezogen werden:

$$\varepsilon_{\text{SC}}^{\text{FP}} = n \cdot c_N + c_{Na} + O_v \tag{3.11}$$

Der Overhead muss hier nur einmal berücksichtigt werden: Zwar kommen zwei zusätzliche Kontextwechsel für die Bestätigungsmeldung der Netzwerkkarte hinzu ($+2O_v$), da die Treiberdomäne aber bereits läuft, wenn das nächste Paket eintrifft, entfällt der sonst notwendige Kontextwechsel in die Treiberdomäne ($-O_v$). Bild 3.4 zeigt den Ablauf.

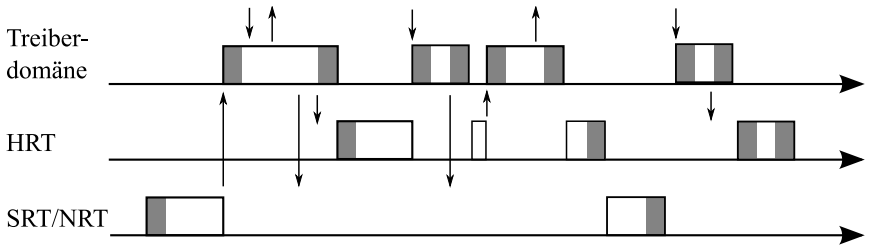


Bild 3.4: Beeinflussung eines eingehenden Pakets durch ein ausgehendes Paket (Singlecore)

Multicore

Bei m Cores können nun auch m Anfragen – von jedem Core eine – aus beliebigen Prioritäts-Klassen für ausgehende Pakete gleichzeitig vorliegen. Vor dem Senden findet eine Sortierung der Sendeanfragen nach Prioritäten analog zu Abschnitt 2.3.4 statt. Da beim Eintritt in die Treiberdomäne weder die Priorität eines eingehenden noch der ausgehenden Pakete bekannt ist, müssen alle Pakete gleich behandelt werden. Das einfachste Verfahren hierfür ist FCFS (First Come First Served). Im Folgenden wird zudem davon ausgegangen, dass

- keine Bestätigungsnachricht von vorhergehenden Übertragungen aussteht,
- die aktuellen Bestätigungsnachrichten erst nach der Übertragung aller Pakete erfolgen und somit keinen Einfluss auf die Weiterleitung der Pakete haben,
- während der Bearbeitungszeit maximal n Pakete von außen ankommen und
- alle $m + n$ Pakete versendet wurden, bevor ein neues Paket zur Weiterleitung eintrifft.

Trifft ein weiteres Paket ein, bevor die Sortierung abgeschlossen ist, kann es zu Deadline-Verletzungen kommen, wie Bild 3.5 zeigt: Eine Task T_1 sendet Pakete periodisch mit p_1. Die Deadline entspricht der Periode. Gleichzeitig oder nacheinander treten weitere Sendewünsche

von anderen Tasks bzw. Domänen auf (T_2 – T_4). Die grauen Kästen auf der Zeitachse repräsentieren die Verarbeitungszeiten c_N für die Sortierung der einzelnen Pakete nach Prioritäten. In diesem Fall liegt ein Designfehler vor. Ohne weitere Mechanismen führt nur die Erhöhung der Prozessorleistung oder Verringerung der erlaubten Rate zur Entschärfung.

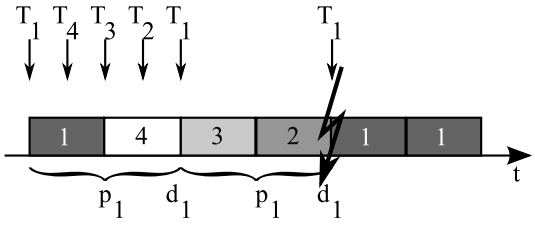


Bild 3.5: Deadlineverletzung in Treiberdomäne

Die Beeinflussung ist am geringsten, wenn die Domänen aus den System- und HRT-Klassen auf anderen Cores laufen als die Treiberdomäne. In diesem Fall können bereits weitergeleitete Pakete von den hochprioren Tasks auf anderen Cores bearbeitet werden, während die Treiberdomäne noch die übrigen Pakete abarbeitet. Das bedeutet auch, dass O_v für die Treiberdomäne nur noch einmal pro Aktivierung in die Berechnung einfließt. Es stellt sich die Frage, wie viele Cores der Treiberdomäne zur Verfügung gestellt werden sollen. Dies wird im Folgenden untersucht.

Als Worst-Case wird angenommen, dass alle Nachrichten gleichzeitig ankommen. Für diesen Fall entfällt der Overhead, da die Treiberdomäne nach der ersten Nachricht bereits aktiv ist und dies auch bleibt, bis alle Nachrichten abgearbeitet sind.

Fall 1: Die Treiberdomäne läuft auf einem Core. Sie erhält die höchste Priorität. Mit obiger Annahme, dass die Bearbeitung aller $m + n$ Anfragen abgeschlossen ist, bevor neue Pakete gesendet werden sollen oder in das System kommen, verzögert sich die Weiterleitung eines eingehenden Pakets um maximal

$$\varepsilon^{\text{FP}}_{\text{MC1}} = (m + n - 1) \cdot c_N \; . \tag{3.12}$$

Falls $n \ll m$ verbessert sich das Delay für die eingehenden Pakete wesentlich, wenn diese bevorzugt behandelt werden, wohingegen der Worst-Case für ausgehende Pakete gleich bleibt.

Fall 2: Die Treiberdomäne bedient sich mehrerer Cores. Die Aufgaben der Treiberdomäne können dabei auf verschiedene Cores verteilt werden. Hier erscheinen zwei Cores (einen für ausgehende und einen für eingehende Pakete) als sinnvoll und ausreichend, da sich die Parallelisierung der Sortierung erst bei großen m und hohen Raten lohnt.

Bei angenommener vollständiger Parallelisierbarkeit, lässt sich die zusätzliche Verzögerung für ausgehende Pakete auf

$$\varepsilon = (m - 1) \cdot c_N$$

und für eingehende auf

$$\varepsilon_{\text{MC2}}^{\text{FP}} = (n-1) \cdot c_N \tag{3.13}$$

verbessern.

Wegen der Forderung, dass die Treiberdomäne nicht auf den gleichen Cores laufen darf wie die Echtzeit-Domänen, ist die Zuteilung von mehreren Cores zur Treiberdomäne erst ab drei Cores sinnvoll.

Im Weiteren wird ein Multicore System vorausgesetzt.

Verbesserung des Zeitverhaltens

Um Pakete aus den System- und HRT-Klassen vorrangig behandeln zu können, wird vorgeschlagen, den Signalisierungsmechanismus, mit dem die Domänen den Weiterleitungswunsch eines Pakets anmelden, wie folgt zu erweitern. Die Priorität einer Echtzeit-Domäne dient gleichzeitig der Priorisierung der Nachrichten. Der Hypervisor annotiert dazu die Ereignisse mit der Priorität der Sende-Domäne innerhalb des Hypercalls. Hierzu muss ggf. der Signalisierungsmechanismus des Microkernels adaptiert werden, falls dieser keine Zusatzinformationen zulässt. Die Treiberdomäne behandelt nun die Ereignisse gemäß der annotierten Priorität.

Meist ist die Bearbeitung eines Ereignisses nicht unterbrechbar. Mit B wird die Blockierungszeit angegeben, die im Worst-Case durch die Bearbeitung von Paketen der Prioritätsklasse QoS oder niedriger, oder durch Kontextwechsel auftreten kann.

Der Realisierbarkeits-Nachweis nach [60] für ein Paket der Priorität i wird auf das nichtpreemptive System angepasst. Analog zu Abschnitt 3.2 wird eine Näherung angegeben.

Für **Multicore/Fall 1** wird wieder angenommen, dass eingehende Pakete bevorzugt behandelt werden. Der minimale Abstand zwischen zwei eingehenden Paketen ist wieder mit $p_{\min}$ angegeben. Die maximale Verzögerung eines ausgehenden Pakets ergibt sich zu:

$$t^k = B + \underbrace{\sum_{j=1}^{i-1} \left\lceil \frac{t^{k-1}}{p_j} \right\rceil \cdot c_N}_{\text{ausgehend}} + \underbrace{\left\lceil \frac{t^{k-1}}{p_{\min}} \right\rceil \cdot c_N}_{\text{eingehend}} \tag{3.14}$$

Die Blockierungszeit B lässt sich auf $2O_v$ beschränken: Da der Worst-Case angenommen wird, treffen alle Nachrichten zu Beginn ein. Da $\forall i \; d_i \leq p_i$ gilt, muss zu diesem Zeitpunkt die Abarbeitung aller früheren Nachrichten bereits beendet sein, andernfalls würde das Abbruchkriterium nie erreicht. Im schlechtesten Fall wurde gerade ein Kontextwechsel aus der Treiberdomäne (z. B. in den Idle-Zustand) eingeleitet, so dass dieser und der Kontextwechsel wieder zurück in die Treiberdomäne als Blockierungszeit anfallen können.

Die eingehenden Pakete können als höchstpriore Pakete angesehen werden und mit $p_0 = p_{\min}$ in die Summe aufgenommen werden:

$$t^k = \sum_{j=0}^{i-1} \left\lceil \frac{t^{k-1}}{p_j} \right\rceil \cdot c_N + 2O_v \tag{3.15}$$

Sobald $t^k = t^{k-1}$ ist die maximale Verzögerung gefunden. Die Deadline d_i^{out} muss demnach größer sein bzw. gewählt werden, als die Summe aus maximaler Verzögerung und eigener Paketbearbeitung:

$$d_i^{\text{out}} \geq t^k + c_N \tag{3.16}$$

In Näherung:

$$d_i^{\text{out}} \geq \frac{i \cdot c_N + 2O_v}{1 - \sum_{j=0}^{i-1} \frac{c_N}{p_j}} + c_N \tag{3.17}$$

Der relevante Scheduling-Fehler für eingehende Pakete ist nur bestimmt durch die Bearbeitungszeit genau eines ausgehenden oder eingehenden Pakets: Dass $p_{\min} > c_N$ sein muss, ist klar ersichtlich, sonst wäre das Abbruchkriterium nie erfüllt. Das bedeutet, falls die Bearbeitung eines ausgehenden Pakets gerade begonnen hat, blockiert es die Bearbeitung des eingehenden um c_N. Nach c_N wird die Bearbeitung des wartenden eingehenden Pakets begonnen. Nachdem $p_{\min} > c_N$ ist, trifft das nächste Paket erst kurz danach ein und muss wiederum c_N warten, bis es seinerseits bearbeitet wird, usw.

$$\varepsilon_{\text{MC1a}}^{\text{FP}} = c_N \, . \tag{3.18}$$

Für **Multicore/Fall 2** kann die Verarbeitung von ausgehenden und eingehenden Paketen getrennt voneinander betrachtet werden.

Für ausgehende Pakete gilt ohne Näherung:

$$t^k = B + \sum_{j=1}^{i-1} \left\lceil \frac{t^{k-1}}{p_j} \right\rceil \cdot c_N \tag{3.19}$$

Die Blockierungszeit B kommt durch die Bearbeitung genau eines niederprioren Pakets zustande: $B = c_N$.

Analog zu Gleichung 3.16 gilt für die minimale Deadline:

$$d_i^{\text{out}} \geq t^k + c_N$$

In Näherung ergibt sich:

$$d_i^{\text{out}} \geq \frac{B + (i-1)c_N}{1 - \sum_{j=1}^{i-1} \frac{c_N}{p_j}} + c_N \tag{3.20}$$

Eingehende Pakete sind durch das RT-Protokoll in den Switches bereits nach Prioritäten sortiert. Es kann höchstens ein Paket mit niedrigerer Priorität vorangestellt sein. In Realität können die

Pakete nur nacheinander eintreffen, im Folgenden wird als Näherung jedoch ein gleichzeitiges Eintreffen angenommen. Die Beschränkung auf eine minimale Ankunftszeit $p_{\min}$ wird aufgehoben und durch die tatsächlichen Perioden der HRT-Pakete ersetzt. Das Modell erlaubt nun auch gleichzeitig eintreffende Nachrichten. Im Mittel ist die Beschränkung $\overline{p^{\text{in}}} > c_N$ jedoch nach wie vor gültig.

Es kann nun ebenfalls Gleichung 3.19 bzw. 3.20 angesetzt werden. Allerdings lässt sich B nicht bestimmen. Da die Priorität erst bei der Verarbeitung selbst feststeht, kann keine Priorisierung der bereits eingegangenen und auf Bearbeitung wartenden Pakete vorgenommen werden. Falls keine Pakete zur Bearbeitung vorliegen, gilt $B = c_N$ durch das eventuell vorangestellte niederpriore Paket aus dem Kommunikationsnetz. Wenn jedoch noch x Pakete auf die Abarbeitung warten, ist $B = (x + 1)c_N$ zu setzen.

Für den Scheduling-Fehler ergibt sich demnach:

$$\varepsilon^{\text{FP}}_{\text{MC2a}} = B + \sum_{j=1}^{i-1} \left\lceil \frac{d_i^{\text{out}} - c_N}{p_j} \right\rceil \cdot c_N \tag{3.21}$$

Es ist zu erkennen, dass sich die Wartezeit in einer Recheneinheit von Fall 2 im Vergleich zu Fall 1 sogar verlängert. Dies liegt daran, dass Pakete nun auch gleichzeitig eintreffen dürfen. Im Fall 1 sind diese Verzögerungszeiten allerdings lediglich aus den Recheneinheiten in die Switches verlagert.

Für Fall 1 lässt sich die Beschränkung $p_{\min}$ nicht aufheben, da nicht garantiert werden kann, dass ausschließlich hochpriore Nachrichten ankommen: Sobald die Bearbeitung der hochprioren eingehenden Pakete beendet ist, werden die ausgehenden Pakete bearbeitet. Während dieser Zeit ist das Kommunikationsnetz nicht belegt und somit für niederpriore Nachrichten freigegeben, die an der Recheneinheit ankommen und die Bearbeitung der ausgehenden Pakete unterbrechen können. Diese niederprioren Nachrichten sind aber in der Formel nicht berücksichtigt.

3.3.4 Deadline-Scheduling

Die Prioritäten der einzelnen Domänen werden nun dynamisch in Abhängigkeit ihrer nächsten Deadline vergeben. Für das vorliegende System wurde nur das *earliest deadline first* (EDF) Scheduling-Verfahren untersucht. Andere Deadline Scheduling-Verfahren, wie z. B. *least laxity first* (LLF) oder *earliest deadline last* (EDL) sind entweder nicht praxistauglich – im Fall von LLF werden beispielsweise bei anfänglich gleicher Laxity unendlich viele Kontextwechsel erzeugt – oder nicht für das System geeignet wie im Fall von EDL, bei dem von exakt periodisch auftretenden Ereignissen ausgegangen wird; eine geringe Verzögerung eines Auslösezeitpunkts hat eine Deadlineverletzung zur Folge. EDL kann z. B. für ein FlexRay-basiertes System eingesetzt werden, bei dem der Auslösezeitpunkt exakt periodisch erfolgt.

Da der Unterschied durch Kontextwechsel zwischen Single- und Multicore Systemen gering ist, werden die Overheads nur in der ersten Gleichung exemplarisch angegeben. Im Weiteren werden sie in die Worst-Case Rechenzeiten eingerechnet. Die Modellierung erfolgt allgemeingültig

für Single- und Multicore Systeme. Für den Echtzeitnachweis wird von Multicore Systemen ausgegangen, wobei die Treiberdomäne auf einem anderen Core alloziert ist als die Echtzeit-Domänen. Tasks bzw. Domänen mit weichen oder ohne Echtzeitanforderungen werden nicht in die Betrachtung einbezogen. Die Zuteilung dieser erfolgt in den Idle-Zeiten der Prozessoren.

Modellierung

Unter obiger Vereinfachung, dass die Abwicklung der Bestätigungsmeldung durch die Netzwerkkarte in c_N eingerechnet wird, ergibt sich der Task-Abhängigkeitsgraph (TAG) gemäß Bild 3.6.

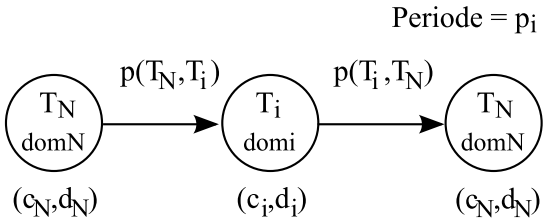


Bild 3.6: Task-Abhängigkeitsgraph (TAG) für eine Recheneinheit

Es wird eine Notation in Anlehnung an [10] verwendet: Eine Task T_j wird durch ein Tupel (c_j, d_j) charakterisiert; c_j ist die WCET, und d_j ist die relative Deadline der Task T_j. Die darauffolgende Task T_{j+1} wird frühestens $p(T_j, T_{j+1})$ nach der Aktivierung von T_j aktiviert. Im vorliegenden System gibt es für die Echtzeit-Tasks einen Gültigkeitszeitpunkt, in den die jeweiligen Scheduling-Fehler der beteiligten Domäne eingehen (Voraussetzung für die Synchronität des Systems). Das bedeutet, dass die Folge-Task frühestens nach Ablauf der Deadline der Vorläufer-Task aktiviert wird: $p(T_N, T_i) = d_N$ bzw. $p(T_i, T_N) = d_i$. Für die Quell-Task (entspricht dem Quellknoten des Graphen) gilt, dass die Aktivierung periodisch mit p_i erfolgt. In Bild 3.7 ist die kummulative Rechenzeitanforderung über der Zeit dargestellt. Sie entsteht, wenn ein Paket für Task T_i eintrifft, entspricht aber nicht dem Worst-Case.

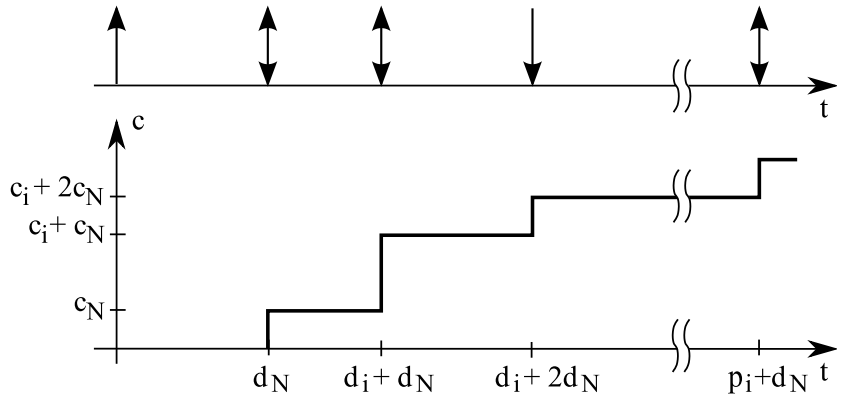


Bild 3.7: Rechenzeitanforderung über der Zeit

Für die Worst-Case Betrachtung bieten sich verschiedene Methoden an. Die Modellierung nach Liu/Layland [59] ist sehr pessimistisch, da hiernach alle drei Tasks synchron, d. h. gleichzeitig, ausgelöst werden müssen. Die RZAF nach [59], im Folgenden als RZAF_1 bezeichnet, ergibt sich für eine Task T_i zu:

$$E_i(I) = 2 \cdot \left(\left\lfloor \frac{I - d_N}{p_i} \right\rfloor + 1 \right) \cdot c_N + \left(\left\lfloor \frac{I - d_i}{p_i} \right\rfloor + 1 \right) \cdot c_i \; . \tag{3.22}$$

Die Modellierung nach Gresser [48] lässt eine Präzedenzbeziehung zwischen Tasks zu. Jedoch werden diese aufgelöst und wie synchrone Tasks behandelt. Dadurch ergibt sich ebenfalls Gleichung 3.22.

Da die Treiberdomäne (T_N) zweimal aufgerufen wird, lässt sich gegenüber [59] und [48] eine Verbesserung dadurch erzielen, dass die zweite Aktivierung der Treiberdomäne mit einer Phase in die RZAF eingebracht wird: $\Phi_N = \min(d_i, p_i - (d_N + d_i))$. Der rechte Teil gibt den Abstand zwischen der Aktivierung der Treiberdomäne durch das ausgehende Paket von T_i und der Aktivierung durch das nächste eingehende Paket für T_i an. Die resultierende RZAF, im Folgenden als RZAF_2 bezeichnet, (Bild 3.8) ergibt sich zu:

$$\begin{aligned} E_i(I) \;\; = \;\; & \left(\left\lfloor \frac{I - d_N}{p_i} \right\rfloor + 1 \right) \cdot c_N + \left(\left\lfloor \frac{I - d_i}{p_i} \right\rfloor + 1 \right) \cdot c_i + \\ & + \left(\left\lfloor \frac{I - d_N - \Phi_N}{p_i} \right\rfloor + 1 \right) \cdot c_N \; . \end{aligned} \tag{3.23}$$

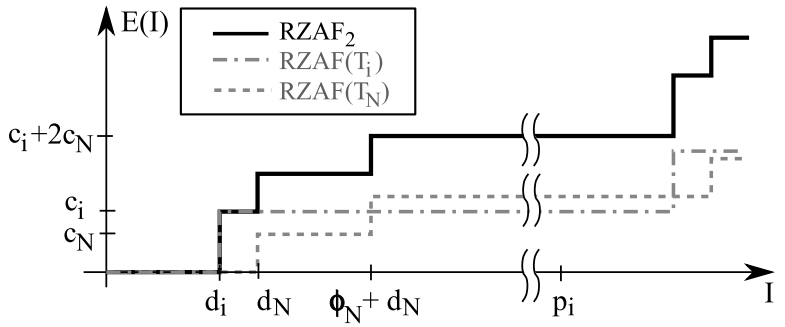


Bild 3.8: RZAF für eine Task inkl. Paketverarbeitung

In Anlehnung an die Modellierung durch „(generalized) multiframe tasks“ [73, 97, 9, 13] in Verbindung mit den sog. „recurring real-time tasks“ [12, 10] wird eine weitere Verbesserung wie folgt erzielt. Die resultierende RZAF wird mit RZAF_3 bezeichnet.

- Betrachtet werden Intervalle I der Größe $0 \leq I \leq p_i$; dies ist wegen $d_i + 2d_N \leq p_i$ und der Periodizität mit p_i ausreichend. Es werden nur die Tasks betrachtet, deren Aktivierung sowie Deadline in diesem Intervall enthalten sind.

- Daraus werden Tupel (c_I, I) gebildet, wobei c_I die akkumulierte Rechenzeit aller betrachteten Tasks im Intervall I ist.
- Die (nach I sortierte) Liste wird reduziert, indem Tupel mit gleichem c_I entfernt werden.

Alternativ kann die Liste wie folgt erzeugt werden:

- Für jede Task im TAG bildet man das Tupel (c_j, d_j), mit $j \in \{N, i\}$.
- Für jede mögliche Taskfolge innerhalb einer Periode werden Tupel gebildet, indem Rechenzeiten und Deadlines jeweils aufsummiert werden. Beispielsweise ergibt die Sequenz $T_i \rightarrow T_N$ das Tupel $(c_i + c_N, d_i + d_N)$.
- Für periodenübergreifende Taskfolgen (Sequenzen, welche die darauffolgende Aktivierung durch ein neues Paket beinhalten) muss die Periode eingerechnet werden. Sequenzen enthalten einen Knoten des TAG höchstens einmal. Beispielsweise ergibt die Sequenz $T_N^0 \rightarrow T_N^1$ (entspricht zwei aufeinander folgenden Aktivierungen von T_N durch das ausgehende und das nächste eingehende Paket) das Tupel $(2c_N, d_N+(p_i-(2d_N+d_i)+d_N)) = (2c_N, p_i - d_i)$. Für den Grenzfall $p_i = 2d_N + d_i$: $(2c_N, 2d_N)$.
- Doubletten werden aus der Tupelliste entfernt. Aus obigen Punkten ergibt sich folgende unsortierte Liste. Um die Berechnung der Werte zu zeigen, sind die Deadlines ausführlich angegeben:
  - Einzelne Tasks: $[(c_N, d_N)(c_i, d_i)(c_N, d_N)]$
  - 2er Sequenzen: $[(c_i+c_N, d_i+d_N)(c_N+c_i, d_N+d_i)(2c_N, d_N+(p_i-(2d_N+d_i)+d_N))]$
  - 3er Sequenzen: $[(c_i + 2c_N, d_i + d_N + (p_i - (2d_N + d_i) + d_N))(c_N + c_i + c_N, d_N + d_i + d_N)(c_N + c_i + c_N, p_i)]$

  Nach dem Entfernen der mehrfach vorhandenen Tupel ergibt sich die (unsortierte) Liste: $[(c_N, d_N)(c_i, d_i)(c_i + c_N, d_i + d_N)(2c_N, p_i - d_i)(c_i + 2c_N, d_i + 2d_N)(c_N + c_i + c_N, p_i)]$
- Die Tupelliste wird nach aufsteigenden Deadlines bzw. bei gleichen Deadlines nach absteigenden Rechenzeiten sortiert.
- Die sortierte Liste wird reduziert, indem die Tupel entfernt werden, deren Rechenzeiten kleiner sind als eine beliebige Rechenzeit mit kürzerer Deadline.

Die zweite Variante lässt sich performanter implementieren, da die Tupel exakt für die Intervallgrößen I erzeugt werden, an denen sich die Rechenanforderung ändert. Bei der ersten Variante muss der Bereich $0 \leq I \leq p_i$ in beliebig feingranularer Auflösung untersucht werden.

Die entstandene Liste ist die tabellarische Darstellung der RZAF.

Beispiel:

Das bisher verwendete Beispiel besitzt folgende Randbedingungen: $d_i < d_N$, $c_i > c_N$ und

$p_i > 2d_i$. Die resultierende Tupelliste ist $[(c_i, d_i)(c_i + c_N, d_i + d_N)(c_i + 2c_N, d_i + 2d_N)]$ und ist damit identisch mit der RZAF aus Gleichung 3.23 bzw. Bild 3.8.

Beispiel:

Folgendes Beispiel verdeutlicht die Verbesserung in den einzelnen Varianten. Untersucht werden die Berechnungen nach Liu/Layland ($RZAF_1$), die verbesserte Modellierung nach Gresser ($RZAF_2$) und die Variante in Anlehnung an die Modellierung der „(generalized) multiframe tasks“ ($RZAF_3$).

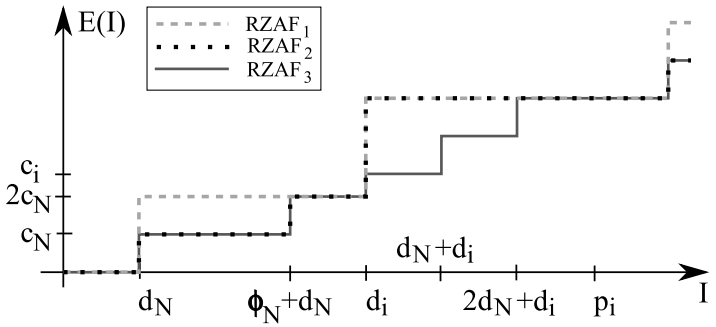
Folgende Randbedingungen gelten: $d_i > d_N$, $c_i > 2c_N$ und $p_i < 2d_i$. $RZAF_1$ ergibt sich selbsterklärend. Für die $RZAF_2$ muss die Phase festgelegt werden: $\Phi_N = p_i - (d_N + d_i)$, d. h. $p_i - (d_N + d_i) < d_i$, da $p_i \overset{def}{<} 2d_i < d_N + 2d_i$:

$$\begin{aligned} E_i(I) &= \left(\left\lfloor \frac{I - d_N}{p_i} \right\rfloor + 1\right) \cdot c_N + \left(\left\lfloor \frac{I - d_i}{p_i} \right\rfloor + 1\right) \cdot c_i + \\ &\quad + \left(\left\lfloor \frac{I - p_i + d_i}{p_i} \right\rfloor + 1\right) \cdot c_N \end{aligned}$$

Für die $RZAF_3$ wird die Tupellisten zunächst nach Variante 1 aufgestellt:

$$\begin{aligned} I_1 &= d_N & : & \ (c_N, d_N) \\ I_2 &= p_i - d_i & : & \ (2c_N, p_i - d_i) \\ I_3 &= d_i & : & \ (c_i, d_i) \\ I_4 &= d_i + d_N & : & \ (c_i + c_N, d_i + d_N) \\ I_5 &= d_i + 2d_N & : & \ (c_i + 2c_N, d_i + 2d_N) \\ I_6 &= p_i & : & \ (c_i + 2c_N, p_i) \end{aligned}$$

Die identische Tupelliste ergibt sich nach Variante 2 durch Sortierung der obigen Liste:
$d_N < p_i - d_i$: immer, da $p_i - d_i \geq 2d_N$.
$p_i - d_i < d_i$: gemäß Randbedingung ($p_i < 2d_i$).
Die restliche Sortierung ergibt sich selbstverständlich:
$[(c_N, d_N)(2c_N, p_i - d_i)(c_i, d_i)(c_i + c_N, d_i + d_N)(c_i + 2c_N, d_i + 2d_N)(c_i + 2c_N, p_i)]$



RZAF der drei Modellierungsvarianten.

Bestimmung der Deadlines

Die Treiberdomäne ist für die Bearbeitung sämtlicher ein- und ausgehender Nachrichten zuständig. Daher muss ihre Deadline so gewählt werden, dass die Applikation mit den strengsten Zeitbedingungen ihre (Ende-zu-Ende) Deadline einhalten kann. Die maximal erlaubte Reaktionszeit einer Task T_i für eine Recheneinheit ist $t_{r,i}$ und wird im Folgenden als bekannt vorausgesetzt. Des Weiteren gilt $t_{r,i} < p_i$. Während $t_{r,i}$ wird die Treiberdomäne zweimal aktiviert, die Domäne i einmal. Die untere Schranke, für die das System die Task T_i gerade noch bedienen kann ist

$$t_{r,i} \geq 2c_N + c_i + x \cdot O_v \; . \tag{3.24}$$

Im Worst-Case ist zum Zeitpunkt des Eintreffens eines Pakets gerade ein Kontextwechsel in eine andere als die Treiberdomäne eingeleitet worden. Dieser und der nachfolgend notwendige müssen in die Bestimmung der Deadline für die Treiberdomäne berücksichtigt werden. Des Weiteren wird für Singlecore-Systeme angenommen, dass der Rechenanteil von T_i, der nach dem Versenden des Pakets anfällt, in die nächste Aktivierung von T_i verschoben wird und in c_i enthalten ist. Damit ergeben sich die Deadlines für ein Singlecore-System zu $d_N = c_N + 2O_v$ und $d_i = c_i + O_v$. In Multicore-Systemen kann der Rechenanteil von T_i, der nach dem Versenden des Pakets anfällt, parallel zur Treiberdomäne bearbeitet werden, wodurch sich $t_{r,i}$ um diese Zeit verkürzt. Im Modell wird dies jedoch nicht berücksichtigt, so dass obige Berechnung auch für Multicore-Systeme gültig ist.

Es ist klar ersichtlich, dass das System mit dieser einen Task ausgelastet ist, wenn nicht garantiert werden kann, dass die Treiberdomäne beim Eintreffen der nächsten Echtzeitnachricht Idle ist.

Im Gegensatz zum vorherigen Abschnitt können die exakten Gleichungen nicht angewandt werden. Beim Einsatz von Deadline-Scheduling muss daher wieder auf die Beschränkung auf n Pakete, die quasi gleichzeitig an der Treiberdomäne eintreffen dürfen, zurückgegriffen werden (Gleichung 3.10). Die Deadline hängt demnach von der kürzesten Reaktionszeit aller Tasks auf der Recheneinheit $\min_{\forall i} t_{r,i}$ und von der Anzahl der blockierenden Verarbeitungen eingehender und ausgehender Pakete ab. Der Overhead wird im Folgenden wieder als in die Verarbeitungszeiten eingerechnet angenommen.

$$n \cdot c_N \leq d_N \leq \frac{t_{r,i} - d_i}{2} \;\text{, mit }\; i | \min_{\forall i} t_{r,i} \; . \tag{3.25}$$

Die obere Schranke leitet sich aus Gleichung 3.24 ab. Die untere Schranke lässt keinen Spielraum, d. h. nach Liu/Layland [59] wäre der Prozessor bereits voll ausgelastet und würde keine weitere Task erlauben. Die zwei weiteren vorgestellten Varianten zur Berechnung der RZAF (RFAZ_2 und RFAZ_3) erlauben hingegen zusätzliche Tasks.

Der zu berücksichtigende maximale Scheduling-Fehler ist abhängig von der Deadline der Treiberdomäne:

$$\varepsilon^{\text{EDF}}_{\text{MC2a}} = d_N - c_N \; . \tag{3.26}$$

Echtzeitnachweis

Im Folgenden wird von Dualcore Systemen ausgegangen, wobei die Treiberdomäne auf einem anderen Core alloziert ist als die Echtzeit-Domänen. Der Nachweis wird für beide Cores separat durchgeführt. Eine Taskmigration zwischen Cores wird nicht betrachtet.

Auf dem Core, der die Treiberdomäne beherbergt, entfallen die Rechenanteile c_i, wobei die Phasen der Aktivierung der Treiberdomäne erhalten bleiben und von den jeweiligen Tasks T_i abhängen. Die Anzahl der Tasks, die sich auf einer Recheneinheit befinden (Taskset τ), ist i. d. R. größer als die Anzahl von Paketen, die von der Treiberdomäne gleichzeitig bearbeitet werden können. Daher wird die resultierende RZAF für die Treiberdomäne wie folgt abstrahiert:

$$E_N(I) = \left(\left\lfloor \frac{I - d_N}{n \cdot p_{\min}} \right\rfloor + 1 \right) \cdot n \cdot c_N \ . \tag{3.27}$$

Wobei n die Anzahl der gleichzeitig zur Verarbeitung in der Treiberdomäne erlaubten Pakete ist und $p_{\min}$ den minimal erlaubten Paketabstand wiedergibt, mit $p_{\min} \geq c_N$.

Auf dem anderen Core entfallen die Rechenanteile c_N und dadurch auch alle Phasen, d. h. auf diesem Core muss von einem synchronen Tasksystem ausgegangen werden.

$$E(I) = \sum_{\forall i | T_i \in \tau} \left(\left\lfloor \frac{I - d_i}{p_i} \right\rfloor + 1 \right) \cdot c_i \ .$$

Eine Entspannung der Rechenzeitanforderung im Worst-Case kann einerseits dadurch erzielt werden, dass die maximal erlaubte Paketankunftsrate und andererseits die bekannte Vorsortierung der ankommenden Pakete in die RZAF einfließt. Diese Möglichkeiten werden hier jedoch nicht weiter betrachtet.

XENs Simple Earliest Deadline First (SEDF) Scheduler

Im Forschungsprojekt IT_Motive 2020 wurden einige Teilaspekte der vorgestellten Architektur prototypisch umgesetzt. Auf Basis der freien Virtualisierungslösung XEN wurde eine Regelungsanwendung (Regelfrequenz 1 kHz) neben QoS-Anwendungen auf einer Recheneinheit betrieben. Um sicherzustellen, dass die Zeitbedingungen für die Regelung nicht verletzt werden, muss der SEDF Scheduler von XEN entsprechend parametrisiert werden. Dies ist im Folgenden dargestellt [81][69].

Die Regelung erhält Sensordaten über das Kommunikationsnetz, verarbeitet diese und schickt sie wiederum über das Kommunikationsnetz an einen Aktor. Die Paketannahme und -weiterleitung erfolgt über eine Treiberdomäne *domN*, der Regler ist in der virtuellen Domäne *domi* implementiert. Die folgende Betrachtung erfolgt nur für die Abläufe auf der Recheneinheit. Insbesondere interessiert die Berechnung der maximalen Verzögerung (Antwortzeit t_r) auf der Recheneinheit.

Der SEDF Scheduler ist ein Singlecore Scheduler, der die virtuellen Prozessoren der Domänen bedient. Die virtuellen Cores einer Domäne müssen vorab pysikalischen Cores zugewiesen werden; eine Verschiebung zur Laufzeit auf andere Cores findet nicht statt. Es erhält immer der virtuelle Core einer Domäne den Prozessor, dessen Deadline am nächsten liegt. Im Demonstrator wurde nur ein Core pro Domäne verwendet. Folgende sind die relevanten Parameter:

p^{SEDF} (Periode). Gibt das minimale Zeitintervall zwischen zwei Aktivierungen des virtuellen Prozessors und damit der Domäne an. Eine Periode beginnt, sobald der Status der Domäne auf lauffähig wechselt (z. B. durch Eintreffen eines Paketes für diese Domäne). Der Aktivierungszeitpunkt sei t_o. Die Deadline berechnet der Scheduler zu $d^{SEDF} = t_o + p^{SEDF}$. Die nächste Aktivierung der Domäne wird ebenfalls erst zum Zeitpunkt $t_o + p^{SEDF}$ zugelassen. Früher auftretende Ereignisse werden bis zu diesem Zeitpunkt verzögert.

s^{SEDF} (Slice). Gibt die maximale Laufzeit der Domäne während einer Periode p^{SEDF} an. Im Standard SEDF Scheduler gilt der Slice auch als aufgebraucht, wenn sich die Domäne in den Zustand blockiert bzw. wartend begibt.

Solange s^{SEDF} größer als die WCET der Domäne ist, verhält sich der SEDF Scheduler wie ein EDF Scheduler. Falls s^{SEDF} jedoch größer als die WCET ist, bzw. die Domäne mehr als s^{SEDF} Zeit für die aktuelle Berechnung benötigt, werden die Berechnungen in nachfolgende Perioden verschoben.

Diese Mechanismen müssen bei der Berechnung der maximalen Bearbeitungszeit berücksichtigt werden.

Zunächst wird die Verzögerung eines Pakets durch die Treiberdomäne bestimmt. Im schlechtesten Fall wird domN zu Beginn einer Periode aufgeweckt und beendet sich unmittelbar darauf selbst. Im selben Moment kommen k Pakete zur Bearbeitung für die domN an. Gemäß der SEDF Scheduling Strategie wird die Bearbeitung auf den Beginn der nächsten Periode verschoben. Angenommen, dass alle k Pakete innerhalb von s_N^{SEDF} bearbeitet werden können, ist die Bearbeitung am Ende der zweiten Periode nach Eintreffen des Pakets von der domN bearbeitet, d. h. weitergeleitet. Demnach ist die Verzögerung durch die domN in eingehender wie in ausgehender Richtung maximal $2p_N^{SEDF}$.

Während der maximalen Antwortzeit t_r muss die Periode der domN 4-mal und die Periode der domi 1-mal berücksichtigt werden (Bild 3.9).

Die maximale Periode der domN p_N^{SEDF} lässt sich wie folgt unter Berücksichtigung mehrerer virtueller Domänen mit Perioden p_i^{SEDF} ermitteln:

$$p_N^{SEDF} \leq [\min_{\forall RTdoms}(t_r) - p_i^{SEDF}] \cdot \frac{1}{4} \,. \tag{3.28}$$

Als nächstes wird s_N^{SEDF} und das maximale k ermittelt. Im schlechtesten Fall muss domN k Pakete innerhalb von s_N^{SEDF} bearbeiten. c_N sei die WCET der domN für die Bearbeitung des größtmöglichen Paketes. t_{cs} ist die Dauer eines Kontextwechsels, der hier berücksichtig werden muss.

$$k \cdot c_N + 2t_{cs} \leq s_N^{SEDF} \leq p_N^{SEDF} \,. \tag{3.29}$$

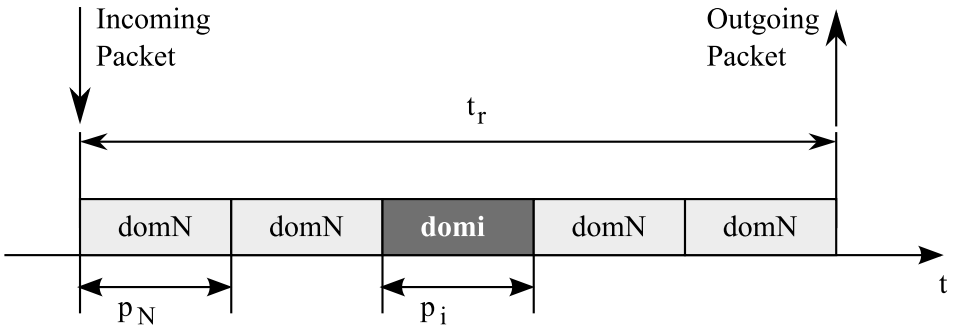


Bild 3.9: Aufteilung der Antwortzeit t_r

Das maximal mögliche k berechnet sich zu:

$$k = \left\lfloor \frac{1}{c_N} \left(p_N^{SEDF} - 2t_{cs} \right) \right\rfloor . \tag{3.30}$$

Der Parameter k kann nicht von der Recheneinheit beeinflusst werden, sondern muss durch das Kommunikationsnetz gewährleistet sein.

3.3.5 Support für fehlertolerante Datenübertragung

Den Redundanztyp der Nachricht erkennt die Treiberdomäne an der neu eingeführten Markierung des Pakets. Trifft ein derartig markiertes Paket ein, wird ein entsprechender Abgleich durchgeführt und erst danach entweder die Nachricht oder eine entsprechende Fehlermeldung an die Zieltask weitergeleitet.

An dieser Stelle gibt es zwei mögliche Umsetzungen:

- Die Nachricht wird wieder als Ethernet-Paket weitergeleitet.
- Die Nachricht wird ohne Ethernet-Header in eine einfache Warteschlange eingereiht und die Zieldomäne über das Vorliegen einer neuen Nachricht informiert.

Die zweite Lösungsmöglichkeit eignet sich besonders für den Einsatz bei den HRT-Applikationen. Da die Kommunikation direkt über Systemaufrufe erfolgt, benötigt die Zieldomäne keinen eigenen Ethernet-Stack. Hierdurch wird der Overhead, der durch die mehrstufige Paketverarbeitung entsteht, minimiert. Diese Variante wird im Weiteren für die fehlertolerante Datenübertragung angenommen.

Die zusätzliche Verarbeitungszeit für den Abgleich wird mit c_V angegeben. Weiterführende zeitliche Betrachtungen sind in Abschnitt 4.1 zu finden.

3.3.6 Anbindung über virtualisierte Netzwerkkarten

Virtualisierte Netzwerkkarten besitzen mehrere interne Paket-Puffer, die jeweils Domänen zugeordnet werden können. Dadurch entfällt die Bearbeitung der Pakete durch die Treiberdomäne. Stattdessen erfolgt das Scheduling sowohl von eingehenden als auch von ausgehenden Paketen in der Netzwerkkarte. Der Hypervisor muss nun lediglich das entsprechende Event für die Zieldomäne setzen. Derzeitige Implementierungen von virtuellen Netzwerkkarten besitzen einen Round-Robin Scheduler. Für das vorliegende System muss deshalb eine Anpassung analog zu Abschnitt 2.3.4 durchgeführt werden.

Als Nachteil kann angesehen werden, dass nun jede Domäne einen der Netzwerkkarte angepassten Treiber statt eines generischen virtuellen Treibers implementieren muss. Zusätzlich muss in jedem dieser Treiber die Unterstützung der fehlertoleranten Datenübertragung (falls in dieser Domäne benötigt) integriert werden. Da die Verarbeitung durch die Treiberdomäne komplett entfällt, reduziert sich die Latenz zwischen dem Eintreffen des Pakets an der Netzwerkkarte und dem Eintreffen des Pakets in der Zieldomäne. Dieser Aufwand sollte bei komplexeren Domänen in Betracht gezogen werden. Für Echtzeit-Domänen, die oft nur eine Funktion bereitstellen, ist der Overhead und Implementierungsaufwand von Netzwerktreiber und Ethernet-Stack unverhältnismäßig hoch.

3.3.7 Hybride Lösung der Kommunikationsnetzanbindung

Die Anbindung des Kommunikationsnetzes über eine Treiberdomäne bietet vor allem für schlanke Echtzeit-Domänen bzw. -Tasks den Vorteil, dass diese über eine einfache, effiziente API mit der Treiberdomäne kommunizieren können, ohne einen Ethernet-Stack implementieren zu müssen: Die eigentliche Anbindung an das Kommunikationsnetz übernimmt die Treiberdomäne und muss demnach nur einmal implementiert werden.

Für SRT- oder NRT-Domänen bieten hingegen die virtualisierten Netzwerkkarten wesentliche Vorteile. Oft ist bei diesen Domänen im Betriebssystem bereits ein Ethernet-Stack integriert und viele der aktuellen Standard-Anwendungen, wie z. B. Navigation oder Multimedia, bauen darauf auf. Es ist also im Sinne der Wiederverwendbarkeit, den Ethernet-Stack dort zu belassen.

Die Aufspaltung in die Verarbeitung von HRT-Paketen durch die Treiberdomäne einerseits und die Anbindung der restlichen Domänen an die virtuellen Netzwerkkarten andererseits wirkt sich vor allem bei der durch eingehende Pakete verursachten Verzögerung aus. Die resultierende Verzögerung kann damit genau bestimmt werden. Die Blockierungszeit B lässt sich auf $2O_v$ beschränken.

Zunächst erfolgt die Betrachtung des FP Schedulers.

Für **Multicore/Fall 1** folgt aus Gleichung 3.14:

$$t^k = \underbrace{\sum_{j=1}^{i-1} \left\lceil \frac{t^{k-1}}{p_j} \right\rceil \cdot c_N}_{\text{ausgehend}} + \underbrace{\sum_{j=1}^{i-1} \left\lceil \frac{t^{k-1}}{p_j} \right\rceil \cdot c_N}_{\text{eingehend}} + 2O_v$$

$$t^k = 2 \cdot \sum_{j=1}^{i-1} \left\lceil \frac{t^{k-1}}{p_j} \right\rceil \cdot c_N + 2O_v \tag{3.31}$$

In Näherung:

$$d_i \geq \frac{2(i-1)c_N + 2O_v}{1 - 2 \cdot \sum_{j=1}^{i-1} \frac{c_N}{p_j}} + c_N \tag{3.32}$$

Für **Multicore/Fall 2** kann die Blockierungszeit B nun ebenfalls auf $2O_v$ beschränkt werden. Damit lässt sich der maximale Scheduling-Fehler aus Gleichung 3.21 bestimmen:

$$\varepsilon_{\text{MC2b}}^{\text{FP}} = \sum_{j=1}^{i-1} \left\lceil \frac{d_i - c_N}{p_j} \right\rceil \cdot c_N + 2O_v \; . \tag{3.33}$$

Für EDF Scheduling bedeutet der hybride Ansatz, dass die gewonnene Prozessorleistung für weitere hochpriore Pakete verwendet werden kann, d. h. die Anzahl an zulässigen hochprioren Paketen, die gleichzeitig von der Treiberdomäne bearbeitet werden können, steigt. Formal bleibt der zu berücksichtigende Scheduling-Fehler unverändert:

$$\varepsilon_{\text{MC2b}}^{\text{EDF}} = d_N - c_N \; . \tag{3.34}$$

3.4 Sensoren und Aktoren

Sensoren und Aktoren unterliegen den gleichen Synchronisierungsmechanismen wie die Recheneinheiten. Falls Sensoren und Aktoren hinter einem Gateway liegen, sieht das System das Gateway als Sensor oder Aktor.

3.4.1 Sensoren

Sensoren sind Datenlieferanten und stehen damit immer am Anfang einer Wirkkette. Für gewöhnlich ist der Beobachtungszeitpunkt oder Aufnahme-Zeitstempel wichtig. Im vorliegenden System muss der Sensorwert zusätzlich mit einem Gültigkeits-Zeitstempel gemäß Abschnitt 3.1 versehen werden.

Aufnahme-Zeitstempel t_m. Jeder Sensor, der direkt am Kommunikationsnetz angeschlossen ist, besitzt eine exakte globale Zeit und liefert den Aufnahmezeitpunkt bei jedem Wert mit. Die Gateways sind dafür zuständig, die Sensorwerte der an ihnen angeschlossenen Sensoren mit diesem Zeitstempel zu versehen. Bei redundantem Anschluss an mehrere Gateways muss deren maximale Zeitdifferenz (η) bei der Auswertung berücksichtigt werden. Sind auch die Bussysteme redundant ausgelegt, kommen die Differenzen durch die unterschiedlichen Verzögerungen auf den Bussen noch hinzu.

Validitätszeitpunkt t_v. Der Zeitpunkt berechnet sich nach Gleichung 3.1 zu:

$$t_v = t_m + \eta_{\mathrm{S}} + \delta + c_c + \varepsilon + \eta_{\mathrm{RE}} \,.$$

bzw. falls die maximale Abweichung der Uhren in allen Knoten identisch ist zu:

$$t_v = t_m + \delta + c_c + \varepsilon + 2\eta \,. \tag{3.35}$$

t_m ist der Zeitstempel des Sensors bzw. des ersten Gateways, c_c ist die Zeit, die für die Bearbeitung des Pakets benötigt wird, ggf. zuzüglich weiterer Rechenzeiten, die vor dem Gültigkeitszeitpunkt liegen müssen.

Ebenfalls möglich ist die Angabe eines Zeitpunktes, nach dessen Überschreitung die Sensordaten als veraltet gelten und nicht mehr verwendet werden dürfen (Invaliditätszeitpunkt). Dieser kann für nicht-sicherheitskritische Funktionen hilfreich sein, wird jedoch für die hier betrachteten sicherheitskritischen Funktionen nicht verwendet.

Damit in unterschiedlichen Recheneinheiten die gleichen Daten gelesen werden, gibt es folgende Möglichkeiten für den Zugriffszeitpunkt.

Implizites Festlegen: Der Zugriffszeitpunkt wird auf den Aktivierungszeitpunkt der Task festgesetzt.

Explizites Festlegen: Die Task selbst legt einen Zugriffszeitpunkt fest, der nach dem Aktivierungszeitpunkt liegt. Die Cotasks müssen sich über diesen Zeitpunkt einig sein. Außerdem muss sichergestellt sein, dass der Zugriffszeitpunkt von der Cotask bereits erreicht ist; andernfalls muss sie warten.

Für beide Möglichkeiten gilt, dass die Task alle Sensorwerte, die zwar bereits vorliegen, aber erst nach diesem Zeitpunkt valide werden, ignoriert. Die verarbeitenden Tasks dürfen demnach nicht zwangsläufig auf den neuesten Daten rechnen[3)].

Das minimale Alter der Sensorwerte ist $a_{min} = t_v - t_m$; das maximale Alter hängt vom tatsächlichen Zugriffszeitpunkt auf die Daten ab, ist aber begrenzt durch die Periode p_S, mit der der Sensor seine Daten verschickt: $a_{max} = a_{min} + p_S$.

[3)] Ein Sensorwert ist erst valide, wenn er auf allen Systemen angekommen und verfügbar ist.

3.4.2 Aktoren

Aktoren sind Senken und stehen am Ende einer Wirkkette. Damit Stellwerte von unterschiedlichen Aktoren synchron ausgeführt werden können, sollte ein Ausführungszeitpunkt angegeben werden können. Fehlt diese Angabe, wird der Stellwert so schnell wie möglich, d. h. zum Gültigkeitszeitpunkt, ausgegeben (implizite Synchronität).

Für Aktoren, die über ein Gateway angeschlossen sind und selbst nicht über eine globale (synchrone) Zeit verfügen, leitet das Gateway den Stellwert zeitgerecht an den entsprechenden Aktor weiter. Handelt es sich um einen Stellwert, der aus einer höheren Taskklasse (> 1) stammt, findet ein Mehrheitsentscheid im Aktor bzw. im Gateway statt.

Bei den Aktoren ist es für höhere Taskklassen nicht notwendig, einen Invaliditätszeitpunkt zu definieren, da die Stellwerte zum Validitäts- bzw. Ausführzeitpunkt ausgegeben werden müssen. Lediglich für Taskklassen ohne redundante Verarbeitung erscheint die Angabe eines Invaliditätszeitpunkts sinnvoll.

4 Fehlerbehandlung

4.1 Fehlertoleranz durch Taskreplikation

Im vorliegenden System wird Taskreplikation eingesetzt, um Fehlertoleranz bereitzustellen. Bei Klasse-2-Tasks laufen zwei identische Cotasks auf unterschiedlichen Rechnern ab. Das Ergebnis wird votiert und bei Übereinstimmung verwendet. Klasse-3-Tasks besitzen drei identische Cotasks auf unterschiedlichen Rechnern. Ein Ergebnis wird als korrekt gewertet, wenn zwei von drei übereinstimmen (engl.: *Triple Modular Redundancy* – TMR).

Zur weiteren Erhöhung der Ausfallsicherheit können anstatt der Task-Klone verschiedene Implementierungen der Task verwendet werden (N-version programming). Dabei muss Binärkompatibilität der Ergebnisse bzw. der abzugleichenden Daten gewährleistet sein.

In den vorhergehenden Abschnitten wurde die Grundlage für die Daten-synchrone Verarbeitung geschaffen. Im Folgenden wird die Methodik der Taskreplikation mit den Timed Messages verbunden.

4.1.1 Fehlerfreie Datenübertragung

Um eine fehlerfreie Datenübertragung sicherzustellen, werden (Zwischen-)Ergebnisse zwischen den Cotasks ausgetauscht und votiert. Durch die Rückübertragung der Votierergebnisse an *alle* Cotasks lässt sich ein Fehlerbild (Syndrom) erzeugen [14].

Mehrheitsentscheide finden in den Recheneinheiten und in den intelligenten Aktoren bzw. im letzten Gateway statt. Die Überprüfung findet bereits auf den Eingangsdaten statt, d. h. die Nachrichten der Cotasks werden einem binären Vergleich unterzogen, noch bevor sie an die nächst höhere Schicht (z. B. an eine verarbeitende Domäne/Task) weitergereicht werden.

Da sich nicht nur die Tasks auf unterschiedlichen Recheneinheiten befinden müssen, sondern auch die Kommunikationspfade disjunkt sein müssen, bietet es sich bei TMR an, jede Recheneinheit mit genau drei weiteren Recheneinheiten zu verbinden. Somit führt der Ausfall einer physikalischen Kommunikationsnetz-Verbindung (Kante) höchstens dazu, dass eine Recheneinheit keine Werte mehr bekommt. Zur weiteren Erhöhung der Verfügbarkeit können auch die Kommunikationswege redundant ausgelegt werden; damit können trotz Ausfalls einer Kante alle Pakete den Recheneinheiten zugestellt werden. Dieser Vorteil wird allerdings durch einen erhöhten Kommunikationsaufwand und die zusätzlich notwendige Überprüfung der redundanten Pakete erkauft. Deshalb wird diese Möglichkeit im Folgenden nicht weiter beleuchtet.

Im Folgenden ist die fehlertolerante Datenübertragung nach [14] dargestellt. Der Ablauf ist unabhängig vom Redundanzgrad; erst bei der Betrachtung der Fehlerbilder muss dieser in Betracht gezogen werden. Die Einzelschritte werden auf das vorliegende System angepasst. Es ergeben sich folgende Unterschiede.

Im vorliegenden System variiert die Übertragungsdauer, d. h. es kann nicht implizit von Datensynchronität ausgegangen werden. Die maximale Übertragungszeit (engl.: *Worst-Case Transmission Delay* – WCTD) muss in die Berechnung einfließen. Dem wird durch die Timed Messages Rechnung getragen. Die maximale Übertragungsdauer eines Pakets der Priorität i wird gemäß Abschnitt 3.2, Gleichung 3.6 mit δ_{WCTD} angegeben.

Zudem gibt es keine sog. „Single Sender"-Kanäle, bei welchen jeweils nur eine Komponente schreibenden und alle anderen Komponenten nur lesenden Zugriff erhalten. Vielmehr kann der physikalische Übertragungskanal von mehreren Sendern genutzt werden. Insofern hat der Ausfall eines physikalischen Senders (z. B. der Transmit-Unit eines Switches) Auswirkungen auf das Gesamtsystem und nicht nur auf einen logischen Sender (z. B. eine Recheneinheit). Andererseits bietet die hier vorgeschlagene redundante Anbindung immer noch alternative Pfade durch das Kommunikationsnetz an, so dass die Recheneinheit in diesem Fall nicht vom System abgetrennt wird.

Senden ①

Unter Verwendung der hybriden Lösung der Kommunikationsnetzanbindung (Abschnitt 3.3.7) teilt eine Task ihren Sendewunsch mittels Hypercall mit. Die Nachricht wird daraufhin durch den Hypervisor verarbeitet, einsortiert und schließlich an die Netzwerkkarte übergeben. Die Zieladresse ist eine Multicast-Adresse, dadurch wird die Nachricht je nach Redundanzgrad an eine oder mehrere Recheneinheiten versendet.

Switched Ethernet zeichnet sich u. a. durch Punkt-zu-Punkt Verbindungen aus. Aus physikalischer Sicht hat ein Sender zu einem bestimmten Zeitpunkt exklusives Senderecht auf dem Übertragungskanal. Allerdings teilen sich verschiedene logische Sendekomponenten (z. B. Tasks) einen physikalischen Übertragungskanal. Die zu sendende Nachricht kann sich dabei maximal um die in Abschnitt 3.2 beschriebene Zeit verzögern.

In der virtuellen Umgebung auf den Recheneinheiten ist die Exklusivität dadurch gewährleistet, dass der Speicherbereich, der für die Kommunikation verwendet wird, immer nur für eine (virtuelle) Komponente blockierungsfrei beschreibbar ist. Alle anderen (virtuellen) Komponenten haben bestenfalls lesenden Zugriff (siehe auch: „Blockierungsfreier Speicherzugriff" in [46]). Die mögliche Verzögerung durch die Paketverarbeitung in den Recheneinheiten ist in Abschnitt 3.3 dargestellt.

Synchronisieren ②

Prinzipiell muss vor dem Votieren eine bestimmte Anzahl an Einzelergebnissen vorliegen, bevor mit dem Votieren begonnen werden kann (bei einem k-aus-n–System mindestens k). Da das

vorliegende System mit aufeinanderfolgenden Gültigkeitszeitpunkten arbeitet, darf nicht vor diesen mit der Votierung begonnen werden. Dies bedeutet auch, dass im fehlerfreien Fall alle n Nachrichten auf allen beteiligten Recheneinheiten bereitstehen, das System also synchronisiert ist.

Votieren ③

Der Mehrheitsentscheid liefert zum einen die fehlerfreie Nachricht, die zur lokalen Weiterverarbeitung freigegeben wird, und zum anderen die Auswertung des Mehrheitsentscheids, die an die auswertende Stelle übermittelt werden kann. Dabei kann es sich um die Sendetasks selbst handeln oder um eine (zentrale) Diagnosestelle. Die Rückmeldungen (die Ergebnisse der Votierungen) werden ebenfalls votiert.

Bestätigen ④

Das Rücksenden der Votierungsergebnisse an die Sendetasks oder eine Diagnosestelle dient zum einen als Bestätigung, dass die Nachricht beim Empfänger angekommen ist und zum anderen zur Überprüfung, ob ein Fehler aufgetreten ist, der eine Fehlerbehandlung erfordert. Erfolgt die Bestätigung an eine Diagnosestelle, kann die Fehlerbehandlung erst entsprechend verzögert eingeleitet werden (sie muss die Sender erst über den Fehler informieren, bevor diese reagieren können). Daher ist es sinnvoll, die Bestätigung an die Sendetasks zu übermitteln und die Fehlerbehandlung bei einem eindeutigen Fehler dort zu initiieren. Nur bei unbekanntem bzw. nicht eindeutigem Syndrom informieren die Sendetasks die Diagnoseeinheit über den Fehler.

Bei internen Synchronisationspunkten (siehe Abschnitt 4.1.3) kann das Rücksenden der Bestätigung entfallen, da die Zielknoten den Quellknoten entsprechen. In diesem Fall kann ein Fehler jedoch nur erkannt, aber nicht behoben werden. Deshalb muss eine übergeordnete Instanz (Diagnosestelle) über den aufgetretenen Fehler informiert werden.

Fehlerbehandlung starten ⑤

Anhand der Ergebnisse der Votierungen kann eine Fehlerbehandlung notwendig sein. Diese ist abhängig vom Redundanzgrad der Task. Welche Fehler erkannt werden können und wie darauf reagiert wird, ist in Abschnitt 4.6 beschrieben. Prinzipiell erfolgt bei der Fehlerbehandlung eine Restauration der Systemfunktionalität z. B. durch erneutes Aufsetzen einer fehlerhaften Task oder durch Umrouten des Nachrichtenverkehrs.

Freigeben der Nachricht ⑥

Für den Fall nicht-synchroner Nachrichten kann wie folgt vorgegangen werden: Nach erfolgreicher Votierung kann die Nachricht zur Bearbeitung für die Empfängertasks freigegeben werden.

War die Votierung nicht erfolgreich, wird stattdessen eine Fehlernachricht generiert und an die Empfängertasks weitergereicht.

Handelt es sich um synchrone Nachrichten, enthalten diese einen Gültigkeits-Zeitstempel, zu dem die Nachricht zur weiteren Verarbeitung freigegeben wird. Falls die Nachricht eine Triggerbedingung für eine Task darstellt, wird die Task (bzw. die zugehörige Domäne) aktiviert. Im vorliegenden System werden für Domänen aus den Klassen HRT und System ausschließlich synchrone Nachrichten verwendet. Die Berechnung des Gültigkeitszeitpunktes einer Timed Message wird in den nächsten Abschnitten entsprechend angepasst.

4.1.2 Redundante Empfänger

Bild 4.1 zeigt den zeitlichen Verlauf einer fehlertoleranten Datenübertragung, ausgehend von drei Cotasks auf drei Recheneinheiten (RE1 bis RE3). Die eingekreisten Ziffern entsprechen den o. g. Teilschritten beim Senden. Der Übersichtlichkeit halber sind nur ausgewählte Kommunikationsbeziehungen eingezeichnet; tatsächlich kommuniziert jede Recheneinheit (RE) mit jedem Empfänger (E) und umgekehrt.

Im folgenden Beispiel werden der Einfachheit halber für alle Recheneinheiten bzw. alle Empfänger identische Rechenzeiten c_{RE} bzw. c_{E} und identische Scheduling-Fehler $\varepsilon_{\mathrm{RE}}$ bzw. ε_{E} angenommen.

In RE1 bis RE3 werde zum Zeitpunkt t_v^0 ein Wert gültig, der als Triggerbedingung für die drei Cotasks eingetragen ist. Die Cotasks generieren Ergebnisse für die Empfänger E1 bis E3, die jeweils dorthin gesendet werden, wo sie spätestens zum Zeitpunkt t' ankommen. Im Anschluss findet die Votierung statt und die Nachrichten werden zum Zeitpunkt t_v^1 in den Empfängern zur weiteren Verarbeitung gültig. Gleichzeitig generieren die Empfänger die Bestätigungsnachrichten und informieren die Sendetasks auf den Recheneinheiten RE1 bis RE3. Die Rückmeldungen werden ab Zeitpunkt t'' votiert und ggf. eine Fehlerbehandlung eingeleitet. Zum Zeitpunkt t_v^2 sind auch die Sendetasks wieder synchron und bereit, auf neue Anfragen zu reagieren.

Die einzelnen Gültigkeitszeitpunkte berechnen sich nach Gleichung 3.1 zu:

$$t_v^1 = t_v^0 + c_{\mathrm{RE}} + \varepsilon_{\mathrm{RE}} + \delta_{\mathrm{WCTD}} + c_{\mathrm{E}} + \varepsilon_{\mathrm{E}} + 2\eta$$

und

$$t_v^2 = t_v^1 + \delta_{\mathrm{WCTD}} + c_{\mathrm{RE}} + \varepsilon_{\mathrm{RE}} + 2\eta\,.$$

Hier wird deutlich, wie wichtig möglichst genau synchronisierte Uhren sind: Bei jeder fehlertoleranten Übertragung fallen 2η an – gleichgültig davon, ob die Recheneinheiten mehrfach angesprochen werden, oder nicht.

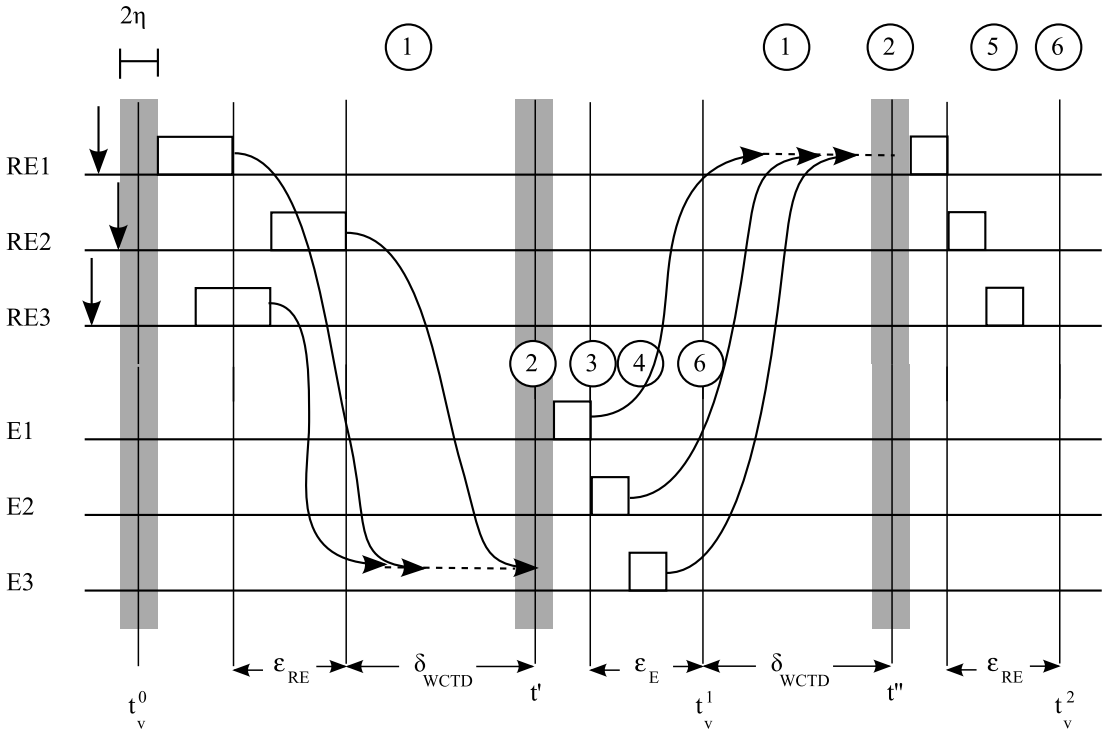


Bild 4.1: Zeitlicher Verlauf für Ergebnisvalidierung in 3 Empfängern

4.1.3 Interne Synchronisationspunkte

Bild 4.2 zeigt den Spezialfall der internen Synchronisation. Dabei gleichen die Cotasks untereinander Zwischenergebnisse ab, um eventuelle Fehler frühzeitig zu erkennen. Wieder sind nur ausgewählte Kommunikationsbeziehungen eingezeichnet; jede Recheneinheit kommuniziert mit jeder Recheneinheit.

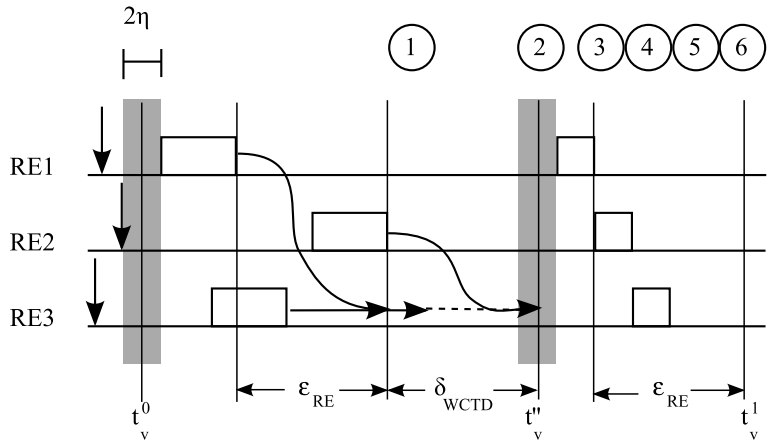


Bild 4.2: Zeitlicher Verlauf für internen Synchronisationspunkt

Das Rücksenden der Bestätigung kann entfallen, allerdings können Fehler dann nur erkannt, aber nicht behoben werden. Deshalb muss eine übergeordnete Instanz (Diagnosestelle) über einen Fehler informiert werden.

4.1.4 Redundante Sensoren

Für sicherheitskritische Anwendungen werden Sensoren oft redundant, d. h. mehrfach ausgelegt. Die Überprüfung auf einen konsistenten Systemzustand kann entweder direkt nach dem Einlesen der Sensorwerte erfolgen, als interner Synchronisationspunkt des TMR-Systems (analog zu Bild 4.2), oder das Voting erfolgt erst über den Ausgangswert am Empfänger (analog zu Bild 4.1).

Beispiel:

Mehrere Sensoren beobachten denselben physikalischen Wert und liefern diesen an drei Recheneinheiten. Dort wird eine Plausibilitätsüberprüfung durchgeführt und die Einzelwerte zu einem Wert fusioniert. Das Ergebnis der Fusion gleichen die Cotasks untereinander ab, bevor sie sie weiterverarbeiten.

4.1.5 Anwendung auf Wirkketten

Wirkketten erstrecken sich von der Sensorwertaufnahme über die Verarbeitung bis hin zur Ausgabe am Aktor. Da eine Verarbeitung sich mehrerer Sensoren mit unterschiedlichen Raten bedienen kann, ist eine Aktivierung der Verarbeitung durch jedes Sensor-Ereignis nicht immer sinnvoll. Die Aktivierung sollte durch ein ausgewähltes Sensorsignal oder zyklisch erfolgen.

Werden innerhalb der Anwendung weitere Sensorwerte abgefragt, muss darauf geachtet werden, dass alle Replikate (Cotasks) auf den selben Daten arbeiten. Hierzu dürfen nicht die aktuellen Sensorwerte abgefragt werden, da nicht bekannt ist, in welchem Zustand sich die Cotasks befinden. Es muss explizit ein Zeitpunkt ausgesucht werden, den alle Cotasks sicher erreicht haben. Entweder verwendet man den Zeitpunkt der Taskaktivierung (t_v des aktivierenden Sensorereignisses bzw. der zyklischen Aktivierung) oder des letzten internen Synchronisationspunktes. Dementsprechend erhöht sich das maximale Alter der abgefragten Sensorwerte um die Differenz zwischen dem gewählten t_v und dem Abfragezeitpunkt.

4.2 Verwendung einer Echtzeit-Datenbank

Auf jeder Recheneinheit werden Sensordaten und – bis zu einem gewissen Grad – deren Historie benötigt. Zusätzlich sollen für eine schnelle Fehlerbehandlung interne Zustände gespeichert werden können. Beispielsweise DeeDS (distributed active real-time database system) [4, 70]

und die KogMo-RTDB (Echtzeitdatenbasis für kognitive Automobile) [46] bieten diese Funktionen an. Allerdings gibt es bei DeeDS keinen durchgängigen Replika-Determinismus und bei der KogMo-RTDB fehlt derzeit noch die Unterstützung verteilter Systeme. Im Gegensatz zu diesen Lösungen ist es für das vorliegende System nicht notwendig, auf jeder Recheneinheit alle Daten vorzuhalten.

Im Folgenden wird kurz auf die wichtigsten Punkte zur Konzeption einer verteilten Echtzeit-Datenbank (dRTDB) eingegangen.

4.2.1 Konzeptionelle Überlegung

Folgende Randbedingungen ergeben sich aus der vorliegenden Systemarchitektur. Ergänzt werden sie durch Überlegungen aus [46].

- Es gibt nur eine schreibende Instanz pro Objekt in der Datenbank.
- Der Lesezugriff erfolgt über gemeinsam genutzte Speicherbereiche von beliebigen Domänen aus (blockierungsfreier Lesezugriff nach [46]).
- Es werden nur Sensorwerte, deren Historie sowie zur Fehlerbehandlung notwendige interne Zustände der Tasks bzw. Domänen gespeichert.
- Neue Sensorwerte kommen über das Netzwerk an der Recheneinheit an.
- Sensorwerte werden über Multicast versendet. Das Kommunikationsnetz bestimmt, an welche Empfänger das Paket geleitet werden muss; Empfänger müssen sich am Kommunikationsnetz-Manager anmelden, wenn sie einen bestimmten Sensorwert abonnieren wollen.
- Der Abgleich von internen Task-Zuständen erfolgt von den Tasks selbst an jedem Synchronisationspunkt, spätestens aber nach Beendigung einer Periode (z. B. am Ende der Berechnung eines neuen Ausgangswerts).
- Applikationsspezifische Daten, die z. B. für das Wiederaufsetzen einer Applikation notwendig sind, müssen durch die Applikation selbst in die Datenbank geschrieben werden.
- Die Objekte müssen in allen Instanzen der Datenbank, d. h. auf allen Recheneinheiten im Verbund, auf denen das Objekt benötigt wird, konsistent sein (Wert und Zeit).

Im vorliegenden System kommen folgende Möglichkeiten der Instanziierung pro Recheneinheit in Betracht:

- Die Datenbank wird in jeder Domäne angelegt,
- sie befindet sich in einer eigenen virtuellen Domäne,
- sie befindet sich im Kontext der Treiberdomäne oder
- sie befindet sich im Kontext des Hypervisors.

Im Hinblick auf die Randbedingungen ist es am sinnvollsten, die dRTDB im Kontext der Treiberdomäne zu betreiben. Über einen Hypercall können weitere virtuelle Domänen einen Wert in die dRTDB schreiben lassen, jedoch erhält nur die Treiberdomäne ein exklusives Schreibrecht. Die dRTDB wird über einen gemeinsam genutzten Speicherbereich in den Adressraum der anderen Domänen eingeblendet. Der Hypervisor konfiguriert die MMU so, dass diese

Domänen nur lesend (read-only) auf den Speicherbereich zugreifen können. In den Domänen sind dRTDB-Clients angesiedelt, die dem Benutzer eine einheitliche API zum Zugriff auf die Objekte der dRTDB anbieten. Die folgenden Beschreibungen beziehen sich auf diese Variante.

Die in der Applikation festgelegten Synchronisationspunkte können zusätzlich dazu genutzt werden, um ein Objekt mit internen Zustandsdaten für eine eventuelle Fehlerbehandlung in der dRTDB zu speichern.

4.2.2 Rolle und Verhalten der Treiberdomäne

Eingehende Pakete. Wenn eine Applikation auf einen Sensorwert zugreifen will, kann sie sich entweder bei der dRTDB für ein Objekt registrieren oder sie meldet sich direkt zum Bezug des zugehörigen Multicast-Pakets an. Für die meisten fehlertoleranten Applikationen sollte die dRTDB-Anbindung gewählt werden. Die dRTDB registriert sich ihrerseits als Empfänger für die entsprechenden Multicast-Pakete. Die dRTDB muss es auch ermöglichen, eine registrierte Task über den Empfang eines Pakets abhängig davon zu benachrichtigen, ob es sich um ein Sensorsignal mit Auslösebedingung handelt oder nicht.

Empfänger des Pakets ist damit die Treiberdomäne und nicht die Zieldomäne. Die Treiberdomäne übernimmt zusätzlich folgende Aufgaben:

- Verarbeiten des Pakets bzw. Auspacken des dRTDB-Objekts.
- Ggf. berechnen von t_v aus dem Aufnahme-Zeitstempel (t_m).
- Eintragen des dRTDB-Objekts mit dem korrekten Gültigkeitszeitpunkt in die dRTDB.
- Ggf. Generieren eines Scheduling-Events, das zum Zeitpunkt t_v das Eventflag für die bezugsberechtigten Zieldomänen setzt. Dieses Event führt zum Zeitpunkt t_v zu einem Rescheduling.

Ausgehende Pakete. Ausgangswerte werden nur dann in der dRTDB gespeichert, wenn sie gleichzeitig Eingangswerte für eine Task auf der gleichen Recheneinheit darstellen. Da sich alle HRT-Applikationen an derselben Instanz der dRTDB anmelden, ist dies leicht zu bewerkstelligen.

Interne Task-Zustände. Die Speicherung interner Zustände, die zur Wiederherstellung einer Cotask erforderlich sind, erfolgt am sinnvollsten an den Synchronisationspunkten.

Deshalb wird vorgeschlagen, die notwendigen, taskabhängigen Objekte zusammen mit dem Hypercall für die Paketübertragung an die Treiberdomäne zu übermitteln. Diese sog. History-States müssen lediglich lokal gespeichert werden: Im fehlerfreien Fall haben alle Replikate den gleichen internen Zustand und liefern das gleiche Ergebnis; dieses wird votiert. Im Fehlerfall muss eine Fehlerbehandlung erfolgen und (mindestens) der letzte interne Zustand der als fehlerhaft identifzierten Task gelöscht bzw. mit den Cotasks synchronisiert werden.

4.2.3 Zeitliche Betrachtung

Die Treiberdomäne benötigt die zusätzliche Rechenzeit c_{RTDB} für die Verwaltung der Datenbank. Allerdings entfällt in den Zieldomänen die Bearbeitung durch den Kommunikationsstack. Es erfolgt lediglich ein Lesezugriff auf den gemeinsam genutzten Speicherbereich per Hypercall. Dadurch reduziert sich die Verarbeitungszeit der Zieldomänen.

Die Zeit für ausgehende Pakete bleibt unverändert, falls keine internen Zustände zu speichern sind. Andernfalls muss diese zusätzliche Rechenzeit (zur Vereinfachung ebenfalls mit c_{RTDB} berechnet) berücksichtigt werden.

4.3 Erweiterte Fehlerbehandlung

Bisher wurde lediglich die Fehlermaskierung durch TMR betrachtet. Im Folgenden wird auf die Fehlererkennung und mögliche erweiterte Mechanismen zur Fehlerbehandlung eingegangen. Bei sicherheitskritischen Funktionen versucht die Fehlerbehandlung den ursprünglichen Redundanzgrad wieder herzustellen. Die beschriebenen Mechanismen können jedoch ebenso für die übrigen Applikationen benutzt werden.

4.4 Randbedingungen

Abhängig von den Randbedingungen ändern sich die Fehlerbilder bzw. die daraus gezogenen Schlüsse. Deshalb sind hier die Randbedingungen aus den vorherigen Kapiteln zusammengefasst dargestellt.

Einfachfehler-Annahme: Es kann nur ein Fehler zu einem Zeitpunkt auftreten.

Voting mit Rückübertragung des Ergebnisses: Auf den Empfängern einer Nachricht findet ein Voting mit Rückübertragung der einzelnen Voting-Ergebnisse an die Sender gem. Abschnitt 4.1.1 statt. Die Auswertung der Fehlerbilder geschieht nach der Rückübertragung in den Sendern.

Knotendisjunkte Pfade für Taskklassen 2 und 3: Die Kommunikation zwischen einem Sender und den 2 oder 3 Empfängern, auf denen sich die Cotasks befinden, geschieht über knotendisjunkte Pfade.

Einfache oder redundante Kommunikation: Abhängig von der geforderten Ausfallsicherheit kann es zwischen zwei kommunizierenden Knoten einen einfachen oder mehrere, redundante Kommunikationspfade geben. Beide Varianten werden hinsichtlich ihres Fehlerbildes betrachtet.

Keine Teilnetze: Im Falle von Mehrfachfehlern ist es möglich, dass sich Teilnetze bilden. Im vorliegenden System ist dies beispielsweise der Fall, wenn der an eine Recheneinheit angeschlossene Switch ausfällt. Die separierte Recheneinheit bildet ein Teilnetz, das

restliche System ein weiteres. In diesem Fall ist sicherzustellen, dass die separierte Recheneinheit in einen passiven Zustand übergeht, um bei einer eventuellen Wiedereingliederung (z. B. wenn es sich um einen transienten Fehler eines Switches gehandelt hat) das restliche System nicht zu kompromittieren.

Keine byzantinischen Fehler: Es wird davon ausgegangen, dass ein Votingergebnis eines Empfängers an allen Sendern identisch vorliegt. Dies kann beispielsweise durch eine Prüfsummenberechnung über das Votingergebnis erfolgen; das Votingergebnis inklusive Prüfsumme wird dann einzeln oder per Multicast an die Sender zurückgesendet. Eine Prüfsummenberechnung innerhalb der einzelnen Senderoutinen ist nicht erlaubt, da sich zwischenzeitlich das Votingergebnis z. B. durch Speicherfehler geändert haben kann.

4.5 Mögliche Fehler und deren Erkennung

Als Fehlerfälle werden Ausfälle und Fehlverhalten im Zeit- und Wertebereich von verschiedenen Komponenten betrachtet:

- Kante im Kommunikationsnetz
- Switch
- Recheneinheit, Hypervisor, Treiberdomäne
- Virtuelle Applikationsdomäne

Intelligente Sensoren und Aktoren sind wie Recheneinheiten zu betrachten.

4.5.1 Kante

Der Ausfall einer Kante hat zur Folge, dass die hierüber laufenden Pakete verloren gehen. Es gibt zwei Möglichkeiten, einen Ausfall zu detektieren. Über den Link-Status erkennt der Controller des physikalischen Layers eines Ports, ob an dem Port eine Verbindung zu einem anderen Teilnehmer vorhanden ist. Diese Information kann an die Diagnosestelle zur Auswertung und ggf. Einleitung einer Fehlerbehandlung weitergereicht werden. Die Signalisierung des Link-Status durch den Controller-Baustein ist herstellerabhängig und nicht näher spezifiziert. Zudem kann es in seltenen Fällen vorkommen, dass zwar ein Link erkannt wird, aber dennoch keine Pakete übertragen werden können. Deshalb ist diese Erkennungsmethode als unsicher einzustufen und deshalb nicht als alleinige Ausfalldetektion geeignet.

Eine indirekte Erkennung erfolgt dadurch, dass der Switch pro Port eine Statistik über die Paketraten bzw. über den maximalen Paketabstand führt und ein anormales Verhalten an die Diagnosestelle meldet.

Eine weitere Erkennung ist durch die Timeouts in den Votierern auf den Recheneinheiten gegeben: Fehlt eine der Nachrichten, wird dies ebenfalls an die Diagnosestelle gemeldet.

Die Erkennung durch die Switches (Link- und Port-Statistik) kann dennoch herangezogen werden – bevor die Diagnosestelle jedoch eine Fehlerbehandlung anstoßen darf, muss der Fehler von weiteren Stellen ebenfalls erkannt und weitergeleitet worden sein. Liegt tatsächlich ein defekter Link vor, werden dies beide Switches an den Enden der Verbindung an die Diagnosestelle melden.

Ein Fehler im Wertebereich, d. h. der Paketinhalt wurde verändert, wird erst in den Recheneinheiten durch eine falsche Prüfsumme erkannt. Dieser Sachverhalt wird an die Diagnosestelle weitergereicht.

4.5.2 Switch

Der Ausfall eines Switches hat ebenfalls Paketverluste zur Folge. Dieser Ausfall wird mit Hilfe der Port-Statistiken der drei Nachbar-Switches erkannt und an die Diagnosestelle weitergeleitet.

Zusätzlich schlagen die Votierungen einiger Recheneinheiten fehl, wenn sie entweder Pakete erwarten, die über den ausgefallenen Switch geroutet wurden, oder die sendende Cotask auf der nun abgeschnittenen Recheneinheit platziert war.

Eine anormale Verzögerung eines Pakets sowie ein manipuliertes Paket wird erst in den Recheneinheiten (Netzwerk-Treiberdomäne) über einen Timeout oder die Prüfsumme erkannt und eine entsprechende Meldung an die Diagnosestelle abgesetzt.

4.5.3 Recheneinheit, Hypervisor, Treiberdomäne

Diese drei Komponenten können größtenteils zusammen betrachtet werden; lediglich einige Hardware-spezifische Fehler einer Recheneinheit können zusätzlich gesondert bestimmt werden. Beispielsweise deutet das häufige Auftreten eines Speicherfehlers, erkannt durch Verfahren wie ECC (engl.: *Error Checking and Correction*), auf einen physikalischen (Teil-)Defekt des Speicherbausteins hin.

Fällt eine der drei Komponenten aus, kann die betroffene Recheneinheit keine Pakete mehr verschicken. Der Wegfall der Pakete wird sowohl im direkt angeschlossenen Switch (Port-Statistik) als auch in den Votierern der konsumierenden Recheneinheiten (Timeout) erkannt und an die Diagnosestelle gemeldet.

Anormale Verzögerungen eines Pakets oder einer Task werden von der Treiberdomäne bzw. dem Hypervisor erkannt. Ist eine dieser Komponenten jedoch selbst von einem Fehler betroffen (z. B. fehlerhafte RTC), erfolgt die Erkennung indirekt über die Detektion von Unregelmäßigkeiten des Paketstroms in den Switches oder den konsumierenden Recheneinheiten.

Manipulierte Pakete werden ebenfalls erst in den Partner-Recheneinheiten erkannt (Prüfsumme).

4.5.4 Virtuelle Applikationsdomäne

Der Hypervisor überwacht das Normalverhalten einer virtuellen Domäne bzgl. der ihm bekannten Paramter. Darunter fallen Ressourcenbedarf (Prozessor, Speicher, I/O) und zeitliches Verhalten (Aktivierungsfrequenz, maximale Ausführungszeit). Die Netzwerk-Treiberdomäne überprüft den maximal erlaubten Paketstrom der angeschlossenen Domänen und meldet ein Fehlverhalten an die Diagnosestelle. Darüber hinaus beschränkt die Treiberdomäne den Paketstrom auf das erlaubte Maß (siehe Abschnitt 3.3).

Manipulierte Pakete werden wiederum erst in den Partner-Recheneinheiten erkannt (Prüfsumme).

4.6 Fehlerlokalisierung

Nach dem Ausfall einer Komponente ist verschiedenen anderen Komponenten des Systems lediglich bekannt, dass ein Fehler aufgetreten ist. Die genaue Quelle ist lokal in den einzelnen Komponenten nicht bestimmbar - es ist eine globale Systemsicht notwendig: Eine Diagnosestelle (siehe auch Systemkomponente „Diagnose- und Fehlermanagement“, Abschnitt 2.6.1) muss die Auswertung der Fehlerbilder aller betroffenen Teilnehmer durchführen. Auch diese Diagnosestelle muss redundant mit gegenseitigem Abgleich arbeiten. Des Weiteren ist es sinnvoll, die Kommunikation zur Diagnosestelle redundant auszulegen, um Paketverluste im Falle eines Ausfalls im Kommunikationsnetz zu vermeiden. Im Folgenden sind mögliche Fehlerbilder (Syndrome) und die Eingrenzung der Fehlerursache aufgelistet.

Ein Sender liefert ein falsches oder kein Ergebnis Ⓐ

Jeder Empfänger erkennt das Fehlverhalten und sendet das Fehlerbild an alle Sender zurück. Dieses Fehlerbild ist auf Grund der Einfachfehler-Annahme eindeutig einer Recheneinheit zuzuordnen. Mögliche Ursachen sind:

- Senderseitiger Ausfall oder ein Fehlverhalten der Recheneinheit, der virtuellen Domäne oder Task, oder der Netzwerk-Treiberdomäne.
- Senderseitiger Ausfall auf Kommunikations-Ebene, d. h. Ausfall der virtuellen Netzwerkkarte, der physikalischen Netzwerkkarte oder des gesamten Switches.

Welche dieser Ursachen tatsächlich verantwortlich sind, kann nur über die Diagnosestelle und mit weiteren Informationen bestimmt werden. Deshalb ist es wichtig, dass jeder erkannte Fehler an die Diagnosestelle gemeldet wird, um ein konsistentes Bild des Gesamtsystems zu erhalten:

Teilausfall der Recheneinheit: Der Ausfall beschränkt sich auf eine Applikation, wenn es weitere aktive Applikationen auf der Recheneinheit gibt, aber keine weiteren Fehler gemeldet wurden.

Gesamtausfall der Recheneinheit: Werden mehrere Fehler aus unterschiedlichen Applikationen über eine Recheneinheit gemeldet, kann von einem Gesamtausfall der Recheneinheit ausgegangen werden. In diesem Fall müssen alle sicherheitskritischen Applikationen dieser Recheneinheit verschoben werden.

Bei einem Empfänger kommt die Nachricht eines Senders nicht an Ⓑ

Dieser Fehler kann nur auftreten, wenn die Kommunikationsverbindung zwischen Sender und Empfänger einfach ausgelegt ist. Falls für den Rückkanal der gleiche Pfad ausgewählt wurde, ist das Fehlerbild Ⓓ sichtbar. Bei redundant ausgelegten Verbindungen kommt die Nachricht beim Empfänger lediglich einmal an. Der Fehler kann vom Empfänger direkt an die Diagnosestelle gemeldet werden, da diese Fehlerinformation für den Sender irrelevant ist.

Die Fehlerursache ist der Ausfall einer Kante entlang des Pfades.

In diesem Fall darf lokal (in den Sendern) keine Fehlerbehandlung eingeleitet werden. Nur das Zusammenführen und Auswerten aller Votingergebnisse in der Diagnosestelle lässt eine eindeutige Schlussfolgerung auf die Fehlerursache zu. Oftmals handelt es sich gerade bei Kommunikationsausfällen nur um transiente Fehler, so dass keine Rekonfiguration notwendig ist. Treffen jedoch mehrere Fehler der Fehlerbilder Ⓑ oder Ⓓ in der Diagnosestelle ein, kann von einem dauerhaften Fehler ausgegangen werden. In diesem Fall findet eine Neuberechnung aller Pfade statt, die die defekte physikalische Kommunikationsverbindung enthalten.

Ein Empfänger antwortet nicht Ⓒ

Lokal ist dieser Fehler nicht von dem Fehlerbildern Ⓑ und Ⓓ zu unterscheiden. Eine lokale Fehlerbehebungsmaßnahme ist daher nicht möglich.

Mögliche Fehlerursachen sind:

- Empfängerseitiger Ausfall oder ein Fehlverhalten der Recheneinheit, der virtuellen Domäne oder Task, oder der Netzwerk-Treiberdomäne.
- Empfängerseitiger Ausfall auf Kommunikations-Ebene, d. h. Ausfall der virtuellen Netzwerkkarte, der physikalischen Netzwerkkarte oder des gesamten Switches.

Analog zu Fehlerbild Ⓐ kann die tatsächliche Ursache erst durch die Diagnosestelle festgestellt werden:

Teilausfall der Recheneinheit: Der Ausfall beschränkt sich auf eine Applikation, wenn es weitere aktive Applikationen auf der Recheneinheit gibt, aber keine weiteren Fehler gemeldet wurden.

Gesamtausfall der Recheneinheit: Werden mehrere Fehler aus unterschiedlichen Applikationen über eine Recheneinheit gemeldet, kann von einem Gesamtausfall der Recheneinheit ausgegangen werden. In diesem Fall müssen alle sicherheitskritischen Applikationen dieser Recheneinheit verschoben werden.

Die Rückmeldung eines Empfängers kommt bei einem der Sender nicht an Ⓓ

Analog zu Fehlerbild Ⓑkann dieser Fehler nur auftreten, wenn die Kommunikationsverbindung zwischen Empfänger und Sender einfach ausgelegt ist. Falls für den Hinkanal der gleiche Pfad ausgewählt wurde, tritt auch Fehlerbild Ⓑ auf. Bei redundant ausgelegten Verbindungen kommt die Nachricht beim Sender lediglich einmal an. Dies muss der Diagnosestelle gemeldet werden.

Die Fehlerursache ist der Ausfall einer Kante entlang des Pfades.

Wieder darf lokal (in den Sendern) keine Fehlerbehandlung eingeleitet werden. Nur das Zusammenführen und Auswerten aller Votingergebnisse in der Diagnosestelle lässt eine eindeutige Schlussfolgerung auf die Fehlerursache zu (siehe Fehlerbild Ⓑ).

Bei einem Empfänger kommt keine Nachricht an Ⓔ

Dieses Fehlerbild unterscheidet sich von Ⓒ dahingehend, dass der Empfänger das Ausbleiben der Nachrichten erkennt und diese Fehlermeldung an der Diagnosestelle ankommt.

Eine mögliche Fehlerursache ist ein partieller Ausfall der Kommunikation derart, dass Nachrichten zwar versendet aber nicht empfangen werden können: In Frage kommen die Netzwerk-Treiberdomäne, die virtuelle oder physikalische Netzwerkkarte oder der integrierte Switch.

Hier ist es wichtig, dass die Recheneinheit differenziert erkennt, dass es sich um eine vollständige Separierung handelt und nicht nur eine Applikation betroffen ist. Wird ein Teilnetz erkannt, muss sich die Recheneinheit passiv schalten. Eine erneute Eingliederung in das Gesamtsystem wird ggf. von der Diagnosestelle initiiert und überwacht.

Ist nur eine Applikation betroffen, ist die Kommunikation mit der Recheneinheit nach wie vor möglich. Dadurch kann die Diagnosestelle der Recheneinheit das Fehlverhalten einer Applikation signalisieren. In diesem Fall liegt kein Teilnetz vor, die Recheneinheit darf sich nicht passiv schalten.

4.7 Fehlerbehebungsmaßnahmen

Nachdem ein Fehler erkannt und klassifiziert wurde, müssen Maßnahmen ergriffen werden, die den ursprünglichen oder einen definierten degradierten Systemzustand herstellen. Je nach einzuhaltenden Zeitbedingungen gibt es verschiedene Möglichkeiten der Fehlerbehebung.

Im Folgenden wird nur auf die Rekonfiguration nach einem Ausfall einer Recheneinheit eingegangen. Die folgenden Überlegungen können jedoch auf das Kommunikationsnetz übertragen werden: Auch hier werden die Pakete nach einem Ausfall einer Kante auf andere Kommunikationsverbindungen verteilt. Die geforderten Zeitbedingungen im Kommunikationsnetz müssen auch nach der Umkonfiguration eingehalten werden.

Es wird davon ausgegangen, dass bereits einer der o. g. Fehler aufgetreten ist, d. h. dass alle lokalen Mechanismen auf Applikationsebene (z. B. Rückwärts- oder Vorwärtsfehlerbehandlungs-Mechanismen) fehlgeschlagen sind. Es stehen drei Kategorien der Fehlerbehebung zur Auswahl: *Zurücksetzen*, *Verwendung von Standby-Systemen* und *Vollständige Migration*.

Fehler sollten immer möglichst schnell behoben werden. Jedoch dürfen dabei keine anderen Echtzeit-Tasks beeinflusst werden.

Im Kommunikationsnetz erfolgt die notwendige Datenübertragung in der dritten HRT-Klasse. Die Priorität orientiert sich an der Priorität der schadhaften Task. An der Berechnung von δ_{WCTD} ändert sich nichts, da die mögliche Blockierung in den Gleichungen bereits mit δ_{max} berücksichtigt ist.

Bei den Recheneinheiten muss zwischen den intakten und der fehlerhaften Recheneinheit unterschieden werden. Die Diagnosestelle oder die fehlerhafte Recheneinheit fragt von den intakten Recheneinheiten – je nach Fehlerbehandlungsmechanismus – die internen Task-Zustände sowie aktuelle und ggf. historische Sensorwerte ab. Dies führt zu einer zusätzlichen Belastung der dortigen Treiberdomäne. Auf der fehlerhaften Recheneinheit steht die Rechenzeit der schadhaften Task sowie die der Tasks mit weichen und ohne Echtzeitanforderungen zur Verfügung, ohne dass die übrigen Echtzeit-Tasks beeinflusst werden. Erlaubt man in diesem Fall für die zu restaurierende Task dynamische Prioriätenvergabe, kann der Scheduler dieser Task T_i für die Zeit c_i mit entsprechender Periode p_i die ursprüngliche Priorität zuweisen. Für die übrigen Zeiten muss die Priorität unter der niedrigsten HRT-Priorität liegen.

Während der Rekonfigurationszeit sollten alle Werte wie im fehlerfreien Betrieb an die Recheneinheit geschickt werden. Jedoch darf die Cotask der fehlerhaften Recheneinheit nicht am Voting teilnehmen. Dadurch werden unnötige zusätzliche Abfragen an die intakten Recheneinheiten vermieden.

4.7.1 Zurücksetzen

Viele Fehler sind transient und durch einen einfachen Neustart der entsprechenden Task zu beheben. Während des Neustarts ist die entsprechende Cotask nicht verfügbar. Für Klasse-3-Tasks ist abzuwägen, ob während dieser Zeit ein degradierter Betrieb als Klasse-2-Task in Kauf genommen werden kann. Falls nicht, muss auf Standby-Systeme (siehe nächster Abschnitt) zurückgegriffen werden.

Für Klasse-2-Tasks darf diese Methode nur angewendet werden, wenn der Neustart innerhalb der nächsten Periode der Task abgeschlossen ist. Andernfalls kann keine Votierung durchgeführt werden; dies ist normalerweise nicht erlaubt. Dauert der Neustart länger als die Periode, muss wiederum auf die Standby-Systeme zurückgegriffen werden.

Klasse-1-Tasks sind im hochsicherheitskritischen Bereich nicht erlaubt. Hier ist abzuwägen, ob ein kurzzeitiger Ausfall eventuell toleriert werden kann.

Die zeitliche Betrachtung erfolgt anhand der einzelnen Schritte, die für einen Neustart einer Task notwendig sind. Es wird davon ausgegangen, dass weiterhin die für die Cotask bestimmten Sensorwerte etc. an der Recheneinheit ankommen und gespeichert werden.

1. Entfernen der schadhaften Task aus dem Speicher.
2. Speicher freigeben bzw. der neuen Cotask zuweisen.
3. Cotask starten.
4. Ggf. Abgleich des inneren Zustands.
5. Eingliederung in den Cotask-Verbund als aktive Cotask.

Die Zeiten für die Schritte 1 bis 3 sind Plattform- und Betriebssystem-abhängig und müssen für jedes System bestimmt werden. Für den Abgleich des inneren Zustands müssen im schlimmsten Fall Daten von anderen Recheneinheiten angefordert werden, falls diese nicht mit den Nachrichten der Cotasks verschickt werden bzw. kein korrekter Wert lokal vorhanden ist. Da im vorliegenden System Ethernet verwendet wird, dessen minimale Payload oftmals über der eigentlich zu übertragenden Anzahl an Daten liegt, bietet es sich an, den inneren Zustand bei jedem Abgleich zwischen den Recheneinheiten mitzuschicken; in diesem Fall kann der zeitraubende Schritt der zusätzlichen Datenübertragung entfallen.

4.7.2 Standard Standby-Systeme

Bei Standby-Systemen werden neben der aktiven Konfiguration auch Teilsysteme bereitgehalten, die im Fehlerfall die Funktion der ausgefallenen Komponenten übernehmen. In virtueller Umgebung muss ein Teilsystem nicht eine komplette Recheneinheit bedeuten, vielmehr können auch virtuelle Domänen oder Tasks als Teilsysteme gesehen werden. Grundsätzlich wird zwischen *Hot-Standby* und *Cold-Standby* unterschieden. Die zwei Varianten unterscheiden sich hinsichtlich der Zeit, bis sie sich in das laufende System eingliedern können.

Cold-Standby

Auf einer separaten Recheneinheit werden die Ressourcen für eine zusätzliche Cotask vorgehalten. Das Binary der Task wird bereits beim Systemstart auf die Recheneinheit geladen. Die Task wird im fehlerfreien Fall nicht in den Schedule einbezogen; auch sonstige Ressourcen werden zunächst nicht belegt. Im fehlerfreien Fall können sich nicht-sicherheitskritische Anwendungen dieser Ressourcen bedienen. Man spricht daher auch von fremdgenutzter Redundanz.

Die notwendigen Schritte zum Starten der Task entsprechen denen beim Zurücksetzen:

1. Speicher der neuen Cotask zuweisen.
2. Cotask starten.
3. Ggf. Abgleich des inneren Zustands.

4. Eingliederung in den Cotask-Verbund als aktive Cotask.

Neben der oben beschriebenen Beschleunigung des Abgleichs des inneren Zustands können die ersten zwei Punkte entfallen, wenn die Task bereits zu Beginn instanziiert, d. h. in den Speicher geladen wird. Für den weiteren (Normal-)Betrieb wird ihr aber keine Prozessorzeit zugewiesen. In diesem Fall kann demnach der Prozessor fremdgenutzt werden, nicht aber der Speicher. Werden diese beiden Hilfsmittel zur Beschleunigung des Startens der Task verwendet, ist eine Cold-Standby Task schneller in den Cotask-Verbund aufgenommen als dies beim Zurücksetzen der Fall wäre. Allerdings muss bedacht werden, dass die nicht-sicherheitskritischen Tasks nun weniger Prozessorleistung bekommen als vorher, d. h. diese nun ggf. degradiert weiterlaufen oder sogar eine Rekonfiguration der nicht-sicherheitskritischen Tasks erfolgen muss. Beim Zurücksetzen hingegen verbleibt das System in der ursprünglichen Konfiguration.

Hot-Standby

Auf einer separaten Recheneinheit wird eine zusätzliche Cotask platziert. Diese läuft zwar aktiv mit, wird allerdings im fehlerfreien Fall nicht in den Votierungsprozess einbezogen. Sie liefert zwar keine Ausgangswerte, muss aber für die interne Validierung die Ausgangswerte der aktiven Cotasks beziehen. Dadurch ist diese Cotask immer auf aktuellem Stand und kann fast ohne Verzögerung den Platz einer ausgefallenen Recheneinheit einnehmen. Die Verzögerung kommt dadurch zustande, dass die Diagnosestelle erst reagieren kann, wenn der Fehler bereits aufgetreten ist. Danach muss sie zum einen der Standby-Cotask signalisieren, dass sie nun zum aktiven Mitglied im Cotask-Verbund gehört. Zum anderen müssen die bereits aktiven Cotasks über den neuen Verbund-Partner informiert werden. Die ehemalige Standby-Cotask liefert daraufhin ebenfalls Ausgangswerte und nimmt am Voting teil.

Diese Variante ist nur bei knappen Kommunikationsnetzressourcen sinnvoll.

Wird, wie in Abschnitt 4.2 vorgeschlagen, eine Echtzeitdatenbasis für den Nachrichtenaustausch und den Abgleich zwischengeschaltet, passiert der Wechsel der Verbund-Mitglieder transparent für die Anwendung. Die Anwendung sollte dennoch über aufgetretene Fehler bzw. einen Wechsel der Taskklasse informiert werden, damit auch auf Applikationsebene ggf. Maßnahmen ergriffen werden können.

4.7.3 Hot-Standby mit reduzierter Reaktionszeit

Eine schnelle Umschaltung kann erzielt werden, wenn die Diagnosestelle nicht aktiv in den Umschaltprozess eingreifen muss, sondern lediglich über den aufgetretenen Fehler informiert wird. In diesem Fall wird die Fehlerlokalisation parallel zur Fehlerbehebungsmaßnahme durchgeführt.

Ein intelligenter Sensor S liefert periodisch Sensorwerte. Eine Task $T1$ berechnet daraus Stellwerte für einen Aktor A. Es soll immer nur eine Task aktiv Stellwerte senden. Fällt die darunter liegende Recheneinheit aus, muss dafür gesorgt werden, dass weiterhin Stellwerte berechnet

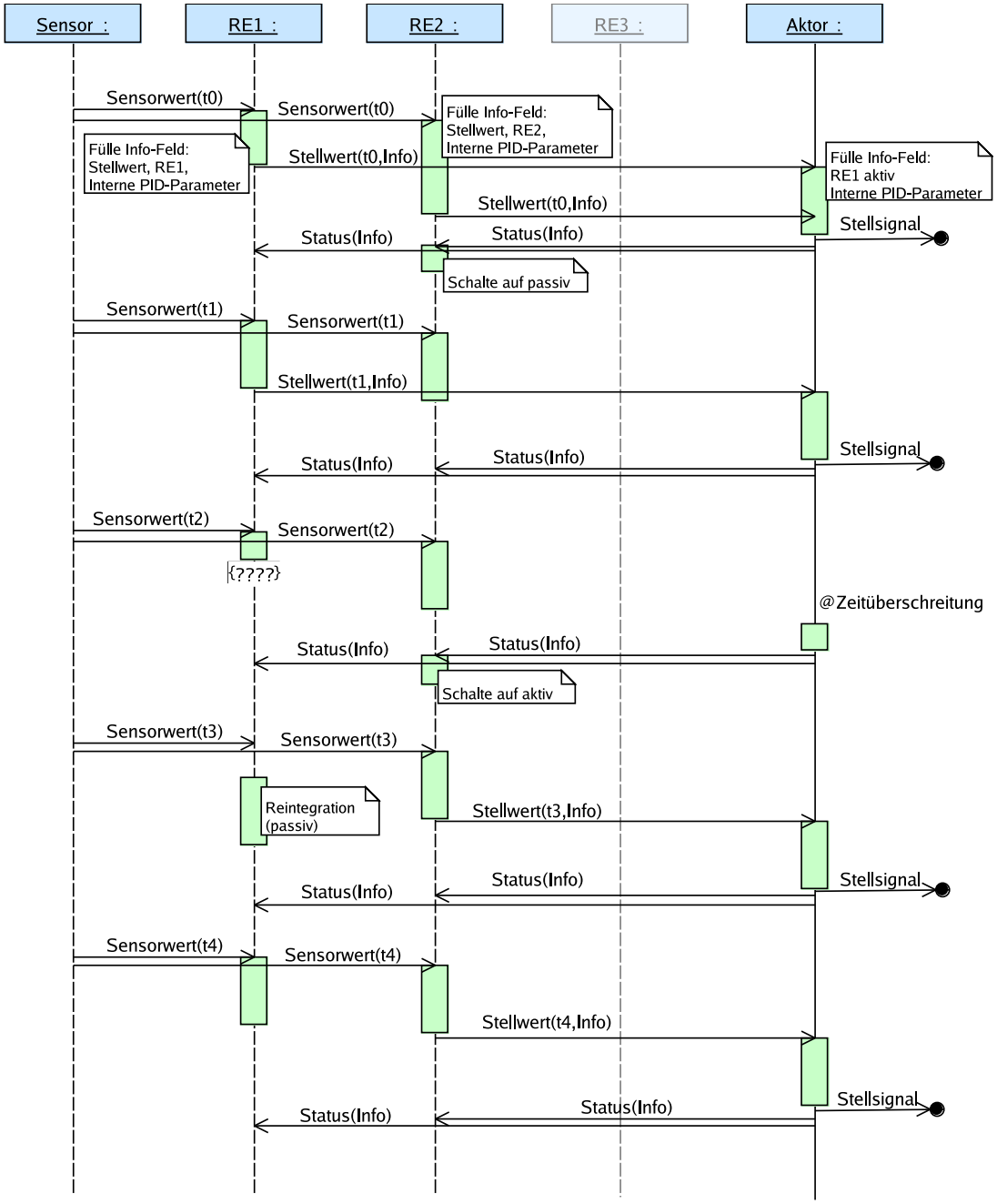


Bild 4.3: Hot Standby ohne Diagnosestelle

und gesendet werden. Hierfür werden zwei Cotasks $T1(1)$ und $T1(2)$ den Recheneinheiten RE1 bzw. RE2 zugeteilt.

Folgende Randbedingungen sind bei der Systemauslegung zu beachten:

- Der Sensor muss seine Werte an alle Cotasks senden.
- Die Cotasks müssen auf unterschiedlichen Recheneinheiten platziert werden.
- Die zwei Wirkketten
  $W_{1(1)}$: S $\rightarrow T1(1)$ auf RE1 $\rightarrow$ A und
  $W_{1(2)}$: S $\rightarrow T1(2)$ auf RE2 $\rightarrow$ A
  dürfen keine gemeinsamen Kommunikationswege besitzen (knotendisjunkt). Da die Treiberdomänen der jeweiligen Recheneinheit gemäß Abschnitt 4.2 als Empfänger auftreten, wird im Folgenden auch die Kommunikation mit der entsprechenden Recheneinheit stellvertretend für die darauf laufende Cotask angegeben.
- Der Aktor sendet eine Statusnachricht an alle Cotasks. Darin sind die aktuell aktive Cotask sowie ggf. interne Zustandsvariablen vermerkt.
- Optional: Die jeweils aktive Cotask kann zusätzlich zum Stellwert ausgewählte interne Zustände versenden, um ein schnelles Einschwingen nach dem Ausfall zu gewährleisten. Bei einem einfachen PID-Regler können dies beispielsweise der I-Anteil sowie die Verstärkungsfaktoren sein, sofern diese variabel sind.
- Optional: Die Recheneinheiten, die Teil der Wirkketten W_1 sind, können einzeln oder über eine Multicast-Adresse erreicht werden. Im Folgenden wird die direkte Adressierung verwendet, d. h. Sensor und Aktor müssen über jeden einzelnen ihrer Kommunikationspartner informiert werden; dies gilt insbesondere beim Hinzufügen, Entfernen und Verschieben von Cotasks auf andere Recheneinheiten.

Bild 4.3 zeigt die Abläufe in einem solchen System. Der Ablauf kann in folgende Phasen gegliedert werden:

Initialisierung: Solange den Cotasks noch nicht bekannt ist, welche der beiden aktiv und welche passiv sein soll, senden beide ihre Stellwerte an den Aktor. Das erste Stellsignal, das am Aktor ankommt, wird verwendet und die sendende Cotask als aktiv vermerkt.

Normaler Betrieb: Der Sensor sendet zyklisch Werte an beide Recheneinheiten. Aktive Cotasks berechnen den Stellwert und senden ihn an den Aktor. Der Aktor verwendet nur den Stellwert der als aktiv vermerkten Cotask; Stellsignale von anderen Cotasks werden ignoriert. Anschließend sendet der Aktor an beide Recheneinheiten eine Statusnachricht, welchen Stellwert er verwendet hat. Eine Cotask, deren Stellsignal nicht verwendet wurde, schaltet auf passiv, d. h. sie empfängt zwar weiterhin die Nachrichten von Sensor und Aktor, generiert aber keine Stellwerte. Falls der Aktor Zusatzinformationen in der Statusnachricht mitsendet, werden diese von der inaktiven Cotask übernommen.

Ausfall der aktiven Recheneinheit: Fällt die Recheneinheit mit der derzeit aktiven Cotask aus, wird das Ausbleiben des Stellsignals am Aktor bemerkt. Der Aktor generiert eine Statusnachricht mit der Information, dass keine Stellwerte angekommen sind, und sendet diese an alle ihm bekannten Sender – in diesem Fall RE1 und RE2. Die Cotask auf RE2 schaltet sich nach dem Empfang dieser Meldung auf aktiv und beginnt nach dem Erhalt des nächsten Sensorwertes mit der Berechnung und dem Versenden des Stellsignals. Sobald der Aktor den Stellwert von $T1(2)$ erhalten hat, vermerkt er $T1(2)$ als aktiv und ignoriert alle weiteren Stellwerte anderer Cotasks.

Ausfall der passiven Recheneinheit: Der Ausfall einer passiven Cotask kann nicht unmittelbar erkannt werden. Die in Abschnitt 4.5 beschriebenen Möglichkeiten zur Fehlererkennung bestehen jedoch weiterhin.

Nachdem die Diagnosestelle den Fehler lokalisiert hat, legt sie eine alternative Recheneinheit fest, z. B. RE3, auf der die ausgefallene Cotask zusätzlich aufgesetzt werden soll. Sobald die Costask ($T1(3)$) auf dieser Recheneinheit gestartet wurde, informiert die Diagnosestelle den Sensor und den Aktor über den zusätzlichen Empfänger. Sensor und Aktor senden ihre Nachrichten nun an RE1, RE2 und zusätzlich an RE3.

(Re-)Integration einer Recheneinheit: Es ist wünschenswert, dass bei der (Re-)Integration einer Cotask keine unnötigen Pakete versendet werden. Daher schalten sich (re-)integrierte Cotasks von Anfang an auf passiv. Allerdings kann nicht immer garantiert werden, dass eine Cotask, die sich in den Verbund reintegriert, auch Kenntnis über ihren Ausfall besitzt und deshalb sofort mit dem Senden von Stellwerten beginnt, oder ggf. aus ihrer Sicht fortfährt. Dies stellt kein Problem dar, da der Aktor die Stellwerte dieser Cotask ignoriert. Mit der nächsten empfangenen Statusnachricht schaltet sich dann auch die reintegrierte Cotask auf passiv.

4.7.4 Vollständige Migration

Bei der vollständigen Migration existieren auf der Ziel-Recheneinheit keinerlei Daten der zu übernehmenden Task. In diesem Fall müssen sowohl das Task-Binary, ggf. die virtuelle Domäne selbst und die zum Abgleich und Betrieb notwendigen Eingangs- und Zustandsdaten nachgeladen werden. Es muss vorab sichergestellt sein, dass die neue Task auf dem Zielsystem ohne Verletzung von Zeitbedingungen laufen kann.

Folgende Schritte sind notwendig:

1. Ggf. Anfordern des Binarys der virtuellen Domäne.
2. Anfordern des Task-Binarys.
3. Speicher der neuen Cotask zuweisen.
4. Cotask starten.

5. Anfordern von notwendigen (historischen und aktuellen) Eingangswerten.
6. Anfordern der inneren Zustandsdaten.
7. Abgleich des inneren Zustands.
8. Eingliederung in den Cotask-Verbund als aktive Cotask.

Das Anfordern der Binarys wird vom Slave der Softwarekomponente „Systemkonfigurator“ durchgeführt. Das Starten der Cotask kann mit dem Anfordern der Eingangsdaten bzw. der inneren Zustandsdaten parallel angestoßen werden.

4.8 Zeitliche Betrachtungen

Die benötigte Zeit bis zur Wiedereingliederung hängt maßgeblich von der Anzahl der zu übertragenden Datenpakete und der Entfernung der benötigten Daten ab: Je mehr Switches durchlaufen werden müssen, desto länger ist die maximale Übertragungsdauer δ_{WCTD}. Die Priorität der Pakete ist die niedrigste verfügbare aus der harten Echtzeitklasse. Dadurch ist gewährleistet, dass keine hochprioren Pakete unnötig verzögert werden.

4.8.1 Datenübertragung

Die Berechnung der Übertragungsdauer im schlimmsten Fall erfolgt mit den im vorherigen Kapitel 3 bestimmten Gleichungen. Dazu muss zunächst die Anzahl der Pakete bestimmt werden. Die maximale Payload des verwendeten Standard-Ethernet-Frames beträgt $b_{\text{max}} = 1500\,\text{Byte}$. Die zu übertragende Datenmenge für Task und virtuelle Domäne sei b_{Binary}, die Datenmenge für die Eingangswerte und den inneren Zustand b_{AppData}.

Gemäß Gleichung 3.6 ergibt sich:

$$\delta_{\text{WCTD}} = \left(\left\lceil \frac{b_{\text{Binary}}}{b_{\text{max}}} \right\rceil + \left\lceil \frac{b_{\text{AppData}}}{b_{\text{max}}} \right\rceil \right) \cdot \left(\sum_{\forall s \in \text{Pfad}} \delta_{Hi}^{s} + \delta_T \right) \tag{4.1}$$

Es gilt

$$\delta_T = \delta_{\text{max}} \text{ für } \left(\left\lfloor \frac{b_{\text{Binary}}}{b_{\text{max}}} \right\rfloor + \left\lfloor \frac{b_{\text{AppData}}}{b_{\text{max}}} \right\rfloor \right) \text{ Übertragungen.}$$

Für die restlichen zwei Übertragungen ist die Übertragungszeit δ_T an die restliche zu übertragende Datenmenge ($(b_{\text{Binary}}$ bzw. $b_{\text{AppData}})$ modulo b_{max}) anzupassen.

Um die Validität der empfangenen Daten zu überprüfen, sollte ein Prüfsummenabgleich mit der dritten Cotask, falls vorhanden, durchgeführt werden. Alternativ muss auch die Diagnosestelle diese Information bereitstellen.

4.8.2 Teilausfall einer Recheneinheit

Bei einem Teilausfall einer Recheneinheit, d. h. nur eine virtuelle Domäne oder Task weist ein Fehlverhalten auf, kann in den meisten Fällen davon ausgegangen werden, dass alle Binarys und ein Großteil der Applikationsdaten lokal und konsistent vorhanden sind. Für den Fall, dass die Diagnosestelle eine vollständige Migration veranlasst, ist es sinnvoll, auf eine benachbarte Recheneinheit zu wechseln, die wenn möglich auf dem vorgegebenen Wirkketten-Pfad (siehe Kapitel 5) liegt. Dadurch verkürzt sich die Übertragungszeit auf nur einen Hop:

$$\delta_{\text{WCTD}} = \left(\left\lceil \frac{b_{\text{Binary}}}{b_{\text{max}}} \right\rceil + \left\lceil \frac{b_{\text{AppData}}}{b_{\text{max}}} \right\rceil \right) \cdot (\delta^s_{Hi} + \delta_T) \tag{4.2}$$

Die Berechnung von δ_T für die einzelnen Übertragungen erfolgt analog zu oben.

Für eine Konsistenzprüfung der Applikationsdaten bietet sich eine Prüfsumme an: Die aktiven und die migrierte Cotask berechnen eine Prüfsumme über die angefragten Applikationsdaten. Stimmt diese nicht überein, müssen die Daten von den aktiven Cotasks an die migrierte übertragen werden.

Da die Zuteilung für diesen Fall immer nach dem gleichen Muster erfolgt, kann zur Designzeit ein Nachweis für alle Teilausfälle durchgeführt werden. Stellt sich dabei heraus, dass ein Teilausfall nicht mit dem vorgegebenen Algorithmus zu lösen ist, muss die Allokation der Tasks für den fehlerfreien Fall angepasst werden.

4.8.3 Gesamtausfall einer Recheneinheit

In diesem Fall können die Binarys nicht mehr von der fehlerhaften Recheneinheit bezogen werden. Zudem müssen sämtliche sicherheitskritischen Funktionen gleichzeitig auf andere Recheneinheiten verlagert werden. Mit Gleichung 4.1 lässt sich die maximale Verzögerung für den Fall der vollständigen Migration bestimmen.

Ein wichtiges Kriterium bei der Rekonfiguration des Systems ist der schnelle und deterministische Abschluss. Dies wird im vorliegenden System wie folgt erreicht. Bereits bei der Zuteilung für den fehlerfreien Fall wird darauf geachtet, dass im Falle des schwerwiegendsten Einfachfehlers, dem Ausfall einer Recheneinheit, genügend Ressourcen zur Verfügung stehen, um die ausgefallenen Tasks aufnehmen zu können, ohne dass das restliche System rekonfiguriert werden muss.

Diese Berechnung kann nicht online durchgeführt werden. Das bedeutet, dass für jeden Gesamtausfall offline eine Zuordnungstabelle erstellt werden muss, in der die neue Konfiguration abgelegt ist.

4.9 Bewertung

Die Verwendung der Dreifach-Redundanz mit Voting sorgt dafür, dass das System auch im Einfachfehlerfall weiterhin funktioniert. Die Reduzierung der Taskklasse ist allerdings nicht immer zulässig. Daher wird im Anschluss an einen Fehler der ursprüngliche Redundanzgrad mit den beschriebenen Maßnahmen wieder hergestellt.

Wie lange eine Task mit einem geringeren als den spezifizierten Redundanzgrad arbeiten darf ist abhängig von der Applikation. Es ist im Einzelfall zu prüfen, ob die Zeit für eine vollständige Migration ausreichend kurz ist, oder ob andere Maßnahmen wie Cold- oder Hot-Standby verwendet werden müssen, um beispielsweise SIL- oder ASIL-Vorgaben zu erfüllen. Die Variante „Hot-Standby mit reduzierter Reaktionszeit“ wurde in einem Versuchsträger erfolgreich umgesetzt.

Das Zurücksetzen einer Task als Fehlerbehebungsmaßnahme ist zu vermeiden, da der Erfolg der Reparatur nicht garantiert werden kann. Für die Berechnung der Zeit bis zur Wiedereingliederung darf demnach nicht nur die Zeit für das Zurücksetzen berücksichtigt werden, sondern zusätzlich die Zeit, die für eine weitere Maßnahme notwendig ist, falls das Zurücksetzen fehlschlägt.

5 Allokationsmethoden für Echtzeit-Tasks

In diesem Kapitel werden Zuteilungsheuristiken für das vorliegende System erarbeitet. Zunächst erfolgt die Allokation für den fehlerfreien Fall unter den Randbedingungen fehlertolerante Verarbeitung und knotendisjunkte Kommunikation. Anschließend wird der Einfachfehlerfall betrachtet. Randbedingung hierbei ist eine möglichst kurze Rekonfigurationszeit, um das System schnellstmöglich in den gewünschten Zustand zu versetzen. Die Betrachtung erfolgt nur für HRT-Tasks. Die Auswertungen der hier vorgestellten Varianten erfolgt in Kapitel 6.

5.1 Verwendete Modellierungen

5.1.1 Task- und Wirkkettenmodell

Für die Allokation werden die Wirkketten W herangezogen. W beschreibt die logische Kommunikationsbeziehung, ausgehend von mindestens einer Quelle (Sensoren oder Recheneinheiten) über eine Verarbeitung (Recheneinheit) zu mindestens einer Senke (Aktoren oder Recheneinheiten). $P(W)$ beschreibt einen tatsächlichen Pfad, der für eine Wirkkette im Kommunikationsnetz verwendet wird.

Da der Typ der Quelle oder Senke meist unerheblich ist, und diese an Switches angeschlossen sind, werden Sensoren, Aktoren, Recheneinheiten und Switches für die Allokation zusammengefasst und als *Knoten* bezeichnet. Eine (physikalische) Kommunikationsverbindung zwischen zwei Knoten wird auch als *Kante* bezeichnet.

Zur Vereinfachung gebe es genau eine Quelle, die eine Wirkkette bzw. deren Verarbeitungsteil anstößt. Nur diese Wirkkette wird für die Allokation herangezogen. Die Übertragungen der übrigen zugehörigen Quellen werden nach der Allokation auf den Kommunikationspfaden berücksichtigt. Gleiches gilt für die Senken. Der Anschlusspunkt der Sensoren und Aktoren und damit die Position von Quellen und Senken sei vorgegeben und nicht veränderbar.

Jede Wirkkette besitzt eine Ende-zu-Ende Deadline. Es wird davon ausgegangen, dass sie höchstens so groß ist wie die Periode, mit der sie angestoßen wird. Ggf. müssen Rechenzeiten in Quellen und Senken berücksichtigt werden. Die Ende-zu-Ende Deadline wird aufgeteilt in den Kommunikationsanteil von der Quelle zur verarbeitenden Recheneinheit, die Verarbeitung selbst und den Kommunikationsanteil von der verarbeitenden Recheneinheit zur Senke.

Für die Kommunikation werden Paketgröße und Priorität angegeben. Die Berechnung der Übertragungszeiten erfolgt gemäß Kapitel 3.

Für die Tasks, die auf den Recheneinheiten verarbeitet werden, werden WCET und Deadline angegeben. Die Deadline wird berechnet aus der Ende-zu-Ende Deadline abzüglich der Übertragungszeiten im schlechtesten Fall. Zusätzlich wird für die Task deren Taskklasse bzw. Redundanzgrad angegeben. Für die Berechnung, ob eine Recheneinheit ausreichend freie Ressourcen besitzt, werden verschiedene Nachweisverfahren verwendet.

5.1.2 Datenmodell für Knoten und Kanten

In der Graphentheorie werden zur Beschreibung der Zusammenhänge zwischen Knoten und Kanten meist Adjazenz- oder Inzidenzmatrizen verwendet. Nachfolgend werden davon abweichend zwei Darstellungsweisen vorgeschlagen, die die physikalische Anordnung der Knoten zueinander berücksichtigt.

Ein Merkmal der in Kapitel 2 beschriebenen Systemstruktur ist, dass jeder Knoten mit genau drei weiteren direkt verbunden ist. Dadurch ergibt sich ein physikalischer Zusammenhang, der durch zwei Ringe mit Zwischenverbindungen beschrieben werden kann (siehe Bild 5.1: Ring 1 bestehend aus den Recheneinheiten mit Zeilenindex 1, Ring 2 bestehend aus den Recheneinheiten mit Zeilenindex 2; Zwischenverbindungen verbinden die Recheneinheiten mit gleichem Spaltenindex). Diesen physikalischen Zusammenhang kann man wie folgt durch eine Matrix nachbilden: Die zwei Ringe werden so abgeflacht, dass ein Ring pro Zeile dargestellt wird. Jede Recheneinheit entspricht einem Eintrag in der Matrix M. Jede Recheneinheit besitzt zusätzlich Metadaten, die für die Allokation benötigt werden, wie z. B. die Auslastung U.

Für die weitere Betrachtung gelten folgende Schreibweisen und Konventionen:

- $\%$ ist der Modulo-Operator.
- Zur besseren Lesbarkeit werden die Indizes hier *nicht* tiefgestellt sondern in Klammern hinter den Bezeichner eingetragen.
- Die Indizes beginnen bei 1.
- Die Indizes geben die Position in der Matrix M an.

Implizite Berücksichtigung der Verbindungswege

Die Kommunikationsverbindungen ergeben sich nach folgenden Regeln: Ein Knoten ist mit seinem rechten und linken Nachbarn über eine Kante verbunden. Wenn es keinen rechten oder linken Nachbarn gibt, wird der erste bzw. letzte Eintrag einer Zeile in der Matrix als Nachbar angenommen. Der dritte Nachbar befindet sich für Knoten in der oberen Zeile unterhalb, und für Knoten in der unteren Zeile oberhalb (Bild 5.1).

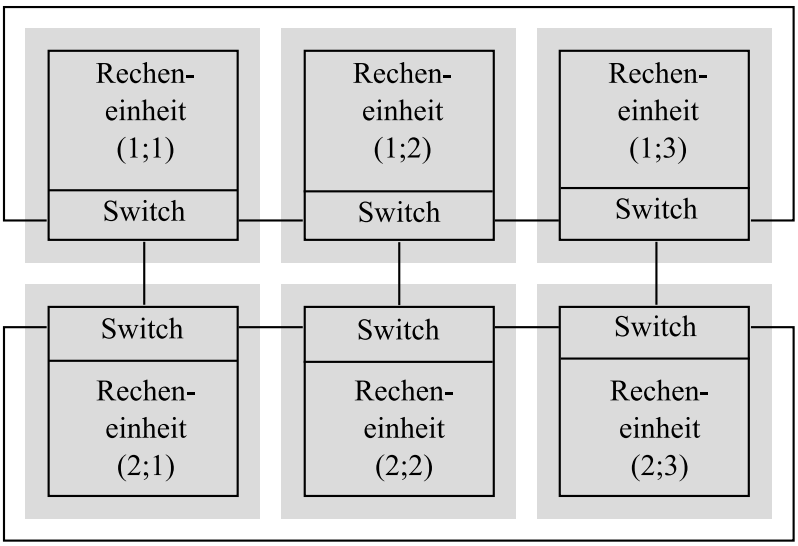


Bild 5.1: Matrix der physikalischen Anordnung der Recheneinheiten

Formal lässt sich dies wie folgt darstellen: Bei n Recheneinheiten gilt, eine Recheneinheit $M(i,j)$ besitzt eine direkte Verbindung mit

- dem *rechten Nachbarn* $M(i,j+1)|j < n/2$, sonst $M(i,1)$,
  oder $M(i,j\%(n/2)+1)$,
- dem *linken Nachbarn* $M(i,j-1)|j > 1$, sonst $M(i,n/2)$,
  oder $M(i,(n/2+j-2)\%(n/2)+1$ und
- dem *unteren* bzw. *oberen Nachbarn* für $i = 1$: $M(i+1,j)$, bzw. für $i = 2$: $M(i-1,j)$,
  oder $M(i\%2+1,j)$.

Diese Schreibweise ist sehr effizient, aber nicht für die Markierung von ausgefallenen Kanten geeignet, wie sie später für die Allokation noch benötigt wird. Deshalb wird folgende Erweiterung durchgeführt.

Explizite Angabe der Verbindungswege

Jede Kante bekommt ebenfalls einen Eintrag in der Matrix (Bild 5.2). Damit lassen sich zum einen ausgefallene Kanten markieren, zum anderen können die Kanten ebenfalls mit Metadaten über die Auslastung annotiert werden, um darauf basierend eine Optimierung durchzuführen.

Die Positionen der Matrix sind entweder mit 0=‚nicht existent‘ oder mit 1=‚existent‘ besetzt. Die Metadaten befinden sich in der dritten Dimension der Matrix.

Bei n Recheneinheiten ergibt sich eine $3 \times n$ Matrix.

- Knoten befinden sich in den Zellen $M(i,j)|i\%2 = 1, j\%2 = 1$,

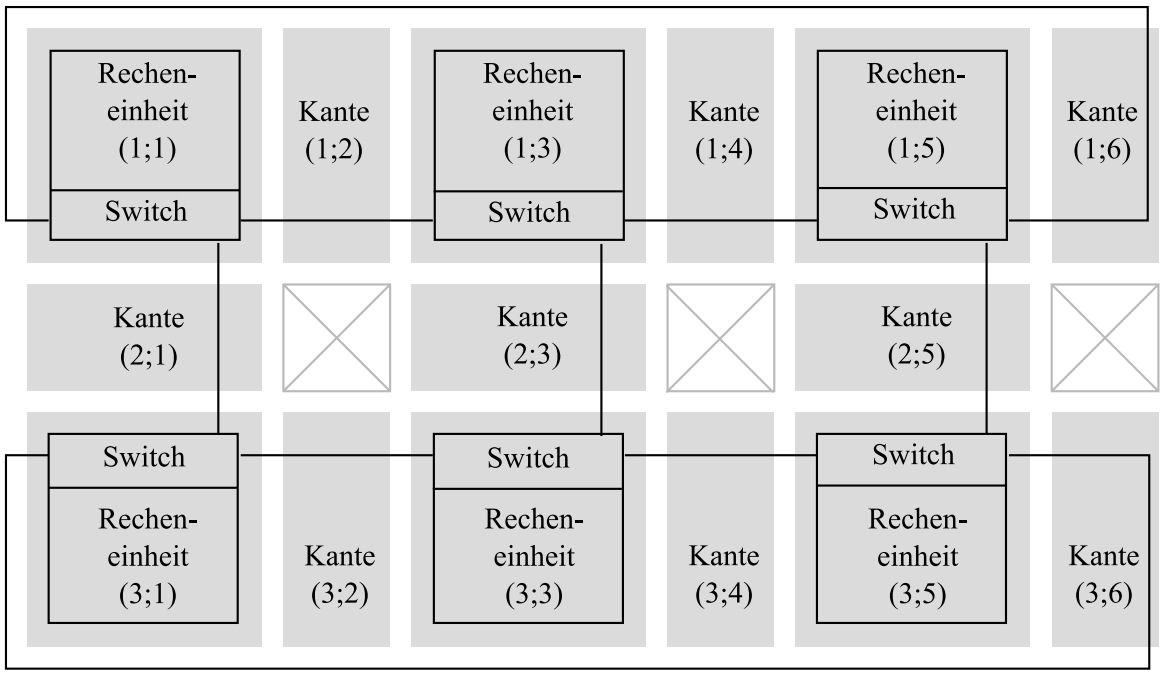


Bild 5.2: Matrix der physikalischen Anordnung mit Kanten

- Kanten befinden sich in den Zellen $M(i,j)|i\%2 = 1, j\%2 = 0$ und $M(i,j)|i\%2 = 0, j\%2 = 1$,
- die übrigen Zellen $M(i,j)|i\%2 = 0, j\%2 = 0$ sind ungültig, d. h. nicht belegt.

Die Nachbarschaft von Recheneinheit $M(i,j)$ ist demnach wie folgt definiert:

- *Rechter Nachbar:* $M(i,(j+1)\%n+1)$,
- *Linker Nachbar:* $M(i,(n+j-3)\%n+1)$ und
- *Unterer* bzw. *oberer Nachbar:* $M((i+1)\%4+1,j)$.

Folgende Fehlerfälle werden modelliert:

1. Ausfall einer Recheneinheit: Der entsprechende Eintrag in M ist 0.
2. Ausfall einer Kante: Der entsprechende Eintrag in M ist 0.
3. Ausfall eines Switches: Die Einträge in M der Kanten, die mit dem ausgefallenen Switch verbunden sind, sind 0.

Passend dazu werden die Operatoren $\oplus$ und $\ominus$ eingeführt. Damit kann man um x Recheneinheiten nach rechts oder links springen. $\oplus$ und $\ominus$ sind nur für j gültig. Die intuitiven Definitionen

$$j \oplus x := (j + 2x - 1)\%n + 1 \text{ und}$$

$$j \ominus x := (n + j - 2x - 1)\%n + 1$$

können nur für den fehlerfreien Fall verwendet werden. Für den Fehlerfall werden die Operatoren ergänzt: Im Falle einer defekten Kante entlang des Pfades, ist das Ergebnis 0 (ungültiger Index).

Dies lässt sich in Pseudocode folgendermaßen darstellen:

```
j ⊕ x:
   for ( cnt=0; cnt<x; cnt++ )
   {
      // ist die Kante rechts ausgefallen? → Abbruch
      if !(M(i, j + 1))
         return 0
      j = (j + 1)%n + 1
   }
   return j

j ⊖ x:
   for ( cnt=0; cnt<x; cnt++ )
   {
      // ist die Kante links ausgefallen? → Abbruch
   if !(M(i, (n + j − 3)%n + 2))
         return 0
   j = (n + j − 3)%n + 1
   }
   return j
```

Weiter werden unäre Operatoren ($+$ und $-$) definiert, um die Indizes der jeweiligen Nachbarknoten zu erreichen. Der Rückgabewert ist der Index des entsprechenden Nachbarn. Existiert kein Nachbar, weil eine Kante fehlt, wird stattdessen der ungültige Index $(0, 0)$ zurückgegeben. In Pseudocode:

Rechter Nachbar:

```
(i, j+):
   // ist die Kante rechts ausgefallen? → Abbruch
   if !(M(i, j + 1))
      return (0, 0)
   // Index für rechten Nachbarn zurückgeben
   j = (j + 1)%n + 1
   return (i, j)
```

Linker Nachbar:

```
(i, j−):
   // ist die Kante links ausgefallen? → Abbruch
   if  !( M(i, (n + j − 3)%n + 2) )
      return  (0, 0)
   // Index für linken Nachbarn zurückgeben
   j = (n + j − 3)%n + 1
   return  (i, j)
```

Unterer bzw. oberer Nachbar:

```
(i+, j):
   // ist die Zwischen-Kante ausgefallen? → Abbruch
   if  !( M(2, j) )
      return  (0, 0)
   // Index für oberen/unteren Nachbarn berechnen
   i = (i + 1)%4 + 1
   return  (i, j)
```

Durch diese Schreibweise lassen sich nun auch Richtungen abbilden: „Ein Paket läuft rechtsherum“ bedeutet beispielsweise, dass das Paket immer zu seinem rechten Nachbarn geschickt wird.

5.2 Heuristische Pfadfindung

Wie in Kapitel 2 beschrieben kann es nur genau drei knotendisjunkte Pfade gleichzeitig zwischen zwei Knoten geben, da pro Recheneinheit nur drei Anschlüsse verfügbar sind (Bild 5.3). Für diese Anschlussart ist die knotendisjunkte Pfadsuche mit der kantendisjunkten gleichwertig: Für einen durchlaufenen Knoten, der weder Quell- noch Zielknoten ist, werden die eingehende und die ausgehende Kante gelöscht; es bleibt also nur eine Kante übrig, die den Knoten mit dem System verbindet, d. h. der Knoten kann nicht mehr als weiterleitender Knoten genutzt werden.

Ziele der heuristischen Pfadfindung sind:

1. Möglichst effizient alle o. g. Pfade zu finden, bzw. falls die Anforderung an Kommunikationsredundanz geringer ist, die entsprechende Anzahl an kürzesten Pfaden zurückzuliefern.

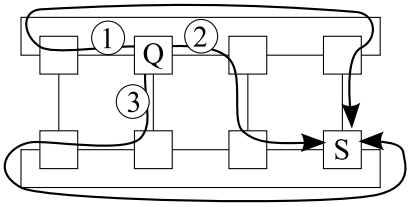


Bild 5.3: Knotendisjunkte Pfade

2. Die Heuristik sollte so ausgelegt sein, dass der Pfad schrittweise verlängert werden kann. Dadurch ist es möglich, die Pfadsuche in die Allokationsheuristik direkt einfließen zu lassen.

Im Folgenden wird eine Heuristik entworfen, die diese Kriterien erfüllt.

Pfad 1: Gehe vom Quellknoten Q aus solange nach links $(i, j \ominus 1)$, bis der Index j mit dem des Zielknoten S übereinstimmt, aber nur solange ein Knoten nicht bereits Teil eines anderen Pfads ist. Falls Index i noch nicht mit dem von S übereinstimmt: $(i+, j)$. Gehe weiter solange nach links $(i, j \ominus 1)$, bis der Index j mit dem des Zielknoten S übereinstimmt.

Pfad 2: Gehe vom Quellknoten Q aus solange nach rechts $(i, j \oplus 1)$, bis der Index j mit dem Zielknoten S übereinstimmt, aber nur solange ein Knoten nicht bereits Teil eines anderen Pfads ist. Falls Index i noch nicht mit dem von S übereinstimmt, $(i+, j)$. Gehe weiter solange nach rechts $(i, j \oplus 1)$, bis der Index j mit dem des Zielknoten S übereinstimmt.

Pfad 3: Gehe vom Quellknoten Q aus zum Nachbarn unter- bzw. oberhalb $(i+, j)$. Gehe solange nach links $(i, j \ominus 1)$, bis der Index j mit dem des Zielknoten S übereinstimmt. Falls Index i noch nicht mit dem von S übereinstimmt, $(i+, j)$.

Es gibt zwei Anordnungen zwischen Quelle und Senke, bei denen der Algorithmus nicht funktioniert.

1. Der Zielknoten S liegt genau unterhalb oder oberhalb von Quellknoten Q: $\mathrm{S}(i_S, j_S) = \mathrm{Q}(i_Q+, j_Q)$
2. Der Zielknotenindex j_S ist j_Q+ des Quellknotens: $\mathrm{S}(i_S, j_S) = \mathrm{Q}(i_Q, j_Q+)$ oder $\mathrm{S}(i_S, j_S) = \mathrm{Q}(i_Q+, j_Q+)$

Für den ersten Fall ergeben sich immer folgende drei Pfade:

Pfad 1a: $(i_Q, j_Q) \rightarrow (i_Q, j_Q+) \rightarrow (i_Q+, j_Q+) \rightarrow (i_Q+, j_Q) = (i_S, j_S)$

Pfad 2a: $(i_Q, j_Q) \rightarrow (i_Q, j_Q-) \rightarrow (i_Q+, j_Q-) \rightarrow (i_Q+, j_Q) = (i_S, j_S)$

Pfad 3a: $(i_Q, j_Q) \rightarrow (i_Q+, j_Q) = (i_S, j_S)$

Der zweite Fall kann automatisch umgangen werden, wenn in Anhängigkeit der Anordnung entweder „links herum“ zuerst (Pfad 1) oder „rechts herum“ zuerst (Pfad 2) ausgeführt wird. Als Kriterium gilt der Abstand zwischen Quelle und Senke. Pfad 3 ändert seine Suchrichtung so, dass sie mit der ersten ausgeführten Pfadsuche übereinstimmt. Damit die Heuristik auch bei

weniger als drei gewünschten Pfaden möglichst den oder die kürzesten Pfade liefert, wird die Suchreihenfolge geändert:

1. Falls die Senke rechts näher liegt: Pfad 2, sonst Pfad 1.

2. Pfad 3 mit Suchrichtung des unter Punkt 1 gewählten Pfades; die Suchrichtung wird dabei an die Pfadnummer angehängt, also: rechts (3r) bzw. links (3l).

3. Der Pfad, der unter Punkt 1 nicht gewählt wurde.

Damit liefert die Heuristik bis auf zwei Fälle immer den oder die kürzesten Pfade: Falls sich jedoch Quelle und Senke auf dem gleichen Ring ($i_Q == i_S$) in einem Abstand von $n/2$ oder $n/2 + 1$ zueinander befinden, liefert die Heuristik als zweiten Pfad den längsten Pfad zurück; dieser ist um maximal zwei Hops länger als der mittellange Pfad. Eine zusätzliche Abfrage umgeht diese Fälle.

```
if ( jQ == jS )
  Pfade: 3a, 1a, 2a
else
{
  if ( (iQ == iS) ∧ ((jQ + n/2 − 2)%n + 1 ≤ jS ≤ (jQ + n/2 − 2)%n + 3) )
    except=true

  if ( (jQ ≤ n/2) ∧ (jQ < jS < jQ + n/2)∨
       (jQ ≥ n/2) ∧ !(jQ − n/2 < jS < jQ) )
    // zuerst rechts herum
    if (except)
      Pfade: 2, 1, 3r
    else
      Pfade: 2, 3r, 1

  else
    // zuerst links herum
    if (except)
      Pfade: 1, 2, 3l
    else
      Pfade: 1, 3l, 2
}
```

Bild 5.4 zeigt die Rechenzeiten der Pfadsuche für die Heuristik und für eine Standardimplementierung eines kantendisjunkten Dijkstra-Algorithmus mit gleichgewichteten Kanten. Der Geschwindigkeitsvorteil der Heuristik ist klar ersichtlich.

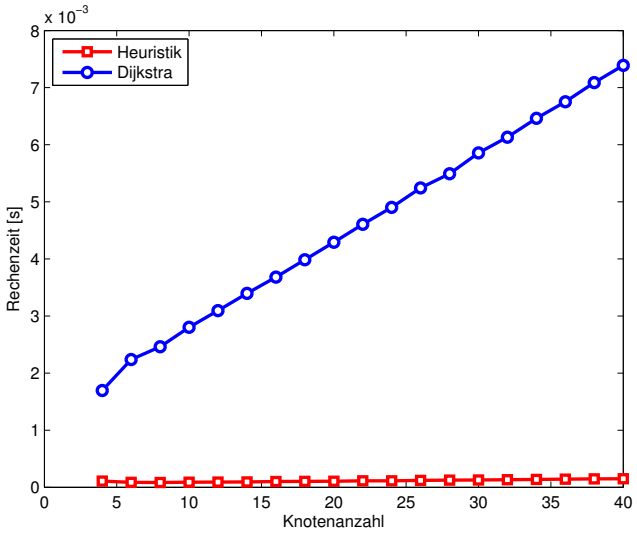


Bild 5.4: Vergleich der Rechenzeiten zur Pfadsuche

5.3 Randbedingungen der Allokation

Bei der Taskzuteilung gibt es sowohl Randbedingungen, deren Einhaltung verpflichtend ist, als auch solche, die die Ausnutzung der verfügbaren Ressourcen verbessern. Im Fokus dieser Arbeit stehen dabei weniger die Verbesserungen z. B. der Reaktionsgeschwindigkeit für nicht sicherheitskritische Applikationen. Vielmehr dienen die zusätzlich eingeführten Randbedingungen primär der Verbesserung der Reaktionszeit der Echtzeit-Tasks und zielen auf die Durchführbarkeit einer Rekonfiguration.

Verpflichtende Randbedingungen sind:

- Alle Zeitbedingungen müssen eingehalten werden; das gilt sowohl für die Einhaltung der Deadlines auf den Recheneinheiten als auch der maximalen Übertragungsdauer im Kommunikationsnetz.
- Die Redundanzanforderung aus den Taskklassen müssen eingehalten werden.
- Cotasks müssen auf unterschiedlichen Recheneinheiten platziert werden.
- Die Kommunikationswege der Wirkketten von Cotasks müssen knotendisjunkt sein. Ist dies nicht möglich, müssen alle Wirkketten über redundante Pfade geleitet werden.

Weitere Randbedingungen zur Verbesserung der Effizienz sind:

- Möglichst gleichmäßige Belastung aller Recheneinheiten.
- Genügend freie Ressourcen, um bei einem Ausfall einer Recheneinheit weitere Tasks aufnehmen zu können.
- Möglichst kurze Rekonfigurationszeit im Fehlerfall.
- Minimierung der Kommunikation (möglichst kurze Pfade).

5.4 Feasibility Tests

Zum einen kann der exakte Echtzeitnachweis gemäß Kapitel 3 verwendet werden. Da bei jeder neu hinzugekommenen Task der Nachweis erneut für alle Tasks durchgeführt werden muss, steigt der Rechenaufwand nichtlinear an. Für eine Überprüfung zur Laufzeit ist der exakte Nachweis daher i. d. R. nicht oder nur bedingt geeignet.

Besser eignen sich die Nachweisverfahren, die eine Überprüfung nur anhand der neu hinzugekommenen Tasks durchführen. Sie skalieren linear mit der Anzahl der Tasks, sind aber weniger genau, das bedeutet, dass weniger Tasks pro Recheneinheit zugelassen werden als mit dem exakten Verfahren. Ein Beispiel für einen nicht-exakten Test liefert Devi in [36]. Wenn, wie im vorliegenden Fall, die Tasks nicht nach aufsteigenden Deadlines sortiert sind, liefert der Test sogar schlechtere Ergebnisse als der einfachere Density-Test ($\sum_{i=1}^{n} \frac{c_i}{d_i} \leq \rho_{\max} = 1$) [68].

Masrur gibt in [68] eine weitere Methode für einen nicht-exakten Nachweis an, der in jedem Fall bessere Ergebnisse liefert als der Density-Test und Devis Test – unabhängig von der Sortierung der Taskliste. Dieser Nachweis hat die gleiche Komplexität wie Devis-Test und kann daher auch zur Rekonfiguration im laufenden Betrieb eingesetzt werden.

Im Folgenden wird beispielhaft der Density-Test verwendet. Um genügend Spielraum für eine spätere Umverteilung der Tasks im Fehlerfall zu lassen, wird die maximal erlaubte Density $\rho_{\max}$ pro Recheneinheit für die Zuteilung im fehlerfreien Fall reduziert. Die Auswirkungen verschiedener $\rho_{\max}$ im fehlerfreien Fall auf die Zuteilbarkeit im Fehlerfall sind in Kapitel 6 dargestellt.

5.5 Zuteilungsheuristiken

Es werden Zuteilungsmethoden auf Basis ausgewählter Bin-Packing Algorithmen untersucht. Zusätzliche Randbedingungen werden eingeführt, um die Effizienz des Gesamtsystems gegenüber den nicht modifizierten Algorithmen zu verbessern. Mit jedem Zuteilungsschritt (d. h. eine Task wird einer Recheneinheit zugeteilt) wird auch die Überprüfung gemäß Abschnitt 5.4 durchgeführt.

5.5.1 Auswahl der Heuristik

Grundlage der Zuteilungsverfahren bilden die folgenden ausgewählten Bin-Packing Algorithmen. Abweichend vom Standardverfahren dürfen Tasks nur auf die Recheneinheiten platziert werden, die auf einem bestimmten Pfad liegen. Dadurch wird die Forderung nach knotendisjunkten Pfaden für Cotasks erfüllt. Für alle Verfahren gilt: Falls die gerade zuzuteilende Task auf der letzten möglichen Recheneinheit keinen Platz findet, gilt das gesamte Taskset als nicht zuteilbar (engl. *infeasible*).

First Fit: Vom Quellknoten aus wird die erste Recheneinheit des ersten Pfades verwendet, um die Task abzulegen. Ist auf dieser Recheneinheit kein Platz mehr für die aktuelle Task vorhanden, wird die nächste auf dem Pfad gelegene Recheneinheit verwendet. Für Tasks aus einer höheren Taskklasse werden diese Schritte für die Cotasks auf anderen Pfaden wiederholt.

Best Fit: Entlang des Pfades vom Quellknoten bis zum Zielknoten wird die Task derjenigen Recheneinheit zugeteilt, die nach der Zuteilung die geringste freie Rechenkapazität besitzt, d. h. $\mathrm{RE}_j|j : \max_{\forall j} \rho_j$.

Worst Fit: Entlang des Pfades vom Quellknoten bis zum Zielknoten wird die Task derjenigen Recheneinheit zugeteilt, die die größte freie Rechenkapazität besitzt, d. h. $\mathrm{RE}_j|j : \min_{\forall j} \rho_j$.

Alle Verfahren liefern die besten Ergebnisse, wenn sie auf eine nach ansteigender Deadline bzw. kleiner werdenden Density sortierten Taskliste angewendet werden [34].

5.5.2 Gegenüberstellung

First Fit und *Best Fit* eignen sich besonders dann, wenn die Anzahl an Recheneinheiten möglichst gering gehalten werden soll [68]. Prinzipiell ist dies auch im vorliegenden System gewünscht; da jedoch die Rechenleistung hauptsächlich durch Anwendungen mit geringen oder ohne Zeitanforderungen bestimmt ist, und somit große Kapazitäten pro Recheneinheit verfügbar sind, tendiert *Best Fit* dazu, einzelne Recheneinheiten auszulasten während andere nur wenige Echtzeit-Tasks beherbergen. Bei *First Fit* hängt die Auslastung der einzelnen Recheneinheiten stark von der Entfernung zu den Sensoren ab. Eine Recheneinheit, an der (bzw. an ihrem integrierten Switch) kein Sensor angeschlossen ist, wird erst mit Echtzeit-Tasks belegt, wenn die sensornahen Recheneinheiten ausgelastet sind. Je stärker eine Recheneinheit ausgelastet ist, desto länger ist die durchschnittliche Reaktionszeit einer Task. Des Weiteren müssen im Falle eines Ausfalls sämtliche Tasks migriert werden, d. h. je größer die Anzahl der zu migrierenden Tasks, desto länger dauert die Rekonfiguration.

Daher ist eine möglichst gleichmäßige Auslastung der Recheneinheiten erstrebenswert, wie sie mit dem *Worst Fit* Algorithmus erzeugt wird.

Im Fehlerfall müssen die Tasks der ausgefallenen Recheneinheit auf andere Recheneinheiten platziert werden. Für den Fall, dass auch hier eine Pfadabhängigkeit gefordert wird, eignet sich ebenfalls *Worst Fit* am besten. Die anderen beiden Algorithmen tendieren dazu, einzelne Recheneinheiten stark auszulasten. Dies kann dazu führen, dass manche Tasks nicht mehr entlang des ausgewählten Pfades zugeteilt werden können.

5.5.3 Vorsortierung des Tasksets

Die Wahrscheinlichkeit, ein Taskset zuteilen zu können, steigt, wenn die Heuristiken auf sortierte Task-Listen angewendet werden. Dabei werden die Tasks mit den strengsten Anforderungen

an das System zuerst zugeteilt. Die Schwierigkeit besteht darin, dass sich unterschiedliche Anforderungen oft gegenseitig ausschließen bzw. behindern. Folgende Sortierkriterien bzw. Kombinationen werden verwendet:

- Nach nicht aufsteigender Density (*NID*).
- Nach nicht aufsteigender Density, bei gleicher Density nach nicht aufsteigenden Taskklassen (*NID,NITC*).
- Nach nicht aufsteigenden Taskklassen, bei gleichen Taskklassen nach nicht aufsteigender Density (*NITC,NID*).

Die Sortierung nach nicht aufsteigender Density berücksichtigt die strengsten Zeitbedingungen innerhalb der Recheneinheiten; Redundanzanforderungen werden nicht berücksichtigt. Die Sortierungen *NID,NITC* und *NITC,NID* berücksichtigen zusätzlich die Redundanzanforderungen der Tasks in Form ihrer Taskklasse.

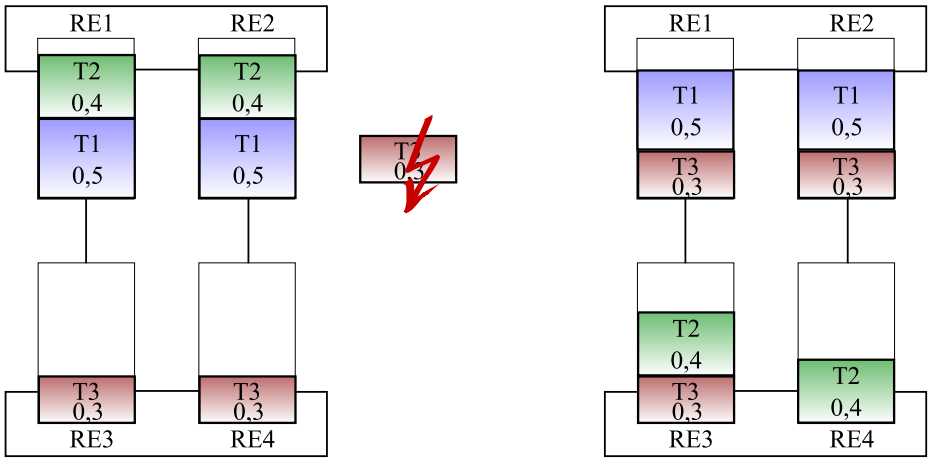
Die Sortiervariante *NID,NITC* ist nahezu identisch mit *NID*, da in realen Systemen kaum mehrere Tasks die gleiche Density besitzen. *NID,NITC* liefert immer ein gleiches oder besseres Allokationsergebnis als die reine Sortierung nach absteigender Density. Deshalb wird auf die Betrachtung der Variante *NID* verzichtet.

Üblicherweise ergibt die Sortierung nach kleiner werdender Density die beste Zuteilung, d. h. die geringste benötigte Anzahl an Recheneinheiten für ein Taskset, bzw. bei fester Anzahl an Recheneinheiten eine höhere Wahrscheinlichkeit, das Taskset überhaupt zuteilen zu können. Dies gilt jedoch nur, wenn eine Task auf jeder beliebigen Recheneinheit platziert werden darf. Zielt man auf eine möglichst kurze Reaktionszeit ab, hängen die zur Verfügung stehenden Recheneinheiten von dem oder den ausgewählten Kommunikationspfaden ab. Der vorgestellte Algorithmus zur Pfadsuche liefert immer zunächst den kürzesten, dann den zweitkürzesten und schließlich den längsten Pfad zurück, so dass bei ungleichmäßiger Verteilung der Anschlusspunkte die Recheneinheiten auf dem kürzesten Pfad schneller ausgelastet sind als die restlichen. Soll nun eine Task mit hohem Redundanzgrad zugeteilt werden, kann die Task unter Umständen auf einem Pfad nicht mehr untergebracht werden, und die Zuteilung schlägt fehl.

Wenn Tasks mit großer Density hauptsächlich mehrfach-redundant ausgelegt werden, verspricht eine Sortierung der Tasks zunächst nach Taskklassen und anschließend nach der Density eine Verbesserung. In diesem Fall können die Tasks geringerer Taskklasse auf einen anderen Pfad umgeleitet werden. Dadurch ergibt sich zwar eine größere Verzögerung durch das Kommunikationsnetz, jedoch steigt die Wahrscheinlichkeit, ein Taskset zuteilen zu können.

Bild 5.5 verdeutlicht beispielhaft die Auswirkung der Sortiervarianten auf das Allokationsergebnis des *Worst Fit* Algorithmus (siehe Abschnitt 5.5.5): Ein Taskset, bestehend aus drei Tasks mit folgenden Parametern, soll auf ein System mit vier Recheneinheiten alloziert werden. Als Start- bzw. Endpunkte der Wirkkette sind hier Recheneinheiten angegeben; tatsächlich ist damit der jeweilige Switch im Verbund aus Recheneinheit und Switch gemeint, an dem intelligente Sensoren bzw. Aktoren an das System angeschlossen sind.

Task	Density	Task-klasse	Start-punkt	End-punkt
T1	0,5	2	RE1	RE2
T2	0,4	2	RE1	RE2
T3	0,3	3	RE1	RE2



(a) NID,NITC (b) NITC,NID

Bild 5.5: Auswirkung der Sortiervarianten auf die Allokation (1).

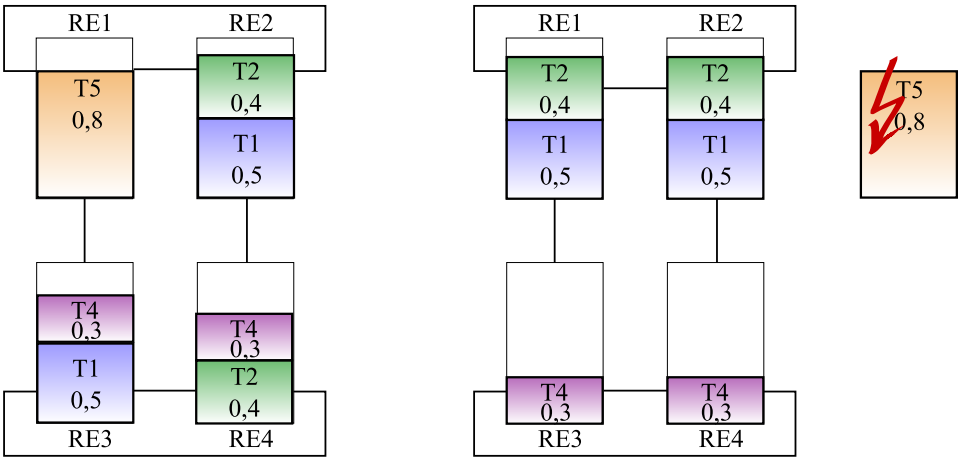
Die Tasks T1 und T2 besitzen eine höhere Density aber eine geringere Taskklasse als T3. Im Falle der Sortiervariante *NID,NITC* schlägt die Zuteilung der dritten Cotask von T3 auf Grund der Redundanzanforderung fehl. Die Variante *NITC,NID* hingegen kann alle (Co-)Tasks platzieren.

Falls es Tasks mit großer Density gibt, die nur einfach ausgelegt werden, kann das Allokationsergebnis bei der Variante *NITC,NID* allerdings auch schlechter ausfallen, wie die Allokation des folgenden, leicht veränderten Tasksets zeigt.

Task	Density	Task-klasse	Start-punkt	End-punkt
T5	**0,8**	**1**	**RE1**	**RE2**
T1	0,5	2	RE1	RE2
T2	0,4	2	RE1	RE2
T4	0,3	**2**	RE1	RE2

Anstelle der Task T3 gibt es nun eine Task T4 mit gleicher Density, aber reduzierter Taskklasse. Zusätzlich muss die Task T5 mit hoher Density, aber geringster Taskklasse alloziert werden. In diesem Taskset ist die Sortiervariante *NID,NITC* erfolgreich, während *NITC,NID* fehlschlägt (Bild 5.6).

Welche der Sortiervarianten besser ist, hängt demnach vom Taskset ab. I. d. R. liefert *NITC,NID* ein besseres Ergebnis, wenn die Sensoren und Aktoren an nur wenigen Switches angeschlos-



(a) NID,NITC (b) NITC,NID

Bild 5.6: Auswirkung der Sortiervarianten auf die Allokation (2).

sen sind. Je gleichmäßiger die Anschlusspunkte auf alle Switches verteilt sind, desto bessere Ergebnisse liefert *NID,NITC*.

5.5.4 Benötigte Meta-Informationen über Tasks

Für die Standardallokationsmethoden sind lediglich die Taskparameter Periode (oder Ereignisstrom), Deadline und WCET notwendig. Dieser Parametersatz – die Meta-Information einer Task – muss ergänzt werden, um die Zuteilung im fehlerfreien Fall wie auch im Fehlerfall mit den beschriebenen Randbedingungen durchführen zu können:

Taskklasse TC_i: Bestimmt den Redundanzgrad einer Task T_i.

Priorität: Gibt die Priorität der Task und der von ihr versendeten Pakete an. Der Parameter wird für die Task nur dann verwendet, wenn auf den Recheneinheiten ein prioritätsbasiertes Scheduling verwendet wird.

Start der Wirkkette W_i^S: Bezeichnet den physikalischen Eintrittspunkt (meist der Anschlusspunkt eines intelligenten Sensors) desjenigen Pakets, das die Bearbeitung der zur Wirkkette gehörigen Task T_i anstößt.

Ende der Wirkkette W_i^E: Bezeichnet den physikalischen Austrittspunkt (meist der Anschlusspunkt eines intelligenten Aktors) des Pakets, das die Task T_i aussendet.

Verwendeter Pfad $P(W)$: Gibt den Pfad an, der für die Wirkkette[1)] W verwendet wurde. Dieser Parameter ist pro Cotask vorhanden und wird während der Zuteilung gesetzt.

[1)] Die Wirkkette beschreibt die logische Kommunikationsbeziehung, der Pfad gibt den tatsächlich genommenen Weg im Kommunikationsnetz an.

5.5.5 Allokationsalgorithmus

Die Zuteilung erfolgt für den fehlerfreien Fall und den Fehlerfall offline. Für die verschiedenen Ausfälle werden Zuteilungstabellen generiert, die die Allokation der ausgefallenen Tasks beinhalten. Es wird von einer vollständigen Migration gemäß Abschnitt 4.7.4 ausgegangen.

Folgende Parameter werden für die Algorithmen verwendet. τ bezeichnet ein unsortiertes, τ^s ein sortiertes Taskset. $\rho_{\max}$ ist der maximal erlaubte Füllgrad einer Recheneinheit (maximal erlaubte Density). $\rho_i = \frac{c_i}{d_i}$ ist die Density der Task T_i. $T_{i(tc)}$ bezeichnet eine Cotask der Task T_i.

Allokation im fehlerfreien Fall

In Pseudocode lässt sich die Allokation im fehlerfreien Fall wie in Bild 5.7 gezeigt darstellen. Der Funktionsaufruf `sort`τ`(`τ`,Variante)` sortiert das übergebene Taskset τ nach der angegebenen Sortiervariante. `berechne_pfade(S,E)` gibt die drei Pfade von `S` nach `E`, sortiert nach aufsteigender Pfadlänge, zurück. `sortRE(`P`,Variante)` sortiert die Recheneinheiten entlang des übergebenen Pfads nach der angegebenen Sortiervariante. Im vorliegenden System werden die Recheneinheiten nach ansteigender Auslastung, repräsentiert durch die Density ρ_{RE}, sortiert (engl. *non decreasing density*, NDD). `cotask_kollision(`T`,RE)` überprüft, ob sich auf der ausgewählten Recheneinheit bereits eine Cotask der Task T befindet. `alloziere(`T`,RE)` teilt die Task T (bzw. eine ihrer Cotasks) einer Recheneinheit `RE` zu und trägt den entsprechenden Pfad in die Metadaten der zugeteilten (Co-)Task T ein.

Um die Wahrscheinlichkeit der Zuteilung zu erhöhen, wird nicht nach dem ersten Fehlversuch abgebrochen, sondern auf den nächst-kürzesten Pfad ausgewichen. Erst wenn die Ressourcen aller Recheneinheiten auf allen Pfaden erschöpft sind, oder nicht genügend Pfade für die Taskklassen zur Verfügung stehen, gilt das Taskset als nicht zuteilbar.

Allokation im Fehlerfall

Beim Ausfall einer Kante müssen die Paketströme, die vorher über die defekte Kante geleitet wurden, auf andere Kanten verlegt werden. Bei den Taskklassen 1 und 2 genügt ein einfaches Umleiten auf einen anderen Pfad. Um die ursprünglichen Redundanzanforderungen bei Taskklasse 3 wieder herzustellen, muss das Paket über zwei disjunkte Pfade durch das Kommunikationsnetz geleitet werden. Für die weitere Betrachtung wird davon ausgegangen, dass im Kommunikationsnetz ausreichend freie Ressourcen vorhanden sind, um die zusätzlichen Pakete ohne Verletzung von Zeitbedingungen weiterleiten zu können. Der Ausfall einer Kante hat somit keine Auswirkung auf die Allokation.

Im Falle eines Ausfalls einer Recheneinheit (`REx`) müssen die fehlenden Tasks auf andere Recheneinheiten verteilt werden. Damit das restliche System möglichst wenig beeinflusst wird, werden ausschließlich die Tasks der ausgefallenen Recheneinheit neu verteilt; die übrige Allokation bleibt bestehen. Hierfür wird die Beschränkung durch $\rho_{\max}$ aus dem fehlerfreien Fall aufgehoben und eine maximale Auslastung pro Recheneinheit von $D = 1$ erlaubt. Die Zuteilung

```
τ^s = sortτ(τ, Variante)
for ( i=1; i≤sizeof(τ); i++ )
{
  P(W_i)[1..3] = berechne_pfade(W_i^S, W_i^E)
  o=0
  for ( tc=1; tc≤TC_i; tc++)
  {
    zugeteilt = false
label:
    o++
    if ( o>3 )
      return(infeasible)
    RE[1..pfadlänge(P(W_i)[o])] = sortRE(P(W_i)[o],NDD)
    for ( j=1; j≤ pfadlänge(P(W_i)[o]); j++ )
      if ( ρ_RE[j] + ρ_i ≤ ρ_max )
      {
        if ( cotask_kollision(T_i,RE[j]) )
          continue
        alloziere(T_i(tc),RE[j])
        ρ_RE[j] = ρ_RE[j] + ρ_i
        zugeteilt = true
        break
      }
    if (zugeteilt)
      break
    else
      goto label
  }
}
```

Bild 5.7: Allokation im fehlerfreien Fall (Pseudocode).

der Tasks erfolgt entlang ihres ursprünglichen Pfades. Damit bleiben die Kommunikationspfade der Wirkketten bestehen; nur die Kommunikation zwischen den Cotasks muss neu geroutet werden. Falls der Abgleich ausschließlich im Aktor geschieht, ändern sich nur für den Rückkanal die Längen der Pfade.

Da auch für höhere Taskklassen immer nur eine der Cotasks verschoben werden muss, spielt die Taskklasse für die Zuteilung keine Rolle. Als Sortiervariante wird daher *NID,NITC* verwendet. Die Randbedingungen für die Zuteilung reduzieren sich im Vergleich zum fehlerfreien Fall: Da nur ein festgelegter Pfad zur Verfügung steht, muss nur noch dieser berücksichtigt werden; alle

anderen Randbedingungen sind identisch. Aus dem in Abschnitt 5.5.2 genannten Grund wird auch hier *Worst Fit* für die Allokation verwendet.

Bild 5.8 zeigt den Zuteilungsalgorithmus für den Ausfall einer Recheneinheit `REx`. Der Algorithmus wird für alle im System vorhandenen Recheneinheiten durchgeführt und das jeweilige Ergebnis in einer Zuordnungstabelle abgelegt. Falls ein Taskset in einem der Fehlerfälle nicht zuteilbar ist, gilt das Taskset für dieses System als nicht zuteilbar. Dies bedeutet, dass die vorhandenen Ressourcen nicht ausreichen, den ursprünglichen Systemzustand nach einem beliebigen Ausfall wieder herzustellen.

Die Funktionen `lese_taskindex(`$\tau_x^s[k]$`)` und `lese_cotaskId(`$\tau_x^s[k]$`)` geben den Index und den Cotask-Identifier einer (unbekannten) Task zurück. `pfad(`$T_{i(tc)}$`)` liefert den bei der Allokation im fehlerfreien Fall gespeicherten Pfad einer Cotask zurück.
`entferneRE(REx,`P_{tmp}`)` löscht eine Recheneinheit aus dem Pfad, so dass auf ihr keine Tasks platziert werden.

```
ρmax = 1
τx = tasks(REx)
τx^s = sortτ(τx,(NID,NITC))
for ( k=1; k ≤ sizeof(τx); k++ )
{
  i = lese_taskindex(τx^s[k])
  tc = lese_cotaskId(τx^s[k])
  Ptmp = pfad(T_i(tc))
  entferneRE(REx, Ptmp)
  RE[1..pfadlänge(Ptmp)] = sortRE(Ptmp,NDD)
  for ( j=1; j ≤ pfadlänge(Ptmp); j++ )
  {
    if ( ρRE[j] + ρi ≤ ρmax )
    {
      if ( cotask_kollision(Ti,RE[j]) )
        continue
      alloziere(T_i(tc),RE[j])
      ρRE[j] = ρRE[j] + ρi
      break
    }
    return(infeasible)
  }
}
```

Bild 5.8: Allokation im Fehlerfall (Pseudocode).

Verbesserung der Wahrscheinlichkeit einer erfolgreichen Zuteilung im Fehlerfall

Die Wahrscheinlichkeit, ein Taskset auch im Fehlerfall zuteilen zu können, kann über eine weitere Randbedingung bei der Zuteilung im fehlerfreien Fall erhöht werden: Wurde der Recheneinheit am Ein- oder Austrittspunkt (W_i^S und W_i^E) bereits eine Cotask zugeteilt, darf die jeweils andere Recheneinheit keine Cotask mehr aufnehmen. Bei Pfaden, auf denen keine oder nur eine Recheneinheit zwischen W_i^S und W_i^E liegt, gibt es im Fehlerfall schlichtweg keine freie Recheneinheit, die die ausgefallene Cotask entlang des Pfades noch aufnehmen könnte, ohne die Randbedingung zu verletzen, dass Cotasks nicht auf identischen Recheneinheiten liegen dürfen.

Gleichzeitig verringert sich durch diese Randbedingung die Chance, eine Zuteilung im fehlerfreien Fall zu finden, falls das System nahe an der maximalen Auslastungsgrenze liegt. Deshalb sollte diese Randbedingung nur für Tasks der Klasse 3 angewendet werden.

Die Überprüfung dieser zusätzlichen Randbedingung kann in die Funktion `cotask_kollision(`T`,RE)` integriert werden: Falls `RE` W_i^S oder W_i^E entspricht, überprüfe auch, ob `RE` gleich W_i^E bzw. W_i^S ist. Hierzu muss die Funktion um einen Aufrufparameter erweitert werden, der die Wirkkette enthält.

6 Simulation der Allokationsvarianten

In diesem Kapitel werden Versuchsreihen durchgeführt, um die unterschiedlichen Allokations- und Sortiervarianten gegenüberzustellen und zu bewerten. Zunächst werden die Parameter der Simulation beschrieben. Es folgen die Ergebnisse der Simulationen und deren Bewertung.

Die Zuteilung erfolgt für jedes Taskset sowohl für den fehlerfreien Fall als auch für den Fehlerfall. Die Simulationen geben einen Überblick, mit welcher Wahrscheinlichkeit, ein Taskset bei definierter Systemauslastung zuteilbar ist. Ein Taskset gilt nur dann als zuteilbar, wenn alle seine Tasks sowohl im fehlerfreien Fall als auch im Fehlerfall, d. h. nach der Rekonfiguration, zuteilbar sind.

6.1 Simulationsumgebung und Eingangsdaten

Eine Systemanordnung mit 8 Recheneinheiten erscheint realistisch und wird deshalb für die Simulation verwendet. Wie in den Kapiteln 2 und 5 beschrieben, ist jede Recheneinheit mit genau drei weiteren verbunden. Die Nummerierung erfolgt fortlaufend „von links nach rechts“ und „von oben nach unten“. Für die Überprüfung der Zuteilbarkeit wird ausschließlich der Density-Test verwendet.

Für die Bezeichnungen werden folgende Abkürzungen verwendet: *FF* steht für *First Fit*, *WF* für *Worst Fit* und *BF* für *Best Fit*. Für die Sortiervarianten wird wie im vorherigen Kapitel *NITC* für „nicht ansteigende Taskklasse“ und *NID* für „nicht ansteigende Density“ bzw. Kombinationen daraus verwendet.

Um das vorgestellte System zu explorieren, sollten sich die generierten Tasksets einerseits an realen Systemen orientieren, andererseits auch zukünftige Funktionen berücksichtigen. Aus diesem Grund werden reale Funktionen im Automobil hinsichtlich ihrer Periode und Auftrittshäufigkeit als Basis für die Tasksets verwendet. Da es derzeit keine mehrfach ausgelegten Tasks gibt, wird für die verschiedenen Funktionen eine Spanne von Taskklassen erlaubt. Die kritischen Funktionen befinden sich eher im kürzeren Periodenbereich, daher sind diesen die höheren Taskklassen zugeordnet. Bei der Generierung eines Tasksets wird die Taskklasse pro Task zufällig im angegebenen Bereich gewählt. Tabelle 6.1 zeigt die zu Grunde liegenden Metadaten.

Die maximale Verzögerung durch das Kommunikationsnetz sowie die Bearbeitung durch die Treiberdomäne sind in d bereits mit einer konstanten Zeit von $0,8\,\text{ms}$ berücksichtigt. Darin enthalten sind zweimal die maximale Übertragungszeit im Kommunikationsnetz mit jeweils

Tabelle 6.1: Metadaten für die Generierung von Wirkketten

$(p;\ d)$ [ms]	Anzahl	Klassen
(2; 1,2)	15	1-3
(5; 4,2)	40	1-3
(10; 9,2)	40	1-3
(20; 19,2)	50	1-3
(40; 39,2)	5	1
(50; 49,2)	70	1-2
(100; 99,2)	70	1-2
(200; 199,2)	70	1
(500; 499,2)	70	1

$100\,\mu$s und zweimal die maximale Verzögerung durch die Bearbeitung des eingehenden Pakets in der Treiberdomäne von jeweils $300\,\mu$s. Nach der Allokation muss die Einhaltung dieser Zeitbedingungen unter Verwendung der Gleichungen aus Kapitel 3 überprüft werden. Steht die Bandbreite des Kommunikationsnetzes noch nicht fest, kann aus dem Allokationsergebnis die erforderliche Bandbreite berechnet werden.

Die Zuteilbarkeit eines Tasksets im fehlerfreien Fall, aber auch im Fehlerfall, hängt stark von der Auslastung des Systems ab. Die Tasksets werden deshalb so generiert, dass die einzelnen Auslastungen der Tasks eine definierte Systemauslastung erzeugen. Da für den Nachweis der Density-Test verwendet wird, erfolgt die Generierung des Tasksets nicht über die meistens verwendeten „Utilization“ ($u_i = e_i/p_i$), sondern direkt über die Density $\rho_i = e_i/d_i$. Der Begriff Auslastung bezieht sich deshalb im Folgenden auf ρ_i statt auf u_i. Ein Prozessor hat eine Auslastung von 100 %, wenn für die Densitys der zugeteilten Tasks gilt: $\sum \rho_i = 1$.

Die einzelnen Densitys sollten gleichverteilt sein, da sonst Verzerrungseffekte bei der Auswertung der generierten Tasksets auftreten können. Hierfür wurde der UUniFast Algorithmus [19] entwickelt, der ein Taskset mit exakt gleichverteilten Densitys erzeugt. Im vorliegenden System kann der Algorithmus allerdings nicht ohne Änderungen übernommen werden. UUniFast generiert unabhängige Auslastungen (hier: Densitys) für die einzelnen Tasks. Cotasks besitzen die gleiche Auslastung und sind demnach nicht mit der Modellvorstellung des unmodifizierten UUniFast Algorithmus vereinbar. Um eine vorgegebene Systemauslastung $\rho_{\max}$ exakt zu erreichen, müssten den Cotasks unterschiedliche Auslastungen zugewiesen werden; dies entspricht nicht der Realität und könnte das Ergebnis verfälschen. Fixiert man andererseits die Auslastung der Cotasks, kann weder die vorgegebene Systemauslastung noch die Gleichverteilung eingehalten werden. Bild 6.1 zeigt den originalen UUniFast Algorithmus in Pseudocode. Eingangsparameter sind die gewünschte Systemauslastung $\rho_{\max}$ sowie die Anzahl der Tasks n. Für jede Task T_i wird eine Auslastung ρ_i generiert.

Der Algorithmus wurde so modifiziert, dass die identischen Auslastungen der Cotasks berücksichtigt werden. Bild 6.2 zeigt den neuen $\text{UUniFast}_{\text{FT}}$ Algorithmus in Pseudocode. Als zusätzlichen Eingangsparameter benötigt $\text{UUniFast}_{\text{FT}}$ die Taskklasse $tc[i]$ der jeweiligen Task T_i. Die Berrechnung von `nextSumU` berücksichtigt nun alle verbleibenden Cotasks im Taskset

```
sumU = ρ_max;
for ( i=1; i≤n−1; i++ )
{
  nextSumU = sumU · rand^(1/(n−i));
  ρ_i = sumU − nextSumU;
  sumU = nextSumU;
}
ρ_n = sumU;
```

Bild 6.1: UUniFast Algorithmus [19] (Pseudocode).

sowie deren zusätzliche Auslastungen. Da gegen Ende der `for`-Schleife durch den Zufallsgenerator und höhere Taskklassen ein zu großer Wert für die Auslastung generiert werden kann, muss eine zusätzliche Überprüfung erfolgen. Diese sowie die anschließende neue Generierung übernimmt die `while`-Schleife.

```
sumU = ρ_max;
corr = 0;
n_all = 0;
for ( k=0; k≤n; k++ )
  n_all += tc[k];

for ( i=1; i≤n−1; i++ )
{
  corr += tc[i];
  nextSumU = sumU · rand^(1/(n_all − corr));
  ρ_i = sumU − nextSumU;

  // unerlaubt großes u_i am Ende vermeiden.
  while(tc[i] · ρ_i ≥ sumU)
  {
    nextSumU = sumU · rand^(1/(n_all − corr));
    ρ_i = sumU − nextSumU;
  }

  sumU = sumU − tc[i] · ρ_i;
}
ρ_n = sumU / tc[n];
```

Bild 6.2: $\text{UUniFast}_{\text{FT}}$ Algorithmus (Pseudocode).

Die geforderte Systemauslastung wird mit dem neuen UUniFast$_{\mathrm{FT}}$ Algorithmus exakt eingehalten, die Verteilung weicht nur minimal von der Gleichverteilung ab. Bild 6.3 zeigt die Abweichung mit Hilfe der kumulativen Verteilungsfunktionen (engl.: *cumulative distribution function* – CDF) von 100 generierten Tasksets (je 500 Tasks).

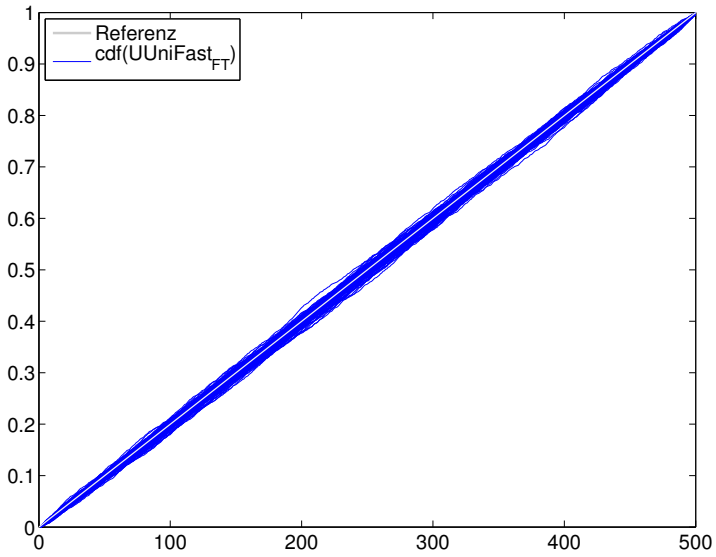


Bild 6.3: Kumulative Verteilungsfunktion (CDF) von UUniFast$_{\mathrm{FT}}$.

Zur besseren Vergleichbarkeit wird in den Grafiken ρ_i auf die maximal erlaubte Density $\rho_{\max}$ normiert und im Folgenden als *relative Density* bezeichnet: $\rho_{\mathrm{rel}} = \rho_i / \rho_{\max}$.

Start und Ende der Wirkketten werden ebenfalls pro Task zufällig generiert, mit folgender Einschränkung. In realen Systemen gibt es Knoten, von denen besonders viele Wirkketten starten bzw. an diesen Enden. Deshalb werden 80 % der Ein- bzw. Austrittspunkte an lediglich 4 Recheneinheiten gekoppelt (RE1, RE3, RE6 und RE8); für die restlichen 20 % der Wirkketten gibt es keine Einschränkung.

Falls nicht anders vermerkt, werden pro Messpunkt $n = 1000$ Tasksets generiert.

6.2 Zuteilbarkeit der Tasksets auf die Recheneinheiten

Die in Kapitel 5 beschriebenen Sortier- und Allokationsalgorithmen werden nun anhand simulativer Ergebnisse gegenübergestellt und bewertet.

Soweit nicht anders vermerkt, stellen die Grafiken die Anzahl der zuteilbaren Tasksets in % über der „relativen Density“ ρ_{rel} dar.

6.2.1 Anmerkungen zu den Referenzmessungen

Ohne Anpassung kann kein Standardverfahren für die Zuteilung verwendet werden. Es gibt verschiedene Möglichkeiten, welche Sortierungen und Zuteilungsalgorithmen als Referenzmessung dienen können. Die Ergebnisse der Allokation unter Verwendug der verschiedenen Varianten sind in Bild 6.4 dargestellt. Falls notiert, wird die Randbedingung, dass Cotasks nie auf derselben Recheneinheit platziert werden dürfen, verwendet.

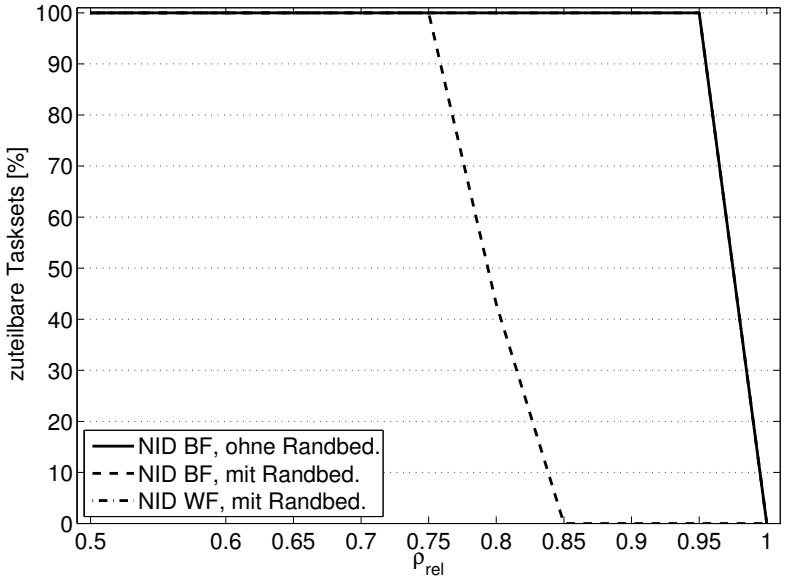


Bild 6.4: Gegenüberstellung alternativer Referenzmessungen.

NID BF, o. RB: Das Taskset wird nach *NID* sortiert und *ohne Beachtung von Randbedingungen* mit Hilfe des *Best Fit* Algorithmus auf die Recheneinheiten zugeteilt. In diesem Fall gibt die Referenz die größtmögliche Anzahl von zuteilbaren Tasksets wieder. Damit lässt sich berechnen, um wie viel schlechter die Zuteilung mit Randbedingungen im Vergleich zum optimalen Fall ist.

NID BF, m. RB: Das Taskset wird nach *NID* sortiert und *unter Beachtung von Randbedingungen* mit Hilfe des *Best Fit* Algorithmus auf die Recheneinheiten zugeteilt. In diesem Fall liefert die Referenz kein optimales Ergebnis, da der Zuteilungsalgorithmus nicht für die Randbedingungen geeignet ist.

NID WF, m. RB: Das Taskset wird nach *NID* sortiert und *unter Beachtung von Randbedingungen* mit Hilfe des *Worst Fit* Algorithmus auf die Recheneinheiten zugeteilt. In Bild 6.4 ist diese Kurve zwar deckungsgleich mit der ersten Kurve. Im Allgemeinen kann das Allokationsergebnis jedoch auf Grund der geforderten Randbedingung nicht optimal sein.

Nur die erste Variante liefert eine klar definierte Größe und wäre daher am besten als Referenz für den fehlerfreien Fall geeignet. Um eine ähnliche Referenz für den Fehlerfall zu bekommen, müsste das gesamte Taskset dabei neu zugeteilt werden. Für eine reine Betrachtung der

Zuteilbarkeit auf die Recheneinheiten ist diese Referenz optimal. Die Berechnung der Kommunikationskosten führt bei dieser Methode allerdings zu ungültigen Werten, da es sich um keine gültige Allokation handelt. Ein Vergleich der Kommunikationskosten ist nur mit einer Verteilung sinnvoll, die auch die Redundanzbedingungen erfüllt. Unter diesen Gesichtspunkten erscheint die dritte Variante als bester Mittelweg:

In jeder Simulation wird eine Referenzmessung angegeben, die mit folgendem, nicht optimalen Standardverfahren erzeugt wird: Das Taskset wird nach nicht aufsteigender Density (NID) sortiert, die Zuteilung erfolgt mittels Worst Fit. Zusätzlich gilt die Randbedingung, dass Cotasks nicht auf derselben Recheneinheit platziert werden dürfen.

6.2.2 Abhängigkeit von Sortiervarianten und Anschlusspunkten

Bei der pfadabhängigen Zuteilung wirkt sich die Verteilung der Anschlusspunkte von Sensoren und Aktoren besonders stark auf das Allokationsergebnis aus. Zur Verdeutlichung der Abhängigkeit von den Sortiervarianten werden jeweils die zwei Sortiervarianten *NITC,NID* und *NID,NITC* sowie die Referenzkurve in einer Grafik gegenübergestellt. Die Tasks werden mit *Best Fit* zugeteilt. Es werden drei Simulationen durchgeführt, in denen die Anzahl der Anschlusspunkte variiert. In Bild 6.5(a) sind die Sensoren und Aktoren nur an zwei Recheneinheiten (RE1 und RE3) angeschlossen. Bild 6.5(b) zeigt die Auswirkung, falls nur 80 % an diesen Recheneinheiten und die restlichen 20 % auf alle Recheneinheiten gleichverteilt sind. Bild 6.5(c) zeigt die Graphen für gleichverteilte Anschlusspunkte (auf alle Recheneinheiten).

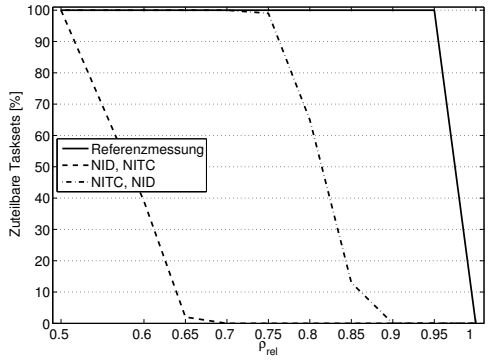
Den Simulationen liegen folgende Parameter zu Grunde. Für die Referenzkurve wird das Taskset nach *NID* sortiert und mit *Worst Fit* zugeteilt. Es werden $n = 100$ Tasksets pro Messpunkt (ρ_{rel}) erzeugt. Es gilt $\rho_{\max} = 0,9$.

Bewertung

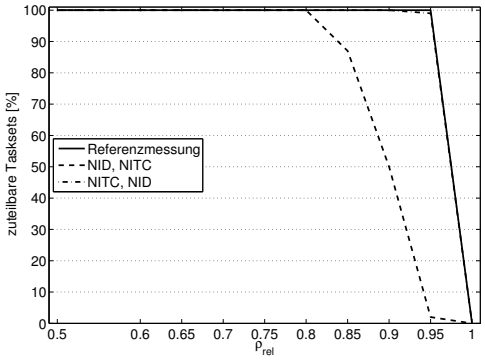
Wie erwartet hat die Sortiervariante *NITC,NID* entscheidende Vorteile gegenüber *NID,NITC* für den Fall, dass die Sensoren und Aktoren an einigen wenigen Switches angeschlossen sind. Dieser Vorteil schrumpft sobald die Anschlusspunkte etwas variieren. Bei einer gleichmäßigen Verteilung der Anschlusspunkte sind alle Sortierverfahren für das verwendete Taskset gleichwertig.

6.2.3 Abhängigkeit von der Allokationsvariante im Fehlerfall

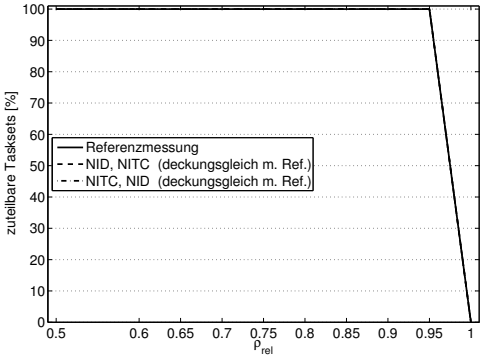
Im Folgenden werden die Auswirkungen der Allokationsvariante auf das Allokationsergebnis bei unterschiedlicher im fehlerfreien Fall erlaubter Density dargestellt. Zunächst erfolgt die Gegenüberstellung für den fehlerfreien Fall. Im nächsten Schritt wird die Zuteilbarkeit im Fehlerfall hinzugenommen. Ein Taskset gilt nur dann als zuteilbar, wenn alle seine Tasks sowohl im fehlerfreien als auch in jedem beliebigen Fehlerfall zuteilbar sind. Die Grafiken geben einen



(a) Ein-/Austrittspunkte: RE1 und RE3



(b) Ein-/Austrittspunkte: 80 % RE1 und RE3, 20 % gleichverteilt auf alle Recheneinheiten



(c) Ein-/Austrittspunkte: Gleichverteilt auf alle Recheneinheiten

Bild 6.5: Abhängigkeit von Sortiervariante und Anschlusspunkten (Zuteilungsverfahren: *Best Fit*, $\rho_{\max} = 0,9$).

Überblick, mit welcher Wahrscheinlichkeit, ein Taskset bei definierter Systemauslastung zuteilbar ist.

Die einzelnen Simulationen erfolgen mit unterschiedlicher maximal erlaubter Density $\rho_{\max}$ im fehlerfreien Fall. Im Fehlerfall, d. h. Ausfall einer Recheneinheit, wird bei der Zuteilung gemäß Abschnitt 5.5.5 immer eine maximal erlaubte Density von $\rho_{\max} = 1$ verwendet. Die relative Density ρ_{rel} wird jeweils im Bereich $[0,5; 1,0]$ variiert. Für jeden Messpunkt werden $n = 1000$ Tasksets generiert.

Die Zuteilung erfolgt im fehlerfreien Fall immer mittels *Worst Fit* und nach *NID,NITC* sortierten Tasks. Im Fehlerfall werden der Referenzallokation die Allokationen mit *Worst Fit*, *Best Fit* und *First Fit* mit nach *NID* sortierten Tasks gegenübergestellt. Die Sortierung der Tasks nach *NITC* im Fehlerfall hat sich in Messungen als geringfügig schlechter erwiesen, so dass im Folgenden nur mit der Sortierung nach *NID* gearbeitet wird.

Für das gewählte Test-Setup ist die Sortierung nach *NID,NITC* der Sortierung *NITC,NID* leicht im Vorteil und wird deshalb für die folgenden Messungen verwendet. Die Sortiervarianten werden im fehlerfreien Fall variiert und mit *Worst Fit* zugeteilt. Bild 6.6 zeigt das Allokationsergebnis im Fehlerfall, unter Anwendung der Zuteilungsverfahren *Worst Fit*, *Best Fit* und *First Fit*. Es gilt $\rho_{\max} = 0,8$.

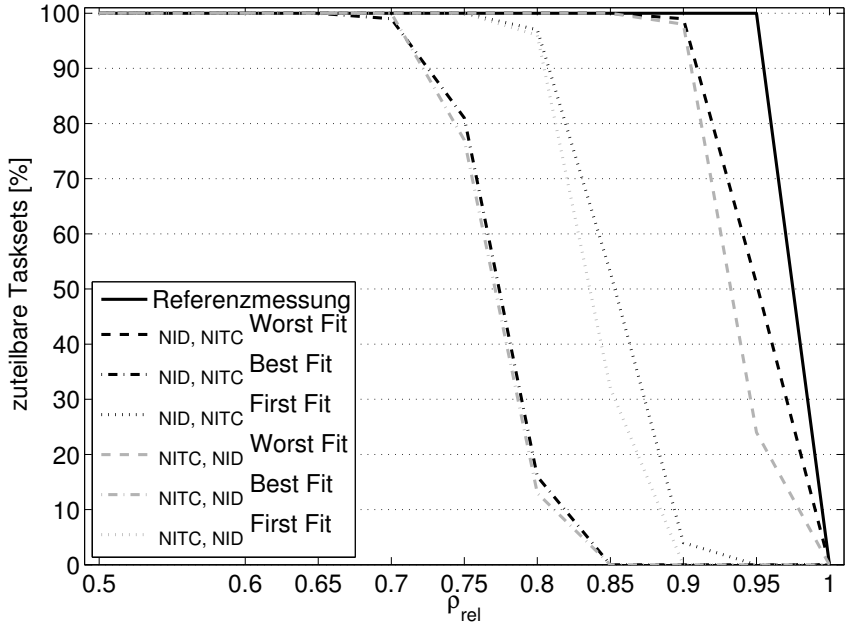


Bild 6.6: Zuteilbare Tasksets im Fehlerfall bei unterschiedlichen Sortierungen ($\rho_{\max} = 0,8$).

Bild 6.7 zeigt die zuteilbaren Tasksets für $\rho_{\max} = 1$ im fehlerfreien Fall. Die generierten Taskssets können bis zu einer maximalen Auslastung von 95 % unabhängig von der gewählten Methode zugeteilt werden. Obwohl hier die Randbedingung der Redundanzanforderung berücksichtigt werden, und damit zwar eine optimale Sortierung aber keine optimale Zuteilung der Tasks stattfinden kann, können die vorgestellten Verfahren genauso viele Tasksets zuteilen wie die optimale Referenzallokation.

Die darauf folgenden Bilder zeigen die Zuteilbarkeit im Fehlerfall bei unterschiedlichem $\rho_{\max}$ im fehlerfreien Fall. Zur Erinnerung: $\rho_{\max}$ wird erst für die Zuteilung im Fehlerfall auf 1 gesetzt. In den einzelnen Grafiken sind die verschiedenen Zuteilungsverfahren aufgetragen, die bei der Zuteilung der Tasks einer ausgefallenen Recheneinheit verwendet werden. Von den betrachteten Heuristiken erweist sich wieder *Worst Fit* als die beste Variante. Der Unterschied kommt hier deutlicher als bei der Allokation im fehlerfreien Fall zum Vorschein. Dies liegt daran, dass die Anzahl der Recheneinheiten, auf die eine Task zugeteilt werden darf, stark eingeschränkt ist. Die Tendenz von *Best Fit*, stärker noch als von *First Fit*, bei der Zuteilung einzelne Recheneinheiten besonders stark auszulasten, ist demnach kontraproduktiv und führt deshalb öfter dazu, dass ein Taskset nicht zugeteilt werden kann.

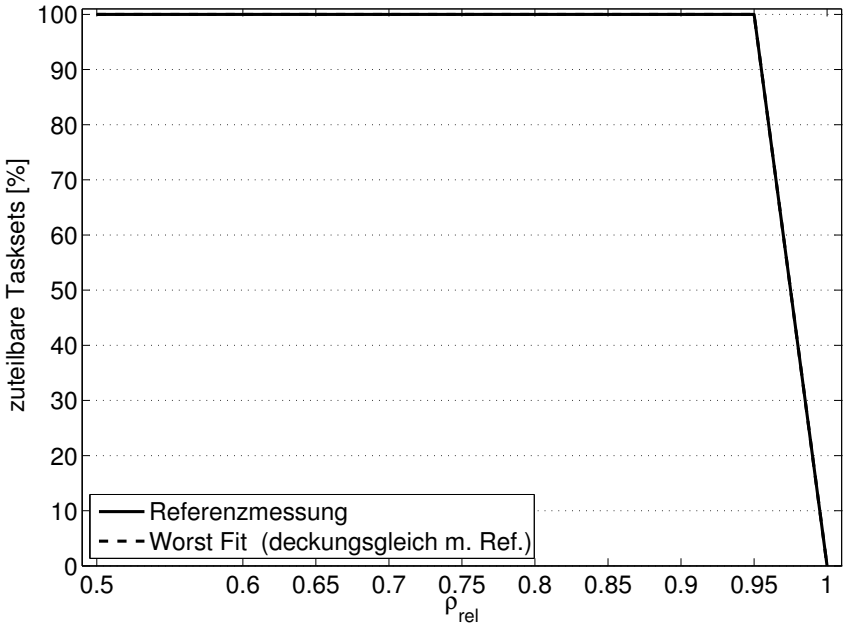


Bild 6.7: Zuteilbare Tasksets im fehlerfreien Fall ($\rho_{\max} = 1$).

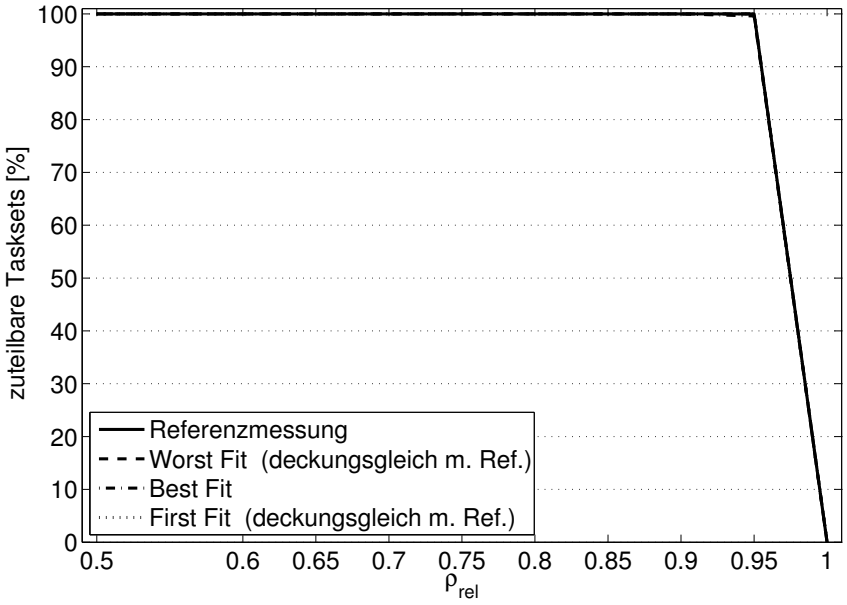


Bild 6.8: Zuteilbare Tasksets im Fehlerfall ($\rho_{\max} = 0,6$).

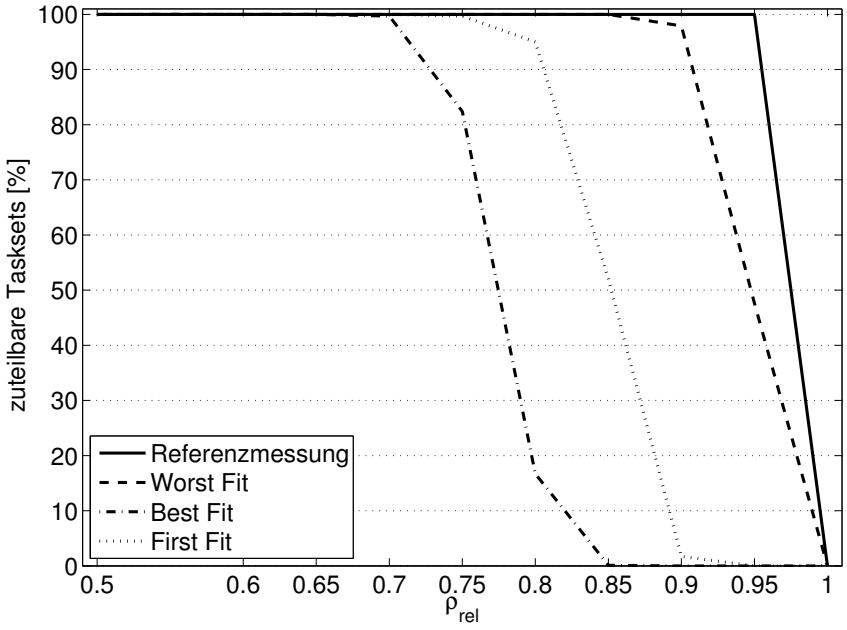


Bild 6.9: Zuteilbare Tasksets im Fehlerfall ($\rho_{\max} = 0,8$).

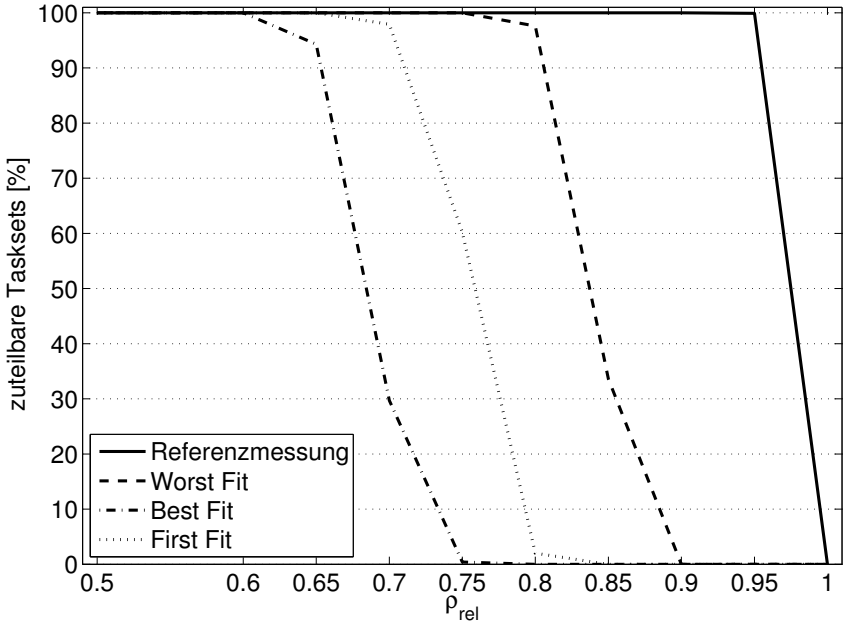


Bild 6.10: Zuteilbare Tasksets im Fehlerfall ($\rho_{\max} = 0,9$).

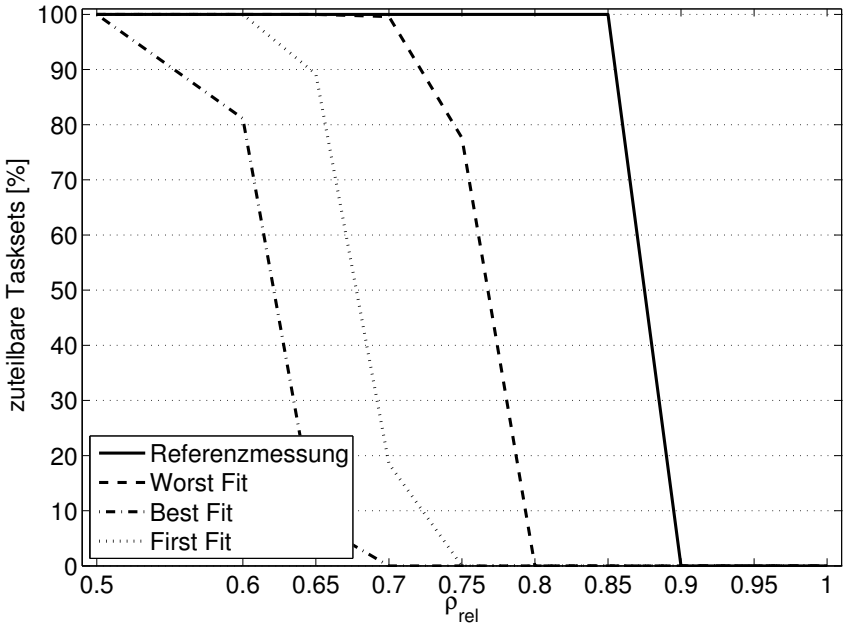


Bild 6.11: Zuteilbare Tasksets im Fehlerfall ($\rho_{\max} = 1,0$).

Bewertung

Da die Anschlusspunkte bei diesen Simulationen im gesamten System verteilt sind, können bei gleicher Allokationsstrategie unter Verwendung der Sortiervariante *NID,NITC* mehr Tasksets zugeteilt werden als mit *NITC,NID*. Erwartungsgemäß liefert *Worst Fit* bei gleicher Vorsortierung der Tasks gegenüber den anderen Zuteilungsstrategien die besten Ergebnisse. Auf Grund der pfadabhängigen Zuteilung liefern die vorgestellten Zuteilungsalgorithmen bzgl. der Referenz immer weniger zuteilbare Tasksets für einen Messpunkt.

6.3 Kommunikationskosten

Als Maß für die Kommunikationskosten wird im Folgenden die Anzahl der durchlaufenen Kanten (Hops) im Kommunikationsnetz verwendet. Dabei werden weder Nachrichtenpriorität noch die übertragene Datenmenge berücksichtigt. Da die Latenz wesentlich von der Anzahl der Hops abhängt, ist diese Betrachtung dennoch gültig.

Für die Referenzmessung werden mehrere Kommunikationspfade von Sensor zu Recheneinheit zu Aktor benötigt, um die geforderte Fehlertoleranz bei einem beliebigen Fehler bereitzustellen. Die Anzahl an Hops wird für die Wirkkette einer Cotask $T_{i(tc)}$ aus der Summe folgender Einzelschritte ermittelt.

1. Berechne die Anzahl der Hops für die zwei kürzesten Pfade zwischen $W^S_{i(tc)}$ und der Recheneinheit (RE_k) auf der die Berechnung durchgeführt wird.

2. Berechne die Anzahl der Hops für die zwei kürzesten Pfade zwischen RE_k und $W^E_{i(tc)}$.

Für die Kommunikationskosten einer Task werden die Kosten der Cotasks addiert. Die Kommunikationskosten des Systems erhält man aus der Summe der Kosten aller Tasks. Diese Berechnung ist gültig für Unicast. Falls Multicast verwendet wird, reduzieren sich die Kosten um die doppelt belegten Pfade für die Pakete $W^S_{i(tc)} \rightarrow \mathrm{RE}_k$ bzw. $\mathrm{RE}_k \rightarrow W^E_{i(tc)}$.

Für die Simulation werden folgende Parameter gesetzt: $\rho_{\max} = 0,7$, $\rho_{\mathrm{rel}} = 0,7$, $n = 1000$. Die Grafiken zeigen die mittleren Kosten sowie die Abweichungen im besten und schlechtesten Fall.

Bild 6.12 zeigt die Kommunikationskosten (in Hops) im Betrieb für den fehlerfreien Fall. In Bild 6.13 sind die Kosten dargestellt, die im Fehlerfall einmalig für die Rekonfiguration anfallen. Rekonfiguration bedeutet Laden der Binarys der ausgefallenen Tasks auf die neuen Recheneinheiten. Hierfür wird die kürzeste Verbindung zwischen neuer Recheneinheit und der nächstgelegenen Recheneinheit, die eine Cotask besitzt, gesucht. Bei Klasse 1 Tasks wird eine durchschnittliche Kommunikationslänge angenommen.

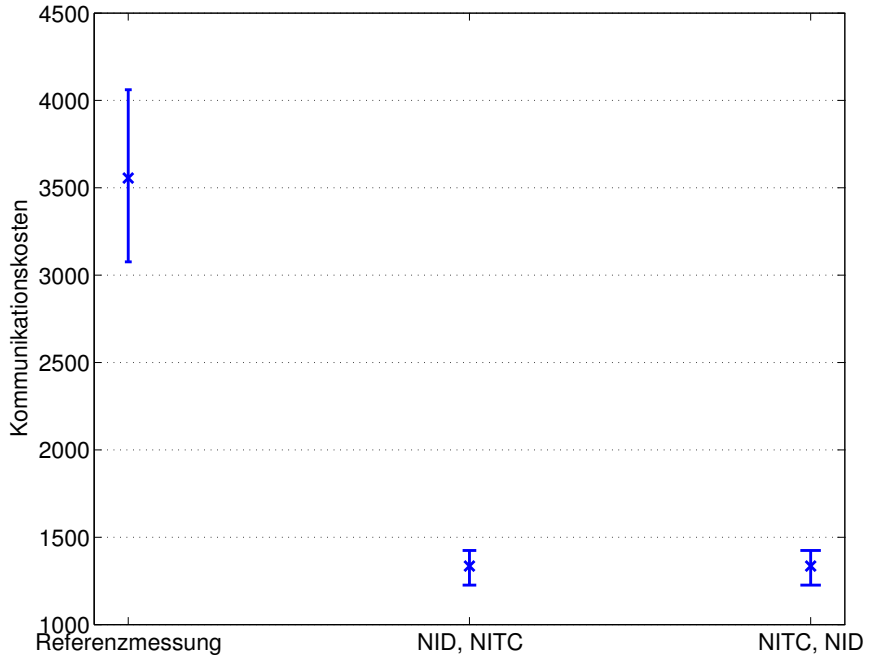


Bild 6.12: Kommunikationskosten, laufender Betrieb.

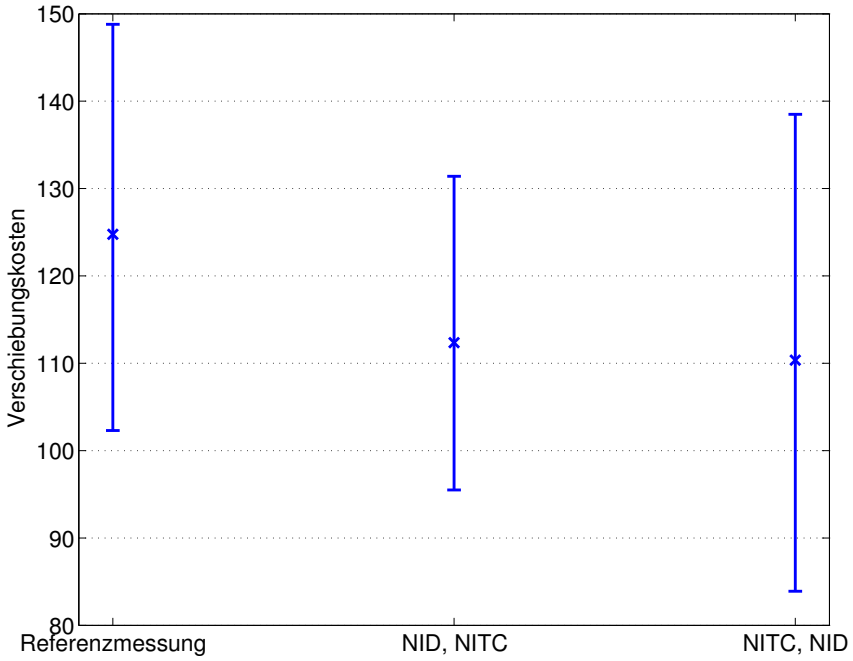


Bild 6.13: Kommunikationskosten bei der Rekonfiguration.

Bewertung

Hier zeigt sich der große Vorteil der vorgestellten Allokationsmethoden im Vergleich zur Referenz. Während die Redundanzanforderung bei der Referenz durch zusätzliche Kommunikationswege bereitgestellt werden muss, wird bei den pfadabhängigen Allokationen von vorn herein die Redundanzanforderung durch knotendisjunkte Pfade erfüllt.

Im Fehlerfall müssen bei der Referenzmessung die Kommunikationspfade von und zu den ausgefallenen Tasks neu berechnet werden. Dies führt dazu, dass sich auch die Latenzen auf den einzelnen Pfaden ändern können. Bei der vorgestellten Methode bleiben Pfade und Prioritäten für die Wirkketten gleich.

Die bei der Rekonfiguration zusätzlich auftretenden Paketströme, um die Binarys der Applikationen auf die neuen Recheneinheiten zu laden, halten sich bei allen Varianten in etwa die Waage.

7 Ergebnisse und Ausblick

Ausgehend von der Problematik heutiger Bordnetzarchitekturen im Automobilbereich wurde in dieser Arbeit eine tragfähige, zukunftssichere Systemplattform vorgestellt. Insbesondere wurden die Auswirkungen verschiedener Aspekte, wie beispielsweise der Virtualisierung oder der Bereitstellung fehlertoleranten Verhaltens, auf das zeitliche Verhalten des Gesamtsystems analysiert und bewertet. Ein Teil des vorgestellten Konzepts konnte erfolgreich in einem Demonstrator umgesetzt und die Machbarkeit gezeigt werden.

7.1 Beiträge dieser Arbeit

Basierend auf einer Analyse aktueller und zukünftiger Anforderungen an eine automotive Systemplattform wurde ein tragfähiger Architekturvorschlag erarbeitet. Es wurde dargelegt, dass die Problematik der zunehmenden Anzahl von Steuergeräten in heutigen Fahrzeugen durch Lösungsansätze aus der IT-Welt (Serverfarmen, Cluster) entschärft werden kann. Für eingebettete Systeme muss jedoch zusätzlich auf die Einhaltung von Echtzeitbedingungen sowie auf eine ressourcenschonende Umsetzung geachtet werden. Letztere wird im vorgestellten System durch das Zusammenführen mehrerer Funktionen auf einige wenige, einheitliche Steuergeräte erreicht, wobei die Abschottung der einzelnen Funktionen gegeneinander mit Hilfe von Virtualisierungstechniken erfolgt. Das resultierende robuste System bietet zudem die Möglichkeit zur dynamischen Rekonfiguration im Fehlerfall. Hierfür wurde ein Softwarekonzept vorgestellt, das die geforderten Systemfunktionen bereitstellt. Die Tragfähigkeit dieses Ansatzes wurde anhand eines prototypischen Aufbaus im Rahmen des Forschungsprojekts IT_Motive 2020 gezeigt.

Des Weiteren wurde das zeitliche Verhalten des vorgestellten Konzepts analysiert. Besonderes Augenmerk wurde dabei auf den Einfluss der virtuellen Umgebung und auf die Synchronität des Systems gelegt. Ein wichtiger Aspekt beim vorgestellten Konzept ist die Paketverarbeitung. Hierfür wurden verschiedene Lösungsmöglichkeiten hinsichtlich allgemeinem Zeitverhalten und Vorhersagbarkeit gegenübergestellt, wobei die hybride Lösung aus Hardware-Virtualisierung und Virtualisierung über eine Netzwerk-Treiberdomäne favorisiert wird. Das Konzept der „Timed Messages“ von Poledna [84] wurde für den Einsatz in der virtuellen Umgebung erweitert. Deren Verwendung stellt die systemweite Datenkonsistenz sicher.

Aufbauend auf diesem Basissystem wurden Maßnahmen zur Bereitstellung fehlertoleranten Verhaltens sowie deren Integration erörtert. Die für eine Systemrekonfiguration notwendigen Mechanismen werden durch das eingangs vorgestellte Softwareframework bereitgestellt. Zusätzlich wurde ein Konzept erarbeitet, das ähnlich zu TMR ein fehlertolerantes Verhalten

ohne Beteiligung des Softwareframeworks ermöglicht. Hierdurch lässt sich die Zeit zur Wiederherstellung eines konsistenten, fehlerfreien Zustands reduzieren. Die Funktionsfähigkeit des Konzept konnte ebenfalls im o. g. prototypischen Aufbau gezeigt werden.

Abschließend wurden Strategien zur Allokation der verarbeitenden Tasks entwickelt, die für den fehlerfreien Fall als auch im Falle eines Ausfalls die Aufrechterhaltung der sicherheitskritischen Funktionen gewährleisten. Anders als bei Standardverfahren wird bei den vorgestellten Heuristiken insbesondere auf geringe Kommmunikationskosten und Einhaltung von Redundanzanforderungen geachtet. Die Vorteile dieser Heuristiken wurden anhand von Simulationen verdeutlicht.

7.2 Ausblick

Mit der prototypischen Umsetzung einiger Aspekte in einem Demonstratorfahrzeug wurde die Anwendbarkeit des vorgestellten Ansatzes bewiesen. Manche der dargestellten Konzepte wurden jedoch noch nicht implementiert. Die Vergangenheit hat gezeigt, dass solche Veränderungen, die nicht nur die Systemarchitektur sondern auch eine Vielzahl von firmeninternen Abläufen der OEMs als auch der Zulieferer betrifft, nur Schritt für Schritt umgesetzt werden. Daher wäre eine Untersuchung sinnvoll, wie Teilaspekte dieser Arbeit in bestehende Systeme bzw. Systemstrukturen integriert werden können.

Beispielsweise wird in dieser Arbeit Ethernet als Kommunikationssystem zugrunde gelegt. Obwohl es in der Automatisierungstechnik und in der Luftfahrt bereits Verwendung findet, wird es in näherer Zukunft voraussichtlich nicht für sicherheitskritische Funktionen in Pkws Verwendung finden. Jedoch zeichnet sich bereits ein Trend ab, nach dem in den nächsten Jahren zunehmend Virtualisierungstechniken eingesetzt werden könnten. Daher müssten die vorgestellten Konzepte an ein aktuelles Bussystem wie beispielsweise FlexRay angepasst werden. Hierfür sind nur geringe Anpassungen notwendig, da bereits bei der Konzeption darauf geachtet wurde, die Verarbeitung und die Kommunikation möglichst getrennt voneinander zu betrachten.

Das Konzept der hybriden Netzwerkanbindung sollte implementiert und reale Messungen bzgl. des Zeitverhaltens durchgeführt werden. Auch das vorgestellte Konzept der Anbindung einer verteilten Echtzeit-Datenbasis (dRTDB) sollte in einem realen Anwendungsfall seine Tragfähigkeit unter Beweis stellen.

Die Überprüfung der Zuteilbarkeit in Kapitel 5 erfolgt anhand des Density-Tests. Falls genügend Ressourcen zur Verfügung stehen, ist dieser sicherlich ausreichend. Sobald man jedoch an die Kapazitätsgrenzen stößt, müssen exaktere Verfahren zur Überprüfung der Zuteilbarkeit verwendet werden, wie sie in Kapitel 3 beschrieben sind. Diese besitzen jedoch eine höhere Komplexität und benötigen daher mehr Zeit, um zu einem Ergebnis zu kommen.

Außerdem bleibt zu prüfen, ob sich Teile dieser Arbeit in AUTOSAR integrieren lassen. Beispielsweise sieht AUTOSAR derzeit keine dynamische Rekonfiguration vor. Der statische Anteil der vorgestellten Redundanzkonzepte könnte jedoch in AUTOSAR umgesetzt werden. Al-

lerdings muss dann im Falle eines Ausfalls eine Verringerung des Redundanzgrads in Kauf genommen werden.

Für sicherheitskritische Anwendungen muss eine Einordnung in die SIL- bzw. ASIL-Stufen erfolgen. Hierzu ist es u. a. erforderlich, die genauen zeitlichen Abläufe bei der Rekonfiguration zu kennen. Während der Rekonfigurationsphase besitzt das System einen geringeren Redundanzgrad als gefordert. Die Wahrscheinlichkeit, mit der ein weiterer Ausfall in dieser Zeit das gesamte System zum Ausfall (Systemzusammenbruch) bringen kann, muss daher in die Einstufung einfließen.

A ASIL – Gefährdungsklassen

Schwere der Verletzung (S)

Im Falle eines Ausfalls ist mit Verletzungen der Klasse S zu rechnen – von „keine“ (S0) bis „Tod wahrscheinlich“ (S3).

S0 Keine Verletzungen. S0 ist nicht sicherheitskritisch und wird deshalb keiner ASIL Klasse zugeordnet.
S1 Leichte bis mittlere Verletzungen.
S2 Schwere bis lebensbedrohliche Verletzungen, Überleben wahrscheinlich.
S3 Lebensbedrohliche und tödliche Verletzungen, Überleben unwahrscheinlich.

Gefahrenpotenzial (E)

Wahrscheinlichkeit mit der die Funktion ausgeführt wird, und ein Ausfall überhaupt eine Auswirkung hat.

E1 extrem unwahrscheinlich.
E2 geringe Wahrscheinlichkeit, $<1\,\%$ der durchschnittlichen Betriebszeit.
E3 mittlere Wahrscheinlichkeit, $1\,\%$ - $10\,\%$ der durchschnittlichen Betriebszeit.
E4 hohe Wahrscheinlichkeit, $>10\,\%$ der durchschnittlichen Betriebszeit.

Beispiele für die Einordnung in Gefahrenpotenziale:
E2: Betanken, Überholen, Rückwärtsfahrt
E3: nasse Straße, Tunnelfahrt
E4: Beschleunigen, Gangwechsel

Kontrollierbarkeit (C)

Wie gut kann der (durchschnittliche) Fahrer den Fehler ausgleichen?

C1 einfach zu kontrollieren.
C2 normalerweise zu kontrollieren.
C3 kaum oder nicht zu kontrollieren.

Beispiele für die Einordnung in die Kontrollierbarkeit:
C1: blockiertes Lenkrad bei Schrittgeschwindigkeit: Fahrer kann anhalten.

C2: blockierende Räder trotz ABS: Fahrer kann einigermaßen kontrolliert anhalten.
C3: Airbag löst bei hoher Geschwindigkeit aus: Durchschnittsfahrer kann das Fzg. nicht unter Kontrolle halten.

Zuordnung zu ASIL Stufen A-D

Die Tabelle zeigt, wie sich die o. g. Klassen in die ASIL Stufen abbilden lassen.

		C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	E1	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	E1	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

Tabelle A.1: Zuordnung zu ASIL Stufen A-D

Literaturverzeichnis

[1] *IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002), 2008.

[2] ABRAMSON, D. et al.: *Intel Virtualization Technology for Directed I/O*. Intel Technology Journal, 10(3), 2006.

[3] ALBERS, K., F. BODMANN und F. SLOMKA: *Hierarchical event streams and event dependency graphs: a new computational model for embedded real-time systems*. In: *18th Euromicro Conference on Real-Time Systems, 2006*, 2006.

[4] ANDLER, S. F., J. HANSSON, J. ERIKSSON, J. MELLIN, M. BERNDTSSON und B. EFTRING: *DeeDS towards a distributed and active real-time database system*. SIGMOD Record, 25(1):38–51, 1996.

[5] *Automotive Safety Integrity Level (ASIL), ISO26262*.

[6] ATTIYA, G. und Y. HAMAM: *Reliability oriented task allocation in heterogeneous distributed computing systems*. In: *9th International Symposium on Computers and Communications*, Band 1, Seiten 68–73, Juni 2004.

[7] *AUTOSAR (R3.1) – Technical Overview*, 2008.

[8] BARHAM, PAUL, BORIS DRAGOVIC, KEIR FRASER, STEVEN HAND, TIM HARRIS, ALEX HO, ROLF NEUGEBAUER, IAN PRATT und ANDREW WARFIELD: *Xen and the art of virtualization*. In: *19th ACM symposium on Operating systems principles, 2003*, Seiten 164–177, New York, NY, USA, 2003. ACM.

[9] BARUAH, SANJOY, DEJI CHEN, SERGEY GORINSKY und ALOYSIUS MOK: *Generalized Multiframe Tasks*. Real-Time Systems, 17:5–22, 1999.

[10] BARUAH, SANJOY K.: *Dynamic- and Static-priority Scheduling of Recurring Real-time Tasks*. Real-Time Systems, 24(1):93–128, 2003.

[11] BARUAH, SANJOY K., ALOYSIUS K. MOK und LOUIS E. ROSIER: *Preemptively Scheduling Hard-Real-Time Sporadic Tasks on One Processor*. In: *Real-Time Systems Symposium, 1990*, Seiten 182–190. IEEE Computer Society Press, 1990.

[12] BARUAH, S.K.: *A general model for recurring real-time tasks*. In: *Real-Time Systems Symposium, 1998*, Seiten 114–122, Dezember 1998.

[13] BARUAH, S.K., DEJI CHEN und A. MOK: *Static-priority scheduling of multiframe tasks*. In: *11th Euromicro Conference on Real-Time Systems, 1999*, Seiten 38–45, 1999.

[14] BAYER, MARTIN: *Echtzeitverhalten fehlertoleranter Mehrrechnersysteme*. Doktorarbeit, Technische Universität München, 1990.

[15] BICKING, F., B. CONRARD und J.-M. THIRIET: *Integration of Dependability in a Task Allocation Problem*. IEEE Transactions on Instrumentation and Measurement, 53(6):1455–1463, 2004.

[16] BINI, E. und G. BUTTAZZO: *A hyperbolic bound for the rate monotonic algorithm*. In: *13th Euromicro Conference on Real-Time Systems, 2001.*, Seiten 59–66, 2001.

[17] BINI, E. und G. BUTTAZZO: *The Space of EDF Feasible Deadlines*. In: *19th Euromicro Conference on Real-Time Systems, 2007*, Seiten 19–28, Juli 2007.

[18] BINI, E. und G.C. BUTTAZZO: *The space of rate monotonic schedulability*. In: *Real-Time Systems Symposium, 2002*, Seiten 169–178, 2002.

[19] BINI, E. und G.C. BUTTAZZO: *Biasing effects in schedulability measures*. In: *16th Euromicro Conference on Real-Time Systems, 2004*, Seiten 196–203, Juli 2004.

[20] BINI, E. und G.C. BUTTAZZO: *Schedulability analysis of periodic fixed priority systems*. Band 53, Seiten 1462–1473, November 2004.

[21] BINI, E., G.C. BUTTAZZO und G.M. BUTTAZZO: *Rate monotonic analysis: the hyperbolic bound*. IEEE Transactions on Computers, 52(7):933–942, Juli 2003.

[22] BINI, E. und A. CERVIN: *Delay-Aware Period Assignment in Control Systems*. In: *Real-Time Systems Symposium, 2008*, Seiten 291–300, Dezember 2008.

[23] BINI, E. und M. DI NATALE: *Optimal task rate selection in fixed priority systems*. In: *Real-Time Systems Symposium, 2005*, Seiten 409–420, Dezember 2005.

[24] BINI, E., M. DI NATALE und G. BUTTAZZO: *Sensitivity analysis for fixed-priority real-time systems*. In: *18th Euromicro Conference on Real-Time Systems, 2006*, Seiten 10–22, 2006.

[25] BINI, E., THI HUYEN CHAU NGUYEN, P. RICHARD und S.K. BARUAH: *A Response-Time Bound in Fixed-Priority Scheduling with Arbitrary Deadlines*. IEEE Transactions on Computers, 58(2):279–286, Februar 2009.

[26] BRIDAL, O., R. SNEDSBOL und L.-A. JOHANSSON: *On the design of communication protocols for safety-critical automotive applications*. Band 2, Seiten 1098–1102, Juni 1994.

[27] BURNS, ALAN und ANDY WELLINGS: *Real-Time Systems and Programming Languages*. Addison-Wesley, 1997.

[28] CHAKRABORTY, S., S. KUNZLI und L. THIELE: *Approximate schedulability analysis*. In: *Real-Time Systems Symposium, 2002*, Seiten 159–168, 2002.

[29] CHEN, DEJI, A.K. MOK und TEI-WEI KUO: *Utilization bound revisited*. IEEE Transactions on Computers, 52(3):351–361, März 2003.

[30] CHEN, DEJI, ALOYSIUS MOK und SANJOY BARUAH: *On Modeling Real-time Task Systems*. In: *Lecture Notes in Computer Science – Lectures on Embedded Systems*, 1996.

[31] CHERKASSKY, V. und C.-I.H. CHEN: *Redundant task-allocation in multicomputer systems*. In: *IEEE Transactions on Reliability*, Band 41, Seiten 336–342, September 1992.

[32] CIRINEI, M., E. BINI, G. LIPARI und A. FERRARI: *A Flexible Scheme for Scheduling Fault-Tolerant Real-Time Tasks on Multiprocessors*. In: *Parallel and Distributed Processing Symposium, 2007*, Seiten 1–8, März 2007.

[33] CLARK, CHRISTOPHER, KEIR FRASER, STEVEN HAND, JAKOB GORM HANSEN, ERIC JUL, CHRISTIAN LIMPACH, IAN PRATT und ANDREW WARFIELD: *Live Migration of Virtual Machines*. In: *2nd Symposium on Networked Systems Design and Implementation*, 2005.

[34] COFFMAN, E. G., M. R. GAREY und D.S. JOHNSON: *Approximation Algorithms for NP-Hard Problems*, Seiten 46–49. PWS Publishing, 1996.

[35] DAL CIN, M.: *Fehlertolerante Systeme*. Teubner, 1979.

[36] DEVI, U.C.: *An improved schedulability test for uniprocessor periodic task systems*. In: *15th Euromicro Conference on Real-Time Systems, 2003*, Seiten 23–30, Juli 2003.

[37] EHRET, JÜRGEN: *Validation of Safety-Critical Distributed Real-Time Systems*. Doktorarbeit, September 2003.

[38] E|ENOVA: *SEIS – Sicherheit in Eingebetteten IP-basierten Systemen*, 2009. Pressemitteilung.

[39] *Safety over EtherCAT*. PC-Control 01/2007, 2007.

[40] *EtherCAT Webseite*.

[41] FOWLER, M.R., E. STIPIDIS und F.H. ALI: *Practical Verification of an Embedded Beowulf Architecture Using Standard Cluster Benchmarks*. Seiten 140–145, Oktober 2008.

[42] FÜHRER, T., B. MÜLLER, W. DIETERLE, F. HARTWICH, R. HUGEL und M WALTHER: *CAN Network with Time Triggered Communication*. In: *7th International CAN Conference*, 2000.

[43] FÜHRER, T., B. MÜLLER, W. DIETERLE, F. HARTWICH, R. HUGEL und M WALTHER: *Time Triggered Communication on CAN*. In: *7th International CAN Conference*, 2000.

[44] GAEDE, KARL-WALTER: *Zuverlässigkeit – Mathematische Modelle*. Carl Hanser Verlag, 1977.

[45] GELFAND, SAUL B. und SANJOY K. MITTER: *Analysis of simulated annealing for optimization*. 24th IEEE Conference on Decision and Control, 1985, 24:779–786, Dezember 1985.

[46] GOEBL, MATTHIAS: *Eine realzeitfähige Architektur zur Integration kognitiver Funktionen*. Dissertation, Technische Universität München, München, 2009.

[47] GOEBL, MATTHIAS, SEBASTIAN DRÖSSLER und GEORG FÄRBER: *Systemplattform für videobasierte Fahrerassistenzsysteme*. In: LEVI, P., M. SCHANZ, R. LAFRENZ und V. AVRUTIN (Herausgeber): *Autonome Mobile Systeme 2005*, Informatik Aktuell, Seiten 187–193. Springer-Verlag, Dezember 2005.

[48] GRESSER, KLAUS: *Echtzeitnachweis ereignisgesteuerter Realzeitsysteme*. Doktorarbeit, Technische Universität München, 1993.

[49] HANSSON, H., L. LAWSON, O. BRIDAL, C. ERIKSSON, S. LARSSON, H. LON und M. STROMBERG: *BASEMENT: an architecture and methodology for distributed automotive real-time systems*. IEEE Transactions on Computers, 46(9):1016–1027, September 1997.

[50] HANSSON, H.A., H.W. LAWSON, M. STROMBERG und S. LARSSON: *BASEMENT: a distributed real-time architecture for vehicle applications*. Seiten 220–229, Mai 1995.

[51] HEINECKE, H., K. P. SCHNELLE, H. FENNEL, J. BORTOLAZZI, L. LUNDH, J. LEFLOUR, J. L. MATÉ, K. NISHIKAWA und T. SCHARNHORST: *AUTomotive Open System ARchitecture-An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E Architectures*, 2004.

[52] HENZINGER, T.A., B. HOROWITZ und C.M. KIRSCH: *Giotto: a time-triggered language for embedded programming*. Proceedings of the IEEE, 91(1):84–99, Januar 2003.

[53] HOLLAND, JOHN H.: *Adaptation in natural and artificial systems*. The MIT Press, 1995 (Erstausgabe: 1975).

[54] KOPETZ, H., A. ADEMAJ, P. GRILLINGER und K. STEINHAMMER: *The time-triggered Ethernet (TTE) design*. 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2005, Seiten 22–33, Mai 2005.

[55] KOPETZ, H. und G. BAUER: *The time-triggered architecture*. Proceedings of the IEEE, 91(1):112–126, Januar 2003.

[56] KOPETZ, H., A. DAMM, C. KOZA, M. MULAZZANI, W. SCHWABL, C. SENFT und R. ZAINLINGER: *Distributed fault-tolerant real-time systems: the Mars approach*. IEEE Micro, 9(1):25–40, Februar 1989.

[57] KOPETZ, HERMANN: *Real-Time Systems – Design Principles for Distributed Embedded Applications*. Kluwer Academic Publichers, 1997.

[58] KOROUŠIĆ-SELJAK, B. und J. E. COOLING: *Optimization of Multiprocessor Real-Time Embedded System Structure*. In: *7th Mediterranean Electrotechnical Conference*, Band 1, Seiten 313–316, 1994.

[59] LAYLAND, JAMES W. und C.L. LIU: *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*. Association for Computing Machinery, 20(1):46–61, Januar 1973.

[60] LEHOCZKY, J., L. SHA und Y. DING: *The rate monotonic scheduling algorithm: exact characterization and average case behavior*. In: *Real-Time Systems Symposium, 1989*, Seiten 166–171, Dezember 1989.

[61] LEHOCZKY, J.P.: *Fixed priority scheduling of periodic task sets with arbitrary deadlines*. In: *Real-Time Systems Symposium, 1990*, Seiten 201–209, Dezember 1990.

[62] LIPARI, G. und E. BINI: *Resource partitioning among real-time applications*. In: *15th Euromicro Conference on Real-Time Systems, 2003*, Seiten 151–158, Juli 2003.

[63] LIU, JANE W.S.: *Real-Time Systems*. Prentice Hall, 2000.

[64] LIU, N.H., K.L. YEUNG und D.C.W. PAO: *Scheduling algorithms for input-queued switches with virtual output queueing*. IEEE International Conference on Communications, 2001, 7:2038–2042, 2001.

[65] LORENTE, J.L., G. LIPARI und E. BINI: *A hierarchical scheduling model for component-based real-time systems*. In: *Parallel and Distributed Processing Symposium, 2006*, April 2006.

[66] MASRUR, A., S. DRÖSSLER und G. FÄRBER: *An off-line variable-size bin packing for EDF*. Technischer Bericht, TU-München, Lehrstuhl für Realzeit-Computersysteme, 2006.

[67] MASRUR, A., S. DRÖSSLER und G. FÄRBER: *Improvements in Polynomial-Time Feasibility Testing for EDF*. In: *Design, Automation & Test in Europe*, 2008.

[68] MASRUR, ALEJANDRO: *Verifying and Allocating Real-Time Tasks on Distributed Processing Units*. Doktorarbeit, Technische Universität München, 2010.

[69] MASRUR, ALEJANDRO, SEBASTIAN DRÖSSLER, THOMAS PFEUFFER und SAMARJIT CHAKRABORTY: *VM-Based Real-Time Services for Automotive Control Applications*. In: *16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2010.

[70] MATHIASON, G., S.F. ANDLER und S.H. SON: *Virtual Full Replication by Adaptive Segmentation*. In: *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2007*, Seiten 327–336, August 2007.

[71] MCLOUGHLIN, I.V. und T. BRETSCHNEIDER: *Achieving low-cost high-reliability computation through redundant parallel processing*. International Conference on Computing Informatics, 2006, Seiten 1–6, Juni 2006.

[72] MCLOUGHLIN, I.V., V. GUPTA, G.S. SANDHU, S. LIM und T.R. BRETSCHNEIDER: *Fault tolerance through redundant COTS components for satellite processing applications*. Joint Conference of The Fourth International Conference on Information, Communications and Signal Processing and Fourth Pacific-Rim Conference on Multimedia, 1:296–299, Dezember 2003.

[73] MOK, A.K. und D. CHEN: *A multiframe model for real-time tasks*. In: *Real-Time Systems Symposium, 1996*, Seiten 22–29, Dezember 1996.

[74] MÜLLER-RATHGEBER, B., M. EICHHORN und H.-U. MICHEL: *A unified Car-IT Communication-Architecture: Design guidelines and prototypical implementation*. Intelligent Vehicles Symposium, 2008, Seiten 709–714, Juni 2008.

[75] MÜLLER-RATHGEBER, B., M. EICHHORN und H.-U. MICHEL: *A unified Car-IT Communication-Architecture: Network switch design guidelines*. IEEE International Conference on Vehicular Electronics and Safety, 2008, Seiten 16–21, September 2008.

[76] *Onboard Diagnosesystem der zweiten Generation*. http://www.obd-2.de.

[77] OBERMAISSER, ROMAN, PHILIPP PETI und FULVIO TAGLIABO: *An integrated architecture for future car generations*. Real-Time Systems, 36(1-2):101–133, 2007.

[78] *openMosix*. http://www.openmosix.org.

[79] *OpenSSI*. http://www.openssi.org.

[80] PELLIZZONI, R. und G. LIPARI: *A new sufficient feasibility test for asynchronous real-time periodic task sets*. In: *16th Euromicro Conference on Real-Time Systems, 2004*, Seiten 204–211, Juli 2004.

[81] PFEUFFER, THOMAS: *Optimierung des Zeitverhaltens von Xens Paketverarbeitungsmechanismus für den Einsatz in verteilten Regelungsanwendungen*. Diplomarbeit, Lehrstuhl für Realzeit-Computersysteme, Technische Universität München, 2009.

[82] POLEDNA, S.: *Fault tolerance in safety critical automotive applications: cost of agreement as a limiting factor*. In: *25th International Symposium on Fault-Tolerant Computing, 1995*, Seiten 73–82, Juni 1995.

[83] POLEDNA, S.: *Tolerating sensor timing faults in highly responsive hard real-time systems*. IEEE Transactions on Computers, 44(2):181–191, Februar 1995.

[84] POLEDNA, S., A. BURNS, A. WELLINGS und P. BARRETT: *Replica determinism and flexible scheduling in hard real-time dependable systems*. IEEE Transactions on Computers, 49(2):100–111, Februar 2000.

[85] POLEDNA, STEFAN: *Replica determinism in distributed real-time systems: a brief survey*. Real-Time Systems, 6(3):289–316, 1994.

[86] *Precision Time Protocol (IEEE1588)*.

[87] RANDELL, BRIAN: *System Structure for Software Fault Tolerance*. IEEE Transactions on Software Engineering, 1:220–232, 1975.

[88] RIES, WALTER: *Prozessorzuteilungsverfahren in fehlertoleranten Mehrrechnersystemen*. Doktorarbeit, Technische Universität München, 1985.

[89] ROBERTS, J.D.: *A Method of Optimizing Adjustable Parameters in a Control System*. Electronic and Communication Engineering, 109(48):519–528, November 1962.

[90] ROSTAMZADEH, B., H. LONN, R. SNEDSBOL und J. TORIN: *DACAPO: a distributed computer architecture for safety-critical control applications*. Seiten 376–381, September 1995.

[91] SALMON, J., C. STEIN und T. STERLING: *Scaling of Beowulf-class Distributed Systems*. November 1998.

[92] SANTOS, R., G. LIPARI und E. BINI: *Efficient On-line Schedulability Test for Feedback Scheduling of Soft Real-Time Tasks under Fixed-Priority*. In: *Real-Time and Embedded Technology and Applications Symposium, 2008*, Seiten 227–236, April 2008.

[93] STANKOVIC, JOHN A., MARCO SPURI, KRITHI RAMAMRITHAM und GIORGIO C. BUTTAZZO: *Deadline Scheduling for Real-Time Systems: EDF and Related algorithms*. Kluwer Academic Publishers, 1998.

[94] STOTT, J, P RODGERS, J MARTINEZ-OVANDO und S WALKER: *Automatic Metro Map Layout Using Multicriteria Optimization*. IEEE Transactions on Visualization and Computer Graphics, 2010.

[95] *Solaris Containers: Optimizing Resource Utilization for Predictable Service Levels*, 2007. White Paper.

[96] SURI, NEERAJ, CHRIS J. WALTER und MICHELLE M. HUGUE: *Advances in ULTRA-Dependable Distributed Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1995.

[97] TAKADA, H. und K. SAKAMURA: *Schedulability of generalized multiframe task sets under static priority assignment*. In: *4th International Workshop on Real-Time Computing Systems and Applications*, Seiten 80–86, Oktober 1997.

[98] THIELEN, HERBERT: *Optimierte Auslegung verteilter Realzeitsysteme*. Doktorarbeit, Technische Universität München, 2000.

[99] *TTTech Computertechnik: TTP. TTP/C protocol specification.*, 1999.

[100] VELASCO, M., P. MARTI und E. BINI: *Control-Driven Tasks: Modeling and Analysis*. In: *Real-Time Systems Symposium, 2008*, Seiten 280–290, Dezember 2008.

[101] VIDYARTHI, DEO PRAKASH und ANIL KUMAR TRIPATHI: *Maximizing reliability of distributed computing system with task allocation using simple genetic algorithm*. Journal of Systems Architecture, 47(6):549–554, 2001.

[102] *Intel Virtualization Technology Virtual Machine Device Queues*, 2007. White Paper.

[103] WANG, SHUHUA: *Specification, allocation and Schedulability Analysis for Fixed-Priority Hard Real-Time Systems*. Doktorarbeit, Technische Universität München, 1999.

[104] WOLF, WAYNE: *High Performance Embedded Computing*. Elsevier Inc., 2007.