AG VERNETZTE SYSTEME
FACHBEREICH INFORMATIK
AN DER TECHNISCHEN UNIVERSITÄT
KAISERSLAUTERN

DIPLOMARBEIT

SPEZIFIKATION, INTEGRATION UND SIMULATION EINES DUTY-CYCLING-PROTOKOLLS FÜR MACZ

Martin Winkler

30. Juli 2009

Spezifikation, Integration und Simulation eines Duty-Cycling-Protokolls für MacZ

Diplomarbeit

Arbeitsgruppe Vernetzte Systeme
Fachbereich Informatik
Technische Universität Kaiserslautern

Martin Winkler

Tag der Ausgabe : 01. Januar 2009
Tag der Abgabe : 30. Juli 2009

Betreuer : Prof. Dr. Reinhard Gotzhein und Dipl.-Inf. Marc Krämer

Ich erkläre hiermit, die vorliegende Diplomarbeit selbständig verfaßt zu haben. Die verwendeten Quellen und Hilfsmittel sind im Text kenntlich gemacht und im Literaturverzeichnis vollständig aufgeführt.

Kaiserslautern, den 30. Juli 2009

(Martin Winkler)

Inhaltsverzeichnis

Kapitel 1

Einleitung

Im Rahmen der Diplomarbeit war ein System bestehend aus Middleware und Protokollen zu entwickeln, das einer auf mehreren Knoten der *Imote2*-Plattform laufenden Applikation ein Framework zur Registrierung und zum Abonnement von Diensten bietet. Die Applikation ist ein Kontroll- und Steuerungssystem für ein inverses Pendel wie in [GKLC] beschrieben. Die angebotenen Dienste sollten dabei in der Lage sein, anspruchsvolle *Quality-of-Service*-Anforderungen zu erfüllen. So sollen periodisch benötigte Dienste nach einmaliger Anmeldung automatisch regelmäßig Daten an die Abonnenten versenden, wobei die Periode von jedem Abonnenten frei wählbar ist. Damit ist es möglich, ein Steuerungssystem für ein inverses Pendel aus drahtlos vernetzten Knoten zu bilden. Das verwendete Medienzugriffs-Protokoll wurde um Energiesparmechanismen erweitert, um die Laufzeit der beteiligten Knoten zu erhöhen. Abschließend wurde eine Simulation zur Überprüfung der Lauffähigkeit des Systems durchgeführt.

Die weitere Arbeit gliedert sich wie folgt. In Kapitel 2 werden die Grundlagen zu dieser Arbeit vorgestellt – zum einen Duty-Cycling-Protokolle, zum anderen die Komponenten, die weiterentwickelt wurden. Kapitel 3 erläutert, welche Änderungen und Erweiterungen an den bestehenden Systemen notwendig sind, um das angestrebte Ziel zu erreichen. Es enthält die formalen Dienstgüte-Anforderungen des Steuerungssystems, beschreibt detailliert die Schnittstelle zwischen Endanwendung und der Middleware und die Erläuterung der Weiterentwicklung der *AmICoM* zur *WNCS-Communication-Middleware*. Die Funktionsweise der neuen Komponenten sowie Algorithmen werden vorgestellt. Kapitel 4 zeigt, wie ein Energiesparmechanismus in das bestehende MAC-Protokoll *MacZ* eingebunden wurde. Es geht in einem Abschnitt auf den virtuellen Versuchsaufbau für den Netzwerksimulator NS2 ein, mit dessen Hilfe das Verhalten des Protokolls und die Leistung der Simulations-Anwendung überprüft wird. Die Arbeit schließt mit Kapitel 5, das aus einer Zusammenfassung der Arbeit und einem Ausblick auf weiterführende Schritte besteht.

Kapitel 2

Grundlagen

In den folgenden Abschnitten werden die Grundlagen zum besseren Verständnis der gestellten Aufgabe vorgestellt. Ferner werden die Systeme erläutert, auf denen diese Arbeit aufbaut.

2.1 Inverses Pendel

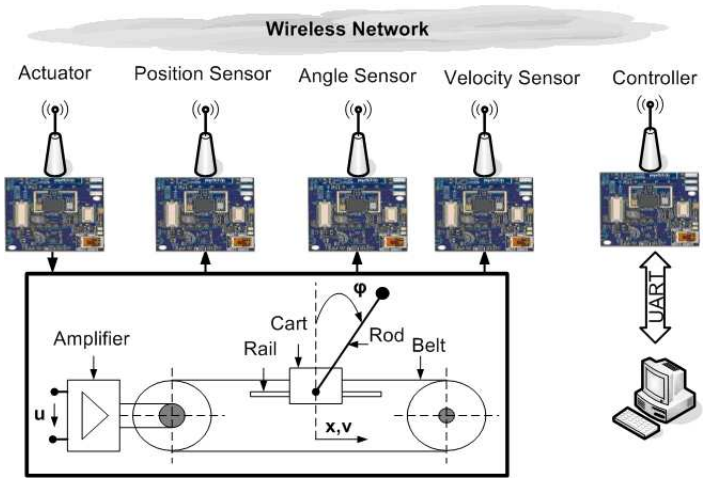

Abbildung 2.1: Aufbau des inversen Pendels und Steuerungssystems (Grafik entnommen aus [GKLC]).

Das inverse Pendel, wie in [GKLC] beschrieben, ist ein instabiles System aus einem beweglichen Wagen, an dem ein Pendel angebracht ist. Abbildung 2.1 zeigt die Bestandteile und den Aufbau. Das Pendel ist eine Stange und kann um den Befestigungspunkt rotieren. Der Wagen sitzt auf einer Schiene und wird über einen Riemen angetrieben. Durch rasches Beschleunigen des Wagens auf der Schiene und schnelle Richtungswechsel kann das Pendel aufgeschaukelt werden, sodass es senkrecht steht. Der Winkel φ bezeichnet die Abweichung des Pendels von der senkrechten Stellung, x und v bezeichnen Abstand und Geschwindigkeit. Der Riemen wird über einen Elektromotor angetrieben. Winkel, Position des Wagens und seine Geschwindigkeit

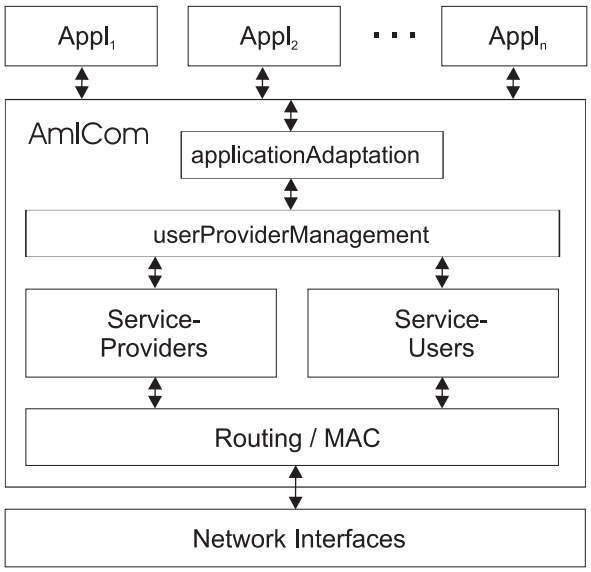

Abbildung 2.2: Architektur der *Ambient Intelligence Communication Middleware* (entnommen aus [FK07]).

werden durch Sensoren (*Angle Sensor, Position Sensor, Velocity Sensor*) erfasst, die drahtlos mit der Pendelsteuerung (*Controller*) kommunizieren. Diese sendet Stellwerte an den Riemenantrieb (*Actuator*). Die Sensoren bilden zusammen mit der Steuerung und dem Aktuator ein *wireless networked control system* (WNCS). Alle Knoten des Steuerungssystems (Sensoren, Controller, Aktuator) sind drahtlos vernetzte Imote2-Knoten. Dieses Pendelsteuerungssystem dient als Szenario für die Ermittlung von Dienstgüteanforderungen an das im Rahmen der Arbeit entwickelte System.

2.2 AmICoM

Mit der *Ambient Intelligence Communication Middleware* [KF07], [FK07] steht eine Schicht zur Verfügung, die Anwendungen das Suchen, Auffinden, Registrieren und Abonnieren von Diensten in einem Netzwerk ermöglicht, ohne dass die Anwendungen spezifische Informationen über Art oder Aufbau des Netzes haben müssen. Abbildung 2.2 zeigt die Architektur schematisch.

Diese Middleware setzt sich aus mehreren Komponenten zusammen. Eine beliebige Anzahl von n Applikationen der gleichen Plattform kann die Middleware parallel nutzen. Die *Application-Adaptation* dient den Anwendungen als Schnittstelle. Sie verteilt Nachrichten, die von der Middleware kommen, an die Anwendung, für die diese Nachricht bestimmt ist. Zur Unterscheidung nutzt sie ein System von selbst generierten IDs – jede angemeldete Applikation erhält eine ID. Nachrichten der Anwendungen an die Middleware werden von der Application-Adaptation mit der ID versehen, sodass die Antwort-Nachricht mit der selben ID der Anwendung zugeordnet werden kann. Das User-Provider-Management verwaltet zur Repräsentation abonnierter Dienste einen ServiceUser pro Dienst und als Stellvertreter registrierter Dienste einen ServiceProvider je Dienst. Das User-Provider-Management bildet

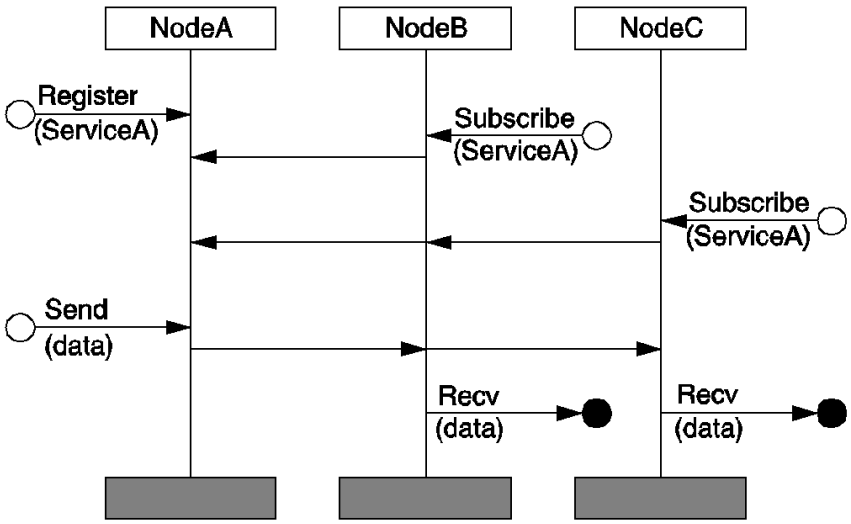

Abbildung 2.3: Kommunikations-Szenario zwischen drei Knoten (entnommen aus [FK07]).

zusammen mit den ServiceUser- und ServiceProvider-Instanzen die *User-Provider-Middleware* der AmICoM. Die *User-Provider-Middleware* verwaltet die Dienste und stellt einen Mechanismus zur Verfügung, um den Applikationen die Registrierung und das Abonnement von Diensten zu erlauben.

Beides wird von einer Applikation mit je einem speziellen Signal an die Middleware ausgelöst. Abbildung 2.3 zeigt die Registrierung eines Dienstes *ServiceA* durch Knoten A (*Node A*), der erst von Knoten B (*Node B*) und anschließend von Knoten C (*Node C*) abonniert wird. Alle Signale, die im Kontext des Dienstes versandt werden, gehen an alle mit dem Dienst assoziierten Knoten – so erhält auch Knoten B die Nachricht von C zum Abonnement des Dienstes, nicht nur der Dienst-Anbieter Knoten A. Daten, die von einem Dienst oder an einen Dienst gesendet werden, werden an alle mit diesem Dienst assoziierten Systeme verteilt. Es ist daher nicht nötig, einzelne Beteiligte zu kennen und direkt zu adressieren. Es ist lediglich der (eindeutige) Dienstname nötig. Jeder Datenversand erfolgt also im Kontext eines konkreten Dienstes. Die Abbildung zeigt auch den Versand von Daten durch den Dienst *ServiceA*. Auch die Daten sind nicht gezielt adressiert – beide Abonnenten erhalten sie. Die Kündigung eines Abonnements oder die Aufhebung einer Registrierung ist ebenso mittels spezieller Signale jederzeit möglich.

Die tieferen Schichten unterhalb der User-Provider-Middleware sind für *Routing* und Medienzugriffskontrolle verantwortlich. Das derzeitige Routing besteht darin, dass Nachrichten an alle Knoten des Netzes verteilt werden, die dann selbst entscheiden, ob die Nachricht für sie von Bedeutung ist. Jede Nachricht ist daher mit einem Dienst assoziiert. Die User-Provider-Middleware prüft dann lediglich, ob auf ihrem Knoten der Dienst abonniert oder registriert wurde. Ist das der Fall, wird die Nachricht angenommen.

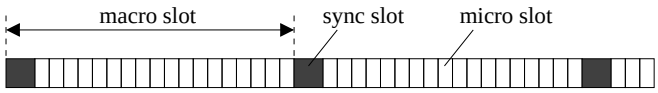

Abbildung 2.4: Aufteilung der Zeit in Synchronisationsphasen und Macroslots (entnommen aus [BGK07]).

2.3 MacZ

Das Medienzugriffsprotokoll MacZ [BGK07] wurde für den Einsatz in Drahtlosnetzwerken entwickelt. Es erzwingt eine strikte Gliederung der Netzzeit in Einheiten fester Länge und Struktur – die sogenannten Macroslots – an die sich alle Kommunikationsteilnehmer halten müssen. Dies ist zur kollaborativen Zugriffssteuerung unerlässlich. Macroslots sind in eine feste Anzahl kleinerer Microslots unterteilt – diese Unterteilung nennt man *Mediumslotting*. Um dieses *Mediumslotting* zu realisieren, müssen die Uhren aller Teilnehmer synchronisiert werden. Hierzu wird BlackBurstSynchronisation eingesetzt [GK08]. Da aber Uhren nie völlig im gleichen Takt laufen, driften die Uhrzeiten der verschiedenen Knoten mit der Zeit auseinander. Daher ist eine regelmäßige Resynchronisation nötig. Die Netzzeit erhält daher die in Abbildung 2.4 dargestellte Aufteilung in Synchronisationsphasen und Macroslots. Die Microslots werden Regionen unterschiedlicher Nutzungsbedingungen zugeteilt, den *Slotregions*. Die Aufteilung ist für alle Teilnehmer bindend. Die folgenden Typen von Slotregionen werden eingesetzt:

- Wettbewerbsbasierte Region (Kurzbezeichnung *Prio*): Der Zugriff auf das Medium ist jedem Teilnehmer gestattet. Doch bevor ein Knoten senden kann, muss er sich um die Zugriffserlaubnis bewerben. Dazu wählt er aus einem bestimmten Bereich zufällig eine Wartezeit. Das Protokoll gestattet es, unterschiedliche Prioritäten zu vergeben. Je nach gewählter Konfiguration des Protokolls, ist der Einfluss der Prioritäten unterschiedlich. Eine Konfiguration gibt Knoten höherer Priorität statistisch kürzere Wartezeiten als Knoten niedrigerer Priorität, eine Andere gibt ihnen garantiert kürzere Wartezeiten. Hat ein Knoten die Wartezeit vollständig abgewartet, und keine anderer Knoten hat die Zugriffserlaubnis erhalten, so darf er mit dem Senden beginnen. Begann in der Zwischenzeit jedoch ein anderer Teilnehmer eine Übertragung, so muss der Bewerber warten bis das Medium wieder frei ist und seine Bewerbung dann fortsetzen. Um zu verhindern, dass eine Übertragung über das Ende der Slotregion hinaus andauert, dürfen nur bis zu einer festen zeitlichen Grenze innerhalb der Region noch neue Übertragungen begonnen werden. Die abschließende, geschützte Phase wird *Receive-Only*-Phase genannt.

- Wettbewerbsfreie Region (Kurzbezeichnung *Res*): Während Res-Slotregionen ist das Senden nur den Teilnehmern erlaubt, die sich Slots innerhalb der Region reserviert haben. Die Reservierung hat netzweite Gültigkeit und ist an Slotnummern innerhalb der Region gebunden. Da eine Kollision von Paketen innerhalb der Region ausgeschlossen werden kann, ist vor dem Versand keine Wartezeit einzuhalten. Die Übertragung darf nicht mehr Zeit beanspruchen als die dafür reservierten Slots belegen.

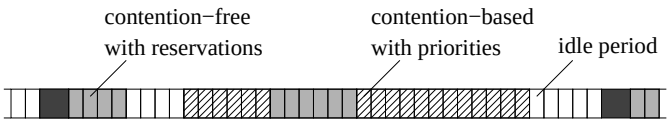

Abbildung 2.5: Aufteilung der Microslots in Slotregionen (entnommen aus [BGK07]).

- Ungenutzte Region (Kurzbezeichnung *Idle*): Die slots sind für keine Nutzung vorgesehen. Kein Teilnehmer darf während dieser Zeit senden.

Eine mögliche Einteilung der Microslots in Slotregionen zeigt Abbildung 2.5. Der Typ der Slotregion bestimmt, wer senden darf und unter welchen Bedingungen – darüber hinaus aber auch noch, was gesendet wird. Es stehen verschiedene Paket-Typen zur Verfügung. So gibt es das Senden von Nutzdaten mit vorheriger Ankündigung nur in der Prio-Region. Dafür stehen die Pakete RTS, CTS, Data und ACK zur Verfügung. Der Sender sendet zuerst ein RTS, erhält er ein CTS vom Empfänger, dann schickt er die Daten, deren Erhalt der Empfänger mit ACK bestätigt. Außerdem ist es noch möglich, ein Datenpaket ohne vorherigen Austausch von RTS und CTS zu versenden. Dies empfiehlt sich vor allem bei sehr kurzen Paketen, da die Kollisionswahrscheinlichkeit geringer ist, und die eigentliche Übertragung durch RTS und CTS unnötig verlängert werden würde. Diese Pakete ohne RTS-CTS-Austausch können sowohl in Res- als auch in Prio-Regionen versandt werden. Als weiterer Paket-Typ existiert noch eine einfache Version Simple des Datenpaketes Data. Es kann ebenfalls in Res- und Prio-Regionen versandt werden und enthält keine Empfängeradresse.

Mit den unterschiedlichen Slot-Regionen ist MacZ leicht an die Anforderungen der Anwendung anzupassen. Fordert die Anwendung z.B. eine geringe Verzögerung der Übertragung, können Kollisionen durch die Zuweisung von Slots in Res-Regionen verhindert werden, was die durchschnittliche Verzögerung senkt. Über die Prioritäten innerhalb von Prio-Regionen kann aber auch bei überwiegend wettbewerbsbasierter Kommunikation sichergestellt werden, dass wichtige Nachrichten schneller übertragen werden.

2.4 Duty-Cycling

Während des Betriebs von Systemen wird vorhandene Hardware oftmals nur zeitweise benötigt. Trozdem benötigen ihre Komponenten durchgehend Energie. Dabei reicht es für manche Komponenten, wie z.B. Sensoren oder Transceiver sie nicht ständig vorzuhalten, sondern nur zeitweise Einzuschalten. Denn ihr Einsatz kann auf regelmäßige, kurze Nutzungsphasen beschränkt werden, gefolgt von längeren, inaktiven Phasen, ohne die Leistung der Anwendung signifikant zu verschlechtern. Jede Zeitspanne, während der solche Hardwarekomponenten ungenutzt sind, bedeutet daher einen potentiell vermeidbaren Energieverbrauch. Ob solche Verbräuche tatsächlich sinnvoll vermieden werden können und das Einsparpotential deshalb genutzt werden sollte, hängt davon ab, ob und in welcher Geschwindigkeit die Hardware, wenn nötig, wieder eingeschaltet werden kann. Ist der Zeitpunkt der nächsten

Nutzung bekannt und die Zeit bis dorthin länger als die Summe aus Abschaltzeit und Anfahrzeit der Hardwarekomponente, dann kann Energie gespart werden.

Ein Mechanismus zum Energie-Sparen, der nach diesem Prinzip arbeitet, ist das sogenannte *Duty-Cycling* [YHE02], [DSJ07], [vDL03], [SDGJ08]. Ein *Duty-Cycling*-Mechanismus legt für eine Hardwarekomponente Arbeitszyklen fest, in denen die Komponente einsatzbereit ist. Zwischen den Zyklen wird die Komponente in einen energiesparenden Zustanden versetzt und ist dann nicht einsetzbar. Möglicherweise existieren mehrere Zustände zwischen denen der Mechanismus auswählen muss. Die so gesteuerten Hardware-Komponenten kann die CPU, der Transceiver, ein Sensor oder ein anderes Bauteil sein, das Energie verbraucht. In dieser Arbeit liegt das Hauptaugenmerk auf der Steuerung des Transceiver-Zustandes. Ein Duty-Cycling-Mechanismus für einen Transceiver sieht vor, dass ein Knoten zur Laufzeit zwischen aktiven Phasen des Transceivers, in denen er über das Medium erreichbar ist und auch darauf zugreifen kann, und passiven Phasen, in welchen der Knoten nicht erreichbar ist, wechselt. In der Regel führt er diese Wechsel während seiner gesamten Lebensdauer durch. Der Anteil der aktiven Phasen an der gesamten Lebensdauer ist dabei der *Duty-Cycle* (Einschaltdauer). In der Regel ist das Intervall zwischen dem Beginn einer Wachphase und dem Beginn der nächsten Wachphase streng periodisch. In diesem Fall entspricht der Duty Cycle dem Verhältnis einer Wachphase zu einer Periodendauer. Der Mechanismus wurde als Grundlage für das Medienzugriffsprotokoll S-MAC von Estrin et al. [YHE02] eingeführt und bedarf einiger grundlegender Voraussetzungen:

1. Knoten, die kommunizieren wollen, müssen synchronisiert sein. Es wird zwischen absoluter Uhrzeit (clock-synchronized) und periodensynchroner Uhrzeit (tick-synchronized) unterschieden.

2. Aktive Phasen der Nachbarknoten müssen bekannt sein. Absolute Zeiten im Falle von clock-synchronization und relativ zum Beginn einer Periode im Falle von tick-synchronization (z.B. "30 ms nach Beginn einer Periode für eine Dauer von 100 ms").

3. Knoten müssen die festgelegten Schedules, sowohl ihren eigenen als auch die der Nachbarn, einhalten.

Die Autoren von S-MAC identifizieren vier verschiedene Quellen von unnötigem Energieverbrauch:

- Overhearing: Empfang von Nachrichten, die nicht für den eigenen Knoten bestimmt sind (diese sind aber nicht zwangsläufig bedeutungslos für empfangende Fremdknoten)

- Idle-Listening: Ein Knoten ist empfangsbereit, es werden allerdings keine Nachrichten empfangen, da andere Knoten zur Zeit nicht senden.

- Interferenz: Die Übertragung einer Nachricht wird durch einen zweiten sendenden Knoten gestört und muss daher wiederholt werden.

- Overhead: Durch Management-Nachrichten wird zum Senden und Empfangen ebenfalls Energie verbraucht, es werden jedoch keine Nutzdaten der Anwendung übertragen.

Überhöhter Energieverbrauch aus den ersten drei Quellen kann durch Duty-Cycling vermindert werden.

2.5 Überblick über existierende Duty-Cycling-Verfahren

Im nachfolgenden Abschnitt werden einige Duty-Cycling-fähige Protokolle vorgestellt, um verschiedene Varianten des Konzeptes und mögliche Vor- und Nachteile zu beleuchten. Die Auflistung hat nicht den Anspruch vollständig zu sein, enthält aber wichtige Vertreter.

2.5.1 S-MAC

Mit Sensor-MAC (S-MAC) [YHE02] stellen die Autoren Ye, Heidemann und Estrin ein speziell für Sensornetzwerke entwickeltes MAC-Protokoll vor. Das vorherrschende Entwicklungsziel war dabei die Reduzierung des Energieverbrauchs, was durch den Low-Duty-Cycle-Betrieb der beteiligten Sensorknoten erreicht wird. Knoten durchlaufen immer wieder die gleichen Phasen in einem Zyklus. Zu Beginn steht eine Synchronisationsphase, in der sich die Knoten resynchronisieren, um den Uhrenversatz auszugleichen. Dabei tauschen sie auch Informationen aus, wann sie das nächste mal in die Schlafphase wechseln. Nach der Synchronisation sind die Knoten empfangsbereit. Für wie lange sie in dieser Empfangsphase bleiben, ist frei konfigurierbar, jedoch für alle Knoten einheitlich. Gängige Duty-Cycles reichen von 1% bis 10% , was bedeutet, dass sie 1% bzw. 10% der gesamten Laufzeit des Netzes in einem aktiven Zustand sind. Das gleiche Verhältnis von Wach- und Schlafdauer gilt auch für einen einzelnen Zyklus. Ist die Zeit der Wachphase abgelaufen, schalten die Knoten ihre Sende- und Empfangs-Hardware ab, befinden sich dann folglich in einer Schlafphase. Erst zum Beginn der nächsten Synchronisationsphase wachen die Knoten wieder auf.

Damit die Knoten zum gleichen Zeitpunkt in ihre Wachphase eintreten und zum gleichen Zeitpunkt ihre Wachphase beenden, müssen sie sich synchronisieren. Das konkrete Synchronisationsverfahren lassen die Autoren aber unbestimmt, sondern verweisen lediglich auf gängige Verfahren, die sicherstellen müssen, dass eine Ausreichende Synchronisationsgenauigkeit erreicht wird.

Das Verhalten der Knoten ist wie folgt: Knoten warten nach dem Einschalten eine feste Zeitspanne. Erhalten sie in dieser Zeit keine Synchronisationsnachricht und damit den Schedule eines anderen Knotens, wählen sie ihren eigenen Schedule und beginnen diesen zu senden. Für dessen Versand müssen sie sich allerdings nach dem gleichen Schema um das Medium bewerben, das auch für den normalen Datenversand eingesetzt wird. Es ist daher unwahrscheinlich, dass die Knoten genau zum

gleichen Zeitpunkt beginnen, ihren Schedule zu versenden. Empfangen Knoten einen Schedule, bevor sie ihren eigenen versenden konnten, so verwerfen sie ihren eigenen und übernehmen den empfangenen. Knoten in Reichweite bilden auf diese Art einen synchronisierten Verbund (virtueller Cluster).

Sollten sich mehrere Zeitpläne etablieren, so sind Knoten an der Grenze zwischen zwei Regionen des Netzes mit getrennten Zeitplänen in beiden Wachphasen aktiv. Hier wird ein Nachteil des Protokolls offenbar: Diese Knoten brauchen ihre Batterie schneller auf und damit drohen die Verbindungsknoten zwischen virtuellen Clustern zuerst auszufallen. Der Zugriff auf das Medium ist wettbewerbsbasiert. Aus einem Intervall wird zufällig eine Wartezeit gewählt. Tritt während dieser Wartezeit eine Belegung des Mediums auf, so unterbricht der Knoten seine Bewerbung um das Medium und wartet, bis die Belegung vorüber ist. Anschließend setzt er seine Wartezeit bis zu deren Ende fort. Will ein Knoten in einer Wachphase einen Datenversand einleiten, so muss er sich um das Medium bewerben. Ist die gewählte Wartezeit abgelaufen, sendet er im Anschluss eine RTS-Nachricht (Request-to-Send). Hierdurch wird anderen Knoten die Sende-Absicht mitgeteilt, sie versuchen dann nicht ihrerseits, Nachrichten zu senden. Ist der Empfängerknoten empfangsbereit, wird er mit einem CTS (Clear-to-Send) reagieren. Unbeteiligte Knoten, die ein RTS oder CTS empfangen, können die enthaltene Übertragungsdauer der angekündigten Übertragung auslesen und vermerken, dass das Medium für diesen Zeitraum belegt sein wird (virtual carrier sensing). Sie werden in dieser Zeit keine Übertragung beginnen.

Die Wachphase ist so bemessen, dass eine Synchronisation stattfinden kann und zwei Knoten mittels RTS und CTS eine Übertragung vereinbaren können. Denn während die vereinbarte Übertragung durchgeführt wird, wechseln unbeteiligte Knoten in ihre Schlafphase bis zum Beginn der nächsten Periode, die wieder mit einer Synchronisation beginnt. Ist die Übertragung abgeschlossen und war erfolgreich, quittiert der Empfänger dies mit einer ACK-Nachricht und kann seinen Transceiver abschalten.

Ein Nachteil ist die Tatsache, dass für den Versand einer Nachricht über mehrere Hops ein Sender warten muss, bis der nächste Knoten aufwacht, um sie weitergeben zu können. Kann pro Wachphase nur ein Hop überwunden werden, entspricht die Ende-zu-Ende-Verzögerung (*latency*) dem Produkt aus Hop-Anzahl und der Periodendauer. Eine Erweiterung von S-MAC [YHE03] adressiert dieses Problem und führt sogenanntes *adaptive listening* ein. In dieser Version von S-MAC verhalten sich die Knoten anders, wenn sie ein RTS oder CTS empfangen, das nicht an sie gerichtet ist. Sie lesen die geplante Dauer der Datenübertragung aus dem *Header*, legen sich für diese Dauer schlafen, aber erwachen am Ende der Übertragung nochmals für einen kurzen Moment (adaptive listen interval). Auf diese Weise kann der Empfängerknoten jetzt mit einem weiteren RTS an den wieder erwachten Knoten eine weitere Übertragung der Daten einleiten. Erhält der erwachte Knoten kein RTS, so kann er sich wieder schlafen legen. Da diese Übertragungen während der Schlafphase der übrigen Knoten stattfinden, muss sichergestellt sein, dass sie beendet werden können, bevor die Schlafphase zu Ende ist, um die Synchronisation des neuen Intervalls nicht zu stören. Daher wird kein Knoten ein *adaptive listening* durchführen oder den Weiterversand einleiten, wenn die verbleibende Zeit bis zum Beginn der neuen Periode kürzer als das *adaptive listen interval* ist.

Als weitere Verbesserung wurde *message passing* eingeführt. Um zu vermeiden, dass bei einer Verfälschung eines Datenrahmens der gesamte Rahmen wiederholt werden muss, wird er in Fragmenten versandt. Der korrekte Empfang eines Fragmentes wird mit einem ACK quittiert. Bleibt ein ACK aus, wiederholt der Sender das entsprechende Fragment. Es erfolgt aber nur einmal ein RTS-CTS-Austausch, der das Medium für die gesamte Dauer der Übertragung reserviert. Da aber auch jedes Fragment und jedes ACK die Restdauer der Übertragung enthalten, können Knoten, die zu früh aufwachen, weil beispielsweise die Wiederholung einiger Rahmen die ursprüngliche Übertragungsdauer verlängerte, dies erkennen und weiterschlafen. S-MAC bietet mit den vorgestellten Mechanismen ein Protokoll, das den Energieverbrauch deutlich senkt, aber dafür eine gesteigerte Übertragungsverzögerung in Kauf nimmt.

2.5.2 T-MAC

Die Autoren von T-MAC (Timeout-MAC) [vDL03] greifen in ihrer Arbeit die in S-MAC [YHE02] verwandten Mechanismen zum Energiesparen auf. Die Aufteilung eines Arbeitszyklus in eine aktive und eine passive Phase wird auch hier als Instrument eingesetzt, um den Energieverbrauch durch *idle listening* – von den Autoren als Hauptgrund für Energieverschwendung identifiziert – zu senken. Sie sehen in der statischen Aufteilung eines Arbeitszyklus in S-MAC jedoch keine optimale Lösung, da diese das tatsächliche Aufkommen von Übertragungen nicht berücksichtigt. Daher ist in T-MAC zwar der Anfang eines neuen Zyklus und damit der Anfang der Aktiv-Phase zeitlich festgelegt und streng periodisch, das Ende der Aktiv-Phase allerdings flexibel. Während der Schlafphase können Nachrichten nicht versandt werden und sammeln sich in den Absenderknoten an. Beginnt nun eine Aktiv-Phase, versuchen alle Knoten, ihre Nachrichten zu senden. Um die Wahrscheinlichkeit von Kollisionen zu senken, wird dafür ein *Contention*-Schema eingesetzt wie es aus gängigen Protokollen bekannt ist. Sender müssen sich erst um das Medium bewerben, indem sie eine zufällige Wartezeit wählen und nur senden, wenn in dieser Zeit kein anderer Sender das Medium belegt. Nach erfolgreicher Bewerbung tauschen Sender und Empfänger RTS und CTS aus; wurden die Daten gesendet, quittiert der Empfänger dies mit einem ACK. Die Knoten können so lange senden, wie sie noch ungesendete Nachrichten haben.

Wurde für eine festgelegte Zeit TA keine Aktivität im Medium registriert, so löst das einen *timeout* aus und der Knoten beginnt seine Schlaf-Phase bis zum nächsten Zyklus. Dafür führen die Autoren den Begriff der *activation events* ein. Dieser bezeichnet Ereignisse, die zum erneuten Setzen des Timers für TA führen. Erst wenn also für die Zeit von TA kein *activation event* auftrat, schaltet der Knoten seinen Transceiver ab. Diese Ereignisse sind:

- der Beginn eines neuen Zyklus,
- der Datenempfang am Transceiver,
- das Stattfinden von Kommunikation auf dem Medium,

- das Ende der eigenen Daten- oder ACK-Übertragung und
- das aus einem fremden RTS oder CTS errechnete Ende einer Datenübertragung eines anderen Knotens.

Wie in S-MAC etablieren sich virtuelle Cluster von Knoten mit dem gleichen Schedule. Wenn Knoten frisch eingeschaltet werden, warten sie eine Zeit ab, ob sie die SYNC-Meldung anderer Knoten empfangen, deren Schedule sie dann übernehmen würden. Tun sie das nicht, senden sie einen selbstgewählten Schedule zusammen mit ihrer SYNC-Nachricht. Miteinander synchronisierte Knoten bilden einen Cluster. Knoten, die mehrere Schedules empfangen, übernehmen ebenfalls die Startzeiten der anderen und wachen zum Beginn mehrerer Aktiv-Phasen auf.

Wie zuvor erwähnt, wird eine Datenübertragung mit einem RTS eingeleitet. Wenn der Empfänger nicht antwortet, kann das mehrere Gründe haben. Es kann eine Kollision beim Empfänger aufgetreten sein, der Empfänger könnte durch eine laufende Übertragung – entweder als Beteiligter oder als Nachbar eines Beteiligten – daran gehindert sein, zu antworten, oder der Empfänger ist schon im Schlaf-Modus. Würde der Absender des RTS nun die Zeit TA ohne Antwort abwarten und sich dann abschalten, so wäre die Chance auf eine erfolgreiche Übertragung noch in diesem Zyklus vertan, obwohl in den ersten beiden Fällen ein weiteres Abwarten die Übertragung noch ermöglicht hätte. Daher entschieden sich die Autoren für eine zweimalige Wiederholung der RTS-Nachrichten, bevor ein Sender den Sendeversuch aufgibt und bis zum nächsten Zyklus wartet.

Empfangen Knoten nicht-für-sie bestimmte RTS oder CTS, bleiben sie wach, da ihnen möglicherweise im Anschluss eine Nachricht übermittelt werden soll. Die Zeitspanne TA muss aber lang genug bemessen sein, um diese Fremd-Übertragung zu registrieren. Ein Knoten könnte außer Reichweite für das RTS an den Nachbarknoten sein. Um den Beginn des antwortenden CTS zu empfangen, muss TA daher länger als die Summe aus maximaler Wartezeit C, RTS-Übertragungsdauer R und der Dauer T zwischen Empfang des RTS und Antwort-CTS sein.

Es ist möglich, den Transceiver nach solchen an andere Knoten gerichteten RTS oder CTS-Signalen bis zum Ende der Datenübertragung zeitweilig abzuschalten, um *Overhearing* der Übertragung zu vermeiden. Die Autoren von T-MAC führen aber an, dass ein Knoten weitere RTS oder CTS während dieser Zeit nicht wahrnehmen kann und unmittelbar im Anschluss an sein Wiedereinschalten des Transceivers durch Senden andere Knoten stört.

Der eingeführte *Timeout*-Mechanismus könnte dazu führen, dass Knoten ihre Transceiver abschalten, bevor sich ein Sender – der eine Nachricht für sie hat – erfolgreich um das Medium bewerben konnte. Die Autoren nannten dieses Phänomen *early sleeping*. Sie führten zu dessen Begegnung einen *Future-Request-To-Send*-Rahmen (FRTS) ein. Wer ein CTS empfängt und daraus ersehen kann, dass der eigene Sendewunsch warten muss, kann unmittelbar nach dem CTS ein FRTS senden, um ein verfrühtes Abschalten des Zielknotens zu unterbinden. Damit das FRTS die Datenübertragung nicht stört, wird diese um die Dauer des FRTS verzögert. Das FRTS enthielt zudem die Dauer der Datenübertragung, die der FRTS-Sender aus dem empfangenen CTS ersehen konnte. Somit kann der angesprochene Knoten bis zum Ende

der Übertragung – schlafend oder wach – warten, um noch oder wieder wach zu sein, wenn das Medium wieder frei ist.

Für den Fall, dass ein Knoten bevorzugt senden möchte statt zu empfangen, da sein Nachrichtenpuffer voll ist, führten die Autoren die *full-buffer priority* ein. Empfängt ein solcher Knoten ein RTS – sollte sich also für den Empfang bereit halten – kann er seinerseits unmittelbar ein RTS versenden um die Übertragung seiner eigenen Nachricht einzuleiten. Er muss sich in diesem Fall nicht um das Medium bewerben, was für ein Antwort-CTS auch nicht nötig gewesen wäre. Der erste RTS-Sender muss sich zu einem späteren Zeitpunkt noch einmal um das Medium bewerben. Allerdings sollte *full-buffer priority* erst zum Einsatz kommen, wenn sich ein Knoten vorher mindestens zweimal vergeblich um das Medium beworben hat.

Mit T-MAC wird das Konzept von S-MAC weiterentwickelt und um eine dynamische Wachphase erweitert, wodurch die *idle-listening*-Zeiten der Knoten gesenkt werden. Energieverschwendung tritt aber immer noch bei Kollisionen auf – einen Mechanismus, um weniger Kollisionen als S-MAC zu erzielen, bietet T-MAC nicht.

2.5.3 RMAC

Das Protokoll RMAC (*routing enhanced MAC*) [DSJ07] nutzt die Informationen der Routing-Schicht, um Nachrichten energie-effizienter zu versenden. Wie vergleichbare Protokolle für Sensornetzwerke setzt auch RMAC Duty-Cycling ein, um den Stromverbrauch zu senken. Daher ist die Laufzeit der Knoten in immerwiederkehrende Intervalle unterteilt, die ihrerseits jeweils in drei Phasen geteilt sind. Den Beginn jedes Intervalls bildet eine SYNC-Phase, in der die Knoten miteinander synchronisiert werden. Die Autoren von RMAC gehen nicht darauf ein, welches Verfahren zur Synchronisation eingesetzt wird, setzen jedoch voraus, dass diese Aufgabe von einem separaten Verfahren mit ausreichender Genauigkeit durchgeführt wird. Darauf folgt eine Data-Phase und zuletzt eine Schlaf-Phase.

Während der Data-Phase kann ein Knoten, der Daten zu versenden hat, den Versand vorbereiten. Dazu sendet er einen *Pioneer-Frame*, kurz PION, an den nächsten Hop. Wie in anderen wettbewerbsbasierten Protokollen, wird vor dem Versand eines PIONs eine zufällige Wartezeit gewählt und *Carrier Sense* durchgeführt. Ein PION enthält die Adresse des Absenders, die Adresse des nächsten Hop, die Dauer der Übertragung, die Zieladresse des endgültigen Empfängers und die Anzahl bisher passierter Hops. Die endgültige Zieladresse muss die Routing-Schicht deshalb beim Nachrichtenversand an RMAC weitergeben. Der PION dient wie in anderen Protokollen das RTS der Ankündigung der Übertragung beim Empfänger. Dieser antwortet ebenfalls mit einem PION. Der Antwort-PION ist aber gleichzeitig die Sende-Anfrage an den nächsten Hop. Dieser zweite PION wird ein zusätzliches Adressfeld des Frames besetzen: die Adresse des vorhergehenden Knotens. Damit kann dieser den zweiten PION eindeutig als seine Antwort identifizieren. Der angegebene Empfänger im zweiten PION wird diese Kette fortsetzen. Das geschieht solange, bis die Data-Phase zu Ende ist oder der endgültige Empfänger erreicht wurde. Eine Kette von PION-Frames informiert alle beteiligten Knoten auf dem Weg eines zu sendenden Datenframes zu seinem Ziel.

Der tatsächliche Datenversand erfolgt erst in der Schlaf-Phase. Ist die Data-Phase zu Ende, schalten alle Knoten bis auf den Initiator einer Kette von PIONs und seinen nächsten Empfänger ihre Empfangs-Hardware ab. Die Knoten wachen immer genau dann wieder auf, wenn sie an der Reihe sind, das Data-Frame zu empfangen und weiterzusenden. Der erfolgreiche Empfang wird mit einem ACK quittiert.

Dieses Scheduling der Aufwachzeitpunkte wird durch die PIONs erreicht: Der Initiator setzt die Anzahl der überwundenen Hops auf 0, jeder weitere Sender erhöht den Wert um 1. Wer ein PION in der Wachphase erhält, kann aus der im Frame enthaltenen Übertragungsdauer des Dataframe $durDATA$, der Dauer eines ACK $durACK$, der Dauer des *Short Inter Frame Spacing* (SIFS) und der Anzahl überwundener HOPS seinen Aufwachzeitpunkt berechnen. Die Autoren geben zur Berechnung der Aufwachzeit $T_{wakeup}(i)$ des i-ten Hop folgende Formel an:

$$T_{wakeup}(i) = (i-1) \cdot (durDATA + SIFS + durACK + SIFS)$$

Die Formel gibt die Zeitspanne vom Beginn der Schlafphase bis zum Aufwachzeitpunkt an.

Jeder Knoten in RMAC führt zudem einen *Network Allocation Vector* (NAV), um virtuelles *Carrier Sense* zu betreiben. Ist der Vektor gesetzt, lässt sich daraus eine momentane Belegung des Mediums ablesen, und ein etwaiger Sendewunsch muss zurückgestellt werden. Wenn ein PION empfangen wird, der nicht an den eigenen Knoten gerichtet ist, werden im NAV drei verschiedene Segmente als belegt markiert:

- Der zeitlich unmittelbar folgende Bereich für die Dauer eines PION (Segment 1). Da der Absenders des PION einen Antwort-PION erwartet, würde eine Übertragung des eigenen Knoten diese Antwort stören.
- Der Zeitpunkt des Datenversands an den Absender des PION (Segment 2) für die Dauer der Datenübertragung. Aus dem PION lässt sich mit der Formel für $T_{wakeup}(i)$ berechnen, zu welchem Zeitpunkt der Absender des PION selbst den Datenrahmen erhält und wie lange der Empfang dauert.
- Der Zeitpunkt der ACK-Nachricht an den Absender des PION (Segment 3) für die Dauer der Nachricht. Um den Empfang der ACK-Nachricht an den Absender des PION nicht zu stören, nachdem dieser die Daten weiter versandt hat, muss auch dieses Segment gesperrt werden.

Erhält ein Knoten ein PION und ist tatsächlich der adressierte Empfänger, so wird er seine Aufwachzeiten berechnen und nur dann, wenn sich sowohl der unmittelbare Versand des Antwort-PION, als auch der Versand eines ACK und der Weiterversand der Daten in der Schlafphase nicht mit belegten Segmenten im NAV überschneiden antworten. Bleibt ein Antwort-PION aus, so wird es der Absender im nächsten Zyklus erneut versuchen.

Ein PION kann allerdings auch verloren gehen und der Absender deshalb keine Antwort erhalten. Da der Sender nicht unterscheiden kann, ob der nächste Hop die Weiterleitung verweigert oder den PION nicht empfangen hat, wird er im laufenden Zyklus keinen PION wiederholen. Geht der Antwort-PION verloren, bedeutet

das, dass der antwortende Knoten zur errechneten Zeit aufwacht. Da der Sender des Datenpaketes allerdings keine Antwort erhielt, sendet er die Daten nicht. In diesem Fall wird ein Timeout den wartenden Knoten nach einiger Zeit wieder schlafen legen. Es kann allerdings passieren, dass auf diesem Wege eine ganze Kette von Knoten unnötigerweise aufwacht und bis zum Timeout vergeblich wartet, weil an einer Stelle der Antwort-PION nicht erkannt, aber von den nächsten Hops empfangen und weitergereicht wurde. Bleibt die ACK-Nachricht auf einen Datenversand aus, so wird der Sender die Nachricht nicht im selben Zyklus wiederholen, da sich dadurch eine zeitliche Verschiebung der Übertragung ergäbe, die die Schedules sämtlicher nachgelagerter Knoten nicht berücksichtigen. Der Sender wird mit einem neuen PION in der nächsten Data-Phase die Übertragung neu *schedulen*. Das bedeutet aber auch eine mögliche Duplikation der Nachricht, wenn die Daten korrekt empfangen wurden und nur das ACK verloren ging. Die Autoren gehen darauf nicht näher ein.

RMAC ist im Gegensatz zu S-MAC in der Lage, Nachrichten in einem Zyklus mehrere Hops weit zu transportieren. Duty-Cycling und das gezielte Aufwecken von Knoten, wenn sie gebraucht werden, reduzieren den Energieverbrauch der Knoten. Eine Kette von Datenübertragungen, vorher durch PION-Frames festgelegt, kann allerdings immer nur zum Beginn der Schlafphase begonnen werden. Benachbarte Knoten können daher nicht im selben Zyklus die Initiierer von Nachrichten sein.

2.5.4 DW-MAC

Das *Demand Wakeup MAC* (DW-MAC) [SDGJ08] wurde mit dem Ziel entwickelt, ein energiesparendes MAC-Protokoll für Sensornetzwerke zu schaffen, ohne jedoch den in anderen Protokollen in Kauf genommenen Übertragungsverzögerungszuwachs. In DW-MAC wiederholt sich wie in S-MAC ein in Phasen aufgeteiltes Intervall. Die erste Phase ist eine Sync-Phase und dient der Synchronisation der Knoten. Darauf folgt eine Data-Phase, dies ist die eigentliche Wachphase, in der alle Knoten erreichbar sind. Die Autoren gehen nicht auf einen konkreten Synchronisationsalgorithmus ein, sondern treffen die Annahme, dass ein separates Protokoll eingesetzt wird, dessen Aufgabe die Synchronisation mit einer ausreichenden Genauigkeit ist. Die letzte Phase bildet auch hier wie bei S-MAC die Schlafphase. In der Schlafphase wachen Knoten nach ihrem Schedule auf, um tatsächlich Daten auszutauschen. Das nötige *Scheduling* fand in der Wachphase zuvor statt. DW-MAC führt die Medienzugriffskontrolle (medium access control - MAC) und das *Scheduling* für den Datenversand in der Schlafphase mit einem speziellen Mechanismus gleichzeitig durch: Zwischen der Wach-Phase (Data-Phase) und der anschließenden Schlafphase besteht eine Abbildung, die jedem Zeitpunkt innerhalb der Wach-Phase eindeutig einen Zeitpunkt der Schlafphase zuordnet.

Ein Scheduling-Rahmen, der in der Wachphase versandt wird, reserviert den zugehörigen Bereich der Schlafphase. Das Verhältnis vom Abstand A_1, den der Sendezeitpunkt des Scheduling-Rahmens vom Beginn der Wachphase hat, zur Länge der Wachphase D_w, ist das gleiche Verhältnis wie das zwischen dem Abstand A_2 – vom Beginn der Reservierung zum Beginn der Schlafphase – und der Länge der Schlafphase D_s. Dieser Zusammenhang besteht auch zwischen der Länge des SCH-Rahmens und der Länge des reservierten Bereiches, sie korrespondieren auch unmit-

telbar miteinander. Hat ein Knoten Daten zu versenden, so wird er zur Vermeidung von Kollisionen ein zufälliges Back-off-Intervall wählen und sich um das Medium bewerben. Bei erfolgreicher Bewerbung sendet er dann einen Scheduling-Rahmen SCH an den Empfänger. Sender und Empfänger können so zum vereinbarten Zeitpunkt aufwachen und Daten austauschen. Auf diese Weise ist sichergestellt, dass nur Knoten, die auch wirklich an der Übertragung beteiligt sind, ihre Schlafphase unterbrechen. Neben dem Scheduling der Wachzeiten erfüllen die SCH-Rahmen auch den Zweck, Kollisionen zu verhindern. Denn würden beim Empfänger während der Datenübertragung zwei Übertragungen kollidieren, so wären auch schon die zugehörigen SCH-Rahmen kollidiert, der Empfänger hätte sie also gar nicht korrekt dekodieren können und wäre in der Schlafphase gar nicht erst erwacht.

Das Protokoll lässt sowohl *Unicasts* als auch *Broadcasts* zu. Der SCH-Rahmen enthält die Information, ob es sich um einen Uni- oder Broadcast-Rahmen handelt. Ist es ein Unicast-Rahmen, so antwortet der Empfänger nach einem *SIFS* nach dem SCH mit einem SCH-Rahmen an den Absender. Damit reserviert er auch gleichzeitig einen Zeitraum zum Versenden des ACK nach erfolgreich empfangenem Datenversand. Handelt es sich um einen Broadcast, so braucht der Empfänger nicht zu antworten und nach dem Datenempfang auch nicht mit einem ACK zu quittieren. Anhand der Absenderadresse und der Sequenznummer im SCH kann ein Empfänger außerdem entscheiden, ob er einen Broadcast-Rahmen schon einmal empfangen hat und wird dann im zugehörigen reservierten Zeitraum während der Schlafphase nicht aufwachen.

Ist ein Empfänger nur Mediator für einen Rahmen und nicht der endgültige Empfänger, so kann er sich unmittelbar nach Erhalt des SCH seinerseits um das Medium bewerben um ein SCH an den nächsten Hop zu senden. So kann ein Rahmen durch *multihop forwarding* mehrere Hops innerhalb eines Zyklus überwinden. Die antwortenden SCH-Rahmen während der Wach-Phase sind dann (ähnlich zu RMAC [DSJ07]) gleichzeitig die Antwort an den Sender und das Schedule-Signal für den nächsten Empfänger.

Um den Versand von Broadcast-Nachrichten zu optimieren, können Routing-Informationen höherer Ebenen genutzt werden. So kann für eine Broadcast-Nachricht ein *immediate forwarder* bestimmt werden, der ein SIFS im Anschluss an das empfangene SCH selbst ein SCH für den *Rebroadcast* zu senden versucht. Andere Empfänger würden nur mit etwas Abstand versuchen, die Broadcast-Nachricht zu wiederholen. So kann ein Broadcast während eines Zyklus gezielt auf einem Pfad möglichst weit transportiert werden.

DW-MAC ähnelt bis auf den Scheduling-Mechanismus, der ohne zusätzliche Zeitangaben im SCH-Rahmen auskommt, stark dem RMAC-Verfahren. Es ist aber dahingehend flexibler, dass es zwei benachbarten Knoten gestattet, im gleichen Zyklus Daten zu versenden, da sie nur zu unterschiedlichen Zeiten ein SCH versenden müssen. In RMAC würden beide Sender ihren Datenversand genau zum Beginn der Schlafphase starten und sich gegenseitig stören, weshalb ein solcher Versand in RMAC nicht innerhalb des selben Zyklus stattfinden kann. Die Autoren können darlegen, dass DW-MAC einen höheren Datendurchsatz und eine geringere Ende-zu-Ende-Verzögerung als S-MAC erzielt.

Kapitel 3

WNCS-CoM

Drahtlose Regelungssysteme (WNCS) benötigen speziell adaptierte Kommunikationsprotokolle, um geeignet arbeiten zu können. Hierzu wurde die AmICoM [FK07] um Funktionen erweitert. Die daraus resultierende Kommunikations-Plattform, die für solche Regelungssysteme entworfen wurde, heißt im folgenden WNCS-CoM.

Abbildung 3.1 zeigt die jeweils korrespondierenden Komponenten der beiden Systeme.

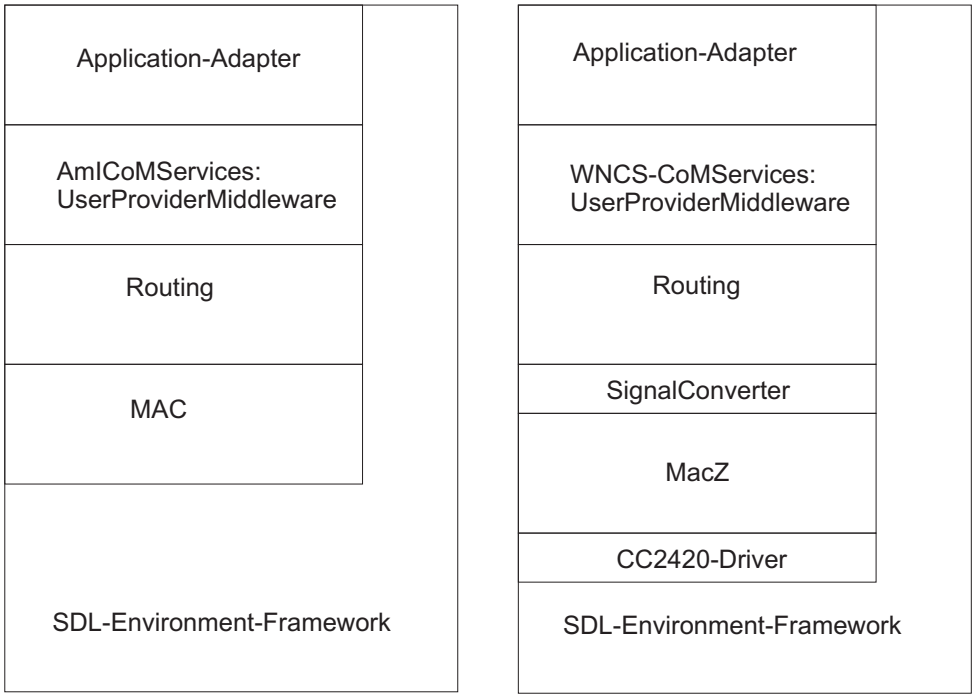

Abbildung 3.1: Architekturen der AmICoM (links) und WNCS-CoM (rechts).

Als Beispiel für ein drahtloses Regelungssystem wird das inverse Pendel gewählt. Die WNCS-CoM ist die nötige Middleware für das Regelungssystem dieses Pendels. Die Komponenten der Pendelsteuerung sind Sensoren, ein Regelungsknoten (Controller) und ein Aktuator. Sie alle sollen drahtlos miteinander vernetzt sein. Außerdem soll die Middleware ein Publish-Subscribe-Framework für die komfortable Nutzung von Diensten bereit stellen, die von anderen Knoten angeboten werden.

Die Anwendung, die das Steuerungssystem des inversen Pendels darstellt, soll auf Dienste zugreifen, die hochwertige Leistungen anbieten. So soll die Pendelkontrolle mehrere Messdienste abonnieren können, um periodisch Messwerte zu erhalten. Die Kontrolle kann daraus Stellwerte berechnen, mit denen der Aktuator gesteuert wird.

Die Messwertdienste verfügen je über einen Sensor und messen, sobald sich ein Abonnent angemeldet hat. Der Abonnent kann bei der Anmeldung über einen Parameter die Abstände, in denen er weitere Messwerte erhält, bestimmen.

Das Steuerungssystem stellt hohe Anforderungen an die von der Netzwerk-Middleware bereitgestellte Dienstgüte. Abschnitt 3.1 erfasst und erläutert die Dienstgüte-Anforderungen des Systems.

Die Middleware, die nötig ist, um diese Dienste realisieren zu können, wurde in SDL spezifiziert. Die Anwendungslogik der Dienste wird in der Programmiersprache C geschrieben. Um dem Programmierer der Anwendungslogik die Nutzung der Middleware zu ermöglichen, wurde ein Adapter entwickelt. Dieser ist ebenfalls zum Teil in C geschrieben und lässt sich daher mit den Mitteln, die diese Sprache bietet, in eine konkrete Anwendung einbinden. Dieser Adapter wird in Abschnitt 3.2 beschrieben.

Die Grundlage für die Dienste wird von einer Middleware bereitgestellt. Sie verwaltet ein System zum Registrieren, Auffinden und Abonnieren von Diensten sowie den periodischen Versand von Daten, falls sich ein Abonnent dafür angemeldet hat. Sie wird in Abschnitt 3.3 vorgestellt.

Um garantieren zu können, dass die periodisch ermittelten Messwerte tatsächlich in den vom Abonnenten benötigten Abständen übertragen werden, muss der Medienzugriff bei einem gemeinsam genutzten Medium streng geregelt und der Anwendung angepasst sein. Da die gesamte Anwendung auf Sensorknoten eingesetzt wird, denen üblicherweise nur Batterien als Energiequelle zur Verfügung stehen, muss das System Möglichkeiten zum Einsparen von Energie nutzen. Das eingesetzte Medienzugriffsprotokoll wurde daher um Duty-Cycling erweitert und speziell auf das zeitliche Auftreten der Anwendungsdaten abgestimmt. Die vorgenommenen Erweiterungen werden in Kapitel 4 erläutert.

3.1 Dienstgüte-Anforderungen der Anwendung

Das inverse Pendel ist ein instabiles System, das ständiger Kontrolle und Steuerung bedarf, wenn es aufrecht erhalten werden soll. Daher stellt das Steuerungssystem an die Netzwerk-Middleware hohe Anforderungen, die bereitgestellte Dienstgüte betreffend. Diese Anforderungen werden nachfolgend formal aufgestellt.

Die Aufstellung folgt dem in [WG07] dargelegten formalen Ansatz für Netzwerk-Dienstgüte. Danach sind die konkreten Anforderungen je nach Sicht (z.B. aus Sicht der Anwendung oder aus Sicht der Hardware) unterschiedlich zu formulieren. So finden sich darin aus Sicht der Anwendung Angaben darüber, wie lange die Zeit zwischen dem Erfassen des Pendelzustandes und dem Anliegen der Stellgröße sein darf. Um diese Anforderung wiederzuspiegeln, muss sie so übersetzt werden, dass aus Sicht der Middleware eine konkrete Aussage über die Übertragungszeit zwischen zwei Knoten gemacht wird.

Zunächst wird für jede der Abstraktions-Ebenen *Anwendung*, *Middleware* und *MAC* je eine Dienstgütedomäne (Quality-of-Service-, kurz QoS-Domäne) definiert. Da sich Dienstgüte aus mehreren Aspekten wie Leistung (Performance), Zuverlässigkeit (Reliability) und Güte der Zusicherungen (Guarantee) zusammensetzt, wird die Dienstgüte-Domäne aus Subdomänen gebildet.

In diesen Subdomänen sind die Wertebereiche festgelegt, aus denen sich konkrete Ausprägungen der Dienstgüte zusammensetzen. Diese konkreten Ausprägungen sind Elemente der Dienstgütedomäne – kurz QoS-Werte.

Sind die Domänen definiert, so wird die Dienstgüteanforderung für das System mittels zweier QoS-Werte und einem Skalierungswert festgelegt: Ein QoS-Wert definiert die minimale Dienstgüte-Anforderung, der andere die optimale Dienstgüte. Beide Werte entstammen der gleichen Domäne und somit der gleichen Sicht. Die Festlegung, welchen QoS-Werten einer anderen Sicht sie entsprechen, geschieht über *Mapping*-Funktionen. Der Skalierungswert beinhaltet eine Nutzenfunktion, über die mehrdimensionale QoS-Werte präferenzgeordnet werden.

Die Dienstgütedomäne für die Anwendungssicht *Pendelsteuerung* setzt sich aus der *Performance*-Subdomäne $P_{Steuerung}$, der *Reliability*-Subdomäne $R_{Steuerung}$ und der *Guarantee*-Subdomäne $G_{Steuerung}$ zusammen. Diese Subdomänen wiederum werden aus Unteraspekten gebildet, die Einfluss auf einen der Aspekte Leistung (Performance), Zuverlässigkeit (Reliability) oder Güte der Zusicherungen (Guarantee) ausüben. Formal gilt daher:

$$\text{QoS-Domäne} \quad Q_{Steuerung} = P_{Steuerung} \times R_{Steuerung} \times G_{Steuerung}$$

Performance-Subdomäne Die Performance-Subdomäne $P_{Steuerung}$ der Dienstgüte-Domäne der Anwendungssicht setzt sich aus den zwei Mengen *Signalintervall* und *Totzeit* zusammen. *Signalintervall* enthält alle zulässigen Werte für die Zeit, die zwischen zwei Steuersignalen vergehen kann. Wir schränken die Genauigkeit auf Millisekunden ein – Zeiten sind daher nur ganzzahlige Vielfache einer Millisekunde. *Totzeit* enthält alle Werte, die als Reaktionszeit des Systems auftreten können, von der Messung der Pendel-Lage bis zur Gegensteuerung. Auch hier werden ganze Millisekunden als Zeitdauer angenommen.

Formal: $P_{Steuerung} = Signalintervall \times Totzeit$
$Signalintervall = \mathbb{N}$ (Zeit zwischen zwei Steuersignalen in ms)
$Totzeit = \mathbb{N}$ (Reaktionszeit des Systems in ms)

Reliability-Subdomäne Die Reliability-Subdomäne $R_{Steuerung}$ der Dienstgüte-Domäne der Anwendungssicht setzt sich aus den vier Mengen *Loss*, *Period*, *Burstiness* und *Corruption* zusammen. *Loss* enthält alle zulässigen Werte für die Anzahl verlorener Steuersignale, *Period* die Zeit in ganzen Millisekunden, auf die sich die Werte aus *Loss* beziehen. *Burstiness* enthält die zulässigen Werte für die Anzahl an aufeinanderfolgenden fehlerhaften Steuersignalen, falls ein Fehler auftritt, *Corruption* die Prozentpunkte zur Angabe der Verfälschungsrate der übermittelten Steuersi-

gnale.

Formal: $R_{Steuerung} = Loss \times Period \times Burstiness \times Corruption$
$Loss = \mathbb{N}_0$
$Period = \mathbb{N}_0$
$Burstiness = \mathbb{N}_0$
$Corruption = \{c \in \mathbb{N}_0 | 0 \leq c < 100\}$

Guarantee-Subdomäne Die Guarantee-Subdomäne $G_{Steuerung}$ der Dienstgüte-Domäne der Anwendungssicht setzt sich aus den drei Mengen *DoC*, *Stat* und *Priority* zusammen. *DoC* (Degree of Commitment) enthält vier mögliche Werte für die Güte der Zusicherungen: *Best-Effort* als geringste Güte, da keine Zusicherung über die Dienstgüte gemacht wird; *Enhanced-Best-Effort* sichert die Einhaltung der vereinbarten Dienstgüte zu, wenn es die Resourcen zulassen, sonst nach Priorität; *Statistical* als Zusicherung, die Dienstgüte werde mit einer statistischen Wahrscheinlichkeit eingehalten; *Deterministic* als höchste Güte der Zusicherungen garantiert die Einhaltung der vereinbarten Dienstgüte. *Stat* enthält statistische Wahrscheinlichkeiten für die Zusicherungsgüte *Statistical* und *Priority* mögliche Prioritäten für den Fall der Zusicherungsgüte Enhanced-Best-Effort.

Formal: $G_{Steuerung} = DoC \times Stat \times Priority$
$DoC = \{bestEffort, enhancedBestEffort, statistical, deterministic\}$
$Stat = \{p \in \mathbb{R} | 0 < p \leq 1\}$
$Priority = \mathbb{N}$

Die aus Werten dieser Domänen gebildeten Anforderungen für das Steuerungssystem sind Abschätzungen, die aus der Systembeschreibung in [GKLC] abgeleitet sind. Als optimale Dienstgüte-Ausprägung für die erbrachte Leistung des Netzwerks wird ein realisiertes *Signalintervall* von 20 ms und eine *Totzeit* von 8 ms angesehen. Mit einer Zuverlässigkeit der Dienstleistung, die sich aus dem Verlust von 0 Nachrichten (*Loss*) (bezogen auf einen Zeitraum von 200 ms (*Period*)), einer Fehlerhäufung von 0 Nachrichten (*Burstiness*) und einer Verfälschungsrate von 0% (*Corruption*) ergibt. Die Dienstgüte wird mit *enhancedBestEffort* zugesichert.

Als minimale Dienstgüte ist ein realisiertes *Signalintervall* von 30 ms und eine *Totzeit* von 24 ms zulässig. Die Zuverlässigkeit der Dienstleistung darf den Verlust eines Steuersignals pro 300 ms nicht überschreiten (das entspricht bei dem gegebenen Signalintervall jedem 10. Signal). Solche Verluste dürfen nicht gehäuft auftreten ($Burstiness \leq 1$). Die Verfälschungsrate darf nur 0% betragen. Die Dienstgüte wird mindestens mit *enhancedBestEffort* zugesichert.

Formal: QoS-Anforderungen *Pendelsteuerung*

- Optimal Value
  - Performance: $Signalintervall = 20\,[ms]$,
    $Totzeit = 8\,[ms]$
  - Reliability: $Loss = 0\,[Steuersignale]$,
    $Period = 200\,[ms]$,
    $Burstiness = 0\,[Steuersignale]$,
    $Corruption = 0\,[\%]$

  - Guarantee: $DoC = enhancedBestEffort$

- Minimal Value
  - Performance: $Signalintervall = 30\,[ms]$,
    $Totzeit = 24\,[ms]$
  - Reliability: $Loss = 1\,[Steuersignal]$,
    $Period = 300\,[ms]$,
    $Burstiness \leq 1\,[Steuersignal]$,
    $Corruption = 0\,[\%]$
  - Guarantee: $DoC = enhancedBestEffort$

Eine Nutzenfunktion u, die Teil der Anforderungs-Spezifikation ist, weist jedem QoS-Wert seinen Nutzen für den Anwender zu. Alle QoS-Werte der Domäne, deren Nutzen zwischen dem des minimalen und dem des optimalen Wertes liegt, bilden (zusammen mit dem Optimum und dem Minimum) die Menge zulässiger QoS-Werte für die Anwendung.

Die Nutzenfunktion $u : Q_{Steuerung} \rightarrow [0..1]$ weist jedem QoS-Wert der QoS-Domäne einen Nutzen zu. Der Nutzen ist normalisiert und entspricht daher einem Wert zwischen 0 und 1. QoS-Werte sind mehrdimensional, ein Wert q der Domäne $Q_{Steuerung}$ setzt sich aus Elementen der Sub-Domänen $P_{Steuerung}$, $R_{Steuerung}$, $G_{Steuerung}$ zusammen: $(p, r, g) = q \in Q_{Steuerung}$
Daher ist die Nutzenfunktion u auf spezialisierten Nutzenfunktionen u_P, u_R, u_G der Sub-Domänen definiert. Der kleinste Nutzen eines Bestandteils des QoS-Wertes bestimmt auch den Gesamtnutzen des QoS-Wertes.

$$u : Q_{Steuerung} \rightarrow [0..1]$$
$$u(q) = min\{u_P(p), u_R(r), u_G(g)\}$$

Die spezialisierte Nutzenfunktion u_P ist auf den Funktionen $u_{Signalintervall}$ und $u_{Totzeit}$ definiert, die den Bestandteilen $s \in Signalintervall$ und $t \in Totzeit$ von $p \in P_{Steuerung}$ jeweils einen Nutzen zuordnen:

$$(s, t) = p \in P_{Steuerung}$$
$$u_P(p) = u_{Signalintervall}(s) \cdot u_{Totzeit}(t)$$

Die Multiplikation bewirkt, wenn einer der Werte für den Nutzen Null ist – und damit unbrauchbar ist – dass der gesamte Nutzen Null wird. Bei einem positiven Wert kleiner als Eins wird der resultierende Nutzen anteilig geschmälert. Die Funktionen $u_{Signalintervall}$ und $u_{Totzeit}$ bilden s bzw. t auf einen Wert zwischen 0 und 1 ab.
Da die optimalen Werte für Signalintervall und Totzeit auf einen Nutzen von 1 abgebildet werden und die minimalen Werte jeweils auf einen Nutzen von immer noch $\frac{1}{3}$ (willkürliche Festlegung), sind die beiden Funktionen $u_{Signalintervall}$ und $u_{Totzeit}$ folgendermaßen definiert:

$$u_{Signalintervall}(s) = \frac{35 - s}{15}$$
$$u_{Totzeit}(t) = \frac{32 - t}{24}$$

Die Funktionen u_R und u_G wurden so gewählt, dass sie jeweils einen Nutzen von 1 liefern, wenn die minimalen *Reliability*- bzw. *Guarantee*-Anforderungen erfüllt sind, und 0 andernfalls:

$$u_R(r) = \begin{cases} 0 & \text{falls} \quad \frac{Loss}{Period} > \frac{Loss_{min}}{Period_{min}} \vee Burstiness > 1 \vee Corruption > 0 \\ 1 & \text{sonst} \end{cases}$$

$$u_G(g) = \begin{cases} 0 & \text{falls} \quad g < enhancedBestEffort \\ 1 & \text{sonst} \end{cases}$$

Zur Vollständigkeit der formalen Anforderungs-Aufstellung sind nach [WG07] noch weitere Angaben nötig. So z.B. die Dienstgüte-Domänen der anderen Sichten, sowie Mapping-Funktionen zwischen den Domänen benachbarter Sichten. Diese werden hier nicht im Detail behandelt und nur die Anwendungssicht ist exemplarisch dargestellt. Die ermittelten QoS-Domänen und *Mapping*-Funktionen für alle Sichten sind in Anhang A enthalten.

3.2 Application-Adapter

Übersicht Der *Application-Adapter* ist ein Modul, das eine in C geschriebene Schnittstelle für den Anwendungsprogrammierer des Steuerungssystems zur Verfügung stellt. Diese Schnittstelle kann er mit den Mitteln der Sprache in seine Applikation einbinden. Die Schnittstelle verwendet nur native Datentypen aus C. Damit bietet der Adapter eine Entkopplung von den in tieferen Schichten genutzten SDL-Datentypen. Zusätzlich zu dieser Schnittstelle besitzt das Modul einen in SDL spezifizierten Teil, der die Schnittstelle zu den sich nach unten anschließenden Schichten wie der Middleware darstellt. Die SDL-Spezifikation besteht aus einem Prozess, der die Applikation des Anwenders im SDL-System repräsentiert. Der Prozess wiederum enthält Transitionen und Prozeduren, um SDL-Signale zu empfangen und zu versenden. Somit kann ein SDL-Signal, das aus tieferen Schichten kommt und an die Anwendung gerichtet ist, vom SDL-Prozess, der diese Anwendung repräsentiert, entgegengenommen werden. In den Transitionen sind Abschnitte in C-Syntax eingebettet, die dann bei der Verarbeitung der SDL-Signale ausgeführt werden sollen. Innerhalb dieser Abschnitte können Funktionen der in C geschriebenen Schnittstelle des Adapters aufgerufen und Parameter übergeben werden. So erreichen die SDL-Signale in Form von Funktionsaufrufen die Benutzer-Applikation. In der entgegengesetzten Richtung führen Funktionsaufrufe der C-Schnittstelle durch die Benutzer-Applikation zu einem Aufruf einer entsprechenden SDL-Prozedur innerhalb des SDL-Prozesses, der die Nutzer-Applikation repräsentiert. Auch hier können Parameter übergeben werden. Aus diesen SDL-Prozeduren heraus können dann wiederum SDL-Signale an die Middleware versandt werden.

Ein in SDL spezifiziertes System wird mithilfe des Kode-Generators *ConTraST* [FGW06] in C-Kode umgewandelt, der anschließend kompiliert wird. Das resultierende Programme läuft in der Laufzeitumgebung *SDL-Environment-Framework*

[FGJ+05] (kurz *SEnF*). Abschnitte, die Programmcode in C-Syntax enthalten, können in der SDL-Spezifikation daher direkt innerhalb eines Textfeldes in der Prozess-Definition und innerhalb von Transitionen in Task-Feldern stehen. Sind diese Abschnitte korrekt gekennzeichnet, erkennt der Kode-Generator den Kode-Baustein und fügt ihn im generierten Kode an dieser Stelle ein.

Der Adapter setzt sich aus einem eigenen SDL-Package, dem Package *UserDefinedApplication*, und zwei externen, C-Kode enthaltenden Dateien zusammen. Die beiden externen Dateien – `application.cpp` und `application.h` – bilden ein Template für Nutzer-Applikationen. Die C Schnittstelle des Application-Adapters besteht aus diesem *Template*, das konkrete Applikationen als Grundlage nehmen können, und den in die SDL-Spezifikation eingebetteten Kode-Fragmenten im SDL-*Package* UserDefinedApplication.

Die *Header*-Datei `application.h` enthält die Deklarationen der Funktionen, mit denen die Benutzer-Applikation den SDL-Prozess ansprechen kann, der sie im SDL-System repräsentiert, und die Deklarationen von Funktionen, mit denen Teile der Hardware gesteuert werden können, wie z.B. LEDs. (Deren Implementierungen sind nach dem Kode-Generieren im Package UserDefinedApplication enthalten.) Ein Aufruf einer dieser Funktionen stellt Applikation-zu-Hardware- oder Applikation-zu-Middleware-Kommunikation dar.

Die Datei `application.cpp` enthält die Anwendungslogik und bindet die Header-Datei `application.h` ein, um die Funktionen nutzen zu können. Sie ist ein Template und soll vom Anwendungsprogrammierer erweitert werden. Der Programmierer der Steuerungsanwendung ergänzt dieses Template um die Anwendungslogik und nutzt die vorgegebenen Funktionen zur Interaktion mit den Programmteilen auf anderen Knoten und der eigenen Knoten-Hardware. Das Template ist so angelegt, dass er die Anwendung in *Tasks* aufteilen kann – Aufgaben, die einmalig oder mehrmals nach Ablauf einer von ihm angegebenen Zeitspanne ausgeführt werden sollen. (1.) Jeder Task ist eine eindeutige, ganzzahlige Nummer zugeordnet. (2.) Die Befehlsabfolgen, die die tatsächliche Durchführung der Aufgabe implementieren, werden in eine Funktion zusammengefasst. (3.) Die schon vorhandene Funktion Execute(no:int):void ist um einen Fall zu erweitern, der die Implementierung mit der Task-Nummer verknüpft. (4.) Dem Entwickler steht eine Initialisierungsfunktion Init():void zur Verfügung, die einmalig beim Systemstart ausgeführt wird. Diese Funktion ist dafür vorgesehen, die Zeitspanne für Tasks festzulegen, nach deren Ablauf eine Aufgabe das erste mal ausgeführt wird. Die Ausführung einer Task nach Ablauf ihrer Zeitspanne wird dann vom System automatisch angestoßen. Das in Listing 3.1 aufgeführte Kode-Fragment zeigt die nötigen Elemente einer Task.

In diesem Beispiel wird die Task eine Sekunde nach Start des Systems das erste mal ausgeführt. Wird die festgelegte Periode nicht geändert, so wird die Task auch weiterhin einmal pro Sekunde ausgeführt. Erst wenn die Periode auf Null gesetzt wird, endet die periodische Ausführung. Für einmal auszuführende Aufgaben sollte deshalb das abgewandelte, in Listing 3.2 gezeigte Schema genutzt werden.

Der Entwickler muss diese Möglichkeit nicht nutzen, sollte sie aber für alle periodisch wiederkehrenden Aufgaben und für solche Aufgaben, die zwar einmalig aber dennoch mit einem Sicherheitsabstand nach Systemstart ausgeführt werden sollen,

```
1  // (1.) eindeutige Nummer
2  #define Task_sayHello 1
3
4  // (2.) Implementierung der Task-Ausführung
5  void sayHello() {
6          printf("Hello world!\n");
7  }
8
9  // (3.) Verknüpfung von Task-Nummer und Implementierung
10 void Execute(int no) {
11         switch(no) {
12                 ...
13                 case Task_sayHello:      sayHello();
14                                          break;
15                 ...
16         }
17 }
18
19 // (4.) Festlegung einer Zeitspanne
20 void Init() {
21         ...
22         SetPeriod(Task_sayHello, 1000);
23         ...
24 }
```

Listing 3.1: Definition und Ausruf einer Task.

```
1                  ...
2                  case Task_sayHello:      SetPeriod(Task_sayHello, 0);
3                                           sayHello();
4                                           break;
5                  ...
```

Listing 3.2: Einmaliger Aufruf einer Task.

einsetzen. Soll z.B. Kontakt zu anderen Knoten hergestellt werden, ist ein zeitlicher Sicherheitsabstand ratsam, um den anderen Knoten Zeit zur jeweiligen eigenen Initialisierung zu geben.

Die folgenden Funktionen der Benutzer-Anwendung (application.cpp) sind für den SDL-Prozess, der die Anwendung im SDL-System repräsentiert, sichtbar. Ein Aufruf einer dieser Funktionen stellt Middleware-zu-Applikation- oder Hardware-zu-Applikation-Kommunikation dar. Weitere Funktionen und eine ausführliche Beschreibung jeder Funktion ist in Anhang B.1 zu finden.

- `void Init()`: Initialisierung der Anwendung.
- `void Execute(number)`: Ausführen von Tasks anhand ihrer Nummer.
- `void Receive(serviceID, data, length)`: Übergabe von Daten an die Applikation.
- `void requestDataForService(serviceID)`: Anforderung von Daten der Applikation.

Um der Anwendung die Verwaltung registrierter oder abonnierter Dienste zu erleichtern, wurde die Datenstruktur *service_t* und ein entsprechendes Array angelegt (s. Listing 3.3).

```
1  struct service_t {
2          int id;
3          char* name;
4          bool subscribed;
5  };
6
7  service_t services[]={
8          {-1, "service_offered_by_me", false},
9          {-1, "service_used_by_me", true},
10         ...
11 };
```

Listing 3.3: Datenstruktur zur Verwaltung der Dienste.

Die Verwendung durch den Anwendungsprogrammierer ist nicht zwingend erforderlich, und für den Fall, dass nur ein Dienst genutzt wird, auch nicht nötig. Der Programmierer kann bei mehreren genutzten Diensten allerdings alle, aus denen er das System zusammenstellen will, im Array *services* anlegen. Die ID -1 kann als ungültig betrachtet werden und als Indikator genutzt werden, der anzeigt, ob ein Dienst schon registriert oder abonniert wurde, da er in diesen Fällen bei Registrierung und Abonnement eine gültige ID erhält. Der Name des Dienstes wird bei der Registrierung oder dem Abonnement benötigt, da die ID nur lokal gültig ist, ein Dienst global aber durch seinen Namen identifiziert wird. Das *Flag* `subscribed` kann genutzt werden, um zwischen Diensten, die der Knoten anbietet und Diensten, die der Knoten als Abonnent nutzt, zu unterscheiden. Zwei getrennte Arrays einzuführen ist eine andere Möglichkeit der Unterscheidung.

Eine Task könnte mit einem zeitlichen Sicherheitsabstand nach Systemstart durchgeführt werden, die in einer Schleife das gesamte Array durchläuft und die Registrierung bzw. das Abonnement der eingetragenen Dienste – gesteuert vom *subscribed*-Flag – anstößt.

Wie schon erwähnt, wird die Schnittstelle zum SDL-Prozess von den Funktionen gebildet, die in der eingebundenen Header-Datei `application.h` deklariert sind. Darunter sind auch die Funktionen zur Registrierung und zum Abonnement von Diensten. Wie sie verwendet werden und aus welchen Funktionen die Schnittstelle besteht, ist in Anhang B.2 beschrieben.

Ein Aufruf einer dieser Funktionen stellt Applikation-zu-Hardware- oder Applikation-zu-Middleware-Kommunikation dar. Der Anwendungsentwickler nutzt die vorgegebenen Funktionen zur Interaktion mit den Programmteilen auf anderen Knoten und der eigenen Knoten-Hardware.

Die Schnittstelle enthält u. a. foldende Funktionen:

- `void Send(serviceID, data, length)`: Versand von Daten.

- `int registerService(name, throughput, reactionDelay)`: Registrierung eines Dienstes durch die Anwendung.
- `void deregisterService(id)`: Aufheben einer Registrierung.
- `int subscribeService(name, period, delay)`: Abonnement eines Dienstes durch die Anwendung.
- `void unsubscribeService(id)`: Kündigung eines Abonnements.
- `void setRequestedData(serviceID, data, length)`: Bereitstellung von angeforderten Daten durch die Anwendung.

Das Programm-Fragment in Listing 3.4 zeigt anhand eines Beispiels die Registrierung und das Abonnement je eines Dienstes. Es nutzt die weiter vorne vorgestellte Datenstruktur `service_t` und das Array `services`.

```
1  struct service_t {
2          int id;
3          char* name;
4          bool subscribed;
5  };
6
7  service_t services[]={
8          {-1, "service_offered_by_me", false},
9          {-1, "service_used_by_me", true}
10 };
11
12 ...
13
14 // Registrierung des Dienstes "service_offered_by_me"
15 services[0].id = registerService(services[0].name, 256, 10);
16
17 // Abonnement des Dienstes "service_used_by_me"
18 services[1].id = subscribeService(services[1].name, 30, 20);
```

Listing 3.4: Registrierung und Abonnement eines Dienstes über die Schnittstelle.

Eine Beispiel-Anwendung, die auch für den Simulationslauf in Kapitel 4 eingesetzt wurde, ist auf der CD enthalten.

Die Generierung und Vergabe der IDs ist sehr einfach gehalten. Ein mit 0 initialisierter Zähler wird um 1 hochgezählt, immer wenn ein Dienst auf dem Knoten registriert oder abonniert wird. Anschließend wird der neue Wert des Zählers als ID vergeben. Die erste vergebene ID ist demnach 1 und nicht 0. Der Zähler enthält daher die Summe aus der Anzahl registrierter und der Anzahl abonnierter Dienste. Eine erneute Vergabe einer ID, nach deren Freiwerden durch Kündigung eines Abonnements oder Löschung einer Registrierung, erfolgt nicht. Theoretisch ist die Vergabe von negativen IDs bei Überschreiten des darstellbaren Bereichs des Datentyps `Integer` möglich, dies hätte aber keinerlei Auswirkungen und ist außerdem unwahrscheinlich. Erst wenn auch der negative Bereich ausgeschöpft ist und positive

IDs, die noch verwendet werden, ein zweites mal vergeben würden, wären die vorher durch diese IDs identifizierten Dienste nicht mehr durch diesen Knoten ansprechbar.

Der in SDL spezifizierte Teil des Application-Adapters ist in einem eigenen Package gekapselt. Das Package `UserDefinedApplication` enthält den Block-Typ `ApplicationAdapter`, der als einzigen Prozess `userApplication` enthält. Dieser repräsentiert die Nutzer-Applikation. Er führt daher alle an sie adressierte und von ihr ausgehende Kommunikation. Er verfügt über zwei Signalwege. Den Signalweg `io_signaling` für Signale, die an die Knoten-Hardware (z.B. LEDs), die UART-Schnittstelle und SPI-Schnittstelle gerichtet sind. Und den Signalweg `com_signaling` für Signale an die *Communication-Middleware*. Die Signalwege werden jeweils in beiden Richtungen eingesetzt. Im entwickelten Steuerungssystem ist eine Instanz des Block-Typs vorhanden.

Der SDL-Prozess des Adapters vergibt und verwaltet die ID, die von der Applikation nach dem Abonnement oder der Registrierung eines Dienstes zur weiteren Interaktion mit dem Dienst eingesetzt wird. Eingehende SDL-Signale werden in entsprechende Funktionsaufrufe der Applikation umgesetzt, beispielsweise beim Empfang von Daten oder wenn periodische Daten von der Applikation angefordert werden, wie in Abb. 3.2 zu sehen ist. Die gezeigte Transition

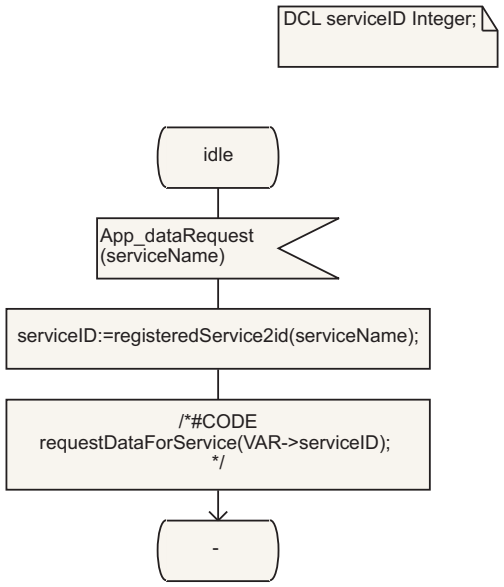

Abbildung 3.2: Beispiel der Abbildung von SDL-Signalen auf Funktionsaufrufe anhand einer periodischen Datenanforderung.

nimmt die Anforderung App_DataRequest entgegen, sucht die entsprechende ID aus einer Liste, die alle hier registrierten Dienstnamen enthält, und ruft die C-Funktion `requestDataForService` der Anwendung auf. Umgekehrt führen Aufrufe der Schnittstellenfunktionen des Prozesses zu entsprechenden SDL-Signalen an die Middleware.

3.3 User-Provider-Middleware

Die Middleware muss eine Reihe von Aufgaben erfüllen. Sie muss Details des Netzwerks verbergen, um die Nutzung zu vereinfachen; sie muss das Suchen und Auffinden von Diensten unterstützen, eine Schnittstelle zur Registrierung und zum Abonnement von Diensten anbieten, und Dienste und Abonnenten verwalten.

Eine Komponente für diese Aufgaben bietet bereits die AmICoM. Die *UserProvider-Middleware* – sie wurde im Rahmen dieser Arbeit erweitert, um Dienste verwalten zu können, die den periodischen Versand von Daten anbieten. Es ist Aufgabe der Middleware, für den periodischen Versand rechtzeitig Daten von den Diensten anzufordern.

Das gesamte System der *AmICoM* aus einer Komponente zur Dienstverwaltung, einer Routing-Komponente und einer MAC-Komponente wurde für das inverse Pendel angepasst, erweitert und in WNCS-CoM umbenannt. Abbildung 3.3 zeigt die jeweils korrespondierenden Komponenten der beiden Systeme.

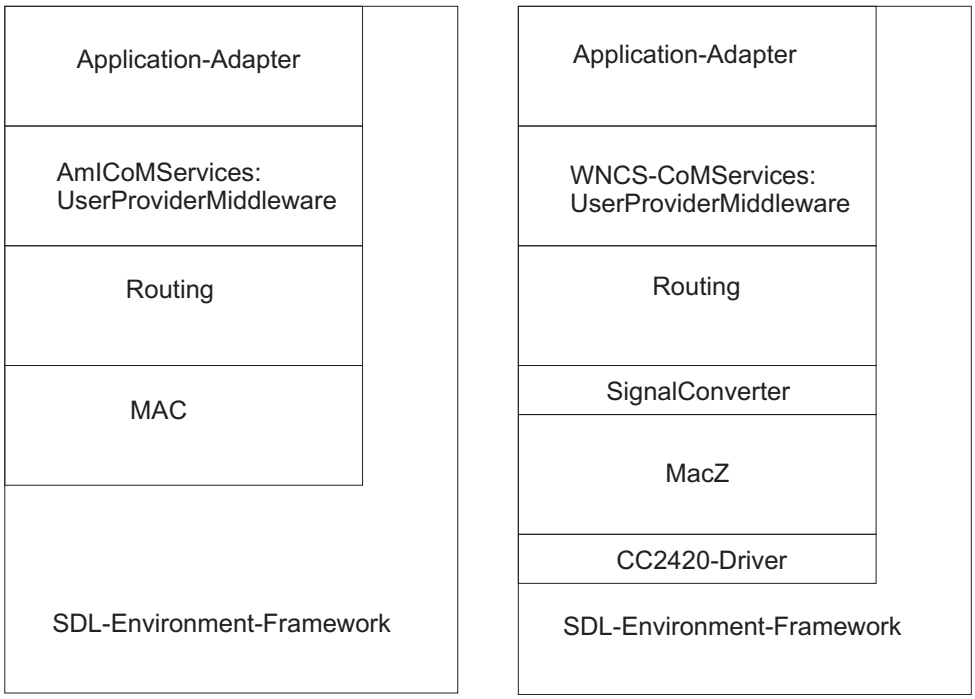

Abbildung 3.3: Architekturen der AmICoM (links) und WNCS-CoM (rechts).

Die Komponente *UserProviderMiddleware* zur Verwaltung der Dienste wurde um die Fähigkeit der rechtzeitigen, periodischen Datenanforderung erweitert. Dafür waren auch die zur Registrierung und zum Abonnement eingesetzten Signale um Parameter zu erweitern. Die Routing-Komponente wurde hingegen beibehalten. Die MAC-Schicht wurde jedoch komplett ersetzt, um auf ein drahtloses Medium zugreifen zu können. Die Routing-Komponente wird in dieser Arbeit nicht behandelt, die Erweiterungen der MAC-Schicht werden in Kapitel 4 erläutert.

3.3.1 Aufbau

Betrachtet man den Aufbau der *UserProviderMiddleware* (Abb. 3.4) fällt zunächst auf, dass ein zusätzlicher Prozess eingeführt wurde – genauer, eine Gruppe von Prozessen des Typs *Scheduler*.

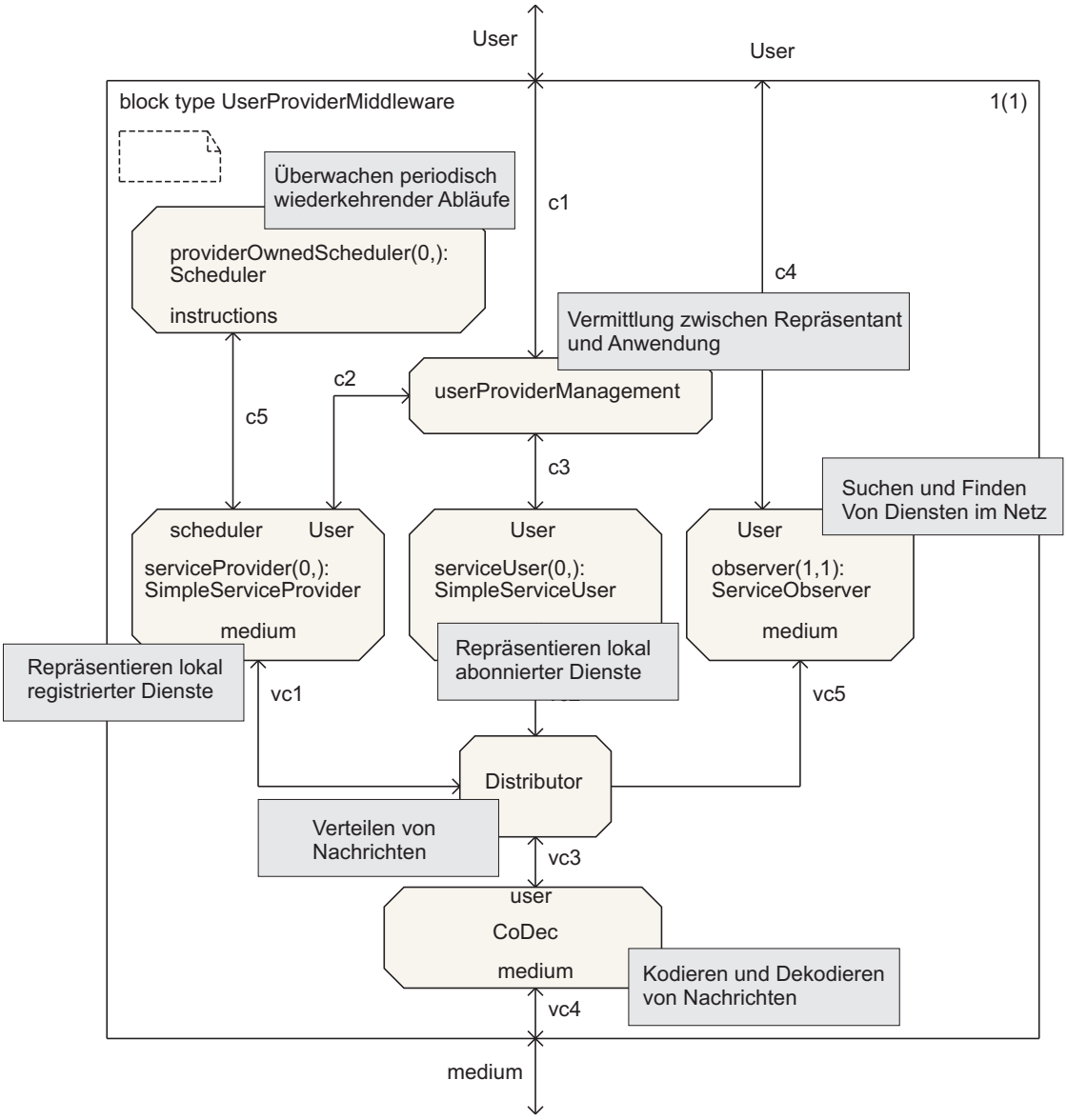

Abbildung 3.4: Architektur der UserProviderMiddleware nach erfolgter Erweiterung.

Wie in der ursprünglichen *User-Provider-Middleware* der AmICoM wird auf einem Knoten für jeden abonnierten Dienst eine eigene Prozess-Instanz des ServiceUser-Prozess-Typs angelegt. Ebenso wird eine Instanz des ServiceProvider-Prozess-Typs für jeden dort registrierten Dienst angelegt. In der WNCS-CoM besitzt jeder ServiceProvider-Prozess zusätzlich eine eigene Scheduler-Instanz. Damit ist prinzipiell jeder Dienst in der Lage, einen automatischen periodischen Datenversand

für seine Abonnenten anzubieten. Denn das Signal zum Abonnieren eines Dienstes trägt als einen Parameter die Periode, in deren Abständen automatisch Daten an den Abonnenten zu versenden sind. Ist die übermittelte Periode Null oder bietet der Dienst keinen automatischen periodischen Versand an, so ignoriert er den Parameter. Wird aber eine von Null verschiedene Periode übermittelt und bietet der Dienst den automatischen Versand, so leitet er die Anmeldung an seinen eigenen *Scheduler*-Prozess. Der Scheduler-Prozess überwacht alle angemeldeten Abonnenten eines Dienstes und deren Perioden. Er veranlasst die Datenanforderung für einen Abonnenten, wenn dessen Periode verstrichen ist. Der Scheduler plant und veranlasst also alle zeitlich gesteuerten Datenanforderungen an den Dienst, hat aber keinen Anteil an der Abwicklung sonstiger Anforderungen an den Dienst. Der Scheduler führt dabei die im folgenden vorgestellte Synchronisation zwischen den Zeitpunkten durch, auf die er den Beginn einer Periode eines Abonnenten legt. Da das System für das inverse Pendel entwickelt wurde, wird im folgenden davon ausgegangen, das Daten-Anforderungen immer einen Messwert eines Sensors anfordern.

3.3.2 Synchronisation der Abonnenten

Sobald ein Dienst mehr als einem Abonnenten periodisch Daten sendet, die zuvor durch Messung oder Berechnung ermittelt werden müssen, können regelmäßig Daten-Anforderungen eingespart werden. Wenn der jeweilige Beginn der periodischen Datenermittlung für zwei verschiedene Abonnenten synchronisiert wurde, sodass die erstmalige Daten-Anforderung für einen der Abonnenten mit einer Anforderung für den Anderen zusammenfällt, werden regelmäßig auch zukünftige Anforderungen zusammenfallen, und beide Abonnenten können mit dem gleichen Messwert bedient werden.

Zur Erläuterung wird zunächst das unsynchronisierte Verhalten gezeigt.

Unsynchronisiertes Verhalten Ohne Synchronisation wartet der Scheduler des Dienstes die Zeitdauer einer Periode p_i nach Erhalt der Anmeldung bis zur Anforderung des ersten Messwertes für einen Abonnenten A_i. Kommt ein zweiter Abonnent A_j hinzu, erhält dieser auch nach einer Periode p_j ab dem Zeitpunkt seiner Anmeldung den ersten Messwert.

In einem ungünstigen Fall mit zwei Abonnenten A_1, A_2 kann der Dienst trotz deren gleicher Periode $p = p_1 = p_2$ als größte Ruheperiode zwischen zwei Messwert-Anforderungen nur $\frac{1}{2}p$ nutzen, um Energie zu sparen.

Abbildung 3.5 zeigt, wie zum Zeitpunkt t_1 ein Abonnent A_2 zu einem bereits angemeldeten Abonnenten A_1 hinzukommt, $\frac{1}{2}p$ vor Zeitpunkt t_2, der Messwert-Anforderung für A_1.

Ein zweiter ungünstiger Fall liegt vor, wenn t_1 und t_2 dicht aufeinanderfolgen, da dann alle zukünftigen Messwert-Anforderungen für beide Abonnenten ebenso dicht aufeinander folgen und die Einhaltung einer möglicherweise nötigen Messwert-Ermittlungsperiode p_{smpl} verhindern. Dieser Fall ist in Abb. 3.6 dargestellt.

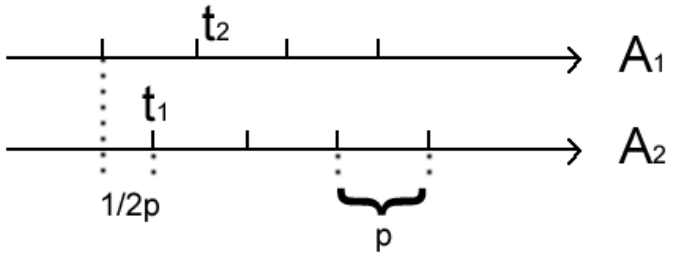

Abbildung 3.5: Zwei unsynchronisierte Abonnenten mit gleicher Periode.

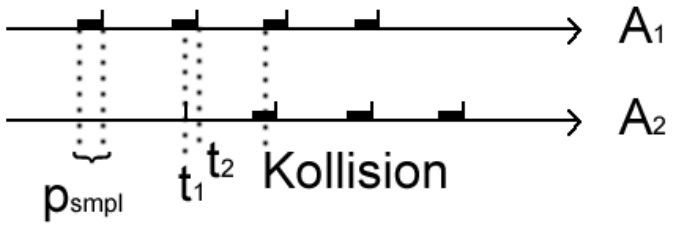

Abbildung 3.6: Zwei unsynchronisierte Abonnenten mit kollidierenden Anforderungs-Zeitpunkten.

Im ersten Fall ist eine Verkürzung der Wartezeit bis zur erstmaligen Messwert-Anforderung für A_2 möglich, sodass diese mit der Anforderung für A_1 zusammenfällt. Im zweiten Fall ist es möglich, den für A_1 angeforderten Messwert nach Ablauf der Messwert-Ermittlungsperiode an beide Abonnenten zu versenden. Das erspart in beiden Fällen die Hälfte aller Anfragen, da eine gemeinsame Anfrage für beide Abonnenten möglich ist. Außerdem verlängert dies die Ruheperiode des Dienstes zwischen zwei Messungen auf die gemeinsame Periode p. Doch dazu ist eine gezielte Synchronisation nötig.

Synchronisiertes Verhalten bei zwei Abonnenten Mit Synchronisation wartet der Scheduler eines Dienstes eine Zeitdauer $\leq p_i$ nach Erhalt der Anmeldung bis zur erstmaligen Anforderung eines Messwertes für einen Abonnenten A_i. Kommt ein zweiter Abonnent A_j hinzu, so erhält er zum Zeitpunkt

$$t_1 + [(t_2 - t_1) \text{mod } p_j]$$

erstmalig einen Messwert. Diese Formel deckt sowohl den Fall $p_i < p_j$ (s. Abb. 3.7) als auch den Fall $p_i > p_j$ (s. Abb. 3.8) ab. Vereinfachend darf hier angenommen werden, dass sich alle Zeitpunkte und Perioden als ganzzahlige Vielfache einer Zeiteinheit darstellen lassen.

Nach einem erstmalig gemeinsam genutzten Messwert zum Zeitpunkt t_2 vergeht genau die Zeit

$$\frac{p_i \cdot p_j}{ggt(p_i, p_j)}$$

bis zum nächsten gemeinsam genutzten Messwert.

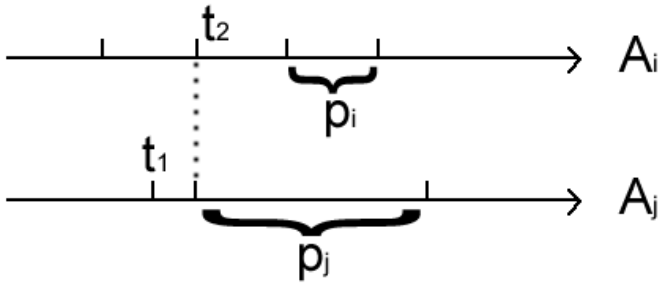

Abbildung 3.7: Zwei synchronisierte Abonnenten, $p_i < p_j$.

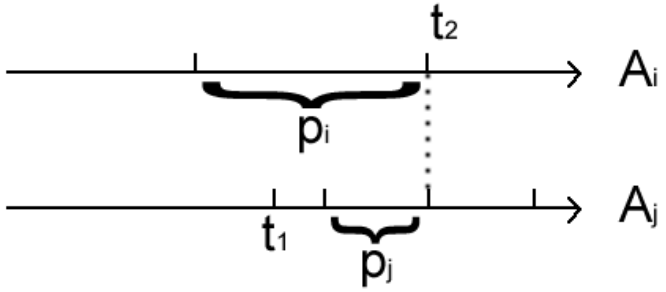

Abbildung 3.8: Zwei synchronisierte Abonnenten, $p_i > p_j$.

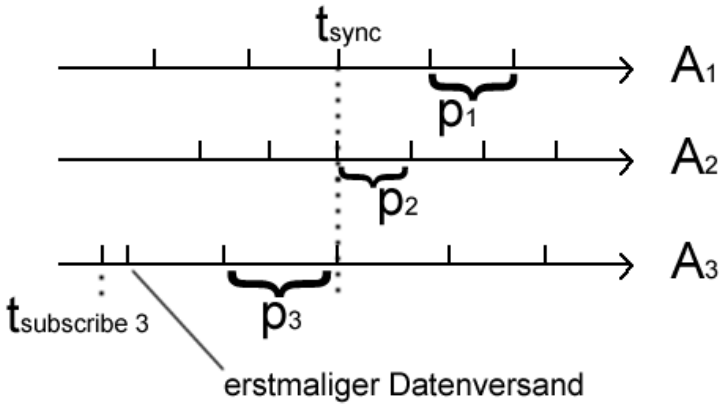

Abbildung 3.9: Drei zum Zeitpunkt t_{sync} synchronisierte Abonnenten.

Synchronisiertes Verhalten bei n> 2 Abonnenten A_1, A_2 aus Abb.3.9 sind synchronisiert, das Ereignis zum Zeitpunkt t_{sync} an dem das nächste mal eine Anforderung eingespart wird, weil beide Anforderungen zusammenfallen, wiederholt sich in den Abständen, die durch die aggregierte Periode $p_{1,2}$, mit

$$p_{1,2} = \frac{p_1 \cdot p_2}{ggt(p_1, p_2)}$$

vorgegeben sind. Kommt ein Abonnent A_3 zum Zeitpunkt $t_{subscribe_3}$ hinzu, wird er mit dem Zeitpunkt t_{sync} synchronisiert. Bezeichnet man immer den nächsten solchen Zeitpunkt mit t_{sync} und sei der Zeitpunkt der Anmeldung eines Abonnenten A_i mit $t_{subscribe_i}$ bezeichnet, dann erhält A_i erstmalig zum Zeitpunkt

$$t_{subscribe_i} + [(t_{sync} - t_{subscribe_i}) \text{mod } p_i]$$

einen Messwert. Durch dieses Vorgehen sind zu jedem Zeitpunkt alle Abonnenten synchronisiert. Die aggregierte Periode $p_{1,2,3}$, mit

$$p_{1,2,3} = \frac{p_{1,2} \cdot p_3}{ggt(p_{1,2}, p_3)}$$

ist der Abstand der Zeitpunkte t_{sync} bei 3 synchronisierten Abonnenten. Zu diesem t_{sync} müsste sich ein vierter Abonnent synchronisieren usw.

Die allgemeine Formel zur Berechnung der aggregierten Periode $p_{1,\ldots,n}$ über alle Perioden p_i, $i = 1, \ldots, n$ lässt sich rekursiv mit Hilfe der Funktion $Agg : \mathbb{N}^+ \rightarrow \mathbb{N}$ definieren. Die aggregierte Periode $p_{1,\ldots,n}$, mit

$$p_{1,\ldots,n} = Agg(n) =_{def} \frac{Agg(n-1) \cdot p_n}{ggt(Agg(n-1), p_n)}$$

$$Agg(1) =_{def} p_1$$

ist der Abstand der Zeitpunkte t_{sync} bei n synchronisierten Abonnenten.

3.3.3 Timer-Pool-Algorithmus zur Synchronisation der Abonnenten

Aus diesen Erkenntnissen lässt sich ein Algorithmus zur Synchronisation der Abonnenten ableiten, der immer dann abgearbeitet wird, wenn ein neuer Abonnent hinzu kommt. Der Algorithmus verwaltet für die Abonnenten eine Reihe von *Timern*, Datenstrukturen zum Erfassen der verstreichenden Zeit, die nach dem Ablauf einer Zeitspanne weitere Schritte auslösen. Dem Algorithmus steht eine Sammlung solcher Timer zur Verfügung, dies ist sein sogenannter *Timer-Pool*. Er kann Elemente aus dem Pool entfernen und auch neue hinzufügen. Zum besseren Verständnis der nachfolgenden Darstellung des Algorithmus in Pseudo-Kode wird angenommen, der Timer-Pool sei als Liste realisiert, in der jedes Element ein einzelner Timer ist. Jeder Timer besitzt selbst eine Liste, in der alle Abonnenten stehen, die bei Ablauf des Timers Messwerte erhalten: die Zuständigkeitsliste des Timers. Außerdem kenne ein Timer seinen Auslösezeitpunkt. Ein Abonnent ist durch eine ID und seine Periode, in denen er Daten benötigt, definiert. Diese Struktur des Timer-Pools ist in Abb. 3.10 dargestellt. Ziel dieses nachfolgend vorgestellten Algorithmus, ist die Reduzierung der Anzahl an angeforderten Messwerten eines Dienstes. Dies wird durch geschicktes Verwalten der angemeldeten Abonnenten und der benötigten *Timer* erreicht. Der Algorithmus ist in Listing 3.5 und 3.6 dargestellt.

Wird ein Abonnent angemeldet, so findet eine Synchronisation mit den anderen bereits angemeldeten Abonnenten statt, nach den Anweisungen in Listing 3.5. Er wird also zunächst mit der oben vorgestellten Formel berechnen, wann erstmalig Daten für den Abonnenten angefordert werden. Anschließend prüft er, ob einer der derzeit laufenden Timer des Timer-Pool zum berechneten Zeitpunkt abläuft. Ist das der Fall, dann trägt er den Abonnenten in die Zuständigkeitsliste des Timers ein. Falls nicht, legt er einen entsprechenden Timer an und fügt ihn dem Timer-Pool hinzu (mit dem Abonnenten als einzigem Eintrag seiner Zuständigkeitsliste).

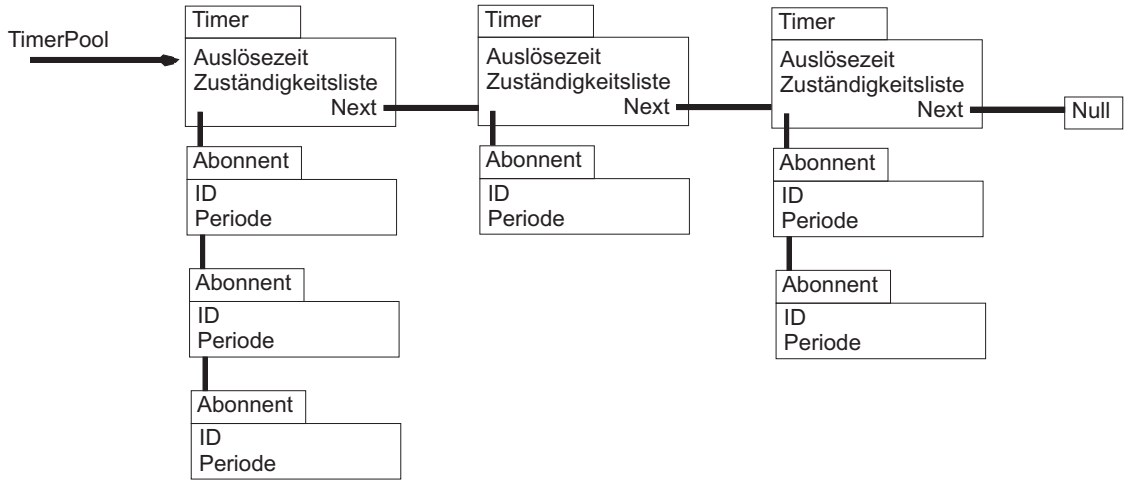

Abbildung 3.10: Schematische Darstellung der Datenstruktur `TimerPool`.

Die Berechnung des Zeitpunktes der ersten Daten-Anforderung wurde zur besseren Übersichtlichkeit in eine eigene Funktion ausgelagert.

Im Pseudo-Kode vernachlässigt ist die Behandlung des Falles, wenn noch kein Abonnent angemeldet ist und t_{sync} daher noch nicht initialisiert wurde. Die Umsetzung der Funktion in SDL (s. Abb. 3.11) ist vollständig und behandelt daher beide Fälle.

Für die Umsetzung in SDL wurde die Datenstruktur *Timer-Pool* mittels eines SDL-Arrays wie in Abb. 3.12 realisiert. Dadurch muss bei der Suche nach einem passenden Timer keine Liste durchlaufen werden, sondern eine Zelle des Arrays kann mit dem berechneten Zeitpunkt als Index direkt adressiert werden. Ein Flag `timerStarted` zeigt an, ob bereits ein Timer läuft, der zu diesem Zeitpunkt abgelaufen sein wird (der Timer *feuert*). Ist das *Flag* nicht gesetzt, so wird ein `RequestTimer` mit der berechneten Zeit gestartet und das Flag gesetzt. Der Timer trägt dabei seinen Auslösezeitpunkt als Parameter. Läuft einer dieser Timer ab, wird anhand des Parameters die entsprechende Zelle im Array ausgelesen und die Zuständigkeitsliste `subscribers` abgearbeitet. Anschließend wird der Zelleninhalt mit dem *Default*-Wert überschrieben. Eine Optimierung der Laufzeitumgebung wird dann dafür sorgen, dass Zellen, die den *Default*-Wert enthalten, keinen Speicherplatz belegen – andernfalls würde das Array immer weiter wachsen und wäre daher für die Realisierung des Timer-Pools nicht geeignet.

Die Suche nach einem geeigneten Timer im Timer-Pool kommt in SDL daher ohne Schleifendurchläufe aus. Abbildung 3.13 zeigt einen Ausschnitt der Spezifikation.

Die Berechnung der Zeitpunkte t_{sync} (in der SDL-Spezifikation `nextSync` genannt) wird nicht durch die Anmeldung eines neuen Abonnenten ausgelöst. Denn ein neuer Abonnent richtet sich nach dem bisherigen Zeitpunkt t_{sync}, verändert ihn daher nicht. Durch den neuen Abonnenten kommt eine weitere Periode zu den Perioden der bisher angemeldeten hinzu. Das hat aber erst bei der erneuten Berechnung von t_{sync} Auswirkungen. Ein Timer läuft zum Zeitpunkt t_{sync} ab, was eine Neuberechnung auslöst. Bei der Realisierung in SDL konnte dabei auf die vorhandenen *RequestTimer* zurückgegriffen werden. Da immer auch garantiert ein *RequestTimer* zum Zeitpunkt

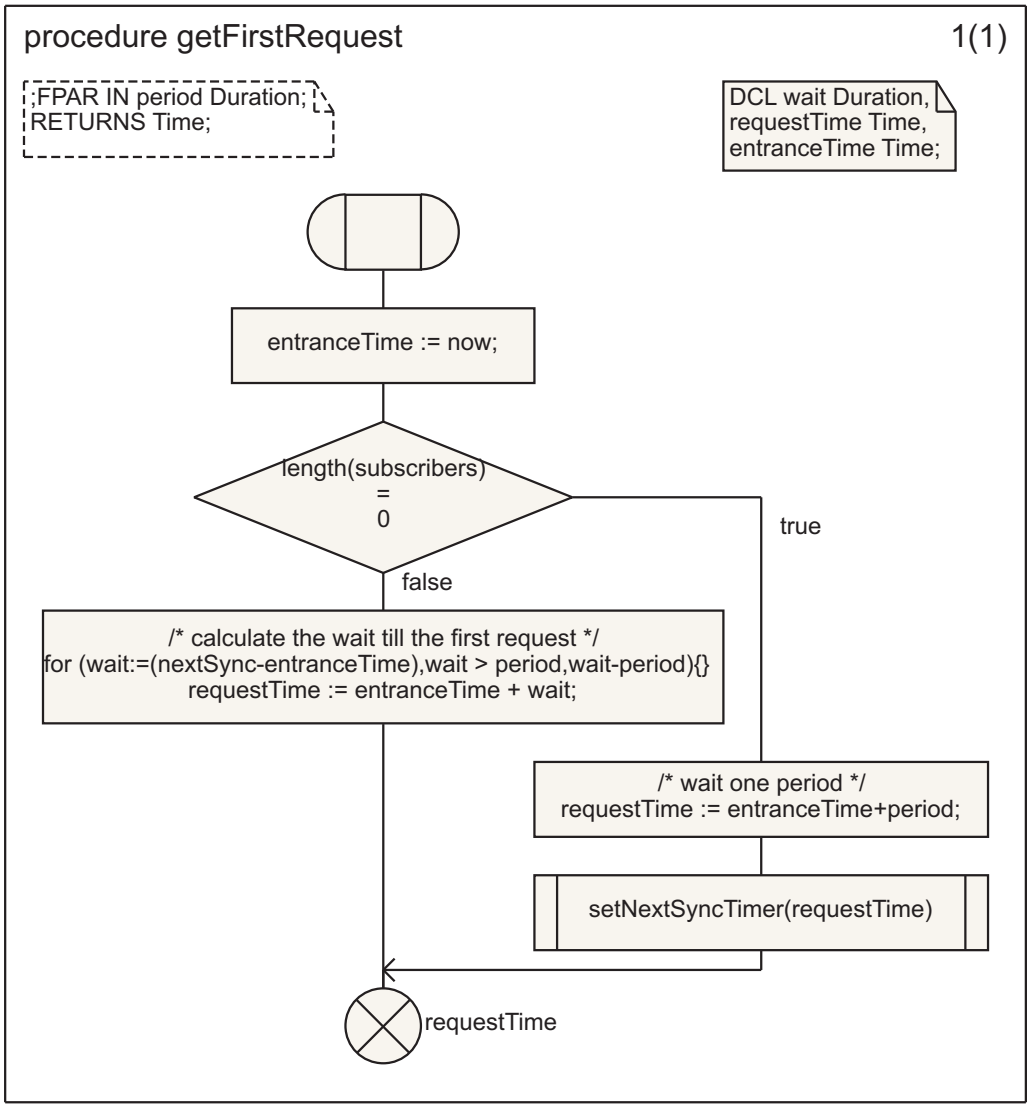

Abbildung 3.11: Umsetzung des Algorithmus in SDL.

```
TIMER RequestTimer(Time);

NEWTYPE SubscriberList String(Integer,empty)
ENDNEWTYPE;

NEWTYPE ScheduledRequest STRUCT
subscribers SubscriberList;
timerStarted Boolean;
isNextSyncTimer Boolean;
default (. empty,false,false .);
ENDNEWTYPE;

NEWTYPE Schedule Array(Time,ScheduledRequest)
ENDNEWTYPE;

DCL timerPool Schedule;
```

Abbildung 3.12: Deklaration der Datenstruktur `TimerPool` in SDL als Array.

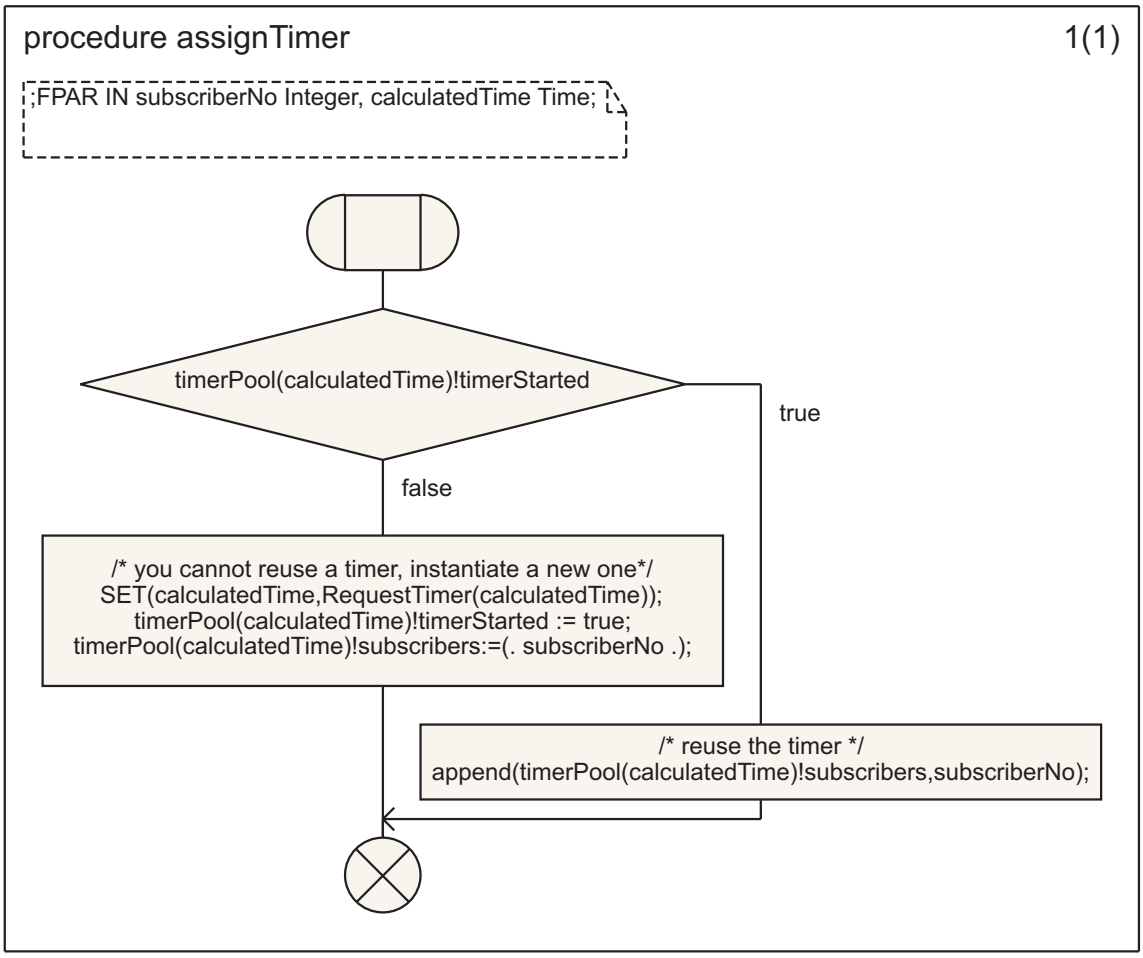

Abbildung 3.13: Umsetzung des Algorithmus in SDL.

```
1  // die Funktion fügt einen weiteren Abonnenten den schon
2  // angemeldeten hinzu und synchronisiert ihn mit den anderen
3  addSubscriber(id, periode) {
4      t_first_data = getFirstRequestTime(periode);
5
6      // find and assign a timer
7      for (element : TimerPool) {
8          if (element.auslösezeitpunkt == t_first_data) {
9              append(element.zuständigkeitsliste, id);
10             timerReused = true;
11             break;
12         }
13     }
14
15     // create a new timer if needed
16     if (timerReused == false) {
17         timer = new Timer();
18         timer.auslösezeitpunkt = t_first_data;
19         append(timer.zuständigkeitsliste, id);
20         append(TimerPool, timer);
21     }
22 }
```

Listing 3.5: Synchronisations-Algorithmus in Pseudo-Kode.

t_{sync} abläuft, wird kein separater *NextSyncTimer* angelegt, sondern genau einem Timer im Pool das Flag `isNextSyncTimer` gesetzt. Läuft dieser Timer ab, wird t_{sync} neu berechnet und ein neuer *RequestTimer* auf t_{sync} gesetzt (mit anfänglich leerer Zuständigkeitsliste). Im Laufe der Zeit werden der Zuständigkeitsliste dieses Timers mehr und mehr Abonnenten hinzugefügt, bis er schließlich unmittelbar vor dem Zeitpunkt t_{sync} nur noch als einziger Timer im Pool enthalten ist, der alle Abonnenten in seiner Liste führt.

Läuft ein Timer ab, fordert der Algorithmus für alle Abonnenten in der Zuständigkeitsliste Daten an. Anschließend entfernt er den abgelaufenen Timer aus dem Timer-Pool. Dann berechnet er für jeden Abonnenten in der Zuständigkeitsliste den jeweils nächsten Daten-Anforderungszeitpunkt und sucht einen jeweils passenden Timer aus dem Timer-Pool, um dessen Zuständigkeitsliste zu erweitern. Gibt es keinen passenden Timer, legt er einen solchen an.

Der Timer-Pool enthält unmittelbar vor einem Zeitpunkt t_{sync} nur noch einen einzigen Timer, in dessen Liste alle Abonneten enthalten sind. Dies ist nützlich, wenn es darum geht, Abonnenten wieder abzumelden. Denn wird ein Abonnent abgemeldet, so wird er sicher noch in einer der Zuständigkeitslisten enthalten sein. Um ein Durchsuchen aller Listen zu vermeiden, wird stattdessen eine Liste aller abgemeldeten Abonnenten geführt. Sobald ein Timer feuert und die Zuständigkeitsliste abgearbeitet werden soll, so wird sie zuvor anhand der Liste abgemeldeter Abonnenten gefiltert, um zu vermeiden, dass für diese Abonnenten noch Daten angefordert werden. Feuert ein *NextSync*-Timer, kann die Liste abgemeldeter Abonnenten anschließend wieder vollständig entleert werden, da garantiert keine weiteren Zuständigkeitslisten mehr existieren. Auch das Filtern vor der Abarbeitung einer

```
1  // die Funktion berechnet anhand der Periode den Zeitpunkt t_first_data
2  // zu dem ein Abonnent das erte mal Daten erhält
3  getFirstRequestTime(periode) {
4      t_entrance = NOW;
5      wartezeit = t_sync - t_entrance;
6      while (wartezeit > periode) {
7          wartezeit = wartezeit - periode;
8      }
9      t_first_data = t_entrance + wartezeit;
10     return t_first_data;
11 }
```

Listing 3.6: Synchronisations-Algorithmus in Pseudo-Kode.

Zuständigkeitsliste kann anschließend solange übersprungen werden, bis wieder ein Abonnent abgemeldet wurde.

Kapitel 4

Duty-Cycling in MacZ

Das Protokoll MacZ [BGK07] verfügte bisher über keine Mechanismen zur Reduzierung des Energieverbrauchs. Knoten, auf denen MacZ für den Medienzugriff eingesetzt wurde, blieben für die gesamte Lebensdauer eingeschaltet. Dadurch sind sie zwar in der Lage, zu jedem Zeitpunkt mit dem Versand eigener Nachrichten zu beginnen. Eine Applikation unterliegt darum auch keiner Einschränkung in Bezug auf den Zeitpunkt, zu dem sie Daten generiert und versenden will. Verwaltungsinformationen, die unterhalb der Applikationsschicht auf einer Protokoll-Ebene anfallen, wie Routing-Informationen, Reservierungsanfragen und Synchronisationssignale sind ebenfalls zu jedem Zeitpunkt mit anderen Knoten austauschbar und müssen nicht im vorraus geplant oder bis zur nächsten Sende-/Empfangsmöglichkeit gesammelt werden. Aber Teile der Hardware wie z.B. Sensoren oder Transceiver werden vorgehalten, ohne dass sie benötigt werden. Jede Zeitspanne, die solche Hardwarekomponenten ungenutzt sind, bedeutet daher einen potentiell vermeidbaren Energieverbrauch. Darum wurde im Rahmen dieser Arbeit ein Duty-Cycling-Mechanismus für MacZ spezifiziert und integriert.

Von den in Abschnitt 2.4 genannten Voraussetzungen, die für den erfolgreichen Einsatz des Mechanismus erfüllt sein müssen, ist die Forderung nach Ticksynchronisation bereits erfüllt. Die Synchronisation der beteiligten Knoten wird durch die Verwendung von Black-Burst-Synchronisation [GK08] sichergestellt. In MacZ ist bisher die Systemlaufzeit schon in *Macroslots* statisch festgelegter Länge unterteilt. Ein *Macroslot* setzt sich aus einer Synchronisationsphase und einer festen Anzahl an *Microslots* zusammen. *Microslots* können zu Slot-Regionen unterschiedlicher Nutzungsbedingungen zusammengefasst werden (siehe Abbildung 4.1):

- **Idle:** Bezeichnet ungenutzte Regionen. Der Transfer von Nachrichten ist in dieser Zeit nicht vorgesehen.

- **Prio:** Bezeichnet wettbewerbsbasierte Regionen. Der Sender einer Nachricht muss sich vor dem Versand um das Medium bewerben, wobei Nachrichten Prioritäten zugeordnet sind, die es erlauben, höher priorisierte Nachrichten garantiert oder statistisch vor Nachrichten niedrigerer Priorität versenden zu können.

- **Res:** Bezeichnet wettbewerbsfreie Phasen. Die Region kann als reserviert angesehen werden, da sie für den Versand der Nachrichten eines bestimmten Knotens verplant ist.

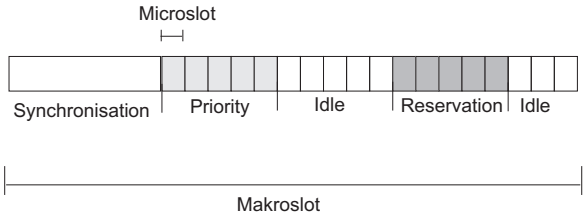

Abbildung 4.1: Ein Beispiel für die Verteilung von Slotregionen in einem Makroslot.

Durch die feste Länge der *Macroslots* kehren Synchronisationsphasen, in denen die Knoten resynchronisiert werden, streng periodisch wieder. Dies ist zum Einen nötig, um neu in das Netz eintretende Knoten mit den anderen zu synchronisieren, zum Anderen, um den ständig vorhandenen Uhrenversatz der schon synchronisierten Knoten wieder zu reduzieren; dieser würde sonst zu stark anwachsen.

Diese Einteilung der Laufzeit in fest strukturierte Zyklen kann für den Duty-Cycling-Mechanismus genutzt werden. Über die allen Knoten gemeinsame *Macroslot*-Konfiguration wird ein Schedule, der Aktiv- und Passiv-Phasen festschreibt, gelegt. Jeder Knoten verfügt über einen individuellen Schedule, der auf die dem Knoten zugeteilte Aufgabe zugeschnitten ist. Die Macroslot-Konfiguration ist daher nicht mit dem Schedule eines Knoten gleichzusetzen. Erstere ist für alle Knoten gleich, Letzterer ist individuell.

4.1 Verteilung von Aktiv- und Schlafphasen

Das Duty-Cycling in MacZ konzentriert sich auf die Zustände der Transceiver-Hardware. Für jeden Knoten muss daher geprüft werden, zu welchen Zeitpunkten und für wie lange gesendet oder empfangen werden soll. Dazu wird eine am Szenario ausgerichtete Verteilung der Slot-Regionen innerhalb des *Macroslots* bestimmt, wie in Abb. 4.2 gezeigt. Der Macroslot-Konfiguration liegen folgende Regeln zu Grunde:

1. Der Macroslot beginnt mit einer Synchronisationsphase.

2. Für Nachrichten, die unregelmäßig oder unplanmäßig auftreten, werden eine oder mehrere wettbewerbsbasierte Regionen vorgesehen. Zu diesen Nachrichten zählen im Inversen-Pendel-Szenario alle anfänglichen Dienstregistrierungen, alle zugehörigen Dienstabonnements, alle Alive-Nachrichten (obwohl streng periodisch, aber da der genaue Zeitpunkt der Dienstregistrierung nicht auf Macroslotgrenzen ausgerichtet ist, ist deren Zeitpunkt des Auftretens innerhalb des Macroslots nicht bekannt) und alle Subscribed-Nachrichten (der Grund hierfür ist analog zu den Alive-Nachrichten).

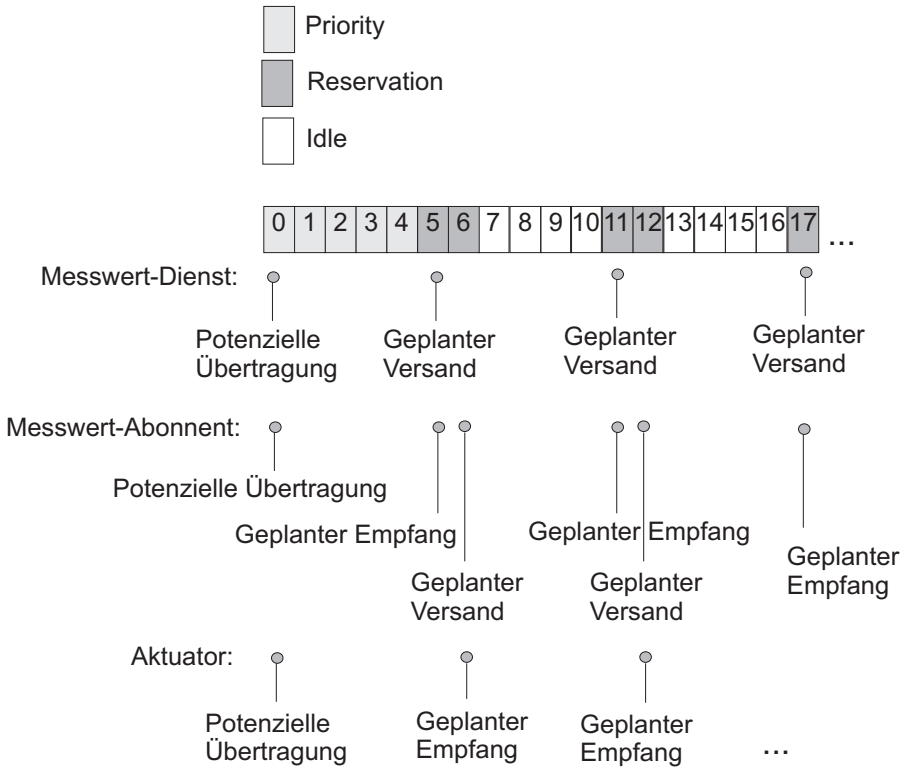

Abbildung 4.2: Ein Beispiel für die Verteilung von Slotregionen – abgeleitet aus dem Kommunikationsverhalten der Knoten.

3. Für Nachrichten, deren zeitliches Auftreten relativ zu Macroslotgrenzen bekannt ist, werden wettbewerbsfreie Regionen eingeführt, so dass diese Nachrichten ohne Verzögerung versandt werden können. Zu diesen Nachrichten zählen – im hier verwendeten Szenario – die Messwerte der Sensoren sowie die Steuersignale des Controllers an den Aktuator.

4. Ungenutzte Regionen werden zu Idle-Regionen

Mit der fertigen *Macroslot*-Konfiguration liegen Informationen vor, die anderen Duty-Cycling-Protokollen, wie den in Kapitel 2 beschriebenen, nicht zur Verfügung stehen. Sie müssen regelmäßig Wachphasen ermöglichen, da sie nicht wissen, ob ein Knoten senden will. Bei MacZ wird Anwendungswissen herangezogen, um die Macroslot-Konfiguration zu erstellen – das gleiche Wissen wird auch zur Planung der Wach- und Schlafphasen genutzt.

4.1.1 Zweckgebundene Wachphasen

Auf Grundlage dieser Konfiguration der Macroslots wird pro Knoten ein Ereignis-Plan (*Schedule*) angelegt. Im Schedule sind keine Wach- oder Schlaf-Phasen verzeichnet, sondern Ereignisse. Ereignisse, die eine Aktivierung des Transceivers erfordern, führen zu Wach-Phasen des Transceivers.

Da während der Synchronisationsphase alle Knoten wach sind, d.h. ihre Transceiver-Hardware eingeschaltet ist, ist die Synchronisationsphase analog zum Macroslot nicht Teil des Schedules. Prinzipiell ist ein solcher Schedule für Aktiv-Passiv-Phasen einer Hardwarekomponente unabhängig von der zugrundeliegenden Macroslot-Konfiguration. Der Schedule zur Energiezustandsregelung des Transceivers enthält aber sinnvollerweise nur dann Aktivphasen, wenn die Macroslot-Konfiguration eine Übertragung zulässt. Da der Transceiver nur eingeschaltet wird, wenn er auch gebraucht wird, ist jede seiner Wachphasen einem eindeutigen Zweck zugeordnet.

Jede Wachphase ist im Schedule mit ihrem Anfangszeitpunkt und ihrem Zweck verzeichnet – je nach Zweck auch noch mit ihrer Dauer (vgl. Tabelle 4.1). Die tatsächliche Länge der Aktivphase ergibt sich erst zur Laufzeit durch Eintreten von Sende- und Empfangs-Ereignissen. Ist der Zweck der Aktiv-Phase erfüllt, muss deren durch die Dauer angegebenes Ende nicht abgewartet werden, um den Transceiver wieder abzuschalten. Da die Anfangszeitpunkte mit den Slotgrenzen der Microslots zusammenfallen, werden sie in Form der Slotnummer des Microslots, in dem der Transceiver gebraucht wird, angegeben und sind damit relativ zum Beginn des Macroslots. Passiv-Phasen werden nicht explizit im Schedule gelistet, sondern ergeben sich implizit aus allen Zeiträumen, die nicht in eine Aktiv-Phase fallen.

Für mögliche Ereignisse lassen sich drei verschiedene Zwecke identifizieren. Diese sind im Einzelnen:

1. **Potenzielle Übertragung** (Potential Transfer) Während einer wettbewerbsbasierten Phase könnten andere Knoten Nachrichten für den eigenen Knoten senden, deren Versand nicht vorher angekündigt war. Ebenso könnten sich an andere Knoten gerichtete Nachrichten auf dem eigenen Knoten ergeben haben. Da ein Nachrichtentransfer zu jedem Zeitpunkt dieser Phase angestoßen werden kann, muss der Knoten für die Dauer der Phase empfangsbereit sein. Des Weiteren wurde durch den Eintrag des Ereignisses in den Schedule auch keine Aussage hinsichtlich der Anzahl der potentiellen Übertragungen getroffen. Auf eine erfolgte Übertragung könnten also weitere folgen.

   Einige Ereignisse können aber das Abschalten des Transceivers während der Phase ermöglichen: Sobald ein RTS-, CTS- oder Daten-Signal empfangen wird, das nicht für den eigenen Knoten bestimmt ist und die Länge der zu übertragenden Daten angibt, kann daraus errechnet werden, wie lange das Medium durch diese Datenübertragung belegt sein wird. Das Protokoll wird seinen Network Allocation Vector (NAV) für diesen Zeitraum setzen, um in dieser Zeit keine eigenen Sendeversuche zu unternehmen. Der Empfang von für den eigenen Knoten bestimmten Nachrichten ist auch durch die laufende Übertragung ausgeschlossen. Der Transceiver kann also während der Phase zeitweilig für alle Zeiträume, in denen der NAV gesetzt ist, abgeschaltet werden.

   Um nicht durch eine noch laufende Übertragung über das Ende der wettbewerbsbasierten Phase hinaus Übertragungen der anschließenden SlotRegion zu stören, dürfen neue Nachrichten nur bis zu einer zeitlichen Grenze in einem festen, ausreichenden Abstand vor dem Ende der Region noch gestartet werden. Die durch Grenze und Phasenende gebildete Region ist die Receive-

Only-Phase, in der nur noch der Empfang möglich ist. Der Transceiver kann daher endgültig ausgeschaltet werden,...

- ...sobald während der Receive-Only-Phase ein Paket empfangen wurde. Denn weitere Nachrichten können jetzt nicht mehr gestartet werden.
- ...falls beim Übergang in die Receive-Only-Phase der eigene Knoten noch am Senden ist. Abgeschaltet wird nach Abschluss des Sendevorgangs. Im Anschluss können keine weiteren Nachrichten begonnen werden – weder an andere Knoten gerichtet noch von anderen an den eigenen Knoten adressiert.
- ...falls beim Übergang in die Receive-Only-Phase der NAV gesetzt ist. Denn das bedeutet, eine Übertragung zwischen anderen Knoten ist noch im Gange. Daran können im Anschluss ebenfalls keine weiteren Nachrichten versandt werden.
- ...falls beim Übergang in die Receive-Only-Phase das Medium nicht belegt ist, was durch Clear-Channel-Assessment (CCA) ermittelt wird. Das bedeutet, dass keine Übertragung stattfindet – und innerhalb der Receive-OnlyPhase kann keine Übertragung begonnen werden. Jedoch könnte folgendes Problem auftreten: Ein Knoten (sogar der eigene) könnte kurz zuvor einen Rahmen zum Versand nach unten an den zuständigen Prozess gereicht haben, dieser hat den Rahmen schon an den Transceiver-Treiber weitergereicht und wartet auf eine Bestätigung. Der Treiber hat es aber noch nicht auf das Medium gelegt. Alle Knoten erkennen das Medium noch als frei. Die Receive-Only-Phase ist dafür gedacht, dass solche Rahmen noch erfolgreich versendet und empfangen werden. Implementiert ein Knoten ein Duty-Cycling, das in diesem Fall den Transceiver abschaltet, zerstörte er damit den Sinn der Receive-Only-Phase (zumindest in diesem speziellen Fall). Daher ist ein *Guard*-Intervall nötig, sodass ein Stück weit in die Receive-Only-Phase hinein das Medium beobachtet wird und erst nach dem Sicherheitsintervall davon ausgegangen wird, dass niemand mehr sendet.
- ...die Receive-Only-Phase zu Ende ist.

2. **Geplanter Versand** (Planned Transmission) Der Versand einer Nachricht ist für einen festgelegten Zeitpunkt innerhalb einer wettbewerbsfreien Phase geplant – alle Knoten wissen dies vorab (zur Zeit statisch festgelegt, zukünftig über ein Reservierungsprotokoll auszuhandeln). Da die Übertragung geplant ist, sind lediglich der Sender und der Empfänger aktiv. Alle anderen Knoten können ihren Transceiver ausschalten. Ob aber tatsächlich eine Nachricht versandt wird, hängt davon ab, ob die Applikation zum geplanten Zeitpunkt eine Nachricht bereitstellt. Daher ergeben sich zwei Ereignisse, die ein Abschalten des Transceivers auslösen:

   - Der Transceiver kann ausgeschaltet werden, sobald das Paket versandt wurde. Denn unabhängig davon, wie viel Zeit für den Versand vorgesehen war, ist durch den Eintrag der Aktivität in den Schedule nur der Versand genau einer Nachricht geplant. Weitere Nachrichten auch innerhalb

der gleichen wettbewerbsfreien Slot-Region sind durch weitere Schedule-Einträge definiert. Für die verbleibende Zeit, die sich aus der geplanten Dauer dieser Aktivität ergibt, kann der Transceiver ausgeschaltet werden.

- Der Transceiver kann ausgeschaltet werden, wenn zum geplanten Versand-Zeitpunkt keine entsprechende Nachricht bereitgestellt wurde. Dies kann vorkommen, wenn die Berechnungszeit zur Erzeugung von Nachrichten länger als geplant ausfiel, oder der Zeitpunkt nur vorsorglich reserviert wurde aber nicht jedesmal genutzt wird. Die Einhaltung eines *Guard*-Intervalls ist hier nicht nötig, da der für das Senden zuständige Prozess auf seine Anfrage an den Paket-Puffer in einem solchen Fall die eindeutige Mitteilung erhält, dass kein Paket vorhanden ist. Ein weiteres Abwarten ist nicht vorgesehen, es findet keine zweite Anfrage an den Puffer nach dem gleichen Paket statt.

3. **Geplanter Empfang** (Planned Reception) Der Empfang einer Nachricht ist für einen festgelegten Zeitpunkt innerhalb einer wettbewerbsfreien Phase geplant – alle Knoten wissen dies vorab. Da die Übertragung geplant ist, sind lediglich der Sender und der Empfänger aktiv. Alle anderen Knoten können ihren Transceiver ausschalten. Ob aber tatsächlich eine Nachricht empfangen wird, hängt davon ab, ob der Sender den geplanten Versand durchführt und die Nachricht störungsfrei über das Medium transportiert wird. Es ergeben sich drei Ereignisse, die ein Abschalten des Transceivers auslösen:

   - Der Transceiver kann ausgeschaltet werden, sobald das Paket empfangen wurde. Denn unabhängig davon, wie viel Zeit für den Empfang vorgesehen war, ist durch den Eintrag der Aktivität in den Schedule nur der Empfang genau einer Nachricht geplant. Weitere Nachrichten auch innerhalb der gleichen wettbewerbsfreien Slot-Region sind durch weitere Schedule-Einträge definiert. Für die verbleibende Zeit, die sich aus der geplanten Dauer dieser Aktivität ergibt, kann der Transceiver ausgeschaltet werden.
   - Der Transceiver kann ausgeschaltet werden, sobald die geplante Dauer der Aktivität verstrichen ist, ohne dass eine Nachricht empfangen wurde. Dies kann vorkommen, wenn der Sender den geplanten Versand nicht durchführt oder die Nachricht nicht störungsfrei über das Medium übertragen wurde.
   - Der Transceiver kann ausgeschaltet werden, sobald das Medium nach einer entsprechend gewählten Zeitspanne t_{guard} ab Beginn der *Planned-Reception*-Phase immer noch nicht belegt ist. Denn führt der Sender tatsächlich einen Versand durch, so wird er damit nach spätestens der Zeitspanne t_{guard} begonnen haben (gemessen ab dem im Schedule eingetragenen Beginn der Aktivität). Diese setzt sich wie folgt zusammen: Die Dauer zur Ausbreitung des Signals vom Sender bis zum Empfänger sei t_{prop}. Der maximal mögliche Uhrenversatz sei $t_{clockDrift}$, die Rechenzeit durch interne Abarbeitung des Protokolls sei $t_{transitions}$ und die zum Erkennen eines belegten Mediums auf Empfängerseite nötige Dauer sei

Index	0	1	2
Zweck	Potenzielle Übertragung	Geplanter Empfang	Geplanter Versand
Slotnummer	0	5	6
Dauer in Slots	5	1	-

Tabelle 4.1: Schedule-Ausschnitt des Messwert-Abonnenten aus Abb. 4.2

.

t_{cca}. Mit diesen Werten lässt sich das *Guard*-Intervall t_{guard} nach oben abschätzen:

$$t_{guard} \geq t_{clockDrift} + t_{transitions} + t_{prop} + t_{cca}$$

Die Konfiguration eines Macroslots wurde für beteiligte Knoten so gewählt, dass die für das Szenario geplante Kommunikation zustande kommen kann. Jeder Knoten hat dabei die gleiche Macroslot-Konfiguration, denn nur dann ist sichergestellt, dass z.B. während wettbewerbsfreier Slotregionen kein anderer, außer den dafür vorgesehenen Knoten zu kommunizieren versucht. Dies würde andernfalls die planmäßige Übertragung stören. Tabelle 4.1 zeigt die ersten drei Einträge im Schedule des Messwert-Abonnenten aus Abb. 4.2. Die Verteilung von Wach- und Schlafphasen kann grundsätzlich unabhängig von der netzweit gültigen Macroslot-Konfiguration gewählt sein. Sie ist für jeden Knoten individuell. Sinnvollerweise sind aber zumindest der Schedule für den Transceiver und die Macroslot-Konfiguration aufeinander abgestimmt.

4.2 Architektur

Die Architektur von MacZ (vgl. Abb. 4.3) verteilt nötige Funktionen auf zwei Schichten. Eine untere Schicht (*BasicLayer*) für grundlegende Funktionen und Mechanismen wie z.B. die Kodierung von *Black Bursts* und die Taktvorgabe mittels Bekanntgabe von Slotgrenzen, und eine höhere Schicht (*ServiceLayer*) für Funktionen, die auf den Grundfunktionen aufbauen und komplexere Aufgaben erfüllen. Dies sind z.B. das Verwalten der Slot-Regionen und das Senden und Empfangen von Nachrichten. Ein Multiplexer leitet Signale aus tieferen Ebenen des Protokoll-Stapels an die zuständige MacZ-Schicht und umgekehrt. Der *ServiceLayer* gliedert sich in die zwei Komponeneten *Storage* und *TxRx* (in Abb. 4.4 dargestellt). Erstere ist für die Pufferspeicherung der Pakete zuständig, die von der Anwendung (oder höhergelegenen Ebenen des Protokollstapels) stammen, letztere für den Rahmenversand und -empfang (Tx und Rx als Abkürzungen für *Transmit* und *Receive*).

Eine für die Energie-Verwaltung zuständige Komponente *EnergyManagement* wurde neu in die MAC-Schicht eingeführt – sie ist für die Durchführung des Duty-Cyclings verantwortlich. Abbildung 4.5 zeigt ihre Einbettung in die Komponente *TxRx*. Da sie zur Erfüllung ihrer Aufgabe Statusmeldungen der Sende- und Empfangskomponente benötigt und auf die Taktvorgabe der Grundlagen-Schicht aufbaut, gehört sie konzeptionell in den *ServiceLayer*. Das in dieser Arbeit realisierte EnergyManagement

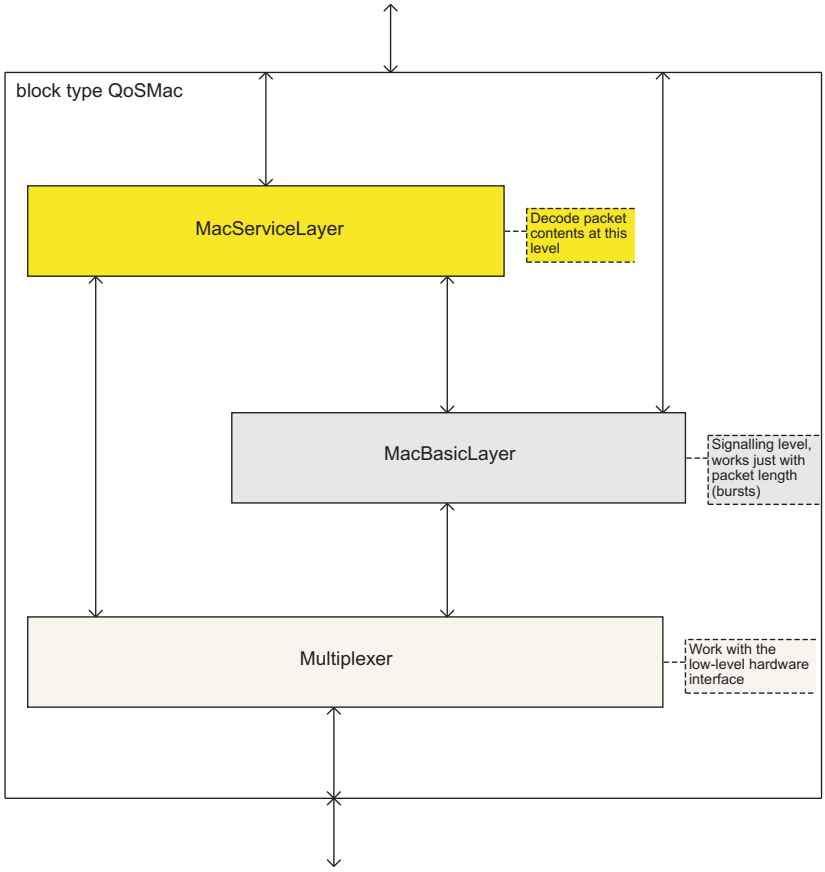

Abbildung 4.3: Schematische Darstellung der Schichten-Architektur von MacZ.

steuert nur die Transceiver-Hardware und bezieht Nachrichten von Funktionen der Sende- und Empfangskomponente (TxRx). Daher wurde sie ebenfalls in diese Komponente eingeordnet.Spezialisierte zukünftige EnergyManagement-Komponenten für andere Hardware-Elemente sind analog dazu in deren Komponenten einzuordnen. So kann im System beispielsweise eine Sensor-Komponente vertreten sein. Erlaubt die Sensor-Hardware verschiedene Energie-Zustände, könnte die Komponente um ein Sensor-Energy-Management erweitert werden.

Der ServiceLayerMultiplexer (SLMux) leitet von unten kommende Signale an die verschiedenen Komponenten des ServiceLayers weiter und deren Signale nach unten. Nach dem Einfügen des EnergyManagements wurde der Multiplexer so erweitert, dass er die Signale NoSync_IND, MacroSlotEnd_IND, MicroSlot_IND(slot) und MacroSlot_IND an das EnergyManagement weiterleitet.

- NoSync_IND zeigt den Verlust der Synchronisation an. Für eine Resynchronisation muss der Transceiver daher wieder eingeschaltet werden.
- MacroSlotEnd_IND zeigt das Ende eines Makroslots an. Da darauf eine Synchronisation folgt, muss der Transceiver wieder eingeschaltet werden.
- MicroSlot_IND(slot) zeigt den Beginn eines Microslots und die Slotnummer an.
- MacroSlot_IND zeigt den Beginn eines Macroslots an.

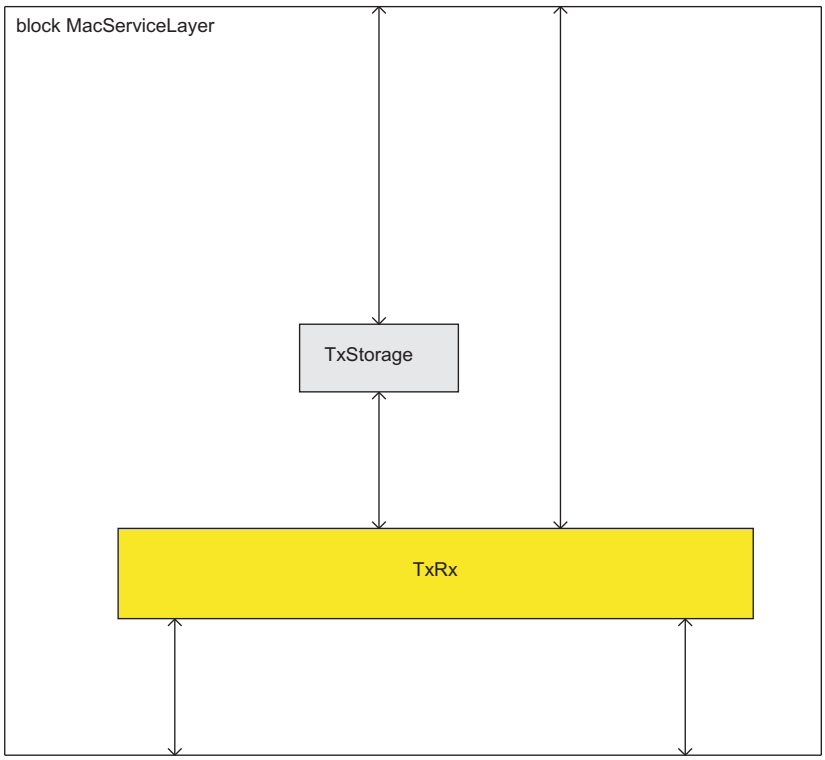

Abbildung 4.4: Komponenten innerhalb des *ServiceLayers*.

Das EnergyManagement benötigt diese Signale, da Ereignisse des Schedules relativ zu Makroslotgrenzen (mit Hilfe der Microslot-Nummern) festgelegt sind. Des Weiteren benötigt es Meldungen über den abgeschlossenen Versand oder Empfang einer Nachricht und der Belegungsdauer eines als belegt erkannten Mediums. Diese Meldungen stammen aus der Komponente *SendRecv*, die die Funktionen für Versand und Empfang enthält. Zwischen den Komponenten *SendRecv* und *EnergyManagement* besteht daher eine Verbindung, wie in Abb. 4.5 zu sehen ist.

Die Komponente *SendRecv* beinhaltet die Funktionen für Senden und Empfangen von Nachrichten. Da sich das Senden und Empfangen während einer *Reserved*-Region vom Senden und Empfangen wärend einer *Priority*-Region unterscheidet, existieren die Funktionen – wie in Abb. 4.6 zu sehen – jeweils für beide Regionstypen. Die *SendRecv*-Komponente ist daher in zwei Teile untergliedert, *Prio* und *Res*, die beide jeweils diese zwei Funktionen darstellen. Die Komponente *Controller* steuert abhängig von der gegenwärtigen Microslotnummer und der Macroslotkonfiguration, welche der beiden Komponenten für Sende- und Empfangsvorgänge eingesetzt wird. Die beiden Komponenten sind nie gleichzeitig aktiv. Es können aber sehr wohl beide gleichzeitig deaktiviert sein, was innerhalb einer *Idle*-Region der Fall ist. Beide haben für ihre Meldungen an das *EnergyManagement* einen eigenen Kanal (`prio2energyMngmt` bzw. `res2energyMngmt`).

Die Prio-Komponente Die Prio-Komponente ist während einer Prio-Region aktiv. Während dieser Region ist das EnergyManagement im Zustand *potential_transfer*. Die Prio-Komponente hat für ihre Aufgabe – Versenden und Empfangen

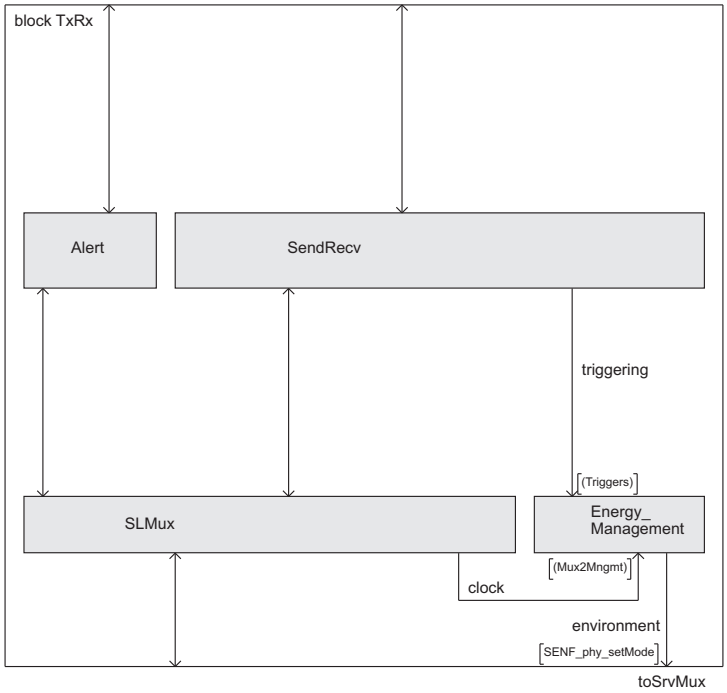

Abbildung 4.5: Elemente der Komponente *TxRx*: *Alert*-Funktion (nicht näher behandelt), Sende- und Empfangskomponente, *ServiceLayer*-Multiplexer und *Energy-Management*.

von Paketen während der Prio-Slotregion – verschiedene Prozesse, die unterschiedliche Funktionen erfüllen, wie z.B. Überwachen des *Network Allocation Vectors* (NAV) oder Durchführen von *Clear Channel Assessment*. Die Prio-Komponente muss dem EnergyManagement melden, wenn ein energierelevantes Ereignis stattfindet. Diese Ereignisse sind weiter vorne aufgeführt. Die Prio-Komponente wurde um das Verhalten erweitert, diese Ereignisse zu melden. So meldet die Komponente immer dann, wenn der NAV gesetzt wird, den Zeitraum, den das Medium voraussichtlich belegt sein wird. Des Weiteren prüft sie beim Übergang in die Receive-Only-Phase...

- ...ob das Medium belegt ist, was durch *Clear Channel Assessment* (CCA) ermittelt wird.
- ...ob der NAV gesetzt ist.
- ...ob der eigene Knoten noch am Senden ist.

Die Kommunikation zwischen den Prozessen der Komponente, um diese Prüfung durchzuführen, ist in Abbildung 4.7 zu sehen. Die recvOnly-Phase ist so realisiert, dass der zuständige Prozess *PrioTxRx* die Zustände `on` und `off` besitzt, und nur im `on`-Zustand sowohl senden als auch empfangen kann, während er im `off`-Zustand nur empfängt. Findet also ein Wechsel zur Receive-Only-Phase statt, so erhält der Prozess das Deaktivierungssignal Disable. Er fragt mit dem Signal TxStatusRequest beim Prozess *FrameTx* nach, ob ein etwaiger Sendevorgang noch nicht abgeschlosen ist. Die Signale NavStatusRequest und CcaBehaviorRequest an den Prozess *NAV* erfragen, ob der Network Allocation Vector gesetzt ist bzw. ob Sendebetrieb im

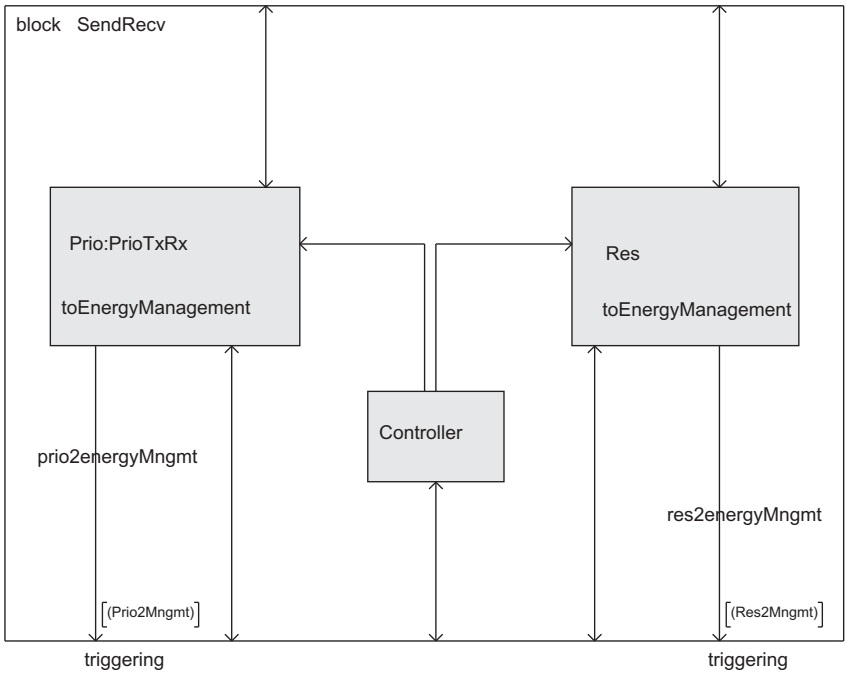

Abbildung 4.6: Die Komponente *SendRecv* verfügt über zwei Komponenten, die beide sowohl Sende- als auch Empfangsfunktionen erfüllen, jedoch zu unterschiedlichen Zeiten eingesetzt werden.

Medium herrscht. Im abgebildeten Fall findet beim Übergang kein Sendevorgang statt, der NAV ist nicht gesetzt und das Medium blieb für die Dauer des *Guard*-Intervalls unbelegt. Da daraus geschlossen werden kann, dass kein Paket mehr ein- oder ausgehen wird, meldet der verantwortliche Prozess *PrioTxRx* das Ende des Ereignisses *potential transfer* an das EnergyManagement. Ein anderer Fall ist in Abbildung 4.8 dargestellt. Der Prozess `FrameTx` meldet nach der Anfrage, dass noch ein Sendevorgang stattfindet. Er vermerkt über das Flag `reportWhenDone`, dass er den Abschluss des Sendevorganges zu melden hat. Der *PrioRxTx*-Prozess wartet diese Meldung ab, um im Anschluss dem EnergyManagement das Ende des Ereignisses *Potential Transfer* zu signalisieren.

Trifft keiner der zuvor genannten Fälle zu, so wird das EnergyManagement auch ohne Erhalt eines Signals PotentialTransferPassed nach Ablauf der Receive-Only-Phase das Ende des Ereignisses *Potential Transfer* erkennen.

Die Res-Komponente Die Res-Komponente ist während einer Res-Region aktiv. Während dieser Region ist das EnergyManagement im Zustand planned_transmission oder planned_reception, je nachdem ob gesendet oder empfangen werden soll. Sie besteht nur aus einem Prozess *ResTxRx* , der sowohl für den Versand als auch den Empfang während der ReservedSlot-Region verantwortlich ist. Da der Abschluss eines geplanten Empfangs oder Versands für das EnergyManagement von Bedeutung ist, wurde der Prozess um diese Meldungen erweitert. Abbildung 4.9 zeigt den Prozess *ResTxRx*, der beim Beginn eines Slots, für den ein Versand geplant ist, von der Paketpuffer-Komponente das zu sendende Paket anfordert, versendet

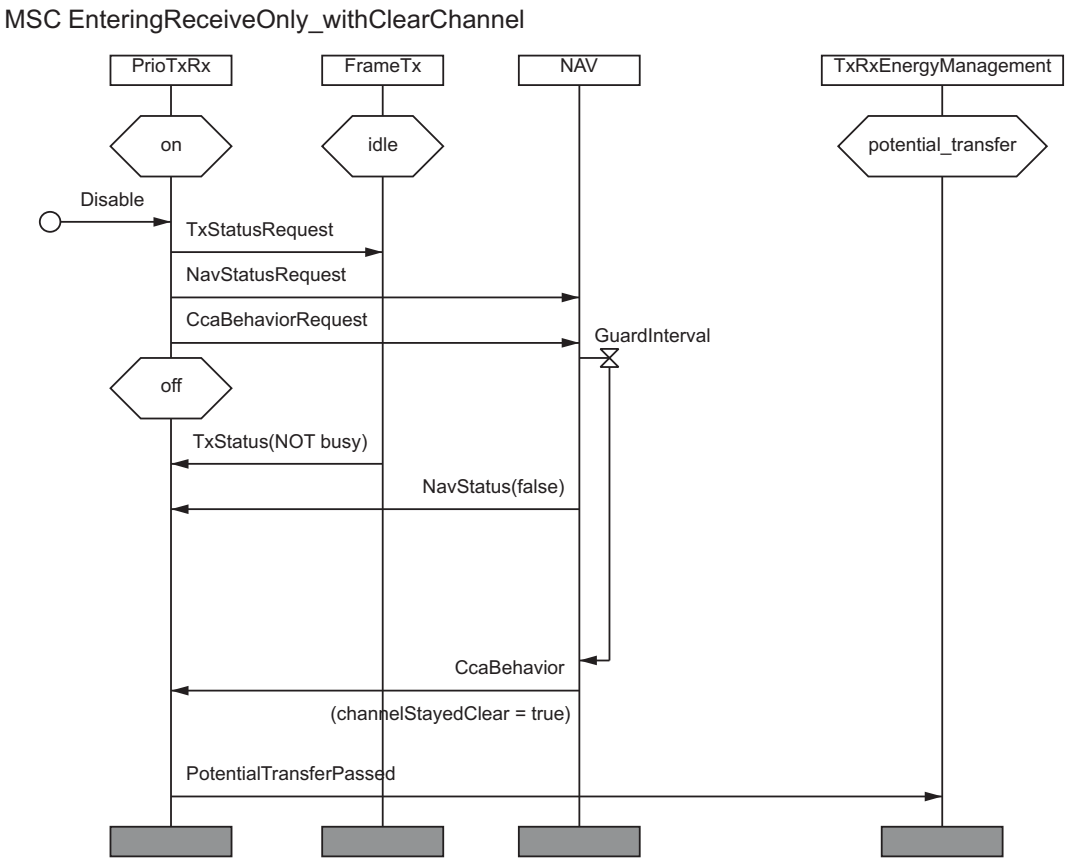

Abbildung 4.7: Signalaustausch beim Übergang in die *Receive-Only*-Phase falls der Knoten an keiner Übertragung teilnimmt und auch keine fremde Übertragung stattfindet. Sobald die Situation vom Prozess *PrioTxRx* erkannt wurde, meldet er das Ende des Ereignisses *Potential Transfer*.

und nach der Bestätigung das Ende des Ereignisses *Planned Transmission* an das EnergyManagement meldet.

Antwortet die Paketpuffer-Komponente, dass kein Paket für diesen Slot bereitliegt, so meldet der *ResTxRx*-Prozess sofort das Ende des Ereignisses an das EnergyManagement. Ob das Paket erfolgreich versandt oder nicht rechtzeitig von der Applikation bereitgestellt wurde, ist für das EnergyManagement nicht von Bedeutung – in beiden Fällen ist das Ereignis vorrüber. Das Ende des Ereignisses *Planned Reception* wird von der Komponente mit dem Signal PlannedReceptionPassed gemeldet, sobald ein Paket empfangen wurde. Findet der Empfang nicht wie geplant statt, so merkt dies das EnergyManagement nach Ablauf der dafür vorgesehenen Zeit ohne eine weitere Meldung. Hier ist noch Verbesserungspotential vorhanden, da durch ein *Guard-Intervall* das Scheitern des geplanten Empfangs schon früher festgestellt werden kann. Dazu ist es nötig, eine Obergrenze für die mögliche Empfangsverzögerung festzulegen. Sie muss den maximal möglichen Uhrenversatz, die Rechenzeit durch interne Abarbeitung des Protokolls auf Versenderseite und die zum Erkennen eines belegten Mediums auf Empfängerseite nötige Dauer enthalten.

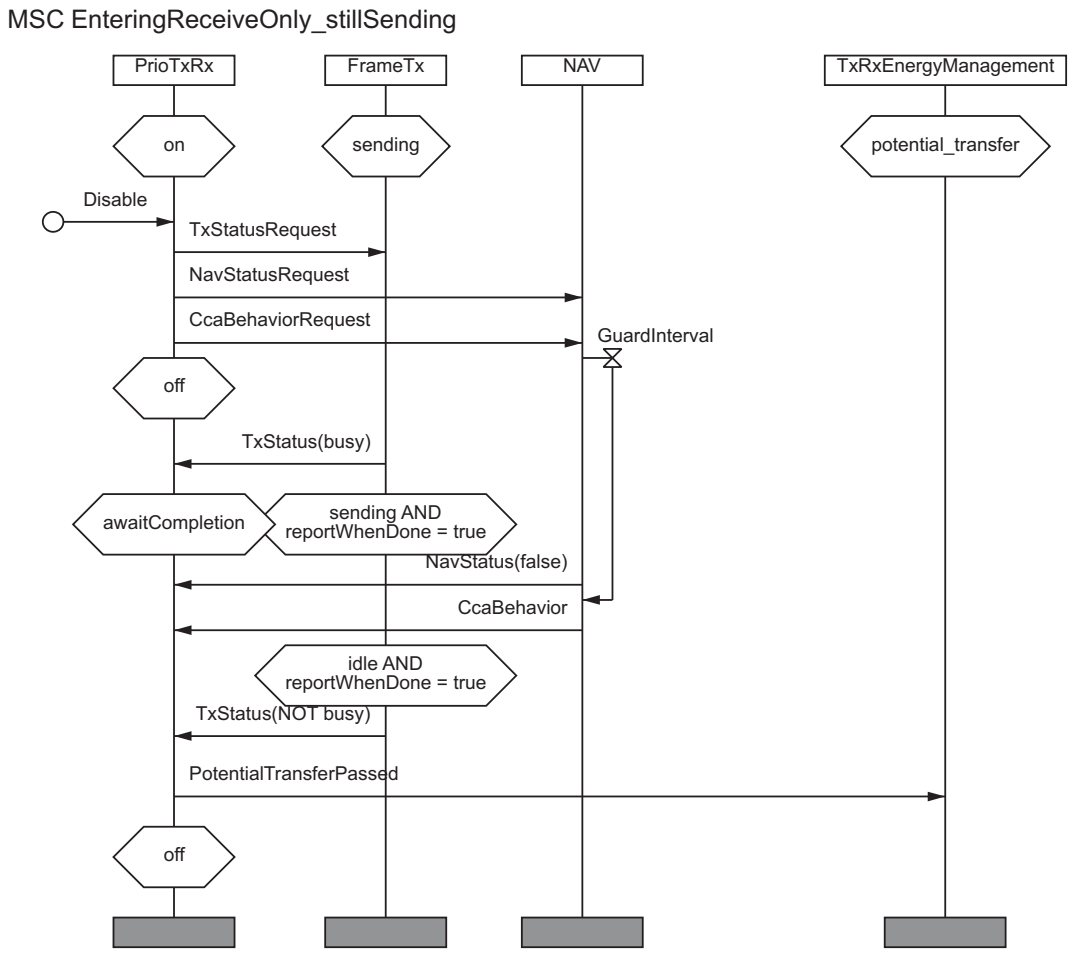

Abbildung 4.8: Signalaustausch beim Übergang in die *Receive-Only*-Phase bei einer laufenden Übertragung des Knotens. Der Prozess *PrioTxRx* wartet das Ende der Übertragung ab, um das Ende des Ereignisses *Potential Transfer* zu melden.

Das EnergyManagement Das EnergyManagement besteht aus einem einzigen Prozess, der energierelevante Meldungen entgegennimmt und den Energiezustand der Transceiverhardware danach steuert. Er betreibt das in **??** beschriebene *transceiver operation signaling*. Dabei wird der Transceiver-Zustand über SDL-Signale gesteuert, die von der Laufzeitumgebung in entsprechende Befehle an den Transceiver-Treiber umgesetzt werden. Der Prozess hält in einer speziellen Datenstruktur den weiter oben beschriebenen Schedule, um die für einen Makroslot festgelegten Ereignisse zu kennen.

Die Realisierung des Schedules und der in ihm vermerkten Ereignisse in SDL geschieht durch ein Feld, das mit Instanzen eines eigenen Ereignis-Datentyps befüllt ist. Aufeinanderfolgende Ereignisse im Schedule belegen benachbarte Zellen im Feld – für die Anordnung der Ereignisse im Feld ist daher nur ihre Reihenfolge ausschlaggebend, nicht der zeitliche Abstand. Es gibt daher keine leeren Zellen im Feld, um etwaige Passiv-Phasen zu repräsentieren. Ein Ereignis belegt immer genau eine Zelle des Feldes, unabhängig von seiner Dauer. Die Datentyp-Deklaration des Schedules ist in Listing 4.1 zu sehen. Danach ist ein Ereignis durch folgende Attribute definiert:

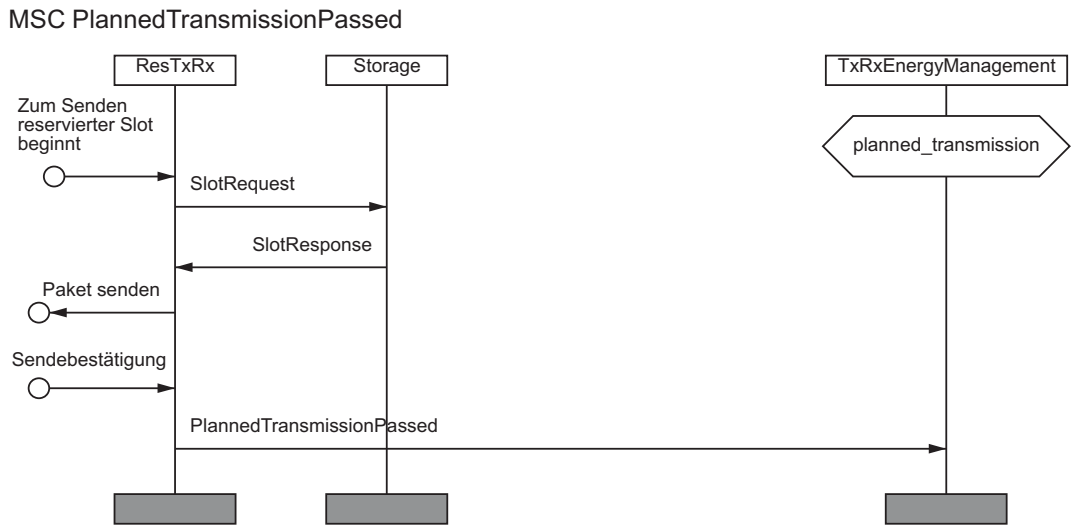

Abbildung 4.9: Planmäßiger Versand während eines reservierten Slots. Der Prozess *ResTxRx* meldet nach dem Erhalt der Sendebestätigung das Ende des Ereignisses *Planned Transmission*.

`purpose`:	Zweck der Transceiver-Aktivierung, mögliche Werte sind *Potential Transfer*, *Planned Transmission* und *Planned Reception*.
`openingSlot`:	Slotnummer des Eintretens des Ereignisses.
`durationInSlots`:	Maximale Dauer des Ereignisses gemessen in Slots. Ist der Zweck *Planned Transmission*, wird keine Dauer angegeben.

Zu sehen ist ebenfalls eine Variable `nextEvent`, sie verweist während der Laufzeit immer auf das nächste der eingetragenen Ereignisse. Der Prozess erhält über den Service-Layer-Multiplexer den Takt in Form von Signalen, die ihm den Beginn und die Nummer eines Mikroslots sowie die Makroslotgrenzen mitteilen.

Da der Wechsel des Transceiverzustandes nicht augenblicklich erfolgt, sondern eine bestimmte Zeitdauer in Anspruch nimmt, muss das EnergyManagement rechtzeitig vor einem festgelegten Ereignis einen Wechsel einleiten. Der Prozess kennt die Slotnummer, zu denen er einen Wechsel des *Transceiver*-Zustands vorbereiten muss. Er hat diese Slotnummern aus dem Schedule und den bekannten Verzögerungen, die der Transceiver für einen Zustandswechsel braucht, berechnet. Da die Zeiten zum Anstoß eines Zustandwechsels jedoch in der Regel nicht exakt mit dem Beginn eines Mikroslots zusammenfallen, sondern inmitten eines solchen Slots liegen können, bedient sich das EnergyManagement parametrisierter Timer. Die Dauer eines Microslots sei t_{micro}. Liegt ein Mikroslot, wie in Abb. 4.10 der mit x markierte, in einem Abstand von n Slots vor einem Ereignis und gilt für die Dauer $t_{powerdown2receive}$ zum Einschalten des Transceivers (aus dem PowerDown- in den Receive-Zustand)

$$n \cdot t_{micro} \geq t_{powerdown2receive} > (n-1) \cdot t_{micro},$$

dann setzt der Prozess einen *WakeUp*-Timer (mit dem Parameter *receive*) auf eine Dauer

$$(n \cdot t_{micro} - t_{powerdown2receive}).$$

```
1  /* enum datatype to classify the events that can occur in a schedule */
2  NEWTYPE Purpose
3  LITERALS planned_transmission, planned_reception, potential_transfer;
4  ENDNEWTYPE;
5
6  /* data structure to hold all needed information
7   * on a concrete scheduled event
8   */
9  NEWTYPE Event STRUCT
10 purpose Purpose;
11 openingSlot Integer; /* usually first slot of a region,
12                       * but could be any one.
13                       * for example in the case of
14                       * multiple planned_transmissions
15                       * in consecutive slots inside the
16                       * same reserved region
17                       */
18 durationInSlots Natural; /* only needed if purpose is
19                           * 'planned_reception' or
20                           * 'potential_transfer', the field's value
21                           * will therefore be ignored if purpose
22                           * is 'planned_transmission'
23                           */
24 ENDNEWTYPE;
25
26 /* data structure to hold the node-specific duty cycle schedule */
27 NEWTYPE EventMap
28 Array(Integer,Event)
29 ENDNEWTYPE;
30
31 /* used (node-specific) duty cycle schedule */
32 DCL schedule EventMap;
33
34 /* pointer (actually just an index) to the next 'Event' entry in an
35 EventMap during runtime, that determines whether the node saves energy
36 e.g. by switching the transceiver off or whether it has to be awake */
37 DCL nextEvent Integer := -1;
```

Listing 4.1: Deklaration der Datenstruktur *Schedule* in SDL.

Läuft der Timer ab, bestimmt sein Parameter, in welchen Zustand der Transceiver zu schalten ist. Der Zeitpunkt wurde dann so berechnet, dass der Transceiver exakt zum Beginn des Ereignisses empfangsbereit ist. Die Umsetzung in SDL ist anhand eines Ausschnittes aus der SDL-Spezifikation in Abb. 4.11 zu sehen. Ergibt die Prüfung, dass es sich um einen `wakeUpSlot` handelt (obere Transition), wird ein Timer gesetzt. Feuert der Timer (untere Transition) wird ein Signal an den Transceiver gesendet.

Die benötigten Zeiten für einen Zustandswechsel stammen aus **??**. Der folgende Pseudo-Kode-Block (Listing 4.2) zeigt den Entscheidungsalgorithmus, der den nächsten Energiezustand des Transceivers festlegt, nachdem das vorhergehende Ereignis beendet ist.

Sobald das vorherige Ereignis zu Ende ist, wird das nächste Ereignis aus dem Schedule gelesen, sein zeitlicher Anfang berechnet und daraus berechnet, wieviel Zeit

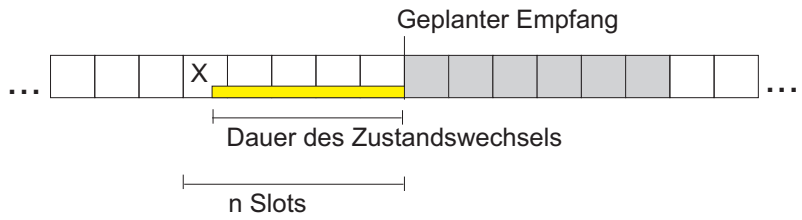

Abbildung 4.10: Schematische Darstellung der Lage des *WakeUpSlots* vor einem Ereignis.

```
1  if (Ereignis.beendet == true) {
2      nextEvent++; // aktualisieren des Pointers,
3                   // auf den getNextEventOccurrence() zugreift
4      nextEventOccurrence = getNextEventOccurrence();
5      potentialSleepingPeriod = nextEventOccurrence - now;
6      nextPurpose = schedule[nextEvent].purpose;
7      switch (nextPurpose) {
8          case 'planned_reception':
9              endingSlot = calculateEndingSlot();
10             if (potentialSleepingPeriod > powerDown2receive_delay) {
11                 changeTxRxState(powerDown);
12                 setWakeUpTimer();
13             } else if (potentialSleepingPeriod > idle2receive_delay) {
14                 changeTxRxState(idle);
15                 setWakeUpTimer();
16             } else {
17                 changeTxRxState(receive);
18             }
19         case 'planned_transmission': // ähnlich zu planned_reception
20         case 'potential_transfer': // analog zu planned_reception
21     }
22 }
```

Listing 4.2: Algorithmus in Pseudo-Kode.

als potenzielle Schlafdauer `potentialSleepingPeriod` zur Verfügung steht. Sein Zweck bestimmt die weitere Prüfung, ob die potenzielle Schlafdauer genutzt werden kann. Denn abhängig davon, ob das nächste Ereignis ein Empfang sein kann (im Falle von potential_transfer und planned_reception) oder ob es ein Versand ist (im Falle von planned_transmission), ist für dieses Ereignis Empfangsbereitschaft herzustellen (Transceiver-Zustand *receive*) oder nur der Transceiver-Zustand *idle* nötig (da im Falle eines stattfindenden Versands die Umschaltdauer von *idle* nach *send* genausolange dauert wie von *receive* nach *send*).

Da planned_reception eine Längenangabe in Form von Slots hat, wird der `endingSlot` berechnet, bei dessen Erreichen das Ereignis spätestens als beendet erkannt wird. Im Anschluss wird geprüft, ob die Zeit für eine Abschaltung des Transceivers reicht (powerDown) oder für einen StandBy-Betrieb (idle). Ist keines der Fall und die potezielle Schlafdauer bis zum nächsten Ereignis zu kurz, dann muss trozdem sichergestellt sein, dass der Transceiver empfangsbereit ist (Zustand *receive*), denn der vorherige Zustand könnte *idle* gewesen sein. Ein `wakeUp`-Timer,

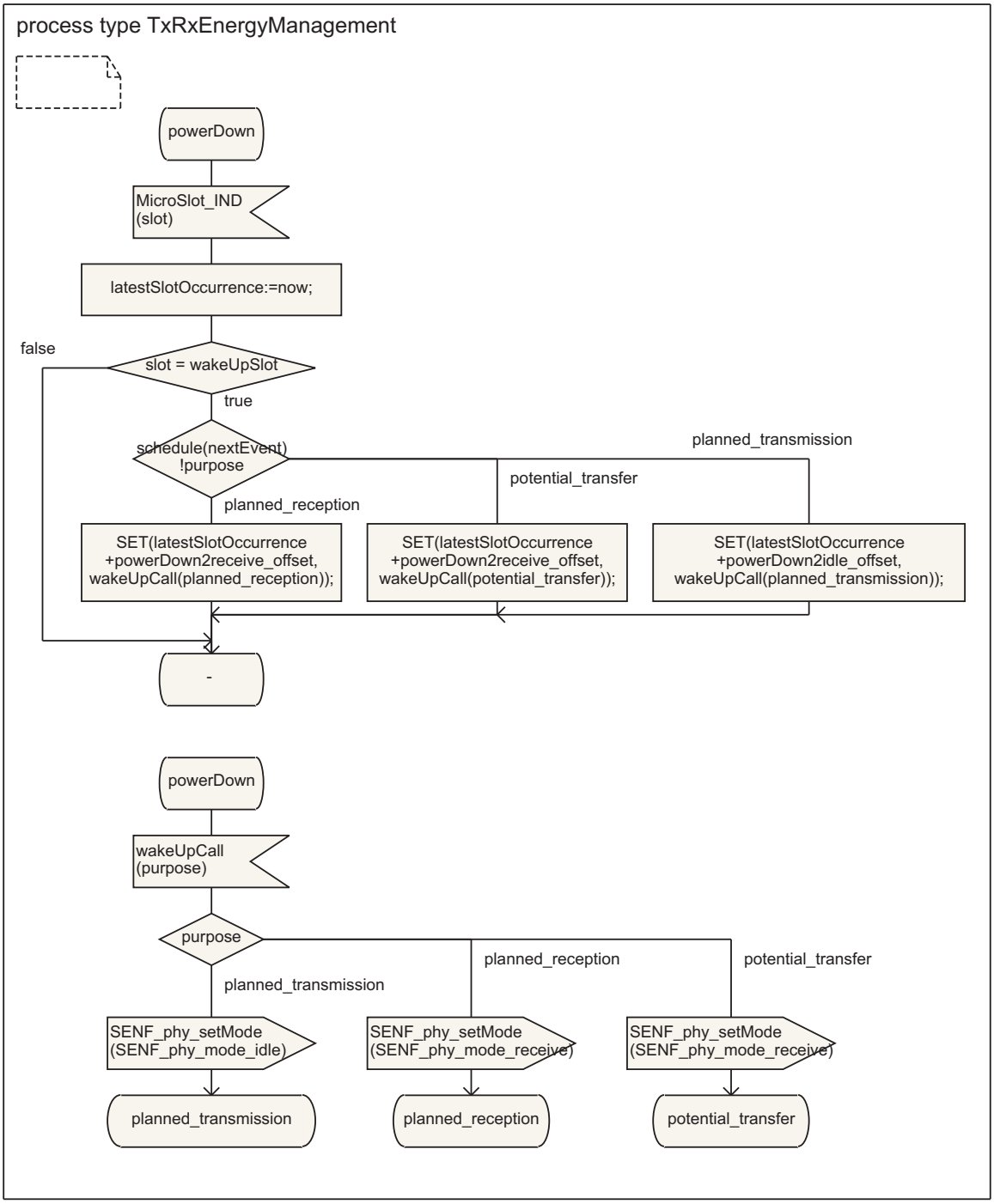

Abbildung 4.11: Umsetzung der Prüfung, ob ein *WakeUpSlot* vorliegt, und Setzen und Auslösen des *WakeUpTimers* in SDL.

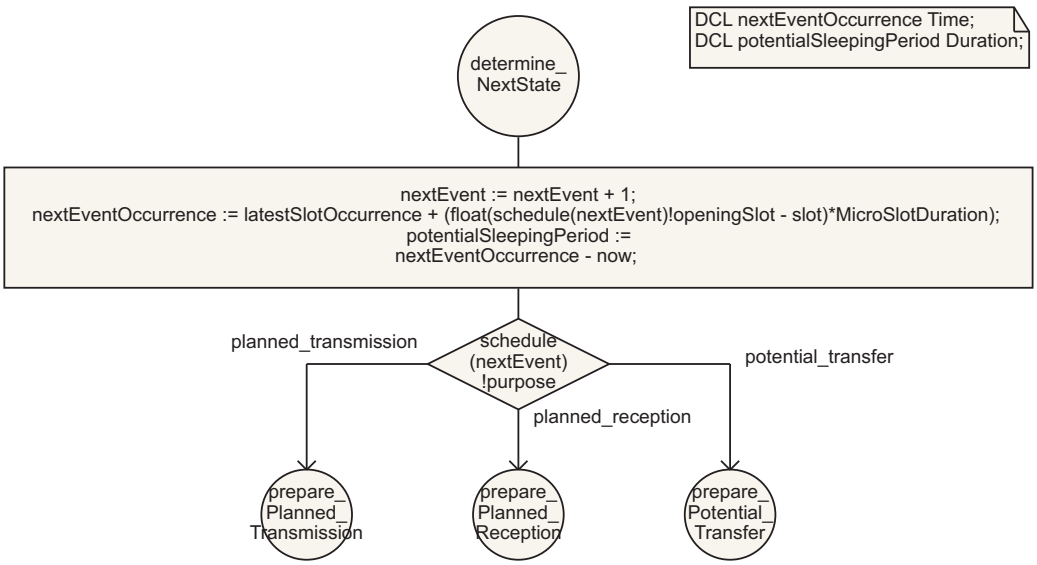

Abbildung 4.12: Umsetzung des Pseudo-Kode-Algorithmus in SDL: Bestimmung der potenziellen Schlafdauer.

wie weiter oben beschrieben, stellt sicher, dass der Empfangszustand rechtzeitig hergestellt wird. Die Umsetzung des Pseudo-Kodes in SDL zeigen die Abbildungen 4.12 und 4.13.

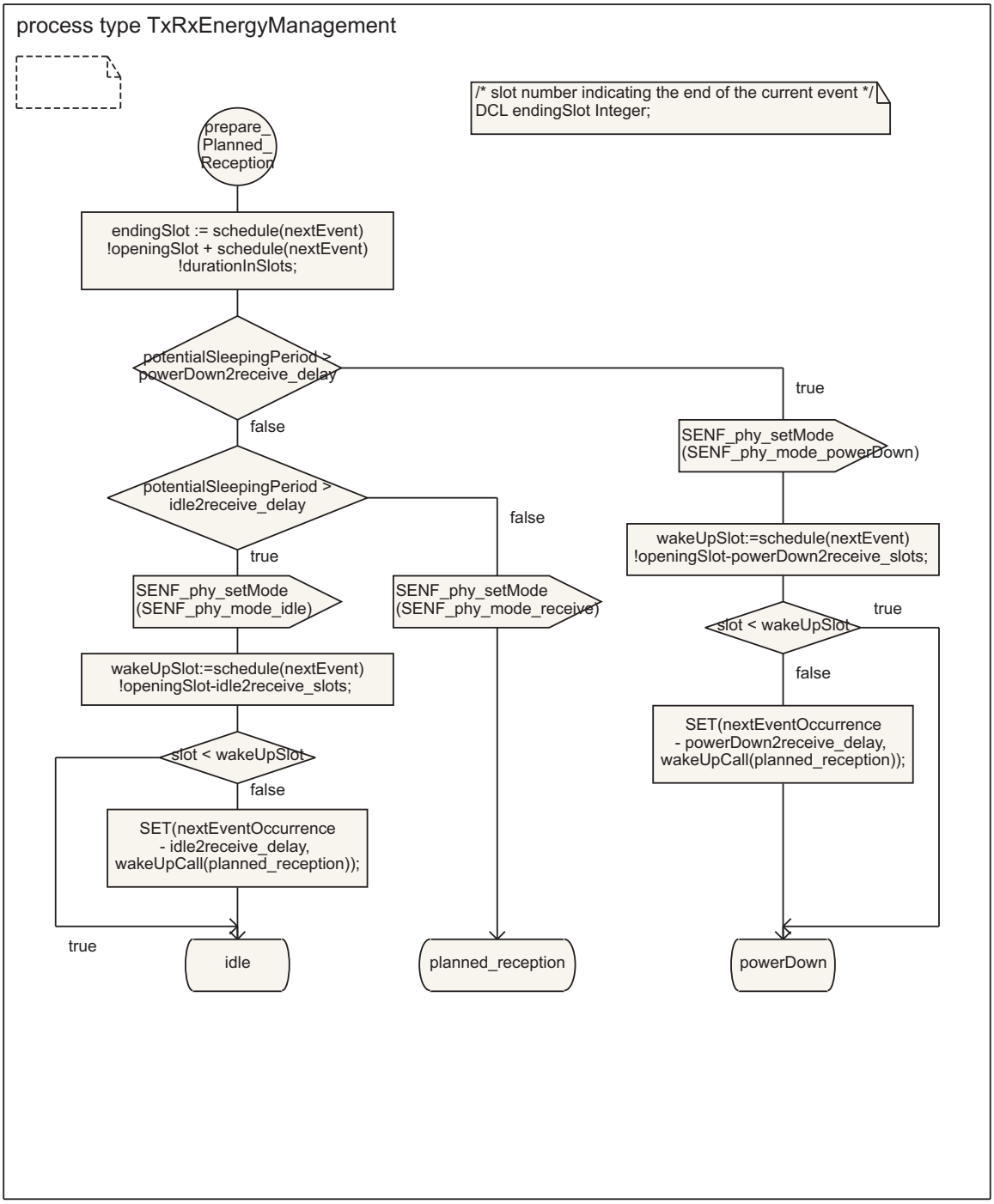

Abbildung 4.13: Umsetzung des Pseudo-Kode-Algorithmus in SDL: Entscheidung über die Nutzung der potenziellen Schlafdauer.

4.3 Simulation

Zur Evaluation der entwickelten Konzepte wird der Netzwerksimulator NS2 eingesetzt. Die Evaluation soll zum Einen zeigen, ob die Kommunikation zwischen verschiedenen Knoten stattfinden kann, und zum Anderen ob das in MacZ integrierte Duty-Cycling-Verfahren den Energieverbrauch stärker reduziert als vergleichbare Protokolle, wie sie in Kapitel 2 vorgestellt wurden. Im Rahmen der Arbeit wurde der erste Punkt an einem Beispiel gezeigt. Der Vergleich mit anderen Protokollen ist aus Zeitgründen entfallen.

Für die Simulation wurden zwei Test-Applikationen geschrieben. Eine stellt den Dienst *dummyService* über die WNCS-CoM zur Verfügung, während die andere diesen Dienst abonniert. Der Quell-Kode beider Applikationen ist auf der CD enthalten.

Der Simulationslauf benötigt eine Beschreibung des Szenarios, die unter anderem die Art des Netzes, die Verteilung der Knoten und deren Startreihenfolge festlegt. Diese Beschreibung wird dem Simulator in Form einer Konfigurationsdatei beim Simulationsstart als Parameter übergeben. Die wichtigsten Punkte des verwendeten Szenarios sind folgende:

- Der simulierte Transceiver ist CC2420, somit findet die Kommunikation drahtlos statt.
- Das Netz besteht aus zwei Knoten: ein Dienstanbieter, ein Dienstnutzer.
- Diese sind 25 Meter von einander entfernt.

Die vollständige Konfigurationsdatei `simulation.tcl` ist auf der CD enthalten.

Macroslot-Konfiguration Die Macroslot-Konfiguartion sah eine wettbewerbsbasierte Slot-Region vor, die sich über den gesamten Macroslot erstreckte. Damit wurde auf die Verwendung von reservierten Slots verzichtet. Denn deren Nutzung hätte eine Synchronisation des Datenversands durch die Applikation mit den reservierten Slots erfordert, was noch nicht realisiert wurde. Entsprechend war im Energy-Management-Schedule nur ein Ereignis mit dem Zweck *Potenzielle Übertragung* (potential_transfer) und der Dauer des Macroslots eingetragen.

Ablauf und Ergebnis Der Dienstanbieter registriert den Dienst *dummyService*, der regelmäßigen Datenversand beliebiger, Abonnenten-spezifischer Periode bietet. Der Nutzer abonniert diesen Dienst mit einem periodischem Datenversand für alle 30 ms. Der Dienst sendet 30 ms nach der Entgegennahme der Anmeldung dem Nutzer erstmals Daten.

Die Simulation zeigte, dass die Knoten kommunizieren können. Die Anmeldung des Dienstnutzers wird entgegengenommen und es werden periodisch Daten versandt. Es bleibt aber zu prüfen, ob auch eine restriktivere Macroslot-Konfiguration, die nur kurze Phasen für Versand und Empfang von Nachrichten enthält und ein entsprechender Energy-Management-Schedule, der den Transceiver zwischenzeitlich deaktiviert, die nötige Kommunikation ermöglicht.

Messung des Energieverbrauchs Die Messung des Energieverbrauch wurde noch nicht vorgenommen. Für diese Messung ist jeder Zustandswechsel des Transceivers mit Zeitpunkt und Folgezustand zu protokollieren. Die Protokollierung der Zustandswechsel während der Simulation kann am besten über Log-Meldungen erfolgen, da der Simulator die Führung je einer eigenen Log-Datei für jeden simulierten Knoten unterstützt. Die Anweisung zur Ausgabe dieser Log-Meldungen erfolgt am sinnvollsten im SDL-Prozess, der den CC2420-Treiber im SDL-System repräsentiert. Im spezifizierten System, das für die Simulation eingesetzt wurde, ist dies der Prozess *SENF-CC2420Driver*. Denn durch die Abarbeitung von Transitionen und Berechnungen im EnergyManagement-Prozess können Verzögerungen von der Anweisung zum Zustandswechsel bis zum tatsächlichen Zustandwechsel auftreten. Je näher der protokollierende Prozess daher an der Schnittstelle zum SDL-Environment-Framework (SEnF) [FGJ+05] liegt, desto geringer sind diese Verzögerungen und damit die Verfälschung der Messung. Es bietet außerdem den Vorteil, dass ein Simulationslauf mit einem anderen Duty-Cycling-Protokoll (um vergleichbare Ergebnisse zu erhalten) mit dem gleichen Mechanismus protokolliert werden kann, da der CC2420-Treiber-Prozess der gleiche ist. Zur besseren Auswertung erhalten die Log-Einträge über Energiezustandswechsel des Transceivers ein eindeutiges Präfix. Die Log-Datei kann dann automatisiert nach solchen Einträgen durchsucht werden. Aus zwei aufeinanderfolgenden Einträgen ergibt sich die Dauer, die der Transceiver in einem Zustand war. Für jeden möglichen Zustand ist die Summe aus allen Zeiträumen zu bilden, die der Transceiver im betreffenden Zustand verbrachte. Die Energieverbräuche des Transceivers in den verschiedenen Zuständen können aus dessen Datenblatt entnommen werden. Mit diesen Angaben lässt sich der Energieverbrauch des Knotens – nur bezogen auf den Tranceiver, ohne CPU – berechnen.

Wird der gleiche Simulationslauf mit einem Vergleichs-Protokoll durchgeführt und Zustandwechsel ebenfalls protokolliert, so ist ein direkter Vergleich der Energieverbräuche möglich.

Die Spezifizierung von Vergleichs-Protokollen ist für die weiterführende Arbeit vorgesehen.

Kapitel 5

Zusammenfassung und Ausblick

In der Arbeit wurde mit der WNCS-CoM eine leistungsstarke Middleware für drahtlos vernetzte Kontrollsysteme (WNCS) vorgestellt. Sie bietet eine einfach zu handhabende Grundlage für die Registrierung von Software-Diensten und deren Nutzung durch Abonnenten. Sowohl der Dienstanbieter, als auch der Nutzer benötigt dazu keine weitere Kenntnis über das Netz. Die Adressierung eines Dienstes erfolgt über seinen netzweit eindeutigen Namen. Beim Abonnement eines Dienstes kann der Abonnent über einen Parameter wählen, ob er ohne weitere Anforderungen regelmäßig Daten des Dienstes erhalten will. Die Synchronisation des Datenversandes an verschiedene Abonnenten innerhalb eines Dienstanbieters ermöglicht energieeffizientes Arbeiten und größtmögliche Pausen zwischen den Datenanforderungen. Ein Duty-Cycling-Mechanismus, der die CPU in ungenutzten Phasen in einen energiesparenden Zustand versetzt, nutzt dieses Energiesparpotential. Ein entsprechender Mechanismus wurde bereits in [GKLC] vorgestellt.

Die vorgestellte Applikations-Schnittstelle, ermöglicht die Nutzung der WNCS-CoM auch ohne SDL-Kenntnisse. SDL-Spezifika wie spezielle Datentypen, deren Erzeugung und Semantik werden durch die Schnittstelle vor dem Nutzer verborgen, er bindet eine *Header*-Datei ein und kann die Standard-C/C++-Datentypen verwenden.

Um die Eignung des Systems für den Einsatz in Sensornetzen zu erhöhen, wurde ein Energy-Management in das verwendete MAC-Protokoll MacZ integriert. Die Sensoren solcher Netze sind in der Regel auf eine Energieversorgung aus einer Batterie angewiesen. Ist deren Energie aufgebraucht, fallen einzelne Knoten aus bis schließlich das gesamte Netz seine Aufgabe nicht mehr erfüllen kann. Das entwickelte EnergyManagement versetzt die Transceiver-Hardware der Knoten während ungenutzter Phasen in energiesparende Zustände. Sie verbrauchen ihre Energie dadurch langsamer und die erreichbare Laufzeit des Netzes wird erhöht. Den Wechsel zwischen energiesparenden Phasen, in denen die Hardware nicht nutzbar ist und Phasen der Nutzung nennt man *Duty-Cycling*. Duty-Cycling wird auch von anderen MAC-Protokollen betrieben. Als großer Vorteil gegenüber anderen Protokollen, die durch den Einsatz solcher Arbeitszyklen Energie sparen, kann in MacZ Anwendungswissen genutzt werden. Die Arbeitszyklen der Knoten sind individuell auf deren Aufgabe und Kommunikationsverhalten ausgerichtet, wodurch der Transceiver gezielter ein- und ausgeschaltet wird als in vergleichbaren Protokollen.Hierfür wurde die Fähigkeit

von MacZ, Slotregionen unterschiedlicher Nutzungsbedingungen zu bieten, genutzt. Zukünftig könnten weitere Hardwarekomponenten in das Duty-Cycling miteinbezogen werden. So kann ein vorhandener Sensor, den der Knoten betreibt und der unterschiedliche Energiezustände bietet, durch regelmäßige Phasen in energiesparenden Zuständen den Energieverbrauch des Knotens senken.

Die Simulationen konnten zeigen, dass das System die Kommunikation der Anwendung unterstützt. Jedoch sind weitere Simulationen mit einem veränderten Arbeitszyklus nötig, da der in der Simulation verwendete Arbeitszyklus keine Schlafphasen enthält.Das Hauptaugenmerk für weiterführende Arbeiten liegt auf der Spezifizierung eines Duty-Cycling-Protokolls und der vergleichenden Energieverbrauchsmessung beider Protokolle. Die Generierung von Nutzdaten in der Applikationsschicht ist zukünftig auf die Slotgrenzen der MAC-Schicht auszurichten, da nur so sichergestellt wird, dass die generierten Daten auch in den dafür vorgesehenen Slots versandt werden können. Bisher wurden nur wettbewerbsbasierte Slots genutzt, eine Nutzung der reservierten Slots verspricht daher eine Verringerung der durchschnittlichen Übertragungsverzögerung und eine zuverlässigere Übertragung.

Anhang A

Formale QoS-Spezifikationen

Im folgenden sind die formalen Dienstgüteanforderungen des Regelungssystems aufgeführt. Für jede Systemebene werden QoS-Domäne, Abbildungsfunktionen auf QoS-Werte anderer Ebenen und QoS-Anforderungen definiert.

A.1 Anwendungsebene

$$\text{QoS-Domäne} \quad Q_{Steuerung} = P_{Steuerung} \times R_{Steuerung} \times G_{Steuerung}$$

- Performance-Subdomäne $P_{Steuerung}$:

  $P_{Steuerung} = Signalintervall \times Totzeit$
  $Signalintervall = \mathbb{N}$ (Zeit zwischen zwei Steuersignalen in ms)
  $Totzeit = \mathbb{N}$ (Reaktionszeit des Systems in ms)

- Reliability-Subdomäne $R_{Steuerung}$:

  $R_{Steuerung} = Loss \times Period \times Burstiness \times Corruption$
  $Loss = \mathbb{N}_0$
  $Period = \mathbb{N}_0$
  $Burstiness = \mathbb{N}_0$
  $Corruption = \{c \in \mathbb{N}_0 | 0 \leq c < 100\}$

- Guarantee-Subdomäne $G_{Steuerung}$:

  $G_{Steuerung} = DoC \times Stat \times Priority$
  $DoC = \{bestEffort, enhancedBestEffort, statistical, deterministic\}$
  $Stat = \{p \in \mathbb{R} | 0 < p \leq 1\}$
  $Priority = \mathbb{N}$

QoS-Anforderungen *Pendelsteuerung*:

- Optimal Value
  - Performance:
    $Signalintervall = 20\,[ms]$,
    $Totzeit = 8\,[ms]$
  - Reliability:
    $Loss = 0\,[Steuersignale]$,
    $Period = 200\,[ms]$,
    $Burstiness = 0\,[Steuersignale]$,
    $Corruption = 0\,[\%]$
  - Guarantee:
    $DoC = enhancedBestEffort$
- Minimal Value
  - Performance:
    $Signalintervall = 30\,[ms]$,
    $Totzeit = 24\,[ms]$
  - Reliability:
    $Loss = 1\,[Steuersignal]$,
    $Period = 300\,[ms]$,
    $Burstiness \leq 1\,[Steuersignal]$,
    $Corruption = 0\,[\%]$
  - Guarantee:
    $DoC = enhancedBestEffort$
- Scalability Value
  - $Up = 0, 1$; $Down = 0, 1$
  - Utility Function:
    $u : Q_{Steuerung} \rightarrow [0..1]$
    $(p, r, g) = q \in Q_{Steuerung}$
    $u(q) = min\{u_P(p), u_R(r), u_G(g)\}$
    $(s, t) = p \in P_{Steuerung}$
    $u_P(p) = u_{Signalintervall}(s) \cdot u_{Totzeit}(t)$
    $u_{Signalintervall}(s) = \frac{35-s}{15}$
    $u_{Totzeit}(t) = \frac{32-t}{24}$

$$u_R(r) = \begin{cases} 0 & \text{falls} \quad \frac{Loss}{Period} > \frac{Loss_{min}}{Period_{min}} \vee Burstiness > 1 \vee Corruption > 0 \\ 1 & \text{sonst} \end{cases}$$

$$u_G(g) = \begin{cases} 0 & \text{falls} \quad g < enhancedBestEffort \\ 1 & \text{sonst} \end{cases}$$

A.2 Middleware-Ebene

QoS-Domäne $Q_{Middleware} = P_{Middleware} \times R_{Middleware} \times G_{Middleware}$

- Performance-Subdomäne $P_{Middleware}$:
  $P_{Middleware} = Sendeintervall \times Verzoegerung$
  $Sendeintervall = \mathbb{N}_0$
  $Verzoegerung = \mathbb{N}_0$
- Reliability-Subdomäne $R_{Middleware}$:
  $R_{Middleware} = Loss \times Period \times Burstiness \times Corruption$
  $Loss = \mathbb{N}_0$
  $Period = \mathbb{N}_0$
  $Burstiness = \mathbb{N}_0$
  $Corruption = \{c \in \mathbb{N}_0 | 0 \leq c < 100\}$
- Guarantee-Subdomäne $G_{Middleware}$:
  $G_{Middleware} = DoC \times Stat \times Priority$
  $DoC = \{bestEffort, enhancedBestEffort, statistical, deterministic\}$
  $Stat = \{p \in \mathbb{R} | 0 < p \leq 1\}$
  $Priority = \mathbb{N}$

Domain Mapping $dm : Q_{Steuerung} \rightarrow Q_{Middleware}$
$dm_P : P_{Steuerung} \rightarrow P_{Middleware}$
$dm_R : R_{Steuerung} \rightarrow R_{Middleware}$
$dm_G : G_{Steuerung} \rightarrow G_{Middleware}$

- Performance Mapping $dm_P(p) = (dm_{P_{Sendeintervall}}(p), dm_{P_{Verzoegerung}}(p))$
  $dm_{P_{Sendeintervall}}(p) = s$
  $dm_{P_{Verzoegerung}}(p) = \frac{t}{2} - 3$
- Reliability Mapping $dm_R(r) = (Loss, Period, Burstiness, Corruption)$
- Guarantee Mapping $dm_G(g) = enhancedBestEffort$

QoS-Anforderungen *Pendel-Middleware* (ergeben sich aus der Abbildungen der Anforderungen der Anwendungsebene in die Middleware-Ebene):

- Optimal Value
  - Performance:
    $Sendeintervall = 20\,[ms]$,
    $Verzoegerung = 1\,[ms]$
  - Reliability:
    $Loss = 0\,[Nachrichten]$,
    $Period = 200\,[ms]$,
    $Burstiness = 0\,[Nachrichten]$,
    $Corruption = 0\,[\%]$

  - Guarantee:
    $DoC = enhancedBestEffort$

- Minimal Value
  - Performance:
    $Sendeintervall = 30\,[ms]$,
    $Verzoegerung = 9\,[ms]$
  - Reliability:
    $Loss = 1\,[Nachricht]$,
    $Period = 300\,[ms]$,
    $Burstiness \leq 1\,[Nachricht]$,
    $Corruption = 0\,[\%]$
  - Guarantee:
    $DoC = enhancedBestEffort$

A.3 MAC-Ebene

QoS-Domäne $Q_{MAC} = P_{MAC} \times R_{MAC} \times G_{MAC}$

- Performance-Subdomäne P_{MAC}:
  $P_{MAC} = Framerate \times Delay$
  $Framerate = \mathbb{R}^+$
  $Delay = \mathbb{N}_0$

- Reliability-Subdomäne R_{MAC}:
  $R_{MAC} = Loss \times Period \times Burstiness \times Corruption$
  $Loss = \mathbb{N}_0$
  $Period = \mathbb{N}_0$
  $Burstiness = \mathbb{N}_0$
  $Corruption = \{c \in \mathbb{N}_0 | 0 \leq c < 100\}$

- Guarantee-Subdomäne G_{MAC}:
  $G_{MAC} = DoC \times Stat \times Priority$
  $DoC = \{bestEffort, enhancedBestEffort, statistical, deterministic\}$
  $Stat = \{p \in \mathbb{R} | 0 < p \leq 1\}$
  $Priority = \mathbb{N}$

Domain Mapping $dm : Q_{Middleware} \rightarrow Q_{MAC}$
$dm_P : P_{Middleware} \rightarrow P_{MAC}$
$dm_R : R_{Middleware} \rightarrow R_{MAC}$
$dm_G : G_{Middleware} \rightarrow G_{MAC}$

- Parameter:
  Länge einer Middleware-Nachricht in Bit $= MsgLength$
  Maximale Länge des Payloads eines MAC-Rahmens in Bit $= MaxLength$
  Anzahl MAC-Rahmen pro Middleware-Nachricht $= \lceil \frac{MsgLength}{MaxLength} \rceil$

- Performance Mapping $dm_P(p) = (dm_{P_{Framerate}}(p), dm_{P_{Delay}}(p))$
  $(s, v) = p \in P_{Middleware}$
  $dm_{P_{Framerate}}(p) = \frac{\lceil \frac{MsgLength}{MaxLength} \rceil}{s}$
  $dm_{P_{Delay}}(p) = v$

- Reliability Mapping $dm_R(r) = (Loss, Period, Burstiness, Corruption)$
- Guarantee Mapping $dm_G(g) = enhancedBestEffort$

QoS-Anforderungen *Pendel-MAC* (ergeben sich aus der Abbildungen der Anforderungen der Middleware-Ebene in die MAC-Ebene):

- Optimal Value
  - Performance:
    $Framerate = 50\,[\frac{Rahmen}{Sekunde}]$,
    $Delay = 1\,[ms]$
  - Reliability:
    $Loss = 0\,[Rahmen]$,
    $Period = 200\,[ms]$,
    $Burstiness = 0\,[Rahmen]$,
    $Corruption = 0\,[\%]$
  - Guarantee:
    $DoC = enhancedBestEffort$
- Minimal Value
  - Performance:
    $Framerate = 33,3\,[\frac{Rahmen}{Sekunde}]$,
    $Delay = 9\,[ms]$
  - Reliability:
    $Loss = 1\,[Rahmen]$,
    $Period = 300\,[ms]$,
    $Burstiness = \leq 1\,[Rahmen]$,
    $Corruption = 0\,[\%]$
  - Guarantee:
    $DoC = enhancedBestEffort$

Anhang B

Funktionen der Anwendungs-Schnittstelle

B.1 Middleware-zu-Applikation- oder Hardware-zu-Applikation-Kommunikation

- `void Init()`
  Dies ist eine Funktion zur Initialisierung der Anwendung. Hier können Perioden für Tasks festgelegt und Datenstrukturen befüllt werden. Die Funktion wird einmalig zu Systemstart vom SDL-Prozess aufgerufen. Der Aufruf erfolgt bevor irgendeine andere Funktion der Anwendung vom Prozess aufgerufen wird. Werden hier keine Perioden für Tasks gesetzt, bedeutet das, dass sich das System reaktiv verhält und darauf wartet, dass ein Empfangs-Ereignis über UART, Transceiver oder IO-Interrupt eintritt.

- `void Execute(int no)`
  Läuft der Timer für eine Task ab, so wird diese Funktion mit der Nummer der Task aufgerufen. Da der SDL-Prozess diese Timer verwaltet, löst er auch die Ausführung aus.

- `void Receive(int serviceID, const char* data, unsigned int length)`
  Wurden Daten von anderen Knoten im System empfangen, ruft der SDL-Prozess diese Funktion auf, um sie an die Anwendung zu übergeben. Der Parameter *serviceID* gibt den Dienst an, von dem die Daten stammen oder für den die Daten bestimmt sind. Das hängt davon ab, ob die Applikation den betreffenden Dienst Registriert oder Abonniert hat. Sendet ein Abonnent an einen Dienst, so wird `serviceID` den Dienst identifizieren für den die Daten bestimmt sind. Dies ist nötig, da eine Anwendung mehrere Dienste registrieren kann, und Anfragen an diese Dienste möglicherweise unterschiedlich verarbeitet. Der Parameter *data* enthält die Daten, deren Länge wird durch *length* angezeigt.

- `void requestDataForService(const int serviceID)`
  Liegt eine Daten-Anforderung eines Abonnenten vor, ruft der SDL-Prozess

diese Funktion auf. Der Aufruf ist nur für den Fall, dass der Knoten einen entsprechenden Dienst anbietet vorgesehen, ein Knoten, der nur Abonnent ist, muss diese Funktion daher nicht implementieren. Der Parameter *serviceID* zeigt an, für welcher konkreten Dienst Daten liefern muss. Dies ist nötig, da eine Anwendung mehrere Dienste registrieren kann, und Anfragen an diese Dienste möglicherweise unterschiedlich verarbeitet. Die Funktion besitzt keinen Rückgabewert, da die Zusammenstellung der Daten, z.B. das Ermitteln eines Messwertes, einige Zeit dauern kann. Die Anwendung muss sicherstellen, dass nach dem Aufruf dieser Funktion ein entsprechender Rückruf einer Funktion des SDL-Prozesses stattfindet, bei dem ihm die angeforderten Daten übergeben werden. Die entsprechende *Callback*-Funktion der SDL-Prozess-Schnittstelle ist *setRequestedData*.

- `void ReceiveUart(unsigned int port, const char* data, unsigned int length)`
  Über diese Funktion kann die Anwendung Daten über UART empfangen. Sie ist nicht zur Nutzung in der Endanwendung vorgesehen, sondern nur zu Debug-Zwecken. Der Parameter *port* gibt die Port-Nummer an (0,1 oder 2), *data* enthält die Daten und *length* gibt deren Länge an.

- `void ReceiveSpi(unsigned int port, unsigned int data)`
  Über diese Funktion kann die Anwendung Daten über SPI empfangen. Aber erst nach einem Aufruf von *SpiSend* der SDL-Prozess-Schnittstelle. Der Parameter *port* gibt die Port-Nummer an, *data* enthält die Daten. Eine Längenangabe ist hierbei nicht nötig, da der verwendete Datentyp eine feste Länge hat.

- `void IoInterruptCaptured(int no)`
  Eine Applikation kann den Wunsch, über bestimmte Interrupts informiert zu werden vorher mittels *IoSetInterrupt* der SDL-Prozess-Schnittstelle registrieren. Tritt ein solcher registrierter Interrupt auf, wird diese Funktion aufgerufen und die Nummer, die den Typ des Interrupts angibt (um zwischen unterschiedlichen Interrupts unterscheiden zu können), übergeben. Denn die Applikation kann Interesse für verschiedene Interrupt-Typen gleichzeitig angemeldet haben.

B.2 Applikation-zu-Hardware- oder Applikation-zu-Middleware-Kommunikation

- `void SetPeriod(int no, unsigned int ms)`

`no`:	Nummer der Task
`ms`:	neue Periode in Millisekunden

  Die Funktion setzt die Ausführungszeitabstände für eine *Task*. Der Parameter `no` gibt die Task-Nummer an, der Parameter `ms` das Zeitintervall ganzzahlig in Millisekunden. Wird die Periode einer Task auf Null gesetzt, bedeutet das die Beendigung der periodischen Ausführung. Der Aufruf beendet einen eventuell schon laufenden Timer. Wird dabei die Null als Wert für `ms` übergeben, wird kein weiterer Timer aufgezogen. Andernfalls wird ein Timer mit dem neuen Intervall angelegt. Das Herunterzählen der Millisekunden bis zum Auslösen des Timers beginnt mit dem Zeitpunkt des Funktionsaufrufes. Es spielt also keine Rolle wie weit fortgeschritten das Herunterzählen des vorherigen Timers zu diesem Zeitpunkt war.

- `void Send(const int serviceID, const char* data, unsigned int length)`

`serviceID`:	Dienstkontext für den Datenversand
`data`:	zu sendende Nutzdaten
`length`:	Länge der Nutzdaten

  Die Funktion versendet die übergebenen Daten an alle Knoten, die mit dem über `serviceID` identifizierten Dienst verbunden sind. Ein Knoten ist mit einem Dienst verbunden, wenn er ihn registriert oder abonniert hat. Ein Aufruf der Funktion führt also zu einem Multicast an alle Beteiligten. Der Datenversand findet im Kontext eines konkreten Dienstes statt. Die Empfänger müssen selbst entscheiden, ob sie die übertragenen Daten benötigen oder nicht. Da der Parameter `data` Nutzdaten variabler Länge enthält, muss deren Länge mit angegeben werden.

- `void SendUart(unsigned int port, const char* data, unsigned int length)`

`port`:	UART-Port für den Datenversand, zulässige Werte von 1 bis 3
`data`:	zu sendende Nutzdaten
`length`:	Länge der Nutzdaten

  Die Funktion versendet die übergebenen Daten über UART. Dafür wird der angegebene Port genutzt. Da der Parameter `data` Nutzdaten variabler Länge enthält, muss deren Länge mit angegeben werden. Nachrichten, die über UART versandt werden, dienen nicht der Kommunikation mit anderen Knoten sondern sind für Debug-Meldungen während der Entwicklung gedacht. Dies ist nötig, wenn die Anwendung auf Imote2-Knoten läuft, da dann Meldungen über den normalen Logging-Mechanismus nicht vom Entwickler eingesehen werden können.

- `void SetLed(unsigned int no, unsigned int status)`

`no`:	Nummer der LED
`status`:	Status der LED

Die Funktion aktiviert oder deaktiviert die angegebene LED der Imote2-Plattform. Zulässige Nummern sind 1, 2 und 3. Die LEDs haben verschiedene Farben. Der Status wird als bool'scher Wert interpretiert, wird also als Status 0 angegeben, deaktiviert das die betreffende LED, andernfalls aktiviert die Funktion die LED. Ob die LED ein- oder ausgeschaltet war, spielt keine Rolle.

- `void ToggleLed(unsigned int no)`
  `no`: Nummer der LED

  Die Funktion schaltet die angegebene LED der Imote2-Plattform um. War die betreffende LED eingeschaltet, deaktiviert die Funktion sie. Andernfalls schaltet sie sie ein. Zulässige Nummern sind 1, 2 und 3. Die LEDs haben verschiedene Farben.

- `void IoSetInterrupt(unsigned int no, bool value)`
  `no`: Nummer des Interrupt-Typs
  `value`: Belegung des Flags

  Die Funktion kann von einer Applikation genutzt werden, um zu registrieren, dass sie über auftretende Interrupts bestimmten Typs informiert werden will. Es können mehrere Interrupt-Typen als für die Applikation von Interesse markiert sein. Für jeden Interrupt Typ existiert ein Flag, das steuert, ob der Applikation ein auftretender Interrupt dieses Typs gemeldet werden soll oder nicht. Das Flag wird gesetzt, wenn `value` den Wert *TRUE* hat, andernfalls wird es als nicht gesetzt markiert. Die vorherige Belegung spielt keine Rolle.

- `void SpiConfig(int port, int bits, int speed, bool master, bool polarity, bool phase, bool frm_polarity)`
  `port`: Nummer des zu konfigurierenden Ports (1, 2 oder 3)
  `bits`: Anzahl an Bits, die übertragen werden
  `speed`: Geschwindigkeit, zulässige Werte von 0 bis 4095
  `master`: Belegung des `Master`-Flags
  `polarity`: Polarität
  `phase`: Phase
  `frm_polarity`: FRM-Polarität

  Die Funktion wird von der Applikation aufgerufen, um den angegebenen SPI-Port des Knotens zu konfigurieren. Die konfigurierbaren Parameter des Ports werden entsprechend der übergebenen Werte gesetzt. Der Aufruf der Funktion erfolgt möglichst in der `Init()`-Funktion der Applikation. Die Funktion konfiguriert den Port Bits mit einer Geschwindigkeit von $\frac{13Mhz}{speed+1}$ (Bits pro Sekunde) zu übertragen. Falls das `Master`-Flag gesetzt ist, wird `clock` generiert, `clock polarity (SPO)` mit Phase (SPH) gesetzt und FRM-Polarität (IFS) invertiert.

- `void SpiSend(int port, int data)`
  `port`: SPI-Port für den Datenversand
  `data`: zu sendende Nutzdaten

  Die Funktion versendet Daten über den angegebenen SPI-Port. Eine Längenangabe ist hierbei nicht nötig, da der verwendete Datentyp eine feste Länge

hat. Date, die über SPI versandt werden, dienen nicht der Kommunikation mit anderen Knoten.

- `int registerService(const char* name, const int throughput, const int reactionDelay)`

`name`:	global eindeutiger Name des zu registrierenden Dienstes
`throughput`:	möglicher Durchsatz des Dienstes in Bytes pro Sekunde
`reactionDelay`:	Antwortzeit in Millisekunden (nach Datenanforderung)

Die Funktion registriert einen Dienst mit dem angegebenen Namen (global eindeutig) und weist ihm eine ID zu (lokal eindeutig). Der Rückgabewert der Funktion ist die dem Dienst zugewiesene ID. Der Name und die angebotene Dienstgüte wird anderen Knoten mitgeteilt. Die Veröffentlichung des Durchsatzes ist derzeit nicht implementiert, dessen Wert daher ohne Wirkung. Die Signatur enthält den Parameter in Hinblick auf zukünftige Erweiterungen. Eine Antwortverzögerung `reactionDelay` entsteht möglicherweise durch die Ermittlung eines Messwertes und wird veröffentlicht, um potentiellen Abonnenten zu ermöglichen, die Verzögerung bei der Daten-Anforderung einzukalkulieren. Die angegebene Antwortverzögerung beinhaltet nur die vom Dienst benötigte Zeit vom Aufruf der Funktion `requestDataForService` seiner Schnittstelle bis zur Herausgabe der Daten mittels `setRequestedData` (Schnittstelle des SDL-Prozesses). Signallaufzeiten und Verzögerungen beispielsweise durch Wettbewerb um das Medium und Übertragungsverzögerungen sind darin nicht enthalten. Die Funktion führt keine Überprüfung durch, ob die Applikation bereits einen Dienst mit diesem Namen registriert hat. Es wird in jedem Fall eine neue ID generiert und zugewiesen.

- `void deregisterService(int id)`

`id`:	ID eines registrierten Dienstes

Die Funktion hebt die Registrierung eines Dienstes auf. Das Abonnement noch vorhandener Abonnenten des Dienstes endet. Die Funktion führt keine Überprüfung durch, ob die übergebene ID gültig ist.

- `int subscribeService(const char* name, const int period, const int delay)`

`name`:	global eindeutiger Name des zu abonnierenden Dienstes
`period`:	Intervall der periodischen Daten-Anforderung in Millisekunden
`delay`:	maximal tolerierbare Antwortverzögerung in Millisekunden

Die Funktion abonniert einen Dienst. Sie generiert eine neue, lokal eindeutige ID und weist sie dem Dienst zu. Der Rückgabewert der Funktion ist die dem Dienst zugewiesene ID. Die Zuweisung einer ID geschieht auch dann, wenn es keinen Dienst dieses Namens gibt. Der abonnierte Dienst wird in den durch `period` bestimmten Abständen Daten senden. Der Abonnent muss bei Erhalt aber selbst entscheiden, ob die Daten für ihn bestimmt sind. Denn sind andere Abonnenten mit abweichender Periode beim Dienst angemeldet, so sendet der Dienst auch zu deren Periodengrenzen an alle Abonnenten. Wird als Periode 0 angegeben, findet kein automatischer Datenversand für diesen Abonnenten statt. Der Abonnent ist dann zwar angemeldet und erhält das, was der Dienst aufgrund weiterer angemeldeter Abonnenten versendet, aber es werden keine

regelmäßigen Daten-Anforderungen speziell für ihn versandt. Der Parameter `delay` bestimmt die maximal tolerierbare Antwortverzögerung in Millisekunden – wird von der derzeitigen Implementierung aber nicht berücksichtigt und ist in Hinblick auf zukünftige Erweiterungen in der Funktions-Signatur enthalten.

- `void unsubscribeService(int id)`

`id`:	ID eines abonnierten Dienstes

  Die Funktion beendet das Abonnement eines Dienstes. Die Funktion führt keine Überprüfung durch, ob die übergebene ID gültig ist.

- `void setRequestedData(const int serviceID, const char* data, int length)`

`serviceID`:	ID des Dienstes, der die Daten liefert
`data`:	angeforderte Nutzdaten
`length`:	Länge der Nutzdaten

  Die Funktion wird von der Applikation aufgerufen, nachdem eine vorhergehende Anforderung von Daten mittels `requestDataForService` stattfand. Der Aufruf bedeutet, dass die angeforderten Daten nun bereit stehen und an die Abonnenten des Dienstes versandt werden sollen. Da eine Applikation mehrere Dienste anbieten kann, muss die ID um den konkreten Dienst zu identifizieren, von dem Daten benötigt werden, sowohl bei der Anforderung mittels `requestDataForService` als auch bei der Herausgabe mittels `setRequestedData` angegeben werden. Bei Aufruf der Funktion wird ein `Timestamp` erzeugt und den Nutzdaten angehängt. Die ist für die Applikation allerdings transparent. Der Zeitstempel kann vom Empfänger genutzt werden, um über die Relevanz der Daten für ihn zu entscheiden. Dem Funktionsaufruf folgt ein automatischer Versand der Daten an die Abonnenten.

Anhang C

CD

Diese CD enthält das im Rahmen der Diplomarbeit entstandene SDL-System zur Verwendung mit Telelogic Tau.

Literaturverzeichnis

[BGK07] Becker, Philipp, Reinhard Gotzhein und Thomas Kuhn: *MacZ - A Quality-of-Service MAC Layer for Ad-hoc Networks.* In: *HIS*, Seiten 277–282, 2007.

[DSJ07] Du, Shu, Amit Kumar Saha und David B. Johnson: *RMAC: A Routing-Enhanced Duty-Cycle MAC Protocol for Wireless Sensor Networks.* In: *INFOCOM*, Seiten 1478–1486, 2007.

[FGJ+05] Fliege, Ingmar, Alexander Geraldy, Simon Jung, Thomas Kuhn, Christian Webel und Christian Weber: *Konzept und Struktur des SDL Environment Framework (SEnF).* Technischer Bericht 341/05, TU Kaiserslautern, 2005.

[FGW06] Fliege, Ingmar, Rüdiger Grammes und Christian Weber: *ConTraST - A Configurable SDL Transpiler And Runtime Environment.* Band 4320 der Reihe *Lecture Notes in Computer Science*, Seiten 216–228. Springer, 2006.

[FK07] Fliege, Ingmar und Jan Koch: *AmICoM - Formally specified service platform for ambient intelligence networks.* Technischer Bericht D2.6.1, Fraunhofer IESE, Juni 2007.

[GK08] Gotzhein, Reinhard und Thomas Kuhn: *Decentralized Tick Synchronization for Multi-Hop Medium Slotting in Wireless Ad Hoc Networks Using Black Bursts.* In: *SECON*, Seiten 422–431, 2008.

[GKLC] Gotzhein, Reinhard, Marc Krämer, L. Litz und A. Chamaken: *Energy-aware System Design with SDL.* 14th International SDL Forum - Lecture Notes in Computer Science. noch unveröffentlicht.

[KF07] Koch, Jan und Ingmar Fliege: *AmICoM - Middleware Support for Ambient Communication.* 2007. 2nd Workshop on Requirements and Solutions for Pervasive Software Infrastructures (RSPSI) at the 9th International Conference on Ubiquitous Computing (UbiComp).

[SDGJ08] Sun, Yanjun, Shu Du, Omer Gurewitz und David B. Johnson: *DW-MAC: a low latency, energy efficient demand-wakeup MAC protocol for wireless sensor networks.* In: *MobiHoc*, Seiten 53–62, 2008.

[vDL03] Dam, Tijs van und Koen Langendoen: *An adaptive energy-efficient MAC protocol for wireless sensor networks.* In: *SenSys*, Seiten 171–180, 2003.

[WG07] Webel, Christian und Reinhard Gotzhein: *Formalization of Network Quality-of-Service Requirements.* In: *FORTE*, Seiten 309–324, 2007.

[YHE02] Ye, Wei, John Heidemann und Deborah Estrin: *An Energy-Efficient MAC Protocol for Wireless Sensor Networks*, 2002.

[YHE03] Ye, Wei, John Heidemann und Deborah Estrin: *Medium Access Control with Coordinated, Adaptive Sleeping for Wireless Sensor Networks.* IEEE/ACM Transactions on Networking, 12:493–506, 2003.