Hochschule RheinMain
Fachbereich Design Informatik Medien
Studiengang Informatik

Master-Thesis
zur Erlangung des akademischen Grades
Master of Science – M.Sc.

Konzeption einer leichtgewichtigen, datenzentrierten Middleware für Sensornetze und eine prototypische Realisierung für ZigBee

vorgelegt von Kai Beckmann

am 21. April 2010

Referent: Prof. Dr. Kröger
Korreferent: Prof. Dr. Gergeleit

Erklärung

Ich versichere, dass ich die Master-Thesis selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Wiesbaden, 21.04.2010 Kai Beckmann

Hiermit erkläre ich mein Einverständnis mit den im Folgenden aufgeführten Verbreitungsformen dieser Master-Thesis:

Verbreitungsform	**ja**	**nein**
Einstellung der Arbeit in die Bibliothek der Hochschule RheinMain	√	
Veröffentlichung des Titels der Arbeit im Internet	√	
Veröffentlichung der Arbeit im Internet	√	

Wiesbaden, 21.04.2010 Kai Beckmann

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Quellcodes

Kapitel 1

Einleitung

Computer und Rechnernetze sind heutzutage allgegenwärtig und durchdringen den Alltag eines jeden in immer stärkerem Maße. Die ständigen Fortschritte in der Halbleitertechnik sorgen dafür, dass Rechner immer leistungsfähiger, kleiner und günstiger werden. Durch diese Entwicklung eröffnen sich immer neue Einsatzgebiete, oder rechnerbasierte Systeme nehmen den Platz von bisherigen Lösungen ein. Insbesondere der Bereich für Anwendungen eingebetteter Systeme profitiert von den günstigen Preisen und weitet sich auf immer mehr Gegenstände des Alltags, der Wirtschaft und auf militärische Anwendungen aus.

Eine Vision, die aus dieser Entwicklung entstanden ist, ist die des „SmartDust" [CES04]. Millionen von staubkorngroßen autarken Rechnerknoten werden großflächig in Gebieten ausgebracht, um kooperativ Überwachungsaufgaben auszuführen. Ursprünglich ist diese Vision im militärischen Bereich angesiedelt, aber im zivilen Umfeld ergeben sich ebenfalls viele Einsatzmöglichkeiten.

Ein solches Netzwerk aus einer großen Anzahl an kleinen, billigen und leicht zu ersetzenden Rechnersystemen, die in Kooperation mit den Knoten im Netzwerk eine gemeinsame Aufgabe erfüllen, meist das Sammeln von Umgebungsdaten, wird Sensornetz genannt und die einzelnen Knoten heißen Sensorknoten. Ein einzelner Sensorknoten ist ein vollständiger autonomer Recher mit eigenem Prozessor, Speicher und Kommunikationsschnittstelle und verfügt über Sensoren oder manchmal auch Aktoren, um mit der Umgebung interagieren zu können. Die Klasse der Sensornetze, die drahtlos über Funk kommunizieren und meist autark mit einer eigenen Energieversorgung ausgestattet sind, werden „Wireless Sensor Networks" (*WSN*) genannt. Diese werden vorrangig in der Literatur betrachtet, da die Probleme, die sich aus der limitierten Energieversorgung und dem

Einfluss der Umwelt auf die Knoten und das drahtlose Netzwerk ergeben, besonders interessant und herausfordernd sind. Die Anforderung, dass die Stückkosten für einen leicht zu ersetzenden Sensorknoten möglichst gering sein sollen, führt dazu, dass Sensorknoten nur über geringe Hardware-Ressourcen wie Rechenkapazität, Speicher und hoher Bandbreite verfügen [ASSC02].

Aktuelle Hardware-Plattformen für drahtlose Sensornetze verwenden einen „System-on-a-Chip"-Ansatz (*SoC*), bei dem Mikrocontroller, Speicher, Energiemanagement, Analog-Digital-Wandler, Funk-Transceiver und weitere Funktionseinheiten in einem Chip untergebracht sind. Für die Herstellung eines einsatzfähigen Sensorknotens werden dann wenig zusätzliche Bauteile benötigt. Immer mehr Funk-Transceiver auf SoC-Plattformen sind für die Verwendung mit dem ZigBee-Funkprotokoll ausgelegt [BPC+07]. Der ZigBee-Standard zielt unter anderem auf die Haus- und Gebäudeautomation und ist für günstige Geräte mit geringen Energieverbrauch ausgelegt. Die Auslegung von ZigBee und entsprechenden SoC für den Massenmarkt und die damit einhergehenden günstigeren Stückkosten für Hardware und Produktion machen drahtlose Sensornetze für viele neue Anwendungsfälle in der Industrie, der Forschung und dem privaten Umfeld interessant.

Bisher ist ein großes Problem im Bereich von Sensornetzen die systematische Entwicklung von adäquater Software. Dies hat seinen Ursprung in unterschiedlichen Bereichen. Der Aufbau und die Verwaltung eines Sensornetzes können sehr komplex sein, da eine sehr große Anzahl an Knoten über unzuverlässige Kommunikationsverbindungen Daten austauschen müssen. Die Topologie kann sich ändern, Knoten können ausfallen oder temporär nicht erreichbar sein [MM07].

Der Einsatz von Middleware stellt einen Ansatz dar, diese Probleme wenigstens teilweise vor dem Entwickler von Anwendungen zu verbergen und ihm eine abstrakte Anwendungsschnittstelle für die Entwicklung von Anwendungen zu bieten. Die Aufgabe von Sensornetzen ist die Ermittlung von Sensordaten aus der Umgebung und ihr Transport. Dabei ist es in der Regel nicht relevant, welcher konkrete Sensorknoten eine Aufgabe wahrnimmt, sondern ausschließlich, dass die Daten ermittelt und zum Ziel geleitet werden [RKM02]. Dies macht den Einsatz einer Middleware interessant, die als Schnittstelle für Anwendungen den „Data Distribution Service" (*DDS*) der OMG verwendet. Die Anwendungsschnittstelle von DDS ist datenzentriert, und es werden eine Reihe von QoS-Richtlinien definiert, die für Sensornetze relevant sein können. Bei dem datenzentrierten Ansatz von DDS erfolgt die Adressierung über die Art der Daten und nicht über Stationsadressen der Sensorknoten.

Dem effektiven Einsatz einer generischen Middleware steht ein anderes Problem von Sensornetzen entgegen: Da die Hardware-Plattformen von Sensorknoten auf einen minimalen Stückpreis und geringen Energieverbrauch optimiert sind und daher nur über eingeschränkte Ressourcen verfügen, sind alle Komponenten eng aufeinander abgestimmt. Die verwendete Hardware und die dafür entwickelte Software werden oft für die jeweilige Anwendung individuell entwickelt und häufig sind auch die Kommunikationsprotokolle direkter Teil der Anwendung. Es ist offensichtlich, dass in diesen Fällen einzelne Bestandteile der Anwendungen oder der Protokoll-Implementierungen schwer wiederverwendet oder an neue Sensorknotenplattformen bzw. Anforderungen angepasst werden können [KW05].

Im Rahmen dieser Arbeit wird untersucht, wie die Software-Entwicklung und die Portierung oder Erweiterung von existierenden Anwendungen auf neue Plattformen mit Hilfe einer datenzentrierten Middleware vereinfacht werden können. Dabei ist es ein Fokus, die einheitliche und standardisierte Anwendungsschnittstelle von DDS für die Middleware zu verwenden. Damit soll es zum einem ermöglicht werden Anwendungen im Bereich der Sensornetze einfach zu portieren oder zu erweitern. Darüber hinaus könnten Teile von Anwendungen oder das Wissen der Entwickler aus anderen Kontexten wiederverwendet werden.

Um den Widerspruch zwischen der Notwendigkeit, Anwendungen, Middleware und Kommunikationssystem eng und anwendungsabhängig miteinander zu verzahnen und dem Wunsch, Anwendungen auf eine abstrakte Schnittstelle aufzusetzen und damit einfach portabel und erweiterbar zu machen, zu lösen, wird in dieser Arbeit der Ansatz der modellgetriebenen Softwareentwicklung verwendet (*MDSD* Model-Driven Software Development [SVEH07]). Hierbei erfolgt die Entwicklung von Software in drei Stufen: Die Anwendungen und die Zielplattformen, also der Problem- und der Lösungsraum, werden zuerst abstrakt und formal modelliert. Die Modelle werden danach über Modelltransformationen verarbeitet und das plattformunabhängige Modell der Anwendung mittels der Informationen über die Zielplattform in ein plattformspezifisches Modell transformiert. Aus diesem Modell wird in der letzen Stufe der plattformspezifische Programmcode generiert. Die „Model-Driven Architecture” (*MDA*) der OMG ist ein Verfahren, das für einen solchen MDSD-Ansatz häufig verwendet wird [Bro04].

Durch den MDSD-Ansatz ist es möglich, dass Anwendungen auf die Anwendungsschnittstelle des DDS-Standards aufsetzen können und die generierte Middleware-Implementierung auf die individuellen Bedürfnisse der Anwendung angepasst wird. Die verwendete Funktionalität des DDS-Standards wird dafür

vom Anwendungsentwickler ausgewählt und der Aufbau der DDS-Middleware für das Sensornetz modelliert. Aus diesen Informationen wird eine angepasste Middleware generiert, die genau die Funktionalität enthält, die von der Anwendung benötigt wird. Die dafür notwendigen Software-Komponenten werden eng verzahnt, und es wird damit sicherstellt, dass nur die für die Erfüllung der jeweiligen Aufgabe unbedingt notwendigen Ressourcen eingesetzt werden. Des Weiteren wird dem Entwickler auf Basis seiner Konfiguration und den bekannten Informationen über die Zielplattform eine frühzeitige Rückmeldung darüber gegeben, ob seine Anforderungen mit den gegebenen Ressourcen der Zielplattform erfüllbar sind.

Zur Überprüfung des aufgezeigten Ansatzes auf seine Eignung wird zum einen eine prototypische Middleware-Implementierung entworfen, die auf einer typischen Sensorknotenplattform mit ZigBee lauffähig sein soll. Zum anderen wird ein Framework bereitgestellt, mit dem die Middleware-Implementierung an spezifische Anforderungen von Anwendungen angepasst und nur mit der unbedingt notwendigen Funktionalität generiert wird.

Diese Master-Thesis entstand im Kontext des „Labors für verteilte Systeme" der Hochschule RheinMain und des dort bearbeiteten Forschungsfelds „Ambient Assisted Living" (*AAL*), welches sich unter anderem mit der Problemstellung beschäftigt, wie Informationstechnologie dafür eingesetzt werden kann, älteren Menschen länger ein selbstbestimmtes Leben in ihrer eigenen Wohnung zu ermöglichen. Dieses Forschungsfeld wird mit Hinblick auf den demografischen Wandel in Europa und den daraus erwachsenden Problemen für die Sozialsysteme von der Europäischen Union [Eic10] gefördert. In diesem Kontext können Sensornetze dazu verwendet werden, in einer Wohnung Daten zu sammeln, die von komplexeren Assistenzsystemen dazu verwendet werden, Gefahren zu erkennen und Entscheidungen über Hilfsmaßnahmen zu treffen. Sensornetze sind für AAL-Anwendungen interessant, da sie auf Grund der günstigen Anschaffungskosten und den flexiblen Einsatzmöglichkeiten von autarken drahtlosen Sensorknoten sehr einfach in einer existierenden Wohnung nachträglich installiert werden können.

Die weitere Arbeit gliedert sich wie folgt: Nach dieser Einleitung werden in Kapitel 2 die Grundlagen, über Sensornetze (in 2.1), das ZigBee-Protokoll (in 2.2) und den DDS-Standard (in 2.3) beschrieben, die für das Verständnis dieser Arbeit notwendig sind. Das Kapitel 3 befasst sich danach mit der Problemanalyse. Dazu werden in Abschnitt 3.1 die Anforderungen an eine Middleware für Sensornetze

untersucht und festgelegt. Die weitere Untersuchung beschäftigt sich mit den Eigenschaften des DDS-Standards. In Abschnitt 3.3 wird das Datenmodell und in Abschnitt 3.4.1 die für diese Arbeit relevante Funktionalität von DDS untersucht. Abschnitt 3.5 analysiert die Aspekte, die für ein Kommunikationsprotokoll zu beachten sind und in Abschnitt 3.6 wird der MDSD-Ansatz genauer untersucht. Das Analysekapitel wird in Abschnitt 3.7 mit den Designentscheidungen abgeschlossen.

Im Kapitel 4 werden die Entwürfe der Architekturen für die eigentliche Middleware (Abschnitt 4.1.2) und den MDSD-basierten Generierungsprozess des anwendungsspezifischen Programmcodes der Middleware-Implementierungen (Abschnitt 4.1.1), vorgestellt. Hierfür werden in Abschnitt 4.2 Metamodelle definiert, die für die Modellierung und Konfiguration des DDS-Systems benötigt werden und in Abschnitt 4.3 wird der Software-Entwicklungsprozess selber dargestellt. Das detailliertere Design der Middleware wird in Abschnitt 4.5 und das des Kommunikationsprotokolls in 4.4 festgelegt.

Die prototypische Implementierung in dieser Arbeit wird im Kapitel 5 dargestellt. Dies umfasst den Prototypen für die eigentliche Middleware in Abschnitt 5.3, 5.5 und 5.5 sowie den MDSD-Prozess, der sie generiert in Abschnitt 5.6. Auf Basis dieser Implementierung wird in Kapitel 6 der gewählte Ansatz in dieser Arbeit auf seine Tauglichkeit evaluiert. Die Ergebnisse dieser Arbeit werden dann in Kapitel 7 zusammengefasst und es wird aufgeführt, welche Möglichkeiten dieser Ansatz für zukünftige Weiterentwicklungen bietet.

Kapitel 2

Grundlagen

2.1 Drahtlose Sensornetze

2.1.1 Einführung drahtlose Sensornetze

Die Entwicklung in der Halbleitertechnik macht es möglich, immer kleinere, billigere und leistungsfähigere Computer herzustellen. Damit eröffnen sich immer neue Anwendungsbereiche, in denen zuvor der Einsatz von Rechnersystemen zu teuer war. Dies trifft vor allem auf die eingebetteten Systeme zu, in denen Rechner Teil von Geräten sind. Eine neue Klasse von Rechnersystemen ist aus diesem Bereich entstanden, die der Sensornetze [FFM+09].

Ein Sensornetz besteht aus eine großen Menge an Sensorknoten, die jeweils ein autonomes Rechnersystem darstellen, und im Verbund kooperativ Aufgaben erfüllen,die häufig in der Sammlung von Umgebungsdaten besteht. Eine besondere Gruppe von Sensornetzen kommuniziert drahtlos über Funk, und die einzelnen Knoten verfügen über eine autarke Energieversorgung. Diese Sensornetze werden „Wireless Sensor Networks” genannt; sie sind zurzeit das primäre Forschungsfeld im Bereich der Sensornetze. Der Grund dafür sind zum einem die besonderen Einschränkungen, denen diese Klasse von Sensorknoten unterworfen ist und zum anderen die vielfältigen Einsatzgebiete, in denen diese Systeme eingesetzt werden können [CES04].

Die Einschränkungen, denen drahtlose Sensornetze unterworfen sind, liegen zum einem darin, dass die Stückkosten für einen drahtlosen Sensorknoten minimal sein sollen. Die Sensorknoten sollen leicht zu ersetzen sein und in großer

Anzahl eingesetzt werden können. Des Weiteren verfügen sie in der Regel über eine autarke Energieversorgung. Oft werden dafür Batterien verwendet, was bedeutet, dass die Menge der verfügbaren Energie endlich ist. Ein Sensornetz ist daher nur solange einsatzfähig, wie eine kritische Mindestanzahl von Sensorknoten über Energie verfügt. Häufig ist es nicht zwingend vorgesehen, die endliche Energievorräte regelmäßig zu erneuern, etwa weil dies bei der großen Anzahl an Sensorknoten nicht immer möglich ist. Daher muss ein drahtloser Sensorknoten auch auf einen minimalen Energieverbrauch optimiert sein und sich im Betrieb möglichst lange in energiesparenden Schlafzuständen befinden.

Die Kombination aus den Ansprüchen, möglichst günstig in der Anschaffung und leicht zu ersetzen sein zu müssen und nur sehr wenig Energie verbrauchen zu dürfen, führt dazu, dass Sensorknoten nur über sehr wenig Ressourcen, wie Rechenkapazität, Speicher und Bandbreite verfügen und die Hardware hochintegriert ist. „System-on-a-Chip"-Ansätze sind für drahtlose Sensorknoten eher die Regel und neben dem Rechnersystem sind in der Regel auch Sensoren oder zumindest Analog-Digital-Wandler und Funk-Transceiver auf demselben Chip untergebracht.

Die möglichen Anwendungen ergeben sich teilweise aus den Einschränkungen, denen sie unterworfen sind. Es ist möglich und praktikabel, eine große Menge an Sensorknoten weit verteilt in einer Umgebung auszubringen oder zu installieren und über längere Zeiträume Daten in dieser Umgebung zu sammeln, zu verarbeiten und basierend auf den Ergebnissen Anwendungsfunktionaliät zu erbringen. Dabei zeigt sich eine weitere Einschränkung, der drahtlose Sensornetze unterworfen sind und die eine interessante Herausforderung bietet. Sensornetze sind der Umwelt in der sie ausgebracht sind, direkt ausgesetzt. Das hat Auswirkungen auf die Art, wie das Netzwerk aufgebaut wird und mit Störungen umgehen können muss. Auch Sicherheit ist ein besonderes Problem. Wenn keine Kontrolle über die Umgebung besteht, können Sensorknoten korrumpiert werden oder die drahtlose Kommunikation kann abgehört, gestört oder negativ beeinflusst werden.

Welche Eigenschaften genau einen Sensorknoten ausmachen, kann sehr anwendungsspezifisch sein. Die Literatur [CES04] [YMG08] und [FFM+09] zusammenfassend kann ein drahtlosen Sensornetz wie folgt definiert werden:

Ein drahtloses Sensornetz besteht aus einer Anzahl von kleinen autonomen Systemen, Sensorknoten genannt, die mit Hilfe von Sensoren physikalische Größen ihrer Umgebung messen und über limitierte Verarbeitungs- und Datenspei-

cherungskapazitäten verfügen. Ein drahtloser Sensorknoten besteht aus einer sensor-, verarbeitungs- und funkbasierten Sende- und Empfangseinheit. Ist der Sensorknoten autark, verfügt er des Weiteren über eine eigene Energieversorgung, die oft endlich ist, und er muss, um eine lange Lebensdauer zu erreichen, energieeffizient arbeiten. Die Sensorknoten kooperieren im Sensornetz, um mindestens eine gemeinsame Aufgabe zu erfüllen. Es gibt spezielle Knoten im Sensornetz, die als Datensenke die Ergebnisse und Daten der Sensorknoten sammeln und dem Anwender zur Verfügung stellen. Die Topologie des Netzes kann sich dynamisch, z.B. durch Umwelteinflüsse ändern. Das Netzwerk passt sich selbstständig an neue Gegebenheiten an, ohne dass ein Eingriff von außen notwendig ist. Drahtlose Sensorknoten können über andere Knoten als Zwischenstation Daten mit nicht direkt erreichbaren Sensorknoten austauschen. Da eine Datenübertragung relativ viel Energie benötigt, wird versucht, die Daten intern im Netz zu verarbeiten oder aufzubereiten und damit die Menge der zu übertragenden Daten zu reduzieren. Auf Grund der Anforderungen an Energieeffizienz und minimalen Ressourcenverbrauch werden für Sensornetze neue Verfahren und Protokolle benötigt.

Im Folgenden soll auf die für diese Arbeit relevanten Bereiche von Sensornetzen eingegangen werden. Dazu werden in Abschnitt 2.1.2 verschiedene Typen von Sensornetzen und mögliche Anwendungsfelder für drahtlose Sensornetze beschrieben. In Abschnitt 2.1.3 wird danach genauer auf die Aspekte Energie und Ressourcen in drahtlosen Sensornetzen eingegangen.

2.1.2 Typen von Sensornetzen und mögliche Anwendungen

Es gibt kein einheitliches Schema für die Gestaltung eines Sensornetz. Der Aufbau und die verwendete Technologie ist immer abhängig von den Anforderungen der Anwendungen und der Umgebung, in der das Sensornetz operieren soll. Daher sollen im Weiteren mögliche Typen von Sensornetzen und eine Auswahl an existierenden Anwendungen beschrieben werden.

2.1.2.1 Typen von Sensornetzen

Die Umgebungen, in denen Sensornetze ausgebracht werden, beeinflussen die Anforderungen, die die Sensornetze erfüllen müssen, signifikant. Daher ist es

nach [YMG08] sinnvoll, die möglichen Typen von Sensornetzen auf Grund der Umgebung, in der sie eingesetzt werden, zu unterscheiden. In [YMG08] werden die folgenden fünf Typen von Sensornetzen aufgeführt:

1. Terrestrische Sensornetze
2. Unterirdische Sensornetze
3. Unterwasser-Sensornetze
4. Multimedia-Sensornetze
5. Mobile Sensornetze

Diese Liste ist sicherlich nicht vollständig, so gibt es inzwischen auch Untersuchungen über den Einsatz von Sensornetzen im Weltraum [DBM+09].

Terrestrische Sensornetze entsprechen am ehesten der Vorstellung eines drahtlosen Sensornetzes. Die Sensorknoten, deren Anzahl sehr groß sein kann, sind in einer definierten Gegend ausgebracht. Die Knoten können entweder nach einem Plan an bestimmte Stellen platziert werden, oder rein zufällig in der Gegend verstreut werden, zum Beispiel von einem Flugzeug. Diese Gruppe von Sensornetzen verwendet häufig funkbasierte Kommunikationssysteme.

Unterirdische Sensornetze bestehen aus Sensorknoten, die unter der Erde vergraben oder in Höhlen, Minen etc. angebracht werden. Zusätzliche Sensorknoten sind an zugänglicheren, meist oberirdischen Stellen installiert und haben die Aufgabe, die Daten des Sensornetzes zu sammeln und an die Auswertungssysteme weiterzuleiten. Unterirdische System sind oft teurerer als terrestrische. Zum einem ist die zuverlässige Kommunikation durch den Boden ein Problem; Wasser, Steine und Mineralien erschweren eine Funkverbindung. Zum anderen ist die Ausbringung der Sensorknoten aufwändiger und eine Wartung danach möglicherweise nicht mehr möglich. Ein solche System muss daher genau geplant werden und die Platzierung der Knoten sollte optimal und nicht zufällig sein.

Unterwasser-Sensornetze sind wie die unterirdischen Sensornetze aufwändiger und teuer in ihrerer Anschaffung und Installation als die terrestrischen Sensornetze. Die Knoten befinden sich im Wasser, entweder mobil schwimmend oder an einer festen Position auf dem Boden oder an einer Boje verankert. Wasser stellt eine raue Umgebung für elektronische Systeme dar; die Sensorknoten

müssen entsprechend geschützt sein und sich an die wechselnden Bedingungen anpassen können. Als Medium für die Kommunikation wird häufig das Wasser verwendet und Daten mittels Schall übertragen.

Multimedia-Sensornetze haben das Ziel, Multimedia-Daten, wie Video-, Audio- oder Bilddaten mit günstigen Sensorknoten aufzunehmen, zu verarbeiten und weiterzuleiten. Sie werden dafür genutzt, Bereiche zu überwachen und definierte Ereignisse zu erkennen und zu melden. Die Verarbeitung der Daten geschieht auf den Knoten im Netz. Die Datenübertragung erfolgt meist über Funk. Auf Grund der Größe der Multi-Media-Daten benötigt ein solches Sensornetz eine große Bandbreite.

Mobile Sensornetze Bei mobilen Sensornetzen sind die Knoten beweglich und können über die Zeit ihren Standort ändern. Es ist dabei zu unterscheiden, ob die Entscheidung zu einer Standortänderung von den Sensorknoten selber getroffen und durchgeführt wird, oder ob die Knoten an sich bewegenden Objekten angebracht sind. Sich selbständig bewegende Sensorknoten eröffnen neue Möglichkeiten für Sensornetze. Sensorknoten können sich zu den für die Anwendungen interessanten Stellen begeben und dort Messungen durchführen. Der Ausfall von Sensorknoten kann durch andere Sensorknoten durch Positionsänderungen ausgeglichen werden. Auch ist es möglich, dass Sensorknoten Positionen einnehmen, die dem Datenaustausch förderlich sind, bzw. diesen erst ermöglichen. Sensorknoten hingegen, die sich nicht selbstständig bewegen können und nicht immer mit anderen Sensorknoten kommunizieren können, müssen häufig ihre Sensordaten intern speichern, bis sie sie an eine Datensenke übertragen werden können.

2.1.2.2 Anwendungen für Sensornetze

Sensornetze sind eine relativ neue Technologie. Das bedeutet, der Forschungsanteil in den in der Literatur beschriebenen Anwendungen ist sehr hoch. Nach [CES04] lassen grob sich die Aufgaben, für die ein Sensornetz eingesetzt werden kann, auf folgende drei Punkte reduzieren:

- Raum-/ Umgebungsüberwachung
- Überwachung von Objekten

- Überwachung der Interaktion von Objekten untereinander oder mit der Umgebung.

Es sollen nun einige in der Literatur beschriebene Anwendungen kurz vorgestellt und damit eine Vorstellung von den Einsatzmöglichkeiten von Sensornetzen gegeben werden.

Das „ZebraNet" [ZSLM04] ist eine gern zitierte Anwendung von drahtlosen Sensornetzen in der Literatur. Das Ziel dieses Projekt war es, Informationen über die Wanderungen von Zebras zu sammeln. Hierfür wurden Sensorknoten entwickelt, die von Zebras als Halsband getragen werden können und über GPS regelmäßig die Position des Zebras ermitteln. Als Energieversorgung sind auf dem Halsband Solarzellen angebracht, und der Sensorknoten ist mit einem Lithium-Ionen-Akku ausgestattet, der von den Solarzellen geladen wird. Die Sensorknoten kommunizieren über einen Funk-Transceiver mit 900 Mhz, der unter realistischen Bedingungen eine Reichweite von bis zu 1 km ermöglicht. Da nicht sehr viele Zebras mit einem Sensorknoten ausgestattet werden können und sich diese bewegen und daher nicht ständig in der Nähe der Empfangsstation befinden, müssen die Sensorknoten die Positionsdaten lokal speichern. Das für das ZebraNet-Projekt entwickelte Protokoll sieht vor, dass die Sensorknoten untereinander ihre gesammelten Positionsdaten austauschen, wenn sie miteinander kommunizieren können. Wenn ein Zebra in der Nähe der Empfangsstation ist, werden die gesammelten Daten übertragen. Damit wird es ermöglicht, auch die Positionen von Zebras zu ermitteln, die nie in der Nähe einer Empfangsstation sind.

Ein weiteres oft in der Literatur zitiertes Sensornetz ist das des „Great Duck"-Projektes [MCP+02]. Das drahtlose Sensornetz wird dort eingesetzt, um Umwelt und „Lebensräume" von Tieren auf Great Duck Island zu überwachen und Erfahrungen mit Sensornetzen in einem realen Umfeld zu sammeln. Für die Sensorknoten wird eine existierende kommerzielle Hardware-Plattform verwendet, die „Mica Hardware Platform", die von zwei AA-Batterien mit Energie versorgt wird. Die an die Hardware-Plattform des Sensorknotens angeschlossenen Sensoren messen Licht, Temperatur, Luftdruck und Luftfeuchtigkeit. In dem Projekt wurden 32 dieser Sensorknoten auf der Insel verteilt und Untersuchungen zu dem Energieverbrauch und dem effektiven Routen von Daten zu einer Datensenke angestellt.

In [BAB+07] werden verschiedene prototypische Anwendungen für Sensornetze im medizinischen Umfeld beschrieben. Die Sensorknoten wurden auf Basis der

kommerziellen Hardware-Plattformen „Tmote Sky" und „SHIMMER" aufgebaut. Beide Plattformen verwenden einen Funk-Transceiver, der das IEEE 802.15.4-Protokoll unterstützt. In einer beschriebenen Anwendung soll ein Sensorknoten an einem Baby befestigt werden und über Beschleunigungssensoren festgestellt werden, wie das Kind liegt. Um das Risiko des plötzlichen Kindstods zu reduzieren, werden die Eltern alarmiert, wenn das Kind auf dem Bauch liegt.

Eine weitere in [BAB+07] vorgestellte prototypische Anwendung verwendet einen Sensorknoten, um die Herzfrequenz von Feuerwehrleuten im Einsatz zu überwachen.

Industrielle Umgebungen sind ein weiteres mögliches Anwendungsfeld für Sensornetze. In [FFM+09] wird ein Forschungsprojekt beschrieben, in dem Sensorknoten mit Temperatursensoren an einer Maschine für die Herstellung von Gegenständen aus Plastik angebracht wurden, um die Temperatur in der Maschine zu kontrollieren. Für diesen Zweck wurden aus verbreiteten Hardware-Komponenten eigene Sensorknoten entwickelt, die über einen Funk-Transceiver, der das IEEE 802.15.4-Protokoll unterstützt, die Temperaturdaten an eine Sammelstelle schicken.

Der Begriff der „Anwendung" ist meist ein Oberbegriff für die zusammenhängende Funktionalität, die von den einzelnen Sensorknoten in Kooperation erbracht wird. Im Unterschied zu dieser Definition wird dies in dieser Arbeit mit dem Begriff System beschrieben, und eine einzelne Anwendung umfasst die autonome Funktionalität, die auf einem einzelnen Sensorknoten läuft und zu dem Gesamtergebnis beiträgt.

2.1.3 Energie und Ressourcen

Drahtlose Sensornetze, deren Sensorknoten nur über eine endliche Menge an Energie verfügen, sollen über einen möglichst langen Zeitraum funktionsfähig sein. Das ist nur möglich, wenn die autonomen Sensorknoten sich eine möglichst lange Zeit in sehr energiesparenden Schlafzuständen befinden und nur für kurze Zeiträume aktiv werden, um Sensordaten auszulesen, zu verarbeiten und zu versenden [CES04].

Unnötiger Energieverbrauch muss vermieden bzw. reduziert werden. Aktuelle Hardware-Plattformen für Sensorknoten ermöglichen es, gezielt einzelne Kompo-

nenten auszuschalten, wenn sie nicht benötigt werden. Dies kann die Sensoren, Analog-Digital-Wandler oder natürlich die Funk-Transceiver umfassen.

Die Übertragung von Daten über Funk verbraucht, verglichen mit den anderen Operationen eines typischen drahtlosen Sensorknotens, relativ viel Energie. Dies gilt nicht nur für den Versand von Daten, sondern auch für den Empfang. Daher ist es keine gangbare Option, dass Sensorknoten eine ständige Empfangsbereitschaft aufrecht erhalten. Diese Einschränkung hat große Auswirkung auf die in einem Sensornets eingesetzten Protokolle und Algorithmen. In diesem Bereich herrscht zurzeit rege Forschungsaktivität [BPC+07].

Da Sensorknoten bereits Daten aus ihrer Umgebung sammeln, liegt es nahe, dass die Umgebung auch genutzt werden kann, um Energie zu gewinnen. Wie in der in Abschnitt 2.1.2.2 vorgestellten Anwendung ZebraNet kann dies über Solarzellen geschehen. Das Beispiel der Solarzellen zeigt aber, dass diese Energieversorgung nicht stetig und zuverlässig ist. Daher ist es notwendig, Energie zu speichern und den Energieverbrauch im Betrieb der verfügbaren Energiemenge anzupassen [SET09]. Neben der Verwendung der Sonne als Energielieferant können auch mechanische Bewegungen oder Kräfte in Energie umgesetzt werden. Beispielsweise kann dies mit Generatoren oder Piezo-Elementen geschehen. Auch Wärme kann direkt über Peltier-Elemente in elektrische Energie umgesetzt werden. Als Energiespeicher werden neben verschiedenen Typen von Akkus auch hochkapazitive Kondensatoren eingesetzt [SET09]

2.2 ZigBee

ZigBee ist ein Standard für ein Kommunikationsprotokoll für Funknetzwerke im Nahbereich mit einer niedrigen Bandbreite und zeichnet sich durch einen geringen Energieverbrauch und günstige Hardware-Kosten aus. Das Ziel von ZigBee ist es, Interoperabilität zwischen Geräten und Produkten verschiedener Hersteller, die den Standard unterstützen, zu ermöglichen. Es gibt daher festgelegte Profile für Geräte und Anwendungen, die entsprechend genutzt und implementiert werden sollen.

Der Protokoll-Stack von ZigBee definiert die OSI-Schichten drei bis sieben (Vermittlungs- bis Anwendungsschicht) und baut für die unteren Schichten auf dem IEEE 802.15.4-Standard auf. ZigBee wurde für Anwendungen in minimalen eingebetteten Systemen entworfen, die möglichst günstig sein sollen und mit

wenig Energie auskommen müssen, beispielsweise für die Hausautomation, zur Fernsteuerung von Geräten oder zum Sammeln von Sensordaten [Far08].

Der ZigBee-Standard wird von der ZigBee-Allianz betreut, welche ein Zusammenschluss aus „Hunderten von Unternehmen" [Far08] wie Texas Instruments, Siemens usw. ist und 2002 gegründet wurde. Seit dem ersten Entwurf von ZigBee wurden verschiedene Versionen des Standards veröffentlicht. Die erste Version *ZigBee 2004* wurde von der Version *ZigBee 2006* abgelöst, welche zu dieser rückwärtskompatibel ist. Danach wurden zwei neue Versionen veröffentlicht: *ZigBee 2007* und *ZigBee 2007 Pro*. Beide neuen Versionen führen neue Funktionalität ein. Der Unterschied zwischen den beiden Versionen ist, dass im Gegensatz zu der Version *ZigBee 2007* die Version *ZigBee 2007 Pro* nicht mehr kompatibel zu den vorherigen Versionen ist. Das führt in der Praxis zu Verwechslungen und Problemen. Für diese Arbeit wird die Version 2006 betrachtet, da die für die spätere Implementierung verwendete Hardware-Plattform nur diese Version unterstützt. Die folgenden Aussagen zu ZigBee und IEEE 802.15.4 basieren, wenn nicht anderes angegeben, auf [Far08], [Ada06] und [All08].

2.2.1 IEEE 802.15.4

Der IEEE 802.15.4-Standard definiert die physikalische Schicht und die Sicherungsschicht („MAC-Schicht") nach dem OSI-Modell für ein „Low-Rate Wireless Personal Area Network" (*LR-WPAN*). Als Funkfrequenzen sind die Bereiche 868/915 MHz und 2,4 GHz vorgesehen, in denen unterschiedlich viele Kanäle und Bandbreiten möglich sind. Das 2,4 GHz-Band kann weltweit eingesetzt werden und ist in 15 Kanäle unterteilt, die eine Bandbreite von 250 kbps ermöglichen. IEEE 802.15.4 ist für Geräte oder Anwendungen gedacht, die selten aktiv sind, nur wenig Daten übertragen müssen und sich die meiste Zeit in energiesparenden Zuständen befinden.

Als Zugriffsverfahren auf den Kanal wird „Carrier Sense Multiple Access mit Collision Avoidance" (*CSMA-CA*) verwendet, bei dem Teilnehmer vor dem Senden überprüfen, ob der Kanal frei ist. Optional kann auf der MAC-Schicht aufbauend ein Beacon-basiertes Verfahren eingesetzt werden, bei dem ein Koordinator eine Struktur vorgibt, die in Zeitschlitze unterteilt ist. Einige dieser Schlitze können Teilnehmern exklusiv zugewiesen werden (*GTS* - Guaranteed Time Slots), was

einer Bandbreitengarantie entspricht und damit gewisse Echtzeiteigenschaften ermöglicht.

Der IEEE-Standard sieht zwei verschiedene Typen von physikalischen Geräten vor, die unterschiedliche Rollen im Netzwerk übernehmen: Das „Full Function Device" (*FFD*), welches über die volle Funktionalität des Standards verfügt und mit allen anderen Teilnehmern kommunizieren kann, sowie das „Reduced Function Device" (*RFD*), das für einfache Anwendungen gedacht ist, nur eine Teilmenge des Standards implementiert und weniger Ressourcen benötigt. Dieser Typ von Geräten befindet sich die meiste Zeit in energiesparenden Schlafzuständen und kann immer nur mit einem FFD kommunizieren.

In einem IEEE 802.15.4-Netzwerk muss es ein Gerät geben, das das Netzwerk aufbaut und verwaltet: den „PAN-Coordinator" (*PAN* - Personal Area Network). Dieser muss ein FFD sein und formt und kontrolliert das Netzwerk . Des Weiteren weist er den anderen Teilnehmern netzinterne Adressen zu. Ein PAN wird durch einen eindeutigen Identifizierungschlüssel referenziert, der durch den PAN-Coordinator bei der initialen Einrichtung des Netzwerkes festzulegen ist. Im Fall des Beacon-basierten Zugriffsverfahrens ist der PAN-Coordinator dafür zuständig, die Knoten im Netzwerk über den Versand von Beacons zu synchronisieren.

Jedes IEEE 802.15.4 Gerät hat eine weltweit eindeutige von dem Hersteller zugewiesene 64 Bit-Adresse, ähnlich der MAC-Adresse von Ethernetadaptern. Nach dem Beitritt zu einem Netzwerk weist der PAN-Coordinator einem Gerät eine in seinem Netzwerk gültige 16 Bit-Adresse zu, damit im weiteren weniger Adressdaten übertragen werden müssen.

In einem IEEE 802.15.4 Netzwerk-Frame können maximal 104 Byte Nutzdaten übertragen werden. Die Integrität der Daten wird über CRC-Felder in jeder Nachricht sichergestellt.

2.2.2 ZigBee-Protokollarchitektur

Der ZigBee-Standard setzt auf dem beschrieben IEEE 802.15.4-Standard auf und ergänzt die Protokollschichten bis zur Anwendung. Basierend auf den beiden Typen von Geräten des IEEE 802.15.4 legt ZigBee drei mögliche Rollen fest, die diese Geräte in einem ZigBee-Netzwerk einnehmen können: „ZigBee-Coordinator", „ZigBee-Router" und „ZigBee-End Device". Der ZigBee-Coordinator entspricht dabei dem „Coordinator" des IEEE 802.15.4-Standards und übernimmt

auch dessen Aufgaben. Der ZigBee-Router muss aus einem FFD bestehen und hat die Aufgabe, ein Mesh-Netzwerk mit anderen Routern und dem Coordinator aufzubauen und Nachrichten weiterzuleiten. Ein ZigBee-End Device kann aus einem FFD oder einem RFD bestehen, muss aber nur die minimale Untermenge des ZigBee-Standards implementieren und kann nur mit genau einem Router oder dem Coordinator kommunizieren. Sofern nicht das Beacon-basierte Zugriffsverfahren verwendet wird, kann sich ein End Device die meiste Zeit in energiesparenden Schlafzuständen befinden und muss nur aktiv werden, wenn es Daten zu übertragen hat. Coordinator und Router hingegen müssen ständig aktiv sein, da jederzeit weiterzuleitende Nachrichten eintreffen können.

ZigBee definiert zwei aufeinander aufbauende Protokollschichten, den „Network Layer" (*NWK*) und den „Application Layer" (*APL*), welche wieder aus Unterschichten und Komponenten bestehen. Die NWK-Schicht baut direkt auf der MAC-Schicht von IEEE 802.15.4 auf. Abbildung 2.1 zeigt eine vereinfachte Darstellung der Schichten und Komponenten, wie sie im Standard beschrieben sind.

Der ZigBee-Standard definiert eine Reihe von Vorgaben und Verfahren für die Verschlüsselung von Daten und Authentifizierung von Teilnehmern im Netzwerk. Sicherheit in einem Sensornetz und vor allem für die Hausautomation bei gleichzeitigen minimalen Ressourceneinsatz ist ein wichtiges eigenständiges Thema und wird aber in dieser Arbeit nicht weiter betrachtet.

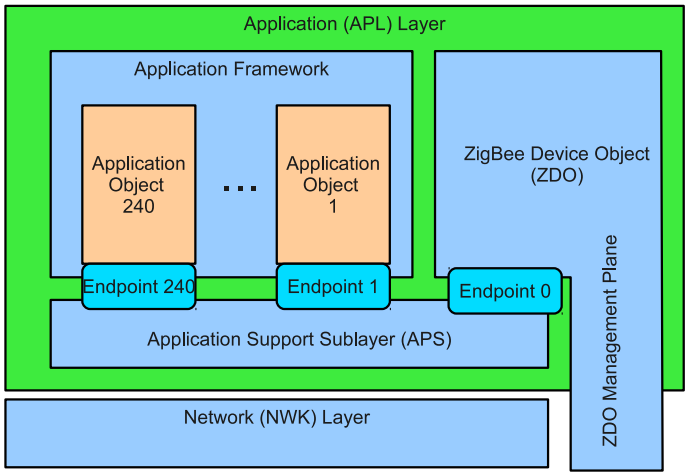

Abbildung 2.1: ZigBee-Protokollschichten (nach [All08])

2.2.2.1 Network Layer – NWK

Die NWK-Schicht (Siehe Abbildung 2.1) von ZigBee hat zwei Aufgaben, den Transport von Daten und das Verwalten des Netzwerkes. Die Verwaltung des Netzwerkes besteht unter anderem aus den Aufgaben, nach Netzwerken zu suchen, neue Netzwerke aufzubauen, Geräte Netzwerken beitreten zu lassen und diesen neuen Geräten eine der Rollen zugewiesen,

Für den Transport der Daten zwischen beliebigen Geräten im Netzwerk ist das Finden, Aufbauen und Erneuern von Routen notwendig, was Aufgabe der NWK-Schicht ist. Der Coordinator und die übrigen Knoten bilden dazu ein Mesh-Netzwerk. Daten können als Uni-, Multi- oder Broadcast versendet werden. Ein Gerät kann mehreren Multicast-Gruppen zugeordnet werden, die jeweils über eine eigene 16 Bit-Adresse referenziert werden. Multicast wird in Abschnitt 2.2.2.3 genauer beschrieben.

2.2.2.2 Application Layer – APL

Der Application Layer von ZigBee ist die oberste Schicht des ZigBee-Protokolls, auf dem die Anwendungen aufsetzen und besteht aus drei Teilen: Dem „Application Support Sublayer" (*APS*), dem „ZigBee Device Object" (*ZDO*) und dem „Application Framework". In Abbildung 2.1 werden diese Teile im Aufbau des Protokoll-Stacks dargestellt. Der APS ist eine Zwischenschicht und stellt dem ZDO und dem Application Framework Datenübertraguns- und Management-Dienste zur Verfügung. Das ZDO kapselt die Funktionalität, die benötigt wird, um die ZigBee-Schichten zu initialisieren, andere Geräte im Netzwerk zu finden und Anwendungen mit entfernten Diensten zu verbinden. Die Anwendungen greifen über „Application Objects", die einzeln über sogenannte „Endpoint"-Adressen adressierbar sind und von denen es maximal 240 auf einem Gerät geben kann, auf den ZigBee-Protokoll-Stack zu.

Application Framework und ZigBee-Anwendungsprofile

Als Basis für die Entwicklung von Anwendungen sieht ZigBee Anwendungsprofile vor, welche über einen 16 Bit-Wert („Profile Identifier") identifiziert werden, der nach Prüfung des Profils durch die ZigBee-Allianz vergeben wird. Damit soll die Interoperabilität von Geräten verschiedener Hersteller ermöglicht werden.

Ein Anwendungsprofil von ZigBee besteht aus zwei Arten von Komponenten: Sogenannten „Clusters" und „Device"-Beschreibungen. Ein „Cluster" wird mittels eines 16 Bit-Wertes (*ClusterID*) identifiziert und besteht aus einer Menge von Attributen, welche ebenfalls jeweils über einen 16 Bit-Wert referenziert werden. Die Attribute stehen für die Daten, die zwischen den Anwendungen ausgetauscht werden sollen. Die „Device"-Beschreibungen werden auch über einen 16 Bit-Wert identifiziert und enthalten Informationen über die Geräte, auf denen die Anwendungen laufen. Bei den Informationen kann es sich beispielsweise um die Rolle oder den Füllstand der Batterie handeln.

Anwendungen können Cluster als „Input-"(Eingangs-) oder „Output-"(Ausgangs-) Cluster realisieren und einem Endpoint zuordnen. Zwischen Eingangs- und Ausgangs-Clustern mit derselben ClusterID können Daten ausgetauscht werden. Der Aufbau einer solchen Verbindung wird „Binding" genannt. Er erfolgt über den Coordinator. Er kann entweder automatisch oder manuell bei der Installation der Geräte in der physikalischen Umgebung erfolgen.

Application-Support Sublayer

Der APS-Sublayer wird von den Application-Objects und dem ZDO genutzt. Er verwaltet die Binding-Informationen, führt die Adressierung der Nachrichten zwischen verbundenen Anwendungen durch, verwaltet die Gruppenkommunikation und sorgt für eine zuverlässige Datenübertragung.

Das Binding hat das Ziel, Anwendungen auf verschiedenen Knoten zu verbinden und danach automatisch die richtige Adressierung der Nachrichten durchzuführen. Dies ermöglicht einer Anwendung nach dem Binding-Vorgang eine indirekte Adressierung der Empfänger ihrer Nachrichten. Dabei muss die Anwendung die Empfänger nicht kennen, sondern sie verschickt die Nachricht ohne Adresse, die entweder auf dem Knoten oder durch den Coordinator hinzugefügt bzw. richtig weitergeleitet wird. Die Gruppenkommunikation wird im nächsten Abschnitt genauer betrachtet und die Funktionsweise der zuverlässigen Datenübertragung ist für diese Arbeit nicht weiter relevant.

2.2.2.3 Gruppenkommunikation

Die vom ZigBee-Standard vorgesehene Gruppenkommunikation hat eine ähnliche Semantik wie ein Multicast und die Funktionalität wird von der NWK-Schicht

und dem APS-Sublayer ermöglicht. Der APS-Sublayer verwaltet auf jedem Gerät eine Tabelle, in der die Zugehörigkeit einer Anwendung (über die Endpoint-ID) zu einer Gruppe registriert wird. Ein Gerät bzw. auch eine Anwendung kann dabei Mitglied in mehreren Gruppen sein.

Nachrichten können von jeder Anwendung an Gruppen geschickt werden, auch wenn sie nicht selber Mitglied der Gruppe sind. Die Nachrichten an eine Gruppe werden an alle Geräte mit Gruppenmitgliedern geschickt und die Zustellung an die richtige Anwendung erfolgt lokal über die Endpoint-ID. Beim Routing der Nachrichten wird aber unterschieden, ob die Nachricht von einem Mitglied oder einem externen Gerät geschickt wird. Dies wird „Member-Mode" (Mitgliedsmodus) bzw. „Nonmember-Mode" genannt. Im letzeren Fall wird die Nachricht als Unicast zu einem Mitglied der Gruppe geleitet und von dort als Broadcast weiter verteilt. Wird die Nachricht von einem Gruppenmitglied verschickt, erfolgt die Übertragung sofort als Broadcast auf der Netzwerkebene. Dieser Broadcast unterscheidet sich von dem Allgemeinen dadurch, dass er eigene Zähler für die maximale Anzahl der „Hops" hat.

2.3 OMG Data Distribution Service

2.3.1 Einführung

Mit dem „Data Distribution Service" (*DDS*) hat die „Object Management Group" (*OMG*) einen offenen Standard für eine datenzentrierte Middleware mit Echtzeiteigenschaften, der auf dem Publish-Subscribe-Paradigma basiert, publiziert. Der Standard definiert die Anwendungsschnittstelle („Application Programming Interface", *API*), die eine Middleware nach dem DDS-Standard Anwendungen bieten muss [OMG07].

Die OMG ist ein offenes, nicht-kommerzielles Industriekonsortium von IT-Unternehmen, welches Standards für die Entwicklung und Kompatibilität von Software entwickelt und veröffentlicht. Unter anderem ist die OMG für die „Unified Modeling Language" (*UML*) und die „Common Object Request Broker Architekture" (*CORBA*) verantwortlich [OMG09].

Der DDS-Standard geht auf die kommerzielle Implementierungen von datenzentrierten Middleware-Lösungen der Firmen RTI (RTI DDS) und PrismTech (OpenSplice DDS) zurück, welche 2003 zusammen einen Vorschlag für einen offenen

DDS-Standard bei der OMG eingereicht haben. Dieser Vorschlag orientierte sich stark an der API der Implementierung von RTI. Im April 2004 wurde die erste Version fertiggestellt und seitdem kontinuierlich weiterentwickelt. Derzeit ist die Version 1.2 vom Januar 2007 aktuell; die Version 1.3 ist in der Entwicklung. Neben dem DDS-Standard für eine API wurde 2006 ein Standard für ein Kommunikationsprotokoll veröffentlicht. Das Protokoll selber heißt „Real-Time Publish-Subscribe" (*RTPS*), der OMG-Standard in dem es definiert ist, ist die „DDS Interoperability Wire Protocol Specification" (*DDSi*), welche aktuell in der Version 2.1 vom Januar 2009 vorliegt. Mit DDSi soll es möglich sein, Implementierungen von DDS unterschiedlicher Hersteller in einem System gemeinsam betreiben zu können [OMG09].

Der Zweck einer DDS-Middleware ist es, den Datenaustausch zwischen Anwendungen in einem verteilten System für den Entwickler zu vereinfachen. DDS verbindet hierzu die Datenzentriertheit mit dem Publish-Subscribe-Paradigma, so dass für die Anwendungen nur die Daten und ihre Eigenschaften im Vordergrund stehen.

Ein DDS-System baut auf einem globalen gemeinsamen Datenraum auf, in dem Anwendungen typisierte Daten als *Publisher* veröffentlichen können, oder als *Subscriber* die Option haben, solche Daten zu abonnieren, wenn sie diese benötigen. Die jeweiligen typisierten Daten sind sogenannten *Topics* zugeordnet, über deren Bezeichnung die Adressierung erfolgt.

Der DDS-Standard definiert eine Reihe von Quality-of-Service-Richtlinien, welche von der Parametrisierung ebenfalls datenzentriert ausgelegt sind und bei entsprechender Implementierung in einer Middleware Zusicherungen bezüglich der Konsistenz oder Redundanz von Daten sowie zeitliche Schranken für die Zustellung über das Netzwerk („Echtzeit") möglich machen.

Die Anwendungsschnittstelle ist in DDS plattformunabhängig („Plattform Independent Model", *PIM*) in der Schnittstellenbeschreibungssprache der OMG („Interface Defintion Language", *OMG IDL*) definiert und ist objektorientiert strukturiert. Der Standard gibt mit der Schnittstelle auch die Funktionen vor, die eine DDS-Middleware Anwendungen zur Verfügung zu stellen hat. Für OMG IDL gibt es von der OMG definierte Abbildungen für verschiedene Programmiersprachen; werden diese auf die PIM-API von DDS angewandt, ergeben sich die plattformspezifischen Modelle („Plattform Specific Model", *PSM*). Durch das einheitliche, abstrakte Modell können Anwendungen, die für eine DDS-Implementierung ent-

wickelt wurden, mit überschaubarem Aufwand auf andere Systeme portiert werden.

Der Zugriff einer Anwendung auf das DDS-System erfolgt über die jeweiligen Topics, welchen ein spezifischer Datentyp zugeordnet ist. DDS sieht vor, dass Anwendungsentwickler diese Datentypen definieren und daraus die Programmcode-Stubs zur Verbindung mit der Middleware generiert werden. Hierfür wird in der Regel ebenfalls OMG IDL verwendet, aus dem ein implementierungsspezifischer IDL-Compiler den Code für die Implementierung der Anwendungsschnittstellen erzeugt, über den auf die spezifischen Datentypen zugegriffen werden kann.

Die API des DDS-Standards besteht aus zwei Schichten mit unterschiedlichen Abstraktionsebenen, auf die Anwendungen zugreifen können. Das „Data-Centric Publish-Subscribe" (*DCPS*) bildet die untere Schicht, welche eine DDS-Implementierung minimal implementieren muss. Diese Schicht erbringt den gesamte Funktionsumfang von DDS, wobei die Abstraktion das Lesen und Schreiben von einzelnen Daten unter einem Topic ist. Der „Data Local Reconstruction Layer" (*DLRL*) liegt über dem DCPS und bietet Anwendungen eine einfachere, objektorientierte Abstraktionsebene für den Zugriff auf DDS. DLRL ist vom DDS-Standard als optional definiert und wird für diese Arbeit nicht benötigt.

2.3.2 DCPS

In Abbildung 2.2 ist der grundsätzliche Aufbau eines DDS-Systems mit den wichtigsten Komponenten dargestellt. Eine grundlegende Idee, auf der DDS basiert, ist die des „globalen Datenraums" (Global Data Space), in dem Daten veröffentlicht und abonniert werden können. Ein Topic bildet eine Assoziation zwischen einem bestimmten Datentyp, einer Menge von QoS-Eigenschaften und einem eindeutigen Namen und ist damit die Einheit der auszutauschenden Daten. Die *Domain* stellt einen Namensraum in dem globalen Datenraum dar, in dem der Name des Topics eindeutig ist und auf die eine Anwendung über einen *DomainParticipant* zugreifen kann [SCv08]. relational

Es wird unterschieden zwischen dem Veröffentlichen und dem Abonnieren von Daten. Für jedes Topic gibt es spezifische *DataWriter* und *DataReader*, die die direkte Sende- und Empfangsschnittstellen von Daten für die Anwendungen repräsentieren und mit dem IDL-Compiler spezifisch für den jeweiligen Datentypen des Topics erzeugt werden. In einem DomainParticipant fassen der *Publisher* und

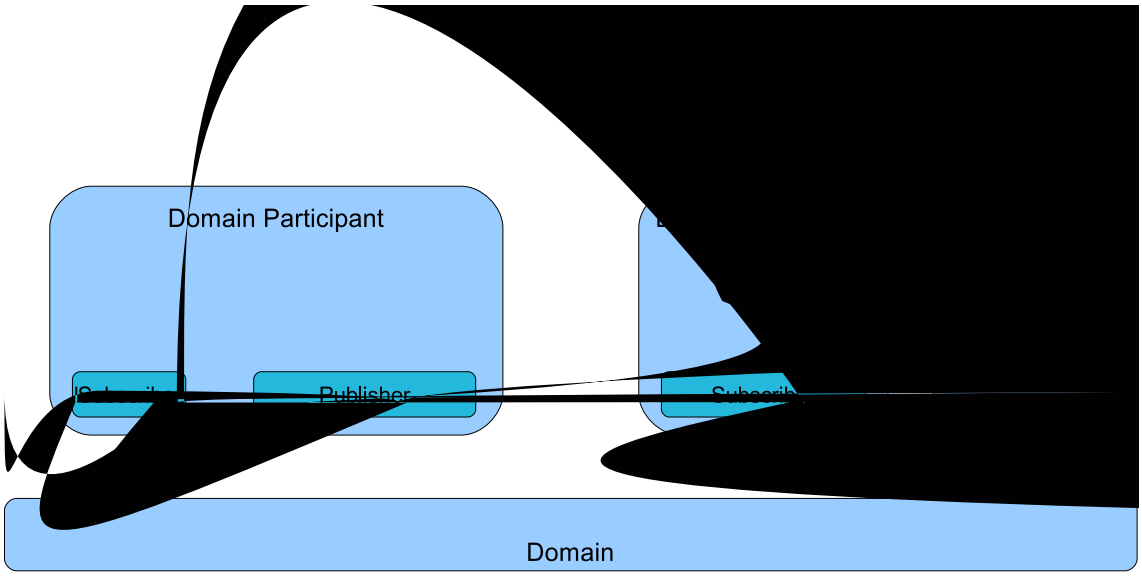

Abbildung 2.2: Konzeptioneller Aufbau von DDS

der *Subscriber* jeweils entsprechend die DataWriter und DataReader zusammen. Die DDS-Objekte werden von den Anwendungen instanziiert und abhängig davon, wie sie das DDS-System verwenden soll, verbunden. Anwendungen, die nur Daten veröffentlichen wollen, benötigen einen DomainParticipant, Publisher und für jedes Topic einen DataWriter. Daten, die eine Anwendung in einen DataWriter schreibt, werden an alle DataReader geschickt, die mit demselben Topic assoziiert sind.

Eine Anwendung hat drei Möglichkeiten, um Daten von einem DataReader zu lesen: das Pollen, blockierendes Lesen, für das Bedingungen definiert werden können und die Registrierung von Callback-Funktionen, die aufgerufen werden, wenn ein Datum eintrifft.

2.3.2.1 Objektmodell

Die API von DDS ist objektorientiert, und die Komponenten eines DDS-Systems sind als Klassen definiert, wovon die wichtigsten in Abbildung 2.3 als UML-Diagramm dargestellt werden. Die bisher beschrieben Komponenten wie Topic, DataWriter, DataReader usw. finden sich darin als Klassen wieder. Die Klasse `Entity` ist die abstrakte Basisklasse für die Klassen, die QoS, Rückruf-Funktionen (*Listener*) und Status-Bedingungen (*Status Condition*) benötigen. Den QoS-Klassen sind spezifische QoS-Richtlinien und -Funktionen zugeordnet, sie implementie-

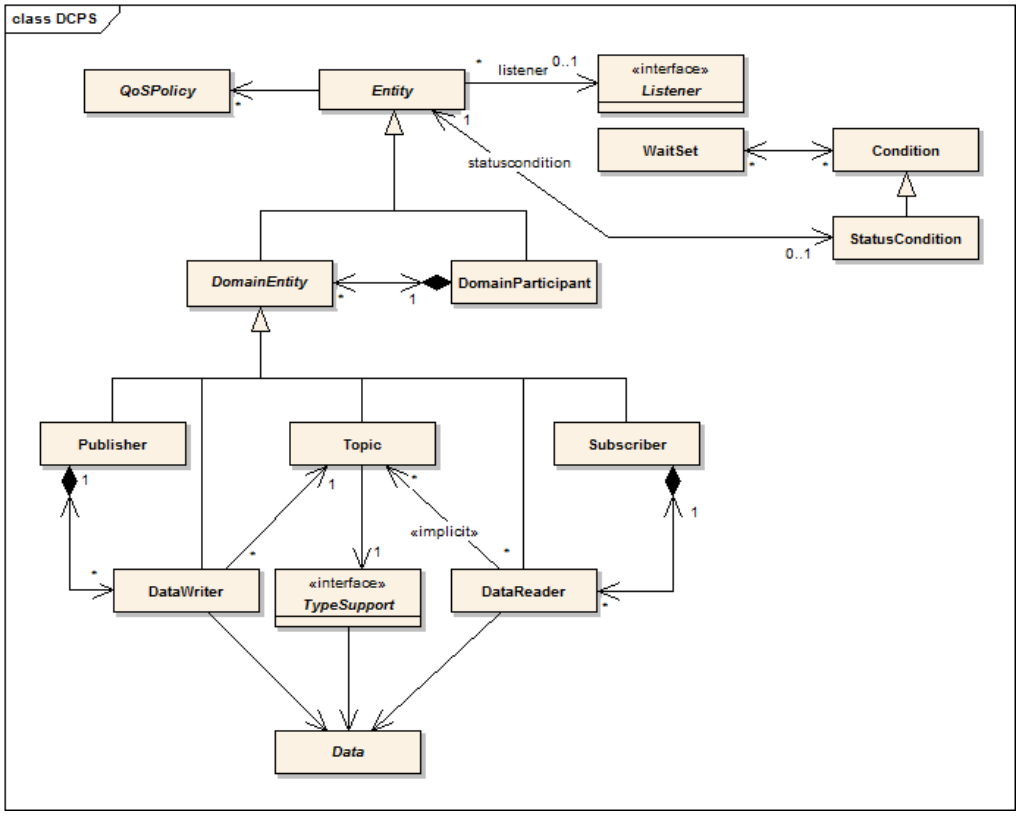

Abbildung 2.3: UML-Klassenmodell von DDS

ren einen großen Teil der Funktionalität von DDS. Das Interface `Listener` stellt die Schnittstelle für die Rückruf-Funktionen dar und zusammen mit der Klasse `StatusCondition` für die Status-Bedingungen ermöglicht es der Anwendung, informiert zu werden, wenn von ihr definierte Bedingungen oder Ereignisse auftreten. Die Klasse `DomainParticipant` unterscheidet sich von den anderen Klassen dadurch, dass dass alle anderen DDS-Entitäten an sie angebunden sind und sie die Verbindung zu einer spezifischen Domain darstellt [OMG07].

2.3.2.2 Topics und Datentypen

Die Klasse `Topic` repräsentiert das Topic in DDS und ist somit die Verbindung zwischen DataWriter und DataReader. Bevor Anwendungen eine DDS-Middleware verwenden können, müssen die auszutauschenden Datentypen deklariert und danach in den Anwendungen selber den Topics zugeordnet werden. Der Datentyp, der einem Topic zugeordnet wird, setzt sich in der Regel aus mehreren anderen zusammen. Die Modellierung kann objektorientiert- oder relational erfolgen; beides wird von DDS unterstützt.

Für ein relationales Schema muss der Datentyp eines Topics Schlüssel enthalten, über die Instanzen (Gruppen mit demselben Schlüssel) von einzelnen Datensätzen identifiziert werden können. Diese Schlüssel werden in DDS als *Keys* bezeichnet und können auch über mehrere Topics als „Foreign-Keys" eingesetzt werden. Ein über einen *Key* referenzierte Gruppe von Datensätzen wird in DDS *Instance* genannt. Ein einzelner Datensatz unabhängig von dem optional enthaltenen *Key* wird als *DataSample* bezeichnet und ist das „Stück Information", das eine Anwendung einem DataWriter übergibt oder von einem DataReader lesen kann. Werden bei der Deklaration der Topic-Datentypen keine *Keys* verwendet, sind alle Datensätze eines Topics gleich und werden entsprechend an den DataReader ausgeliefert.

Für die Deklaration der Topic-Datentypen wird in der Regel OMG IDL verwendet, für die Angabe der *Keys* geht aber jede DDS-Implementierung ihren eigenen Weg. Im Folgenden soll ein kurzes Beispiel für einen Datentyp namens *Foo* gegeben werden; für die Definition des *Keys* wird die Syntax von OpenSplice verwendet:

```
1 module Beispiel {
2     struct Foo {
3         short temperatur;
4         long long zeitpunkt;
5         unsigned short sensorID;
6     };
7 #pragma keylist Foo zeitpunkt sensorID
8 };
```

Quellcode 2.1: OMG IDL-Beispiel

Mit dem Schlüsselwort `module` wird bei OpenSplice ein Namensraum für den Programmcode definiert. Dieser wird durch den IDL-Compiler für jede Plattform spezifisch umgesetzt, beispielsweise bei C++ mit dem Sprachkonstrukt `namespace` oder `package` in Java.

Der Datentyp *Foo* enthält drei weitere Datentypen: *temperatur*, *zeitpunkt* und *sensorID*, wobei die beiden letzteren in Zeile 7 als *Keys* definiert werden. Eine Instanz vom Datensätzen ist damit über den Zeitpunkt der Messung und den Temperatursensor identifizierbar.

Aus der Deklaration der Datentypen, in unserem Beispiel der des Datentyps *Foo*, werden durch den IDL-Compiler die Programmcode-Stubs für die Anwendung generiert. Dies sind in Abbildung 2.3 die Klassen `FooTypeSupport` für den Datentyp *Foo*, sowie `FooDataWriter` und `FooDataReader` für den spezifischen DataWriter und DataReader für dieses Datentyp, die von den Klassen `TypeSupport`, `DataWriter` und `DataReader` erben. Damit ist dieser Datentyp aber noch keinem bestimmten Topic zugeordnet, dies geschieht erst durch die Anwendung, wenn ein *Topic* mit dem generierten *FooTypeSupport* verbunden wird. Damit ist es möglich, auch mehrere unterschiedliche *Topics* mit demselben Datentyp zu verwenden.

Neben der Basisklasse `Topic` gibt es zwei weitere spezielle Klassen für ein Topic: `MultiTopic` und `ContentFilteredTopic`. Diese zeichnen sich dadurch aus, dass ihnen kein in IDL definierter Datentyp, sondern andere Topics zugeordnet sind. Sie fassen mehrere Topics oder Teile von Topics zusammen und eine Anwendung kann über eine SQL-ähnliche Sprache die vorzunehmende Auswahl definieren.

ContentFilteredTopic: Wenn eine Anwendung nicht an allen Daten eines Topics interessiert ist, sondern nur an denen, deren Werte bestimmte Bedin-

gungen erfüllen, kann sie diese mit einem `ContentFilteredTopic` herausfiltern. Dies entspricht der `WHERE`-Klausel in SQL.

MultiTopic: Wenn mehrere *Topics* oder Teile als neues *Topic* zusammengefasst werden sollen, kann ein `MultiTopic` verwendet werden. Es können auch Filter auf den Werten der Datentypen definiert werden. Die Auswahl der Topics und der Teile ihrer Datentypen geschieht ebenfalls mit der SQL-ähnlichen Sprache. Die `FROM`-Klausel referenziert die Topics und die Teile der relevanten Datentypen werden nach der `SELECT`-Klausel angegeben. Die Filter auf die Werte werden wie bei SQL nach der `WHERE`-Klausel aufgeführt. Wenn die ausgewählten Datentypen sich unterscheiden, müssen sie vereinheitlicht werden; dies geschieht analog zu der der `NATURAL JOIN`-Operation von SQL.

Eine Demonstration dieser SQL-ähnlichen Sprache sei mit dem folgenden Beispiel gegeben:

```
SELECT temperatur AS FehlerTemp
FROM 'MaschinenTemperatur'
WHERE temperatur > 200
```

Hierbei wird angenommen, dass der Datentyp *Foo* aus dem vorhergehendem Beispiel einem Topic mit dem Namen *MaschinenTemperatur* zugewiesen ist. Ein *ContentFilteredTopic* mit der dargestellten Bedingung würde dann alle Temperaturwerte, die größer sind als 200, enthalten, und eine Anwendung könnte auf den Wert im Datensatz über das Attribut *FehlerTemp* zugreifen.

2.3.3 Quality of Service

Die Spezifikation von Quality-of-Service-Richtlinien (QoS) nimmt im DDS-Standard eine zentrale Rolle ein; er definiert hierzu eine Gruppe von QoS-Klassen, welche von der abstrakten Klasse `QosPolicy` erben. Sie sind bestimmten Basisklassen wie Topic, DataReader, DataWriter etc. zugeordnet, die über die QoS-Parameter konfiguriert und damit in ihrem Verhalten beeinflusst werden.

In der Tabelle 2.1 sind alle Klassen der QoS-Richtlinien von DDS mit ihrer Zuordnung zu den Basisklassen aufgeführt. Einige QoS-Parameter können im laufenden Betrieb geändert werden; dies ist in der Spalte „Änderbar” vermerkt.

QoS-Richtlinie	Zuord.	RxO	Änderbar
User Data	D, P, DR, DW	nein	ja
Topic Data	T	nein	ja
Group Data	Pub, Sub	nein	ja
Durability	T, DR, DW	ja	nein
Durability Service	T, DW	nein	nein
Presentaion	Pub, Sub	ja	nein
Deadline	T, DR, DW	ja	nein
Latency Budget	T, DR, DW	ja	ja
Ownership	T, DR, DW	ja	nein
Ownership Strength	DW	–	ja
Liveliness	T, DR, DW	ja	nein
Time Based Filter	DR	–	ja
Partition	Pub, Sub	nein	ja
Reliability	T, DR, DW	ja	nein
Transport Priority	T, DW	–	ja
Lifespan	T, DW	–	ja
Destination Order	T, DR, DW	ja	nein
History	T, DR, DW	nein	nein
Resource Limits	T, DR, DW	nein	nein
Entity Factory	DPF, DP, Pub, Sub	nein	ja
Writer Data Lifecycle	DW	–	ja
Reader Data Lifecycle	DR	–	ja

Tabelle 2.1: Liste der QoS-Richtlienen im DDS-Standard

Die Bedeutung der QoS-Parameter unterscheidet sich je nach Basisklasse und deren Rolle als Datenquelle oder -senke. Die Parameter auf Seite der Datenquelle definieren, welche Zusicherungen die Anwendungen oder die Middleware bezüglich der Qualität machen können. Auf der Seite der Datensenken hingegen definieren sie, welche Anforderungen die Anwendungen an die Qualität der Daten oder des Dienstes allgemein stellen. Die Möglichkeit zum Aufbau einer „Subscription" zwischen DataReader und DataWriter kann neben demselben Topic auch von den gesetzten QoS-Parametern abhängig gemacht werden. Beispielsweise legt die QoS-Richtlinie *Reliabilty* die Verlässlichkeit der Datenübertragung fest. Ein DataWriter mit der QoS-Einstellung „Best Effort" (Unzuverlässig) kann Daten nur ohne verlässliche Übertragung bereitstellen. Hat ein DataReader den QoS-Parameter „Reliable" (Zuverlässig) gesetzt, dann kann keine Subscription zustande kommen. In dem Fall, dass der DataReader „Best-Effort" verlangt und der DataWriter „Reliable" anbietet, wäre eine Subscription möglich, da die Bereitstellung einer niedrigeren Servicequalität immer möglich ist. Der DDS-Standard definiert für alle QoS-Parameter, die dies betrifft, eine solche Ordnung der Service-Qualität. In der Tabelle 2.1 sind die QoS-Richtlinien, bei denen diese DDS-Funktionalität zu tragen kommt in der Spalte „RxO"[1] markiert.

Die QoS-Richtlinien von DDS sind primär datenzentriert ausgelegt, das bedeutet, dass sie sich auf die Qualität der Bereitstellung von Daten und nicht auf die Verbindungseigenschaften von DDS-Anwendungen beziehen.

2.3.4 RTPS

Das „Real-Time Publish-Subscribe"-Protokoll definiert eine Datenübertragung auf Ebene des Netzwerkes mit dem Ziel, DDS-Implementierungen verschiedener Hersteller untereinander interoperabel zu machen. Dieses Protokoll ist aber für DDS-Implementierungen optional. Beispielsweise bietet OpenSplice DDS die Auswahl zwischen einem proprietären Protokoll und RTPS.

Wie auch der DDS-Standard nimmt die Spezifikation von RTPS Anleihen bei dem OMG-Standard CORBA. Als Datenkodierungsformat wird beispielsweise die „Common Data Representation" (*CDR*) verwendet, welche die Datentypen von OMG IDL verarbeitet. RTPS kann als Protokoll auf einem verbindungslosen, unzuverlässigen Kommunikationssystem aufsetzen und Multicast einsetzen. Implizit

[1] RxO steht im DDS-Standard für „Requested / Offered"

wird dabei die Verwendung von UDP/IP angenommen, was bedeutet, dass Routing nicht Bestandteil des RTPS-Protokolls, sondern von IP ist.

RTPS unterstützt verschiedene QoS-Richtlinien, welche auf die von DDS abgebildet werden können. Dabei hat es den Anspruch, auch Zusicherungen bezüglich Echtzeiteigenschaften machen zu können [CF09]. Die Spezifikation erfüllt die Anforderungen, die der DDS-Standard stellt; Protokoll-Implementierungen können aber auch unabhängig von DDS sein und für andere Anwendungen eingesetzt werden.

Die Spezifikation des Protokolls umfasst vier Module:

- Die Objektstruktur, die die Schnittstelle zur DDS-Anwendung festlegt („Structure Module")
- Der Aufbau des Nachrichtenformats (Message Module)
- Das Protokollverhalten („Behavior Module")
- Das Auffinden (*Discovery*) von Teilnehmern, Diensten und Daten (Discovery Module)

Die Objekte von RTPS lassen sich direkt Objekten von DDS zuordnen und werden entsprechend auch von diesen verwendet. Daten werden dabei von „Writer" zu „Reader"-Objekten geschickt und diese RTPS-Objekte entsprechen dabei dem DataWriter bzw. DataReader eines Topics.

Das RTPS-Nachrichtenformat sieht eine Nachricht („Message") vor, welche aus einem Header und Teilnachrichten („Submessage") besteht.

Kapitel 3

Analyse

3.1 Anforderungen an eine Middleware für Sensornetze

3.1.1 Aufgabenstellung

Wie bereits in der Einleitung und den Grundlagen (2.1) dargelegt wurde, sind Sensornetze für eine Vielzahl an interessanten Aufgaben geeignet, die ohne diese Entwicklung nicht oder nur mit viel größeren Kosten lösbar wären. Die Entwicklung von Software für Sensornetze wird durch die Hardware-Beschränkungen der drahtlosen Sensorknoten und die Abhängigkeit von einer Energieversorgung erschwert. Aufgabe dieser Arbeit ist es daher, eine Middleware-Lösung zu entwickeln, die den Entwickler bei der Konzeption, Implementierung, Erweiterung und ggf. Portierung von Anwendungen für Sensornetze unterstützt bzw. diese vereinfacht und verallgemeinert.

Für eine solche Middleware-Lösung sind im Rahmen dieser Arbeit zwei Bereiche zu untersuchen. Zum einem ist dies die Konzeption einer Middleware für Sensornetze und die Frage wie diese den Besonderheiten von drahtlosen Sensornetzen gerecht werden kann. Daneben werden die Möglichkeiten für Werkzeuge und Prozesse untersucht, die geeignet sind, den Anwendungsentwickler bei dem Entwurf von Anwendungen, aufbauend auf der zu entwickelnden Middleware, zu unterstützen und es ermöglichen, die Middleware an den jeweiligen Anwendungsfall anzupassen und auf die Bedürfnisse von Sensornetzen zu optimieren.

Gerade die Heterogenität von Sensornetzen und die Notwendigkeit, Anwendungen spezifisch auf die jeweilige Plattform zu optimieren, erschwert es, dass Anwendungen einfach portiert werden können. In [LVCA+07] wird beschrieben, wie mittels modellgetriebener Softwareentwicklung (*MDSD* Model-Driven Software Development) solchen Problemen bei der Entwicklung von Anwendungen für Sensornetze begegnet werden kann. Wie in der Einleitung beschrieben, werden MDSD-Ansätze auch im Bereich der eingebetteten System für analoge Problem- und Anforderungsfelder verwendet. Daher soll untersucht werden, ob dieser Ansatz auch im Rahmen dieser Arbeit für die Anpassung einer Middleware Anwendung finden kann.

Für die Konzeption der Middleware ist es notwendig zu untersuchen, welche Anforderungen drahtlose Sensornetze und darauf laufende Anwendungen an die Middleware stellen können. Für die Bestimmung der möglichen Anforderungen wird zuerst der Problembereich für einen allgemeinen Anwendungsfall eines drahtlosen Sensornetzes beschrieben, aus dem die ersten Anforderungen an die Middleware und die unterstützenden Werkzeuge abgeleitet werden. Diese werden danach aktueller Literatur über Sensornetze und dafür geeignete Middleware gegenübergestellt und ggf. ergänzt. Danach erfolgt die Auswahl einer Untermenge, die die zu berücksichtigenden Anforderungen für diese Arbeit bildet.

3.1.2 Allgemeiner Anwendungsfall für eine Middleware für Sensornetze

Sensornetze wurden in den Grundlagen 2.1 beschrieben; ihre Eigenschaften und ihr Aufbau unterscheiden sich je nach Anwendung. Der im Folgenden beschriebene allgemeine Anwendungsfall für drahtlose Sensornetze soll eine Schnittmenge der vielen Möglichkeiten bilden. Er basiert auf einem Netzwerk aus mehreren Datenquellen und einer oder auch mehreren Datensenken, deren Fähigkeit, untereinander kommunizieren zu können, sich auf Grund von Umwelteinflüssen oder Ausfall einzelner Knoten ständig verändern kann. Abbildung 3.1 zeigt eine grafische Abstraktion dieses Sensornetzes. Es besteht aus einigen unterschiedlichen Datenquellen und einer zentralen Datensenke, die als Gateway zu externen Systemen dient, und deren Knoten über ein drahtloses Mesh-Netzwerk verbunden sind, sowie mehreren kleineren Datensenken in den Sensorknoten, die die notwendigen Daten für interne Regelungsaufgaben empfangen. Aktoren, deren

Steuerung über eine entsprechende Datensenke für Steuerungsbefehle erfolgt, sind Teil einiger Sensorknoten und ermöglichen es dem Sensornetz, mit der Umgebung zu interagieren.

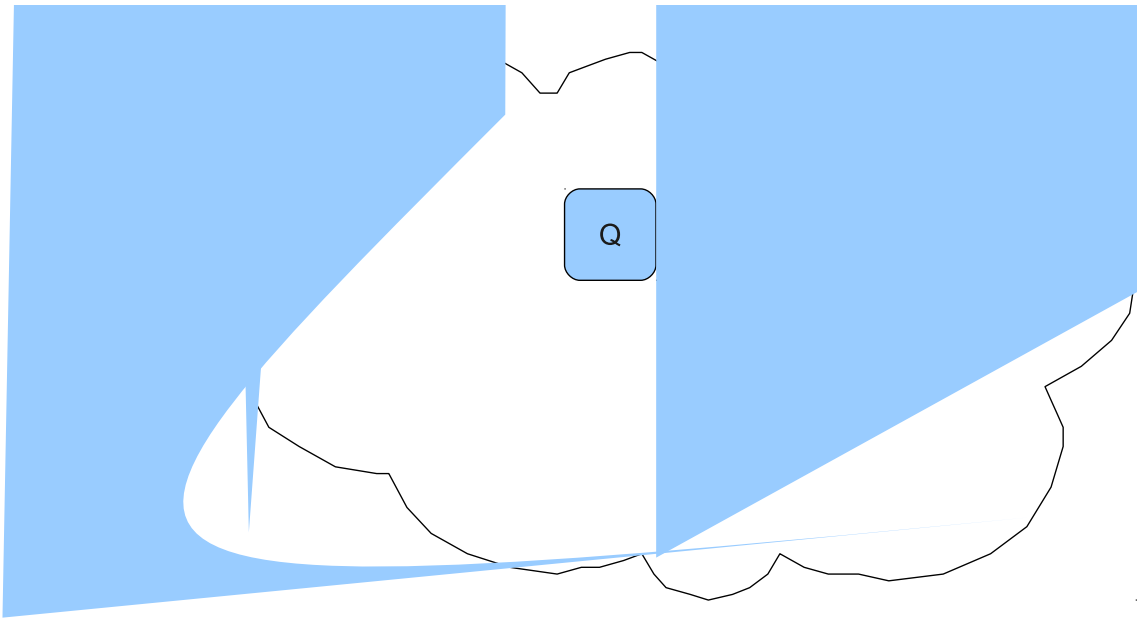

Abbildung 3.1: Ein abstraktes Sensornetz mit Datenquellen, Senken, Aktoren und einem Gateway.

Gateway-Knoten verbinden das Sensornetz mit externen Systemen, welche als Datensenke für die Daten der Sensorknoten oder als Datenquelle für Steuerungsbefehle auftreten. Die Knoten in dem betrachteten Sensornetz sind heterogen, das heißt, sie haben unterschiedliche Sensoren, Aktoren, verbleibende Energiereserven, oder sie basieren auf unterschiedlichen Sensorknoten-Hardware-Plattformen. Das bedeutet, dass sie unterschiedliche Prozessoren und Architekturen verwenden oder über unterschiedlich viel Speicher bzw. Rechenkapazität verfügen. Auch unterscheidet sich die eingesetzte Systemsoftware dieser Knoten, dies betrifft die API der Treiber, Betriebssystemfunktionen etc.. Bezüglich der Energieversorgung kann man die Sensorknoten in zwei Gruppen einteilen: Erstens Knoten, die über eine ständige Energieversorgung verfügen, z.B. weil sie an ein Stromnetz angeschlossen sind, und zweitens autarke Knoten, die über einen endlichen Energievorrat verfügen, mit dem sie über ihre Lebenszeit haushalten müssen, wie es meist bei drahtlosen Sensorknoten der Fall ist. Um wenig Energie zu verbrauchen und aus Kostengründen, verfügen die autarken Sensorknoten

nur über sehr wenig Hardware-Ressourcen wie Speicher, Rechenkapazität oder Bandbreite. Die Lebenszeit des Sensornetzes hängt davon ab, wie lange die autarken Knoten über Energie verfügen.

Das Sensornetz ist veränderlich in seiner Struktur, der Anzahl und Art der Sensorknoten und den Kommunikationsmöglichkeiten mit benachbarten Knoten. Auch kann es mobile Knoten geben, die entweder ohne eigenes Zutun oder aktiv, basierend auf eigenen Entscheidungen bzw. Entscheidungen des Sensornetzes, ihren Standort ändern können. Dennoch soll gewährleistet sein, dass keine wichtigen Daten verloren gehen, sondern zuverlässig von den Datenquellen zu den Datensenken gelangen.

Verschiedene Datenquellen, insbesondere wenn sie sich räumlich nah beieinander befinden, können gleiche oder ähnliche Daten produzieren. Für Datensenken ist es nicht zwangsläufig wichtig von welcher Datenquelle sie Daten beziehen. Des Weiteren können Datensenken unterschiedliche Anforderungen an die Qualität und Verfügbarkeit der Daten haben. Diese Anforderungen sind quantifizierbar und es wird erwartet, dass die Datenquellen, bzw. das Sensornetz diese Anforderungen an die Qualität und Verfügbarkeit von Daten erfüllen können. Daten können zum einem in regelmäßigen zeitlichen Abständen von Datenquellen bereitgestellt oder von Datensenken benötigt werden, zum anderen wenn bestimmte Ereignisse auftreten, oder wenn sie definierte Eigenschaften besitzen.

Software-Werkzeuge unterstützen den Anwendungsentwickler dabei, Anwendungen für das Sensornetz zu planen, zu modellieren und zu implementieren. Diese Werkzeuge kennen die Eigenschaften der Sensorknoten und des geplanten Netzes und optimieren die Komponenten hinsichtlich Anwendungszweck und maximaler Lebenszeit.

3.1.3 Abgeleitete Anforderungen

Aus dem allgemeinen Anwendungsfall lassen sich direkt die folgenden Anforderungen an eine Middleware-Lösung für Sensornetze ableiten:

1. **Minimaler Ressourcenverbrauch der Middleware**
   Autonome Sensorknoten mit wenig Ressourcen.

2. **Ressourcenbewusstsein**
   Ressourcen können unterschiedlich sein, insbesondere verfügbare Energie definiert die Lebensdauer des Systems.

3. **Datenzentrierung**
   Ziel ist, dass Daten von den Datenquellen zu den Senken kommen, nicht, dass Knoten A mit Knoten B redet.

4. **Offener Standard, Portierbarkeit, Plattformunabhängigkeit**
   Einfache Anwendungsentwicklung für heterogene Systeme.

5. **Anpassbarkeit an die Veränderlichkeit des Sensornetzes**
   Unterstützung der besonderen Dynamik des Sensornetzes, sowie die Möglichkeit des Auftretens von mobilen, neuen und defekten Knoten.

6. **Quality-of-Service mit Datenzentrierung**
   Anwendungen haben quantifizierbare Anforderungen an die Verfügbarkeit und Qualität der Daten.

7. **Abstrakte Modellierung des Systems und Generierung einer anwendungsspezifischen Middleware-Implementierung**
   Ziel ist die Anpassbarkeit, Portierbarkeit und anwendungsspezifische Optimierung der Middleware, um weniger Ressourcen zu benötigen und den Entwicklers zu unterstützen.

8. **Abstraktion der Sensor/Aktor-Hardware über eine Middleware-Schnittstelle**
   Die Anwendungsentwicklung soll durch eine einheitliche, plattformunabhängige Schnittstelle für die Sensor/Aktor-Hardware vereinfacht werden.

Minimaler Ressourcenverbrauch

Insbesondere drahtlose Sensorknoten verfügen aus Gründen des Energieverbrauchs und auch der Stückkosten nur über sehr wenig Ressourcen wie Speicher, Rechenkapazität und Bandbreite [CES04]. Diese Ressourcen muss sich eine Middleware mit den Anwendungen und ggf. der Systemsoftware teilen. Daher ist die primäre Anforderung, dass eine Middleware auf minimalen Ressourcenverbrauch konzipiert und implementiert ist, zusammen mit den Anwendungen auf

den jeweiligen Sensorknoten ausführbar ist und dennoch die spezifizierte Funktionalität erbringen kann.

Neben den beschränkten Hardware-Ressourcen muss auch der Energieverbrauch minimal sein, wenn nur eine endliche Menge von Energie zur Verfügung steht. Aber auch unabhängig davon ist dies aus Gründen des Umweltschutzes anzustreben. Die Lebenszeit der drahtlosen Sensorknoten und damit des gesamten Sensornetzes hängt direkt von der verfügbaren Energie ab.

Diese Anforderung 1 ist so grundsätzlich, dass sich die weiteren Anforderungen daraus ableiten lassen oder zumindest davon beeinflusst werden.

Ressourcenbewusstsein

Eine Middleware muss sich der verfügbaren Ressourcen, besonders des verbleibenden Energievorrats der autarken Sensorknoten, bewusst sein und die Verbrauchsplanung in die Entscheidungen des operativen Betriebes einbeziehen, um die wenigen Ressourcen effizient ausnutzen zu können, die Energievorräte der autarken Knoten nicht zu vergeuden und somit möglichst lange die geforderte Funktionalität erbringen zu können.

Wie der Anwendungsfall für Sensornetze vorgibt, sind die Sensorknoten zum einen heterogen in ihrer Hardware-Ausstattung, Architektur und Funktionalität, zum anderen können sie sich mit der Zeit unterschiedlich verändern. Komponenten können ausfallen, die Umgebung kann sich ändern und der Energievorrat kann sich auf unterschiedliche Weise erschöpfen. Das bedeutet für eine Middleware, dass sie die Verteilung von Funktionen und Aufgaben, wie zum Beispiel verteilte Speicher-, Rechen- oder Datenweiterleitungsaufgaben, auf die Knoten von diesen Faktoren abhängig gestalten sollte. Dies sollte insbesondere von der verfügbaren Energie auf den jeweiligen Knoten abhängen.

Datenzentrierung

Die Aufgabe eines Sensornetzes ist es, Daten von den Datenquellen zu den Datensenken zu transportieren. Anwendungen, die als Datensenke fungieren, sind an bestimmten Daten interessiert, deren Eigenschaften, Qualität und Verfügbarkeit sie definieren können. Der Anwendungsfall beschreibt das Problem, dass sich das Sensornetz dynamisch ändern kann und daher nicht sichergestellt ist, von welcher spezifischen Datenquelle die Daten zur Datensenke gelangen. Die

Anwendungsschnittstelle der Middleware soll all dies für die Anwendung und den Entwickler abstrahieren und auf das für sie Relevante, die Daten, reduzieren.

Anwendungen in der Rolle einer Datensenke sollen die Beschreibungen der austauschbaren Daten, ggf. angereichert mit gewünschten Eigenschaften wie Anforderungen an die Qualität und/oder Verfügbarkeit, verwenden, um der Middleware mitzuteilen, welche Daten sie benötigen. Entsprechendes gilt auf der Seite der Anwendungen, die als Datenquelle agieren. Diese geben an, welche Daten, ggf. mit welchen Eigenschaften, sie zur Verfügung stellen können. Basierend auf diesen Informationen, die damit als Adressierung dienen, soll es die Aufgabe der Middleware sein, die richtigen Daten von passenden Datenquellen zu den Senken zu bringen. Die Middleware für Sensornetze soll damit einen datenzentrierten Ansatz verfolgen (Anforderung 3).

Offener Standard, Portierbarkeit, Plattformunabhängigkeit

Das beschriebene allgemeine drahtlose Sensornetz ist sehr heterogen; die Knoten verwenden unterschiedliche Komponenten, Architekturen, Software etc.. Die Aufgabe einer Middleware ist es, den Anwendungen eine abstrakte Schnittstelle zu bieten, die die Komplexität und Heterogenität des unterlagerten Systems vor ihnen verbirgt und damit die Anwendungsentwicklung vereinfacht [TS03]. Für Sensornetze kann zusätzlich gefordert werden, dass Anwendungen auch möglichst unabhängig von der verwendeten Hardware der Knoten sein sollen und die Middleware dies unterstützt.

Eine Middleware oder zumindest ihre Anwendungsschnittstelle sollte auf einem offenen, verbreiteten und akzeptierten Standard beruhen, der plattformunabhängig definiert ist und es somit ermöglicht, existierende Anwendungen und Software-Komponenten auf neue Systeme mit überschaubarem Aufwand zu portieren und somit wieder zu verwerten.

Anpassbarkeit an die Veränderlichkeit des Sensornetzes

Es ist eine generelle Anforderung an eine Middleware, dass sie die Komplexität des Kommunikationssystems und damit auch Veränderungen in der Topologie für die Anwendungen abstrahiert [TS03]. Für Sensornetze hat dies besondere Bedeutung und kann Herausforderungen umfassen, die über die Fähigkeiten von traditionellen Systemen hinausgehen.

Kommunikationsverbindungen zwischen Sensorknoten, die nicht drahtgebunden aufgebaut sind, werden durch Umwelteinwirkungen beeinflusst, daher können sie sich dynamisch ändern. Die Middleware oder das von ihr verwendete Kommunikationssystem muss daher in der Lage sein, die Topologie anzupassen und mit transienten Störungen, der Bildung von Partitionen im Netz oder mobilen Knoten umgehen zu können. Statusänderungen und benötigte Daten sind nicht erreichbaren Knoten nach ihrer Rückkehr ins Netz zu übermitteln. Das Sensornetz kann durch neu ins System kommende Knoten wachsen; diese sind einzubinden und die Middleware muss mit steigender oder sinkender Anzahl von Sensorknoten gut skalieren können, sofern dies bei der Entwicklung der Anwendungen vorgesehen ist. Nach Anforderung 2 hat dies unter Berücksichtigung der verfügbaren Ressourcen, insbesondere der Energie, zu geschehen.

Quality-of-Service mit Datenzentrierung

Der Anwendungsfall legt fest, dass Anwendungen als Datensenke die Qualität, Verfügbarkeit oder bestimmte Eigenschaften von Daten beschreiben können, um für die Erfüllung ihrer Funktionalität notwendige Mindestanforderungen festlegen zu können. Auf der anderen Seite können Anwendungen, die als Datenquelle agieren, beschreiben, welche Anforderungen sie erfüllen können. Eine Middleware als Mittler zwischen Datenquellen und Datensenken muss daher Funktionalität zur Beschreibung, Zusicherung, und Bereitstellung von Quality-of-Service bezüglich der auszutauschenden Daten bieten.

Traditionell wird Quality-of-Service als zeitabhängige Anforderung/Zusicherung einer Kommunikationsverbindung zwischen zwei Punkten definiert [TS03], was allerdings der geforderten Datenzentrierung (Anforderung 3) widerspricht, wonach gefordert wird, dass eine Adressierung für die Anwendungen auf den auszutauschenden Daten und nicht auf den spezifischen Datenquellen bzw. Datensenken basiert. Daher soll auch die Verwendung der Quality-of-Service-Funktionalität der Middleware für die Anwendungen datenzentriert sein.

Abstrakte Modellierung des Systems und Generierung einer anwendungsspezifischen Middleware-Implementierung

Eine Middleware ist normalerweise unabhängig von den darauf aufbauenden Anwendungen; sie kann damit Funktionen enthalten, die von einer spezifischen

Anwendung nie genutzt werden. Nicht genutzte Funktionen benötigen dennoch Ressourcen und vergrößern den Footprint der Middleware; dies ist für drahtlose Sensorknoten auf Grund der ersten Anforderung zu vermeiden. Eine Middleware sollte daher auf die im System vorgesehenen Anwendungen optimiert sein.

Nach dem Anwendungsfall sollen Software-Werkzeuge den Anwendungsentwickler auf allen Ebenen der Software-Entwicklung für Sensornetze unterstützen. Für die Middleware kann daher, auch auf Grund der Forderungen nach Plattformunabhängigkeit und einfacher Portierbarkeit der Anwendungen (4), gefordert werden, dass sowohl Anwendungen als auch die Middleware abstrakt modelliert werden sollen und, unter Nutzung weiteren plattformspezifischen Wissens über den Lösungsraum, optimierter Programmcode generiert wird. Für die Middleware soll dieser generierte Programmcode nur noch die Funktionen enthalten, die auch benötigt werden, und auf die verfügbaren Ressourcen zugeschnitten sein.

Abstraktion der Sensor/Aktor-Hardware über Middleware-Schnittstelle

Die Middleware soll die Komplexität und Heterogenität des Sensornetzes und der jeweiligen Sensorknoten-Hardware abstrahieren (Siehe auch Anforderungen 4, 7 und 5). Traditionell bezieht sich dies für Middleware vor allem auf die Kommunikationsebene [TS03], für Sensornetze ist aber auch die Heterogenität der Sensor- und Aktor-Hardware ein Problem für die Entwicklung, Erweiterung und Portierung von Software. Diese können von ihrer Funktion ebenfalls datenzentriert abstrahiert werden; Sensoren sind demnach Datenquellen und Aktoren Datensenken. Die Art und Struktur der Daten lassen sich ebenfalls beschreiben und ihre Eigenschaften bzw. Anforderungen an die Daten quantifizieren. Daher soll nun zusätzlich für die Middleware-Lösung gefordert werden, dass dieselbe abstrakte, datenzentrierte Schnittstelle auch für die lokale Sensor- und Aktor-Hardware verwendet werden soll und Anwendungen darauf aufbauen können. Das Anbinden der jeweiligen Hardware ist Aufgabe der Middleware. Diese Anforderung ist dementsprechend eine Erweiterung der Anforderungen der Datenzentrierung (3), der Plattformunabhängigkeit (4) und der Code-Generierung (7).

3.1.4 Abgleich mit Anforderungen in der Literatur

Nach der Untersuchung der Anforderungen, die sich aus dem allgemeinen Anwendungsfall für Sensornetze ergeben, soll nun analysiert werden, inwieweit die-

se mit Untersuchungen zu diesem Thema in der Literatur übereinstimmen.

[HM06], [MM07] und [RKM02] befassen sich mit der Thematik der Middleware für Sensornetze, untersuchen existierende Middleware-Lösungen und darauf aufbauende Anwendungen und fassen gemeinsame Eigenschaften und allgemeine Anforderungen an Middleware für Sensornetze zusammen. Diese sollen im Folgenden als Basis für den Abgleich mit den aufgestellten Anforderungen dienen und werden im Weiteren nur referenziert, wenn es Unterschiede oder Erweiterungen gibt.

Die Anforderung 1 des minimalen Ressourcenverbrauchs gilt für jegliche Software, die auf drahtlosen Sensorknoten eingesetzt werden soll (Siehe auch [CES04]).

Auch die Anforderung 2 bezüglich des Ressourcenbewusstseins der Middleware findet sich in der Literatur wieder. [MM07] verbindet damit als weitergehenden Schritt die Option, Daten auf den Knoten zu verarbeiten, um weniger Bandbreite zu verbrauchen, da Rechenzeit weniger Energie brauche als Datenübertragung.

Die Schlussfolgerung, dass für Sensornetze eine datenzentrierte Kommunikation besser geeignet ist als eine klassische Punkt-zu-Punkt-Verbindung wird auch von [MM07] und [RKM02] gezogen.

Der Anspruch, dass eine Middleware-Schnittstelle offen sein sollte und auf einem Standard basiert (Anforderung 4) wird konkret nur in [HM06] gefordert, dies kann aber als generelle Anforderung an Middleware gesehen werden [TS03].

Dass sich Sensornetze durch eine besondere Dynamik der Topologie und schwankende Verfügbarkeit der Sensorknoten bzw. der Kommunikation auszeichnen und dies von einer Middleware besonders berücksichtigt werden muss (Anforderung 5), wird auch in der Literatur [MM07] [HM06] beschrieben. Als separate Anforderung wird dort noch die Skalierbarkeit gesehen, welche hier in Anforderung 5 enthalten ist.

Die Anforderung der Unterstützung von Quality-of-Service durch eine Middleware ist ein eigener Forschungsbereich im Sensornetz-Umfeld. Dass dies Teil der Funktionalität einer Middleware sein sollte, deckt sich in der Literatur mit der Anforderung 6. [CV04] untersucht bisherige Bemühungen, Quality-of-Service in Sensornetzen zu ermöglichen. Dass die Festlegung der Qualität auf Basis der Daten und nicht von Punkt-zu-Punkt-Verbindungen erfolgen muss wird dort, wie auch in der bisher angegeben Literatur, ebenfalls gefordert.

Die Anforderung 7, dass eine Middleware-Lösung für Sensornetze einen MDSD-Ansatz verfolgen soll, um die Anwendungsentwicklung zu vereinfachen, wird in der angegeben Übersichtsliteratur nicht gefordert. In [LVCA+07], [ADBS09] und [KMSB08] werden solche Ansätze für die Entwicklung für Sensornetzanwendungen verwendet. [HM06] fordert, dass „anwendungsspezifisches Wissen" von einer Middleware für Sensornetze verwendet werden soll, gibt selbest aber keine Lösung hierfür an.

Die mit Anforderung 8 geforderte Verwendung der Middleware-Schnittstelle zur Abstraktion der Sensor- und Aktor-Hardware wird nicht auf diese Weise in der Literatur beschrieben. In [KPJ06] werden „virtuelle Sensoren" verwendet, um Sensordaten transparent über das Netzwerk zu verteilen. In [Kno09] wird ein „Software-Stecker" für die Entwicklung von Anwendungen für eingebettete Systeme in der Automatisierungstechnik beschrieben, der einen ähnlichen Ansatz darstellt.

3.1.5 Weitere Anforderungen in der Literatur

Über die am Anwendungsfall festgemachten Anforderungen werden in der angeführten Literatur noch die folgenden Punkte als relevant eingeordnet:

- Datenverarbeitung im Netzwerk
- Sicherheit
- Einfache Verwendbarkeit bzw. Selbstkonfiguration

Datenverarbeitung im Netzwerk

Die Datenverarbeitung im Netzwerk ließe sich, wie schon angeführt, unter die Anforderung 2, einordnen. Im Bereich der Sensornetze ist dies ein eigenständiger Forschungsbereich [YMG08] und soll daher hier auch getrennt betrachtet werden. Ziel ist, die Datenübertragung auf Relevantes zu reduzieren, um Energie zu sparen. Eine Möglichkeit hierzu ist die Daten-Aggregation, bei der Daten nach definierten Kriterien auf den Knoten zusammengefasst werden [HSI+01]. Einfache Kriterien sind das Bilden von Minimum, Maximum oder Durchschnittswerten oder die Entfernung von Duplikaten oder gleichartigen Daten. Daneben ist auch die Erkennung von definierten Ereignissen in diesen Bereich einzuordnen, sowohl auf

den Knoten der Datenquelle wie auch auf den Knoten im Netzwerk, über die die Daten geleitet werden. Die Datenverarbeitung im Netzwerk ermöglicht es, neben der Reduzierung der Bandbreite auch Aufgaben an das Sensornetz direkt zu delegieren, und eine Middleware sollte Mechanismen und Schnittstellen vorsehen, dies zu konfigurieren und zu nutzen.

Sicherheit

Durch die zumeist verwendete Datenübertragung über Funk in Sensornetzen können Sensordaten und Informationen abgehört werden, oder Dritte können Daten verfälschen oder auch eigene einspeisen. Es ist klar, dass es Anwendungen gibt, bei denen dies verhindert oder erschwert werden muss, beispielsweise im medizinischen oder militärischen Bereich.

Sicherheit in Sensornetzen wird dadurch erschwert, dass nur wenig Ressourcen bereitstehen, sichere Verfahren zur Verschlüsselung oder Authentifizierung aber aufwendig sind. Hinzu kommt, dass Sensorknoten selten in einer gesicherten Umgebung ausgebracht werden. Das bedeutet, dass Dritte in den physischen Besitz einzelner Sensorknoten kommen können und somit Zugriff auf die Hardware und die darauf gespeicherten Informationen erhalten. Auf Grund der endlichen Energieressourcen kann ein Angriff auch darauf abzielen, diese zu erschöpfen und somit das Sensornetz zu deaktivieren [MM07].

Eine Middleware sollte daher von Anfang an unter dem Gesichtspunkt der Sicherheit entwickelt werden. Nach [HM06] sind die zu betrachtenden Aspekte die Geheimhaltung, Integrität, Aktualität und Verfügbarkeit der Daten.

Selbstkonfiguration und einfache Administration

Sensornetze können aus einer sehr großen Anzahl an Knoten bestehen, welche einmal in einer Gegend ausgebracht werden und dann autonom agieren müssen. Die Ausbringung kann auch zufällig erfolgen, beispielsweise durch den Abwurf aus einem Flugzeug. Es ist also nicht immer mit vertretbarem Aufwand für Benutzer möglich, das Sensornetz zu warten und zu administrieren. Dennoch kann das Netz für eine lange Zeit in Betrieb sein und die Aufgaben können sich währenddessen ändern. Eine Middleware für Sensornetze sollte sich daher nach [MM07] möglichst ohne Eingriff der Benutzer konfigurieren und Möglichkeiten vorsehen, um neue Anwendungen oder Updates einspielen zu können.

3.1.6 Festlegung der endgültigen Anforderungen

Es hat sich gezeigt, dass die Anforderungen, die sich aus dem Anwendungsfall ergeben haben, auch in der Literatur gefordert werden, wobei der Ansatz, Methoden der modellgetriebenen Software-Entwicklung (MDSD) als Basis einer Middleware-Lösung für Sensornetze zu verwenden, noch nicht weit verbreitet ist. Für diese Arbeit soll die folgende Liste an Anforderungen an eine Middleware-Lösung festgelegt werden, die eine Vereinigungsmenge der relevanten Anforderungen aus den Abschnitten 3.1.3, 3.1.4 und 3.1.5 darstellt:

1. Minimaler Ressourcenverbrauch
2. Ressourcenbewusstsein
3. Datenzentrierung
4. Offener Standard, Portierbarkeit, Plattformunabhängigkeit
5. Quality-of-Service mit Datenzentrierung
6. Abstrakte Modellierung des Systems und Generierung einer anwendungsspezifischen Middleware-Implementierung
7. Anpassbarkeit an die Veränderlichkeit des Sensornetzes
8. Abstraktion der Sensor/Aktor-Hardware über die Middleware-Schnittstelle
9. Datenverarbeitung im Netzwerk
10. Sicherheit
11. Einfache Verwendbarkeit bzw. Selbstkonfiguration

Es ist davon auszugehen, dass jede Anwendung, die auf der zu entwickelnden Middleware-Lösung aufbaut, andere Prioritäten bei den Anforderungen setzt, dennoch sind alle Anforderungen relevant und zu berücksichtigen. Daher ist der Aspekt, dass die Middleware an die jeweiligen Anwendungen angepasst wird und nur wirklich benötigte Funktionen enthält, von zentraler Bedeutung. Diese anwendungsspezifische Anpassbarkeit sollte auch auf die Anforderungen, die eine Middleware erfüllen soll, angewendet werden können.

Für den späteren Entwurf und die Realisierung der Middleware im Rahmen dieser Arbeit ist dennoch eine Auswahl und Priorisierung notwendig, da die Erfüllung aller Anforderungen in der gegebenen Zeit nicht möglich ist. Die Anforderungen und Aspekte, die den generellen Aufbau maßgeblich beeinflussen, haben die höchste Priorität und Aspekte, die nachträglich in ein System hinzugefügt werden können die geringste. Als Kern-Anforderungen können 1, 3, 5 und 6 angesehen werden.

3.2 Eignung des Data Distribution Service für Sensornetze

3.2.1 DDS und Sensornetze

Der in den Grundlagen in Abschnitt 2.3 2.3 beschriebene „Data Distribution Service" der OMG ist ein Schnittstellenstandard für eine datenzentrierte Middleware mit umfangreichen QoS-Eigenschaften. Der Standard ist offen, frei verfügbar und plattformunabhängig definiert, mit dem Ziel, darauf aufbauende Anwendungen unabhängig von spezifischen DDS-Implementierungen oder zumindest einfach auf andere Systeme portierbar zu machen. Damit erfüllt DDS direkt die aufgestellten Anforderungen 3, 4, 7 und 5 in Abschnitt 3.1.6 an eine Middleware für Sensornetze.

Es gibt eine Reihe von kommerziellen und auch freien DDS-Implementierungen, welche eine unterschiedliche Teilmenge des Standards implementieren und sich auf verschiedene Anwendungsbereiche und deren unterschiedliche Anforderungen spezialisieren. Die ausgereiftesten Implementierungen, wie OpenSplice oder RTI DDS, zielen auf den Bereich der Unternehmens- oder Militäranwendungen und sind damit möglicherweise seitens ihrer Anforderungen an Speicher, Rechenkapazität und Bandbreite für eingebettete Systeme ungeeignet.

Nach Anforderung 1 in Abschnitt 3.1.6 ist ein minimaler Ressourcenverbrauch für eine Middleware für Sensornetze zwingend notwendig. Der Speicher-Footprint eines DDS, das die gesamte Funktionalität des Standards enthält, wird schwerlich auf einen drahtlosen Sensorknoten passen. Der optionale Teil des DDS-Standards, der das offizielle Kommunikationsprotokoll definiert, baut auf dem IP-Protokoll auf, welches in Sensornetzen kaum verwendet wird. Allerdings ist DDS modular; es ist möglich, nur eine Untermenge der Funktionalität zu implemen-

tieren und dennoch konform zur Schnittstellendefinition zu sein. Wenn DDS für Sensornetze eingesetzt werden soll, ist zu prüfen, ob und wie der Ressourcenverbrauch als wichtigste Anforderung soweit angepasst werden kann, dass eine Verwendung auf drahtlosen Sensorknoten möglich ist.

3.2.2 DDS-Implementierungen und ihre Eignung für Sensornetze

Wenn eine existierende DDS-Implementierung für Sensorknoten verwendet werden soll, ist davon auszugehen, dass Anpassungen notwendig sind, um zum einem die Anforderungen an den Ressourcenverbrauch zu erfüllen und zum anderen das Kommunikationssystem des Sensornetzes zu verwenden. Es ist davon auszugehen, dass diese Anpassungen sehr tiefgehend sein werden, was es zwingend notwendig macht, vollständig auf den Quellcode der Middleware ohne Beschränkungen zugreifen zu können. Dies ist nur bei freier Software (Open-Source) oder in einer Kooperation mit dem Hersteller kommerzieller Software möglich.

3.2.2.1 OpenSplice DDS

OpenSplice DDS von der Firma PrismTech wurde in den Grundlagen in Abschnitt 2.3 kurz beschrieben. Die ursprünglich kommerzielle Software ist in einer Version unter der LGPLv3 als Open-Source freigegeben und wird von PrismTech aktiv weiter entwickelt. Es steht damit vollständig im Quell-Code zur Verfügung, implementiert einen Großteil des DDS-Standards (laut PrismTech vollständig[1]) und wäre damit möglicherweise geeignet, als Basis für Sensornetze zu dienen.

OpenSplice besteht aus mehreren Komponenten, darunter fallen auch die eigentlichen Anwendungen, die als eigenständige Prozesse auf einem Knoten laufen und über ein Shared-Memory-Segment interagieren. Diese Architektur ist grundlegend für OpenSplice und benötigt Betriebssystemfunktionen wie Shared Memory und Semaphore. Allein die Binärdatei des *spliced*-Deamon, der die Koordination der anderen Komponenten übernimmt, ist in der Version 4.3 für Linux 40 KiB groß. Hierzu kommen noch die Bibliotheken, die die Anwendungen einbinden müssen. Es gibt zwei verschiedene Netzwerkkomponenten, die OpenSplice zur

[1] http://www.opensplice.com/

Auswahl anbietet, eine implementiert das DDS-Kommunikationsprotokoll RTPS und die andere ein proprietäres Protokoll. Beide bauen auf der Funktionalität von IP auf und sind nicht auf kleine Netzwerkpaketgrößen optimiert [OMG09].

OpenSplice wurde nicht im Hinblick auf Systeme mit minimalen Ressourcen entwickelt; es benötigt komplexere Betriebssystemfunktionalität und ein IP-basiertes Netzwerk. Eine Anpassung für Sensornetze würde einen großen Eingriff in die elementare Struktur und eine neue Netzwerkanbindung benötigen. Angesichts dessen wäre der Aufwand dafür möglicherweise größer, als ein komplett neues System zu entwickeln. Allerdings kann OpenSplice dafür geeignet sein, als Gateway zwischen DDS-Systemen in traditionellen Netzwerken und denen in Sensornetzen zu dienen. Die Architektur, die auf autonomen Prozessen basiert, könnte eine Erweiterung um eine Netzwerk-Komponente für ein Sensornetz-DDS vereinfachen.

3.2.2.2 TinyDDS

Mit TinyDDS gibt es eine Implementierung von DDS für Sensornetze [BS08] [BS09]. Dieses von der Universität Massachusetts entwickelte System setzt auf TinyOS auf und ist unter einer BSD-Lizenz frei verfügbar. TinyDDS implementiert als Untermenge der DDS-Schnittstelle den einfachen Publish-Subscribe-Datenaustausch zwischen Anwendungen und legt den Fokus auf nicht-funktionale-Eigenschaften. Dazu wurde die DDS-Schnittstelle erweitert, um Datenverarbeitung in der Middleware der Anwendung anbieten zu können, was Daten-Aggregation und die Erkennung von Ereignissen umfasst. Das Routing der Daten und der Aufbau einer logischen Netzwerk-Topologie wird von TinyDDS selbstständig durchgeführt. Der Anwendungsentwickler kann hierzu aus einer Gruppe von Routing-Algorithmen wählen. TinyOS verwendet ein agentenbasiertes Optimierungssystem, um Quality-of-Service zwischen Anwendungen im Sensornetz aushandeln zu lassen. Hierfür gibt es außerhalb des Sensornetzes einen leistungsstarken Koordinator, der die Optimierung mit evolutionären Algorithmen [BS09] durchführt.

TinyDDS ist eng auf TinyOS abgestimmt und in der Programmiersprache nesC implementiert, was auch für alle Anwendungen gilt, die die Middleware verwenden sollen. nesC ist eine Erweiterung von C für Sensornetze und baut auf dem Komponenten- und Event-Modell von TinyOS auf. Die DDS-API wurde deshalb der nesC-Struktur angepasst, was die Portierbarkeit von anderen DDS-Anwendungen stark erschwert. Des Weiteren ist TinyDDS nur in Verbindung mit

TinyOS einsetzbar und somit auf die von TinyOS unterstützten Sensorknoten-Plattformen beschränkt.

Die Implementierung von TinyDDS zeigt, dass DDS grundsätzlich für Sensornetze geeignet ist und die besonderen Anforderungen in diesem Bereich erfüllen kann. Allerdings ist durch die Festlegung auf TinyOS und somit nesC als Programmiersprache für Anwendungen die flexible Nutzung eingeschränkt. Auf TinyDDS aufbauende Anwendungen lassen sich schwer auf andere DDS-Implementierungen portieren. Sensorknoten-Plattformen, die nicht von TinyOS unterstützt werden, sind auch nicht für TinyDDS verwendbar. Neben dieser Plattformabhänigkeit von TinyOS erfüllt TinyDDS auch weitere Anforderungen nicht, die in dieser Arbeit an eine Middleware für Sensornetze gestellt werden. Die Aufteilung in verschiedene Softwareschichten wird auch im Programmcode weitergeführt, generische Funktionalität, die nicht von Anwendungen benötigt wird, bleibt dort erhalten. Es ist nicht vorgesehen, die DDS-Schnittstelle auch zur Abstraktion der lokalen Sensor-Hardware auf den Knoten zu verwenden, und es werden nicht alle für Sensornetze relevanten QoS-Eigenschaften unterstützt.

3.2.3 Weitere Fragestellungen für die Untersuchung von DDS

Die im vorherigen Abschnitt betrachteten DDS-Implementierungen sind nicht geeignet, die Anforderungen an eine Middleware für Sensornetze zu erfüllen. Es ist daher im Folgenden zu untersuchen, ob und ggf. wie eine DDS-Implementierung die Anforderungen in Abschnitt 3.1.6 erfüllen kann. Das Beispiel OpenSplice zeigt, dass eine vollständige Implementierung des Standards mehr Programmspeicher benötigen würde, als ein drahtloser Sensorknoten zur Verfügung hat. Eine Reduktion des Funktionsumfangs und Ausrichtung auf die Sensorknoten-Plattformen ist daher notwendig. Auf der anderen Seite zeigt TinyDDS, dass eine zu starke Anpassung und Optimierung, in diesem Fall auf TinyOS, dazu führen kann, dass die Middleware nicht mehr portabel und universell einsetzbar ist. Es ist daher zu untersuchen, wie ein Mittelweg gefunden werden kann, der in der Lage ist, die aufgestellten Anforderungen zu erfüllen.

Im Weiteren sind dazu die folgenden Fragestellungen zu untersuchen:

1. Wie kann der Speicherbedarf reduziert und begrenzt werden?
2. Welche Funktionalität von DDS wird benötigt?

3. Wie kann DDS in unabhängig kombinierbare und konfigurierbare Module zerlegt werden?

4. Welche Eigenschaften muss ein Kommunikationsprotokoll für DDS haben?

Der nötige Speicherbedarf unterteilt sich in den für Programme und Daten. Die Größe der Binärdatei hängt von den implementierten Funktionen ab und wird damit in der Fragestellung 2 berücksichtigt. Der Speicherbedarf hängt zu einem Teil ebenfalls davon ab, aber zum anderen auch von den Daten, die die Middleware zu verteilen hat. Hierfür ist es notwendig, dass sie Daten intern vorhalten kann, zum Beispiel um sie neuen Knoten im System zur Verfügung stellen zu können oder eine zuverlässige Datenübertragung zu ermöglichen. Das verwendete Datenmodell für die Middleware legt fest, wie groß die Daten werden können und hat damit Einfluss auf den Speicherverbrauch der Middleware. Es ist daher auf die Anforderungen von Anwendungen und Sensorknoten-Hardware zu untersuchen und auf die relevanten Datentypen zu reduzieren, was zusätzlich zu einer Reduktion des Funktionsumfangs führt.

Nach Anforderung 6 soll die Middleware für die jeweiligen Anwendungen optimiert und generiert werden. Dazu ist es notwendig, zu wissen welche Funktionalität von DDS allgemein benötigt wird und wie diese in konfigurierbare Teile zerlegt werden kann.

Das offizielle Kommunikationsprotokoll von DDS ist RTPS, welches auf dem IP-Protokoll aufbaut und damit für Sensornetze nicht geeignet ist. Für diese Arbeit soll ZigBee als Kommunikationslösung für die Middleware verwendet werden. Daher ist zu untersuchen, wie DDS auf diesem Protokoll umgesetzt werden kann.

3.2.4 Anwendungsfall AAL-Haushalt

Computersysteme werden immer leistungsfähiger, kleiner und günstiger. Immer mehr elektrische und elektronische Geräte können daher mit eingebetteten Computersystemen ausgestattet werden, um diese Geräte zu kontrollieren und dem Benutzer eine umfangreichere Schnittstelle zur Bedienung zu bieten. Es ist möglich, kleine, vernetze und eingebettete Systeme aufgrund ihres immer günstigeren Preises in immer mehr Bereichen einzusetzen und Anwendungen zu ermöglichen, die vorher nur mit hohen Kosten oder überhaupt nicht möglich waren. Mit dieser Entwicklung rückt die Vision der „Smart Homes” in greifbare Nähe.

Intelligente Haushaltsgeräte und neue Anwendungen für solche „Smart Homes" werden das private Leben zu Hause immer stärker beeinflussen und verändern.

Durch „intelligente Haushaltsgeräte" und das Potenzial, viele Dinge in einer Wohnung elektronisch steuern zu können, ergeben sich neue Möglichkeiten, Menschen zu unterstützen. Vor allem älteren Menschen kann zu einem selbstbestimmten Leben zu Hause verholfen werden. Dies ist im Hinblick auf die demografische Entwicklung in den Industrieländern ein wichtiges Thema und unter dem Begriff Ambient Assisted Living (*AAL*) ein aktuelles von Bund und EU[1] gefördertes Forschungsfeld. Aktuelle Informationen über die Bewohner und die Ereignisse in einer Wohnung sind die Grundlage dafür, dass ein technisches System unterstützend tätig werden kann. Sensornetze sind dafür konzipiert, Daten der Umgebung zu sammeln, auch sind sie kostengünstig und einfach nachträglich in existierende Umgebungen, wie Wohnungen, zu installieren. Die Anwendungsentwicklung für Sensornetze ist anspruchsvoll, doch eine Middleware nach dem DDS-Standard ist für Sensornetze geeignet und soll die Entwicklung, Erweiterung und Portierung von Anwendungen vereinfachen. Das Potenzial von Sensornetzen und einer DDS-Middleware sollen daher in diesem Anwendungsfall beschrieben werden.

Der im Folgenden beschriebene Anwendungsfall soll einige Teilaspekte aus einem Haushalt im AAL-Kontext herausgreifen und beschreiben. Er stellt eine Untermenge des allgemeinen Anwendungsfalls für Sensornetze dar, der in Abschnitt 3.1.2 beschrieben wurde. Hier sind Sensorknoten in der Wohnung verteilt oder in Haushaltsgeräten eingebaut; es ist dabei zwischen „eingebauten" und autarken Sensorknoten zu unterscheiden. „Eingebaute Sensorknoten" haben eine stetige Energieversorgung, die der autarken Knoten ist endlich. Die Sensorknoten sammeln über ihre Sensoren Daten oder interagieren über Aktoren mit den Geräten, den Bewohnern oder anderen Objekten in der Wohnung. Neben dem Sensornetz gibt es noch andere von den Bewohnern verwendete Computersysteme, wie PCs, Router, Settop-Boxen etc., wobei für diesen Anwendungsfall angenommen wird, dass eines dieser leistungsfähigeren Systeme als ein zentrales Smart Home-Managementsystem (SHMS) dient. Anwendungen für das Smart Home greifen auf das Sensornetz zu, um ihre Funktionalität zu erbringen. Die Anwendungen können entweder direkt auf dem Sensornetz oder, wenn sie mehr Ressourcen benötigen, auf dem SHMS ausgeführt werden.

[1] http://www.aal-europe.de, http://www.aal-deutschland.de

Middleware nach dem DDS-Standard verbindet alle Anwendungen des Smart Homes mit den Daten der Sensoren und den Funktionen der Aktoren. Das DDS für das Sensornetz wird über Gateway-Systeme mit DDS-Implementierungen für traditionelle IP-basierte Systeme verbunden. Alle Anwendungen des Smart Homes setzen damit auf der einheitlichen DDS-Schnittstelle auf und können auf alle für sie relevanten Daten zugreifen. Zur weiteren Vereinfachung der Anwendungsentwicklung für die Sensorknoten abstrahiert die DDS-Schnittstelle nicht nur den Datenaustausch zwischen den Knoten bzw. Anwendungen, sondern auch die Sensor- und Aktor-Hardware lokal auf den Sensorknoten.

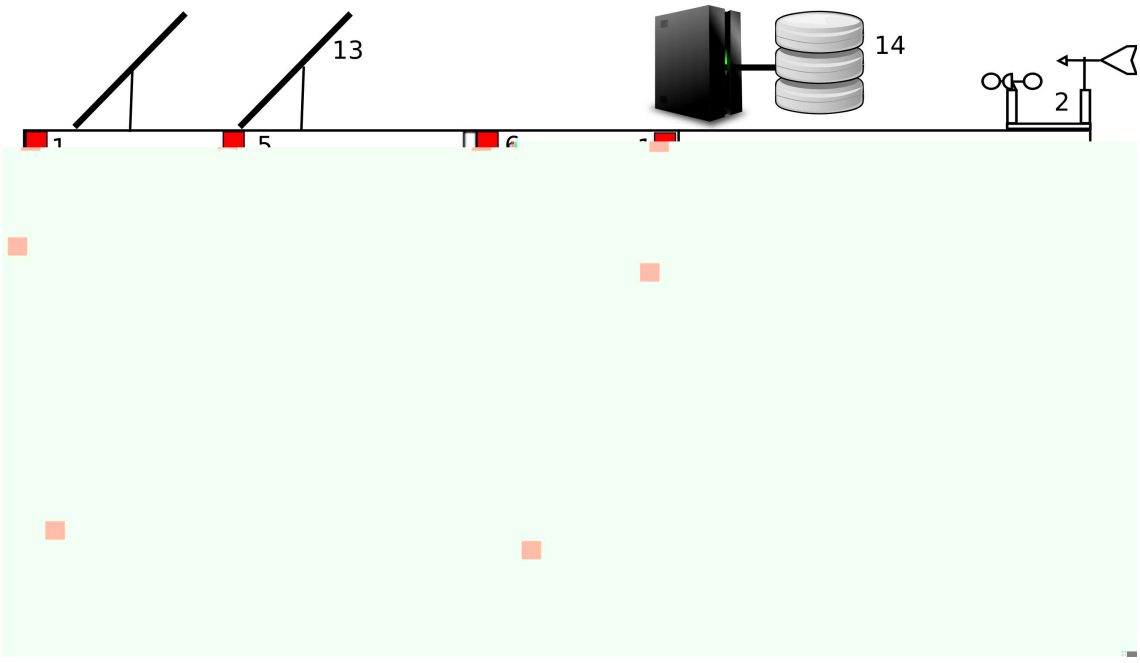

Abbildung 3.2: Anwendungsfall AAL Haushalt

Abbildung 3.2[1] zeigt eine schematische Darstellung des Anwendungsfalls. Die Sensorknoten in dem System werden als kleine rote Quadrate veranschaulicht, die über Funk miteinander verbunden sind. Für den Fall, dass sie Bestandteil eines Gerätes sind, sind sie als außen am Gehäuse hängend dargestellt. In Tabelle 3.1 sind die in der Abbildung dargestellten Sensorknoten aufgelistet, und es ist für jeden Typ von Knoten angegeben, welche Sensoren und Aktoren er enthält. Basierend auf den Sensorknoten sollen nun ausgewählte Anwendungen für den Anwendungsfall des AAL-Haushalts beschrieben werden.

[1] Einige Teile der Grafik stammen von http://www.openclipart.org/

Nummer	Sensor/Aktor
1	Raumsensoren
	Temperatur Bewegungssensor Rauch Luftfeuchtigkeit Licht
2	Wetter/Außensensoren
	Temperatur Licht Wind Regen
3	Fenster/Lüftung
	Rolladen (Aktor) Zustand (offen, geschlossen) Einbruch Lüftung (Aktor)
4	Heizung
	Steuerung (Aktor) Wärmemenge
5	Alarmgeber
	Lichtsignal Akustisches Signal
6	Lampen
7	Personal Device
	Temperatur (Fieber) Hautwiderstand (Stress) Puls Bewegung Notfallknopf Vibrationsalarm (Aktor)

Nummer	Sensor/Aktor
8	PDA
	Steuerung, Managementzugang Infomationsdisplay
9	Haushaltsgeräte
	Funktion/Zustand/Fehler Funktionssteuerung (Aktor) Stromverbrauch
10	Haushaltsroboter
	Saubermachen (Aktor) Zustand Boden Hindernisserkennung Mobil
11	TV
	Informationen, Mitteilungen
12	PDA – Nachbar
	Informationen Ausgewählte Funktionen
13	Solarpanele/Stromerzeugung
	Strommenge
14	Hausmanagement-System
	Überwachung, Beobachtung Entscheidungen, Steuerung Datenspeicherung

Tabelle 3.1: Sensorknoten im AAL-Haushalt und ihre Funktionen (Abbildung 3.2)

Alle Sensorknoten, die Temperaturen messen können, veröffentlichen die Temperaturwerte in dem DDS-Topic `Temperatur`. Dieses enthält den Temperaturwert im Bereich -50 bis +200 °C in $1/10$ °C Auflösung, eine ID des Sensorknotens und die Position, auf der sich der Sensorknoten in oder außerhalb der Wohnung befindet. Diese Werte können jede Sekunde gemessen und veröffentlicht werden. Eine Anwendung auf dem Sensorknoten eines Heizkörpers steuert die Leistung der Heizung mit einem Wert zwischen 0 und 254, den er einem lokalen Topic mit dem Namen `HeizRegler` übergeben kann. Die abgegebene Wärmemenge eines Heizkörpers wird alle 5 Minuten berechnet und im Topic `Wärmemenge` zur Verfügung gestellt, dieses enthält die Wärmemenge, eine Raum-ID und die Seriennummer des Sensors. Die Messwerte von Helligkeitssensoren werden im Topic `Helligkeit` alle 20 Sekunden veröffentlicht, neben dem Helligkeitswert enthält es noch die ID und Position des Sensorknotens. Die Fenster abonnieren zwei Topics zur Steuerung des Rolladens (Topic `Rolladen`) und zum Öffnen bzw. Schließen des Fensters (Topic `FensterAufZu`). In diesen können Datensätze mit dem gewünschten Zustand und eine Fläche, in der der Zustand gelten soll, verbreitet werden. Die Vorgaben der Zustände in einem Gebiet, in dem sich die Fenstersteuerung befindet, werden durch die Steuerungen umgesetzt. Der Zustand eines Fensters selber wird in den Topics `ZustandFenster` und `Glasbruch` veröffentlicht. Das Topic `ZustandFenster` enthält die Raum-ID, die Richtung in der das Fenster liegt, den Zustand des Fensters (Auf oder Zu) und wird bei jedem Zustandswechsel erzeugt. Beim Auftreten eines Glasbruches wird im Topic `Glasbruch` alle 100 ms ein Datensatz veröffentlicht, das die Fenster- und Raum-ID enthält und den Zeitpunkt, zu dem der Glasbruch aufgetreten ist. Bewohner besitzen mit dem „Personal Device" einen Sensorknoten, der Gesundheitsdaten über sie sammelt und mit ihnen interagieren kann. Die Temperatur, der Hautwiderstand und der aktuelle Puls werden in entsprechenden Topics durch eine Anwendung auf dem „Personal Device" alle 3 Minuten zur Verfügung gestellt. Die Datensätze enthalten den jeweiligen Wert und die ID des Benutzers. Die Situation eines ausgesetzten Puls wird nach ihrer Erkennung alle 100 ms im Topic `MediNotfall` mit der ID und Position des Betroffenen solange bekannt gegeben, bis der Puls wieder einsetzt. Die Position und ID des Besitzers des „Personal Device" wird jede Sekunde im Topic `PersonPos` bekannt gegeben. Das „Personal Device" besitzt einen Notfallknopf. Wird dieser gedrückt, wird dieses Ereignis im Topic `Notfall` mit der ID des Besitzers und dem Typ des Notfalls (*Notfallknopf_gedrückt*) bekannt gegeben.

Neben den Positionsdaten der „Personal-Devices” können Personen in einem Raum auch über die Bewegungs- und Infrarot-Sensoren der Sensorknoten erkannt werden. Deren Daten werden kooperativ von den Sensorknoten ausgewertet und unter dem Topic `PersonenImRaum` als die geschätzte Anzahl an Personen im Raum jede Sekunde mitgeteilt. Als „Key” zur Identifikation des Raumzustands enthält das Topic noch die Raum-ID.

Die Anwendungen zur Regelung der Raumtemperatur laufen auf den Sensorknoten der Heizungen. Das SHMS gibt die Soll-Größen für jeden Raum vor und veröffentlicht diese bei jeder Änderung in einem individuellen Topic für jeden Raum (`TempRaumSoll`, als „Key” wird die Raum-ID verwendet). Jede Heizungssteuerung abonniert das Topic und entnimmt nur die Daten mit dem „Key” für den Raum, in dem sie sich befindet, sowie die Topics `Temperatur` und `ZustandFenster`. Bezüglich der beiden letzteren reduziert sie die möglichen Werte bei der Erzeugung des Abonnements durch die Angabe einer Fläche, zum Beispiel der des Raumes, in der sich die Steuerung befindet, bzw. die ihrem Wirkungsumfeld entspricht.

Die Menge der Temperaturwerte wird außerdem durch das Setzen der QoS-Ei-

die Anwendung über die Steuerung der Lampen oder der Stellung der Rollläden in den Räumen. Die Steuerung der Lampen erfolgt über das Topic `Licht`, in dem Steuerungsbefehle veröffentlicht werden, die aus der Fläche, innerhalb der der Befehl gelten soll, dem gewünschten Zustand der Lampe (Ein/Aus) sowie der gewünschten Helligkeit und Farbtemperatur bestehen. Die Sensorknoten der Lampen abonnieren das Topic `Licht` für das Gebiet, in dem sie sich physisch befinden und steuern die angebundenen Lampen den Steuerungsbefehlen entsprechend.

Eine Anwendung zur Notfallerkennung läuft auf dem SHMS. Diese erzeugt für jeden Bewohner ein *MultiTopic* mit den medizinischen Daten des entsprechenden Bewohners. Enthalten sind die Daten über Puls, Temperatur, Hautwiderstand und Position. Ist aus den medizinischen Daten ein Notfall erkennbar oder wird der Notfallknopf gedrückt, wird ein Ereignis im Topic `Notfall` erzeugt. Es enthält die Art des Notfalls, den Namen und die Position des Betroffenen. Dieses Topic kann von Notfalldiensten, Angehörigen oder auch Nachbarn abonniert werden.

3.3 Datenmodell für ein Sensorknoten-DDS

In diesem Abschnitt sollen die Anforderungen an ein Datenmodell für die zu entwickelnde DDS-basierte Middleware untersucht werden, wie es in Abschnitt 3.2.3 als Fragestellung für die Eignung von DDS für drahtlose Sensornetze aufgestellt wurde. Das Datenmodell hat Einfluss auf den Ressourcenverbrauch der Middleware, daher ist das Hauptaugenmerk auf die Größe der möglichen Datentypen eines Datenmodells zu legen. Daneben ist die Relevanz einzelner Datentypen wichtig. Diese wird durch die Untersuchung existierender Sensornetze, ihrer Komponenten und der Anforderungen der Anwendungen ermittelt.

3.3.1 DDS-Datenmodell

Wie in den Grundlagen in Abschnitt 2.3.2.2 beschrieben, gibt der DDS-Standard kein Datenmodell für die Beschreibung der auszutauschenden Daten explizit vor. Allerdings wird in der Regel OMG IDL für die Beschreibung der Datentypen verwendet und auch implizit von DDS vorgesehen. Von OMG IDL werden nur die Teile zur Spezifikation von primitiven und zusammengesetzten Datentypen benö-

tigt. Darüber hinaus unterstützen einige DDS-Implementierungen, wie OpenSplice, auch Namensräume für das Datenmodell [Pri09].

Der IDL-Datentyp, der einem Topic in DDS zugeordnet wird, ist eine Struktur (*struct*), ähnlich dem `struct` in der Programmiersprache C. In dieser Struktur sind weitere Datentypen enthalten, die die Art der auszutauschenden Daten für das Topic beschreiben.Sie setzen sich aus einfachen oder komplexeren zusammengesetzten Datentypen zusammen.

Für ein Datenmodell für ein Sensorknoten-DDS ist die Abschätzung des Speicherverbrauchs der Datentypen notwendig. Die Größe der möglichen Datentypen im Datenmodell ist für den Speicherbedarf der Middleware und die Menge der zu übertragenden Daten zwischen den Knoten relevant.

3.3.1.1 Einfache Datentypen

In Tabelle 3.2 sind die einfachen Datentypen aufgeführt, die OMG IDL vorsieht [OMG08a]. OMG IDL schreibt vor, dass für die Datentypen vom OMG IDL bei der Umsetzung bestimmte Mindestanforderungen einzuhalten sind. Diese sind in der zweiten Spalte der Tabelle als der minimale Wertebereich aufgeführt. Die dritte Spalte enthält die minimale Größe des Datentyps, um diesen Wertebereich darstellen zu können.

Bei den Ganzzahl-Datentypen ist der mögliche Wertebereich direkt abhängig von der Größe des Datentypes. Für ein Datenmodell für ein Sensorknoten-DDS ist zu entscheiden, welche Ganzzahl-Datentypen benötigt werden. Dasselbe gilt für die Gleit- und Festkomma-Datentypen; ihre Größe ist statisch und ihre Relevanz für das Datenmodell muss untersucht werden. Der Datentyp *octet* repräsentiert einen 8 Bit-Wert, der bei einer Datenübertragung nicht verändert werden darf. Dies wird oft benötigt, wenn Register oder Bitmasken zu übertragen sind. Der Datentyp *boolean* benötigt ein Bit, um dargestellt zu werden, wird aber in der Regel durch größere Datentypen repräsentiert, da ein Bit nicht direkt adressierbar ist.

Die Textzeichen *char* und *wchar* sind allein betrachtet ohne große Relevanz, ein *char* ließe sich auch mit einem *octet* abbilden. Die Datentypen *string* und *wstring* sind als eine Folge von *char* bzw. *wchar* definiert. Sie können mit oder ohne Längenangabe verwendet werden. Mit der Angabe der Länge ist vorgegeben, wie viele Zeichen der Datentyp maximal enthalten kann. Ohne eine Angabe kann

OMG IDL Datatype	Wertebereich	Datengröße
Ganzzahl-Typen		
short	$-2^{15} \ldots 2^{15}-1$	16 Bit
long	$-2^{31} \ldots 2^{31}-1$	32 Bit
long long	$-2^{63} \ldots 2^{63}-1$	64 Bit
unsigned short	$0 \ldots 2^{16}-1$	16 Bit
unsigned long	$0 \ldots 2^{32}-1$	32 Bit
unsigned long long	$0 \ldots 2^{64}-1$	64 Bit
Gleitkomma-Typen		
float	$2^{-149} \ldots 2^{128}$	32 Bit
double	$2^{-1074} \ldots 2^{1024}$	64 Bit
long double	$2^{-16445} \ldots 2^{16384}$	79 Bit
Byte		
octet	8 Bit	8 Bit
Textzeichen		
char	ISO 8859-1	8 Bit
wchar	implementierungsabhängig	
Boolischer Typ		
boolean	{TRUE,FALSE}	1 Bit
„Any Type”		
any	Jeder IDL-Datentyp	
Festkommazahl		
fixed	31 Stellen	plattformabhängig
Zeichenketten		
string	Sequenz von char	$n * 8\,\mathrm{Bit}$
wstring	Sequenz von wchar	$n * 16\,\mathrm{Bit}$
Enumeration		
enum	2^{32}	32 Bit

Tabelle 3.2: OMG IDL-Datentypen [OMG08a]

er theoretisch unendlich lang werden. Für Sensornetze ist, sofern überhaupt Zeichenketten benötigt werden, nur die begrenzte Version sinnvoll, da die Verfügbarkeit von Speicher eingeschränkt ist. Die Definition einer oberen Schranke sollte abhängig von der Anwendung und der Gesamtgröße des jeweiligen Datentyps sein.

Der Datentyp *enum* wird verwendet, um eigene Aufzählungstypen zu definieren. Dies erleichtert die Anwendungsentwicklung. Die vorgesehene Größe für diesen Datentyp ist 32 Bit; viele Anwendungen benötigen aber nicht so viele Elemente in einer Aufzählung. In einem Sensornetz wären unbenutzte Elemente Speicherplatzverschwendung. Eine Alternative zu einem *enum* ist es, einen kleinen Ganzzahl-Datentyp zu verwenden und die Elemente mit `const`-deklarierten Variablen festzulegen.

Der *any*-Datentyp kann für jeden beliebigen anderen IDL-Datentyp stehen. Damit lässt sich seine maximal benötigte Größe nicht festlegen. Existierende DDS-Implementierungen wie OpenSplice unterstützen diesen Typ nicht; es gibt nur wenige Anwendungsfälle, in denen er für ein DDS-System benötigt werden könnte.

3.3.1.2 Zusammengesetzte Datentypen

Als zusammengesetzte Datentypen werden nun *struct*, *union*, *array* und *sequence* betrachtet.

Der *struct*-Datentyp ist wichtig für DDS, da er als Container für den Datentyp verwendet wird, der einem Topic zugeordnet ist. Es ist aber auch möglich, Strukturen zu definieren, die nicht als Topic verwendet werden. Existierende DDS-Implementierungen sehen spezielle Kommentare vor, um eine Topic-Struktur zu markieren. Es gibt hierfür keine einheitliche Regelung des DDS-Standards oder der Hersteller. Eine *struct*-Deklaration kann theoretisch unendlich groß werden, für ein Datenmodell eines Sensorknoten-DDS wäre eine Größenbeschränkung ratsam.

union entspricht in OMG IDL einer Kombination von `union` und `switch` in der Programmiersprache C. Die Definition sieht vor, dass dynamisch zur Laufzeit ausgewertet wird, welcher Datentyp verwendet wird.

Mit dem IDL-Datentyp *array* können Arrays fester Größe aus anderen Datentypen deklariert werden. Sofern es für Sensornetzanwendungen relevant ist, sollte ein Datenmodell auch hier maximale Größen festlegen.

Sequenzen sind Folgen von anderen IDL-Datentypen und lassen sich mit *sequence* deklarieren. Die maximale Anzahl an Elementen in der Sequenz lässt sich festlegen, andernfalls ist sie nicht beschränkt. Sequenzen mit statischer Größe lassen sich auch mit dem *array*-Datentyp realisieren.

3.3.2 Von Sensorknoten-Hardware unterstützte Datentypen

3.3.2.1 Mikrocontrollerfür Sensorknoten

Die Eigenschaften und Besonderheiten von Hardware-Plattformen haben Einfluss auf die Effizienz der Verarbeitung und Speicherung von verschiedenen Datentypen. Die Hardware für Sensorknoten unterliegt besonderen Einschränkungen, die sich aus der Notwendigkeit der Energieeffizienz ergeben. Für das Datenmodell ist daher zu untersuchen, welche Datentypen von typischer Sensorknoten-Hardware unterstützt werden bzw. welchen Aufwand ihre Verarbeitung und Speicherung benötigen.

Es existieren verschiedene generische Hardware-Plattformen für Sensorknoten. Diese sind oft das Ergebnis von Forschungsprojekten und für den akademischen Bereich ausgelegt. Für reale Anwendungen werden individuelle Systeme entwickelt, die genau den jeweiligen Anforderungen entsprechen. In [Vdd06] wird beschrieben, wie solche Systeme typischerweise aufgebaut werden, in [PSC09] und [VCSM03] werden existierende Plattformen für Sensornetze untersucht. Sensorknoten bestehen primär aus sogenannten COTS-Komponenten („Commercial-off-the-Shelf"), Bauteile, die üblicherweise in größeren Stückzahlen erhältlich und günstig sind. Der Kern jedes Sensorknotens ist der Mikrocontroller. Wie in Abschnitt 2.1 der Grundlagen beschrieben, sind für autarke, drahtlose Sensorknoten geeignete Mikrocontrollerauf minimalen Energieverbrauch ausgelegt und enthälten den Prozessor, Arbeits- und persistenten Speicher, verschiedene E/A-Schnittstellen und weitere Komponenten.

Aus der angeführten Literatur wurden eine Auswahl an aktuell für Sensorknoten verwendete Mikrocontrollerentnommen und mit ihren Eigenschaften und unterstützter Funktionalität, die die Datentypen des Datenmodells beeinflussen, in Tabelle 3.3 dargestellt.

Die für den Anwendungsfall eines AAL-Haushalts benötigten Mikrocontrollerlassen sich in zwei Gruppen einteilen: Zum einem Mikrocontrollerin Sensorkno-

	ATmega128/L	MSP430	CC2430 Intel 8051	PIC16F87/88	AT91RM9200 (ARM920T)
Architektur	8 Bit	16 Bit	8 Bit	8 Bit	32 Bit
	RISC	RISC	CISC	RISC	RISC
	Harvard	Neumann	Harvard	Harvard	Harvard[6]
Frequenz (MHz)	$\leq$ 16/8	$\leq$ 25/12[1]	$\leq$ 32	$\leq$ 20	$\leq$ 180
Register					
Anzahl	32	16	4 Bänke a 8	RAM	31 (16)[8]
Breite	8 Bit	16 Bit	8 Bit	8 Bit	32 Bit
RAM	4 KiB	$\leq$ 16 KiB	8 KiB[4]	368 Byte	16 KiB[7] + ext.
Flash	128 KiB	$\leq$ 256 KB	32 bis 128 KiB[2]	7 KiB	128 KiB ROM
EEPROM	4 KiB	256 Byte[3]		256 Byte	—
Wortbreite					
RAM	8 Bit	16 Bit	8 Bit	8 Bit	8,16,32 Bit
Flash	16 Bit	1,8,16 Bit	32 Bit	14 Bit	—
Byte Order	Little-Endian	Little-Endian	Big-Endian	—	Beide
FPU	nein	nein	nein	nein	nein (od. ext.)
Mult-HW	ja	optional[2]	ja	nein	ja
Division			ja	nein	nein
Wortbreite	8 Bit	8 oder 16 Bit	8 Bit	—	32 Bit
ADC					
Auflösung	10 Bit	10 oder 12 Bit[2]	12 Bit	10 Bit	—
Kanäle	8	5 bis 16[2]	8 + 2[5]	7	—
Ergebnis	16 Bit	16 Bit	16 Bit	16 Bit	—
AES-HW	nein	optional[2]	ja, 128 Bit	nein	nein
Referenz	[Atm09b]	[Ins09]	[Chi08]	[Mic05]	[Atm09a]

[1] 12 MHz für die Stromsparvariante
[2] Modellabhängig
[3] Emuliert
[4] Davon 4 KiB in tiefen Schlafzuständen persistent
[5] 2 Kanäle werden für die interne Messung von Temperatur und der verfügbaren Energiemenge der Batterie verwendet
[6] Der Prozessorken ARM9TDMI hat eine Harvard-Architektur
[7] Zusätzlich jeweils 16 KiB Daten und Instruktionscache
[8] Von den 31 Registern sind 16 für den Benutzer sichtbar, die anderen werden intern zur Beschleunigung der Verarbeitung verwendet.

Tabelle 3.3: Häufig verwendete Mikrocontrollerfür Sensorknoten

ten, die eine ständige Energieversorgung besitzen, und Mikrocontrollerfür autarke Sensorknoten, die nur über einen endlichen Energievorrat verfügen. Die Mikrocontrollerfür autarke Sensorknoten sind auf minimalen Energieverbrauch ausgelegt und sind daher weniger leistungsfähig, verfügen also über weniger Speicher und Rechenkapazität. Für eine Middleware für Sensornetze stellen diese Systeme die obere Grenze bezüglich der verfügbaren Ressourcen dar. Bis auf den ARM-Mikrocontrollersind die in der Tabelle aufgeführten Mikrocontrollerdieser Gruppe zuzuordnen.

Die in der Tabelle aufgeführten Mikrocontrollerfür drahtlose Sensorknoten basieren überwiegend auf einer 8 Bit-RISC-Prozessorarchitektur, wobei mit der MSP430-Reihe von TI 16 Bit in dieser Klasse eingeführt werden. Als kleinster gemeinsamer Nenner ist also von einer optimalen Größe für Datentypen von 8 Bit für die Datenverarbeitung auszugehen, wohingegen größere Datentypen mehr Maschinenoperationen für einen Verarbeitungsschritt benötigten würden. Die ALUs der untersuchten Mikrocontrollerunterstützen, mit Ausnahme des PICs, die Multiplikation von Ganzzahlen in ihrer Wortbreite, das Ergebnis hingegen ist doppelt so groß. Einzig der 8051 des CC2430 verfügt über Maschinenbefehle zur Division, auf den anderen Systemen muss dies in Software erfolgen. Keines der Systeme bietet eine FPU, damit muss die Verarbeitung von Gleitkommazahlen vollständig in Software erfolgen und ist damit sehr ressourcenintensiv.

In realen Projekten werden die Mikrocontrollernicht mit der maximal möglichen Taktfrequenz betrieben, um weniger Energie zu verbrauchen. Nach den in [PSC09] und [VCSM03] aufgeführten Systemen liegt diese eher zwischen 4 und 8 MHz. Die Menge des eingebauten Arbeitsspeichers bewegt sich im unteren KiB-Bereich, in der Regel sind es zwischen 4 und 16 KiB. Flash-Speicher wird bei allen Systemen als persistenter Speicher verwendet und ist bei den überwiegend auf der Harvard-Architektur aufbauenden Systemen in Daten- und Programmspeicher getrennt. Hierbei stehen, mit Ausnahme des PICs, bei den untersuchten Mikrocontrollermindestens 32 KiB bereit, die in realen Anwendungen auch extern erweitert werden können [ZSLM04]. Die MSP430s besitzen allerdings keine externen Adress- und Datenleitungen, was eine direkte Anbindung erschwert und die Zugriffzeiten erhöht. Die Adressierung und Wortbreite der Daten im Flash-Speicher unterscheidet sich bei den verschiedenen Systemen und oft auch von der Wortbreite der Recheneinheit. Abweichend ist hier der MSP430, welcher es ermöglicht, sowohl 1, 8 als auch 16 Bit große Daten im Flash zu adressieren. Die Byte-Reihenfolge (*Endianness*), in der Daten gespeichert werden oder auf

sie zugegriffen wird, ist bei den Systemen unterschiedlich, wobei die meisten *Little-Endianness* verwenden (Nur der von Intel entworfene 8051 verwendet *Big-Endianness*). Dies ist bei der Datenübertragung zwischen Sensorknoten mit heterogenen Plattformen zu beachten.

Die eingebauten ADCs haben entweder eine Auflösung von 10 Bit oder von 12 Bit und mehrere Kanäle. Die Daten werden vom ADC auf allen Systemen in zwei 8 Bit-Registern geschrieben, direkt angeschlossene Sensoren benötigen damit 16 Bit-Ganzzahldatentypen zur Darstellung der Rohdaten.

Der CC2430 und einige MSP430 besitzen einen AES-Verschlüsselungscoprozessor für die Kommunikation, welcher auch von Anwendungen und damit auch von einer Middleware verwendet werden kann. Der CC2430 ist als System-on-Chip (*SoC*) als Basis für Sensorknoten entworfen worden, es kann davon ausgegangen werden, dass zukünftige Systeme diese Hardwareunterstützung ebenfalls bieten.

Zusammenfassung

Aus der Untersuchung der Auswahl an Mikrocontrollern für Sensorknoten ergibt sich für das Datenmodell eine Präferenz für kleine Ganzzahldatentypen, da diese am besten von den Systemen verarbeitet und gespeichert werden können. Diese sollten möglichst nahe an der Wortbreite der Architektur der Systeme liegen. Die Verarbeitung von größeren Ganzzahlen ist aufwendiger, aber immer noch zu vertreten. Keines der Systeme unterstützt hingegen Gleitkommazahlen, was die Verarbeitung dieser Datentypen sehr aufwändig macht. Das Datenmodell sollte daher diese Typen vermeiden oder nur Festkommazahlen unterstützen.

Die persistente Speicherung von Daten im Flash-Speicher bedarf besonderer Sorgfalt, da sich die adressierbare Wortbreite von der sonstigen unterscheidet. Bei der Speicherung von Datentypen des Datenmodells im Flash ist darauf zu achten, dass kein Speicherplatz durch ungünstig gewählte Datengrößen ungenutzt bleibt.

Da keine einheitliche Byte-Reihenfolge verwendet wird, ist inbesondere bei der Kommunikation in heterogenen Sensornetzen auf die richtige Ordnung der Bytes zu achten.

Der Arbeitsspeicher ist in allen Systemen relativ klein und muss von Anwendungen, Middleware und ggf. Betriebssystem geteilt werden. Für die Middleware be-

deutet dies, dass es eine obere Grenze für die Menge an Daten gibt, die sie vorhalten kann. Die Menge dieser Daten hängt von der Anzahl und der Größe der Datentypen ab. Es ist also notwendig, dass das Datenmodell eine obere Grenze für die Größe eines Datentyps festlegt. Da der verfügbare Speicher und die Menge der vorzuhaltenden Daten aber von der jeweiligen Anwendung abhängig ist, sollte dies nicht statisch sein, sondern von der Anwendung abhängig gemacht werden.

Diese Beschränkungen gelten für Systeme mit geringen Stromverbrauch, diesbezüglich unkritische Systeme können über mehr Ressourcen verfügen. In Zukunft werden Senorknoten-SoC, die Mikrocontroller, Funk und weitere Hardware auf einem Chip vereinen, wie der CC2430, eine größere Rolle in Sensornetzen spielen, da sie die Entwicklung vereinfachen und die Stückpreise reduzieren. Allerdings werden die hier aufgezeigten Einschränkungen für das Datenmodell auch für zukünftige Systeme gelten; die nächste Generation von SoC für Sensorknoten von TI , die CC430-Serie beispielsweise, vereint den hier untersuchten MSP430 mit einem Funk-Transceiver [TI09].

3.3.2.2 Sensoren

Die Sensoren ermöglichen es den Sensorknoten, Daten aus ihrer Umgebung zu erfassen; sie sind die Grundlage für die Sensornetzanwendungen. Sensoren wandeln physikalische Größen mittels physikalischer Effekte in elektrische Signale um. Diese Signale können elektrischer Widerstand, Spannung oder Strom sein, die sich weiterverarbeiten lassen [Her05]. Der physikalische Effekt, der verwendet wird, eine physikalische Größe zu messen, ist in der Regel noch von anderen physikalischen Größen, also Umgebungseinflüssen wie Temperatur oder Spannungsschwankungen abhängig. Dies ist in der Auswertung des elektrischen Signals entsprechend zu berücksichtigen. Oft wird elektrische Spannung als Signal verwendet, da sich diese gut messen und auswerten lässt.

Für die Verarbeitung in einem Sensorknoten muss das analoge elektrische Signal mittels eines ADC in einen digitalen Wert umgewandelt werden. Neben der Geschwindigkeit, mit der ein ADC ein elektrisches Signal erfassen und umwandeln kann, ist u.a. die Auflösung für die Qualität der Sensordaten entscheidend. Es gibt „intelligente” Sensoren, die einen ADC enthalten und vom Mikrocontrollerausgelesen werden können.

	Temperatur SHT21	Luftfeuchtigkeit SHT21	Licht TSL2250	Luftdruck MS5534A	Beschleunigung LIS3LV02DQ
ADC	ja		ja	ja	ja
Auflösung	14 Bit	12 Bit	12 Bit	15 Bit	12 Bit
Datentyp	16 Bit o.V.		16 Bit pro Kanal[1]	16 Bit	3x 16 Bit[2] Vorz.
Rohwert	ja		ja	ja	nein
Auslesesystem	I2C		SMBus	Seriell	I2C/SPI
Auslesezeit					
ungenau	8 Bit: 3 ms		80 ms[1]	—	—
genau	14 Bit: 85 ms	12 Bit: 22 ms	400 ms[1]	35 ms	1,5 ms
Referenz	[sen09]		[TAO03]	[Int05]	[STM05]

[a]Hat zwei Photodioden, eine für sichbares und eine für infrarot Licht, welche beide getrennt ausgelesen werden müssen.

[b]Werte für drei Achsen

Tabelle 3.4: Sensoren für physikalische Größen des Anwendungsfalls

Auf Grund der Vielfalt der möglichen Sensoren soll im Folgenden eine Auswahl, die sich aus dem Anwendungsfall des AAL-Haushaltes ergibt, betrachtet werden. Als zu messende Größe werden Temperatur, Luftdruck, Licht, Luftfeuchtigkeit und Beschleunigung betrachtet und in Tabelle 3.4 dazu jeweils ein Sensortyp mit seinen Eigenschaften aufgelistet.

Alle hier aufgeführten Sensoren können an einen Mikrocontrollerüber eine serielle Schnittstelle angebunden werden. Sensoren, die direkt an den ADC des Mikrocontrollerangeschlossen werden sind ebenfalls einsetzbar, auch wenn sie hier nicht betrachtet werden.

Bei den in der Tabelle aufgelisteten Sensoren zeigt sich, dass sich die Rohdaten der verschiedenen Sensoren gleichen, 16 Bit sind ausreichend, um sie darzustellen. Abhängig von dem Funktionsumfang der Sensoren sind weitergehende Verarbeitungsschritte auf den Mikrocontrollernotwendig, um aus den Rohdaten der Sensoren verwertbare physikalische Werte zu berechnen. Hierfür werden oft weitere physikalische Größen der Umwelt wie die Temperatur benötigt, die aber von den Sensoren selbst ermittelt werden können. Es gibt aber auch Sensoren, wie den LIS3LV02DQ, der diese Schritte intern erledigt und direkt verwertbare Ergebnisse ausliefert.

Für das Datenmodell bedeutet dies, dass Ganzzahlen, sowohl mit als auch ohne

Vorzeichen, notwendig sind, um die Rohwerte der Sensoren verteilen zu können. Gleitkomma-Datentypen werden hingegen für die Rohwerte nicht benötigt.

Es ist aber zwischen den Rohwerten von den Sensoren und den endgültigen Werten, die vom Mikrocontrollerberechnet werden, zu unterscheiden. Die physikalischen Werte müssen nicht linear über einen Wertebereich verteilt sein; logarithmische Verteilungen sind häufig anzutreffen. Gleitkommazahlen sind dazu geeignet, diese Werte darzustellen. Allerdings kann der Informationsgehalt der realen Werte nicht größer sein als der der Rohdaten. Bei entsprechend gewählter Repräsentierung lassen sich Gleitkomma-Datentypen oft vermeiden oder auch kleinere Datentypen für die Daten wählen.

3.3.2.3 Kommunikationssystem

Das Datenmodell kann von den verwendeten Kommunikationssystemen beeinflusst werden. Diese legen fest, wie viele Daten mit welcher Geschwindigkeit übertragen werden können. Für Sensornetze werden meistens funkbasierte Systeme verwendet, da diese mit wenig Aufwand ausgebracht werden können. Sie sind in der Regel den LR-WPAN (Low Rate Wireless Personal Area Network) zuzuordnen und zeichnen sich durch einen geringen Stromverbrauch aus, was auf Kosten der möglichen Bandbreite und Reichweite geht. Fest installierte Sensorknoten können aber auch über eine Verkabelung verbunden sein. Für Sensornetze gibt es verschiedene Protokolle und Standards, die sich in der Datenübertragungsgeschwindigkeit, dem Energieverbrauch und der Anwendungsschnittstelle unterscheiden. Es ist nicht möglich, die Middleware auf ein System festzulegen, wenn diese flexibel und hardware-unabhängig sein soll. Auch ein Mischbetrieb mit verschiedenen Systemen, z.B. festverdrahtete und funkbasierte, ist für Sensornetze möglich. Die Aufgabe der Middleware ist es dann, als Vermittler zu agieren. Wenn Kommunikationssysteme das Datenmodell beeinflussen, ist sicherzustellen, dass das Modell der Middleware für alle Systeme gültig ist, oder deren Modelle zumindest aufeinander abgebildet werden können.

Die maximale Größe der Nutzdaten eines Pakets (Frame) einer Protokollschicht ist eine feste Größe. Müssen größere Daten übertragen werden, ist es notwendig, dass diese aufgeteilt und in mehreren Paketen übertragen werden. Die Implementierung einer solchen Fragmentierung in einem Kommunikationsprotokoll ist aufwändig und erhöht den Speicherbedarf auf einem Sensorknoten. Für Sensornetze, bei denen erhöhter Ressourcenverbrauch zu vermeiden ist, ist daher

System	Datenrate	Laststrom	Nutzdaten	Bemerkungen	Ref
CC1000	max 76,8 Kbps	TX 10,4 mA RX 7,4 mA	119 Byte	Reine Paketgröße Payload abhängig vom Protokoll	[PSC09]
CC2420	250 Kbps bei 2,4 GHz	TX 17,4 mA RX 19,7 mA			[Chi08]
IEEE 802.15.4			102 Byte		[Ada06]
ZigBee			87 Byte		[All08]
KNX			8/255 Byte	Standard/ Extended-Frame	[Ass09]
TP1	9600 bit/s			Twisted-Pair	
PL110	1200 bit/s			Power-Line	
RF (868,3 MHz)	32768 cps				
CAN	125 Kbps – 1 Mbps	—	8 Byte		[Ets94]
Ethernet	100 Mbps – 1 Gbps	—	1500 Byte		

Tabelle 3.5: Vergleich von Kommunikationssystemen

zu überlegen, ob es nicht möglich ist, darauf zu verzichten. Im Falle, dass keine Fragmentierung unterstützt oder festgelegt wird, oder dass die Verwendung nur in Ausnahmefällen erlaubt ist, ist die maximale Größe der Nutzdaten eines Paketes die maximale Größe, die eine Nachricht der Middleware haben kann. Das Datenmodell einer Middleware muss in diesem Fall die maximale Größe, die ein Datum haben kann, festlegen oder beschränken, so dass es zuzüglich des Protokoll-Overheads der Middleware in ein Paket passt.

Als eine Auswahl an möglichen Kommunikationssystemen und Protokollen für Sensornetze sollen die in Tabelle 3.5 aufgeführten betrachtet werden. Sie umfassen neben zwei nach [PSC09] gebräuchlichen Funk-Receivern und Protokollen auch einige drahtgebundene Systeme, die für Sensornetze verwendet werden können. Das IP-Protokoll stellt das System dar, das in der Regel außerhalb des Sensornetzes eingesetzt wird, und an das sicherlich eine Anbindung erfolgen muss.

Die beiden Funk-Transceiver von TI stehen stellvertretend für zwei Kategorien. Der CC1000 als reiner Transceiver implementiert nur den Physical-Layer [TI07], während der CC2420 den IEEE 802.15.4-Standard [TI04] unterstützt und mit dem ZigBee-Protokoll verwendet werden kann. IEEE 802.15.4 und ZigBee sind für

Sensorknoten-Plattformen beliebt [PSC09] [BPC+07], da die Hardware günstig ist und wenig Energie benötigt. KNX ist ein Feldbus-Standard für die Gebäudeautomation und der Nachfolger des EIB (Europäischer Installationsbus). Er sieht als Kommunikationsmedien Twisted-Pair-Verkabelung, Powerline und Funk vor und kann für fest installierte Sensorknoten in einem Gebäude von Interesse sein. CAN als weiterer Feldbus kommt aus dem Automotive-Umfeld und ist inzwischen auch in anderen Bereichen weit verbreitet [Ets94]. Des Weiteren verwendet CAN eine datenzentrierte Adressierung, was es für DDS interessant machen könnte.

Die benötigte Leistung zum Senden und Empfangen von Daten ist in der Tabelle nur für die Funk-Transceiver angegeben, da diese von drahtlosen Sensorknoten eingesetzt werden und die Leistung zusammen mit der Datenübertragungsrate und Betriebsspannung den Energieverbrauch für das Senden oder Empfangen von Daten bestimmt. Verkabelte Sensorknoten und somit fest installierte haben hingegen meist eine stetige Energieversorgung, so dass der Energieverbrauch eine weniger wichtige Rolle einnimmt.

Für das Datenmodell zeigt sich, dass die Beschränkung der möglichen Größe eines Datums die notwendige Energie zum Übertragen bzw. Empfangen dieses Datum begrenzend beeinflusst. Die Beeinflussung ist schwer zu quantifizieren, da andere Faktoren, wie das Einschalten und Initialisieren des Receivers, ebenfalls großen Einfluss auf den Energieverbrauch haben bzw. generell gewisse Grundkosten entstehen.

Die mögliche Paketgröße der Kommunikationssysteme und Protokolle unterscheidet sich stark. Das Minimum stellen CAN und der Standard-Frame von KNX mit 8 Byte dar, wobei der Extended-Frame von KNX auch 255 Byte groß sein kann. Bei gewünschter Kompatibilität verschiedener Systeme durch die Middleware und Vermeidung von Fragmentierung gibt die kleinste mögliche nutzbare Datengröße eines Systems die maximale Datengröße für alle verwendeten Systeme vor. CAN unterscheidet sich auf Grund seiner datenzentrierten Adressierung stark von den anderen aufgeführten Systemen, die Knoten adressieren. Es ist daher anzunehmen, dass eine Anbindung von CAN an die Middleware sich stark von der Anbindung der anderen Systeme unterscheidet und es kann daher festgelegt werden, dass das Datenmodell für CAN-Geräte sich von dem übrigen unterscheiden kann und eine Abbildung nur von CAN auf andere Systeme garantiert werden kann.

Die Protokolle KNX und ZigBee bieten die Möglichkeit, bei der Entwicklung von

Anwendungen auf in den Standards enthaltene Profile aufzubauen, die mögliche Endgeräte und ihre Funktionen beschreiben [Ass09] [Far08]. ZigBee wurde in den Grundlagen in Abschnitt 2.2 vorgestellt. In den Profilen von ZigBee und KNX ist ein Gerät oder eine Anwendung mit einer eindeutigen ID und den auslesbaren Daten sowie den möglichen Operationen, die sie anbieten, definiert. Diese Profile stellen eine Art Datenmodell für spezifische Anwendungen oder Geräte dar und sind daher einschränkend. Das Datenmodell für DDS hingegen hat den Anspruch, den Rahmen vorzugeben und ansonsten flexibel zu sein, um möglichst beliebigen Anwendungsfällen gerecht zu werden. Beide Protokolle bieten aber auch die Option, unspezifizierte Datenfelder zu übertragen, so dass eine Middleware mit einem eigenen Datenmodell dennoch auf diese Protokolle aufbauen kann.

3.3.3 Anwendungsfall AAL-Haushalt

Der in Abschnitt 3.2.4 vorgestellte Anwendungsfall beschreibt die für die Anwendungen notwendigen DDS-Topics und die Art und Struktur der Daten, die über über die Topics ausgetauscht werden sollen. Daraus ergeben sich die Anforderungen, die von dieser Seite an das Datenmodell gestellt werden.

Die Datentypen der Topics sind aus mehreren anderen Datentypen zusammengesetzt. Neben den Datentypen zur Darstellung von Sensordaten, wie sie schon im Abschnitt der Sensoren 3.3.2.2 untersucht wurden, sind weitere Datentypen zur Repräsentation weiterer Informationen notwendig. Für die Sensordaten kann nach der vorherigen Untersuchung davon ausgegangen werden, dass Ganzzahlen mit maximal 32 Bit ausreichend sind.

Es wird gefordert, dass alle Sensorknoten und Bewohner das Haushaltes eine eindeutige ID besitzen. Diese muss eindeutig innerhalb des Namensraum des Systems sein. Aus Gründen des Datenschutzes sollte die Gültigkeit sich auf den Bereich beschränken, den die Benutzer kontrollieren können. Abhängig von der möglichen Anzahl der Benutzer wird ein vorzeichenloser Datentyp mit einer Größe von 8 Bit bis 32 Bit benötigt (128 Bit, wenn z.B. IPv6-Adressen als ID verwendet werden sollten). Da die ID in vielen Datensätzen enthalten ist, die ggf. in größerer Menge vorzuhalten sind, sollte der Datentyp möglichst klein sein.

Zur Definition von Zuständen lassen sich Enumerationen verwenden; für die meisten Anwendungen im Anwendungsfall sind 8 Bit ausreichend. Für binäre Zu-

stände wie z.B. „Ein oder Aus”, ist ein boolescher Datentyp ausreichend.

Bei vielen Daten ist der Standort der abonnierenden oder veröffentlichenden Sensorknoten von Bedeutung. Hier ist zwischen einer Position und einem abgegrenzten Gebiet zu unterscheiden. Eine Position ist ein Punkt in einem Koordinatensystem, ein Gebiet hingegen definiert sich als Fläche, die z.B. durch mehrere Punkte definiert wird. Für das benötigte Koordinatensystem gibt es verschiedene Optionen, die sich in der Größe der Datentypen zur Darstellung eines Punktes unterscheiden. Für ein globales System sind jeweils 32 Bit für die X, Y und Z-Achse ausreichend (Am Äquator gäbe es damit eine Auflösung von 9 cm). Ein lokales Koordinatensystem, das nur in dem Gebiet des Systems gültig ist, kommt mit weniger großen Datentypen aus. Zur Unterstützung von Positionsangaben benötigt das Datenmodell daher die Möglichkeit, Tupel von zwei oder drei Ganzzahlen für einen Punkt und Sequenzen von Tupeln für Flächen definieren zu können. Alternativ können unveränderliche Flächen auch mit IDs referenziert werden.

Eine Repräsentierung von Zeit ist notwendig, um Sensordaten unterschiedlicher Sensorknoten miteinander in Relation bringen oder kausal ordnen zu können. Zeitmessung erfolgt über das Inkrementieren von Ganzzahlvariablen, entsprechend können Zeitpunkte mit Ganzzahlen repräsentiert werden. Die notwendige Größe der Datentypen hängt von der benötigten zeitlichen Auflösung durch die Anwendungen ab. Unabhängig davon definiert der DDS-Standard einen eigenen Datentyp für einen Zeitpunkt (Timestamp) und sieht implizit vor, dass dieser jedem übertragenden Datum angehängt wird. Darüber hinaus sollte das Datenmodell die Definition von eigenen Zeit-Datentypen erlauben, die auf den spezifischen Anforderungen der Anwendung basieren.

Mitteilungen an Benutzer oder die Selbstbeschreibung von Geräten benötigen Zeichenketten, welche aber durch das Datenmodell in der Größe beschränkt werden können, da sie viel Speicher und Bandbreite bei der Übertragung benötigen.

3.3.4 Anforderungen an das Datenmodell

Als Ausgangsbasis für die Definition eines Datenmodells für die zu entwickelnde DDS-Middleware für Sensornetze dienen die in Abschnitt 3.3.1 vorgestellten OMG IDL-Datentypen. Das Datenmodell beeinflusst über die Größe der unterstützten Datentypen die Menge an Speicher, die die Middleware zum Betrieb benötigt. Nach der primären Anforderung 1 an eine Middleware muss der Res-

sourcenverbrauch und damit der benötigte Speicher klein sein, damit die Middleware auf einem drahtlosen Sensorknoten einsetzbar ist. Für das Datenmodell wurde in den bisherigen Abschnitten untersucht, welche grundsätzlichen Anforderungen Sensorknoten-Hardware und der Anwendungsfall des AAL-Haushaltes an das Datenmodell haben, sowie, wie aufwendig die Verarbeitung und Speicherung von Datentypen auf den untersuchten Systemen ist. Datentypen oder Eigenschaften des ursprünglichen DDS-Datenmodells, die entweder nicht benötigt oder mit unverhältnismäßig hohem Aufwand verbunden wären, sollen daher in dem Datenmodell für die Sensornetz-Middleware nicht enthalten sein.

Ganzzahltypen mit oder ohne Vorzeichen werden in allen untersuchten Bereichen benötigt. Die verwendete Größe sollte sich an der Architektur der verwendeten Mikrocontrollerorientieren; bei den untersuchten aktuellen Systemen wäre 32 Bit eine „weiche" obere Schranke. Dies ist in der Regel auch ausreichend, um Sensordaten in der von den Sensoren erreichbaren Auflösung darzustellen. Der OMG IDL-Datentyp *octet* lässt sich immer dann einsetzen, wenn ein Byte ohne Konvertierung oder Interpretationen seitens der Middleware oder des Kommunikationssystems benötigt wird. Es ist damit möglich, dass Anwendungen Datentypen verwenden, die von der Middleware nicht unterstützt werden.

Die Verarbeitung von Gleitkommazahlen auf den Mikrocontrollern für drahtlose Sensorknoten ist sehr aufwendig, da diese über keine eigene FPU verfügen und ihre Speicherung vergleichsweise viel Speicher benötigt. Daher ist es grundsätzlich angebracht, ihre Verwendung zu vermeiden. Aufbereitete Sensordaten, deren physikalische Einheiten Gleitkommazahlen benötigen, können bei der Wahl einer entsprechenden Repräsentierung ohne Informationsverlust auch mit Ganzzahlen dargestellt werden, da die Rohdaten ebenfalls diesen Datentypen entsprechen. Da der Anwendungsfall des AAL-Haushaltes ebenfalls keine Gleitkommazahlen benötigt, sollte ihre Unterstützung im Datenmodell optional sein.

Enumerationen werden benötigt, um Zustände oder Aufzählungen definieren zu können. Mit minimal 32 Bit sind bei OMG IDL aber mehr Elemente möglich, als die meisten Anwendungen für Sensornetze benötigen. Damit ergäbe sich vergeudeter Speicher, Bandbreite und damit Energie, vor allem, wenn solche Zustände häufig übertragen werden. Ein ressourcensparendes Datenmodell sollte nur so viele Bits für einen solchen Datentyp verwenden, wie für die Anzahl der deklarierten Elemente notwendig. Entsprechendes gilt für den booleschen Datentyp, der aber in der Regel auf die minimal adressierbare Größe abgebildet werden muss.

Der *any* Datentyp kann eine unterschiedliche Datengröße haben; für den Betrieb oder die Planung von Ressourcen wird mit der maximal möglichen Größe gearbeitet werden müssen. Es ist offensichtlich, dass damit im realen Betrieb Ressourcen ungenutzt bleiben. Da er seitens des Anwendungsfalls oder der untersuchten Sensordaten auch nicht benötigt wird, ist im Datenmodell auf ihn zu verzichten. Ähnliches gilt auch für den *union*-Datentyp von OMG IDL; hier kann zwar die Anzahl der möglichen enthaltenen Datentypen auf relevante beschränkt werden, aber weder der Anwendungsfall noch die untersuchten Sensoren benötigen ihn.

Bei den komplexen Datentypen, wie sie von OMG IDL vorgesehen sind, werden die Strukturen benötigt, da diese als Datentyp-Container für die DDS-Topics dienen. Des Weiteren können sie dafür genutzt werden, die Tupel für den Punkt in einem Koordinatensystem zu definieren. Entsprechend werden auch die Sequenzen von OMG IDL für die Definition von Flächen benötigt. Für die Übertragung von Informationen, die direkt von Menschen lesbar sind, werden Strings benötigt. Dies kann auch für die Anbindung von anderen Systemen gelten, mit denen nur über Strings kommuniziert werden kann. Strings können in OMG IDL auch aus Unicode-Zeichen bestehen, die mehr Speicherplatz als 8 Bit-Zeichen benötigen und deren Verarbeitung auch aufwendiger ist. Die Unterstützung von Unicode ist relevant für Anwendungsprogramme, aber der Aufwand für Sensornetze zu hoch, als dass es Teil des Datenmodells sein sollte.

Strukturen, Sequenzen, Arrays und Strings sind Datentypen mit optional variabler Datengröße. Diese Variabilität ist, wie auch schon bei dem *any*-Datentyp, für die Planung und den Betrieb von Systemen mit minimalen Ressourcen ein Problem und sollte daher von Datenmodell ausgeschlossen werden.

Die maximale Größe, die ein (zusammengesetzter) Datentyp haben darf, ist nicht eindeutig festzulegen, sondern hängt von verschiedenen Faktoren ab. Unter anderem vom verfügbaren Speicher auf dem Sensorknoten für die Middleware, der Anzahl der Datensätze, die von ihr vorgehalten werden müssen, sowie der Menge an möglichen Nutzdaten der verwendeten Kommunikationssysteme, wenn keine Fragmentierung unterstützt werden soll. Explizit eine allgemein gültige Maximalgröße für das gesamte Datenmodell festzulegen schränkt die Flexibilität der Middleware ein und reduziert zusätzlich die Effizienz der Datenübertragung, wenn Kommunikationssysteme eingesetzt werden, die größere Nutzdaten übertragen können oder unterschiedliche Systeme nie miteinander kommunizieren. Anforderung 6 legt fest, dass Werkzeuge die Entwicklung der Anwendungen un-

terstützen und die Middleware an die jeweiligen Begebenheiten anpassen. Für maximale Flexibilität und Effizienz sollten in diesem Prozess alle Anforderungen, konfigurierten Parameter und die deklarierten Datentypen auf Konsistenz bezüglich des verfügbaren Speichers überprüft werden und im Falle von Problemen der Anwendungsentwickler bei der Lösung durch die Werkzeuge unterstützt werden.

3.4 Ein modulares und leichtgewichtiges DDS für Sensornetze

In Abschnitt 3.2.1 wurde festgestellt, dass der DDS-Standard geeignet ist, die Anforderungen aus Abschnitt 3.1.6 an eine Middleware für Sensornetze zu erfüllen. Der DDS-Standard wurde nicht speziell für Sensornetze entworfen, daher würden für die Implementierung der gesamten DDS-Funktionalität mehr Ressourcen benötigt werden, als drahtlose Sensorknoten besitzen. Aus diesem Grund wurden in Abschnitt 3.2.3 Aspekte aufgeführt, unter deren Berücksichtigung die weitere Untersuchung stattfinden soll. Dementsprechend wird im Folgenden eine Auswahl an für Sensornetze relevanten DDS-Funktionalitäten getroffen und diese im Hinblick auf die Reduzierung des Ressourcenverbrauchs untersucht. Parallel dazu wird betrachtet, wie sich diese Funktionalität auf Modell- bzw. Konfigurationsebene modularisieren und damit für die geplante anwendungsspezifische Generierung der Middleware getrennt einbinden lässt.

3.4.1 Relevante DDS-Funktionalität

Auch wenn, wie gezeigt, der DDS-Standard grundsätzlich für Sensornetze geeignet ist, wurde er für Systeme mit mehr verfügbaren Ressourcen entworfen. Daher spezifiziert er Funktionalität, welche entweder für Sensornetzanwendungen nicht relevant oder nur mit unverhältnismäßig hohem Ressourcenaufwand zu implementieren ist. Es ist zu untersuchen, welche Funktionalität von DDS benötigt wird und unter den Rahmenbedingungen eines drahtlosen Sensorknotens implementierbar ist.

Die Modularisier- und Konfigurierbarkeit der DDS-Funktionalität wird dadurch erschwert, dass sie in der Regel durch das Zusammenspiel verschiedener DDS-Objekte erbracht wird, insbesondere durch die Objekte, die die QoS-Richtlinien

implementieren. Diese Objekte sind dabei nicht genau einer Funktionalität zugeordnet, sondern können für verschiedene Aspekte verantwortlich sein. Die sich dadurch ergebende Abhängigkeit zwischen den Objekten und der Funktionalität ist darüber hinaus noch von der jeweiligen Konfiguration abhängig, was insgesamt zu einem nicht trivialen Abhängigkeitsproblem führt, wenn die Funktionalität der Middleware für spezifische Anwendungen feingranular konfiguriert werden soll.

Die für ein Sensornetz relevante Funktionalität von DDS soll hier in zwei Teile untergliedert werden: in eine minimale Funktionalität, die immer benötigt wird, und in eine optionale, welche eingebunden werden kann, wenn eine Anwendung sie benötigt. Diese Teile sollen im Weiteren als Basis- und erweiterte Funktionalität betrachtet werden.

3.4.1.1 Basisfunktionalität

Die minimal notwendige Funktionalität von DDS für den Betrieb in einem Sensornetz ist die Basisfunktionalität und umfasst alles, was notwendig ist, um Daten zu beschreiben, zu finden und von den Datenquellen zu den Datensenken zu leiten. Abstrakt sei sie mit den folgenden Punkten beschrieben:

Daten definieren: Die Definition von Syntax und Semantik der auszutauschenden Daten im Informationsmodell des gesamten Middleware-Systems basiert auf dem Datenmodell von DDS. Dabei werden die Datentypen der auszutauschenden Daten deklariert und in einem zweiten Schritt mit einem eindeutigen Namen und optional zusätzlichen Eigenschaften zu einem DDS-Topic verbunden. Die Topics bilden als Informationseinheiten den möglichen Datenraum, in den DDS-Anwendungen agieren können [SH08].

Datenschnittstellen erzeugen: Aus der deklarierten Syntax der austauschbaren Datentypen werden die plattformspezifischen Anwendungsschnittstellen von DDS als Stubs erzeugt. Basierend darauf kann eine Anwendung Daten unter einem Topic veröffentlichen oder beziehen.

Datenrelation herstellen: Das Herstellen einer Relation zwischen einem definierten Topic und den Datenquellen, die dieses bedienen können, sowie den Datensenken, die dessen Daten benötigen (Publish-Subscribe-Paradigma), ermöglicht es der Middleware, die Daten zu den jeweiligen Empfängern zu leiten.

Daten austauschen: Das Übertragen der spezifischen Daten zwischen Daten-

Diese Gruppen von DDS-Funktionalität lassen sich getrennt betrachten, und es ist vorstellbar, dass Anwendungen Teilmengen unterschiedlicher Zusammensetzung benötigen könnten.

3.4.1.3 QoS-Funktionalität

Die im DDS-Standard definierten QoS-Richtlinien wurden in den Grundlage in Abschnitt 2.3 vorgestellt und sind spezifischen Objekten zugeordnet, die den Anwendungen als Schnittstelle für die Beeinflussung des Verhaltens der Middleware dienen. In [SCv08] wurden die QoS-Richtlinien von DDS Gruppen von QoS-Funktionalität zugeordnet. Aufbauend darauf soll auch hier eine erste grobe Einteilung der QoS-Funktionalität durchgeführt werden:

Datenverfügbarkeit: Zusicherung der Verfügbarkeit (Datenvorhaltung) und Aktualität (Gültigkeit) von Daten für aktuell und zukünftig assoziierte Datensenken.

Datenzustellung: Zusicherungen hinsichtlich der Art, Qualität und Zuverlässigkeit der Datenzustellung. Dies umfasst QoS-Richtlinien für Transaktionen und erweiterte Namensräume bezüglich Redundanz, (zeitlicher) Ordnung und zuverlässiger Übertragung von Daten.

Datenpünktlichkeit: Dieser Bereich umfasst zeitliche Zusicherungen für die Zustellung von Daten hinsichtlich ihrer Priorität und ihres geforderten Eintreffens bei einer Datensenke.

Ressourcen-Management: Hier sind Richtlinien enthalten, die den Ressourcenverbrauch auf den Knoten durch die Middleware begrenzen.

Konfiguration: Zusätzlich zu den bisher aufgeführten Bereichen definiert der DDS-Standard benutzerspezifische QoS-Richtlinien, deren Bedeutung von den Anwendungen abhängt.

Diese Einteilung ist noch zu grob, um als eine Modularisierung für die Middleware verwendet werden zu können; so fehlt noch die entsprechende Zuordnung der DDS-Objekte und ihrer spezifischen Funktionen.

Die einzelnen QoS-Richtlinien in den hier aufgeführten Gruppen wären eine Basis für eine mögliche Konfiguration. Hierbei tritt aber das Problem der Abhängigkeiten zwischen den Objekten im DDS-Standard besonders hervor, da einige QoS-

Richtlinien andere benötigen. Als Beispiel sei die zuverlässige Datenübertragung aufgeführt. Diese wird durch die QoS-Richtlinie „Reliability" repräsentiert, über die eine Anwendung festlegen kann, ob Daten zuverlässig übertragen werden sollen. In diesem Fall muss die Middleware Daten vorhalten, bis sichergestellt ist, dass alle Empfänger sie erhalten haben. Die Menge der vorzuhaltenden Daten wird aber zum einem über die QoS-Richtlinie „History" als weiche Grenze und die „Ressource Limits" als harte nach oben hin begrenzt. Der DDS-Standard sieht hierbei vor, dass alle QoS-Richtlinien auf einem Knoten implementiert sind, was für diese Arbeit nicht gewünscht ist.

Für die Lösung dieser Abhängigkeiten sollen zwei mögliche Lösungswege betrachtet werden. Der erste besteht darin, die QoS-Richtlinien in Module zu unterteilen und bei der Konfiguration eines Moduls für eine Anwendung alle abhängigen Module automatisch mit auswählen zu lassen. Der Anwendungsentwickler wäre dabei gezwungen, auch diese mit Parametern zu konfigurieren. Allerdings würde der Umfang der Middleware auf Grund der teilweise sehr weitläufigen Abhängigkeiten schnell stark zunehmen. Der Vorteil dieser Lösung ist, dass es nur notwendig ist, die Abhängigkeit der QoS-Richtlinien untereinander zu untersuchen. Des Weiteren können die QoS-Module unter der gesicherten Annahme der Verfügbarkeit der abhängigen Module mit dem entsprechenden Zugriff auf deren vollständige Funktionalität implementiert werden.

Für den zweiten hier betrachteten Lösungsweg muss die Untersuchung der Abhängigkeiten zwischen den QoS-Richtlinien und auch der anderen DDS-Funktionalität tiefer gehen. Diese müssen hierzu alle in atomare Funktionseinheiten zerteilt und deren Abhängigkeiten untereinander und von den möglichen Konfigurationsparametern festgestellt werden. Auf dieser Ebene kann für eine gewählte QoS-Richtlinie und ihre statische Konfiguration durch den Anwendungsentwickler festgestellt werden, welche Teile der Implementierung dazu nötig sind und entsprechend eine optimale Middleware generiert werden.

Die zweite Option ist sicherlich für eine DDS-Middleware für Sensornetze anzustreben, aber im Rahmen dieser Arbeit ist sie auf Grund des Umfangs für eine solch detaillierte Untersuchung der DDS-Funktionalität nicht umsetzbar. Von daher soll nur die erste vorgestellte Lösung weiter betrachtet werden.

3.4.1.4 Auswahl von Funktionalität

Für die weitere Untersuchung der DDS-Funktionalität soll von dieser eine Teilmenge ausgewählt werden, die für den Nachweis der Eignung des hier gewählten Ansatzes für ein DDS für Sensornetze mittels einer prototypischen Implementierung ausreichend ist. Hierfür sollen die folgenden für Sensornetze interessanten Bereiche genauer betrachtet werden:

- Basisfunktionalität
- Daten als Ereignisse
- QoS
  - Ressourcen-Management
  - Datenpünktlichkeit
  - Datenzustellung (Redundanz)

Mit einem DDS-System lassen sich auch ereignisbasierte Systeme realisieren, was für Sensornetze von Interesse ist, da dabei nur für Anwendungen relevante Daten übertragen werden. Dies kann gegenüber einem System, in dem Daten periodisch ermittelt und weitergeleitet werden, die verbrauchte Bandbreite reduzieren.

Die QoS-Richtlinien bezüglich des Ressourcen-Managements sind für drahtlose Sensorknoten essentiell und können sowohl im Betrieb als auch bei der Planung und Generierung der Middleware verwendet werden.

Das Wissen um die zeitlichen Anforderungen bezüglich der zu übertragenden Daten, also der Bereich der Datenpünktlichkeit, kann dazu genutzt werden, die Datenübertragung in der Middleware zu optimieren, um damit Energie und Bandbreite zu sparen. Der Aspekt der Redundanz aus der Gruppe der Datenverfügbarkeit ist interessant, da es in Sensornetzen oft vorkommt, dass Gruppen von Sensorknoten dieselbe oder eine ähnliche Art von Daten produzieren, was neben der generellen Redundanz auch für Optimierungen seitens der Middleware genutzt werden kann.

3.4.2 Untersuchung der ausgewählten Funktionalität

Die ausgewählte Funktionalität wird im Weiteren genauer betrachtet. Sie wurde bisher sehr abstrakt umrissen, so dass als erstes aufgeführt wird, welche spezifischen Funktionen und Komponenten des DDS-Standards ihr zugeordnet sind.

3.4.2.1 Basisfunktionalität

Die Basisfunktionalität umfasst alles, was notwendig ist, um ein minimales DDS-System aufzusetzen und Daten zwischen den Anwendungen auszutauschen. DDS-Funktionalität, die darüber hinaus geht, wird entweder auf dieser aufbauen oder sie um weitere Aspekte erweitern bzw. auch ersetzen.

Der DDS-Standard sieht vor, dass die austauschbaren Datentypen mit OMG IDL deklariert und durch einen IDL-Compiler spezifische Programmcode-Stubs erzeugt werden. Der generierte Programmcode für die Anwendungsschnittstellen sollte nur die Funktionen enthalten, die die Anwendung auch benötigt. Daher ist es notwendig, dass die Konfiguration der Middleware für eine Anwendung Parameter enthält, mit denen die benötigten DDS-Schnittstellen ausgewählt werden können.

Der IDL-Compiler kann kein eigenständiges Werkzeug sein, sondern muss ein Teil des gesamten Prozesses, der die Middleware-Implementierung generiert, sein. Es muss zusätzliche Konfiguration berücksichtigt werden und nur die wirklich benötigte Teilmenge der Schnittstellen-Implementierung soll Teil der generierten Middleware sein. Das Datenmodell von DDS, auf das die Erzeugung der DDS-Schnittstellen aufbaut, wurde bereits in Abschnitt 3.3 betrachtet.

Für die minimale DDS-Funktionalität werden die Anwendungsschnittstellen zum Zugriff auf die erzeugten Datentypen, zum Schreiben von Daten in einen DataWriter und zum Lesen von Daten aus einem DataReader benötigt. Bei einer statischen Konfiguration des DDS-Systems kann bei Generierung der Middleware direkt Programmcode für den optimierten Aufbau der DDS-Struktur und die Initialisierung des DDS-Systems erzeugt werden, womit sich die Menge der zu implementierenden DDS-Schnittstellen reduzieren lässt (es sei denn, ihr Vorhandensein wird konfiguriert). Für einen DataWriter verbleibt damit die `write`-Methode als Teil der Basisfunktionalität. Zum Lesen der Daten ist die einfachste Methode das Pollen der Anwendung an einem DataReader, welcher die Daten nur einmal zur Verfügung stellt.

Für das Finden von passenden Datenquellen („Discovery") sieht der DDS-Standard sogenannte „Built-In"-Topics vor, deren zugehörige DataReader und DataWriter auf jedem Knoten der Middleware enthalten sein sollen. Über diese Topics werden Informationen über die Knoten und die bedienten Topics verbreitet, womit dann die Middleware die passenden Verbindungen zwischen Datenquellen und Datensenken herstellen kann. Die vom DDS-Standard definierten Daten, die über diese Topics verteilt werden sollen, sind im Kontext von drahtlosen Sensornetzen jedoch zu umfangreich. Es kann daher notwendig sein, diese Funktionalität an die Bedürfnisse von Sensornetzen anzupassen. Um an dieser Stelle eine Kompatibilität bezüglich der DDS-Schnittstelle zu ermöglichen, sofern dies von einer Anwendung benötigt wird, könnte ein Modul dieses Verhalten emulieren und die Daten entsprechend abbilden.

Alternativ zu einer dynamischen Discovery ist es auch möglich, dass in der Konfiguration des Systems feste Relationen zwischen spezifischen Datenquellen und Datensenken festgelegt werden, welche in dem generierten Programmcode abgebildet werden. Diese Option würde offensichtlich weniger Aufwand und somit Ressourcen benötigen.

Für die Datenübertragung und auch den allgemeinen Betrieb der DDS-Middleware wird ein Kommunikationsprotokoll benötigt. Da dies ein eigenständiges Thema ist, das von der Untersuchung in diesem Abschnitt abhängig ist, wird es später getrennt betrachtet.

3.4.2.2 Daten als Ereignisse

Der DDS-Standard unterscheidet nicht zwischen einfachen Daten und Ereignissen; die Semantik der zu übertragenden Daten liegt im Zuständigkeitsbereich der Anwendung. Für eine ereignisbasierte Verwendung einer DDS-Middleware ist es aber wünschenswert, dass eine Anwendung über den Empfang eines Ereignisses zeitnah informiert wird. Die zuvor definierte Basisfunktionalität sieht nur eine Polling-Schnittstelle vor; für eine Erweiterung zur Unterstützung von Ereignissen sollen deshalb die DDS-Schnittstellen für Callback-Funktionen und das blockierende Lesen an einem DataReader betrachtet werden.

Für die Registrierung von Callback-Funktionen sieht der DDS-Standard die *Listener* vor, welche mit vielen DDS-Objekten verwendet werden können. Hierzu werden die Bedingungen, wann die Callback-Funktion durch DDS aufgerufen werden

soll, bei der Initialisierung mit angegeben. Für die hier vorgenommene Betrachtung ist nur das Eintreffen von neuen Daten bei einem DataReader von Interesse.

Die vom DDS-Standard vorgesehene Semantik zur Verwendung der *Listener* sieht nur vor, dass eine Anwendung über das Eintreten von Ereignissen, hier neuen Daten, informiert wird; wie sie darauf reagieren kann, ist nicht festgelegt. So kann sie direkt in der Callback-Funktion über die Polling-Schnittstelle die neuen Daten lesen oder auch erst später. Wenn diese Freiheit so eingeschränkt würde, dass die Daten nach dem Verlassen der Callback-Funktion von der Middleware als gelesen angesehen werden, könnte durch die ausschließliche Verwendung von Callback-Funktionen zum Lesen von Daten die Notwendigkeit des Vorhalten von Daten in einem DataReader wegfallen. Dies kann geeignet sein, den Ressourcenverbrauch der Middleware zu reduzieren, würde aber die Kompatibilität mit dem DDS-Standard verletzen.

Das blockierende Lesen kann ebenfalls an verschiedenen DDS-Objekten erfolgen. Über die Definition eines *WaitSet* legt eine Anwendung über Bedingungen fest, was für Daten oder Mitteilungen über Ereignisse sie lesen möchte. Analog zu dem *Listener* soll nur das Lesen von Daten aus einem DataReader betrachtet werden.

Mit dem *WaitSet* wird die DataReader-Schnittstelle erweitert, so dass die Polling-Funktionalität ersetzt werden kann. Damit ist es ebenfalls möglich, die Menge der vorzuhaltenden Daten zu reduzieren. Allerdings ergeben sich externe Abhängigkeiten für die Unterstützung von Nebenläufigkeit und die Signalisierung von Ereignissen.

3.4.2.3 Ressourcen-Management

Die QoS-Funktionalität von DDS für das Ressourcen-Management verteilt sich auf die QoS-Richtlinien *Resource Limits* und *Time Based Filter*. Sie ermöglichen es, den Ressourcenverbrauch von Speicher der Middleware und der Verarbeitungskapazität einer Anwendung nach oben zu begrenzen. Diese Funktionalität ist darum gerade für Sensornetze von Interesse und erfüllt die Anforderung 2 nach Ressourcenbewustsein einer Sensornetz-Middleware.

Die definierten Parameter der QoS-Richtlinie *Ressource Limits* legen fest, wie viele Datensätze ein DataWriter oder DataReader lokal vorhalten darf. Dies begrenzt den maximalen Speicherbedarf einer DDS-Instanz auf einem Knoten nach

oben. Dieser Parameter ist laut DDS-Standard nach der Instanziierung eines DDS-Objekts nicht mehr veränderbar und damit in dem Kontext dieser Arbeit für die Konfiguration des DDS-Systems und der Anwendungen relevant.

Diese QoS-Richtlinie beeinflusst die Implementierungen aller anderen, die auf die Vorhaltung von Daten angewiesen sind bzw. dafür sorgen, beispielsweise *Reliability*, *History* und *Ownership*.

Mit Hilfe der QoS-Richtlinie *Time Based Filter* kann eine Anwendung die minimal erlaubte Frequenz festlegen, mit der Daten bei einem DataReader eintreffen dürfen. Damit ist es es ihr auch dynamisch möglich, die Menge der zu verarbeitenden Daten an die vorhandenen Rechenkapazität anzupassen, beispielsweise wenn die Energieversorgung sich verändert. Die Parametrisierung der QoS-Richtlinie kann getrennt sowohl für die Planung und Konfiguration eines DDS-Systems und darin enthaltener Anwendungen, als auch im Betrieb von einem QoS-Module verwendet werden. Letzeres kann entweder, als allein stehendes Modul implementiert, empfangene Daten auf einem Knoten verwerfen oder als komplexere Lösung über einen Rückkanal in das Sensornetz die Datenrate der DataWriter reduzieren.

Der *Time Based Filter* beeinflusst allerdings andere QoS-Richtlinien, beispielsweise die *Reliability*-Richtlinie. Werden Daten verworfen, die zuverlässig übertragen werden sollten, ist es notwendig, dass eine Implementierung der *Reliability*-Richtlinie dies erkennt und angemessen reagiert.

3.4.2.4 Datenpünktlichkeit

Der DDS-Funktionalität für Datenpünktlichkeit werden die QoS-Richtlinien *Deadline*, *Latency Budget* und *Transport Priority* zugeordnet. Sie definieren Parameter, mit denen zeitliche Vorgaben bezüglich der austauschbaren Daten gemacht werden.

Die *Transport Priority* kann als allein stehendes Modul betrachtet werden, das einem DataWriter zugeordnet ist. Der im Betrieb veränderliche Parameter beschreibt eine Priorität, welche im Betrieb mit den Daten des jeweiligen DataWriter assoziiert ist und innerhalb der Verarbeitung der Middleware berücksichtigt wird. Sie kann eventuell an das unterlagerte Kommunikationssystem übergeben werden, sofern es eine Priorisierung von Nachrichten unterstützt. Es ergibt sich

damit eine Abhängigkeit zum einem für die interne Verarbeitung in der Middleware, was viele andere Komponenten beeinflussen kann, und zum anderen für eine Schnittstelle zu dem Kommunikationssystem, um die Priorität übergeben zu können.

Die QoS-Richtlinie *Deadline* wird für periodisch versendete Daten verwendet, und ihr Parameter legt die maximale Zyklusdauer fest, bis ein Datum veröffentlicht werden muss. Für diese QoS-Richtlinie werden immer zwei Seiten benötigt: die Datenquelle, die ihre Kapazität angibt, und die Seite der Datensenke, welche ihre Anforderungen festlegt. Diese QoS-Richtlinie kann daher bei der Planung des DDS-Systems verwendet werden, wenn dabei Wissen über die relevanten zeitlichen Werte besteht. Im Betrieb lässt sich der Parameter der QoS-Richtlinie auf zwei Arten verwenden. Die erste ist die Erkennung von Fehlern auf Seite der Datensenke, wenn keine Daten in der festgelegten Zeit eintreffen. Die andere ist die Optimierung des Datentransports durch die Middleware. Stehen beispielsweise Daten eher zur Verfügung, als eine Datensenke es fordert, kann deren Auslieferung verzögert werden, wenn sich damit Vorteile ergeben. Beide Optionen beeinflussen andere DDS-Funktionalität und könnten unabhängig voneinander für eine Anwendung konfiguriert werden.

Mit dem Parameter für das *Latency Budget* kann eine Anwendung festlegen, wie viel Zeit die Middleware für die Auslieferung der Daten hat. Der Parameter kann sowohl von der Datenquelle als auch von der Datensenke festgelegt werden. Analog zu der möglichen Verwendung der Parameter der *Deadline*-Richtlinie kann, sofern das Wissen über die zeitlichen Werte während der Konfiguration des DDS-Systems vorliegt, dieser Parameter für eine statische Optimierung der Middleware verwendet werden. Für den laufenden Betrieb ist diese Parametrisierung ebenfalls geeignet, der Middleware zu helfen, die Datenverteilung zu optimieren.

3.4.2.5 Datenzustellung

Der Bereich der Datenzustellung umfasst eine Vielzahl von QoS-Richtlinien und wird hier vorerst auf den Aspekt der Redundanz von Daten beschränkt, wofür die beiden QoS-Richtlinien *Ownership* und *Ownership Strength* zuständig sind. Diese sind gerade für Sensornetze interessant, da hier häufig Sensorknoten ähnliche Daten sammeln, was zum einem die Verfügbarkeit dieser Daten erhöht und zum anderen dazu genutzt werden kann, die Daten zusammenzuführen, um weniger Bandbreite zu verbrauchen.

Die Implementierung der beiden QoS-Richtlinien wird benötigt, um mit redundanten Daten umgehen zu können. Hierbei wird allen DataWriter-Instanzen, die ein Topic bedienen, ein *Ownership Strength*-Parameter zugewiesen, welcher die Durchsetzungsfähigkeit der Daten des jeweiligen DataWriters repräsentiert. Über die QoS-Richtlinie *Ownership* kann dann konfiguriert werden, ob entweder alle verfügbaren Daten oder nur die mit dem größten „Strength"-Wert zugestellt werden sollen.

Die Zuweisung eines *Ownership Strength*-Parameters könnte sowohl statisch während der Konfiguration der Middleware und Anwendungen geschehen, als auch auf Grund von Entscheidungen im laufenden Betrieb. Im letzteren Fall ist die entsprechende Funktionalität als Modul zu kapseln und über die Konfiguration auswählbar zu machen.

Die Verwendung dieser Redundanzfunktionalität kann auf vielfältige Art und Weise in der Implementierung einer Middleware geschehen. Die Filterung der Nachrichten kann zum Beispiel auf dem Empfängerknoten geschehen, während der Weiterleitung der Daten im Sensornetz oder auf einem zentralen Knoten. Diese verschiedenen Möglichkeiten können als getrennte Module implementiert und entsprechend bei der Konfiguration ausgewählt werden. Damit ergäben sich mehr Abhängigkeiten, als hier betrachtet werden können. Allgemein kann festgestellt werden, dass sich Auswirkungen auf die QoS-Richtlinien *Reliability*, *Deadline* etc. ergeben, da diese für die Daten eines Topics spezifiziert sind und daher von unterschiedlichen DataWriter erfüllt werden können.

3.5 Kommunikationsprotokoll

Nach der Untersuchung der benötigten Funktionalität für eine DDS-Middleware für Sensornetze sollen nun die Aspekte, die für ein Kommunikationsprotokoll benötigt werden, analysiert werden. Aus den allgemeinen Anforderungen an eine Sensornetz-Middleware (siehe Abschnitt 3.1.6) und der benötigten DDS-Funktionalität (Abschnitt 3.4.1.4) ergeben sich Anforderungen an das Kommunikationsprotokoll, welche in diesem Abschnitt als erstes aufgezeigt werden. Für die prototypische Implementierung im Rahmen dieser Arbeit ist ZigBee als Basis für ein Kommunikationsprotokoll vorgegeben, daher wird im Anschluss an die Anforderungen untersucht, wie die Funktionalität des DDS-Standards auf ZigBee abgebildet werden kann.

3.5.1 Anforderungen

Aus den allgemeinen Anforderungen an eine Sensorknoten-Middleware und der benötigten DDS-Funktionalität ergeben sich direkt die in der folgenden Übersicht aufgeführten Anforderungen an das Design eines Kommunikationsprotokolls:

1. Ermöglichung von Implementierungen mit geringem Ressourcenverbrauch
2. Modularisierung des Protokolls
    - Abbildbarkeit von unterschiedlichen Funktionalitätsteilmengen einer generierten Middleware auf eine entsprechende Teilmenge des Protokolls
    - Selbstbeschreibung der einzelnen Elemente des Protokolls in einer Nachricht
    - Erweiterbarkeit
3. DDS-Quality-of-Service-Eigenschaften
4. Discovery, hohe Dynamik im Netz
5. Klar abgegrenzte Abstraktionsschnittstelle für die Verwendung von unterlagerten Kommunikationssystemen
6. Festlegung einer effizienten Byte-Reihenfolge und Datenkodierung

Geringer Ressourcenverbrauch

Wie auch bei den bisherigen Untersuchungen dieser Arbeit bezüglich einer DDS-Implementierung für Sensornetze ist die Frage des Ressourcenverbrauchs für ein Middleware-Protokoll von großer Bedeutung. Dieses hat Einfluss auf den Verbrauch von Speicher, Bandbreite und Rechenkapazität, bei der Erzeugung, Verarbeitung, Sortierung und Weiterleitung der Nachrichten.

Wie bereits in der Einleitung zu dieser Arbeit beschrieben, ist ein Weg, diesen Ressourcenverbrauch minimal zu halten, die enge Verzahnung von Anwendung, Middleware und Protokoll, was zu Problemen der Erweiterbarkeit und Wiederverwertbarkeit von Anwendungen und Middleware führt. Im Rahmen dieser Arbeit soll diesem Problem durch die Generierung der Middleware begegnet werden.

Der Verbrauch von Bandbreite ist insofern von Interesse, da er in einem Sensornetz gleichbedeutend mit dem Verbrauch von Energie ist. Allerdings ist die verbrauchte Energiemenge nicht direkt proportional zu der zu übertragenden Datenmenge, da es gewisse Grundkosten für den Empfang und Versand von Daten gibt. Aus diesem Grund kann es vorteilhaft sein, wenn statt vieler kleiner unidirektionaler Nachrichten eine große mit den gesammelten Daten an eine Gruppe von Empfängern geschickt wird, auch wenn diese Daten enthält, die nicht alle benötigen.

Byte-Reihenfolge und Datenkodierung

Die Untersuchung der Hardware-Plattformen für Sensorknoten in Abschnitt 3.3.2.1 hat gezeigt, dass es keine einheitliche Byte-Reihenfolge der Prozessor-Architekturen in diesem Bereich gibt. Für das Middleware-Protokoll ist daher zu entscheiden, ob eine einheitliche Byte-Reihenfolge für die Datenübertragung gewählt werden soll oder ob die Information über die jeweils verwendete Byte-Reihenfolge ein Teil des Protokolls ist [TS03]. Bei der erster Option kann es zu einem unnötigen Konvertierungsaufwand kommen, wenn Knoten mit gleicher Architektur miteinander kommunizieren, deren Byte-Reihenfolge sich aber von der für die Datenübertragung gewählten unterscheidet. Dieses Problem würde zwar bei der zweiten aufgeführten Möglichkeit wegfallen, aber dafür würde das Protokoll komplizierter werden und die Information über die Byte-Reihenfolge müsste entweder in jeder Nachricht mitgeschickt werden, oder die Sensorknoten müssten die Information, welcher Knoten welches Format verwendet, einmal aushandeln und dann lokal vorhalten.

Neben der Byte-Reihenfolge gibt es bezüglich der zu wählenden Datenkodierung weitere Fragen zu untersuchen. So ist es zum einen möglich, Daten zu komprimieren oder in eine Repräsentierung zu überführen, die besser bzw. einfacher übertragbar ist. Dies würde die benötigte Bandbreite reduzieren können, dafür entstünde aber durch die ggf. aufwändige Umwandlung ein größerer Verbrauch an Speicher und Rechenkapazität. Auch ist es wahrscheinlich, dass eine solche Implementierung auf Grund der komplexeren Verarbeitung mehr Programmcode benötigt. Für den Entwurf eines Protokolls ist daher abzuwägen, ob die Vorteile des geringeren Bandbreitenverbrauchs die möglichen Nachteile aufwiegen.

Modularisierung

Es ist das Ziel dieser Arbeit, die Implementierung der Middleware für jede Anwendung auf einem Sensorknoten individuell mit nur der jeweils benötigten Funktionalität zu erzeugen. Das kann dazu führen, dass die Middleware auf den Sensorknoten über einen heterogenen Funktionsumfang verfügt. Das Kommunikationsprotokoll muss dem in dreierlei Hinsicht Rechnung tragen. Zum einem muss es selber modular sein, so dass die Implementierung für einen Sensorknoten ebenfalls nur die Teile enthält, die auch benötigt werden. Da es passieren kann, dass ein Sensorknoten eine Nachricht empfängt, zu deren Verarbeitung für ihn unbekannte Funktionalität benötigt wird, muss der gesamte Inhalt einer Nachricht des Protokolls soweit selbstbeschreibend sein, dass ein Knoten die für ihn relevanten und verarbeitbaren Teile einer Nachricht lesen und den Rest überspringen kann. Als drittes ist abzusehen, dass mit der zukünftigen Entwicklung der Middleware neue Funktionalität hinzukommen kann, daher muss das Protokoll flexibel und ausbaufähig gehalten sein und es ermöglichen, dass Sensorknoten mit unterschiedlichen Versionsständen der Middleware in einem Netzwerk kooperieren können.

DDS-Funktionalität

Die geforderte DDS-Funktionalität muss entweder durch das Protokoll der Middleware oder durch das unterlagerte Kommunikationssystem abgedeckt werden können. Für die definierten QoS-Richtlinien bedeutet dies am Beispiel der zuverlässigen Datenübertragung, dass sie entweder durch eine Implementierung des Protokolls selber oder durch eine unterlagerte Netzwerkschicht bereitgestellt und über QoS-Parameter des DDS-Standards konfiguriert werden kann.

Dasselbe gilt auch für den Aufbau des Middleware-Systems, die Verwaltung der Knoten, das Finden von Diensten und den Auf- und Abbau von Abonnements zwischen Datenquellen und Datensenken. Dieser Teil der Middleware-Funktionalität muss entweder durch die Definition des Protokolls abgedeckt oder durch eine Implementierung delegiert werden. Sollte es Teil der Protokolldefinition sein, ist zu untersuchen, ob die Verfahren, die der DDS-Standard hierfür spezifiziert, in dem Kontext eines Sensornetzes anwendbar sind.

3.5.2 Optionen für eine Abbildung von DDS auf ZigBee

Nun soll untersucht werden, wie der ZigBee-Standard als Basis für eine DDS-Middleware im Rahmen dieser Arbeit verwendet werden kann und welche Optionen es hierfür gibt. Die Eigenschaften und der Aufbau des ZigBee-Protokolls wurden bereits in den Grundlagen in Abschnitt 2.2 beschrieben und werden daher als bekannt vorausgesetzt.

Für die Abbildung von DDS auf ZigBee gibt es verschiedene Optionen, die sich in der verwendeten Menge an ZigBee-Funktionalität unterscheiden. Das Konzept von ZigBee ähnelt dem vom DDS insofern, als dass auch dort die für eine Anwendung transparente Zustellung von Daten an die passenden Empfänger möglich ist. Die Art der Definition von Syntax und Semantik der Daten und assozierter Geräte unterscheidet sich allerdings von DDS, so dass dort Einschränkungen der Abbildung möglich sein könnten.

3.5.2.1 Direkte Abbildung

Die erste Option ist eine möglichst direkte Abbildung von DDS auf ZigBee, wobei möglichst viel DDS-Funktionalitäten an ZigBee delegiert wird. Die ZigBee-Funktionalität des „Bindings" über die ClusterIDs und der indirekten Adressierung lassen sich direkt auf das Publish-Subscribe-System von DDS abbilden. Dabei entsprechen die ZigBee-Cluster den DDS-Topics und die Cluster-Parameter den Datentypen der Topics. Die DataWriter registrieren als ZigBee-Application-Objects die Cluster-Topics als *ausgehend* und die DataReader analog als *eingehend*. Mit dem automatischen Binding sorgt dann das ZigBee-Protokoll dafür, dass DataReader und DataWriter sich finden und die gesendeten Daten entsprechend zustellen, ohne dass die Middleware hierfür Funktionalität bereitstellen muss.

Für diesen Weg einer direkten Abbildung von DDS auf ZigBee muss daher ein ZigBee-Profil für DDS definiert werden. In diesem Profil werden dann die möglichen Cluster mit ihren Parametern und den entsprechenden Datentypen festgelegt und damit auch alle möglichen Topics in dem System, welches das Profil einsetzt. Eine spätere Erweiterung des Systems um neue Topics würde entsprechend eine Erweiterung des Profils notwendig machen. Hierbei wäre es sowohl möglich, jeweils ein eigenes Profil pro System bzw. Anwendung zu definieren als auch ein einziges für alle DDS-Systeme. Da bei einer standardkonformen Verwendung von ZigBee die Profil-IDs von der ZigBee-Allianz individuell zugeteilt

werden, wäre die zweite Möglichkeit eines Profils für alle DDS-Anwendungen einfacher umzusetzen.

Durch ein statisches Profil für DDS entsteht bei der Integration von verschiedenen Systemen oder Anwendungen in ein anderes oder ein neues System das Problem, dass die in den verschiedenen Systemen verwendete Identifizierung für die ZigBee-Cluster-Topics nirgends mehrfach verwendet werden darf. Andernfalls müsste das gesamte System und alle betroffenen Anwendungen angepasst werden, was in einem existierenden System nicht immer möglich ist. Wenn die Systeme jeweils eigene Profile verwenden, entsteht dieses Problem natürlich nicht. Allgemein wird mit dieser Lösung die Flexibilität von DDS bezüglich der Erweiterung und Integration von bestehenden oder neuen Anwendungen aufgegeben. Für das Datenmodell wurde diese Problematik bereits in Abschnitt 3.3.2.3 der Analyse festgestellt.

3.5.2.2 Teilweise Abbildung

Neben einer vollständigen Abbildung von DDS auf ZigBee, was die Flexibilität von ZigBee einschränken würde, ist eine Alternative, nur statische Teile abzubilden und die besonderen DDS-Aspekte als Aufgabe der Middleware anzusehen. Statische Teile von DDS sind bei diesen Ansatz zum Beispiel die festen Built-In-Topics, welche für das Finden von anderen DDS-Knoten und ihrer Kapazitäten benötigt werden. Der jeweilige Datenaustausch für ein spezifisches Topic kann dann über einen generischen ZigBee-Cluster erfolgen, der entweder keine genaue Vorgabe für die Datentypen macht oder eine Folge von Bytes annimmt. Damit ist es aber nicht möglich, die indirekte Adressierung von ZigBee für den Datenaustausch zu verwenden, so dass hierfür eine direkte Knotenadressierung durch die Middleware erfolgen muss.

Diese partielle Verwendung von ZigBee-Funktionalität würde es ermöglichen, ein statisches Profil für alle DDS-Systeme festzulegen. Allerdings ist hier nicht abschätzbar, ob eine solche Beantragung bei der ZigBee-Allianz erfolgversprechend ist.

3.5.2.3 Leichtgewichtige Abbildung

Neben dem Ansatz, von ZigBee ausgehend möglichst viel Funktionalität auf DDS zu übertragen, kann das Problem auch basierend auf den minimalen Anforderun-

gen von DDS an ein unterlagertes Kommunikationssystem betrachtet werden. Der DDS-Standard wurde, wie in den Grundlagen 2.3 beschrieben, implizit mit Hinblick auf UDP/IP als unterlagertes Kommunikationsprotokoll entwickelt und ist deshalb mit einer verbindungslosen und unzuverlässigen Datenübertragung zwischen den Knoten im Netzwerk einsetzbar. Das DDS-Kommunikationsprotokoll RTPS setzt darüber hinaus auf Multi- und Broadcast im Netzwerk, um effektiv Daten an mehrere Empfänger routen zu lassen. RTPS benötigt allerdings das Routing des unterlagerten IP-Protokolls, welches nicht Teil der RTPS-Protokollspezifikation ist. Alle darüber hinaus gehende Funktionalität wird durch den DDS-Standard definiert und durch die Middleware-Implementierungen und das RTPS-Protokoll bereitgestellt.

Aus Sicht des DDS-Standards würde daher von ZigBee das Folgende benötigt:

1. Direkte Adressierung der Knoten
2. Routing der Nachrichten
3. Empfang und Versand von Nachrichten als Datagramm
4. Unicast
5. Multicast
    - Mehrere Multicastgruppen (Bsp. eine pro Topic)
    - Dynamisches Hinzufügen und Entfernen von Knoten aus bzw. in die Multicastgruppe
6. Broadcast
7. Datenintegrität der Nachrichten

Diese Funktionalität wird auch von den verschiedenen ZigBee-Protokollschichten bereitgestellt. Eine direkte Adressierung der Knoten wird auf jeder Ebene unterstützt, womit auf die indirekte Adressierung von ZigBee verzichtet werden könnte. Die Zuordnung von Adressen zu Empfängern einer Nachricht wäre damit Aufgabe der DDS-Middleware. ZigBee sieht zwei Arten von Adressen vor: Eindeutige für jeden Knoten mit 64 Bit Länge und kürzere mit 16 Bit, die Geräten durch das ZigBee-Protokoll dynamisch zugewiesen werden (vgl. Grundlagen Abschnitt 2.1).

Die Middleware sollte zur Reduzierung des Speicherverbrauchs die Zuteilung der kürzeren Adressen von ZigBee nutzen.

Nachrichten können von ZigBee optional zuverlässig übertragen werden, wofür die Protokollimplementierung Daten vorhalten muss. DDS sieht optional die Funktionalität vor, Daten auf den Datenquellen für neue oder kurzfristig nicht erreichbare Empfänger vorzuhalten. Um doppelte Datenvorhaltung auf einem Sensorknoten zu vermeiden, sollte in diesem Fall daher DDS für die zuverlässige Datenübertragung zuständig sein.

Die Gruppenkommunikation von ZigBee wurde bereits in den Grundlagen in Abschnitt 2.2.2.3 beschrieben. Nachrichten können damit einer Gruppe zugestellt werden, welche über eine eigene Adresse referenziert wird. Diese Funktionalität könnte von einer DDS-Middleware genutzt werden, um sowohl Empfänger gleicher Daten zusammenzufassen, als auch als Broadcast-Gruppe, um DDS-Systeme und andere ZigBee-Geräte voneinander zu trennen.

Das Finden und Zusammenführen von Datenquellen und passenden Datensenken ist eine Basisfunktionalität von DDS. Hierfür sieht der Standard vor, dass alle DDS-Knoten einige feststehende DataReader und DataWriter eingebaut haben, über welche regelmäßig mitgeteilt wird, welche Knoten es gibt und welche Daten sie benötigen bzw. anbieten. Diese „Built-In-Topics" stehen dem Standard nach auch den Anwendungen zur Verfügung. Diese im Standard definierten Topics legen aber sehr umfangreiche Informationen fest, was die auszutauschenden Daten groß und sie somit für Sensornetze ungeeignet macht. Sofern dieses Verfahren auch für eine Sensorknoten-Middleware verwendet werden sollte, ist eine entsprechende Reduzierung der Daten notwendig. Die Verwendung dieses Verfahrens hat den Vorteil, dass keine spezielle ZigBee-Funktionalität benötigt wird, sondern über die bisher beschriebene Verwendung von ZigBee implementiert werden kann.

3.5.3 Auswahl einer Abbildung von DDS auf ZigBee

Die im Abschnitt beschrieben drei Optionen für eine Verwendung von ZigBee durch eine DDS-Middleware haben unterschiedliche Vor- und Nachteile. Zu beachten ist aber auch die Erfüllung der Anforderungen an ein Kommunikationsprotokoll, die in Abschnitt 3.5.1 festgelegt wurden, notwendig. Mit der wichtigste Aspekt dieser Anforderungen ist die beschränkte Verfügbarkeit von Speicher für

Programmcode und Daten auf kleinen autarken Sensorknoten. Daher soll als erstes ZigBee unter diesem Aspekt untersucht werden, gleichfalls im Hinblick auf die Frage, welche Möglichkeiten es gibt, diesen zu beeinflussen. Auf Basis dieser Ergebnisse kann dann eine Entscheidung getroffen werden.

3.5.3.1 Speicherverbrauch von ZigBee

Auf einem ZigBee-Gerät teilen sich Anwendungen und der ZigBee-Stack die verfügbaren Ressourcen. Der ZigBee-Standard definiert eine sehr umfangreiche Funktionalität, welche vor allem bei Geräten mit den ZigBee-Rollen Router und Coordinator entsprechend zu einem dominanten Speicher-Footprint des Protokoll-Stacks gegenüber den Anwendungen führt. Der Standard sieht nur für die Geräte der End-Device-Klasse eine reduzierte ZigBee-Funktionalität vor. Für die DDS-Middleware sind die ZigBee-Router von besonderem Interesse, da sie das Mesh-Netzwerk aufspannen und damit für die Bereiche der transparenten Datenverarbeitung und Aggregation in der Middleware zuständig sein könnten. Es ist offensichtlich, dass eine solche erweiterte DDS-Funktionalität den allgemeinen Speicher-Footprint der Middleware erhöhen wird, was damit aber in direkter Konkurrenz zu dem Speicherverbrauch eines vollständigen ZigBee-Stacks steht, insbesondere wenn die Middleware hierfür nicht auf Funktionalität von ZigBee zurück greifen kann.

Um eine grobe Abschätzung für den typischen Speicherverbrauch des ZigBee-Protokolls auf einem Sensorknoten durchzuführen, wurde die ZigBee-Implementierung von TI (zStack) in der Version 1.4.3 untersucht. Der zStack wird mit einer Reihe von Beispiel-Anwendungen ausgeliefert, welche übliche Anwendungsfälle von ZigBee abdecken. Eine Auswahl dieser Beispiele wurde für die Abschätzung des Speicherverbrauchs übersetzt, wobei als Zielplattform die CC2430 Sensorknotenplattform [Chi08] verwendet wurde. Für die Erzeugung der Anwendungen wurde die IAR-Entwicklungsumgebung in der Version 7.51 verwendet. Die Anwendungen werden damit ohne Debug-Informationen auf minimalen Speicherverbrauch optimiert und ansonsten mit der mitgelieferten Standardkonfiguration übersetzt. Der CC2430 wurde bereits in Abschnitt 3.3.2.1 als eine zurzeit übliche Sensorknotenplattform in Hinblick auf das mögliche Datenmodell einer DDS-Middleware untersucht. Die verwendete Konfiguration der Anwendungen ist für das „SmartRF04EB Evaluation Board” ausgelegt, welches als Test- und Entwicklungsplattform für den CC2430 dient und bereits nur die minimal notwendige Un-

	SampleLight	SimpleApp	GenericApp
EndDevice			
CODE	96099	80398	87100
DATA	4014	3687	3966
Router			
CODE	103252	105686	105955
DATA	5788	6843	5888
Coordinator			
CODE	104457	99137	101618
DATA	6797	6748	7043

Tabelle 3.6: Belegter Programm- und Datenspeicher bei zStack Beispielanwendungen in Byte.

termenge der ZigBee-Funktionalität einbindet. In Tabelle 3.6 ist der sich damit ergebende Speicherverbrauch der Beispielanwendungen dargestellt.

Für die Abschätzung wurden die Beispiele „SampleLight", „SimpleApp" und „GenericApp" ausgewählt und für jede ZigBee-Rolle, also EndDevice, Router und Coordinator übersetzt. Die Beispiele sind in den Spalten der Tabelle aufgelistet und der Speicherverbrauch, der sich für die jeweilige Rolle ergibt, in den Zeilen. Die Beispiele decken verschiedene Arten ab, wie Anwendungen den zStack verwenden können. „SampleLight" soll laut TI eine typische ZigBee-Anwendung darstellen, mit der Lampen gesteuert werden können. Diese Anwendung verwendet das fest definierte ZigBee-Profil für die Hausautomation und somit Funktionalität der ZCL (ZigBee-Cluster-Library). „SimpleApp" hingegen verwendet die TI-eigene Vereinfachung der ZigBee-Schnittstelle und das Beispiel „GenericApp" soll als Grundgerüst für eigene ZigBee-Anwendungen verwendet werden können. Der anwendungsspezifische Anteil der Beispiele ist einfach gehalten, so dass er an dem Speicherverbrauch vernachlässigbar wenig Anteil (< 1 KiB) hat.

Es zeigt sich aus den Werten der Abschätzung für den zStack, dass der grundsätzliche Speicherverbrauch von ZigBee den Speicher des betrachteten CC2430 fast vollständig füllt und dies interessanterweise sogar verhältnismäßig unabhängig von der gewählten ZigBee-Rolle für die Geräte ist. Anwendungen für die Rolle des ZigBee-Routers verbrauchen in allen gewählten Beispielen mehr als 100 KiB für Programmdaten, wobei der hier betrachtete CC2430 über 128 KiB Flash-Programmspeicher verfügt. Auch der Speicherbedarf für Daten füllt auf der hier betrachteten Hardware-Plattform mit der vorgegebenen Konfiguration den verfügbaren Speicher von 8 KiB fast vollständig aus, wobei dieser weniger statisch ist

und sich über eine entsprechende Konfiguration reduzieren ließe. Auf Basis der Daten aus der Analyse der Hardware-Plattformen in Abschnitt 3.3.2.1 kann man feststellen, dass ähnliche Hardware-Plattformen für kleine autarke Sensorknoten generell nicht über mehr Speicher verfügen.

Daraus ergibt sich die Schlussfolgerung, dass einer DDS-Middleware, die auf dem zStack aufbaut, Programmspeicher mit einer Größe von ca. 20 KiB und sehr wenig Datenspeicher zur Verfügung steht. Obwohl diese Schlussfolgerung nur auf der Abschätzung des Speicherverbrauchs von Beispielanwendungen für den zStack aufbaut, kann angenommen werden, dass sich Implementierungen anderer Hersteller für ihre jeweiligen Hardware-Plattformen in ähnlichen Dimensionen bewegen, da die zu implementierende Funktionalität durch den ZigBee-Standard vorgegeben ist und die Implementierungen auf die Zielplattformen optimiert werden.

3.5.3.2 Schlussfolgerung

Auf Grund der Abschätzung des zur Verfügung stehenden Speichers für eine DDS-Middleware auf einem typischen Sensorknoten und der geforderten Funktionalität, die diese erfüllen soll, ist anzunehmen, dass diese zusammen mit einer vollständigen Implementierung des ZigBee-Protokolls mehr Speicher benötigt, als auf autarken Sensorknoten zur Verfügung steht.

Der ZigBee-Standard sieht vor, dass Anwendungen nur Teile der vorgesehenen Funktionalität verwenden können. Diese sind wie in den Grundlagen beschrieben in Schichten und Module unterteilt und entweder immer notwendig oder optional. Sofern die Anwendungen die optionalen Komponenten nicht benötigen, müssen sie auch nicht eingebunden werden.

Dies bedeutet für die beiden ersten aufgeführten Optionen für eine Abbildung von DDS auf ZigBee, die möglichst viel ZigBee-Funktionalität nutzen sollen, dass es dafür entweder notwendig ist, eine andere Klasse von Sensorknoten zu verwenden oder die DDS-Funktionalität stark zu reduzieren. Die Alternative dazu wäre, nur eine Teilmenge von ZigBee zu nutzen, also die dritte, in Abschnitt 3.5.2.2 aufgezeigte Option.

Nach den Anforderungen in Abschnitt 3.1.6 soll die DDS-Middleware auch für autarke drahtlose Sensorknoten einsetzbar sein, damit scheidet die Option, eine

besser ausgestattete Hardware-Plattform zu verwenden, aus, da diese zur Bearbeitungszeit dieser Arbeit noch mehr Energie benötigen, als autarke Sensorknoten zur Verfügung haben (Siehe Abschnitt 3.3.2.1). In den vorherigen Abschnitten wurde auf Basis derselben Anforderungen die gewünschte DDS-Funktionalität für die zu entwickelnde Middleware umrissen. Durch die geplante individuelle Generierung soll es möglich sein, den Speicherverbrauch für eine spezifische Anwendung zu minimieren. Dennoch ist es sinnvoll, davon auszugehen, dass selbst die minimal notwendige Basis-Funktionalität eines DDS-Knotens mehr Speicher benötigt als verfügbar ist. Daher verbleibt als letzte Option die ausschließliche Verwendung der minimal von DDS benötigten Netzwerkfunktionalität und die Entfernung aller nicht benötigten ZigBee-Bestandteile.

3.5.4 Universelle Abstraktionsschnittstelle für unterlagerte Kommunikationssysteme

Im Abschnitt wurde festgelegt, dass von ZigBee nur die minimal von DDS benötigte Funktionalität verwendet werden soll. Diese kann aber auch von anderen Kommunikationssystemen, wie sie in Sensornetzen vorkommen können, und aus denen in Abschnitt 3.3.2.3 eine Auswahl betrachtet wurde, bereitgestellt werden. Der Vorteil einer definierten Abstraktionsschnittstelle des Middleware-Protokolls für die unterlagerten Kommunikationssysteme ist, dass diese relativ einfach gegen andere ausgetauscht werden können. Somit könnte die Middleware einfach erweitert werden, um Anwendungen in heterogenen Sensornetzen zu betreiben und somit verschiedene Netzwerkarchitekturen zu vereinen.

Die in Abschnitt 3.5.2.3 vorgestellte Option für eine Verwendung von ZigBee orientierte sich an den minimalen Anforderungen von DDS, die sich indirekt von UDP/IP ableiten. Für Sensornetze ist eine vollständige Trennung zwischen Anwendung, Middleware und Kommunikationssystem ungünstig, da es dabei zu einer mehrfachen Datenvorhaltung kommen kann. Zusammen mit den Anforderungen an ein Middleware-Protokoll sollte eine abstrakte Schnittstelle daher der Middleware mindestens Zugriff auf die folgenden Punkte gewähren:

- Bereitstellung des Zugriffs auf das Datenfeld in dem zu verwendenden spezifischen Nachrichten-Frame
- Abstraktion netzwerkspezifischer Adressen durch Kapselung

- Initiieren des Versands einer Nachricht an die übergebene gekapselte Adresse
- Registrierung einer Callback-Funktion für den Empfang von Nachrichten
- Bereitstellung der gekapselten Netzwerkadresse des Senders einer empfangenden Nachricht
- Gruppenkommunikation
  - Gekapselte Gruppenadressen
  - Anlegen einer Gruppe
  - Hinzufügen eines Knotens zu einer Gruppe
  - Entfernen eines Knotens aus einer Gruppe
- Bereitstellung einer gekapselten Broadcast-Adresse
- Optional: Übergabe einer Priorität für die zu sendende Nachricht

3.6 MDSD-Ansatz für die Generierung der Middleware

3.6.1 Einführung MDSD

Die Schlagwörter „Model-Driven Development" (*MDD*), „Model-Driven Engineering" (*MDE*), „Model-Driven Architecture" (*MDA*) oder „Model-Driven Software-Development" (*MDSD*) sind zur Zeit im Bereich der Software allgegenwärtig und werden teilweise unterschiedlich interpretiert, wobei die zu Grunde liegende Idee identisch ist: Die Entwicklung von Software grundsätzlich auf formalen Modellen mit unterschiedlichen Ebenen der Abstraktion aufzubauen und aus diesen den Anwendungsprogrammcode als Teil des Softwareentwicklungszyklus zu generieren. Einen guten Einstieg in MDSD und die grundlegenden Konzepte und verwendeten Werkzeuge gibt [SVEH07], in dem MDSD wie folgt definiert ist:

> Modellgetriebene Softwareentwicklung (Model Driven Software Development, MDSD) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen.

In der folgenden Beschreibung des MDSD-Ansatzes für die Entwicklung von Software wird sich auf [SVEH07] bezogen, sofern nicht anders angegeben.

Grundlegend für MDSD ist die Definition von Modellen, die das zu entwickelnde Anwendungssystem in relevanten Aspekten beschreiben und als Basis für die weitere Verarbeitung geeignet sind. Eine mögliche Modellierung findet immer im Kontext eines Problemraums, hier *Domäne*, statt und hat das Ziel, ihre Inhalte formal zu beschreiben. Für die Beschreibung der Inhalte werden Metamodelle verwendet, die die Struktur der Domänen formal definieren und eine abstrakte Syntax und statische Semantik besitzen. Ein Beispiel für ein Metamodell ist die „Unified Modeling Language" (*UML*) für die Modellierung von Software.

Der Unterschied einer abstrakten Syntax eines Metamodells zu einer konkreten ist, dass sie die Elemente des Metamodells und ihre Beziehung zueinander beschreibt und nicht, mit welcher konkreten syntaktischen Notation die Modelle definiert werden. Die statische Semantik wiederum besteht aus den Bedingungen (*Constraints*), welche festlegen, wann ein Modell *„wohlgeformt"* [SVEH07] ist. Ein Metamodell mit zusätzlich einer konkreten Syntax und darauf bezogenen Semantik ist eine domänenspezifische Sprache („Domain Specific Language", *DSL*), mit der Modelle für eine Domäne, als eine Art Programmiersprache, definiert werden können. Ein so beschriebenes Modell ist formal definiert und kann weiterverarbeitet und zum Beispiel für die Generierung von Programmcode verwendet werden. Für diese Aufgaben ist zu unterscheiden zwischen plattformabhängigen und -unabhängigen Modellen. Eine Plattform beschreibt eine konkrete Umgebung mit spezifischen Eigenschaften wie die Hardware- oder die Software-Umgebung, auf bzw. in der der generierte Programmcode ausgeführt werden soll, also einen spezifischen Lösungsraum. Ein Modell, das Inhalte in diesem Lösungsraum beschreibt ist ein plattformspezifisches Modell („Plattform Spezific Model", *PSM*). Im Gegensatz dazu ist ein formales Modell, das Inhalte eines abstrakten Problemraums beschreibt, plattformunabhängig und wird „Plattform Independent Model" (*PIM*) genannt. Eine Überführung eines PIM in ein PSM geschieht durch eine Modelltransformation.

Die Modelltransformation überführt ein formales Modell in ein anderes Modell oder in eine andere Repräsentierung der enthaltenen Informationen. Ist das Ergebnis der Transformation ein anderes Modell, spricht man von einer „Model to Model Transformation" (*M2M*), ist es Programmcode wird sie „Model to Code Transformation" (*M2C*) genannt bzw. es wird allgemein von „Generierung" gesprochen.

Die Model-Driven Architecture der OMG [MM03] beschreibt ein standardisiertes Verfahren, welches für MDSD verwendet werden kann, aber nicht alle Aspekte unterstützt [SVEH07]. Die MDA verwendet als Metametamodell für die formale Spezifikation eine Metamodells ausschließlich MOF (Meta Object Facility) und empfiehlt, als DSL UML zu verwenden.

3.6.2 Anforderungen an einen MDSD-Ansatz

Der allgemeine Anwendungsfall für eine Middleware für Sensornetze in Abschnitt 3.1.2 und die daraus abgeleiteten Anforderungen in Abschnitt 3.1.6 beschreiben eine Entwicklungsumgebung für Anwendungen, die auf einer Middleware basieren. Die Funktionalität der Middleware, die die Anwendungen benötigen, soll konfiguriert werden und zusammen mit weiteren Beschreibungen und Konfigurationen bezüglich der spezifischen Plattformen die Basis für die anwendungsspezifische Generierung der Middleware-Implementierung dienen. Die eigentlichen Anwendungen sollen dann auf der abstrakten und plattformunabhängigen Schnittstelle der Middleware aufbauen und somit von den heterogenen Sensorknoten-Hardware-Plattformen getrennt und portabel werden.

Für die Verwendung eines MDSD-Ansatzes bedeutet dies, dass es verschiedene Problemräume und eine Menge an möglichen Lösungsräumen gibt. Für die Modellierung wird daher für jeden Problemraum ein passendes Metamodell benötigt. Die möglichen Lösungsräume müssen ebenfalls modellierbar sein.

Aus den Begründungen der Anforderungen in Abschnitt 3.1.6, dem allgemeinen Anwendungsfall und den bisherigen Untersuchungen zu DDS in Abschnitt 3.3, 3.4 und 3.5 lassen sich die folgenden Bereiche, die konfiguriert oder modelliert werden müssen, ableiten:

- Aufbau bzw. Struktur des DDS-Systems
- Deklaration der Topic-Datentypen
- Konfiguration der benötigten DDS-Funktionalität
- Konfiguration der DDS-Komponenten, insbesondere QoS-Richtlinien
- Auswahl bzw. Beschreibung der Zielplattform mit vorhanden Ressourcen, Komponenten usw.

- Vorgabe von nichtfunktionalen Anforderungen der Anwendungen

Es ist möglich, dass sich Teile dieser Punkte zu gemeinsamen Problemräumen zusammenfassen lassen und mit einem Metamodell auskommen. Es ist aber dennoch davon auszugehen, dass die in dieser Arbeit zu entwickelnde Komponente, die die Middleware generiert, mehrere Modelle als Eingabe bekommt.

Die Komponenten, die die DDS-Funktionalität bereitstellen, haben, wie in in Abschnitt 3.4.1 festgestellt, untereinander Abhängigkeiten. Für den Generierungsprozess bedeutet dies, das das Eingangsmodell, dass das DDS-System beschreibt, erweitert werden muss. Allgemein ist offensichtlich, dass die verschiedenen Eingangsmodelle und Konfigurationen sich untereinander beeinflussen oder nur gemeinsam in Kombination für die Generierung verwendbar sind. Daher wird es notwendig sein, dies bereits auf Ebene der Metamodelle vorzusehen und Prozesse zu definieren, wie Modelle kombiniert und transformiert werden können.

Die Verwendung eines MDSD-Ansatzes kann als „State-of-the-Art" angesehen werden; viele Projekte mit ähnlichen Anforderungen verwenden solche Verfahren in unterschiedlichen Ausprägungen. In [LVCA+07], [ADBS09] und [KMSB08] werden Projekte beschrieben, die einen solchen Ansatz im Bereich der Sensornetze verwenden.

Eine Kernkomponente der verschiedenen MDSD-Ansätze ist es, Werkzeuge zu verwenden, die den Anwendungsentwickler bei seiner Arbeit unterstützen. Auch die Fokussierung dieser Art der Software-Entwicklung durch die OMG mit ihrem MDA-Ansatz führten dazu, dass es eine Reihe von fertigen Werkzeugen und Frameworks gibt, die für MDSD-basierte Projekte verwendet werden können.

Im Weiteren soll untersucht werden, auf welche Werkzeuge oder Frameworks in dieser Arbeit aufgebaut werden kann. Des Weiteren ist es von Interesse, ob es in der Literatur Anwendungsfälle gibt, die den in dieser Arbeit betrachteten ähneln, und die mit einem MDSD-Ansatz gelöst wurden, damit man auf diesen Ergebnissen aufbauen kann.

3.6.3 MDSD-Werkzeuge

In [SVEH07] wird MDSD grundlegend beschrieben und in [LVCA+07], [ADBS09], [KMSB08] und [VSK05] werden Ansätze beschrieben, in denen MDSD für die

Entwicklung von Anwendungen für Sensornetze bzw. eingebettete Systeme verwendet wurde. In all diesen Quellen werden Teile des Eclipse-Projektes[1] als Hilfsmittel oder Basis verwendet, um unter anderem Metamodelle zu definieren, Modelle zu parsen, zu transformieren und am Ende Programmcode zu erzeugen.

Das Eclipse Modeling Framework (*EMF*) [Fou] ist ein Framework, um aus strukturierten Modellen Programmcode erzeugen zu können und wird in vielen Projekten eingesetzt [SVEH07]. EMF verwendet das Metametamodell *Ecore*, mit dem Metamodelle definiert werden können, deren abgeleitete Modelle mit EMF und weiteren Eclipse-Werkzeugen verarbeitet werden können. Mit dem UML2-Projekt stellt Eclipse auch eine „Implementierung" des Metamodells des UML2-Standards der OMG bereit, die Ecore verwendet. UML-Modelle, die mit dieser Metamodell-Implementierung definiert sind, lassen sich damit von EMF und weiteren Eclipse-Werkzeugen verarbeiten.

Das ursprünglich unabhängige openArchitectureWare-Projekt ist ein spezifisches Framework für MDSD und nun Teil von EMF [Fou]. Es stellt die folgenden Werkzeuge für jeden Schritt eines MDSD-Ansatzes bereit [SVEH07]:

Xtext: Ein Werkzeug, mit dem textuelle DSLs definiert werden können. Die Definition umfasst konkrete als auch abstrakte Syntax und verwendet eine Erweiterung der Backus-Naur-Form zur Beschreibung. Aus der Definition der DSL werden dann ein entsprechendes Metamodell, Parser für die DSL und ein spezifischer Editor für Eclipse erzeugt.

Xtend: Xtend ist eine funktionale Sprache, mit der Metamodelle erweitert oder verändert werden können. Sie kann auch für die Modelltransformation eingesetzt werden.

XPand: XPand ist eine Template-Sprache, die statisch typisiert und funktional ist. Sie unterstützt die Modularisierung von Templates, Polymorphismus und Aspekte.

Check: Eine deklarative Validierungssprache für Modelle, um die Einhaltung von Constrains zu überprüfen. Die Sprache Check verwendet dasselbe Typsystem wie auch Xtend und Xpand.

[1] http://www.eclipse.org

3.6.4 MDSD-Architektur

Für diese Arbeit wird ein MDSD-Ansatz benötigt, der auf Basis von unterschiedlichen Konfigurationen bzw. Modellen die benötigte Middleware- und Sensorknoten-Funktionalität auswählen und eine auf minimalen Ressourcenverbrauch optimierte Middleware-Implementierung generieren kann. In [SVEH07] bzw. [VSK05] wurde beispielhaft für ein aus Komponenten bestehendes eingebettetes System eine MDSD-basierte Architektur vorgestellt, deren Generierungsprozess dem in dieser Arbeit benötigten ähnelt und auf Werkzeugen des Eclipse-Projektes aufbaut. Die schematische Darstellung der Architektur ist in Abbildung 3.3 dargestellt und soll nun untersucht werden.

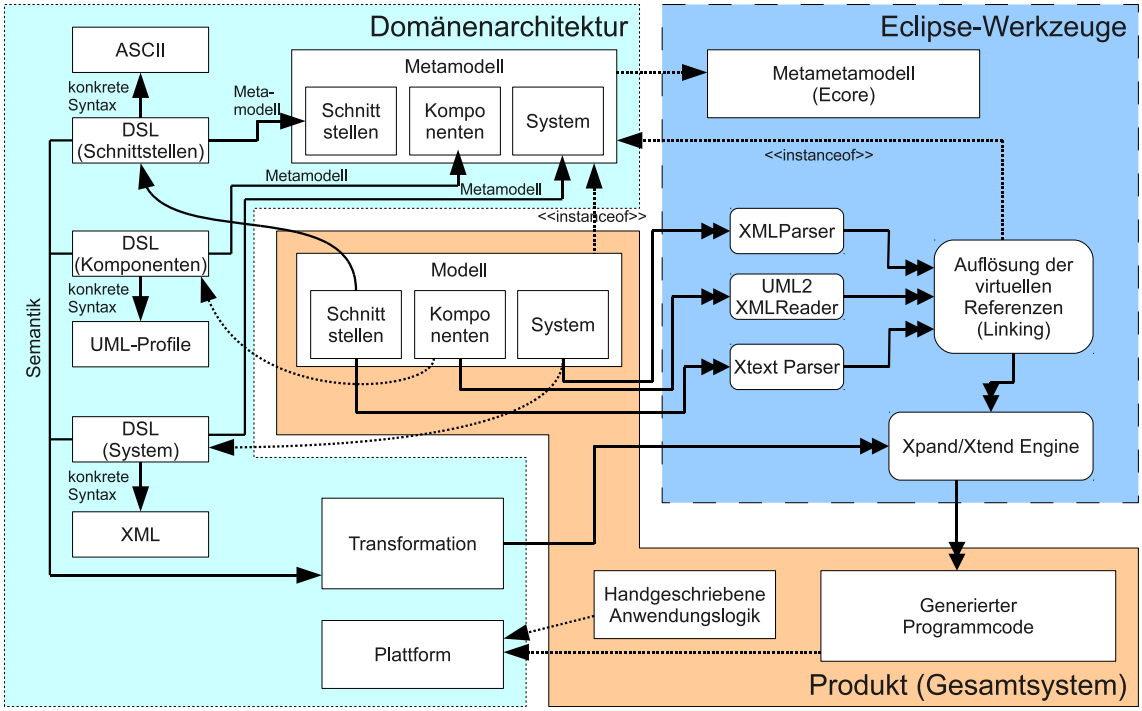

Abbildung 3.3: Ein Beispiel für eine MDSD-Architektur zur Generierung einer komponentenbasierten eingebetteten Anwendung (nach [SVEH07]).

Die dargestellte Architektur besteht aus drei Teilen: der Domänenarchitektur, also die Teile, die spezifisch für eine Domäne sind, den Eclipse-Werkzeugen und der eigentlichen Anwendung. Die Domänenarchitektur enthält Informationen über die Domäne, also die Metamodelle und zugehörigen DSLs, sowie das Wissen über die Plattform und die Semantik für die Transformation der Metamodelle. Mit Hilfe

der Eclipse-Werkzeuge wird die eigentliche Generierung der Anwendung durchgeführt.

Für die Modellierung der Anwendung werden die Schnittstellen mit einer textuellen DSL, die Komponenten mit einem UML-Profil und das System mit einer XML-basierten DSL beschrieben, die alle dem jeweiligen Metamodell entsprechen. Die Definitionen der drei unterschiedlichen anwendungsspezifischen Modelle werden mit Parsern von Eclipse getrennt eingelesen und auf Basis des gemeinsamen Metamodells in ein Modell transformiert. Dieses Modell wird mit in der Xtend-Sprache beschriebenem domänenspezifischen Wissen weiter transformiert und mittels Xtend-Templates in den generierten Programmcode überführt. Dieser ergibt zusammen mit der handgeschriebenen Anwendungslogik die Anwendung.

Dieser in [SVEH07] beispielhaft aufgezeigte Prozess entspricht dem, der für dieser Arbeit benötigt wird. Daher könnte die dort beschriebene Architektur für den Entwurf als Basis verwendet werden.

3.7 Designentscheidungen

Auf Basis der durchgeführten Analyse werden nun die Designentscheidungen für die Entwicklung einer DDS-basierten Middleware für Sensornetze festgelegt, deren Programmcode anwendungsspezifisch generiert werden soll.

Festlegung eines Namens für die Middleware

Als Namen für die zu entwickelnde DDS-basierte Middleware für Sensornetze soll „sDDS” verwendet werden. Dies steht für „sensor network Data Distribution Service” und soll einerseits die Implementierung der Schnittstellen des DDS-Standards herausstellen und die Zielplattform der Sensornetze betonen.

MDSD-Ansatz

In dieser Arbeit soll der in Abschnitt 3.6 untersuchte MDSD-Ansatz für die Programmcodegenerierung auf Basis einer formalen Modellierung des Problemraums verwendet werden. Für die Modellierung der einzelnen Teilaspekte der sDDS-Middleware sollen jeweils eigenständige Metamodelle entworfen werden,

damit die Modellierung getrennt mit unterschiedlichen Werkzeugen erfolgen kann.

Das Beispiel für eine MDSD-basierte Architektur, das in Abschnitt 3.6.4 analysiert wurde, soll als Grundlage für die Architektur in dieser Arbeit verwendet werden. Dasselbe gilt für die in Abschnitt 3.6.3 untersuchten Werkzeuge des Eclipse-Projekts, die für die Verwendung in einem MDSD-basierten Software-Entwicklungsprozess vorgesehen sind.

Prototypische Implementierung

Die Implementierung soll in dieser Arbeit nur prototypisch erfolgen. Die Kernelemente der Anforderungen, die in Abschnitt 3.1.6 der Analyse aufgestellt wurden, sind soweit zu implementieren, dass es möglich ist, den gewählten Ansatz auf seine grundsätzliche Eignung zur Erfüllung dieser Anforderungen zu untersuchen. Hierbei soll aber der vollständige MDSD-basierte Software-Entwicklungszyklus durchlaufen werden können und es soll eine auf der Zielplattform lauffähige Middleware-Implementierung erzeugt werden. Diese Reduktion ist notwendig, da die untersuchten Aspekte zu umfangreich sind, als dass eine weitergehende Umsetzung im Rahmen dieser Arbeit möglich ist.

Datenmodell

Die Anforderungen an ein Datenmodell für die sDDS-Middleware wurden in Abschnitt 3.3 untersucht. Basierend auf den Ergebnissen aus Abschnitt 3.3.4 sollen für die prototypische Implementierung der sDDS-Middleware die folgenden Datentypen im Datenmodell enthalten sein:

- Ganzzahlen
- Strukturen (`struct`) zur Bildung von Topics zugeordneten Datentypen
- Zeichenketten mit fester und beschränkter Anzahl an Zeichen
- Sequenzen mit fester Anzahl an Elementen
- Einzelne Bytes und ein boolescher Datentyp

DDS-Funktionalität

Die für diese Arbeit relevante Funktionalität des DDS-Standards wurde in Abschnitt 3.4.1 untersucht. Aus dieser Menge soll die in Abschnitt 3.4.2.1 untersuchte Basisfunktionalität weiter betrachtet werden, da sie zwingend notwendig ist, um einen lauffähigen Prototypen zu implementieren. Der zu entwickelnde Prototyp der Middleware soll in der Lage sein, Daten verschiedener Topics zwischen unterschiedlichen Sensorknoten auszutauschen. Darüber hinaus sollen die QoS-Richtlinien *History* und *LatencyBudget* unterstützt werden und als Beispiel für die generelle Unterstützung von QoS-Richtlinien des DDS-Standards dienen.

Der DDS-Standard sieht vor, dass aus einer Beschreibung der auszutauschenden Datentypen eine angepasste Anwendungsschnittstelle für den Austausch dieser Datentypen generiert wird. Dieser Vorgang soll in dieser Arbeit in den MDSD-Ansatz integriert werden, da die auszutauschenden Datentypen, wie in Abschnitt 3.2.3 festgestellt wurde, einen signifikanten Einfluss auf den Ressourcenverbrauch einer Middleware-Implementierung haben können.

Die prototypische sDDS-Middleware soll die implementierten Teile der DDS-Schnittstelle konform zu dem DDS-Standard umsetzen, damit Anwendungen, die auf dieser Schnittstelle aufsetzen, portabel sind.

Kommunikationsprotokoll

ZigBee ist das Kommunikationsprotokoll, auf das der Prototyp der sDDS-Middleware in dieser Arbeit aufsetzen soll. In Abschnitt 3.5.2 wurden verschiedene Optionen für eine Verwendung von ZigBee für eine DDS-Implementierung untersucht. In Abschnitt 3.5.3.2 wurde festgestellt, dass eine leichtgewichtige Abbildung unter den in dieser Arbeit geltenden Bedingungen am besten geeignet ist. Daher soll im Weiteren diese in Abschnitt 3.5.2.3 vorgestellte Option verfolgt werden.

Des Weiteren wurde in Abschnitt 3.5.4 eine universelle Abstraktionsschnittstelle für ein unterlagertes Kommunikationssystem beschrieben, auf dem die sDDS-Middleware aufbauen könnte. Der weitere Entwurf von sDDS soll die geforderten Eigenschaften dieser Schnittstelle erfüllen und die Unterstützung von verschiedenen, möglichst unterschiedlichen Kommunikationssystemen vorsehen.

Weitere Funktionalität

Die Möglichkeit, die Middleware-Schnittstelle auch für die Abstraktion von Sensor- bzw. Aktor-Hardware der Sensorknotenplattformen zu verwenden, wie es in Abschnitt 3.1.3 überlegt wurde, soll nicht weiter betrachtet werden. Dies ist ein eigenständiger Aspekt, der unabhängig von der notwendigen Basisfunktionalität der sDDS-Middleware ist.

Allerdings soll eine abstrakte und universell einsetzbare Schnittstelle für die Bereitstellung von Betriebssystemfunktionen entworfen werden, auf die sowohl die Middleware-Implementierung als auch Anwendungen aufsetzen können. Die Entscheidung, ob für die Implementierung dieser Funktionen entweder auf vorhandene Funktionalität der Zielplattform abgebildet oder durch den MDSD-Ansatz bei Bedarf generiert wird soll im MDSD-Prozess zur Generierung der Middleware-Implementierung getroffen werden.

Kapitel 4

Design

4.1 Grobarchitektur

Basierend auf den Anforderungen in Abschnitt 3.1.6 und den Designentscheidungen in Abschnitt 3.7 dieser Arbeit werden in den folgenden beiden Abschnitten zwei Architekturen entworfen. Basierend auf dem in Abschnitt 3.6 betrachteten MDSD-Ansatz wird in Abschnitt 4.1.1 ein Framework entworfen, das auf Basis von anwendungsspezifischen Anforderungen den Programmcode für eine angepasste DDS-basierte Middleware-Implementierung generiert. Dieses Framework wird im weiteren sDDS-Middleware-Framework oder auch nur Middleware-Framework genannt.

Die Grobarchitektur der DDS-basierten Middleware, die von dem sDDS-Middleware-Framework generiert werden soll, wird in Abschnitt 4.1.2 vorgestellt. Wie in dem Abschnitt 3.7 der Designentscheidungen festgelegt, hat die Middleware den Namen sDDS. Im Folgenden wird der Begriff sDDS-Middleware oder sDDS für die allgemeine Middleware bzw. ihr Konzept verwendet. Das anwendungsspezifische Ergebnis, aus dem Generierungsprozess des sDDS-Middleware-Frameworks hingegen wird als sDDS-Middleware-Implementierung oder sDDS-Implementierung bezeichnet.

4.1.1 Middleware-Framework

Nach der Analyse der Anforderungen an eine Middleware für drahtlose Sensornetze in Abschnitt 3.1.6 soll der Entwickler von „Werkzeugen” bei der Planung

und Konfiguration von Systemen basierend auf der sDDS-Middleware unterstützt werden. In Abschnitt 3.6 der Analyse wurde der MDSD-Ansatz auf seine Eignung für diese Arbeit untersucht. In den Designentscheidungen in Abschnitt 3.7 wurde daraufhin festgelegt, dass im Rahmen dieser Arbeit ein solcher MDSD-Ansatz für die Entwicklungs eines *Middleware-Framework* verwendet werden soll, das in der Lage ist aus einer anwendungsspezifischen Modellierung eine angepasste DDS-basierte Middleware-Implementierung mit dem Namen sDDS zu erzeugen. Die Architektur des Frameworks und die verwendeten Prozesse für die Generierung sollen sich den Designentscheidungen (vgl. Abschnitt 3.7) nach an dem in Abschnitt 3.6.4 der Analyse betrachteten Beispiel für eine MDSD-basierte Architektur orientieren.

Die Abbildung 4.1 zeigt die Architektur des Middleware-Frameworks, welches die sDDS-Implementierungen generiert. Der Aufbau ist ähnlich dem des MDSD-Beispiels in Abschnitt 3.6, und es werden die in der Analyse aufgeführten „State-of-the-Art-Werkzeuge" des Eclipse-Projektes verwendet.

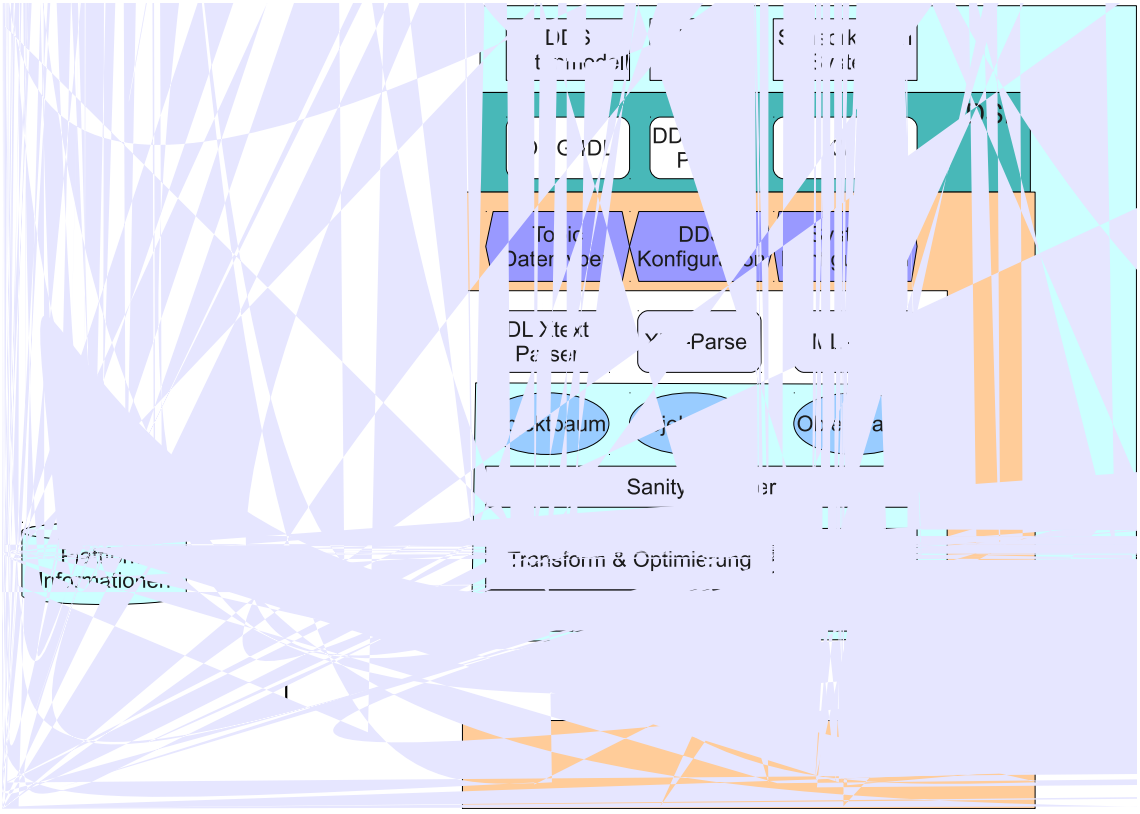

Abbildung 4.1: MDSD-basierte Architektur für das Middleware-Framework

Das Framework für die Generierung der sDDS-Implementierung besteht aus mehreren Teilen, die zusammen eine „Toolchain" für Modellierung, Konfiguration, Optimierung, plattformspezifischen Anpassung und Generierung des Programmcodes der Middleware sDDS bilden.

Für die Modellierung des anwendungsspezifischen sDDS-Systems und weiterer benötigter Komponenten werden jeweils domänenspezifische Metamodelle bzw. DSLs verwendet (vgl. Abschnitt 3.6.4). Für dasselbe Metamodell können unterschiedliche DSLs verwendet werden, für die Datentypen beispielsweise ist die original OMG IDL oder ein XML-basiertes Schema verwendbar. Es ist auch möglich, aber hier nicht weiter betrachtet, das die Eingabe der Modelle durch spezielle Editoren des Frameworks unterstützt wird.

Als nächster Schritt der Toolchain werden die definierten Modelle durch passende Parser für die DSLs eingelesen und in eine interne Objekt-Repräsentierung überführt, die das jeweilige Metamodell widerspiegelt. Auf dieser Basis können die Modelle auf die Einhaltung von systemspezifischen Bedingungen der Zielplattformen und generell auf Konsistenz geprüft werden.

Nach der Überprüfung der Modelle wird im nächsten Schritt aus ihnen die Struktur der Middleware aufgebaut und hierzu die minimal notwendige Funktionalität und entsprechender Module ermittelt. Durch Modelltransformationen werden diese Informationen in die bestehenden Modelle eingefügt und diese in einem einzigen Gesamtmodell vereinigt. Ein Transformationsschritt dabei ist es, weitere Informationen über die Zielplattform zu verwenden, um ein PSM zu erzeugen.

Für die unterstützten Plattformen enthält das Framework angepasste Programmcode-Templates, deren Struktur in Relation zu dem Metamodell des jeweiligen PSM steht. Aus diesen Templates und dem PSM erzeugt das Framework im letzten Schritt die angepasste sDDS-Implementierung mit den von den Anwendungen benötigten Funktionalitäten und DDS-Schnittstellen, sowie für die Middleware oder auch die Anwendungen notwendige Systemsoftware. Die Übersetzung des generierten Programmcodes und der Anwendungen ist danach Aufgabe des Entwicklers oder Teil anderer Werkzeuge und wird in dieser Arbeit nicht weiter betrachtet.

Das Middleware-Framework soll auf den Werkzeugen des Eclipse-Projekts aufbauen und daher das Metametamodell „Ecore" verwendet, um die Metamodelle zu definieren und die Modelltransformationenen mit dem Werkzeug „Xtend" durchzuführen. „Xtext" wird für die Beschreibung der spezifischen DSLs und für

die Erzeugung der entsprechenden Parser verwendet. Für die Generierung des Programmcodes wird das „Xpand”-Werkzeug verwendet. Die Module der sDDS-Middleware werden dazu in der „Xpand” eigenen funktionalen Sprache als Programmcode-Templates definiert.

Auf Basis der zu verwendenden Eclipse-Werkzeuge und der vorgestellten Architektur werden in Abschnitt 4.2 die notwendigen Metamodelle für die Modellierung des sDDS-Systems und der abhängigen Komponenten definiert. In der Implementierung (Abschnitt 5.6.2.2) werden für diese Metamodelle dann DSLs unter Verwendung des Xtext-Werkzeuges festgelegt. In Abschnitt 4.3 wird der Software-Entwicklungsprozess genauer vorgestellt, der mit diesem Middleware-Framework vorgesehen ist und dem Anwendungsentwickler unterstützen soll.

4.1.2 sDDS-Middleware

Das im vorherigen Abschnitt beschriebene Middleware-Framework erzeugt für jede Anwendung (oder jedes Anwendungssystem) eine spezielle, angepasste sDDS-Implementierung, welche in drahtlosen Sensornetzen verwendbar ist. Der Entwurf für die Architektur der sDDS-Middleware unterliegt daher der besonderen Anforderung, dass verschiedene, teilweise gegensätzliche Anforderungen erfüllt werden müssen (siehe Abschnitt 3.1 und 3.5). Um auch auf autarken drahtlosen Sensorknoten lauffähig zu sein, muss der Ressourcenverbrauch der sDDS-Implementierung klein sein, aber gleichzeitig die Anwendungsschnittstelle des DDS-Standards implementieren, so dass Anwendungen einfach portiert werden können. Die gesamte Architektur soll darüber hinaus so modular angelegt sein, dass einzelne Funktionen hinzugefügt oder weggelassen werden können (Abschnitt 3.2.3) und das das Kommunikationsprotokoll von dem unterlagerten Kommunikationssystem getrennt ist (Abschnitt 3.5.1). Gleichzeitig ist es wünschenswert, dass durch die Generierung in der endgültigen Implementierung keine Grenzen zwischen den Modulen und Komponenten bestehen und sie Ressourcen gemeinsam nutzen und damit Redundanzen, insbesondere bei identischen Daten im Speicher, vermieden werden. Auch die Heterogenität der existierenden Sensorknoten-Hardware-Plattformen ist zu berücksichtigen, damit die Middleware diese Systeme für den Anwender transparent verbinden kann. Es ist offensichtlich, dass bei dem Entwurf einer Architektur für eine Middleware, die all dies leisten soll, Kompromisse notwendig sind.

Abbildung 4.2 zeigt die Grobarchitektur der generischen sDDS-Middleware, welche ein Kompromiss zwischen einer Möglichkeit zur optimalen Ausnutzung der Ressourcen eines Sensorknotens und der Verwendung einiger Abstraktionen und generischer Teile ist.

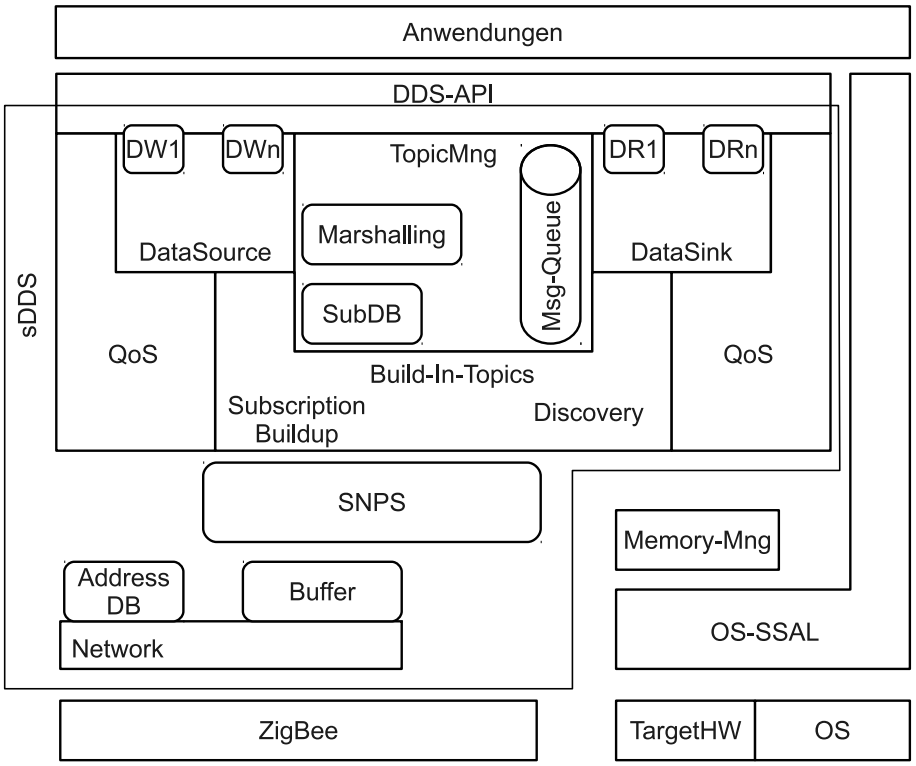

Abbildung 4.2: Grobarchitektur der sDDS-Middleware für einen Sensorknoten mit ZigBee

Auf dieser Ebene der Architektur ist noch keine Modularisierung der DDS-Funktionalität ausgeführt, sondern sie enthält die Komponenten der DDS-Basisfunktionalität, welche ein minimales DDS-System ausmacht. Diese Komponenten sind:

- DataSource
- DataSink
- Network und Protokoll
- Topic Management
- QoS für DataSource und DataSink

- Discovery und Subscription-Verwaltung
- Operating System – System Software Abstraction Layer (OS-SSAL)

Die Anwendungen, die die Middleware verwenden, greifen über die jeweilige plattformspezifische Anwendungsschnittstelle des DDS-Standards auf die Middleware zu, wie es in Anforderung 4 in Abschnitt 3.1.6 gefordert wird. Dafür wird das in den Grundlagen 2.3 beschriebene Objektmodell von DDS nicht auf die Architektur der Middleware übertragen. Stattdessen entsprechen die Architekturkomponenten Rollen, wie sie in einem DDS-basierten Sensornetz vorkommen. Diese Komponenten implementieren die entsprechenden Teile der DDS-Schnittstelle und emulieren für die Anwendungen das Verhalten der jeweiligen DDS-Objekte. Diese Zusammenfassung von ähnlicher Funktionalität in einer Komponente soll es ermöglichen, dass eine Implementierung von sDDS effektiv mit Ressourcen umgehen kann und dabei helfen, parallele Strukturen zu vermeiden.

Die DataSource-Komponente nimmt in einem Sensornetz die Rolle einer Datenquelle ein und die DataSink-Komponente die einer Datensenke. Beide implementieren die DDS-Schnittstelle für die Anwendungen und sorgen für die Annahme, Verarbeitung, Verteilung und Bereitstellung der Daten.

Das Topic Management verbindet DataSource und DataSink sowohl lokal als auch im gesamten sDDS-System. Lokal enthält es die Implementierung für die Verarbeitung der Topic-spezifischen Datentypen und verwaltet die aufgebauten Subscriptions und die ggf. vorzuhaltenden Daten. Die Implementierung der Built-In-Topics und ihre interne Verwendung für den Auf- und Abbau von Subscriptions sind dem Topic Management unterlagert und stehen den anderen Komponenten zur Verwendung bereit.

Das Protokoll SNPS ist eine eigenständige Komponente, die von den anderen verwendet wird, um SNPS-Nachrichten zu erzeugen oder zu verarbeiten. Sie setzt auf der in Abschnitt 3.5.4 umrissenen Schnittstelle zum Kommunikationssystem auf, welche von der Network-Komponente implementiert wird.

Unabhängig von der sDDS-Middleware selber ist die Abstraktionsschicht OS-SSAL, welche generische Betriebssystemfunktionalität bereitstellt und sowohl von sDDS als auch von Anwendungen verwendet werden kann. Die bereitgestellten Funktionen können in der generierten Implementierung entweder auf Funktionen, die auf der Sensorknoten-Plattform bereitstehen, oder auf eigene Implementierungen abgebildet werden.

4.2 Metamodelle

In diesem Abschnitt wird der Entwurf der Metamodelle für die Beschreibung und Konfiguration des Systemaufbaus der sDDS-Middleware vorgestellt. Diese Metamodelle sind notwendig, da in der Analyse in Abschnitt 3.1.6 die Anforderung 6 aufgestellt wurde, nach dem die Middleware und das System abstrakt modellierbar und damit plattformunabhängig sein sollte, damit aus dieser Beschreibung mittels eines MDSD-Ansatzes eine anwendungsspezifische Middleware-Implementierung generiert werden kann.

Die Modellierung eines sDDS-Systems umfasst mehrere getrennt betrachtbare Bereiche, deren Eigenschaften bereits in der Analyse in Abschnitt 3.4.2 untersucht wurden. Für die Deklaration von Datentypen für die DDS-Topics wird eine Datenmodell benötigt, das in Abschnitt 3.3 unter den Anforderungen und Einschränkungen von drahtlosen Sensornetzen untersucht wurde. Als Ergebnis dieser Analyse wurden in Abschnitt 3.3.4 die Datentypen festgelegt, die für diese Arbeit relevant sind und die im Weiterem durch ein entsprechendes Datentyp-Metamodell vorzusehen sind.

Modellierung und Konfiguration des Aufbaus der Middleware selber, wie sie von den Anwendungen benötigt wird, ist davon abhängig, welche DDS-Funktionalität die Middleware enthält und nach welchen Gesichtspunkten diese konfiguriert werden können. In Abschnitt 3.4.2 der Analyse wurde dabei auch untersucht, wie diese Funktionalität modularisiert werden und welche Abhängigkeiten sich zwischen den Modulen ergeben. Als gewählte Modulasierungsebene wurde in Abschnitt 3.4.1 die der DDS-Objekte und QoS-Richtlinien gewählt, was durch das Metamodell abzubilden ist.

In dem Abschnitt 3.3.4 der Analyse über die Anforderungen an ein Datenmodell wurde gefordert, dass auf Basis der deklarierten Datentypen und der Modellierung bzw. Konfiguration des sDDS-Systems vor der Generierung der Middleware-Implementierung überprüft werden soll, ob Bedingungen eingehalten werden, die sich aus den Kapazitäten der Zielplattform ergeben. Des Weiteren folgt aus der Anforderung 4 nach einer Plattformunabhängigkeit in Abschnitt 3.1.6 für die Modellierung und Anwendungsschnittstelle, dass für die Generierung einer Middleware-Implementierung plattformspezifische Informationen benötigt werden, die ebenfalls über ein Metamodell definiert werden. Dies umfasst unter anderem die Spezifikation des jeweiligen Kommunikationssystems, da die Middleware bzw. ih-

re Protokoll-Implementierung auf der in Abschnitt 3.5.4 beschriebenen universellen Abstraktionsschnittstelle für Kommunikationssysteme aufbauen soll. Für die Generierung des Programmcodes muss daher die jeweilige Implementierung dieser Schnittstelle ausgewählt werden.

4.2.1 Überblick Systemmodellierung

Die im folgenden beschrieben Metamodelle werden für die Modellierung eines sDDS-Systems mit den austauschbaren Daten in einem DDS-Datenraum (Dataspace), dem Aufbau und der Konfiguration der sDDS-Middleware für die Anwendungen und der Beschreibung und Konfiguration der spezifischen Sensorknotenplattform verwendet. Für die einzelnen Bereiche werden einzelne spezialisierte Metamodelle definiert, deren Elemente sich gegenseitig referenzieren und in ein gemeinsames Metamodell überführt werden können. Damit soll es ermöglicht werden, dass mit unterschiedlichen für die jeweiligen Problembereiche angepassten DSLs die anwendungsspezifischen Modelle definiert werden können.

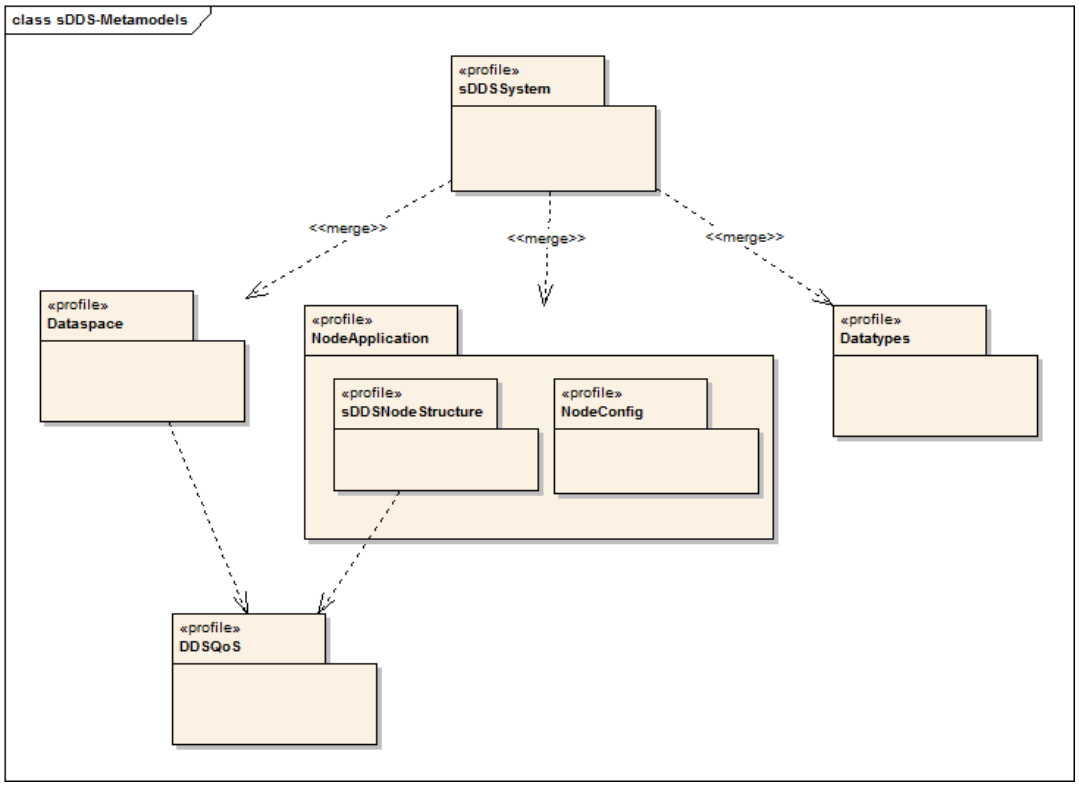

Abbildung 4.3: Metamodelle für die sDDS-Systemmodellierung

In Abbildung 4.3 sind die Metamodelle als Pakete mit ihren Abhängigkeiten untereinander dargestellt. Datentypen werden mit dem Metamodell *Datatypes* modelliert, der Datenraum eines sDDS-System, also die DDS-Topics und ihre Konfiguration, mit dem Metamodell *Dataspace*. Für die Beschreibung des jeweiligen Knotens, auf dem die jeweiligen Anwendungen und spezifische Middleware-Implementierung laufen sollen, wird das Metamodell *NodeApplication* verwendet, das sich in zwei Metamodelle unterteilt: *sDDSNodeStructure* für Aufbau und Konfiguration der sDDS-Middleware und *NodeConfig* für die Beschreibung der Knoten-Plattform.

Die QoS-Richtlinien des DDS-Standards sind im Metamodell *DDSQoS* abgebildet und Elemente aus den Metamodellen *Dataspace* und *sDDSNodeStructure* verwenden dieses, um QoS-Richtlinien und ihre Parametrisierung zu definieren.

4.2.2 *Datatypes*-Metamodell

Zur Deklaration der Datentypen der Daten, die in einem sDDS-System Topics zugeordnet und somit zwischen den Anwendungen austauschbar sind, wird das Metamodell *Datatypes* spezifiziert, das sich an der Definition von OMG IDL [OMG08a] und dem UML-Profil für DDS der OMG [OMG08b] bezüglich der Datentypen orientiert.

Die Auswahl der vorgesehenen Datentypen folgt den Ergebnissen der Analyse in Abschnitt 3.3 und den Designentscheidungen in 3.7 und bildet damit eine Untermenge der OMG IDL-Datentypen. Der Aufbau des Metamodells ist in Abbildung 4.4 dargestellt. Darin sind die primitiven Datentypen (*SimpleType*), einfache Collections, wie Strings, und Sequenzen von primitiven Datentypen mit fester Angabe über die Elementanzahl festgelegt. Des Weiteren werden zusammengesetzter Datentypen vorgesehen, die für die Deklaration der Topic-Datentypen (*TopicType*) benötigt werden. Alle Datentypen können Teil eines solchen Topic-Datentyps sein und sind daher von der Klasse *TopicField* abgeleitet und werden darin über einen Namen referenziert.

In dem Metamodell sind Felder für plattformspezifische Informationen vorgesehen, die im Prozess der Generierung der Middleware für Anpassungen verwendet werden können.

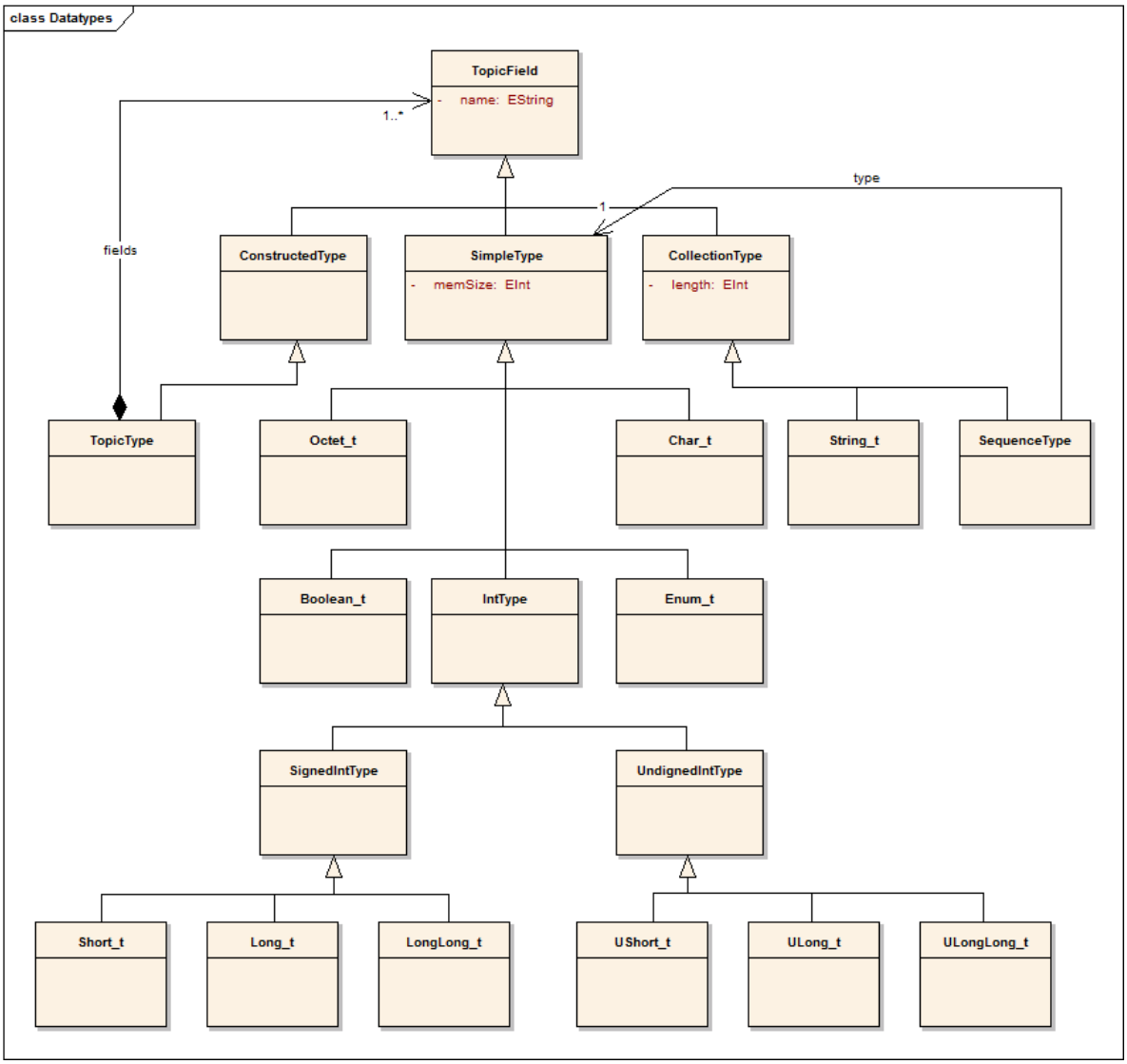

Abbildung 4.4: Metamodell für die Topic-Datentypen in sDDS

4.2.3 *Dataspace*-Metamodell

Dem DDS-Standard nach spannt eine DDS-Domain einen globalen Datenraum auf, dem die Topics zugeordnet sind, welche von den Anwendungen verwendet werden können [OMG07]. Das *Dataspace*-Metamodell, welches in Abbildung 4.5 dargestellt ist, dient dazu, diese Datenräume, die in dem Metamodell durch die Metaklasse *Domain* repräsentiert werden, innerhalb eines sDDS-Middleware-Systems zu definieren.

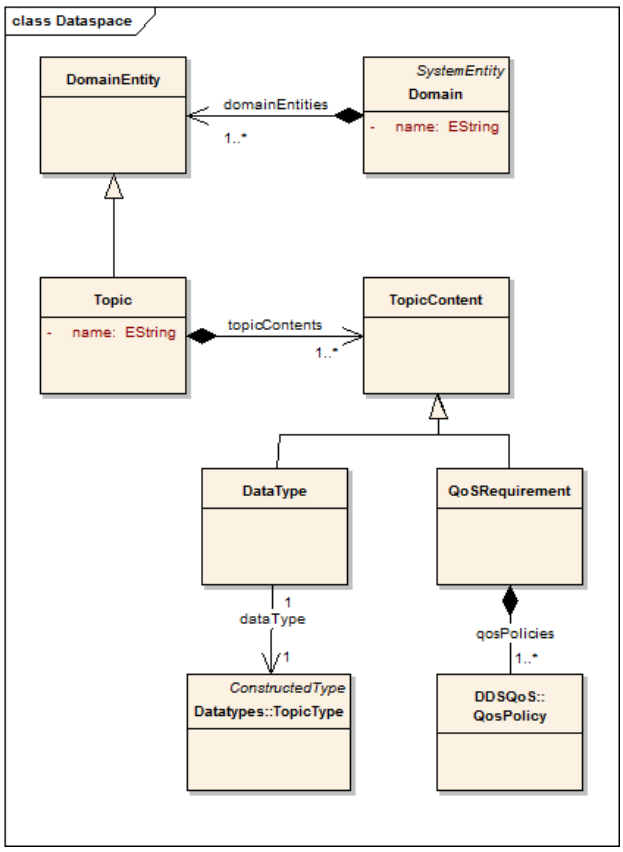

Abbildung 4.5: Metamodell für die Modellierung der globalen Datenraume eines sDDS-Systems

Jede Domain verfügt in einem System über einen eindeutigen Namen und besteht aus einer Menge von *DomainEntitys*, von denen die *Topics* abgeleitet sind. Ihre Definition referenziert einen zuvor im Datentyp-Modell deklarierten Topic-Datentyp und besitzt einen in der Domain eindeutigen Namen und optional eine Festlegung von für dieses Topic zu erfüllenden QoS-Richtlinien.

4.2.4 *NodeApplication*-Metamodell

Das *NodeApplication*-Metamodell besteht aus zwei untergeordneten Metamodellen, die dazu dienen, den Aufbau und die Konfiguration der sDDS-Middleware auf einem Knoten und die spezifische Knoten-Plattform selber zu beschreiben und zu konfigurieren. Ein Knoten des Middleware-Systems wird dabei durch eine Instanz der *NodeApplication*-Metaklasse repräsentiert und mit einem eindeutigen Namen beschrieben. Diese Instanz enthält dann zum einen das spezifische Modell für den Aufbau und Konfiguration der sDDS-Middleware, wie sie von den auf diesem Knoten ausgeführten Anwendungen benötigt wird. Zum anderen ist die Beschreibung der Knoten-Plattform Teil einer Instanz der *NodeApplication*.

4.2.4.1 *sDDSNodeStructure*-Metamodell

Der Aufbau des Metamodells *sDDSNodeStructure* für die Beschreibung und Konfiguration der von sDDS auf einem Middleware-Knoten orientiert sich an der Objektstruktur und Benennung des DDS-Standards. Auf Grund der erweiterten Konfigurierbarkeit, die für die sDDS-Middleware notwendig ist und Unterteilung der Modellierung in verschiedene Metamodelle wird ein eigenes Metamodell für die sDDS-Struktur entworfen.

Die Metaklasse *DomainParticipant*, die in Abbildung 4.6 dargestellt ist, referenziert die Zugehörigkeit zu einer Instanz des *Dataspace*-Metamodells und somit den Datenraum, in dem eine Anwendung operieren kann. Analog zu der Objektstruktur des DDS-Standards kann ein DomainParticipant Publisher und Subscriber haben und diese jeweils DataWriter und DataReader. Die DataReader und DataWriter müssen mit jeweils einem Topic assoziert sein, das mit dem *Dataspace*-Metamodell definiert wurde.

Alle den DDS-Objekten entsprechenden Metamodellklassen können spezifische Metamodellklassen für die genauere Definition und Konfiguration enthalten. An dieser Stelle kann das Metamodell erweitert werden, um die spezifischen Anforderungen der Anwendungen an das DDS-System genau spezifizieren zu können. Diese Erweiterung kann die Teile der DDS-Anwendungsschnittstelle oder Verhalten einzelner DDS-Komponenten umfassen, wie es in Abschnitt 3.4.2 der Analyse der relevanten DDS-Funktionalität aufgezeigt wurde. Für die in der Abbildung 4.6 dargestellte Version des Metamodells ist die Konfiguration eines sDDS-Systems

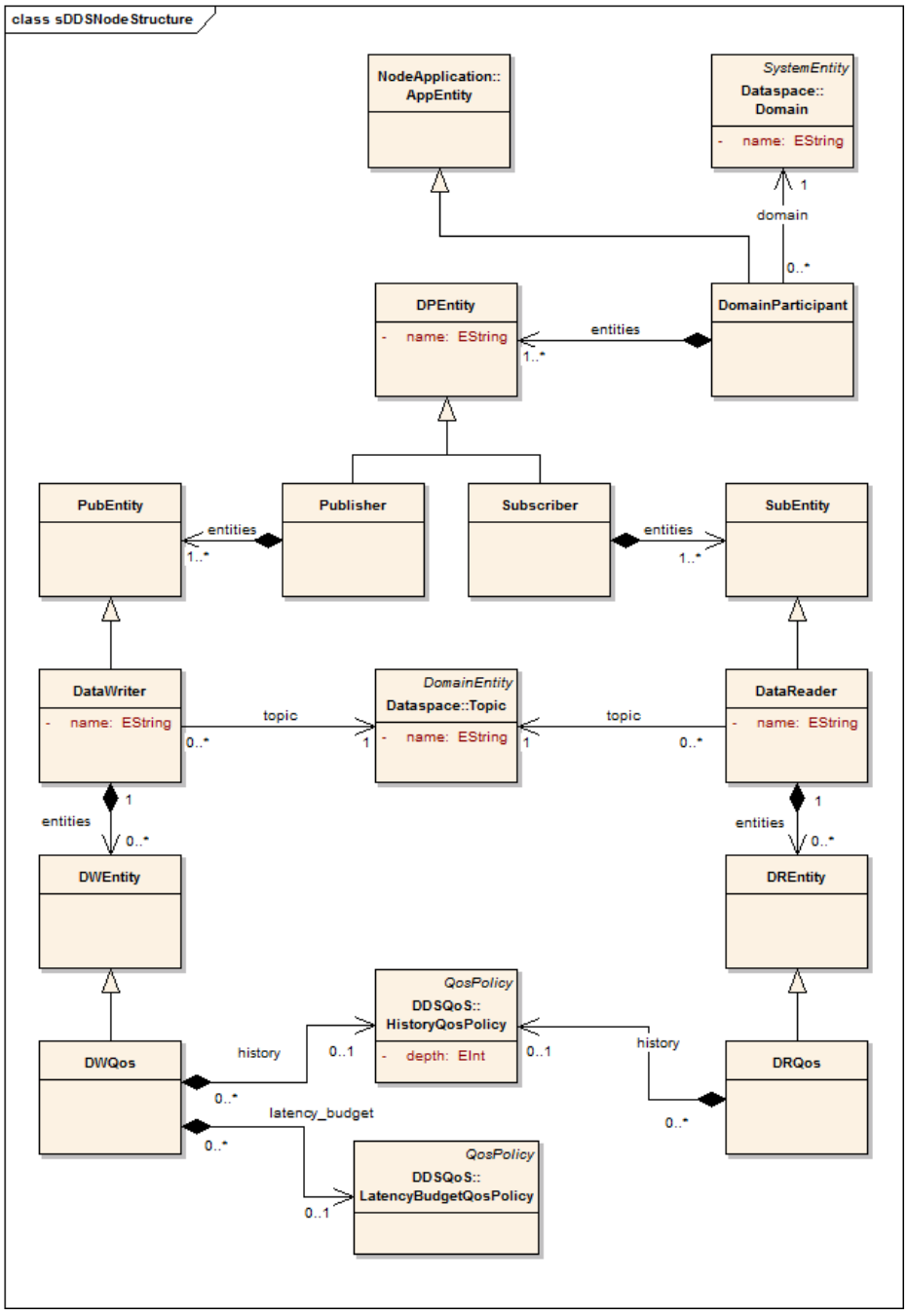

Abbildung 4.6: Metamodell für die Beschreibung der Struktur eines von Anwendungen benutzbaren sDDS-Systems

auf die in Abschnitt 3.4.1.4 gewählte Untermenge an QoS-Richtlinien und Basisfunktionalität des DDS-Standards beschränkt, wobei die QoS-Richtlinien in dem Metamodell *DDSQoS* definiert sind.

4.2.4.2 *NodeConfig*-Metamodell

In Abbildung 4.7 ist das Metamodell *NodeConfig* für die Definition der Plattform des Middleware-Knotens dargestellt, auf der Anwendungen und sDDS-Middleware laufen sollen.

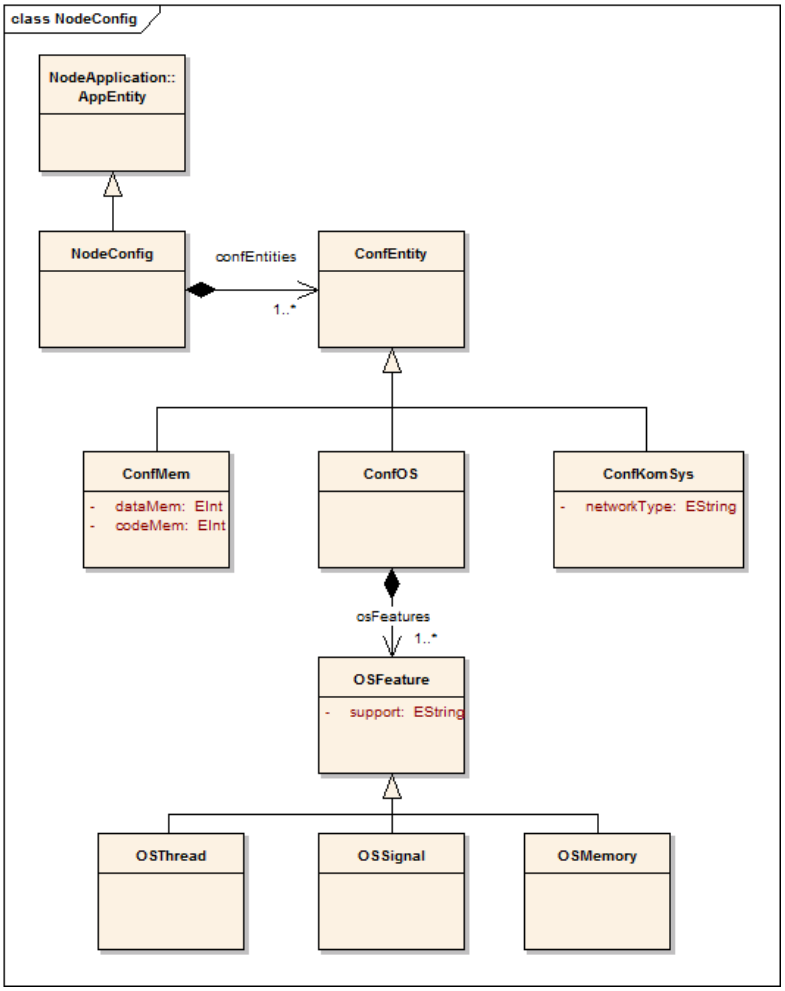

Abbildung 4.7: Metamodell für die Beschreibung der Plattformumgebung, auf der Middleware und Anwendungen laufen sollen

Wie auch bei dem Metamodell *sDDSNodeStructure* für die Modellierung des

sDDS-Systems berücksichtigt *NodeConfig* nicht alle Möglichkeiten für eine Konfiguration und Beschreibung einer Plattform für einen Middleware-Knoten. Zur Veranschaulichung des Prinzips sind beispielhaft Metamodellklassen für die Definition des für die Middleware verfügbaren Speichers, des zu verwendenden Kommunikationssystems und über die Existenz von Betriebssystemfunktionen auf der Plattform gegeben.

4.2.5 *DDSQoS*-Metamodell

Das *DDSQoS*-Metamodell bildet die Untermenge der QoS-Richtlinien des DDS-Standards ab, die in Abschnitt 3.4.1 als relevant für diese Arbeit angesehen wurde und wird von den zuvor beschriebenen Metamodellen *Dataspace* und *sDDSNodeStructure* verwendet, um QoS-Richtlinien für ihre Elemente definieren zu können.

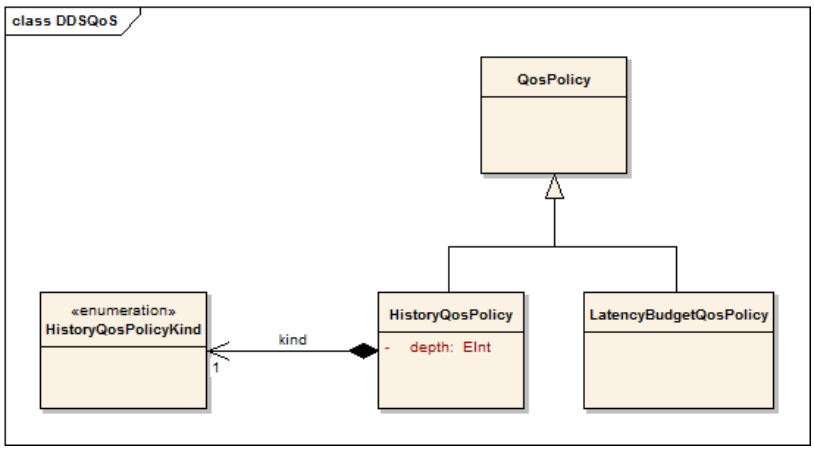

Abbildung 4.8: Metamodell für die Modellierung der QoS-Richtlinien des DDS-Standards

Abbildung 4.8 zeigt das *DDSQoS*-Metamodell, mit den im Rahmen dieser Arbeit relevanten QoS-Richtlinien des DDS-Standards. Das Metamodell orientiert sich in der Struktur, Benennung und Parametrisierung an dem UML-Profil für DDS [OMG08b]. Eine spätere Erweiterung des Metamodells um weitere QoS-Richtlinien des DDS-Standards ist daher einfach möglich, in dem diese analog definiert und in die abhängigen Metamodelle *Dataspace* und *sDDSNodeStructure* entsprechende Referenzen eingefügt werden.

4.2.6 Vereinigung der Metamodelle für die Systemmodellierung

Die von den einzelnen Metamodellen abgeleiteten Modelle für die Definition eines spezifischen sDDS-Systems sind für die Generierung der Middleware zu einem gemeinsamen Modell zu transformieren. In Abbildung 4.9 ist dargestellt, wie die einzelnen bisher beschriebenen Metamodelle in Zusammenhang stehen.

Demnach besteht ein sDDS-System aus den möglichen Topic-Datentypen, den Domains als Datenräume für die Anwendungen und der Beschreibung der Middleware-Knoten für die Anwendungen (hier repräsentiert als „Application") selber. Jedes sDDS-System wird über einen eindeutigen Namen referenziert, der als Namensraum für die Middleware und Anwendungen verwendet werden soll.

4.3 Software-Entwicklungsprozess

In Abschnitt 4.1.1 wurde die Architektur des sDDS-Middleware-Frameworks vorgestellt, das einen MDSD-Prozess für die Generierung einer an die Anwendungsanforderungen angepassten sDDS-Middleware-Implementierung umsetzt. In diesem Abschnitt soll dieser Prozess genauer beschrieben werden.

Der Prozess für die Generierung eine Middleware-Implementierung teilt sich in verschiedene Schritte auf. In Abbildung 4.10 wird die Abfolge der Schritte grafisch dargestellt, die aus einer anwendungsspezifischen Modellierung eine angepasste sDDS-Middleware-Implementierung generieren.

Die Prozess beginnt damit, dass ein Anwendungsentwickler die Anforderungen an die Middleware analysiert. Dies umfasst die folgenden Fragestellungen:

- Welche DDS-Funktionalität wird benötigt
- Welche Daten sollen ausgetauscht werden
- Welche Middleware-Knoten sollen welche Aufgaben wahrnehmen
- Welche Hardware-Plattform wird eingesetzt
- und welche Ressourcen stehen darauf zur Verfügung
- Welche zusätzliche Funktionalität wird benötigt

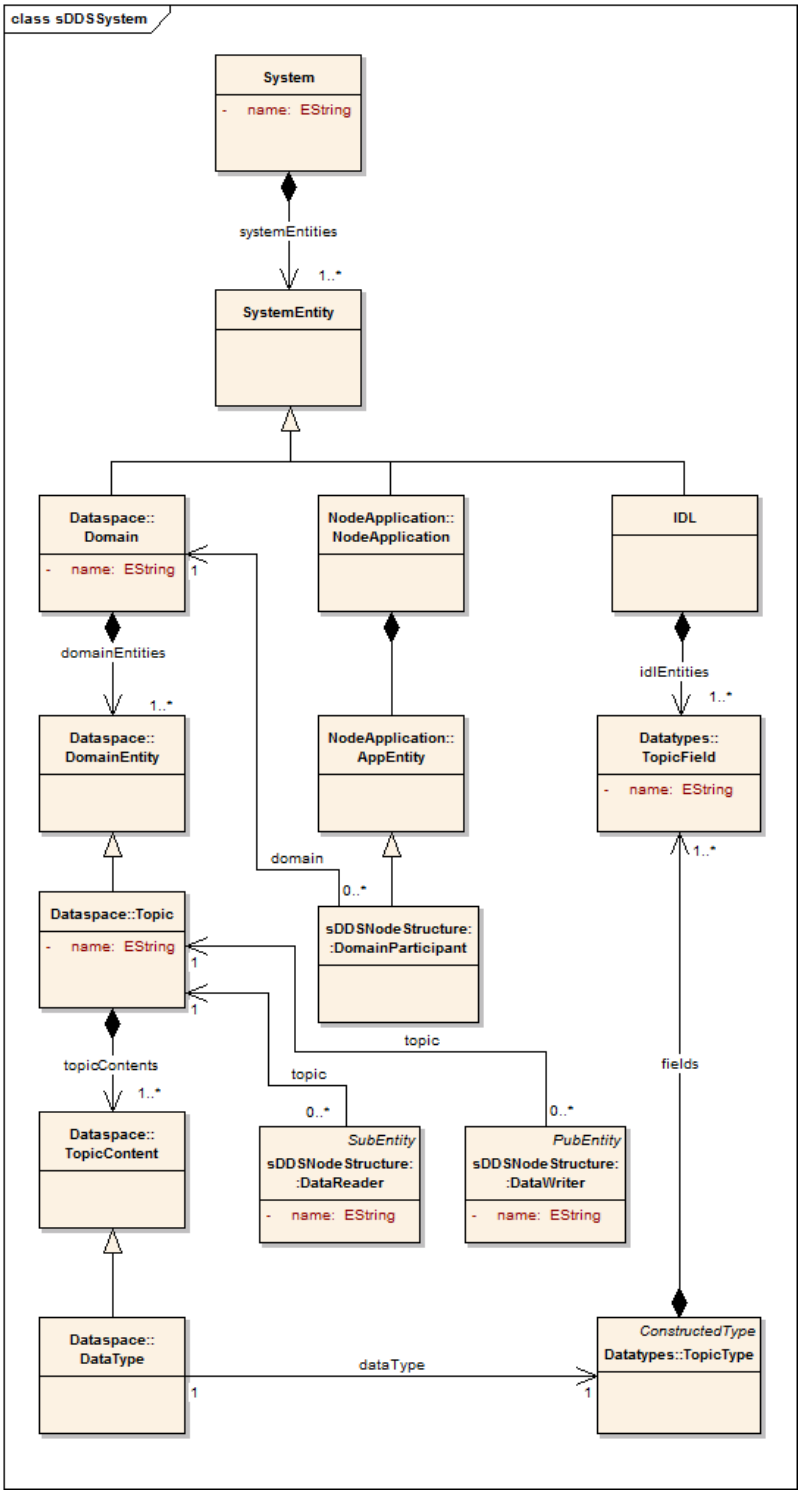

Abbildung 4.9: Metamodell zur Beschreibung des Zusammenhangs der sDDS-Metamodelle

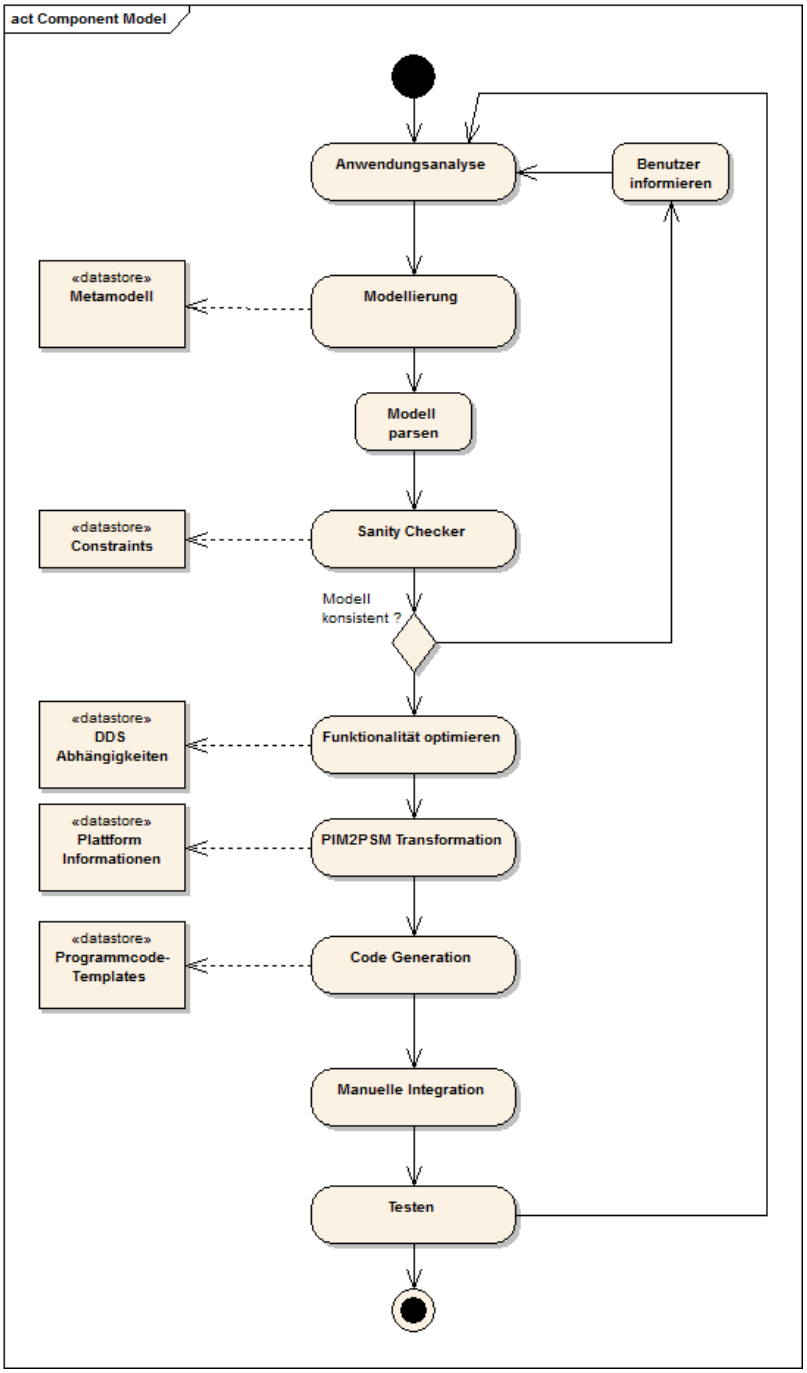

Abbildung 4.10: Darstellung des MDSD-Prozesses für die Generierung eine angepassten sDDS-Middleware-Implementierung aus der anwendungsspezifischen Modellierung

Um diese Anforderungen formal festlegen zu können, werden die in Abschnitt 4.2 definierten Metamodelle und davon abgeleiteten DSLs verwendet. Es kann auch mehrere DSLs für denselben Problemraum geben, so dass ein Entwickler die Definitionssprache auswählen kann, die ihm am besten geeignet erscheint. Für die Modellierung kann das sDDS-Middleware-Framework auch eigene Editoren oder andere Eingabemöglichkeiten bereitstellen.

Nach der Modellierung der Anforderungen an das DDS-System und der Auswahl der benötigten DDS-Funktionalität werden die Modelle vom sDDS-Framework eingelesen. Die eingelesenen Modelle werden im nächsten Schritt auf ihre Konsistenz und die Einhaltung von Bedingungen überprüft, die sich aus der gewählten Zielplattform und der Konfiguration für die zu generierende Middleware-Implementierung ergeben können. In Abschnitt 3.4.2 wurden verschiedene QoS-Richtlinien des DDS-Standards untersucht und festgestellt, dass sie für diese Art der Überprüfung geeignet sind. Dies trifft insbesondere auf Richtlinien wie die *Ressource Limits* zu, die eine statische Analyse anhand der verfügbaren und benötigten Speichermenge erlauben.

Sollte bei der Überprüfung des Modells festgestellt werden, dass es Bedingungen nicht einhält, wird dem Entwickler eine Rückmeldung gegeben. Auf Basis der Rückmeldung kann die Analyse des Anwendungssystems verfeinert und das Modell angepasst werden. Dieser Vorgang kann mehrfach wiederholt werden.

Erfüllt das Modell die gegebenen Bedingungen und ist es konsistent, dann analysiert das sDDS-Framework die konfigurierte DDS-Funktionalität. Wie in Abschnitt 3.4.1 festgestellt wurde, sind die modellier- und konfigurierbaren Komponenten des DDS-Standards oftmals abhängig von anderen Komponenten und gewählten Konfigurationen. In dieser Arbeit wurden diese Abhängigkeiten nur oberflächlich untersucht, da sich in Abschnitt 3.4.1.3 zeigte, dass der Aufwand für eine vollständige, feingranulare Analyse zu hoch ist, um im Rahmen dieser Arbeit durchgeführt werden zu können. Wenn Informationen über die Abhängigkeiten vorhanden sind, kann das Framework feststellen, welche weiteren Komponenten benötigt werden. Diese werden dem Modell über eine Transformation hinzugefügt. Dabei kann, unter Berücksichtigung der Abhängigkeiten bezüglich der DDS-Komponenten, das Modell auf die Bedürfnisse von Sensornetzen optimiert werden. Die zu berücksichtigenden Anforderungen von drahtlosen Sensornetzen wurden in Abschnitt 3.1 untersucht.

Der Verarbeitungsschritt die DDS-Abhängigkeiten aufzulösen und das Modell zu

optimieren, kann alleinstehend oder mit dem nächsten Schritt zusammengefasst sein. Dieser fügt über Modelltransformationen Aspekte der Zielplattform in das Modell ein und wandelt es in ein plattformabhängiges Modell um. Dies kann beispielsweise darin bestehen, dass die abstrakten Datentypen gegen plattformspezifische ausgetauscht werden und festgelegt wird, ob bestimmte Funktionalität als Programmcode generiert werden muss oder auf auf der Zielplattform vorhandene abgebildet werden kann.

Aus dem plattformspezifischen Modell der sDDS-Middleware, das alle notwendige Funktionalität enthält, bzw. referenziert, soll im nächsten Arbeitsschritt der Programmcode für die sDDS-Middleware-Implementierung generiert werden. Hierfür greift das Framework auf Templates zurück, die für die jeweilige Zielplattform und Funktionalität bereitstehen müssen. Der generierte Programmcode stellt Anwendungen die konfigurierten Schnittstellen von DDS zur Verfügung. Die Anwendung kann direkt auf den modellierten DataReader- und DataWriter-Objekten arbeiten und muss die Middleware nicht zur Laufzeit selber konfigurieren.

Nachdem der Programmcode der sDDS-Middleware-Implementierung generiert wurde, ist es Aufgabe des Anwendungsentwicklers, den Programmcode der Anwendung mit dem der Middleware zusammenzuführen und für die Zielplattform zu übersetzen. Die fertige Anwendung kann danach getestet und validiert werden. Ergebnisse dieses Prozesses können wieder in die Definition der Anforderungen an die Middleware übertragen und damit die bisherige Modellierung bzw. Konfiguration angepasst werden. Danach kann das sDDS-Middleware-Framework eine neue Version der Middleware-Implementierung generieren. Dieser Zyklus kann beliebig wiederholt werden.

4.4 Sensor-Network Publish-Subscribe Protokoll

4.4.1 Übersicht SNPS

Basierend auf der Analyse in Kapitel 3.5 dieser Arbeit wurde das „Sensor-Network Publish-Subscribe”-Protokoll für die sDDS-Middleware entworfen. Der Name wurde analog zu dem des RTPS-Protokoll gewählt, und es wurde dabei die Ausrichtung auf Sensornetzwerke betont.

Entsprechend den Ergebnissen der Analyse stehen für den Entwurf des SNPS-Protokolls die Aspekte des effektiven Umgangs mit Ressourcen, die flexi-

ble Modularisierung analog zu der Funktionalität der sDDS-Middleware und die spätere Erweiterbarkeit im Vordergrund. Um dem gerecht zu werden, verwendet SNPS einen fein granularen, modularen Ansatz mit dem eine darauf aufsetzende Middleware einen komplexeren Funktionsumfang ausdrücken und umsetzen kann. Hierzu sind als Anleihe an RTPS die Informationen, die eine DDS-basierte Middleware zwischen ihren Netzwerkknoten austauschen muss, in atomare Informationseinheiten zerteilt, den Submessages. Diese sind die Bausteine, aus denen komplexere Informationen zusammengesetzt und in einer Nachricht übertragen werden. Die für zu übertragende Information verwendeten Submessages bilden dabei einen *Kontext*, in dem sie und ihre kontextabhängige Bedeutung zu interpretieren sind. Es ist dabei möglich bzw. gewünscht, dass mehrere Informationen in einer Nachricht übertragen und diese in bestehende Kontexte eingebunden werden, damit Redundanz in den Informationen reduziert werden kann. Dabei werden die Kontexte sequenziell aus den Submessages aufgebaut.

Eine SNPS-Nachricht besteht damit ausschließlich aus einer solchen Folge von Submessages, die als Folge von Bytes kodiert von beliebigen unterlagerten Kommunikationssystemen verarbeitet und somit auch über die in Abschnitt 3.5.4 beschriebenen universellen Abstraktionsschnittstelle übertragen werden kann.

Der Ansatz, die Submessages als Bausteine für die Kodierung einer Information zu verwenden und ihre Interpretation von dem jeweiligen Kontext abhängig zu machen, ermöglicht es, die meisten Anforderungen der Analyse des Protokolls in Abschnitt 3.5.1 zu erfüllen. Mit der Übertragung von mehreren Informationen in einer Nachricht und der Verwendung der Kontexte zur Vermeidung von Redundanz ist es möglich, die durchschnittliche Nachrichtenlänge pro übertragener Information zu reduzieren und, wie in Abschnitt 3.5.1 untersucht, die Bandbreite eines drahtlosen Sensorknotens effektiver auszunutzen. Der sequenzielle Aufbau der Nachrichten ist auch geeignet, die Verarbeitung (Auslesen und Interpretieren) durch eine Middleware-Implementierung für drahtlose Sensorknoten zu vereinfachen.

Im selben Abschnitt 3.5.1 der Analyse wurde gefordert, dass, um der Heterogenität des Funktionsumfangs der sDDS-Middleware in einem System begegnen zu können, auch das Protokoll modular ausgelegt sein soll, wobei es aber dennoch alle Knoten verarbeiten können. Hierzu sind die Submessages soweit selbstbeschreibend, dass jeder Middleware-Knoten die Informationen, die für ihn relevant und interpretierbar sind, aus einer beliebigen Nachricht extrahieren kann.

Bei der Selbstbeschreibung eines Protokolls ist unter anderem abzuwägen zwischen dem impliziten Wissen um die Länge bestimmter Daten und der expliziten Kodierung von Längeninformationen in der Nachricht. Die implizite Längeninformation reduziert die zu übertragenden Daten, führt aber zu dem Problem, dass alle Knoten dieses Wissen benötigen, um eine Nachricht verarbeiten zu können, was die Erweiterbarkeit einschränkt. SNPS verwendet daher eine Mischform aus beiden Ansätzen. Es gibt verschiedene Typen von Submessages, die in einer dreistufigen Hierarchie angeordnet sind: „Basic", „Extended" und „Supplement"-Submessages. Basis-Submessages, die im normalen Betrieb sehr häufig vorkommen, sind fest definiert, besonders klein gehalten und haben eine feststehende Länge. Eine eingeschränkt mögliche Erweiterung dieser Typen führt aber zu einer zunehmenden Inkompatibilität mit anderen Versionen des Protokolls. Die zweite Stufe der Hierarchie bilden die Extendend-Submessages, sie kommen seltener vor und benötigen mehr Platz, haben aber ebenfalls eine fest definierte Länge. Die Supplement-Submessages bilden die letzte Stufe der Hierarchie und kodieren für den zu transportierenden Inhalt eine explizite Längeninformation.

Das Wissen um die feste Länge von Basic- und Extendend-Submessages ist ein integraler Bestandteil des Protokolls. Eine sDDS-Middleware-Implementierung, deren Funktionalität nur auf einer Untermenge von Submessages aufbaut, kann daher nicht benötigte Teile einer empfangenen Nachricht überspringen, ohne dass sie diese verarbeiten muss.

Für die Abbildung der DDS-Funktionalität sieht SNPS zum Einen dedizierte Submessages vor, sofern sich diese mit atomaren Informationseinheiten implementieren lässt. Neben der Basisfunktionalität von DDS (siehe auch Abschnitt 3.4.2.1 der Analyse der DDS-Funktionalität) betrifft dies auch grundlegende QoS-Funktionalität, deren effiziente Implementierung eine direkte Unterstützung auf Ebene des Protokolls benötigt. Um den Anforderungen an minimalen Ressourcenverbrauch gerecht zu werden, sieht das Protokoll an einigen Stellen mehrere Typen von Submessages für dieselbe DDS-Funktionalität vor, die jeweils unterschiedlich viel Daten kodieren können. Sofern bei der Verwendung der kleineren Datenmenge von den Vorgaben des DDS-Standards abgewichen wird und Anwendungen an diesen Stellen das exakte Verhalten fordern, muss die Middleware-Implementierung entweder andere Submessages verwenden oder das definierte Verhalten der DDS-Schnittstelle emulieren.

Komplexere Funktionalität von DDS ist auch auf Ebene der Middleware zu implementieren. Dafür sind die zu transportierenden Informationen mittels der de-

finierten Submessages zu kodieren. So soll die „Discovery" von Topics und der Aufbau von Subscriptions zwischen Datenquelle und Datensenke mittels der im DDS-Standard vorgesehenen Built-In-Topics geschehen, die auf der Basisfunktionalität von DDS aufbauen.

Auf Basis der Analyse für das Kommunikationsprotokoll in Abschnitt 3.5.4 wurde in den Designentscheidungen festgelegt, dass SNPS auf eine universelle Abstraktionsschnittstelle für das unterlagerte Kommunikationssystem aufbauen soll, welches unter anderem das Routing der Nachrichten durchführt. Wie bereits in der Analyse in Abschnitt 3.1 beschrieben wurde, ist es für Sensornetzanwendungen üblich, dass die Grenzen zwischen Anwendung, Middleware und Protokoll aufgebrochen werden, um möglichst effektiv mit geringem Ressourcenverbrauch zu arbeiten. Wie in den Grundlagen in Abschnitt 2.1.1 beschrieben, nimmt Routing in Sensornetzen eine wichtige Stellung für ein energiebewusstes Verhalten ein, welches als Anforderung 2 in Abschnitt 3.1.6 der Analyse für eine DDS-basierte Middleware aufgestellt wurde. Da das eigenständige Thema Routing in dieser Arbeit nicht weiter berücksichtigt werden konnte, enthält auch SNPS keine definierte Funktionalität hierzu. Dennoch wurde eine spätere Erweiterung berücksichtigt, da nicht alle Basic-Submessages definiert wurden und generische Status- und Adressen-Submessages für entsprechende Lösungen verwendet werden können.

4.4.2 SNPS-Nachrichten

4.4.2.1 Allgemein

Eine SNPS-Nachricht besteht aus einem Header und einer Menge weiterer Submessages. Es gibt wie in 4.4.1 beschrieben drei Arten von Submessages, aus denen eine Nachricht zusammengestellt werden kann: „Basic", „Extendend" und „Supplement"-Submessages. Die Kodierung der Submessages ist auf 8 Bit ausgerichtet, so dass in einer Implementierung des Protokolls eine einfache Verarbeitung in einem Byte-Array möglich ist. Abbildung 4.11 zeigt den Aufbau der Submessage-Typen. Die Basic-Submessages sind der Basistyp, von dem sich die anderen ableiten. Bisher sind nicht alle Submessages vollständig spezifiziert, da der umfassende Entwurf für die relevante DDS-Funktionalität noch nicht feststeht und spätere Erweiterungen möglich sein sollen.

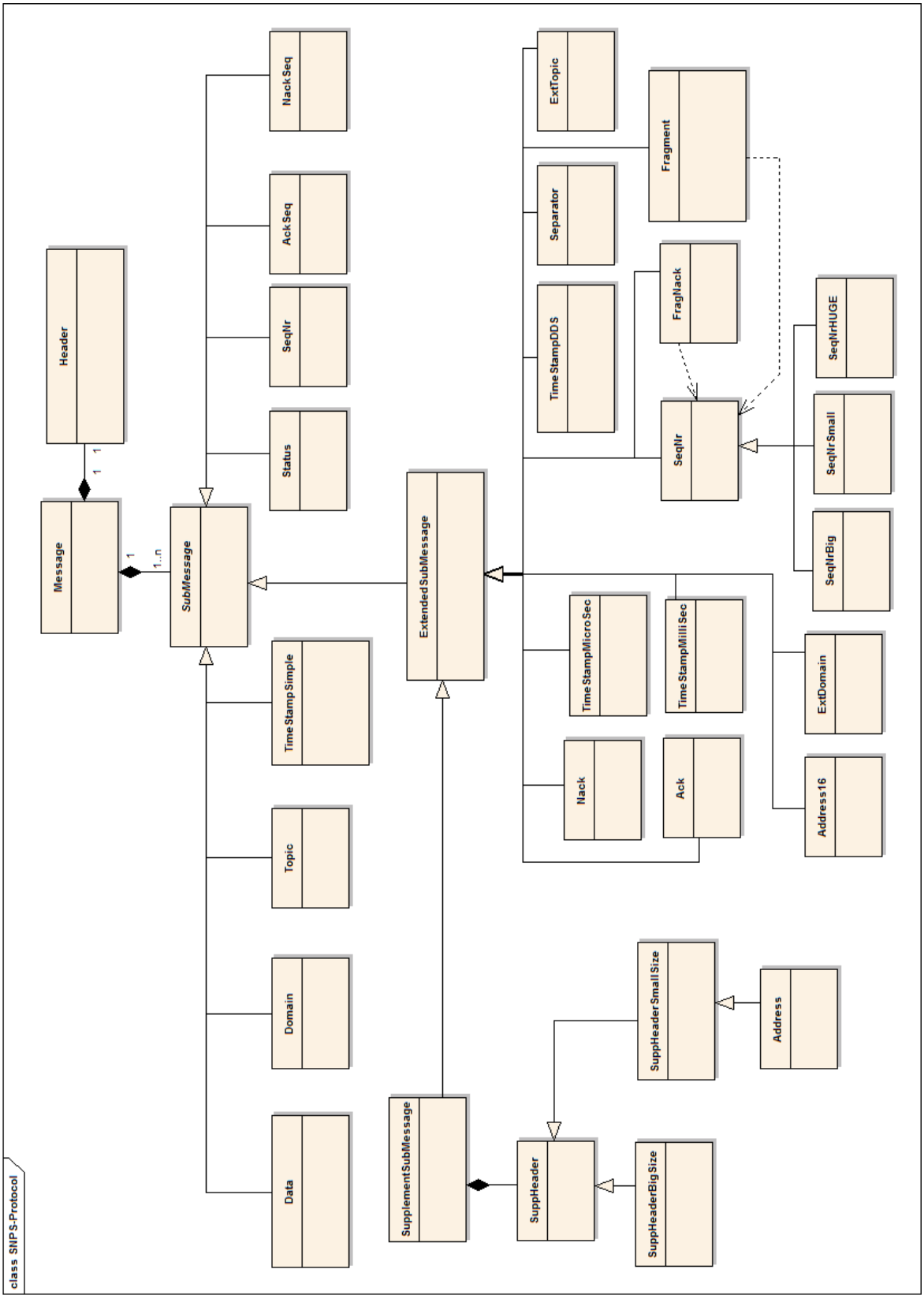

Abbildung 4.11: Nachrichtentypen des SNPS-Protokolls

Jede SNPS-Nachricht wird eingeleitet von einem Header, welcher die Version des Protokolls und die Anzahl der Submessages in dieser Nachricht enthält, damit empfangende Knoten erfahren, ob sie die Nachricht verarbeiten können. Die Information über die Anzahl der Submessages kann von Protokoll-Implementierungen beliebig verwendet werden. Jede Submessage stellt eine atomare Informationseinheit dar und wird von einem eigenen Header eingeleitet, der deren Typ definiert. Um sicherzustellen, dass das implizite Wissen um die Längen der Submessages feststeht, werden diese hier sehr detailliert beschrieben.

Im Weiteren sollen die einzelnen Typen der Submessages genauer beschrieben werden und im Anschluss, wie sich eine SNPS-Nachricht aus Submessages aufbaut.

4.4.2.2 Basic-Submessages

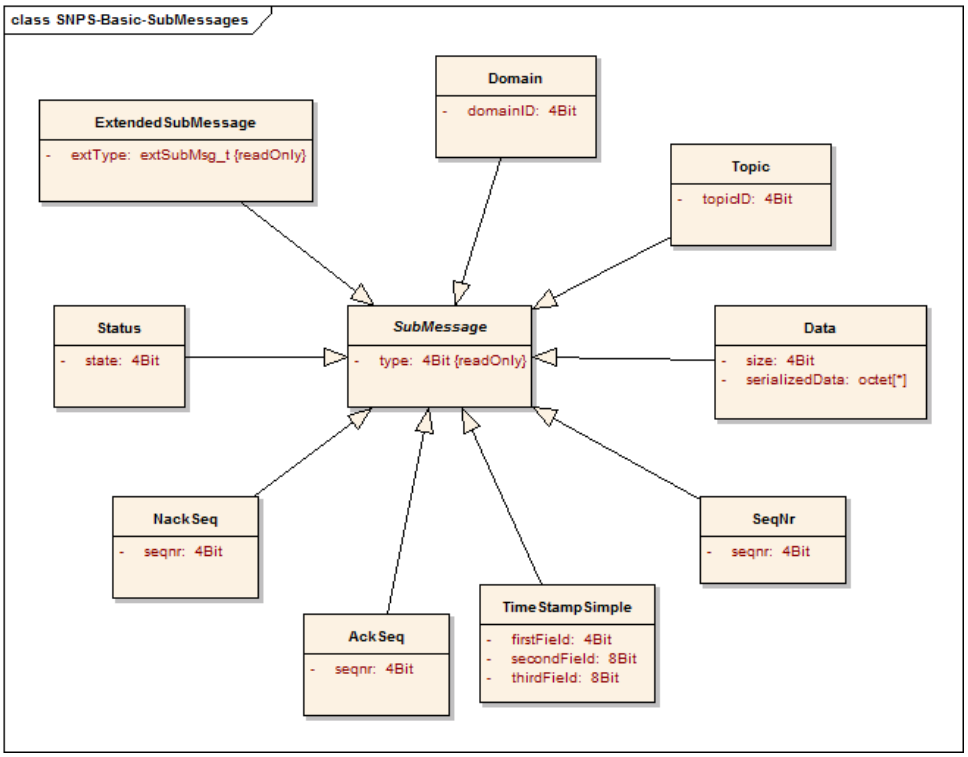

Abbildung 4.12: Darstellung der Basic-Submessages des SNPS-Protokolls

Wie die Darstellung 4.12 zeigt, werden für die Kodierung des Typs der Basic-Submessages 4 Bit verwendet und die verbleibenden 4 Bit stehen für den jeweiligen

Inhalt zur Verfügung. Die Basic-Submessages werden für häufig vorkommende Informationen oder Erbringung von Funktionalität in einer DDS-Middleware verwendet, bzw. auch für solche, die mit 4 Bit für den Inhalt auskommen.

Zur Zeit sind 9 Typen der Basic-Submessages definiert, deren Inhalt, Semantik und Größe festgelegt ist. Für eine Erweiterung stehen damit 6 weitere Typen zur Verfügung, wobei hier die Einschränkung auftritt, dass dann deren Größe älteren Implementierungen des Protokolls nicht bekannt ist und sie somit nicht kompatibel wären.

Die in einem sDDS-System wohl am häufigsten benötigte Funktionalität ist die Übertragung von Daten zwischen DataWriter und DataReader. Daher sind die notwendigen Submessages dafür vollständig vom Basic-Typ. Insgesamt werden im Rahmen dieser Arbeit die folgenden Basic-Submessages festgelegt:

Domain: Kodiert die Domain, der der Inhalt der folgenden Submessages zugeordnet ist, mit 4 Bit. In einem sDDS-System können damit die ersten 15 Domains adressiert werden, für die Referenzierung von mehr Domains muss eine Extended-Submessage verwendet werden.

Topic: Kodiert das Topic, dem der Inhalt der folgenden Submessages zugeordnet ist, mit 4 Bit. Für mehr als 15 Topics muss die entsprechende Extended-Submessage verwendet werden.

Data: Ist der Header für einen Daten-Payload, welcher von dem DataWriter in eine Folge von Bytes kodiert wurde. Die maximale Datengröße sind hierbei 15 Byte, für einen größeren Payload muss eine Extended-Submessage verwendet werden.

TimeStampSimple: Diese Submessage stellt 20 Bit für die Kodierung eines Zeitstempels zur Verfügung. Für genauere, längere Zeitstempel muss eine entsprechende Extended-Submessage verwendet werden.

Status: Für die Kodierung eines Status stehen 4 Bit als generisches Bit-Feld zur Verfügung, dessen Verwendung der Middleware obliegt.

SeqNr: Eine einfache Sequenznummer mit 4 Bit sollte in einem Sensornetz für viele Anwendungen reichen. Größere stehen als Extended-Submessages zur Verfügung.

AckSeq: Submessage zur Bestätigung (*Ack* eng.: Acknowledgement) des Empfangs einer Nachricht oder zur Anforderung einer Bestätigung durch den Empfänger. Ist eine 4 Bit Sequenznummer nicht ausreichend, muss eine Extended-Submessage verwendet werden.

NackSeq: Submessage die für die Mitteilung über das Ausbleiben einer Nachricht (*Nack*) verwendet werden kann. Als Payload für eine Sequenznummer stehen 4 Bit zur Verfügung.

4.4.2.3 Extended-Submessages

Die Extended-Submessages sind eine Spezialisierung der Basic-Submessages und verwenden deren 4 Bit Datenfeld für die Kodierung des Typs der Extended-Submessage. Die Bedeutung der 16 möglichen Typen ist bereits festgelegt und in Abbildung 4.13 dargestellt, so dass eine Erweiterung hier nicht möglich ist. Allerdings sind einige Typen als Platzhalter zu sehen, deren genauer Inhalt und Semantik noch nicht festgelegt ist. Bis zu deren endgültigen Festlegung ist die Größe der Submessages nicht bekannt.

Ein Teil der Extended-Submessages hat dieselbe Semantik wie einige Basic-Submessages, stellt aber mehr Platz für die Kodierung des Inhalts zur Verfügung. So werden weitere Arten von Sequenznummern und Zeitstempeln vorgesehen sowie Typen, um mehr Topics und Domains zu adressieren, als mit 4 Bit möglich ist. Die genaue Größe ist dabei aber noch nicht festgelegt, da dafür eine genauere Bedarfsanalyse notwendig ist. Eine andere Gruppe von Extended-Submessages besteht nur aus dem 8 Bit großen Header und stellt durch ihre Verwendung in einer Nachricht eine atomare Informationseinheit dar, die den Kontext der weiteren Submessages beeinflusst. Im Folgenden werden die Extended-Submessages genauer beschrieben, die nicht dieselbe Semantik wie die bisher beschriebenen Basic-Submessages haben:

ACK: Ein Signal, welches auf zwei Arten verwendet werden kann. Zum Einen wird es als Anforderung für eine Bestätigung (Ack) über den Erhalt der im aktuellen Kontext der Nachricht zugehörigen Submessages verwendet. Die zweite Verwendung ist die Einleitung einer solchen Bestätigung. In beiden Fällen muss in deren Kontext entweder eine Sequenznummer oder einen Zeitstempel enthalten, damit dieser referenzierbar ist.

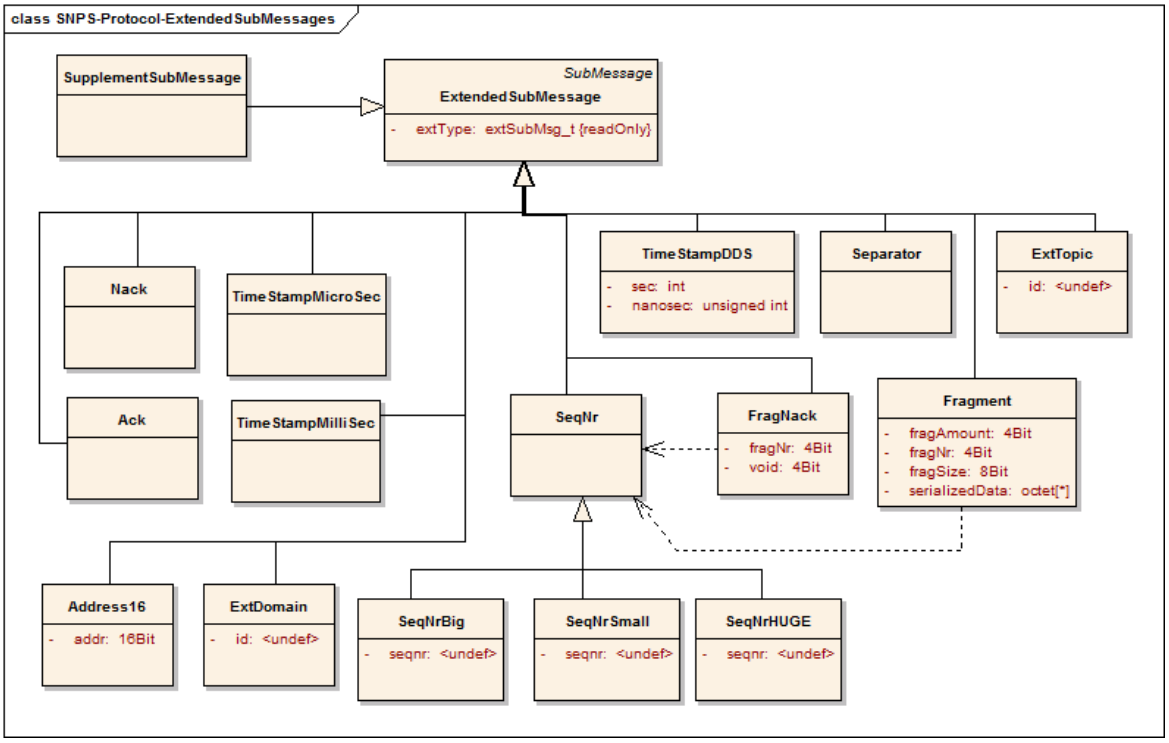

Abbildung 4.13: Darstellung der Extended-Submessages des SNPS-Protokolls

Nack: Ein Signal, welches ein Nack einleitet, in dessen Kontext eine Sequenznummer oder ein Zeitstempel stehen muss.

Fragment und FragNack: Für die Übertragung einer Payload mit mehr als 15 Byte wird dieser Nachrichtentyp verwendet. Daten können in 16 Fragmente aufgeteilt werden, wobei jedes Fragment bis zu 256 Byte Daten enthalten kann. Es ist auch möglich, dass mehrere Fragmente in einem einzigen Nachrichten-Frame enthalten sind, wenn er die entsprechende Größe hat. Die maximale Größe für einen Daten-Payload ist bei SNPS daher 4 KiB.

Separator: Der Separator ist ein Signal, das den Kontext der Nachricht wieder auf den Ausgangszustand zurücksetzt.

Address16: Diese Submessage enthält die Adresse eines Knotens im Netzwerk und ist 16 Bit groß.

4.4.2.4 Supplement-Submessages

Die Supplement-Submessage (eng.: Ergänzung) leitet sich von der Extendend-Submessage ab und soll die flexible Erweiterung des SNPS-Protokolls in der Zukunft ermöglichen. Die bisher definierten Typen von Submessages haben eine implizite Längeninformation über ihren Typ. Die Supplement-Submessages enthalten diese Information explizit in ihrem Header, zusammen mit der Kodierung ihres Typs. Über diese Art der Submessages kann das SNPS-Prot3.386 806.2izitesid-

4.4.3 Nachrichtenaufbau

Die Submessages von SNPS stehen in Relation zu den anderen Submessages innerhalb einer Nachricht und bauen so den Kontext auf, innerhalb dem ihr Inhalt zu interpretieren ist. Die Kontexte bilden eine Sequenz, so dass eine Middleware-Implementierung auf drahtlosen Sensorknoten diese mit wenig Ressourcenaufwand verarbeiten kann, und beziehen sich auf die Struktur der Daten, wie sie sich aus dem DDS-Standard ergibt. Die Kontexte können verschachtelt sein und bestimmte Submessages sind dazu bestimmt, diese zu verändern.

In einem DDS-System werden primär Daten zwischen den Knoten ausgetauscht, denen weitere Informationen zugeordnet sind. Als zusätzliche Informationen sind minimal die Domain und das Topic notwendig, damit Daten einem DataReader bzw. einem DataWriter zugeordnet werden können. Daher gibt es drei primäre Kontexte, die ineinander verschachtelt werden können:

- Domain-Kontext
- Topic-Kontext
- Daten-Kontext

Eine Data-Submessage leitet einen Daten-Kontext ein, und bis dieser Kontext, beispielsweise durch eine weitere Domain-, Topic- oder Data-Submessage, verändert wird, sind die folgenden Submessages mit dem Payload der Data-Submessage assoziiert. Die Middleware-Implementierung eines drahtlosen Sensorknotens muss nur die Submessages interpretieren können, die für ihre enthaltene Funktionalität benötigt wird, weitere können ignoriert werden. Daher skaliert die

Abbildung 4.15: Darstellung des Aufbau zweier SNPS-Nachrichten. Nachricht *a* enthält die minimal notwendigen Submessages, um Daten DataReadern zuordnen zu können. Nachricht *b* zeigt verschachtelte Kontexte für zwei verschiedene Topics und drei Daten-Payloads.

Die Nachricht *a* in Abbildung 4.15 zeigt die minimal notwendige Nachricht, um Daten von einem DataWriter zu einem DataReader zu schicken. Dazu ist die Angabe einer Domain und eines Topics notwendig, um die Daten einem DataReader zuordnen zu können. Die Adressierung der Empfänger ist Aufgabe der unterlagerten Netzwerkschichten. Die Nachricht *a* wird eingeleitet durch den SNPS-Header mit der Version und der Anzahl der folgenden Submessages (hier wäre es entsprechend ein Wert von 3). Danach folgen die Domain- und Topic-Submessage, die die DDS-Domain „A" und das darin enthaltene DDS-Topic „a" jeweils referenzieren. Der Daten-Payload wird durch die Data-Submessage eingeleitet, die die Länge enthält.

Die Nachricht *b* zeigt, wie mehrere Kontexte verschachtelt werden können, der SNPS-Header wurde bei der Darstellung weggelassen. Der Kontext der Domain „A" gilt für die gesamten dargestellten Submessages, demnach sind sie für potenzielle DataReader in dieser Domain bestimmt. Der Kontext, den die Submessage für das Topic „a" aufspannt, gilt bis zu der des Topics „b". Dementsprechend sind die Daten „1" und „2" für DataReader des Topics „a" und das <Datum> „3" für die des Topics „b". Innerhalb eines Daten-Kontextes könnten weitere Submessages enthalten sein, die dem Datensatz beispielsweise einen Zeitstempel zuordnen.

4.5 sDDS

In diesem Abschnitt wird das Design für die sDDS-Middleware beschrieben. Der Aufbau ist eng an den der Grobarchitektur angelehnt und auf den Einsatz auf Plattformen mit wenig verfügbarem Speicher ausgelegt. Dementsprechend finden sich bereits ausgeprägte Anpassungen an diesen Umstand im Klassenmodell.

Da die sDDS-Middleware-Implementierung generiert werden soll und dabei verändert und an die Zielplattform angepasst wird, sind einige Design-Details veränderbar. Dies trifft insbesondere auf die Frage zu, wie Datenstrukturen referenziert oder gespeichert werden. Auf dieser Ebene des Design wird nicht explizit vorgegeben, wie dies zu geschehen ist. Stattdessen werden die erlaubten Kardinalitäten spezifiziert, und es ist Aufgabe des Generierungsprozesses der Middleware-Implementierung, eine effektive Lösung auszuwählen und einzusetzen.

Der hier vorgestellte Entwurf deckt, wie in den Designentscheidungen in Abschnitt 3.7 festgelegt, eine Untermenge der in Abschnitt 3.4.1 als relevant für Sensornet-

ze aufgeführten Funktionalität von DDS ab. Es wird primär die Basisfunktionalität von DDS vorgesehen, sowie einige exemplarische QoS-Richtlinien, um deren generelle, konfigurierbare Einbindung zu zeigen.

Im Weiteren wird ein Überblick über die allgemeine Klassenstruktur gegeben und danach die Komponenten im Detail betrachtet. Nachdem die Struktur feststeht, werden die wichtigsten Abläufe beschrieben und welche Komponenten dabei involviert sind.

4.5.1 Überblick sDDS-Struktur

In Abbildung 4.16 sind die wichtigsten Klassen von sDDS und ihre Beziehungen zueinander dargestellt sowie die Klassen des DDS-Standards, die von ihnen implementiert werden, oder deren Funktionalität von der sDDS-Middleware emuliert wird, wie in Abschnitt 4.1.2 über die Grobarchitektur von sDDS beschrieben. Wie bereits in Abbildung 4.2 auf Seite 109 der Grobarchitektur konzeptionell dargestellt wurde, besteht sDDS aus einigen primären Komponenten, die für ihre Funktionserbringung auf andere Komponenten zugreifen.

Das Topic hat im DDS-Standard die Rolle DataReader und DataWriter logisch zu verbinden und ist einem spezifischen Datentyp zugeordnet. Dementsprechend ist in diesem Entwurf für die sDDS-Middleware die Klasse `Topic` der zentrale Zugriffspunkt auf die spezifischen Daten, ihre Kodierung und die Informationen, welche Knoten im Netzwerk diese benötigen, bzw. bereitstellen. Für die Knoten, auf denen DataWriter bzw. DataReader für das repräsentierte Topic existieren speichert die `Topic`-Instanz Referenzen auf die gekapselten Netzwerkadressen der Knoten. Für den Versand von Daten werden nur die Adressen der DataReader benötigt. Die Option, weitere Adressen zu speichern, wird für die erweiterte Funktionalität und das mögliche Routing von Daten in der Middleware benötigt. Eine `Topic`-Instanz hält des Weiteren die Daten vor, bis sie von einer Anwendung gelesen werden oder die festgelegte Obergrenze an Datensätzen pro `Topic`-Instanz überschreiten. Im letzteren Fall verdrängen dann die neuen Datensätze die ältesten noch nicht gelesenen.

Die Komponente, die für die Verteilung von Daten des lokalen Sensorknotens zuständig ist, ist die Klasse `DataSource`. Diese implementiert die Schnittstelle der DDS-Klasse `Publisher` und es existiert immer nur eine Instanz auf einem Middleware-Knoten. Diese `DataSource` enthält des Weiteren die Instanzen der

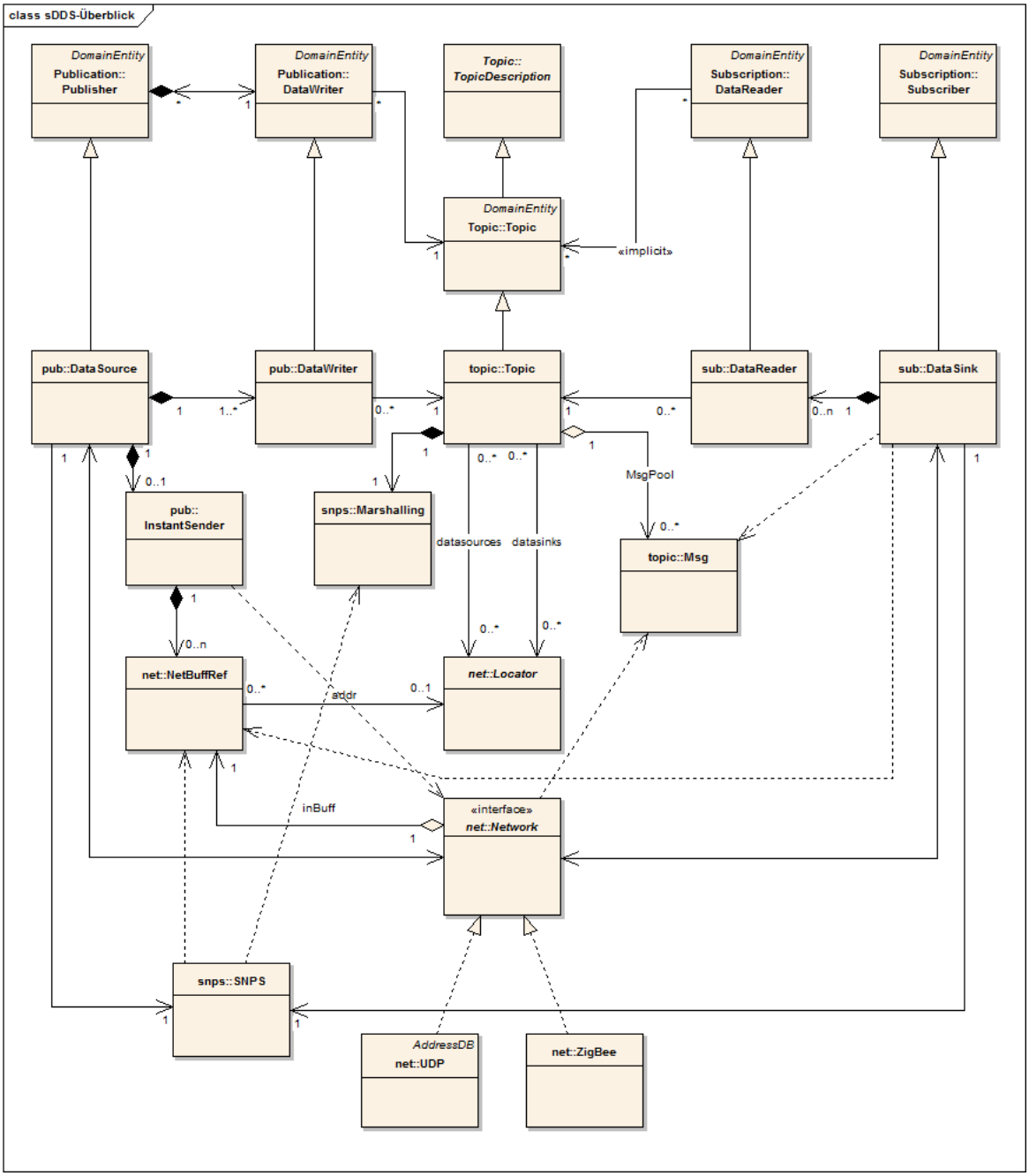

Abbildung 4.16: Übersicht über die wichtiges UML-Klassen von sDDS und ihre Beziehungen untereinander, sowie ihrer Zuordnung zu den Klassen des DDS-Standards

Klasse `DataWriter`, die von der entsprechenden DDS-Klasse abgeleitet sind. Diese sind fest in die Klasse `DataSource` integriert, damit bei einer Implementierung das Auftreten von Redundanzen verhindert werden kann.

Analog zu der Verteilung von Daten ist die Klasse `DataSink` für den Empfang und die Auswertung von Daten zuständig. Sie implementiert entsprechend die Schnittstellen der DDS-Klasse `Subscriber` und enthält die Instanzen der Klasse `DataReader` als festen Bestandteil. Auch von der Klasse `DataSink` gibt es nur eine Instanz auf jedem Knoten der Middleware.

Die Verbindung zu dem Kommunikationssystem der jeweiligen Hardware-Plattform erfolgt über die in Abschnitt 3.5.4 der Analyse des Kommunikationsprotokolls beschriebene universelle Schnittstelle. In diesem Entwurf wird diese von dem Interface `Network` dargestellt, von dem bisher zwei mögliche Ableitungen für spezifische Plattformen vorgesehen sind: Eine Klasse `ZigBee` für die Anbindung von ZigBee auf drahtlosen Sensorknotenplattformen und die Klasse `UDP` für die Implementierung des Interfaces mittels Verwendung des verbreiteten UDP/IP-Protokolls.

Das in Abschnitt 4.4 definierte Protokoll SNPS wird von der Klasse `SNPS` auf Ebene der Submessages und von der Klasse `Marshalling` auf Ebene der spezifischen Datenkodierung realisiert.

Die Netzwerk-Komponenten sind des Weiteren für die Bereitstellung und Verwaltung der gekapselten Netzwerkadressen zuständig und für die Bereitstellung von Datenpuffern für die Nachrichten. Die Klasse `NetBuffRef` nimmt in der sDDS-Middleware eine zentrale Stellung ein. Sie stellt den Zugriff auf eine SNPS-Nachricht bereit und enthält dafür den Datenpuffer und Referenzen auf Zustandsinformationen, die für die Kodierung oder Dekodierung einer SNPS-Nachricht benötigt werden. Instanzen der Klasse `NetBuffRef` werden von den verschiedenen sDDS-Komponenten ausgetauscht und für deren jeweilige Aufgaben verwendet.

4.5.2 Network

Der Entwurf der Klassen für das Netzwerk besteht aus mehreren getrennten Bereichen. Zuerst wird die in Abschnitt 3.5.4 der Analyse geforderte universelle Schnittstelle für Kommunikationssysteme nach den dort gemachten Vorgaben definiert. Danach werden die daraus abzuleitenden Klassen bezüglich der gekapselten Netzwerkadressen und Datenpuffer festgelegt. Die Eigenschaften der

von der Schnittstelle abgeleiteten spezifischen Klassen für ZigBee und UDP/IP werden zum Schluss beschrieben.

4.5.2.1 Univittchrae4o nnP0 -278 bes.kt6ibchnittstelle T 0 g 0 G0 g 0 G0 g 0 GET1 0 0 1123.54

ang(nn90[(v)25(on)n89[-aschr)-15(cheten)n90[f bder wer er

nnP02hoe(nm-6)4[Desch363[Em8(w60(vdm278auss)-278ihm(nnP0)gblesoe.()]TJ 0 22.7554 Td [Die(nn51

Nachrichten kann auf zwei Arten von der Middleware-Implementierung durchgeführt werden. Erstens über das Abfragen der Netzwerkschnittstelle mit der Methode `poll()`. Als zweite Option kann die Middleware-Implementierung das Interface `RecvHandler` implementieren und mit der Methode `setRecvHandler` bei der Netzwerk-Implementierung registrieren. Diese soll dann für jede ankommende Nachricht die im Interface `RecvHandler` vorgesehene Callback-Funktion aufrufen und der Middleware-Implementierung die neue Nachricht übergeben.

4.5.2.2 Ableitungen des `Network`-Interfaces

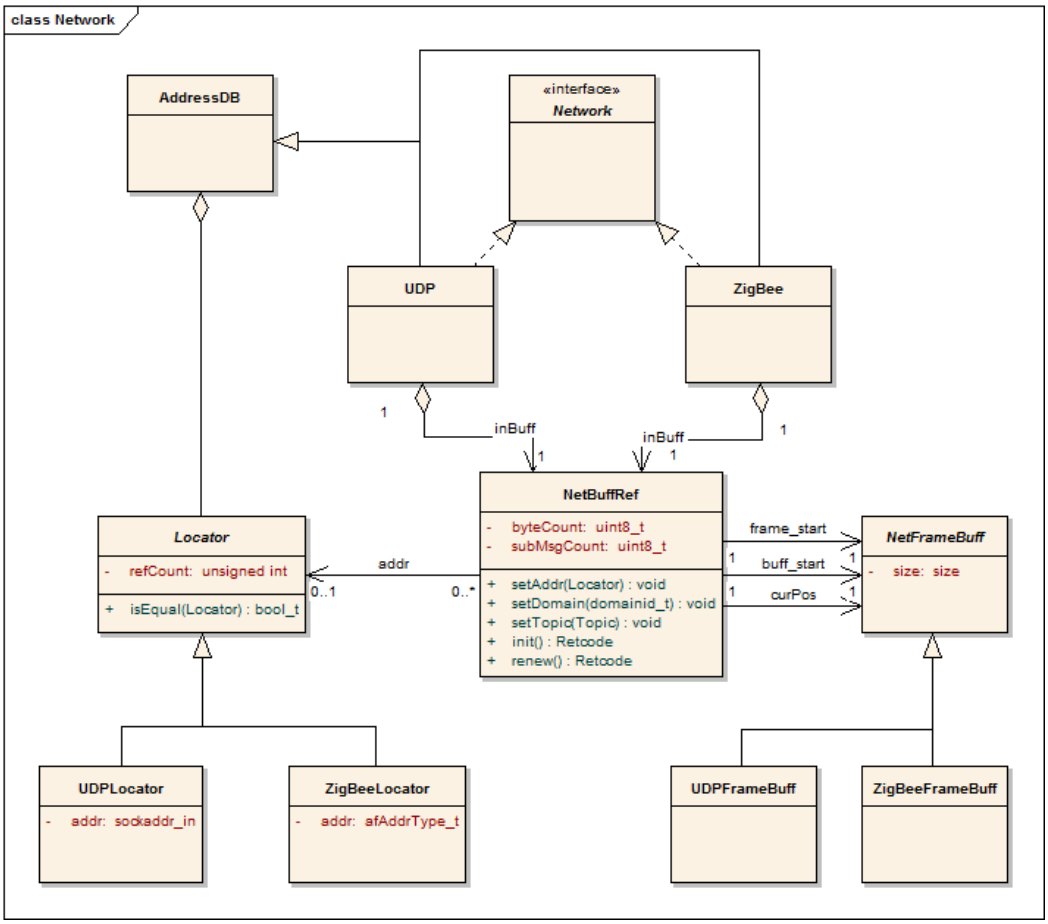

Abbildung 4.18: Darstellung der Ableitung des `Network`-Interfaces und die zentrale Klasse `NetBuffRef` für die Verarbeitung von Nachrichten in sDDS

Abbildung 4.18 stellt die in sDDS zentrale Klasse `NetBuffRef` und die bei-

gramm dar. Für die spezifischen Netzwerkplattformen sind die abstrakten Klassen `Locator` und `NetFrameBuff` abzuleiten und das Interface `Network` zu implementieren. Beide abgeleiteten `Network`-Klassen haben genau eine Referenz auf eine `NetBuffRef`-Instanz. Die Verwendung der Klasse `NetBuffRef` wird an späterer Stelle detailierter ausgeführt.

In diesem Entwurf für eine Ableitung des `Network`-Interfaces leiten die beiden Klassen `UDP` und `ZigBee` auch von der Klasse `AddressDB` ab, welche für die Verwaltung der bekannten Adressen auf einem sDDS-Knoten zuständig ist. Die Intention hinter der Klasse `AddressDB` ist es, dass individuelle Adressinformationen nur einmal in einer sDDS-Instanz existieren und von anderen Komponenten referenziert werden können. Daher sieht die Klasse `Listener` auch einen Referenzzähler vor.

Mit dem hier aufgeführten Entwurf für die Netzwerkschnittstelle ist es nur möglich, genau eine Ableitung in einer Middleware-Implementierung zu verwenden. Der gleichzeitige Betrieb mit mehreren Arten von Kommunikationssystemen bedarf einer Erweiterung des hier vorgestellten Designs von sDDS.

4.5.3 Einbettung von SNPS

Das Protokoll SNPS wird in diesem Entwurf von sDDS mittels der beiden Klassen `SNPS` und `Marshalling` abgebildet, deren Methoden statisch sind. Des Weiteren wird das Interface `TopicMarshalling` definiert, das für den datentyp-spezifischen Teil eines Topics zu implementieren ist und die Methoden enthält, die benötigt werden, um einen Topic-Datentyp zu kodieren bzw. zu dekodieren. Diese Klassen sind unabhängig von der jeweiligen Implementierung der Netzwerkschnittstelle. Sie verwenden ausschließlich die Klasse `NetBuffRef` und den darin enthaltenden abstrahierten Nachrichtenpuffer. Abbildung 4.19 zeigt die Klassen als UML-Klassendiagramm.

Die Aufgabe der Klasse `SNPS` ist es, die SNPS-Submessages in einen Nachrichtenpuffer zu schreiben bzw. sie aus diesem auszulesen. Dementsprechend gibt es für jeden Submessage-Typ zwei entsprechende Methoden. Die Klasse selber enthält keine Zustandsinformationen, sondern verwendet dafür die Klasse `NetBuffRef`, die sowohl den Nachrichtenpuffer als auch die Zustände als Attribute enthält. Für den Fall, dass eine sDDS-Implementierung einen Submessage-Typ nicht verarbeiten kann oder benötigt, gibt es die Methoden `discardSubMsg`

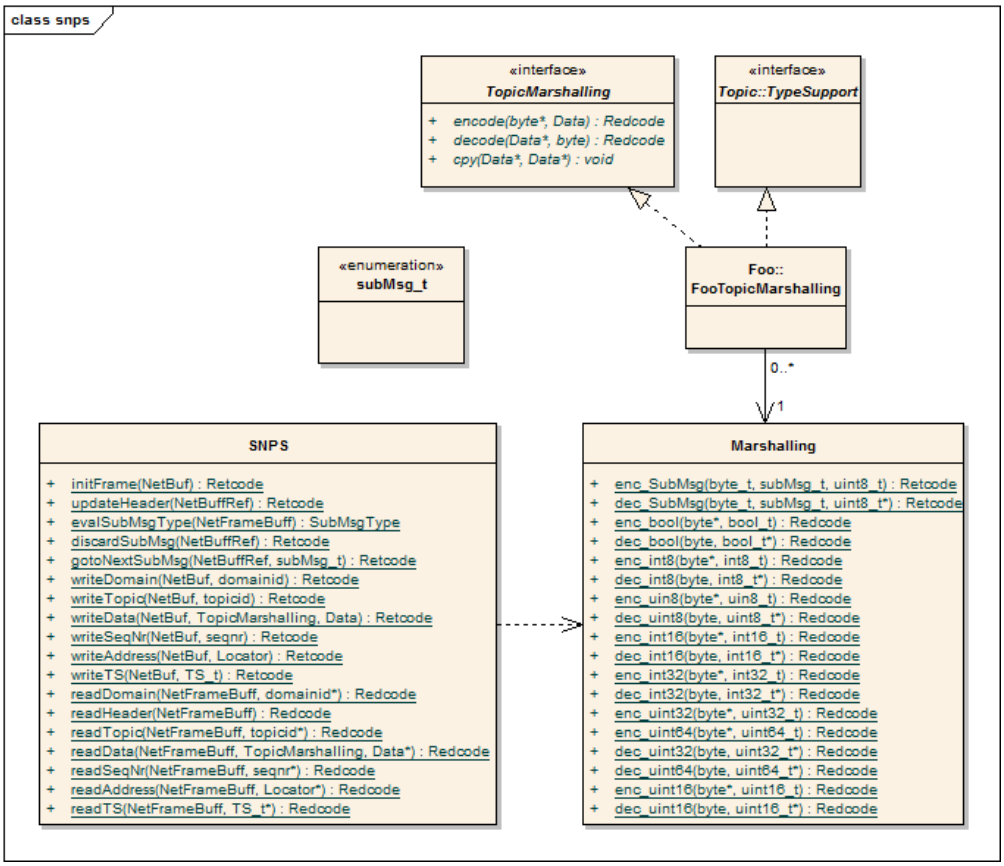

Abbildung 4.19: Darstellung der UML-Klassen `SNPS` und `Marshalling` für die Verwendung des SNPS-Protokolls

und `gotoNextSubMsg`. Erstere springt zu der nächsten Submessage in der Nachricht, während die zweite Methode zu der nächsten Submessage vorrückt, deren Typ übergeben wurde. Die `gotoNextSubMsg`-Methode wird dann benötigt, wenn eine zusammenhängende Gruppe von Submessages übersprungen werden soll.

Die Implementierungen des Interfaces `TopicMarshalling` und der Klasse `SNPS` greifen auf die Methoden der Klasse `Marshalling` zurück, um die primitiven Datentypen des Datenmodells zu kodieren bzw. dekodieren. Auch das Schreiben bzw. Lesen einer generischen Submessage wird von zwei Methoden dieser Klasse durchgeführt. Ein Implementierung dieser Klasse sollte nur die Methoden enthalten, die auch von der Middleware selber bzw. zur Verarbeitung der unterstützten Topic-Datentypen notwendig sind. Dasselbe gilt für die Methoden der Klasse `SNPS` und der benötigten Submessages. Das Überspringen von Submessages ist eine immer benötigte Funktionalität.

4.5.4 Topic Management

Der zentrale Punkt des Entwurfs des Topic Managements, wie es in Abbildung 4.2 auf Seite 109 der Grobarchitektur gezeigt wurde, ist die Klasse `Topic`, welche die Schnittstellen der gleichnamigen DDS-Klasse implementiert. Die Abbildung 4.20 zeigt die UML-Darstellung der Klassen, die für das Topic Management zuständig sind.

Das Topic Management hat die Aufgabe Daten zwischen den Knoten der Middleware zu verteilen und an die richtigen Endpunkte zu leiten. Dabei sieht der hier gemachte Entwurf vor, dass die Topic-Klasse passiv ist und ihre zentral gehalten Daten von anderen Entitäten des sDDS-Systems direkt zugreifbar sind. Eine Implementierung der Klasse `Topic` hat hierbei den wechselseitigen Ausschluss sicherzustellen.

Im Folgenden werden die Aufgaben des Topic Managements und die damit betrauten Klassen näher beschrieben. Neben der primären Klasse `Topic` ist dies die Klasse `TopicDB`, welche alle Topics eines Knotens referenziert und die Klasse `Msg`, welche ein Datencontainer für das Vorhalten von Anwendungsdaten ist.

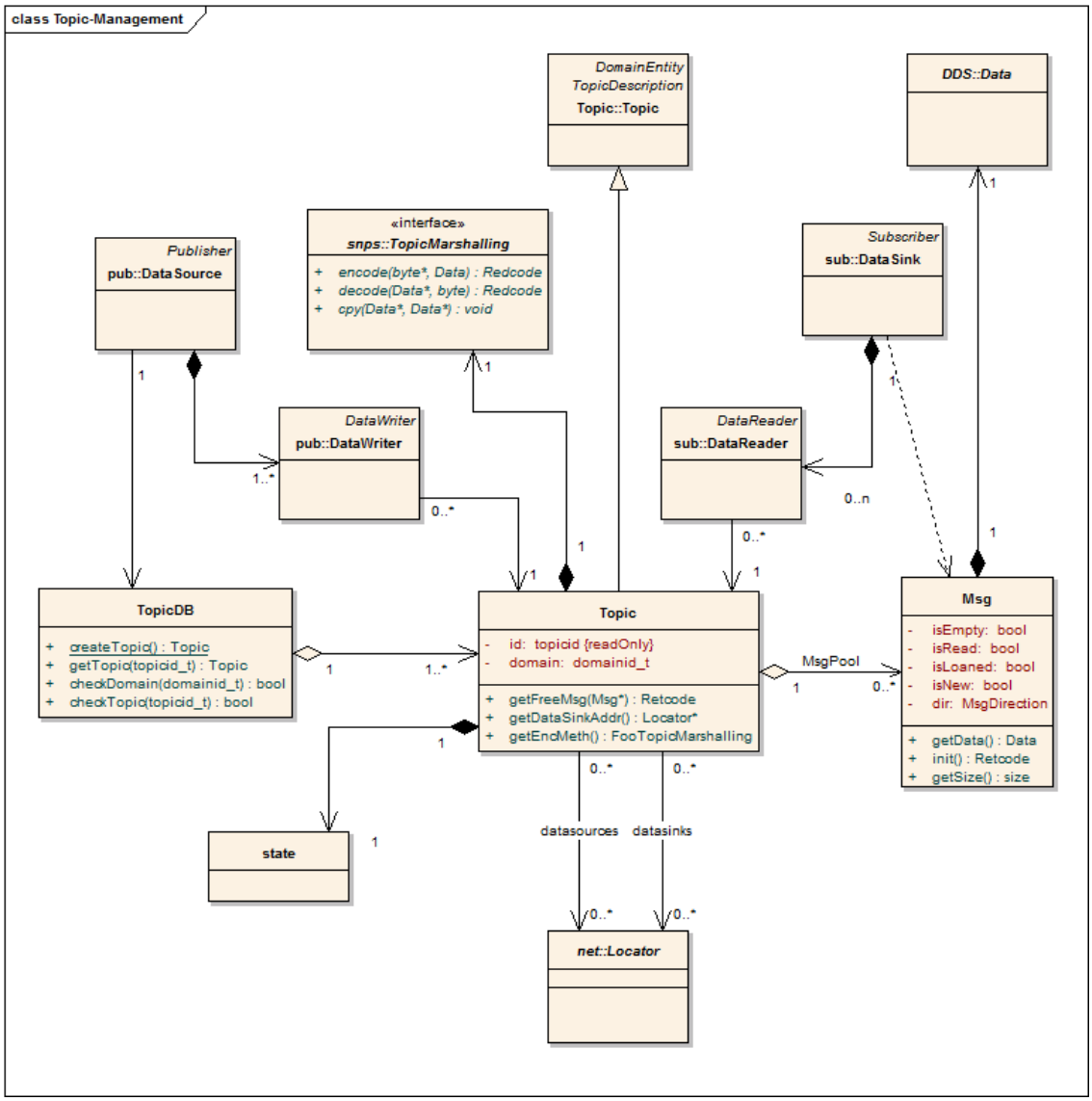

Abbildung 4.20: UML-Klassen für das Topic Management

4.5.4.1 Topic

Die Klasse `Topic` ist der zentrale Datensammelpunkt in einem sDDS-System. Intern werden die Topics über einen vorzeichenlosen Ganzzahlentyp referenziert, die *TopicID*. Der DDS-Standard sieht vor, dass die Topics über Namen referenzierbar sind. Der Entwurf von sDDS sieht bisher keine Funktionalität vor, solche Topic-Namen auf die *TopicID* abzubilden. Dies ist ein eigenständiges Thema und kann auf unterschiedliche Weise angegangen werden. Für diese Arbeit wird davon ausgegangen, dass die Abbildung der Topic-Namen auf die *TopicID*s statisch ist und bei der Generierung der sDDS-Implementierung festgelegt wird.

Es gibt keine explizite Trennung zwischen den Topics verschiedener Domains auf einem Knoten, wie es beispielsweise bei der DDS-Implementierung OpenSplice DDS der Fall ist.

Jedes Topic bedient einen ihm zugeordneten Datentyp. Für die interne Verarbeitung der Daten in der Middleware ist die Art und Struktur der Daten nicht von Bedeutung, und sie werden in der abstrakten Klasse `Data` gekapselt.

TopicMarshalling

Für die Kodierung und Dekodierung der Daten des dem Topic zugeordneten Datentyps legt das Interface `TopicMarshalling` entsprechende Methoden fest. Bei der Generierung der Middleware muss das Interface `TopicMarshalling` für jeden definierten Topic-Datentyp implementiert und als Referenz bei der Initialisierung des Topics diesem zugewiesen werden. Ein Topic-Datentyp setzt sich in der Regel aus weiteren Datentypen des Datenmodells von sDDS zusammen. Für diese Datentypen gibt es die Konvertierungsmethoden der Klasse `Marshalling`, welche bei der Generierung der `TopicMarshalling`-Implementierung verwendet werden sollen.

Die Methode zur Erzeugung bzw. zum Auslesen einer Daten-Submessage bekommt eine `TopicMarshalling`-Implementierung übergeben und delegiert an diese die Kodierung bzw. Dekodierung der Topic-spezifischen Datensätze.

TopicDB

Alle Topics auf einem sDDS-Middleware-Knoten sind über die Klasse `TopicDB` zugreifbar. Diese realisiert das Singelton und Factory-Pattern und kann Topics

dynamisch erzeugen, wenn dies anwendungsseitig benötigt wird. Die spezifische Initialisierung der Topics muss mittels generierten Programmcodes geschehen.

Die Klasse `TopicDB` gibt anderen Entitäten über die Topic-ID Zugriff auf die Topics und kann überprüfen, ob eine Domain auf dem jeweiligen Knoten bedient wird.

4.5.4.2 Subscription Management

Das Subscription Management ist Teil des Topic Managements, da das Topic in einem sDDS-System speichert, welche Middleware-Knoten mit ihm assoziiert sind. Dies geschieht ab dem Moment, in dem die Klasse instantiiert wird, unabhängig davon, ob auf dem Knoten DataReader oder DataWriter existieren. Die Grenze für die Menge der vorgehaltenen Adressdaten wird bei der Erzeugung der Middleware abhängig von dem verfügbaren Speicher der Plattform festgelegt. Wenn diese Grenze überschritten wird, ist in diesem Entwurf vorgesehen, dass keine neuen Subscriptions aufgebaut werden können. Spätere Erweiterungen können hier bessere Vorgehensweisen definieren.

Die Speicherung von assoziierten Knotenadressen in einem lokalen Topic hat den Zweck, auf den Knoten der Middleware verteilt und redundant das Wissen über die Kapazitäten der Knoten im sDDS-System vorzuhalten. Auf Basis dieses Wissens kann erweiterte DDS-Funktionalität entworfen und implementiert werden, beispielsweise Routing, zuverlässige Datenübertragung usw.. Diese Funktionalität ist optional und kann bei der Generierung der Middleware ausgeschlossen werden.

4.5.4.3 Datenvorhaltung

Der DDS-Standard sieht wie in Abschnitt 2.3 der Grundlagen und Abschnitt 3.4.1.3 der Analyse beschrieben die Vorhaltung von Daten innerhalb der Middleware vor, bis entweder die vorgegeben Grenzen bezüglich der Menge überschritten sind oder die Daten nicht mehr benötigt werden. Damit ist es möglich, dass Anwendungen auf ältere Daten zugreifen können, wenn die Knoten, auf denen sie laufen, neu ins System kommen oder temporär nicht mit anderen Knoten kommunizieren konnten. Entsprechend sieht der DDS-Standard vor, dass Daten sowohl auf Seite der DataReader als auch der DataWriter Daten vorzuhalten sind.

In diesem sDDS-Entwurf erfolgt die Datenvorhaltung durch die lokale `Topic`-Instanz und ist damit Teil des Topic Managements. Da drahtlose über Sensorknoten nur sehr wenig Speicher verfügen, ist die Datenvorhaltung problematisch und Effizienz und weitgehende Konfigurier- und Anpassbarkeit von großer Bedeutung. Um die Konformität zu dem definierten Verhalten des DDS-Standards beibehalten zu können, muss für jeden DataReader mindestens ein Datensatz vorgehalten werden können, wie es in Abschnitt 3.4.1.3 der Analyse über die DDS-Schnittstelle des DataReader festgestellt wurde.

Die einzelnen Datensätze werden in sDDS mit Instanzen der Klasse `Msg` verwaltet. Die mögliche Anzahl dieser Instanzen pro Topic wird bei der Generierung der Middleware festgelegt, und diese sind danach fest einem Topic zugeordnet. Sie enthalten einen datentypspezifischen Datenpuffer, in dem die Daten dekodiert in der von der Anwendung benötigten Struktur vorliegen. Die Klasse `Msg` speichert für jeden Datensatz einige Verwaltungsinformation. Wichtig ist dabei die Information, ob der Datensatz bereits von der Anwendung gelesen wurde (`isRead`) und ob die Anwendung den Speicher, in dem die Daten stehen, gerade selber verwendet (`isLoan`). Da ein Topic sowohl aus-, wie auch eingehende Daten vorhält, wird gespeichert, ob der Datensatz von einem lokalen DataWriter oder einem externen stammt.

4.5.5 DataSource

Die Rolle der *Datenquelle*, wie sie in Abbildung 4.2 auf Seite 109 der Grobarchitektur dargestellt ist, übernimmt die Klasse `DataSource`. Auf einem sDDS-Knoten gibt es immer nur eine Instanz dieser Klasse, so dass die gesamte Kommunikation nach außen über sie läuft. Abbildung 4.21 zeigt das UML-Diagramm für die Klassen, die für den Versand von Daten zuständig sind. Im Weiteren wird detailierter auf die einzelnen Aspekte eingegangen.

4.5.5.1 DDS-Anwendungsschnittstelle

Die Anwendungsschnittstelle von DDS sieht für das Veröffentlichen von Daten die Klasse `DataWriter` vor, die für jeden spezifischen Topic-Datentyp abgeleitet wird. In sDDS wird von der DDS-Klasse `DataWriter` eine gleichnamige sDDS-Klasse abgeleitet, deren Instanzen Teil der `DataSource` sind. Jede Instanz der sDDS-Klasse `DataWriter` hat eine Referenz auf das Topic, welches

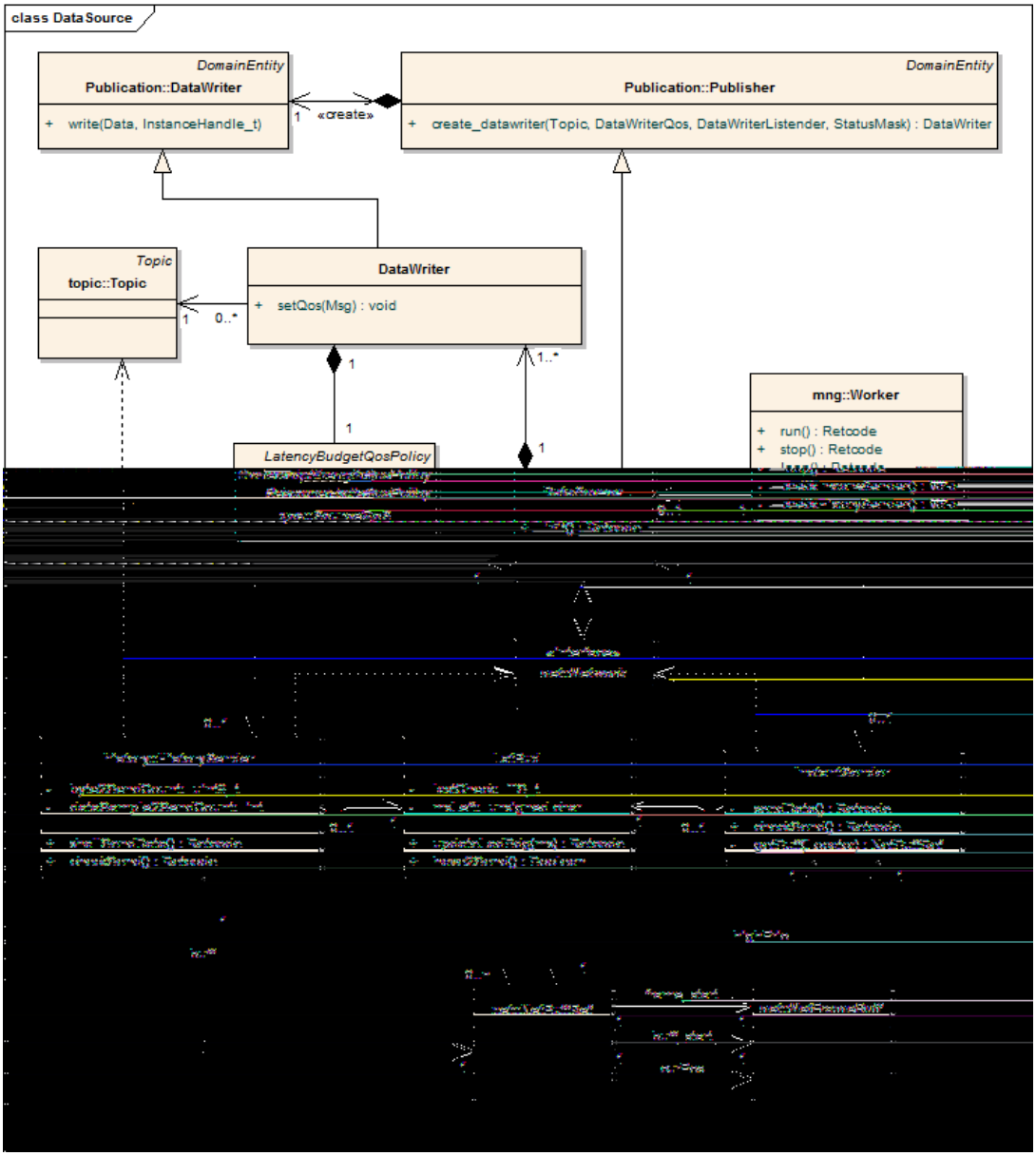

Abbildung 4.21: UML-Klassen für die Rolle der Datenquelle im sDDS

für den jeweiligen Topic-Datentyp zuständig ist, und speichert QoS-Parameter für die konfigurierte QoS-Funktionalität der sDDS-Implementierung.

Die Klasse `DataSource` implementiert die Schnittstelle der Klasse `Publisher`, was in diesem Entwurf vorerst auf die Methode `create_datawriter()` beschränkt ist.

Der DDS-Standard legt fest, dass die Schnittstelle für die Veröffentlichung von Daten, also die Methode `write()`, an den jeweiligen Datentyp angepasst ist und dementsprechend eine Ableitung der DataWriters für jeden Topic-Datentyp existiert. Dieses Verhalten wird in sDDS emuliert. Wie in Abschnitt 4.5.4 über das Topic Management bereits beschrieben wurde, besitzt jede Topic-Instanz spezifische Methoden für die Umwandlung der Topic-Datentypen. Damit benötigt die sDDS-Klasse `DataWriter` keine spezifische Anpassung, sondern wird für alle Topic-Datentypen verwendet. Die Implementierung von sDDS muss entsprechend die Schnittstelle des Topic-Datentyp-spezifischen DataWriter implementieren und delegiert die Verarbeitung der gekapselten Daten an die `DataSource`-Instanz.

Die QoS-Funktionalität einer sDDS-Implementierung, die die `DataSource`-Instanz betrifft wird, über die QoS-Parameter der DataWriter konfiguriert, die statisch bei der Generierung der jeweiligen sDDS-Implementierung gesetzt werden. Eine dynamische Anpassung durch die Anwendung soll Aufgabe zukünftiger Erweiterungen sein. Dementsprechend wird die DDS-Schnittstelle für QoS nicht implementiert.

4.5.5.2 Datenversand

Das Senden der Daten wird durch die bei der Konfiguration der sDDS-Middleware gewählten QoS-Richtlinien beeinflusst. In Abschnitt 3.4.2 wurden die QoS-Richtlinien History und LatencyBudget als relevant für die sDDS-Middleware eingestuft. History limitiert die Anzahl der vorzuhaltenden Nachrichten und der Parameter von LatencyBudget gibt der Middleware eine Information, wie schnell ein Datensatz verschickt werden muss.

In der Analyse in Abschnitt 3.5.1 wurde festgestellt, dass es vorteilhaft sein kann, wenn Daten nicht sofort, sondern gesammelt an Gruppen von Empfängern verschickt werden, auch wenn es dabei möglich ist, dass einzelne Knoten Teile der

Nachricht nicht benötigen. Dabei würden die Daten aber nicht sofort sondern verzögert versendet werden. Wenn eine Anwendung zeitliche Vorgaben bezüglich der Auslieferung der Daten hat, kann die Funktionalität der QoS-Richtlinie LatencyBudget verwendet werden.

Für den Versand von Daten gibt es verschiedene Optionen. Diese sind in diesem Entwurf von der gewählten Konfiguration der QoS-Richtlinien History und LatencyBudget abhängig und sollen beispielhaft zeigen, wie QoS-Richtlinien in sDDS integriert werden können. Es werden hier zwei unterschiedliche Module für den Versand von Daten vorgestellt, von denen in der Konfiguration der sDDS-Middleware über die QoS-Richtlinien eine auszuwählen ist.

HistorySender

Wie in Abschnitt 4.5.4 über den Entwurf des Topic Managements beschrieben, kann das Topic Daten in Instanzen der Klasse `Msg` vorhalten. Die Funktionalität der QoS-Richtlinie History soll darauf aufbauen. Wenn die Richtlinie in der Konfiguration aktiviert wurde, werden die Daten der Anwendung mit einer `Msg`-Instanz des jeweiligen Topics assoziiert.

Der Versand der Nachrichten erfolgt über eine Instanz der Klasse `HistorySender`, die bei dieser Konfiguration Teil der DataSource ist. Sie sammelt Informationen über die zu versendenden Daten und wenn definierte Bedingungen erfüllt sind, wird eine SNPS-Nachricht zusammengesetzt und versendet.

Wenn der QoS-Parameter LatencyBudget gesetzt ist, legt die Nachricht, die als nächstes versendet werden muss, den Zeitpunkt des Versands fest. Tritt dieser ein, wird die SNPS-Nachricht mit allen Daten zusammengebaut. Bei diesem Zusammenbau ist der `HistorySender` in der Lage, die Reihenfolge der SNPS-Submessages zu optimieren und damit die Nachrichtenlänge zu minimieren. Eine Instanz der Klasse `Worker` hat die Aufgabe zyklisch zu überprüfen, ob Daten versendet werden müssen und ggf. die `HistorySender`-Instanz darüber zu informieren. Alternativ wird bei der Verarbeitung jedes neues Datensatzes durch die `HistoySender`-Instanz diese Überprüfung durchgeführt.

InstantSender

Die Klasse `InstantSender` ist eine Alternative zu `HistorySender`, welche verwendet werden muss, wenn kein History-QoS benötigt wird. Da die Datensät-

ze der Anwendungen nicht zentral vorgehalten werden, ist es notwendig, dass diese sofort in entsprechende Nachrichtenpuffer geschrieben werden. Eine Optimierung der Reihenfolge ist damit nicht notwendig, so dass die Nachrichtenlänge größer sein kann als bei der Klasse `HistorySender`.

Der `InstantSender` kann bei der Konfiguration der QoS-Richtlinie LatencyBudget ebenfalls verschiedene Daten in einer Nachricht zusammenfassen und verzögert versenden. Dabei kann das Problem entstehen, dass hochpriore Datensätze ausgebremst oder mit niederprioren zusammengefasst werden und diese dann die Übertragunszeit der hochprioren Daten erhöhen. Um dem entgegenzuwirken und allgemein die Flexibilität für die Zuteilung von Nachrichten an Empfängergruppen zu vergrößern, verfügt der `InstantSender` über mehrere Nachrichtenpuffer, wobei einer davon speziell für hochpriore Daten reserviert sein muss. Hochpriore Daten sind dann als SNPS-Nachricht sofort in diesen reservierten Puffer zu schreiben und zu versenden. Eine Zusammenfassung mit anderen Datensätzen darf nicht stattfinden. Eine Ausnahme sind notwendige Managementinformationen für diesen Datensatz wie Zeitstempel oder Sequenznummern. Für normale Datensätze werden Nachrichtenpuffer bevorzugt, die bereits einer ähnlichen Gruppe von Empfängern zugeordnet sind. Andernfalls wird ein neuer Nachrichtenpuffer ausgewählt.

Werden in Nachrichtenpuffer kodierte Datensätze nicht sofort versendet, sondern verzögert versendet, wird auch hier eine Instanz der Klasse `Worker` verwendet, um regelmäßig den nächsten Versandtermin zu ermitteln. Analog zu dem `HistorySender` kann diese Überprüfung zusätzlich auch bei der Verarbeitung jedes neuen Datensatzes stattfinden.

4.5.6 DataSink

Die Rolle der *Datensenke* in einem Sensornetz übernimmt in sDDS die Instanz der Klasse `DataSink`. Sie existiert immer nur einmal auf einem sDDS-Knoten und hat die Aufgabe eingehende Nachrichten zu verarbeiten und den Anwendungen die enthaltenden Daten zur Verfügung zu stellen. Die Klasse `DataSink` implementiert bzw. emuliert dafür die DDS-Schnittstellen der DDS-Klassen `Subscriber` und `DataReader`. Für die Implementierung der DDS-Schnittstellen der Klasse `DataReader` wird eine gleichnamige sDDS-Klasse abgeleitet, deren Instanzen integraler Bestandteil der `DataSink`-Instanz sind.

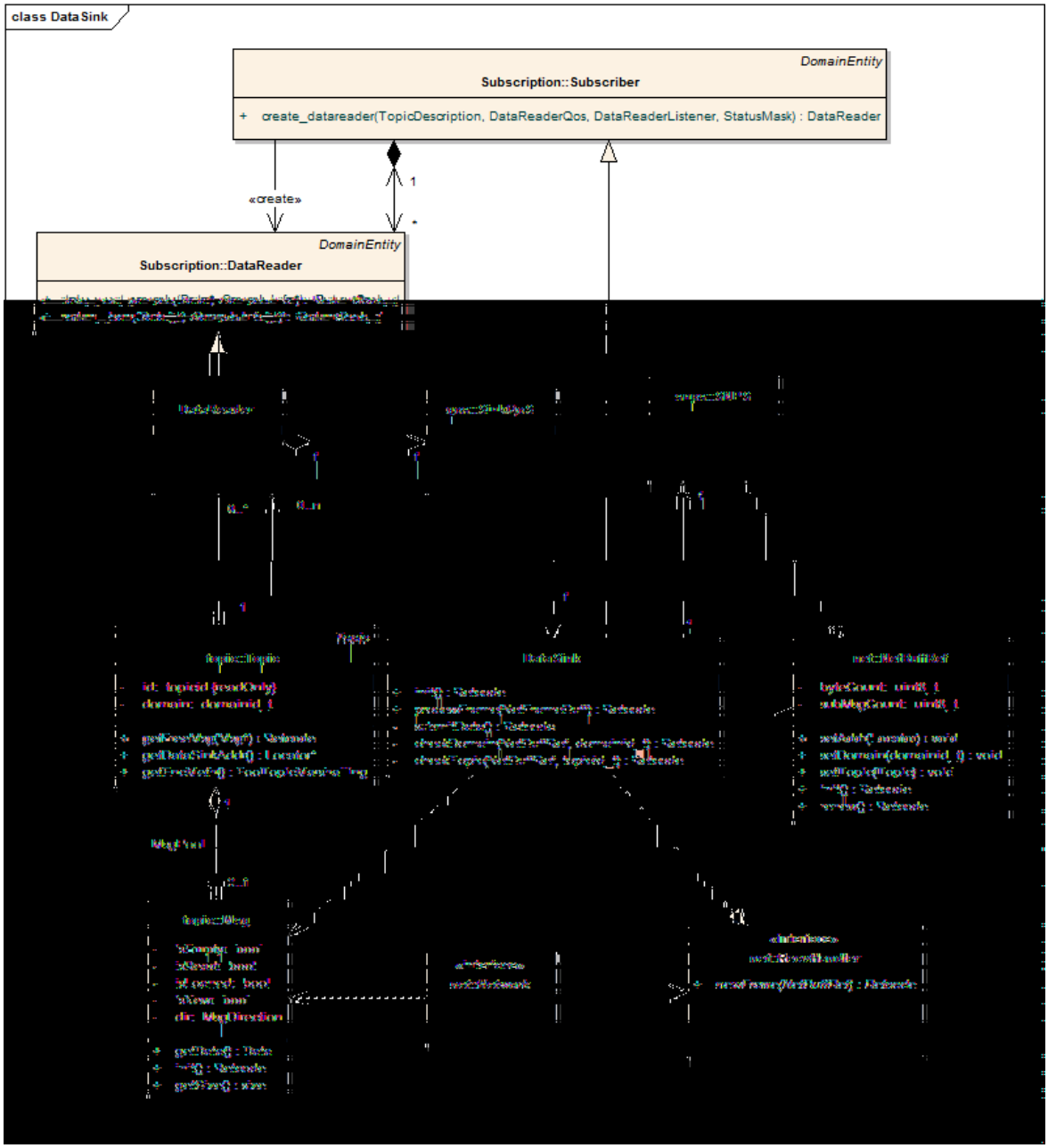

Abbildung 4.22: UML-Diagramm der Klassen für den Empfang und die Verarbeitung von Nachrichten in einem sDDS-Systems

Da jeder Knoten in der Lage sein muss, Daten empfangen zu können, ist es notwendig, dass auf jedem eine Instanz der Klasse `DataSink` existiert. Abbildung 4.22 zeigt die UML-Klassen, die für den Empfang von Daten zuständig sind und ihre Relationen untereinander. Im weiteren werden die einzelnen Aspekte genauer beschrieben.

4.5.6.1 DDS-Anwendungsschnittstelle

Die Konzeption der Klasse `DataSink` und der von ihr verwendeten Komponenten ist der Klasse `DataSource` ähnlich. Die zentrale Klasse `DataSink` implementiert die Schnittstelle der DDS-Klasse `Subscriber`, welche bei diesem Entwurf vorerst auf die Methode `create_datareader()` beschränkt ist.

Die sDDS-Klasse `DataReader` wird von der gleichnamigen DDS-Klasse abgeleitet, und ihre Instanzen besitzen Referenzen auf die korrespondierenden `Topic`-Instanzen und QoS-Parameter für den jeweiligen DataReader. Die datentypspezifischen Schnittstellen der DDS-Klasse `DataReader` werden dadurch emuliert, dass die spezifischen Parameter gekapselt und an die generische Schnittstelle der Klasse `DataSink` weitergeleitet werden.

Als Schnittstelle zur Anwendung sieht der Entwurf in dieser Version nur die Polling-Schnittstelle des DataReaders vor, wie in Abschnitt 3.4.2.1 der Analyse als Teil der zu implementierenden Basisfunktionalität von DDS festgelegt. Um den späteren Implementierungsaufwand zu reduzieren, wird hierfür die Methode `take_next_sample()` verwendet, da diese immer nur einen Datensatz der Anwendung zurückgibt. Die Semantik dieser Methode, wie sie vom DDS-Standard definiert ist, sieht vor, dass ein Datensatz nur einmal gelesen werden kann.

Wie in Abschnitt 4.5.4 bezüglich des Entwurfs des Topic Managements beschrieben, werden die Datensätze von der Middleware in den Instanzen der Klasse `Msg` bei den jeweiligen Topics vorgehalten, bis eine Anwendung sie liest. Daher ist minimal eine Datenvorhaltung von einem Datensatz notwendig. Sollte ein neuer Datensatz auf dem Knoten eintreffen, ohne dass der letzte Datensatz gelesen wurde, ist entweder der älteste nicht gelesene zu überschreiben oder der neu eingetroffene zu verwerfen. Die Entscheidung hierzu obliegt der jeweiligen Implementierung und deren Konfiguration.

Der DDS-Standard sieht die Funktionalität vor, dass eine Anwendung entweder Speicher für den zu lesenden Datensatz bereitstellen oder sich diesen von der

Middleware „leihen" kann. Diese Funktionalität ist in diesem Entwurf vorgesehen und wird als Methode `return_loan()` des DataReader durch die Klasse `DataSink` implementiert. Dafür wird der Anwendung der Speicherbereich mit dem dekodierten Datensatz der `Msg`-Instanz übergeben. In der jeweiligen `Msg`-Instanz muss der Vorgang des „Ausleihens" über das entsprechende Attribut `isLoaned` vermerkt werden. Danach darf die Datenstruktur erst wiederverwendet werden, wenn sie von der Anwendung freigegeben wurde.

4.5.6.2 Datenempfang

Für den Empfang von Nachrichten implementiert die Klasse `DataSink` die Callback-Funktion des Interfaces `Network`, über die sie von der Netzwerk-Implementierung über neue Nachrichten informiert wird. Diese verarbeitet sie unter Zuhilfenahme der Methoden der Klasse `SNPS` für die Protokollimplementierung.

Die empfangene Nachricht ist in einer Instanz der Klasse `NetBuffRef` gekapselt, die auch die Informationen bezüglich der Zustände der Kontexte der Submessages in der Nachricht speichert. Die Verarbeitung der Nachricht erfolgt in der Methode `processFrame()` und umfasst die Überprüfung, ob die Daten der Nachricht relevant für den jeweiligen Knoten von sDDS sind. Datensätze sind immer einer Domain und einem Topic zugeordnet. Existieren auf einem Knoten Topics, die in der Nachricht referenziert werden, dann wird der Datensatz gelesen, andernfalls übersprungen.

Ist eine Datennachricht fertigt dekodiert, wird ihr Inhalt mittels der spezifischen Kodierungsmethoden des Topics übersetzt und in einer `Msg`-Instanz des Topics gespeichert. Sind innerhalb des Kontexts eines Datensatzes weitere Informationen kodiert, beispielsweise Zeitstempel oder Sequenznummern, dann werden diese ebenfalls dekodiert und der `Msg` angehängt, sofern sie für die Funktionalität des Knotens relevant sind. Werden die Informationen in einer Submessage nicht benötigt, sind sie zu überspringen.

Anwendungen verwenden die beschriebene `take_next_sample()`-Methode, um neue Nachrichten zu lesen. Diese greift auf das jeweilige Topic zu und entnimmt den ältesten noch nicht gelesenen Datensatz. Sollte in der Konfiguration der Middleware festgelegt worden sein, dass auch die DDS-Klasse `DataSampleInfor` implementiert werden soll, ist eine entsprechende Datenstruktur anzulegen und mit den Werten aus der `Msg`-Instanz zu füllen. Sollte die

Klasse `Msg` die entsprechenden Daten nicht enthalten, benötigt eine Middleware-Implementierung Funktionalität, die die benötigten Informationen emuliert. Dies kann durch Konvertierung existierender oder statischer Informationen geschehen.

4.5.7 OS-SSAL

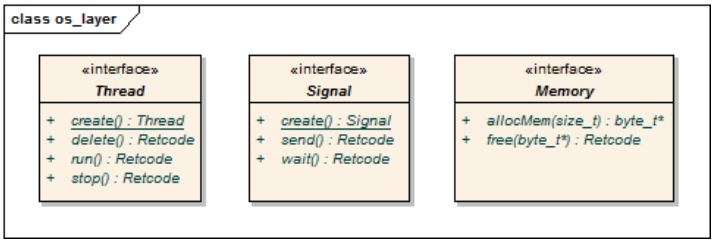

Abbildung 4.23: UML-Darstellung der Interfaces für den OS-SSAL

Die Schnittstelle für die Abstraktion der Betriebssystemfunktionalität, wie sie in Abschnitt 4.23 der Analyse gefordert wurde, ist in Abbildung 4.23 dargestellt und wird in dieser Arbeit OS-SSAL (Operation System – System Software Abstraction Layer) genannt. Vorerst wurde sie auf die Funktionalität der Nebenläufigkeit, Signalisierung und Speicherverwaltung beschränkt und auf die minimal notwendigen Methoden für ihre Verwendung reduziert.

Das Interface `Thread` stellt die Basisfunktionalität für die Verwendung von Threads, also Nebenläufigkeit, bereit. Dies umfasst das Erzeugen, Starten, Stoppen und Löschen eines Threads. Verbunden mit dieser Funktionalität ist die des Interfaces `Signal`. Auf ein Signal kann ein Thread warten, bis es von einem anderen Thread gesendet wird. Das Interface `Memory` ermöglicht es, Speicher zu allokieren und wieder freizugeben.

Die Funktionalität dieser Interfaces soll durch eine Implementierung entweder auf existierende Funktionalität der Plattform abgebildet werden oder durch eine eigene für die Plattform angepasste Implementierung realisiert werden. Bevor Programmcode dazu generiert wird, ist zu prüfen, ob die Funktionalität auf Basis der Konfiguration der Middleware benötigt wird.

4.5.8 Verarbeitung von SNPS-Nachrichten

Nach dem die statische Modellierung der Klassen von sDDS vorgestellt ist, soll nun das Zusammenspiel der Komponenten für die primären Aufgaben der Middleware beschrieben werden. Die Hauptaufgaben der sDDS-Middleware sind zuerst das Annehmen von Daten der Anwendung, die Verpackung dieser Daten in eine SNPS-Nachricht und das Versenden an die Empfängerknoten im Netzwerk. Dort sind die Nachrichten nach dem Empfang zu dekodieren und die Daten der Anwendung zur Verfügung zu stellen. Der dritte Schritt ist dann das Lesen der Daten durch die Anwendung selber.

4.5.8.1 Versenden von Daten

In Abbildung 4.24 wird der Vorgang zur Erzeugung einer SNPS-Nachricht aus den übergebenen Daten der Anwendung dargestellt. Die Darstellung wurde etwas vereinfacht, um sie nicht mit Details zu überfrachten. Daher wird die Rückkehr der Methoden der Klasse `SNPS` nicht explizit dargestellt. In Abschnitt 4.5.5.2 wurden zwei alternative Klassen für den Versand vorgestellt: `HistorySender` und `InstantSender`. Im Weiteren wird die Verwendung des `InstantSender` beschrieben.

Der Vorgang geht davon aus, dass es den Topic-Datentyp *Foo* gibt, für den die DDS-spezifische Anwendungsschnittstelle generiert wurde. Der Vorgang beginnt damit, dass die Anwendung die `write()`-Methode des spezifischen `FooDataWriter` aufruft und ihr eine `Foo`-Datenstruktur übergibt. Wie in Abschnitt 4.5.5.1 dargestellt, wird diese auf die generische `write()`-Methode der Klasse `DataSource` abgebildet, die diesen Aufruf an den `InstantSender` delegiert.

Der `InstantSender` kodiert die notwendigen Informationen direkt in den Netzwerkpuffer einer Instanz der Klasse `NetBuffRef`. Es ist möglich, dass eine `InstantSender`-Instanz mehrere dieser `NetBuffRef`-Objekte besitzen kann, um die Verteilung der Nachrichten zu optimieren. In diesem Fallbeispiel erfolgt die Auswahl auf Basis der Empfänger der Daten. Die Anwendung verwendet bei der Initiierung des Vorgangs eine Referenz auf den Topic-spezifischen DataWriter bzw. dessen Repräsentierung innerhalb der Implementierung der Klasse `DataSource`. Über diese Referenz existiert eine Assoziation zu dem jeweiligen

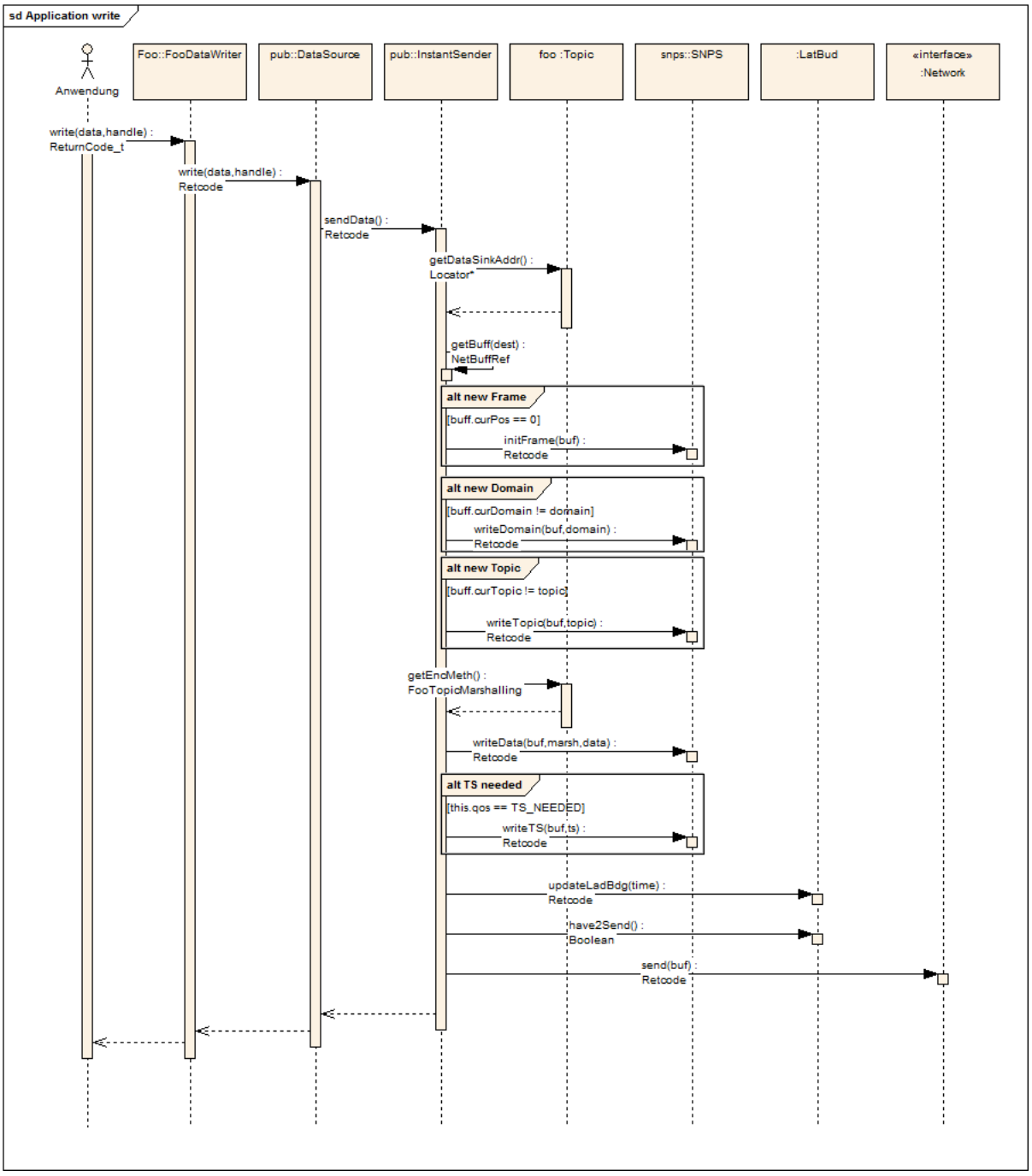

Abbildung 4.24: UML-Sequenzdiagramm über den Vorgangs des Schreibens von Daten in sDDS

Topic und damit auf die Liste der Empfänger und die spezifischen Datenkonvertierungsmethoden.

Über die Methode `getDataSinkAddr()` der Klasse `Topic` werden die gekapselten Empfängeradressen für den Datentyp geholt und intern in der Klasse `DataSource` eine passende Instanz von `NetBuffRef` ausgewählt. Die Adressen werden dem Puffer zugeordnet, wenn dies nicht schon vorher geschehen ist. Bei der anschließenden Kodierung der Daten ist von verschiedenen Fallunterscheidungen auszugehen, abhängig davon, welche Daten bereits vorher in den Nachrichtenpuffer geschrieben wurden. Der Aufbau der Kontexte im SNPS-Protokoll wurde in Abschnitt 4.4.3 beschrieben. Die Schritte für die Kodierung der sind in der folgenden Reihenfolge durchzuführen:

- Im Fall, dass bisher noch keine Daten in dem Nachrichtenpuffer geschrieben wurden, wird dieser mit der Methode `initFrame()` der Klasse `SNPS` initialisiert, indem der SNPS-Header in den Nachrichtenpuffer geschrieben wird.
- Sollte in dem `curDomain`-Attribut des `NetBuffRef` keine Domain referenziert sein, oder diese sich von der Domain des zu schreibenden Datensatzes unterscheiden, dann wird die Domain-Submessage in den Nachrichtenpuffer kodiert.
- Sollte in dem `curTopic`-Attribut des `NetBuffRef` kein oder ein anderes Topic referenziert sein als das dem Datensatz zugeordnete, dann wird eine entsprechende Topic-Submessage kodiert.
- Für die Kodierung der Daten wird die datentypspezifische Kodierungsmethode von dem jeweiligen Topic mittels der Methode `getEncMeth()` geholt.
- Das Datum und eine Referenz auf die Kodierungsmethode wird der Methode `writeData()` der Klasse `SNPS` übergeben, die den Datensatz zusammen mit der entsprechenden Submessage als Header in den Nachrichtenpuffer schreibt.
- Nach dem Datensatz können weitere damit assoziierte Informationen in dessen Kontext geschrieben werden. Dies wird in Diagramm 4.24 für das Beispiel eines Zeitstempels gezeigt.

- Nach dem Schreiben der Daten wird die Zeit, bis der der Nachrichtenpuffer mit den enthalten Daten versendet werden muss, aktualisiert. Die Zeitdauer ist vom gesetzten QoS-Parameter der QoS-Richtlinie *LatencyBudget* abhängig.
- Danach wird überprüft, ob der Nachrichtenpuffer versendet werden muss. In der Darstellung 4.24 geschieht dies nur für den aktuell verwendeten. Sofern mehrere existieren, ist es auch möglich, dass diese nacheinander überprüft werden.
- Abhängig von der verbleibenden Zeit bis der Nachrichtenpuffer versendet werden muss, wird entschieden, ob er an die Implementierung des Interface `Network` übergeben wird, oder der Kontrollfluss zu der Anwendung zurückkehrt.

Die dargestellte Ausführung der Kodierung und optionalen Versandes einer Nachricht stellt eine Möglichkeit dar. Abhängig von weiterer konfigurierten DDS-Funktionalität sind weitere Zwischenschritte notwendig. Bei der Nichtverwendung der LatencyBudget-QoS-Richtlinie fallen die letzten Schritte weg und die Nachricht wird sofort versendet.

4.5.8.2 Empfangen einer SNPS-Nachricht

Nach der Beschreibung der notwendigen Schritte für den Versand einer SNPS-Datennachricht werden nun die Verarbeitungsschritte für den Empfang dieser Nachricht auf der Seite der Datensenke beschrieben.

Die Abbildung 4.25 stellt den Ablauf der Schritte von dem Empfang einer Nachricht über die Dekodierung bis zur Bereitstellung des enthaltenden Datensatz innerhalb eines lokalen Topics als UML-Sequenzdiagramm dar. Wie auch schon bei der Darstellung (Abbildung 4.24) des Versands einer Nachricht in Abschnitt 4.5.8.1 ist die UML-Syntax vereinfacht und die Rückkehr der Methoden der Klasse `SNPS` ist nicht dargestellt.

Nachdem eine Nachricht von der Implementierung des `Network`-Interfaces empfangen wurde, wird deren Payload in einer Instanz der Klasse `NetBuffRef` gespeichert und diese über die implementierte Callback-Funktion `RecvHandler` der `DataSink`-Instanz übergeben. Die Verarbeitung der Nachricht geschieht

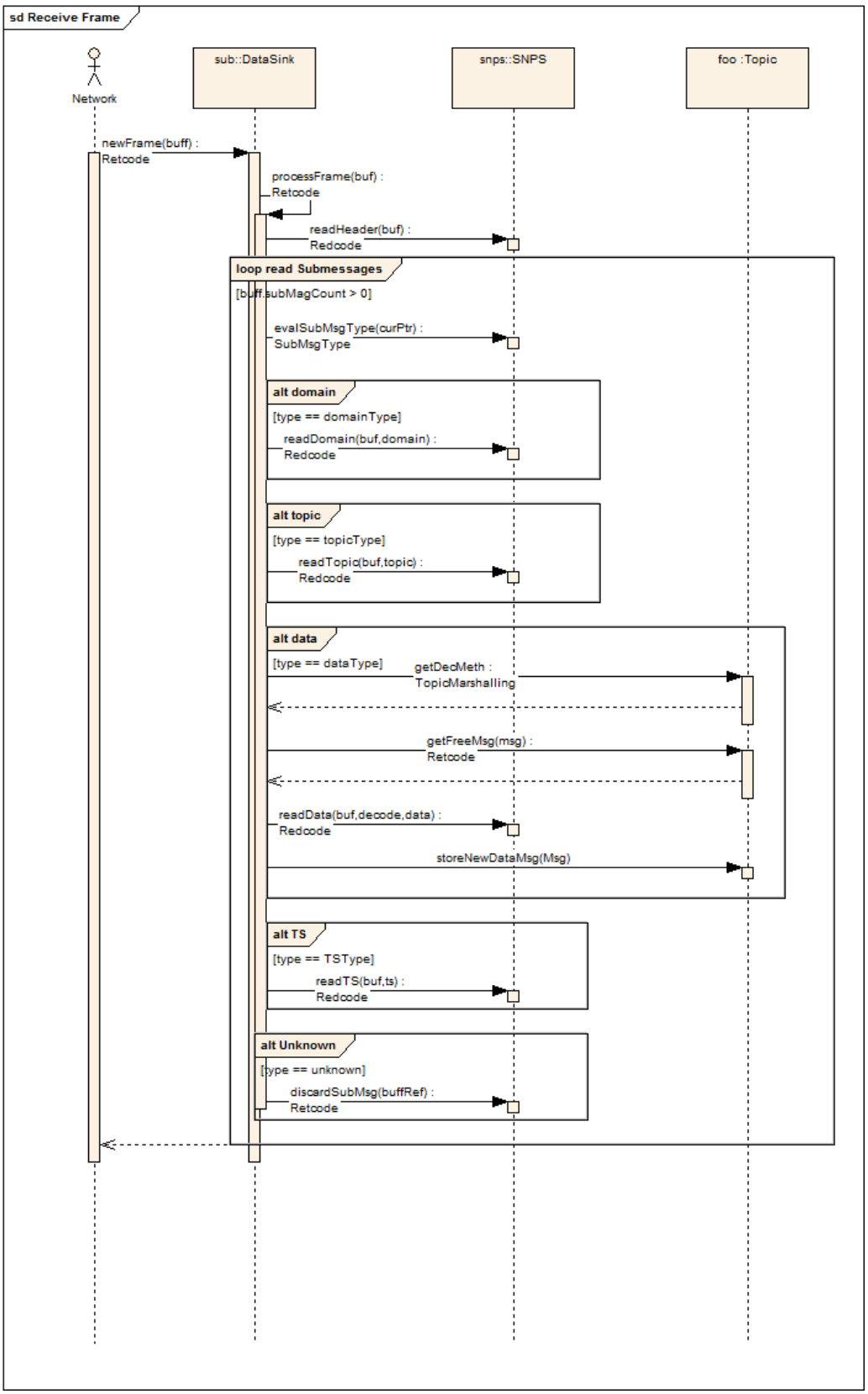

Abbildung 4.25: UML-Sequenzdiagramm für die Darstellung der Schritte für den Empfang und Dekodierung einer SNPS-Nachricht

in der `DataSink`-internen Methode `processFrame()`, um eine Trennung der Funktionalität von der Callback-Funktion zu erreichen.

Wie in Abschnitt 4.5.2.2 beschrieben, speichert die Instanz der Klasse `NetBuffRef` die notwendigen Zustandsinformationen für die Verarbeitung der Nachrichten. Nach der Überprüfung des Headers der Nachricht mittels der Methode `readHeader()` der SNPS-Protokollimplementierungsklasse `SNPS`, in welcher die Version des Protokolls der Nachricht mit der Version der Protokollimplementierung des Knotens abgeglichen wird, werden in einer Schleife die Submessages der Nachricht nacheinander verarbeitet. Für jeden Schleifendurchgang wird der Typ der Submessage ermittelt. Die Interpretation der jeweiligen Submessage ist von dem in Abschnitt 4.4.3 beschriebenen Kontext abhängig. Die folgende Liste beschreibt die entsprechenden Möglichkeiten für die in Diagramm 4.25 dargestellten Submessages:

- Wird eine Domain-Submessage gelesen, dann wird eine Referenz auf die Domain in der `NetBuffRef`-Instanz gesetzt. (Dieser Vorgang ist nicht in dem Diagramm dargestellt)
- Wird eine Topic-Submessage gelesen, dann wird eine Referenz auf das entsprechende Topic in der `NetBuffRef`-Instanz gesetzt. (Dieser Vorgang ist nicht in dem Diagramm dargestellt)
- Wird eine Daten-Submessage gelesen, dann wird abhängig von der Topic-Referenz in der `NetBuffRef`-Instanz die entsprechende Dekodierungsmethode von dem zugehörigen Topic mittels der Methode `getDecMeth()` geholt, sowie eine freie `Msg`-Instanz für die Speicherung des zu dekodierenden Datensatzes. Die `NetBuffRef`- und `Msg`-Instanz werden danach zusammen mit einer Referenz auf die Dekodierungsmethode der Methode `readData` der Klasse `SNPS` übergeben, die die Dekodierung des Datensatzes durchführt. Danach wird dem entsprechenden Topic mitgeteilt, dass die `Msg`-Instanz einen neuen Datensatz enthält, der ausgelesen werden kann.
- Sollten Submessage-Typen enthalten sein, die die sDDS-Implementierung nicht kennt, werden diese verworfen.

Sollten mehrere Datensätze in einer Nachricht übertragen worden sein, werden diese nacheinander dekodiert und als `Msg`-Instanz bei den zugehörigen Topics registriert. Nach der Verarbeitung der Nachricht kehrt die Ausführung zurück zu der Netzwerkimplementierung.

4.5.8.3 Lesen von Daten durch die Anwendung

Der vorherige Abschnitt hat den Vorgang für den Empfang und die Dekodierung einer SNPS-Nachricht beschrieben. Nun wird dargestellt, wie eine Anwendung einen solchen empfangenden neuen Datensatz lesen kann. Das UML-Sequenzdiagramm, das diesen Vorgang darstellt, ist in Abbildung 4.26 abgebildet.

Auch hier wird angenommen, dass es den Topic-Datentyp *Foo* gibt, für den die spezifischen DDS-Schnittstellen existieren. Die Anwendung fragt zunächst mittels der Methode `take_next_sample()` am spezifischen DataReader `FooDataReader` einen neuen Datensatz ab. Dieser Aufruf wird auf die generische Methode `takeNextData()` der Klasse `DataSink` weitergeleitet, die aus der Referenz auf den spezifischen DataReader das assoziierte Topic ermittelt.

Das Topic verwaltet die Instanzen der Klasse `Msg` und eine ungelesene Nachricht wird mit der Methode `getUnreadMsg()` abgerufen. Wie diese gefunden wird, ist abhängig von der Art, wie Nachrichten von der Topic-Implementierung verwaltet werden.

In Abschnitt 4.5.6.1 wurde die Funktionalität definiert, mit der eine Anwendung sich Speicher für den Datensatz von der Middleware leihen kann. Abhängig davon, ob die Anwendung Speicher für den Datensatz bereitstellt, wird der Datensatz in den bereitgestellten Speicher kopiert oder eine Referenz auf den Datensatz in der `Msg`-Instanz zurückgegeben. Im letzteren Fall wird das Attribut `isLoaned` auf *Wahr* gesetzt, damit die `Msg` nicht wiederverwendet werden kann, bis die Anwendung den Speicher freigibt.

Die Freigabe des Speichers eines Datensatzes geschieht mit der Methode `return_loan()` und ist in Abbildung 4.26 nach dem Lesevorgang dargestellt.

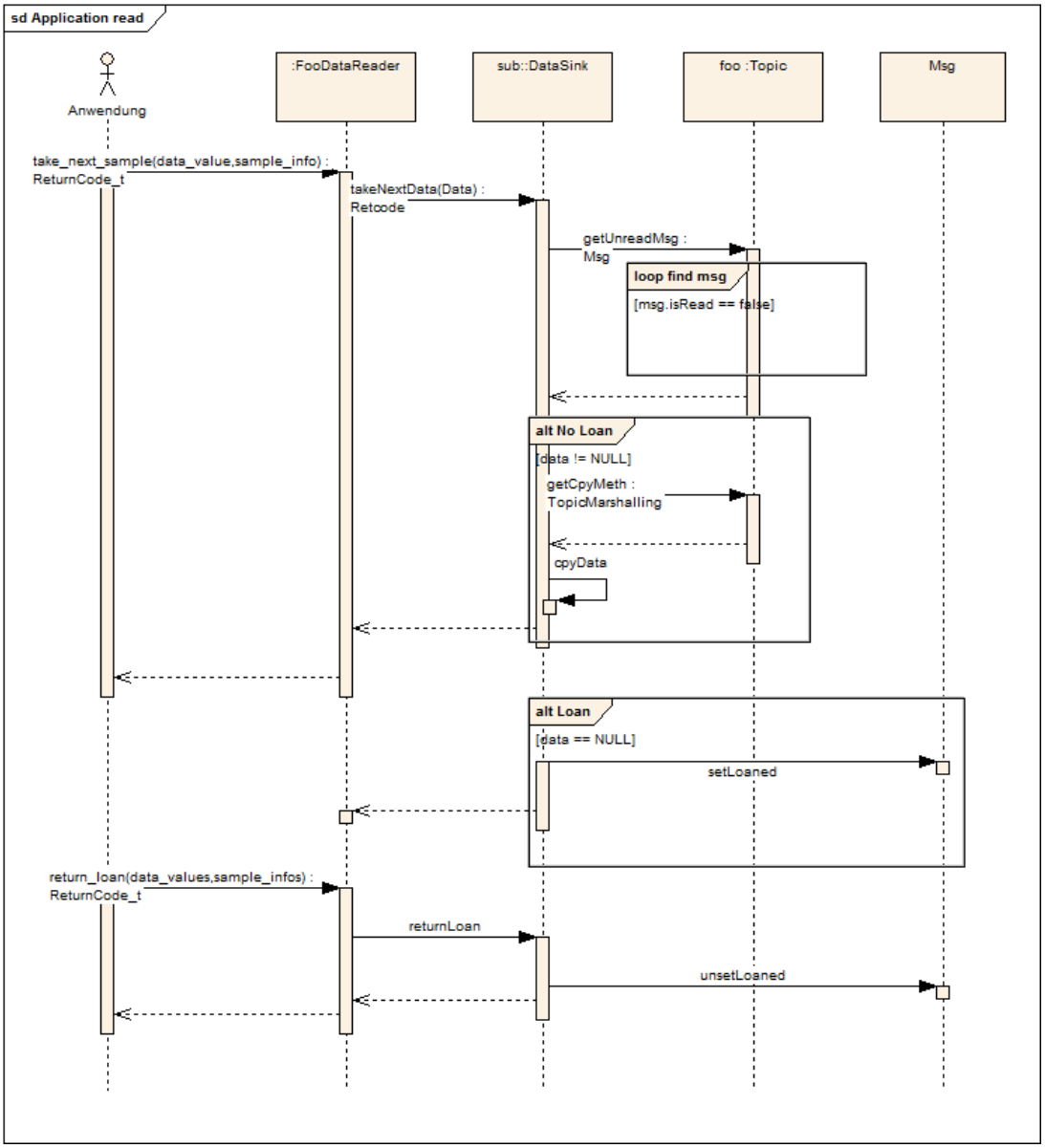

Abbildung 4.26: UML-Sequenzdiagramm für den Vorgang des Lesens eines Datensatzes durch eine Anwendung

Kapitel 5

Implementierung

5.1 Einführung

Das Ziel der Implementierung im Rahmen dieser Arbeit ist es auf prototypischer Ebene zu überprüfen, ob das in den vorherigen Kapiteln aufgestellte Konzept für eine DDS-basierte Middleware, die anwendungsspezifisch generiert wird, die aufgestellten Anforderungen erfüllen kann. Hierfür werden nicht alle Aspekte umgesetzt, die im Kapitel 3 analysiert bzw. gefordert oder im Kapitel 4 entworfen wurden. Aus zeitlichen Gründen wurde der Umfang der Implementierung auf die Teile beschränkt, die zwingend notwendig sind, um den in dieser Arbeit gewählten Ansatz bewerten zu können.

Die Anwendungsschnittstelle von sDDS soll der Schnittstelle entsprechen, wie sie im DDS-Standard spezifiziert wird. Der Standard definiert die Schnittstelle in der Syntax von OMG IDL. Für in OMG IDL spezifizierte Schnittstellen gibt es eine Reihe von standardisierten Abbildungen auf Programmiersprachen, wie ANSI-C, C++ und Java. Für die Implementierung von sDDS sollte daher eine Programmiersprache verwendet werden, für die es eine solche festgelegte Abbildung von OMG IDL gibt, damit sDDS seitens der Anwendungsschnittstelle kompatibel zu dem DDS-Standard ist und damit zu den Anwendungen, die darauf aufsetzen. Für drahtlose Sensorknoten ist damit die Verwendung von ANSI-C nahe liegend, da Java auf Grund der Ressourcenbeschränkungen keine Option ist und die Verfügbarkeit von C++-Compilern für die Plattform von Sensorknoten eingeschränkt ist. Mit der definierten Abbildung von OMG IDL auf ANSI-C ist auch ein Vorgehen für die Verwendung einer objektorientierten Struktur vorgegeben. Daher wird für

die prototypische Implementierung der sDDS-Middleware die Programmiersprache ANSI-C verwendet.

Die sDDS-Middleware-Implementierung soll das ZigBee-Protokoll als unterlagerte Kommunikationsschnittstelle verwenden. In Abschnitt 3.5.3.1 wurde die ZigBee-Implementierung zStack von TI auf der ebenfalls von TI hergestellten CC2430-Hardware-Plattform untersucht. Sowohl der zStack als auch die CC2430-Plattform werden für die Implementierung von sDDS verwendet. Des Weiteren kann der zStack nur mit der IAR-Entwicklungsumgebung vollständig eingesetzt werden. Daher wurde auch diese Software für die CC2430- und zStack-spezifischen Implementierungsarbeiten eingesetzt.

Auf Grund des prototypischen Charakters der Implementierung wurde von dem Aspekt, alle Komponenten der sDDS-Middleware anwendungsspezifisch zu generieren, Abstand genommen und entsprechend dem in Abschnitt 4.5 vorgestellten Entwurf von sDDS die Implementierung in drei Teile unterteilt:

- Ein universeller Teil, der für jede mögliche Zielplattform übersetzt werden kann
- Ein spezifischer Teil, der die Netzwerk- und Betriebssystemschnittstelle implementiert
- Ein anwendungsspezifischer Teil, der den Aufbau und die Konfiguration des sDDS-Systems durchführt

Im Rahmen der Generierung des Programmcodes wird primär der dritte Teil erzeugt, der auch spezifische Datenstrukturen enthält, die von den anderen Teilen referenziert werden. Die Auswahl der jeweiligen Komponente für die Implementierung der plattformspezifischen Funktionalität ist Aufgabe des MDSD-basierten Entwicklungsprozesses.

Die Implementierung erfolgte in drei Phasen: Zuerst wurde der generische Teil der sDDS-Middleware auf einem Linux-basierten Entwicklungssystem entwickelt und für diese Umgebung die plattformspezifischen Teile implementiert. Dementsprechend verwendet diese Version die POSIX-Schnittstelle für die Bereitstellung von Betriebssystemfunktionalität und UDP/IP für die Netzwerkimplementierung. In dieser Phase konnten die üblichen Werkzeuge zur Unterstützung der Software-Entwicklung wie Debugger und Profiler verwendet werden.

In der zweiten Phase wurde der plattformspezifische Teil neu für die Zielplattform implementiert, alle Teile mit der plattformspezifischen Entwicklungsumgebung übersetzt, auf die Zielsysteme ausgebracht und getestet.

Die dritte Phase umfasste die Entwicklung des sDDS-Middleware-Frameworks unter Verwendung der definierten Metamodelle. Basierend auf diesen wurden Editoren, Parser, Modelltransformationscode und die Programmcodegeneratoren entwickelt, bzw. mit den verwendeten Eclipse-Werkzeugen erzeugt. Der prototypische Programmcode der sDDS-Middleware aus den ersten beiden Phasen diente dabei als Basis für die Templates aus denen der anwendungsspezifische Teil der Middleware generiert wird.

Das Ergebnis der dritten Phase und damit der prototypischen Implementierung im Rahmen dieser Arbeit ist ein Werkzeug, das geeignet ist, grundsätzlich alle Schritte des Entwicklungsprozesses einer anwendungsspezifischen Middleware durchzuführen. Diese Schritte umfassen die Unterstützung der formalen Modellierung und Konfiguration der sDDS-Middleware für spezifische Anwendungen, die Überprüfung der Modelle und Konfigurationen auf definierte Bedingungen, einfache Modelltransformationen, die plattformspezifische Änderungen an dem Modellen durchführen und die Generierung von Programmcode für die sDDS-Middleware-Implementierung, deren enthaltene Funktionalität der Konfiguration entspricht.

Der Aufbau des weiteren Implementierungskapitels spiegelt diese drei Phasen wieder. Zuerst werden aber in Abschnitt 5.2 die Entwicklungsumgebung, die verwendeten Werkzeuge, die Zielplattform und ihre spezifischen Entwicklungsumgebung beschrieben sowie die generelle Struktur und der Styleguide des Programmcode. Danach werden in Abschnitt 5.3 interessante Aspekte der Implementierung des generischen Teils der sDDS-Middleware beschrieben. In Abschnitt 5.4 wird am Beispiel der Linux-basierten Entwicklungsplattform gezeigt, wie die plattformspezifischen Schnittstellen implementiert werden können. Die zweite Phase, die die sDDS-Middleware auf die Zielplattform bringt, wird im Abschnitt 5.5 beschrieben. Dabei wird primär auf die Besonderheiten der CC2430-Sensorknoten-Plattform eingegangen und die notwendige Implementierung der entsprechenden Schnittstellen von sDDS beschrieben. In Abschnitt 5.6.1 schließlich wird zuerst die Entwicklung des sDDS-Middleware-Frameworks beschrieben und danach in Abschnitt 5.6.4 wie der Programmcode der entwickelten sDDS-Prototypen in den Entwicklungs- und Generierungsprozess eingebunden wird.

5.2 Implementierungsumgebung

Für die prototypische Entwicklung der sDDS-Middleware und des Frameworks für die Modellierung und Generierung der spezifischen Middleware-Implementierung wurden verschiedene existierenden Hardware-Plattformen, Werkzeuge und Software-Komponenten eingesetzt. Auf deren Basis wird ebenfalls die Evaluierung der Einsatzfähigkeit des in dieser Arbeit erarbeitet Ansatzes durchgeführt. Allgemein wurden als Entwicklungplattformen Rechner mit Windows- und Linux-Betriebssystem verwendet. Das Versionskontrollsystem „Subversion" wurde für die Verwaltung des Programmcodes der Implementierung und für die Erstellung dieser Arbeit verwendet.

5.2.1 sDDS-Middleware-Framework

Das Middleware-Framework wurde mit den in Abschnitt 3.6.3 aufgeführten State-of-the-Art-Werkzeugen des Eclipse-Projektes aufgebaut. Die folgenden Werzeuge und deren Versionen wurden verwendet:

- Eclipse IDE Version Galileo 3.5.1
- Eclipse Version EMF 2.5
- Xpand Version 0.7.2
- Xtend Version 0.7.2
- Xtext Version 0.7.2

Die Entwicklung fand unter der Linux-Distribution Ubuntu 9.10 mit einem Kernel der Version 2.6.31 und einer AMD64-Architektur statt.

5.2.2 sDDS-Middleware

Die prototypische Implementierung der sDDS-Middleware verwendet ANSI-C als Programmiersprache im Funktionsumfang des Standards von 1999. Die primäre Entwicklung fand unter Linux mit dem Editor Vim in der Version 7.2 und dem C-Support-Plugin statt. Für die Entwicklung auf der Zielplattform des CC2430

wurde die „IAR Embedded Workbench IDE” in der Version 7.51 unter Windows XP verwendet.

Die Version des sDDS-Prototypen für Linux verwendet UDP/IP als unterlagertes Kommunikationsschnittstelle und wurde mit dem gcc in der Version 4.4.1 für die AMD64-Architektur übersetzt. Für verwendete Betriebssystemfunktionen wurde die POSIX-Schnittstelle verwendet. Programmfehler wurden mit dem gdb-Frontend DDD in der Version 3.3.11 und dem Proile-Werkzeug „valgrind” in der Version 3.5 untersucht.

Als Zielplattform für den sDDS-Prototypen wird die CC2430-Hardware-Plattform und die ZigBee-Implementierung zStack von TI verwendet. Für die Implementierung standen zwei ZMN2430-Module zur Verfügung, die einen CC2430 und weitere notwendige elektronische Bauteile enthalten, die notwendig mit überschaubaren Aufwand eigene Sensorknotenanwendungen zu entwickeln. Der Mikrocontrollerdes CC2430 hat eine 8051-Architektur und wurde in Abschnitt 3.3.2.1 bereits betrachtet. Die IAR-Entwicklungsumgebung umfasst eine Bestandteile, die notwendig sind, Anwendungen für diese Plattform zu entwickeln. Der Zugriff auf ein ZMN2430-Modul erfolgt über ein „SmartRF05 Evaluation Board” von TI, das über USB mit dem Entwicklungsrechner und die Debug-Schnittstelle des CC2430 mit den Sensorknoten verbunden ist. Der in die IAR-Entwicklungsumgebeung intigierte „Debugger” kann auf vier „Hardware-Breakpoints” der CC2430-Mikrocontrollerzugreifen. Damit ist es möglich Anwendungen direkt auf der Zielplattform schrittweise zu durchlaufen und zu debuggen.

5.2.3 Verwendeter OOP-Style Guide für ANSI-C

sDDS verwendet eine objektorientierte Architektur analog zu der des Designs in Abschnitt 4.5. Sie weicht an einigen Stellen vom Schema des objektorientierten Programmierparadigma ab, wenn dies für einen geringen Speicheerverbrauch notwendig ist.

Die Anwendungsschnittstelle von DDS wurde für ANSI-C entsprechend des OMG-Profils für die Abbildung von OMG IDL auf ANSI-C festgelegt. Hierbei wurde auf eine Kompatibilität zu der Schnittstelle von OpenSplice DDS in der Version 4.3 geachtet.

Jede Klasse wird mittels zweier Dateien implementiert: Die Header-Datei deklariert die Struktur und die Methoden der Klasse, und der Programmcode ist in einer

Quellcode-Datei definiert. Beide Dateien tragen dabei den Namen der Klasse. Die Deklaration der Struktur der Klasse bekommt das Postfix „`_type`", und ein `typedef` legt den Namen der Klasse als einen Zeiger auf diesen Typ der Struktur fest. Schnittstellen und abstrakte Klassen sind nur über eine Header-Datei deklariert; die Implementierung erfolgt in einer Quellcode-Datei mit dem Namen der implementierenden Klasse.

Für die Bennennung der Methoden einer Klasse gibt es die Konvention, dass der Funktionsname mit dem Namen der zugehörigen Klasse beginnt und von einem „_" gefolgt mit dem Methodennamen abgeschlossen wird. Der erste Parameter der Methode ist dabei immer ein Pointer auf die Instanz des Objektes, hier der Klassenstruktur. Dies entfällt zur Vermeidung von unnötigen Overhead bei Klassen, die nur einmal pro Laufzeitumgebung existieren. Bei diesen wird die Referenz auf die Instanz weggelassen und stattdessen die entsprechende Datenstruktur „*static*" im Geltungsbereich der Implementierungsdatei deklariert.

Alle Referenzen auf Instanzen von Klassen des DDS-Standards sind als *void*-Pointer ausgeführt, die in der Implementierung der Schnittstelle in den richtigen Typ umgewandelt werden.

5.3 Der universelle sDDS-Prototyp

5.3.1 Überblick über die sDDS-Komponenten

Der sDDS-Prototyp ist der erste Schritt für die Implementierung in dieser Arbeit. Der Entwurf in Kapitel 4.5 beschreibt eine Architektur, die aus zwei Teilen besteht: Die universellen Komponenten von sDDS, die ggf. anwendungsspezifisch angepasst werden und die plattformspezifischen Komponenten, die die Schnittstellen des Netzwerks und der Betriebssystemfunktionalität implementieren.

Die Komponenten von sDDS, die ausschließlich auf dem Sprachumfang von ANSI-C basieren, sind auf allen Plattformen, für die es einen C-Compiler gibt, übersetzbar und damit universell. Die Komponenten, die andere Bibliotheken oder Schnittstellen verwenden, weisen plattformspezifische Abhängigkeiten auf.

Die anwendungsspezifischen Anpassungen beziehen sich zum einem auf den Aufbau der DDS-Objekte und deren vorgesehene Anzahl im System, zum anderen auf die benötigten Schnittstellen des DDS-Standards und spezifische Funk-

tionalität von sDDS. Aus dem Aufbau der DDS-Struktur ergeben sich notwendige Konfigurationen für die anderen sDDS-Komponenten bezüglich der Menge der notwendigen Datenstrukturen. Zusammenfassend können die Komponenten von sDDS in die folgenden Gruppen aufgeteilt werden:

- Vollständig universelle Komponenten
- Universelle Komponenten, die anwendungsspezifischer Konfiguration unterliegen
- Plattformspezifische Komponenten
- Vollständig anwendungsspezifische Komponenten

Um die Entwicklung des sDDS-Middleware-Frameworks zu vereinfachen, wurde die Menge an zu generierenden Programmcodedateien minimiert. Für die universellen Komponenten, die anwendungsspezifischer Konfiguration unterliegen wurde daher auf Präprozessor-Techniken zurückgegriffen. Aus der anwendungsspezifischen Konfiguration lässt sich die benötigte Menge von Datenstrukturen oder die Notwendigkeit für das Einbinden von bestimmter Funktionalität ermitteln. Während der Generierung der sDDS-Middleware-Implementierung ist die Datei `CONSTANTS.h` zu erzeugen, die die Präprozessordefinitionen enthält, die die Konfiguration im universellen Teil des Programmcodes von sDDS durchführen. Die anschließende Anpassung geschieht mit den Mitteln des jeweiligen C-Compilers.

Die plattformspezifischen Komponenten implementieren die Interfaces `Network`, `Thread` und `Memory` (Siehe Abschnitt 4.5.2 und 4.5.7). Die universellen Komponenten bauen danach auf diesen Interfaces auf und sind damit unabhängig von der jeweiligen Implementierung. Die Auswahl der jeweiligen Implementierung geschieht über die Konfiguration der Middleware. Auch bei diesen Komponenten werden Präprozessordirektiven für eine genauere Konfiguration verwendet.

Die vollständig anwendungsspezifischen Komponenten müssen durch das Middleware-Framework erzeugt werden. Dies umfasst die Deklaration der Topic-Datentypen, die von der Anwendung benötigten DDS-Schnittstellen und ihre Abbildung auf die universellen sDDS-Konstrukte sowie den Aufbau der DDS-Struktur auf die die Anwendung über die DDS-Schnittstellen nach der Initialisierung zugreifen kann.

Wie in Abschnitt 3.6.2 festgelegt wurde, soll die Anwendung nicht gezwungen werden, das gesamte sDDS-System mit den Klassen-Instanzen selber zur Laufzeit aufbauen und konfigurieren zu müssen. Der optimierte Programmcode dafür wird aus der Modellierung des sDDS-Systems erzeugt und die Anwendung kann nach der Initialisierungsphase das sDDS-System direkt verwenden.

5.3.2 Implementierung des universellen Teils

Im Folgenden wird die Implementierung der universellen Klassen bzw. Komponenten von sDDS beschrieben. Sofern dabei anwendungsspezifische Anpassungen vorgesehen sind, werden diese extra aufgeführt. Dasselbe gilt für die Komponenten, die Teile der DDS-Schnittstelle implementieren oder emulieren.

5.3.2.1 Initialisierung

Einige Komponenten von sDDS benötigen eine Initialisierung, bevor sie verwendet werden können. Für diesen Zweck besitzen die entsprechenden Klassen `init()`-Methoden. Diese `init()`-Methoden sind in der Regel anwendungsspezifisch, da in ihnen, abhängig von der Konfiguration der sDDS-Middleware, Datenstrukturen erzeugt und initialisiert werden. Die Konfiguration der einzelnen Komponenten wird mittels Präprozessordirektiven durchgeführt.

Es gibt die globale Initialisierungsmethode `sDDS_init()`, deren Implementierung, auf der Modellierung der sDDS-Middleware basierend, generiert wird. Diese Methode muss von jeder Anwendung beim Systemstart aufgerufen werden. Danach können die erzeugen sDDS-Objekte über ebenfalls in der Generierung festgelegte globale Variablen zugegriffen werden. Ein Beispiel für eine generierte `sDDS_init()`-Methode ist im Listing 5.1 angegeben.

```
1  rc_t sDDS_init()
2  {
3      Memory_init();
4      LocatorDB_init();
5      Network_init();
6      DataSource_init();
7      DataSink_init();
8
9      fooTopic = (DDS_Topic) FooTopic_create();
10     Topic topic = (Topic) fooTopic;
11
```

```
12      fooDataWriter = (DDS_FooDataWriter)  DataSource_create_datawriter(topic, NULL, NULL,
            NULL);
13      fooDataReader = (DDS_FooDataReader) DataSink_create_datareader(topic, NULL, NULL,
            0);
14
15      return SDDS_RT_OK;
16  }
```

Quellcode 5.1: Beispiel für eine generierte `sDDS_init()`

CDR für vorzeichenlose als Binärzahl und für vorzeichenbehaftete im Zweierkomplement kodiert. Da dies auch die Repräsentierung auf den meisten Plattformen darstellt, ergibt sich kein zusätzlicher Konvertierungsaufwand.

Die Klasse `Marshalling` fasst alle Kodierungsmethoden zusammen. Es war hierfür abzuwägen zwischen der Möglichkeit, für jede Kodierung individuellen Programmcode zu generieren oder einer Folge von Methodenaufrufe dieser Klasse. Es wurde sich für letzteres entschieden, da zum einem der Programmcode ebenfalls von der Datenkodierung und der SNPS-Protokollimplementierung verwendet werden kann. Zum anderen bietet sich damit eine einfach konfigurierbare Modularisierung, da nur die Methoden eingebunden werden müssen, die auch verwendet werden. Um den modularen Charakter von sDDS hervorzuheben, ist da diese Lösung besser geeignet. Eine spätere Veränderung ist jederzeit möglich.

Intern werden die einzelnen Kodierungsmethoden auf zwei generische Methoden abgebildet: `encode()` und `decode()` kodieren bzw. dekodieren eine angegebene Menge an Bytes in einen Puffer bzw. aus diesem in ein Speicherbereich.

Neben den Methoden für die Kodierung der primitiven Datentypen gibt es zwei Methoden für die Kodierung bzw. Dekodierung von Submessage-Headern, wie sie in Abschnitt 4.4.2 definiert wurden. Der primäre Header aller Submessages besteht immer aus 8 Bit. 4 Bit definieren den Typ der Basic-Submessage und 4 Bit deren Payload oder den Typ der Extended-Submessage. Dementsprechend wird das Byte, das geschrieben wird, wie folgt zusammengesetzt:

```
1 uint8_t write = (type | (value << 4));
```

5.3.2.3 Hilfsklasse `NetBuffRef`

Die im Abschnitt 4.5.2.2 des Entwurfs vorgestellte Klasse `NetBuffRef` ist von großer Bedeutung in dieser sDDS-Implementierung. Sie kapselt eine SNPS-Nachricht mit ihrem Nachrichtenpuffer und den Zustandsinformationen für die Kodierung bzw. Dekodierung des SNPS-Protokolls. Im Entwurf wurde sie den Netzwerk-Komponenten zugeordnet, ist aber nicht plattformspezifisch, sondern universell einsetzbar, da der plattform- und anwendungsspezifische Teil der Nachrichtenpuffer ist, der über die Klasse `NetFrameBuff` gekapselt wird.

```
1  struct NetBuffRef_t{
2
3      // Menge der Submessages in der SNPS-Nachricht
4      uint8_t subMsgCount;
5
6      // Pointer auf den gekapselten Nachrichtenpuffer
7      NetFrameBuff frame_start;
8      // Pointer auf den Beginn des Nachrichtenpuffers
9      byte_t* buff_start;
10     // Menge der bisher geschriebenen / gelesenen Bytes
11     uint8_t curPos;
12
13     // QoS Zeitpunkt, zu dem die Nachricht versendet sein muss
14     pointInTime_t  sendDeadline;
15
16     // Adresse der diese Nachricht zugeordnet ist
17     struct Locator_t* addr;
18     // Topic des aktuellen Kontextes
19     Topic curTopic;
20     // Domain des aktuellen Kontextes
21     domainid_t curDomain;
22 };
23 typedef struct NetBuffRef_t* NetBuffRef;
```

Quellcode 5.2: Klasse `NetBuffRef`

Das Anlegen einer Instanz der Klasse `NetBuffRef` benötigt einen plattform- und anwendungsspezifischen Teil, daher wird die `init()`-Methode dieser Klasse von dem plattformspezifischen Teil von sDDS implementiert und in dieser die Instanz der Klasse `NetFrameBuff` erzeugt und der `NetBuffRef`-Instanz hinzugefügt.

Die SNPS-Kontexte für die Verarbeitung der SNPS-Nachrichten werden in der Klasse `NetBuffRef` zum einem über einen Pointer auf das Topic und zum anderen über die numerische Repräsentierung der Domain gespeichert.

5.3.2.4 SNPS-Protokoll

Wie im Entwurf Abschnitt 4.5.3 definiert, implementiert die Klasse `SNPS` die Methoden für die Erzeugung und Verarbeitung von SNPS-Submessages. Die Methoden operieren dabei auf der jeweiligen Instanz der Klasse `NetBuffRef`, die die Zustände für die Verarbeitung vorhält, so dass die Methoden der Klasse `SNPS` zustandslos sind.

Die Implementierung der Methoden ist von ihrem Aufbau bei allen ähnlich, so dass nur beispielhaft im Listing 5.3 die Implementierung der Methode zum Kodieren eines Topics dargestellt wird.

```
1  #define START (ref->buff_start + ref->curPos)
2
3  rc_t SNPS_writeTopic(NetBuffRef ref, topicid_t topic)
4  {
5      Marshalling_enc_SubMsg(START, SNPS_SUBMSG_TOPIC, (uint8_t)topic);
6      ref->curPos +=1;
7      ref->subMsgCount +=1;
8
9      return SDDS_RT_OK;
10 }
```

Quellcode 5.3: Methode `writeTopic()` der Klasse `SNPS`

Der Pointer auf die aktuelle Position im Nachrichtenpuffer wird durch das Makro `START` berechnet und zusammen mit dem Inhalt der Methode zum Schreiben einer Basic-Submessage übergeben. Danach wird in der `NetBuffRef`-Instanz die aktuelle Position und die Anzahl der enthalten Submessages aktualisiert.

Wie auch in der Implementierung der Klasse `Marshalling` findet hier keine Überprüfung der verbleibenden Bytes im Nachrichtenpuffer statt. Die Überprüfung, ob ein Datensatz zuzüglich weiterer Informationen in einen Puffer passt, wird auf höherer Ebene vorgenommen. Um die Menge des Programmcodes minimal zu halten, wird davon abgesehen, auf jeder Ebene diese Art von Überprüfungen vorzunehmen.

5.3.2.5 Topic Management

Die Implementierung der Klasse `Topic` ist insofern interessant, als dass hier die Möglichkeiten für eine anwendungsspezifische Konfiguration direkt gezeigt werden kann. Der Entwurf in Abschnitt 4.5.4 zeigt die Klasse und ihre Assoziationen zu anderen Komponenten. Dazu gehören auch Adressen von anderen Knoten, die DataReader oder DataWriter für das jeweilige Topic haben, oder die Kodierungs- und Dekodierungsmethoden für den Topic-spezifischen Datentyp. Besteht die Anwendung auf einem Knoten nur aus einem DataWriter für ein Topic, dann muss die Instanz des Topics nur die Adressen für die Knoten mit DataReadern kennen und die Kodierungsmethode für den Datentyp besitzen. Das Listing 5.4 zeigt die Konfiguration am Beispiel der Deklaration der Klassenstruktur `Topic`.

```
1
2  #ifdef sDDS_TOPIC_HAS_PUB
3  struct MsgPool{
4      struct Msg_t pool[sDDS_TOPIC_APP2_MSG_COUNT];
5  };
6
7  struct datasources{
8      Locator list;
9      uint8_t count;
10 };
11 #endif
12
13 #ifdef sDDS_TOPIC_HAS_SUB
14 struct datasinks{
15     Locator list;
16     uint8_t count;
17 };
18 #endif
19
20 struct Topic_t {
21
22 #ifdef sDDS_TOPIC_HAS_SUB
23     struct datasinks dsinks;
24     rc_t (*Data_encode)(byte_t* buff, Data data, size_t* size);
25 #endif
26 #ifdef sDDS_TOPIC_HAS_PUB
27     struct datasources dsources;
28     struct MsgPool msg;
29     rc_t (*Data_decode)(byte_t* buff, Data data, size_t* size);
30     rc_t (*Data_cpy)(Data dest, Data source);
31 #endif
32
33     domainid_t domain;
34     topicid_t id;
35 };
36 typedef struct Topic_t* Topic;
```

Quellcode 5.4: Deklaration der Klassenstruktur `Topic` mit enthaltener Konfiguration über Präprozessordirektiven

Die Präprozessorkonstante `sDDS_TOPIC_HAS_SUB` wird definiert, wenn auf anderen Knoten im Sensornetz DataReader existieren, die Daten dieses Topics benötigen. Analog wird mit `sDDS_TOPIC_HAS_PUB` konfiguriert, dass es im Sensornetz Publisher, also DataWriter für dieses Topic gibt, deren Daten lokal empfangen werden sollen.

Das bedeutet, dass, wenn auf einem Knoten nur ein DataWriter existiert, die Funktionalität zum Empfang und der Verarbeitung von von ihm verwendeten Datentypen nicht benötigt wird.

Obwohl diese Konfiguration anwendungsspezifisch ist, muss der Programmcode der Klasse `Topic` nicht generiert werden. Es reicht, wenn er zusammen mit der generierten Deklaration der Konfigurationseinstellungen in der Header-Datei `CONSTANTS.h` übersetzt wird.

Ein weitere anwendungsspezifische Komponente ist die von `Topic` referenzierte Klasse `Msg`. Die Klasse selber ist vollständig unabhängig, aber sie referenziert einen Speicherbereich, um den Wert des jeweiligen Topic-Datentyp vorzuhalten. Die Allokation des Speicherbereichs für den Wert des Datentyps und seine Zuweisung an eine `Msg`-Instanz geschieht durch den generierten Programmcode zum Aufbau und zur Initialisierung des sDDS-Systems.

5.3.2.6 Daten senden

Das Versenden von Daten ist Aufgabe der Klasse `DataSource`, die in Abschnitt 4.5.5 definiert wurde. Das Vorgehen für die Kodierung der SNPS-Nachricht entspricht dem in Abschnitt 4.5.8.1 dargestellten Ablauf und wird daher nicht weiter betrachtet. Interessant ist die Frage, wie die Abbildung auf die DDS-Schnittstelle geschieht und wie der richte Nachrichtenpuffer für die verzögerte Nachrichtenübertragung gefunden wird.

Die DDS-Schnittstelle für das Veröffentlichen ist die `write()`-Methode der Topic-Datentyp-spezifischen Ableitung der DDS-Klasse `DataWriter`. In Listing 5.5 ist aus der datentypspezifisch generierten Header-Datei die Deklaration der `write()`-Methode dargestellt und danach die ebenfalls generierte Implementierung, die diese Methode auf die universelle Implementierung der Klasse `DataSource` abbildet.

```
1
2
3  // Foo.h
4  DDS_ReturnCode_t DDS_FooDataWriter_write(
5      DDS_DataWriter _this,
6      const Foo* instance_data,
7      const DDS_InstanceHandle_t handle);
8
9  // generierte Programmcodedatei
10
11 DDS_ReturnCode_t DDS_FooDataWriter_write(
12     DDS_DataWriter _this,
13     const Foo* instance_data,
14     const DDS_InstanceHandle_t handle)
```

```
15 {
16     rc_t rc;
17     rc = DataSource_write((DataWriter) _this, (Data) instance_data, (void*) handle);
18     if (rc == SDDS_RT_OK) {
19         return DDS_RETCODE_OK;
20     } else {
21         return DDS_RETCODE_ERROR;
22     }
23
24 }
```

Quellcode 5.5: Schnittstelle für das Versenden von Daten

Diese Abbildung innerhalb der Implementierung durchzuführen, ist nicht optimal; die Verwendung von Präprozessor-Macros für eine direkte Substitution der Signaturbestandteile der Methoden wäre leichtgewichtiger. Der gewählte Weg ist aber anschaulicher und ergibt einen besseren Test-Zugang.

Die Instanz des `DataWriter` wird in der generierten Initialisierung festgelegt und ist eingebettet in die Instanz der `DataSource`, so dass sie über den dort gespeicherten Pointer auf die assoziierte `Topic`-Instanz und optionale QoS-Parameter zugreifen kann.

Die hier vorgestellte Implementierung verwendet den in Abschnitt 4.5.5.2 vorgestellten `InstantSender` und verschickt die Daten sofort. Die Option, den Versand verzögern zu können, wurde noch nicht umgesetzt. Der Vorgang des Auswählens eines geeigneten Nachrichtenpuffers wurde implementiert und soll nun vorgestellt werden. Das Listing 5.6 gibt den relevanten Ausschnitt aus der `write()`-Methode der Klasse `DataSource` wieder.

```
1
2 rc_t DataSource_write(DataWriter _this, Data data, void* waste)
3 {
4
5     // Pointer auf den zu verwendenden Nachrichtenpuffer
6     NetBuffRef buffRef = NULL;
7
8     // Liste mit Zieladressen fuer den Datensatz
9     Locator dest = _this->curTopic->dsinks.list;
10
11    // Suche nach einem Nachrichtenpuffer mit derselben Adresse
12    // wie die aktuelle Zieladresse
13    for (int i = 0; i < sDDS_NET_MAX_OUT_QUEUE; i++){
14        Locator try = dataSource->sender.out[i].addr;
15        if (dest != NULL && try != NULL && Locator_isEqual(dest, try)){
16            buffRef = &(dataSource->sender.out[i]);
17            break;
18        }
```

```
19      }
20      // Sollte kein passender Nachrichtenpuffer gefunden worden sein,
21      // dann waehle einen freien Puffer aus
22      if (buffRef == NULL){
23          for (int i = 0; i < sDDS_NET_MAX_OUT_QUEUE; i++){
24              if (dataSource->sender.out[i].curPos == 0){
25                  buffRef = &(dataSource->sender.out[i]);
26                  break;
27              }
28          }
29      }
30      // Sollte kein freier Nachrichtenpuffer existieren, dann
31      // nehme den fuer hochpriore Daten
32      if (buffRef == NULL){
33          buffRef = &(dataSource->sender.highPrio);
34      }
35
36      ...
37  }
```

Quellcode 5.6: Auswahl eines passenden Nachrichtenpuffers für die Datenübertragung in der Klasse `DataSource`.

Der gezeigte Programmcode wurde auf die relevanten Teile für das Verständnis der weiteren Beschreibung reduziert. Der `_this`-Pointer zeigt auf die Datenstruktur der sDDS-Klasse `DataWriter`, die den Pointer auf das `Topic` enthält. Die `Topic`-Instanz hält die Liste mit den gekapselten Adressen für die Empfänger des zu versenden Datensatzes. Da die Instanz der Klasse `DataSource` nur einmal existiert, gibt es den globalen Pointer `dataSource`, um auf dessen Datenstruktur zuzugreifen unddie vorhandenen in `NetBuffRef`-Instanzen gekapselten Nachrichtenpuffer zu durchlaufen. Die erste Schleife überprüft, ob eine `NetBuffRef`-Instanz an dieselbe Adresse gerichtet ist; die zweite Schleife sucht für den Fall, dass es keine solche `NetBuffRef`-Instanz gibt, einen freien Nachrichtenpuffer.

Für den Fall, dass es keinen freien Nachrichtenpuffer bzw. keinen für denselben Empfänger gibt, wird als Ausweg der hochpriore Nachrichtenpuffer verwendet. Hier ist zu beachten, dass dies dazu führen kann, dass ein niederpriorer Datensatz sofort versendet wird und dies einen hochprioren, der danach auftreten kann, verzögern würde. Sollte dies für eine Anwendung ein Problem darstellen, dann ist bei der Konfiguration der Middleware darauf zu achten, dass genügend Nachrichtenpuffer bereitstehen und die Wahrscheinlichkeit für diesen Fall auf ein akzeptables Maß gesenkt wird.

5.3.2.7 Daten empfangen

Der Ablauf des Empfangs von Daten durch die Klasse `DataSink` wurde in Abschnitt 4.5.8.2 konzeptionell beschrieben. Die Implementierung der DDS-Schnittstelle wurde ähnlich wie bei der `DataSource` durchgeführt und wird daher hier nicht weiter beschrieben. Die Klasse `DataSource` setzt auf dem in Abschnitt 4.5.2 beschrieben Interface `Network` auf und bekommt von einer registrierten Callback-Methode neue Nachrichten zugestellt.

Die Verarbeitung geschieht in der Methode `processFrame()` (Siehe Abschnitt 4.5.6.2); für die Implementierung ist dabei der Umgang mit den Kontexten der SNPS-Nachrichten von Interesse und soll im Folgenden kurz vorgestellt werden. Das Listing 5.7 zeigt den auf das Wesentliche reduzierten Rumpf der Methode `prozessFrame`.

```
1  rc_t DataSink_processFrame(NetBuffRef buff)
2  {
3
4      // Ueberpruefe den SNPS-Header
5      if (SNPS_readHeader(buff) != SDDS_RT_OK){ ... }
6
7      // globale Variable fuer die Msg-Instanz
8      Msg msg = NULL;
9
10     while (buff->subMsgCount > 0)
11     {
12         subMsg_t type;
13         // Lese den Typ der Submessage
14         SNPS_evalSubMsg(buff, &type);
15
16         switch (type){
17             case (SNPS_T_DOMAIN) :
18                 // Kontextwechsel -> Daten speichern
19                 submitData(&msg);
20                 domainid_t domain;
21                 SNPS_readDomain(buff, &domain);
22                 // Ueberpruefe Relevanz der Domain
23                 // Ueberspringt den Kontext ggf.
24                 checkDomain(buff, domain);
25                 break;
26             case (SNPS_T_TOPIC) :
27                 submitData(&msg);
28                 topicid_t topic;
29                 SNPS_readTopic(buff, &topic);
30                 // Ueberpruefe Relevant des Topics
31                 // Ueberspringt den Kontext ggf.
32                 checkTopic(buff, topic);
33                 break;
34             case (SNPS_T_DATA) :
```

```
35                  submitData(&msg);
36                  // Lesen der Daten und Speichern
37                  // in der globalen msg-Variable
38                  parseData(buff, &msg);
39                  break;
40          ...
41              default:
42              // Unbekannte Submessage-Typ ueberspringen
43                  SNPS_discardSubMsg(buff);
44          }
45
46      }
47      // Ende der SNPS-Nachricht
48      // Verbliebene Daten speichern
49      submitData(&msg);
50
51      ...
52  }
```

Quellcode 5.7: Rumpf der Methode `processFrame` der Klasse `DataSink` für die Verarbeitung von SNPS-Nachrichten

Der Ablauf ist grundsätzlich wie in Abschnitt 4.5.8.2 des Entwurfes vorgestellt. Bei jeder SNPS-Nachricht wird zuerst der Header überprüft und bei Inkompatibilität die Nachricht verworfen. Danach werden die Submessages ausgewertet. Hierfür werden verschiedene Hilfsfunktionen eingesetzt, um den Programmcode übersichtlich zu halten.

Das Ziel bei dieser Implementierung war es, die Menge an Zustandsinformationen klein und die Verarbeitung einer SNPS-Nachricht sequenziell zu halten. Die Kontexte im SNPS-Protokoll werden durch die Topic- und Domain-Variablen der `NetBuffRef`-Instanz und die `msg`-Variable festgehalten. Das Problem bei der Implementierung ist es, bei der Änderung von Kontexten die vorherigen Kontexte richtig abzuschließen. Dies trifft hier auf den SNPS-Kontext für Daten-Submessages zu. Ein Daten-Kontext wird mit einer Daten-Submessage geöffnet, die mit der Hilfsfunktion `parseData()` verarbeitet wird. In dieser Funktion werden die Daten in eine neue `Msg`-Instanz dekodiert und diese der `msg`-Variable zugewiesen. Solange die `msg`-Variable etwas referenziert, ist der Daten-Kontext offen und weitere Submessages, wie beispielsweise Zeitstempel o.ä., könnten dieser Nachricht beigefügt werden. Mit dem Auftreten einer Domain-, Topic, oder neuen Daten-Submessage wird der Kontext geschlossen, die `Msg`-Instanz mit der Hilfsfunktion `submitData()` dem assoziierten Topics übergeben und die Referenz der `msg`-Variable auf `NULL` gesetzt.

Die Verarbeitung von SNPS-Nachrichten basierend auf Funktionalität zukünftiger

Erweiterungen wird in einem ähnlichen Muster ablaufen. Es wird ggf. notwendig sein, weitere mögliche Kontexte zu berücksichtigen.

5.4 sDDS-Linux-Prototyp

Bisher wurde der Teil der sDDS-Implementierung beschrieben, der universell mit einem C-Compiler für verschiedene Plattformen übersetzt werden kann. Nun wird der Teil beschrieben, der plattformspezifische Funktionalität kapselt bzw. implementiert. Als Plattform wurde die in Abschnitt 5.2 beschriebene Linux-Entwicklungsumgebung verwendet. Nach dem Entwurf in Abschnitt 4.5 ist dafür das Interface `Network` sowie die Interfaces des OS-SSAL zuständig. Es werden im Folgenden die Implementierung der Netzwerk-Schnittstelle und des `Memory`-Interfaces des OS-SSAL genauer beschrieben.

5.4.1 UDP-Implementierung des Interfaces `Network`

Der Linux-Prototyp von sDDS verwendet die POSIX-Schnittstellen und UDP/IP als unterlagertes Kommunikationssystem. Wie in Abschnitt 4.5.2.2 dargestellt, gibt es für UDP die Ableitung des Interfaces `Network` sowie UDP-spezifische Kapselungen von Adressen.

```
1  struct UDPLocator_t{
2      struct Locator_t loc;
3      struct sockaddr_in addr;
4  };
5
6  bool_t Locator_isEqual(Locator l1, Locator l2)
7  {
8      struct UDPLocator_t* a = (struct UDPLocator_t*) l1;
9      struct UDPLocator_t* b = (struct UDPLocator_t*) l2;
10     if (memcmp(&(a->addr.sin_addr.s_addr), &(b->addr.sin_addr.s_addr), 4) == 0){
11         return true;
12     } else {
13         return false;
14     }
15 }
```

Quellcode 5.8: Ausschnitt für die Implementierung der gekapselten IP-Adressen

Das Listing 5.8 zeigt den interessanten Teil der gekapselten Netzwerkadressen. Der generische Teil ist in der in Abschnitt 4.5.2.2 definierte Klasse `Locator`

enthalten. Für die UDP-spezifische Adresse wurde das nur lokal sichtbare C-struct `UDPLocator` deklariert, das die `Locator`-Struktur und die IP-Adressenstruktur `struct sockaddr_in` enthält. Über eine Typumwandlung des Pointers auf den `Locator`-Typ kann der universelle Teil von sDDS auf die `Locator`-spezifischen Teile der Struktur zugreifen. Das UDP-Netzwerkmodul hat selber Zugriff auf die plattformspezifische Adresse in einer `Locator`-Instanz. Dies wird in der dargestellten Implementierung der Methode `isEqual()` der Klasse `Locator` gezeigt, um die gekapselten IP-Adressen auf Gleichheit zu überprüfen.

Die Implementierung des `Network`-Interfaces soll der Programmcodeausschnitt im Listing 5.9 illustrieren.

```
1  struct Network_t {
2      int fd_uni_socket;
3      int port;
4      pthread_t recvThread;
5      rc_t (*newFrame)(NetBuffRef buff);
6  };
7
8  static struct Network_t net;
9  static struct NetBuffRef_t inBuff;
10
11 rc_t Network_init()
12 {
13     ...
14
15     net.fd_uni_socket = socket(AF_INET, SOCK_DGRAM, 0);
16     ...
17
18     rc = bind(net.fd_uni_socket, (struct sockaddr*)&addr, sizeof(addr));
19
20     // Initialisierung des Eingangsnachrichtenpuffers
21     NetBuffRef_init(&inBuff);
22
23     if (pthread_create(&net.recvThread, NULL,recvLoop, (void*) &inBuff) != 0){ ... }
24     ...
25 }
26
27 void* recvLoop(void* netBuff)
28 {
29     ...
30     while (true){
31
32         // Saeubern des alten Nachrichtenpuffers
33         NetBuffRef_renew(buff);
34         ...
35         // Empfange neue UDP-Nachricht
```

```
36          ssize_t recv_size = recvfrom(net.fd_uni_socket, buff->buff_start,
                buff->frame_start->size, 0,
37                  (struct sockaddr*) &(addr), &addrFieldLen);
38
39          // Senderadresse extrahieren
40          struct UDPLocator_t sloc;
41          memcpy(&(sloc.addr), &addr, sizeof(struct sockaddr_in));
42
43          // Ueberpruefung, ob Sender bekannt ist
44          Locator loc;
45          if (LocatorDB_findLocator((Locator)&sloc, &loc) != SDDS_RT_OK){
46              // Bisher unbekannter Sender -> Neuen Locator anlegen
47              if (LocatorDB_newLocator(&loc) != SDDS_RT_OK){ ... }
48              memcpy(&(((struct UDPLocator_t*)loc)->addr), &addr, sizeof(struct
                    sockaddr_in));
49          }
50          // Referenzzaehler erhoehen
51          Locator_upRef(loc);
52          loc->isEmpty = false;
53          loc->isSender = true;
54          // Absenderadresse der SNPS-Nachricht anfuegen
55          inBuff.addr = loc;
56          // Nachricht zur Verarbeitung an Callback-Funktion weiterreichen
57          (*(net->newFrame))(buff);
58          Locator_downRef(loc);
59
60      }
61
62      return SDDS_RT_OK;
63  }
```

Quellcode 5.9: Ausschnitt aus der Implementierung des `Network`-Interfaces

Die Netzwerkimplementierung besitzt eine eigene private Klassenstruktur, die alle für den Betrieb relevanten Informationen vorhält. Dies sind bei der UDP/IP-Version für Linux: der Socket, der Port, der Thread für den Empfang von Nachrichten und der Funktions-Pointer für die Callback-Funktion der Klasse `DataSink`. In dieser prototypischen Implementierung von sDDS ist es vorgesehen, dass immer nur ein Typ von Netzwerk verwendet werden kann. Daher gibt es auch nur eine `Network`-Instanz pro Knoten, und die Klassenstruktur ist statisch definiert und in der Implementierung global zugreifbar. Entsprechendes gilt für den Nachrichtenpuffer (`NetBuffRef`) für den Empfang von Nachrichten.

Wie in Abschnitt 5.3.2.1 dargestellt wurde, haben sDDS-Komponenten in der Regel eine Initialisierungsmethode. Die `init()`-Methode für die UDP/IP-Komponente legt unter anderem den Socket an, führt das Binding mit dem gewählten Port aus, initialisiert den Eingangsnachrichtenpuffer und legt einen POSIX-Thread an, der die Aufgabe hat, UDP-Nachrichten zu empfangen. Da diese UDP-

Implementierung des `Network`-Interfaces bereits plattformspezifisch für Linux ist, wurde für die Erzeugung des Threads nicht die OS-SSAL-Schnittstelle verwendet.

Der Empfangs-Thread hat die Aufgabe, neue UDP-Nachrichten zu empfangen und die Adresse des Senders zu extrahieren. In Abschnitt 4.5.4.3 des Entwurfs wurde festgelegt, dass die sDDS-Knoten Informationen über das Middleware-Netz sammeln und lokal vorhalten, sofern Speicherplatz verfügbar ist. Des Weiteren sollen Datenstrukturen nur einmal angelegt und danach referenziert werden, um Speicherplatz zu sparen. Hier wird dies am Beispiel der Knotenadressen durchgeführt. Als erstes wird überprüft, ob die Adresse bereits bekannt ist. Die Klasse `LocatorDB` referenziert alle gekapselten Adressen auf einem Knoten und wird dementsprechend befragt. Sollte die Adresse unbekannt sein, wird sie neu angelegt und mit der empfangenen SNPS-Nachricht assoziiert. Da die `Locator`-Instanzen von mehreren sDDS-Komponenten verwendet werden können, verfügen sie über einen Referenzzähler.

Die `DataSink`-Instanz des Knotens wird über den Aufruf der registrierten Callback-Funktion die neu empfangene Nachricht zur Verarbeitung übergeben. Nach Rückkehr der Funktion wird der Referenzzähler der Adresse wieder dekrementiert, damit die Datenstruktur ggf. wiederverwendet werden kann.

5.4.2 Übersetzungsprozess

Der Linux-Prototyp der sDDS-Middleware wurde in der ersten Phase der Implementierung entwickelt. Dementsprechend gab es zu diesem Zeitpunkt noch kein sDDS-Middleware-Framework für die Generierung des Programmcodes. Die bisher aufgeführten „generierten Teile” wurden händisch entworfen, um später als Basis für die Erstellung der Programmcode-Templates zu dienen.

Der Prozess für die Übersetzung der Linux-Version von sDDS entspricht dem üblichen Vorgehen für Software-Entwicklung im Unix-Umfeld. Der Programmcode ist in einer Ordnerstruktur eingeordnet, die zwischen Header- und Implementierungsdateien unterscheidet und die verwendeten Namensräume widerspiegelt. Ein „Makefile” enthält die Anweisungen für den Compiler und Linker, um aus den sDDS-Komponenten, der Konfiguration in der Datei `CONSTANTS.h` und dem Programmcode der Anwendung ein Binärpaket mit dem Maschinencode zu erzeugen, das alle Teile enthält.

Bei der Integration der Linux-Version in das Middleware-Framework wurde der Übersetzungsprozess mittels eines Makefiles aus der Generierung ausgenommen. Das bedeutet, dass die generierten Dateien mit dem universellen Teil manuell übersetzt werden müssen, oder ein Makefile manuell anzupassen ist. Das sDDS-Middleware-Framework beschränkt sich zurzeit auf die Generierung des anwendungsspezifischen sDDS-Programmcodes. Plattformspezifische Übersetzungsprozesse zu unterstützen, ist ein Punkt für zukünftige Erweiterungen.

5.5 sDDS-CC2430-Prototyp

5.5.1 Einführung

Diese Arbeit verwendet die SoC-Hardware-Plattform CC2430 von TI als Zielplattform für den sDDS-Prototypen. Diese wurde in der Analyse Abschnitt 3.3.2.1 das erste Mal als typische Hardware-Plattform für Sensorknoten beschrieben. Als ZigBee-Implementierung wird der in Abschnitt 3.5.3.1 untersuchte zStack verwendet, der von TI für die CC2430-Plattform in der ZigBee-Version 2006 bereitgestellt wird. In Abschnitt 5.2 wurde kurz das Entwicklungssystem „IAR Embedded Workbench IDE" vorgestellt, das für die Erzeugung von Programmcode für diese Plattform verwendet werden muss, da nur hierfür alle zStack-Bibliotheken verfügbar sind.

Die Arbeit für die Implementierung des CC2430-Prototyps von sDDS umfasste die Implementierung der plattformspezifischen Schnittstellen von sDDS und die Einbettung des sDDS-Programmcodes in ein Projekt der IAR-Entwicklungsumgebung und den damit verbundenen Übersetzungsprozess. In Abschnitt 3.5.3.2 der Analyse von ZigBee wurde festgestellt, dass es notwendig sein wird, die ZigBee-Funktionalität auf ein Minimum zu reduzieren, damit sDDS-Middleware, Anwendung und ZigBee-Stack in den Programmspeicher des Zielsystems passen. Diese Anpassung von zStack ist ebenfalls Bestandteil dieser Implementierungsarbeit.

5.5.2 ZigBee-Anpassung

Der zStack umfasst nicht nur die reine ZigBee-Protokoll-Implementierung, sondern auch eine Umgebung, in die Anwendungen eingebettet werden und auf Be-

triebssystemfunktionalität zugreifen können. Dies vereinfacht die Entwicklung von Anwendungen für ZigBee ungemein, da damit primär nur die reine Anwendungslogik erstellt werden muss. Des Weiteren stellt TI eine Reihe von Beispielprojekten mit dem zStack bereit, die verwendet werden können, um davon eigene Anwendungen abzuleiten. Für die sDDS-Implementierung wurde dieser Weg gewählt und das Beispielprojekt „GenericApp" als Grundlage verwendet.

5.5.2.1 Reduktion des ZigBee-Speicherverbrauchs

Die Tabelle 3.6 auf Seite 91 zeigt unter anderem den Speicherverbrauch des GenericApp-Beispiels in der vorgegebenen Standardkonfiguration. Der CC2430 verfügt über 128 KiB Flash-Speicher, und das Beispielprojekt benötigt davon bereits mehr als 100 KiB. Die Reduktion des Speicherverbrauchs von ZigBee war daher der erste Arbeitsschritt.

Das GenericApp-Projekt sieht eine Reihe von Präprozessordirektiven vor, mit denen die verwendete ZigBee-Funktionalität konfiguriert werden kann. Keine dieser Funktionalitäten wird für sDDS benötigt, so dass über diese Konfigurationsschnittstelle eine erste Reduktion des Speicherverbrauchs möglich war.

Da das GenericApp-Projekt ein Beispielprojekt für ein bestimmtes Evaluationsboard des CC2430 ist, sind eine Reihe von Hardware-Treibern und weitere Programmcode zur Verwendung dieser Funktionalität enthalten. Um zwischen Geräten das Binding von ZigBee-Cluster zu initiieren werden Hardware-Taster verwendet. Für diese Arbeit wird die Unterstützung dieser Funktionalität nicht benötigt und wurde daher manuell aus dem Programmcode des Beispielprojekts entfernt.

Zur weiteren Reduktion des Speicherverbrauchs wurden manuell im Programmcode die ZigBee-Funktionen für Sicherheit bzw. Verschlüsselung und für das Binding soweit entfernt, wie es ohne weitere größere Eingriffe in die generelle zStack-Funktionalität möglich war.

Diese Schritte waren bisher ausreichend, um die sDDS-Implementierung mit der CC2430-Plattform zu verwenden. Dabei ist aber zu beachten, dass bisher nur eine Teilmenge an vorgesehener Funktionalität implementiert wurde. Die genauere Analyse von zStack mit Hinblick auf mögliche Optionen, weitere Funktionalität manuell aus dessen Programmcode zu entfernen, ist Aufgabe zukünftiger Arbeiten, sofern diese Plattform weiterhin verwendet werden soll.

5.5.2.2 Relevante zStack-Schnittstellen für sDDS

Für die Implementierung der plattformspezifischen Schnittstellen von sDDS sind die zStack-Schnittstellen für das Versenden, Empfangen und Adressieren von Nachrichten sowie für die Allokation von Speicher relevant.

Die Datenstruktur `afAddrType_t` dient der Speicherung von Adressen auf Ebene der Anwendungen. Die Funktion `AF_DataRequest()` verschickt einen Datenpuffer an eine entfernte Anwendung, die über die angegebene Knotenadresse und ClusterID referenziert wird. Für den Empfang von Nachrichten oder allgemeinen Ereignissen der zStack-Implementierung muss eine Anwendung bei der Initialisierung des Systems eine „Task" und eine damit assoziierte Callback-Funktion registrieren. Diese wird von zStack aufgerufen, wenn ein Ereignis eintritt. Das Filtern des Ereignisses, das den Empfang einer Nachricht darstellt, ist Aufgabe der Anwendung.

5.5.3 Implementierung des spezifischen Teils

Für die Implementierung des plattformspezifischen Teils von sDDS muss, wie bei der Linux-Version, das `Network`- und das `Memory`-Interface implementiert werden. Dies wurde auch hier auf die Funktionen für den Empfang und Versand sowie die Kapselung von Daten reduziert. Die Unterstützung der Multicast-Funktionalität der Netzwerkschnittstelle ist Aufgabe zukünftiger Erweiterungen.

5.5.3.1 Veränderte Initialisierung

Die Verwendung von zStack macht es notwendig, die Struktur der Initialisierung des universellen Teils von sDDS zu verändern. Anwendungen oder Komponenten, die aktiv arbeiten, werden als „Task" bezeichnet und registrieren in der Initialisierungsphase Callback-Funktionen, die aufgerufen werden, wenn Ereignisse eintreten. Für zyklische Aufgaben kann eine Anwendung einen „Timer" registrieren, der ein solches Ereignis produziert und damit die Ausführung der Anwendungsfunktionalität anstößt.

Für die sDDS-Implementierung bedeutet dies, dass zwei solcher Tasks benötigt werden; eine für die Anwendung und eine für die Netzwerk-Implementierung, die die Nachrichten vom ZigBee-Protokoll entgegennimmt. Die Initialisierung der

Tasks muss getrennt erfolgen, daher ist es nicht möglich, dass die Anwendung mit der generierten `sDDS_init()`-Methode auch die Netzwerk-Implementierung initialisiert. Daher wurde die Methode `ZigBee_init()` implementiert, die der zStack-Semantik entsprechend die Initialisierung durchführt.

Eine weitere Abweichung zu der Initialisierungsmethode der Linux-Version von sDDS ist der Nachrichtenpuffer für den Empfang einer Nachricht. Für die Linux-Version musste der Nachrichtenpuffer der Funktion zum Lesen der UDP-Nachricht übergeben werden. Bei zStack hingegen wird dieser Speicher mit dem Inhalt der Nachricht der registrierten Task zur Verfügung gestellt. Die statische `NetBuffRef`-Instanz wird daher nicht mit einem Nachrichtenpuffer initialisiert, sondern dieser wird ihr jeweils beim Empfang einer Nachricht zugewiesen.

5.5.3.2 Implementierung des Interfaces `Network`

Die Kapselung der ZigBee-spezifischen Adressen geschieht analog zu den IP-Adressen der Linux-Version von sDDS. Die Datenstruktur `afAddrType_t` wird einer in der Implementierungsdatei lokal deklarierten `Locator`-Ableitung hinzugefügt, und über eine entsprechende Typumwandlung können die universellen sDDS-Komponenten auf den `Locator`-spezifischen Teil zugreifen.

Die Implementierung der Methode `send()` des Interfaces `Network` unterscheidet sich nicht stark von der UDP-Implementierung. Um einen Vergleich zu ermöglichen, ist sie dennoch im Listing 5.10 dargestellt.

```
1
2  rc_t Network_send(NetBuffRef buff) {
3
4      afAddrType_t* dest = (afAddrType_t*) &(((struct ZigBeeLocator_t*)
           buff->addr)->addr);
5
6      if ( AF_DataRequest( dest, &(net.epDesc),
7                           GENERICAPP_CLUSTERID,
8                           (byte)buff->curPos,
9                           (byte *)buff->buff_start,
10                          &(net.transID),
11                          AF_DISCV_ROUTE, AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
12    { ... }
13
14 }
```

Quellcode 5.10: Implementierung der `send()`-Methode mit der zStack-Schnittstelle

Die Parameter der Funktion `AF_DataRequest` sind ZigBee-spezifisch. Neben der Zieladresse `dest`, die aus der übergebenen `Locator`-Instanz gewonnen wird, müssen weitere Parameter angegeben werden, auch wenn sie für die verwendete ZigBee-Funktionalität nicht benötigt werden. Die statische Variable `net.epDesc` enthält die Datenstruktur für die Selbstbeschreibung des Knotens bzw. der Anwendung. Diese Selbstbeschreibung ist im ZigBee-Standard spezifiziert und muss von einem ZigBee-Gerät bereitgestellt werden. Ähnliches gilt für den dritten Parameter, der die ClusterID angibt, der die Nachricht zugeordnet ist. Da die Verteilung der Daten auf einem Knoten Aufgabe der sDDS-Middleware ist, kann ein systemweit einheitlicher Wert für diese ID genommen werden, sofern auf allen Knoten die sDDS-Implementierung diesem zugeordnet ist. Über die letzten beiden Parameter kann das Routing-Verhalten der ZigBee-Implementierung beeinflusst werden.

Das Empfangen von Daten unterscheidet sich, wie bereits aufgeführt, von dem Ablauf in der Linux-Version mit UDP/IP. Jede zStack-Task registriert eine Callback-Funktion, die aufgerufen wird, wenn ein Ereignis für die Task auftritt. Für die Netzwerkimplementierung von sDDS für zStack bedeutet dies, dass in dieser Callback-Funktion die ZigBee-Nachrichten ausgefiltert werden müssen und der internen Verarbeitungsmethode übergeben werden. Das Listing 5.11 zeigt die relevanten Teile dieses Vorgangs.

```
1
2  UINT16 ZigBee_ProcessEvent( byte task_id, UINT16 events )
3  {
4      ...
5      if ( events & SYS_EVENT_MSG ) {
6          MSGpkt = (afIncomingMSGPacket_t *)osal_msg_receive( net.taskID );
7
8          while ( MSGpkt ) {
9              switch ( MSGpkt->hdr.event ) {
10
11                 ...
12                 case AF_INCOMING_MSG_CMD:
13                     IncommingFrame (MSGpkt);
14                     break;
15                 ...
16             }
17         ...
18     }
19         ...
20 }
```

Quellcode 5.11: Filtern der ZigBee-Nachrichten aus den Ereignissen für die

sDDS-Task

Die Funktion `ZigBee_ProcessEvent()` ist die Callback-Funktion der Task für die Netzwerkimplementierung von sDDS für den CC2430. Mit der zStack-Funktion `osal_msg_receive()` können die Ereignisse, die einer Task zugeordnet sind, abgerufen werden. Dies geschieht hier in einer Schleife solange, bis alle Ereignisse verarbeitet sind. Die für die sDDS-Version relevanten Ereignisse sind vom Typ `AF_INCOMING_MSG_CMD` und werden an die lokale Funktion `IncomingFrame()` zur Verarbeitung übergeben.

Die Verarbeitung einer eingehenden ZigBee-Nachricht selber geschieht wieder analog zu dem Vorgehen in der Linux-Version von sDDS, mit dem Unterschied, dass diesmal der Nachrichtenpuffer mit dem Daten vom zStack bereitgestellt wird. Das Listing 5.12 zeigt die Teile der Funktion `IncomingFrame()`, die sich von der bereits vorgestellten Linux-Version unterscheiden oder erwähnenswert sind.

```
1  void IncomingFrame (afIncomingMSGPacket_t* msg)
2  {
3      NetBuffRef buff = (NetBuffRef) &inBuff;
4      NetBuffRef_renew(buff);
5      buff->buff_start = msg->cmd.Data;
6
7      struct ZigBeeLocator_t sloc;
8      osal_memcpy(&(sloc.addr), &(msg->srcAddr), sizeof(afAddrType_t));
9
10     Locator loc;
11     if (LocatorDB_findLocator((Locator)&sloc, &loc) != SDDS_RT_OK){
12         if (LocatorDB_newLocator(&loc) != SDDS_RT_OK){ ... }
13         ...
14         osal_memcpy(&(((struct ZigBeeLocator_t*)loc)->addr), &(msg->srcAddr),
               sizeof(afAddrType_t));
15         ...
16     }
17     ...
18     (*(net->newFrame))(buff);
19     ...
20 }
```

Quellcode 5.12: Empfang einer SNPS-Nachricht in der CC2430-Version von sDDS und Weiterleitung der Nachricht an den universellen Teil von sDDS

Die empfangene ZigBee-Nachricht ist in der Struktur `afIncomingMSGPacket_t` enthalten, die als Parameter übergeben wird. Der eigentliche Payload ist in der Struktur das Attribut `cmd.Data`, dessen Speicheradresse mit der der `NetBuffRef`-

Instanz für eingehende Nachrichten assoziiert wird. Damit kann der universelle Teil von sDDS wie gewohnt die in dem Nachrichtenpuffer enthaltene SNPS-Nachricht verarbeiten.

Die Extraktion der Adresse des Senders der Nachricht geschieht analog zu dem Vorgehen in der Linux-Version, mit dem Unterschied, dass diesmal die plattformspezifischen Hilfsfunktionen des zStacks verwendet werden, um die Adressen zu kopieren oder zu vergleichen.

Die Übergabe der empfangenden Nachricht, gekapselt in der `NetBuffRef`-Instanz, an die Callback-Funktion der `DataSink`-Instanz ist identisch mit der Linux-Version.

5.5.4 Übersetzungsprozess

Der Übersetzungsprozess und das Vorgehen, wie die erzeugte Binärdatei auf das Zielsystem übertragen und dort ausgeführt wird, unterscheidet sich stark von der Linux-Version von sDDS.

Die IAR-Entwicklungsumgebung stellt ein ein sich geschlossenes Entwicklungssystem für Anwendungen dar. Da die Teile des zStacks, die nicht als Quellcode von TI bereitgestellt werden, nur für diese Umgebung bereitstehen, gibt es keine Alternative für den Entwicklungs- und Übersetzungsprozess für die CC2430-Plattform unter Verwendung des zStacks. Die IAR-Entwicklungsumgebung ist nur für die Windows-Plattform verfügbar.

Die Benutzerdokumentation des zStacks empfiehlt, für das Anlegen eines neuen Projektes ein existierendes für dieselbe Hardware-Plattform zu kopieren und manuell den Projektnamen in allen Konfigurationsdateien zu ändern. Danach soll dieses neue Projekt in die IAR-Entwicklungsumgebung importiert und manuell über die Konfigurationsmenues an die Anwendung und die Zielplattform angepasst werden. Danach kann die Anwendung entweder direkt in der Umgebung entwickelt, oder existierender Programmcode importiert und dem Projekt hinzugefügt werden.

Der von TI vorgeschlagene Weg für die Verwendung des zStacks und der IAR-Entwicklungsumgebung wurde auch für die Implementierung der CC2430-Version von sDDS verfolgt. Dies führte dazu, dass im Rahmen dieser Arbeit kein einheitlicher Entwicklungsprozess von dem sDDS-Middleware-Framework und

der dort generierten sDDS-Implementierung zu dem Zielsystem CC2430 entwickelt wurde, da der Aufwand für die Integration der IAR-Entwicklungsumgebung in den MDSD-Prozess zu hoch war. Die beiden Schritte sind daher getrennt. Zuerst wird die Middleware und ggf. die Anwendung entwickelt und dann in die IAR-Umgebung manuell eingefügt.

Die Konfiguration von Projekten der IAR-Entwicklungsumgebung basiert auf unterer Ebene größtenteils auf XML-Dateien. Es kann daher möglich sein, das sDDS-Middleware-Framework zu erweitern, damit es diese XML-Konfigurationsdateien ebenfalls erzeugt oder anpasst.

5.6 sDDS-Middleware-Framework

5.6.1 Einleitung

In Abschnitt 4.1.1 wurde die Architektur des sDDS-Middleware-Frameworks vorgestellt, die auf dem in Abschnitt 3.6 untersuchten MDSD-Ansatz basiert und Werkzeuge und Frameworks des Eclipse-Projektes einsetzt. Die Implementierung in dieser Arbeit setzt diesen MDSD-Prozess um und das dazu notwendige Vorgehen ist weitestgehend durch die verwendeten Eclipse-Werkzeuge vorgegeben.

Das Xtext-Framework stellt die Kernkomponenten für die Entwicklung des sDDS-Middleware-Frameworks bereit. Für die in Abschnitt 4.2 definierten Metamodelle der sDDS-Middleware wurden in der Xtext-eigenen Programmiersprache eine DSL definiert. Aus dieser DSL-Beschreibung wurden die Grundgerüste für die weiteren Schritte im MDSD-Prozess erzeugt. Dies umfasst die Erzeugung des Metamodells im Ecore-Format, einen Eclipse-basierten Editor und ein Grundgerüst für die Programmcodegenerierung mit dem Eclipse-Werkzeug Xpand. Dieser Vorgang zur Erzeugung des Grundgerüstes stellt selber einen MDSD-Ansatz dar. Daher wird in der weiteren Beschreibung der Implementierung zwischen den generierten und manuell hinzugefügten, bzw. veränderten Komponenten unterschieden. Damit konnten die Vorteile eines MDSD-Ansatzes bereits für die Entwicklung des Frameworks genutzt werden, das dem Entwickler von Anwendungen für Sensornetze auf Basis der sDDS-Middleware einen solchen Ansatz bieten soll.

Um die Entwicklung des sDDS-Middleware-Frameworks im Rahmen dieser Arbeit zu vereinfachen, wurde nur eine DSL für die in Abschnitt 4.2 definierten Metamodellen definiert, statt wie in Abschnitt 4.1.1 vorgesehen verschiedene DSLs, aus denen der Entwickler die am besten geeignete auswählen kann. Durch die Bereitstellung nur einer DSL ist es möglich den Prozess für die Generierung des Grundgerüstes durch Xtext zu vereinfachen und die ansonsten notwendigen Transformationen zu vermeiden, um die unterschiedlichen Modelle zu vereinigen. Die Unterstützung von verschiedenen Eingabeformaten für die Konfiguration und Modellierung der Middleware, wie es in der Architektur des sDDS-Middleware-Frameworks in Abschnitt 4.1.1 vorgesehen ist, ist daher Aufgabe von zukünftigen Erweiterungen.

Das Vorgehen für die Implementierung des sDDS-Middleware-Frameworks besteht zusammengefasst aus den folgenden aufeinander aufbauenden Schritten:

1. Eine DSL mit Xtext definieren und das Metamodell erzeugen
2. Das erzeugte Metamodell um Aspekte erweitern, die nicht Teil der DSL sind
3. Erzeugung des Grundgerüstes für die Programmcodegenerierung
4. Implementierung der Modelltransformation, um plattformspezifische Informationen in die Modelle einzufügen
5. Implementierung der Programmcode-Templates auf Basis der sDDS-Prototypen und des Metamodells mit Xpand

5.6.2 Entwicklung der DSL

5.6.2.1 Einführung in Xtext

Die in Abschnitt 4.2 beschriebenen Metamodelle für die anwendungs- und plattformspezifische Modellierung und Konfiguration der sDDS-Middleware sind die Basis für die in dieser Arbeit mit Xtext entwickelten DSLs. Xtext besitzt eine eigene Programmiersprache für die Definition von DSLs, die an die Backus-Naur-Form angelehnt ist. Mit dieser Sprache wird die Syntax der textuellen DSL festgelegt. Xtext bildet die Nichtterminalsymbole direkt auf gleichnamige Elemente des Metamodells ab. Terminalsymbole können entweder als Schlüsselwörter verwendet werden oder als Attributwerte Metamodellelementen zugewiesen werden. Ein kurzes Beispiel soll dies erläutern:

```
1 Model : (elements+=Type)*;
2
3 Type : SimpleType | Entity;
4
5 SimpleType : 'type' name=ID;
6 Entity : 'entity' name=ID '{' (properties+=Type); '}';
```

Ein Metamodell wird immer mit dem Nichtterminalsymbol `Model` eingeleitet, das als Wurzel für die weiteren Elemente dient. Es gibt in dem gezeigten Beispiel drei Metamodellklassen: `Type`, `SimpleType` und `Entity`, wobei die beiden letzeren von `Type` abgeleitet sind. Der String *type* ist das Schlüsselwort, das einen `SimpleType` einleitet und *entity* das Schlüsselwort für die Metaklasse `Entity`. Der String, der jeweils nach dem Schlüsselwort folgt, wird der Metaklasse als Attribut `name` zugewiesen. Da beide abgeleiteten Klassen dieses Attribut besitzen, ist es im von Xtext erzeugten Metamodell ein Attribut der Klasse `Type`. Die Metaklasse `Entity` kann unter dem Attribut `properties` weitere Instanzen von `Type` besitzen. In der DSL werden diese als Liste zwischen geschweiften Klammern angegeben.

Das Beispiel und die gegebene Erklärung zeigt die direkte Abbildung von Nichtterminalsymbolen auf die Elemente des Metamodells, die über Schlüsselwörter der DSL eingeleitet werden. Für diese Arbeit bedeutet dies, dass sich aus den in Abschnitt 4.2 definierten Metamodellen die grundsätzliche Struktur der DSL ergibt, die um geeignete Schlüsselwörter ergänzt werden muss. Des Weiteren referenzieren einige Metaklassen von sDDS Elemente aus anderen Metaklassen. Die DSL-Definitionsprache von Xtext besitzt für diesen Fall Mechanismen, um Assoziationen zwischen Instanzen von Metaklassen festlegen zu können. Dafür ist es notwendig, dass Metaklassen-Instanzen eindeutige Namen haben, die in dem jeweiligen abgeleiteten Modell als Referenz verwendet werden können. Auf diesen Aspekt wird in der weiteren Beschreibung der Implementierung genauer eingegangen.

Im Weiteren soll an einigen Beispielen für die Modellierung des sDDS-Systems gezeigt werden, wie die Metamodelle in die DSL für sDDS Arbeit umgesetzt wurden.

5.6.2.2 DSL für die Modellierung des DDS-Systems

In diesem Abschnitt soll am Beispiel einiger einfacher DSLs für die in Abschnitt 4.2 definierten Metamodelle beschrieben werden, wie auf Basis von Xtext das Grundgerüst für das sDDS-Middleware-Framework erzeugt wurde. In diesen Beispielen werden alle für diese Arbeit relevanten Einzelheiten dargestellt. Die Definition der anderen DSLs für die Metamodelle in dieser Arbeit wurde analog durchgeführt.

Datatypes-Metamodell

Das Metamodell, das das Datenmodell von sDDS beschreibt und damit für die Deklaration der Topic-Datentypen benötigt wird (Siehe Abschnitt 4.2.2 für die Definition) ist ein anschauliches Beispiel für die Entwicklung einer DSL mit Xtext. Wie in Abschnitt 4.2.2 beschrieben, orientiert sich die Struktur des *Datatypes*-Metamodells an dem UML-Profil für DDS und der Definition von Datentypen mit OMG IDL. OMG IDL ist eine DSL für die Beschreibung von Schnittstellen für Anwendungen und wird von dem DDS-Standard implizit für die Deklaration der Datentypen vorgesehen. Daher lag es nahe, in dieser Arbeit die Syntax von OMG IDL für die DSL des *Datatypes*-Metamodells zu verwenden.

Das Listing 5.13 zeigt die relevanten Teile der DSL-Definition für das Datenmodell, es wird aber der Übersichtlichkeit wegen nur ein Teil der Datentyp-Definitionen dargestellt.

```
1  IDL: 'IDL' '{'  (idlEntities+=TopicField)+  '}';
2
3  TopicField: ConstructedType | SimpleType  | CollectionType;
4
5  ConstructedType: TopicType;
6
7  TopicType: 'struct' name=ID '{' (fields+=TopicField)+ '}' ';'
8            '#pragma' pragma=ID ;
9
10 SimpleType: (Boolean_t | Char_t | Enum_t | IntType | Octet_t) ';';
11 IntType: SignedIntType | UnsignedIntType;
12 SignedIntType: Short_t | Long_t | LongLong_t;
13 UnsignedIntType: ULongLong_t | UShort_t | ULong_t;
14 Boolean_t: 'boolean' name=ID;
15 Char_t: 'char' name=ID;
16 Long_t: 'long' name=ID;
```

Quellcode 5.13: DSL für das *Datatypes*-Metamodell in Xtext

Wird diese Definition mit dem in Abschnitt 4.2.2 vorgestellten *Datatypes*-Metamodell verglichen, zeigt sich die direkte Abbildung der Metaklassen auf Nichtterminalsymbole. Die Deklaration von Datentypen in einem Modell, das mit dieser DSL beschrieben wird, erfolgt in einer Umgebung, die mit dem Schlüsselwort *IDL* eingeleitet und von geschweiften Klammern umschlossen wird. Innerhalb dieses Bereiches wird die Syntax von OMG IDL für die Deklaration von Datentypen verwendet. Die Syntax wurde für die Topic-Datentypen erweitert, um eingebettete Datentypen als „Keys" zu definieren und sie von normalen Strukturen unterscheiden zu können. Die Syntax wurde von OpenSplice DDS übernommen, bei dem mit dem Schlüsselwort `#pragma` die Definition des „Keys" erfolgt, dies wurde in den Grundlagen in Abschnitt 2.3.2.2 beschrieben und im Listing in Abschnitt 2.3.2.2 auf Seite 25 ein Beispiel für die Syntax dargestellt. Da diese Funktionalität in dieser Arbeit noch nicht unterstützt wird, hat das Metamodell dafür keine Entsprechung, und die DSL sieht vor, dass der dem Schlüsselwort folgende String in dem Attribut `pragma` der Metaklasse `TopicType` gespeichert wird. Spätere Erweiterungen können diesen String auswerten oder das Metamodell um diesen Aspekt erweitern.

Dataspace-Metamodell

Eine wichtige Funktionalität von Xtext für diese Arbeit ist die Möglichkeit, Assoziationen zwischen Metaklassen-Instanzen zu bilden. Mit dem in Abschnitt 4.2.3 definierten *Dataspace*-Metamodell soll der globale DDS-Datenraum für Anwendungen definiert werden. Das bedeutet, es werden die im System vorgesehenen Topics mit ihren Eigenschaften festgelegt. Jedes Topic muss einem Topic-Datentyp zugeordnet sein, welcher mit dem *Datatypes*-Metamodell definiert wird. Das bedeutet in der Modellierung muss es möglich sein einen definierten Datentyp mit der Definition eines Topics zu verbinden. Des Weiteren können den Topics QoS-Richtlinien zugeordnet werden, deren QoS-Parameter im Modell konfiguriert werden. Die QoS-Richtlinien sind in dem Metamodell *DDSQoS* definiert und werden im *Dataspace*-Metamodell verwendet.

Der Teil der sDDS-DSL für das *Dataspace*-Metamodell, welcher mit Xtext in dieser Arbeit definiert wurde, ist im Listing 5.14 dargestellt.

```
1 Domain: 'domain' name=ID '{' (domainEntities+=DomainEntity)+ '}';
2 DomainEntity: Topic;
3 Topic: 'topic' name=ID '{' (topicContents+=TopicContent)+ '}';
4 TopicContent: DataType | QoSRequirement;
5 DataType: 'dataType' dataType=[TopicType];
6 QoSRequirement: 'qos' '{' (qosPolicies+=QosPolicy)+ '}';
```

Quellcode 5.14: DSL für das *Dataspace*-Metamodell in Xtext

Die gewählte Syntax für die DSL von sDDS hat den generellen Aufbau, dass ein Element des Metamodells immer von einem Schlüsselwort eingeleitet wird, das dem Namen der jeweiligen Metaklasse entspricht oder diesen abkürzt. Fungiert eine Metaklasse als Container für Instanzen weiterer Metaklassen, dann werden diese als Liste zwischen geschweiften Klammern eingeschlossen.

Die Metaklasse `Datatypes` hat eine Assoziation zu einem Topic-Datentyp, der mit der Metaklasse `TopicType` definiert wird. In der Syntax von Xtext wird eine Assoziation zu einer anderen Metaklasseninstanz mit eckigen Klammern definiert, in denen der Typ der assoziierten Metaklasse angegebenen ist. Demnach wird in Zeile 5 des Listings 5.14 dem Attribut *dataType* der Metaklasse `DataType` eine Assoziation zu einem `TopicType` zugewiesen.

Der von Xtext generierte Parser für die DSLs löst Assoziationen zwischen Instanzen der Metaklassen über das gesetzte Attribut *name* auf. Dies soll hier mit der Syntax der DSL an einem Beispiel erläutert werden:

```
1  IDL {
2      struct Foo {
3          octet bar;
4      };
5      #pragma Foo
6  }
7
8  domain Beispiel {
9      topic FooTopic {
10         dataType Foo
11     }
12 }
```

In dieser Konfiguration wird der Topic-Datentyp *Foo* deklariert, der aus einem primitiven Byte-Datentyp besteht, der den Namen *bar* hat. Die Deklaration des Datentyps geschieht in der OMG IDL-Umgebung, die mit dem Schlüsselwort *IDL* eingeleitet wird und in der die OMG IDL-Syntax verwendet wird.

Die Modellierung eines globalen Datenraums, der einer DDS-Domain entspricht, wird mit dem Schlüsselwort *domain* eingeleitet und benötigt einen in dem sDDS-System eindeutigen Namen, der auf die DomainID von sDDS abgebildet werden kann. Die DDS-Domain *Beispiel* in der hier betrachteten Beispielkonfiguration enthält das Topic *FooTopic*. Die Assoziation zu dem Topic-Datentyp wird mit dem Schlüsselwort *dataType* eingeleitet, nachdem der Name des Topic-Datentyps, hier *Foo*, angeführt ist.

5.6.2.3 Zusammenfügen der Metamodelle

Wie in Abschnitt 5.6.1 bereits aufgeführt, wurde für die prototypische Implementierung des sDDS-Middleware-Frameworks nur eine gemeinsame DSL für alle in Abschnitt 4.2 definierten Metamodelle definiert. In diesem Abschnitt soll dieser Implementierungsschritt beschrieben werden und welche Probleme sich damit ergeben und in späteren Versionen von sDDS gelöst werden müssen.

In Abschnitt 4.2.6 wurden in der Darstellung 4.9 auf Seite 121 die Zusammenhänge zwischen den Elementen der Metamodelle von sDDS dargestellt. Diese Struktur wurde in der DSL für sDDS abgebildet. Das Listing 5.15 zeigt die Umsetzung mit der Syntax von Xtext.

```
1 Model : (systems+=System)+;
2
3 System: 'namespace' name=ID (systemEntities+=SystemEntity)+;
4
5 SystemEntity: NodeApplication | Domain | IDL;
```

Quellcode 5.15: Vereinigung der Metamodelle von sDDS in einer DSL

Eine Konfigurationsdatei kann mehrere Systembeschreibungen enthalten, die über das Schlüsselwort *namespace* eingeleitet werden. Der nach dem Schlüsselwort folgende String wird als Name des Namensraum verwendet und in dient bei der Generierung der anwendungsspezifischen Teile der sDDS-Middleware als Präfix für die Bezeichner der Komponenten.

Jede Systembeschreibung selber besteht aus eine Menge an Knotenkonfigurationen (*NodeApplication*-Metaklasse), Datentypdeklarationen (*IDL*-Metaklasse) und Definitionen von Datenräumen, also DDS-Domains. Elemente in diesen Gruppen können sich wie beschrieben gegenseitig referenzieren.

Bei der Implementierung der DSL für sDDS zeigte sich, dass es ein Problem mit der Benennung von Objekten in Zusammenhang mit den vorgesehenen Namensräumen des DDS-Standards gibt. Der von Xtext generierte Parser für eine DSL löst die Assoziation zwischen zwei Instanzen über den Namen der Instanzen auf. DDS-Domains fungieren als eine Art Namensraum. Es ist daher möglich, dass zwei Topics mit demselben Namen in zwei unterschiedlichen Domains existieren. In der Modellierung des sDDS-Systems auf einem Middleware-Knoten referenziert ein DataReader oder DataWriter ein Topic über dessen Namen und ist selber in der Umgebung einer Domain eingebunden (Siehe Abbildung 4.6 auf Seite 117 im Abschnitt 4.2.4.1, der Definition des *sDDSNodeStructure*-Metamodells). Der von Xtext generierte Parser hat aber kein Wissen über diese Semantik und kann daher keine eindeutige Zuordnung zu einem Topic finden.

Xtext sieht die Option vor, dass die spezielle Semantik von DSLs für die Auflösung von Assoziationen, definiert und während der Generierung des Parsers als Erweiterung hinzugefügt werden kann. Für diese Erweiterung ist es notwendig, Java-Interfaces zu implementieren und manuell die richtige Metaklassen-Instanz im Ecore-Objektbaum zu finden. In dieser Arbeit wurde dieser Schritt nicht durchgeführt, daher werden Namensräume auf Ebene der Domains nicht durchgesetzt und alle Namen müssen in der Modellierung von sDDS eindeutig sein.

5.6.3 Meta- und Modelltransformationen

Xtext erzeugt aus der definierten DSL-Syntax unter anderem ein Metamodell auf Basis von Ecore. Dieses ist die Basis für die Verarbeitung der Modelle im sDDS-Middleware-Framework. Da als Ausgangsbasis für die Erzeugung des Metamodells nur die definierte DSL-Syntax verwendet wird, sind Teile des in Abschnitt 4.2 definierten Metamodells, die nicht über die DSL konfiguriert werden, nicht enthalten. Des Weiteren ist es notwendig, im Prozess der Generierung der sDDS-Middleware-Implementierung Modelltransformationen durchzuführen, welche das Modell selber verändern. Für beide Aufgaben wird das Eclipse-Werkzeug Xtend verwendet, das für Änderungen an Modellen (also auch Metamodelle) eine eigene funktionale Sprache besitzt. Diese Sprache operiert auf dem Typsystem, das auch von Xtext verwendet wird.

5.6.3.1 Generierte Metamodelle erweitern

Im Rahmen der Implementierung dieser Arbeit wurde die Erweiterung des generierten Metamodells auf einen Teilaspekt reduziert. Das Metamodell *Datatypes* (Siehe Abschnitt 4.2.2) sieht für die Ableitungen der Metaklasse `SimpleType` das Attribut *memSize* vor, dessen Werte der Größe des Datentyps auf der Zielplattform entsprechen sollen. Da dieses Attribut nicht bei der Deklaration der Datentypen in der DSL gesetzt wird, ist es nicht im von Xtext generierten Metamodell enthalten.

Das Xtext-Framework sieht die Option vor, während der Generierung des Metamodells ein in Xpand geschriebenes Programm auf ein Zwischenergebnis des generierten Metamodells anzuwenden, das die Änderungen an der Struktur des Metamodells durchführt. Für das generierte Metamodell von sDDS war es notwendig das Attribut *memSize* der Metaklasse `SimpleType` hinzuzufügen. Das Listing 5.16 zeigt den vollständigen Xtend-Programmcode für diesen Vorgang.

```
1  import ecore;
2  import ecoce::EcorePackage;
3
4  process(xtext::GeneratedMetamodel this) :
5          process(ePackage);
6
7  process(EPackage this):
8          eClassifiers.typeSelect(EClass).process();
9
10 process(EClass this):
11         switch (this.name) {
12                 case "SimpleType":
13                     addAttribute(this, "memSize", getDataType("EInt"))
14                 default: null
15         };
16
17 addAttribute(EClass this, String name, EClassifier dtName):
18         let a = new EAttribute: a.setName(name)
19         -> a.setEType(dtName)
20         -> this.eAttributes.add(a);
21
22 ecore::EClassifier getDataType(String name):
23         JAVA de.dopsy.Helper.getEcoreDataType( java.lang.String);
```

Quellcode 5.16: Einfügen des Attributes *memSize* in die Metaklasse `SimpleType` mit Xtend

Das Xtend-Programm beginnt in der `process`-Funktion in Zeile 4, die den Xtext-internen Datentyp `GeneratedMetamodel` übergeben bekommt. Xtend unter-

stützt Polymorphie und wählt für den jeweiligen Typ eines zu verarbeiteten Elementes die entsprechende Funktion aus. Die Änderungen an dem Metamodell basieren auf dem Ecore-Metametamodell, und da die Veränderung in einer Metaklasse durchgeführt werden soll, werden in Zeile 8 alle Elemente mit dem Ecore-Typ *EClass* ausgewählt und weiter verarbeitet.

Die `process`-Funktion in Zeile 10 führt auf Basis des Typs der übergebenen Metaklasse eine Fallunterscheidung durch, wobei hier nur der Typ `SimpleType` von Interesse ist. Für dieses Element wird die Funktion `addAttribute` aufgerufen, der die Referenz auf das Element, den Namen des hinzuzufügenden Attributes und eine Referenz auf den Datentypen des Attributes übergeben werden.

Ein Attribut besteht immer aus einem Namen und einen Datentyp. Das Attribut *memSize* gibt die Menge an Speicher an, daher sollte als Datentyp ein Ganzzahlentyp gewählt werden. In Ecore kommt dafür die Klasse `EInt` in Frage. Bei der Implementierung zeigte sich das Problem, dass es in der Xtend-Umgebung nicht möglich ist, eine Referenz auf einen Ecore-Datentyp zu bekommen. Es musste eine Java-Erweiterung verwendet werden, die mit der Funktion `getDataType` gekapselt wurde. Ein Java-Programm kann eine Referenz auf einen Datentyp von dem Singelton `eINSTANCE` erhalten. Die implementierte Java-Erweiterung gibt diese Referenz in das Xtend-Programm zurück, in dem es in Zeile 18 verwendet wird, um eine neue Instanz der Metametaklasse `EAttribute` zu erzeugen und der Metaklasse `SimpleType` hinzuzufügen.

5.6.3.2 PIM2PSM-Transformation

Wie im vorherigen Abschnitt wird die Modelltransformation, die plattformspezifische Informationen in ein ein Modell einfügt, mit Xtend durchgeführt. In dieser Arbeit wurde dieser Vorgang, wie bei der Änderung des generierten Metamodells, auf einen Teilaspekt reduziert: Das Hinzufügen der plattformspezifischen Speichergröße für die Datentypen.

Dieser Vorgang wurde stark vereinfacht implementiert und hatte das Ziel, das generelle Verfahren für diese Aufgabe zu testen. Die PIM2PSM-Transformation wird vor der Programmcodegenerierung durchgeführt und besteht aus der Xtend-Funktion, die verkürzt in Listing 5.17 dargestellt ist.

```
1  PIM2PSM(Model m):
2          m.eAllContents.typeSelect(SimpleType).SetDataTypeSize();
3
4  SetDataTypeSize(SimpleType d):
5          switch (d.metaType){
6                  case Long_t: d.setMemSize(4)
7                  case Short_t : d.setMemSize(2)
8                  case ULong_t: d.setMemSize(4)
9                  case Octet_t: d.setMemSize(1)
10                 default: d.setMemSize(0)
11         };
```

Quellcode 5.17: PIM2PSM-Transformation mit Xtend für das Einfügen von plattformspezifischen Datentypgrößen

Die Funktion `PIM2PSM` bekommt das gesamte Modell des sDDS-Systems übergeben und ruft für jede Instanz der Metaklasse `SimpleType` die Funktion `SetDataTypeSize` auf. In der Funktion wird für jede Ableitung eines primitiven Datentyps manuell und „fest verdrahtet" ein Wert für die Größe des repräsentierten Datentyps gesetzt.

Zukünftige Versionen des sDDS-Middleware-Frameworks sollten an dieser Stelle auf ein Modell der Zielplattform zugreifen und die Größe der Datentypen dort auslesen und setzen. Generell zeigt sich mit dieser minimalen PIM2PSM-Transformation, dass Xtend ein mächtiges Werkzeug für die Manipulation von Modellen ist, und dass schnell und einfach auch komplexere Transformationen programmiert werden können.

5.6.4 Programmcodegenerierung

5.6.4.1 Vorgehensweise

Wie im Ablauf der Implementierung des sDDS-Middleware-Frameworks in Abschnitt 5.6.1 beschrieben, wurde das Grundgerüst für die Programmcodegenerierung aus der Syntax der sDDS-DSL von Xtext generiert. Das Grundgerüst umfasst einen Produktionsprozess, um aus einer textuellen Modellierung über Templates Programmcode zu erzeugen. Die Templates werden mit der Xpand-Sprache definiert. In dieser funktionalen Sprache kann auf die Elemente der Modelle zugegriffen werden und basierend auf deren Struktur und Parametrisierung Programmcode erzeugt werden. Dabei ist es möglich, Xtend-Programme als Hilfsfunktionen zu verwenden, um komplexere Operationen durchzuführen.

Die notwendige Implementierung für die Programmcodegenerierung umfasst damit ausschließlich die „Programmierung" der Xpand-Templates und einiger Xtend-Hilfsfunktionen. Dafür ist es notwendig, die Struktur der Metamodelle auf die des Programmcodes der sDDS-Prototypen abzubilden. Dieser Vorgang stellt bereits einen ersten Test dar, um zu überprüfen, wie gut das Design der sDDS-Middleware bzw. dessen prototypische Implementierung auf den Entwurf der Metamodelle und damit der geplanten Konfigurierbarkeit passt.

Wie in dem Abschnitt 5.3.1 über die Implementierung der sDDS-Middleware-Prototypen beschrieben, wurde in dieser Arbeit die Menge der zu generierenden Dateien eingeschränkt. Ausschließlich die anwendungsspezifischen Dateien für den Programmcode müssen generiert werden. Die Konfiguration der universellen Komponenten erfolgt über Präprozessordirektiven, die in einer generierten Konfigurationsdatei auf Basis des Modells festgelegt werden.

Die Modellierung der sDDS-Middleware enthält für jeden Middleware-Knoten oder ggf. für jede Gruppe von Knoten mit identischer Hardware und Anforderungen an die sDDS-Middleware eine Instanz der Metaklasse `NodeApplication`. Jedes dieser Teilmodelle benötigt eine eigene sDDS-Middleware-Implementierung, für die ein eigener Satz anwendungsspezifischer sDDS-Implementierungsdateien generiert wird. Hierfür wird für jeden konfigurierten Middleware-Knoten ein Ordner angelegt, der den Namen der jeweiligen *NodeApplication*-Instanz als Suffix enthält.

Insgesamt wurden für die folgenden Dateien Xpand-Templates erstellt:

Konfigurationsdatei: Eine C-Header-Datei mit den Präprozessordirektiven für die Konfiguration des universellen sDDS-Komponenten.

sDDS-Header-Datei: Eine C-Header-Datei für die Deklaration der Topic-Datentypen und deren spezifischer DDS-Schnittstellen, die die Anwendung benötigt. Des Weiteren enthält sie `extern`-Deklarationen für die modellierten DataReader und DataWriter, auf die die Anwendungen nach der Initialisierung des sDDS-Systems zugreifen können.

sDDS-Implementierungsdatei: Eine C-Implementierungsdatei für die Topic-Datentyp-spezifischen Datenkonvertierungsmethoden, die Abbildung der DDS-Schnittstellen auf universelle sDDS-Methoden und die Implementierung der anwendungsspezifischen Initialisierung des sDDS-Systems.

Anwendungsskelett: Eine C-Implementierungsdatei für die Linux-Plattform, die ein Programmskelett für eine generische sDDS-Anwendung darstellt, und in die direkt Anwendungslogik eingefügt werden kann.

Im Weiteren werden interessante Aspekte der Implementierung der Xpand-Templates für diese Dateien beschrieben.

5.6.4.2 Erzeugung der Konfigurationsdatei

Die Konfigurationsdatei `CONSTANTS.h` ist sehr übersichtlich und das Xpand-Template, das sie erzeugt, eignet sich gut, die generelle Vorgehensweise von Xpand zu erklären. Wie bei der Verarbeitung eines Modells mit Xtend, die in Abschnitt 5.6.3.1 beschrieben wurde, bekommt eine initiale Xpand-Funktion das gesamte Modell übergeben und delegiert die Verarbeitung an spezialisierte Funktionen. Eine solche spezialisierte Funktion wird in dieser Arbeit für die Erzeugung der Konfigurationsdatei verwendet. Listing 5.18 zeigt einen Ausschnitt des entsprechenden Templates. In Listing 5.19 wird als anschauliches Beispiel eine mögliche `CONSTANTS.h`-Datei gezeigt. Die verwendeten Xtend-Hilfsfunktionen werden nicht dargestellt.

```
1  DEFINE createConstantsFile(String ns, String path, System sys) FOR Application»
2  «FILE path + 'CONSTANTS.h'»
3  /** foo bla fasel */
4  «makeHeader("CONSTANTS")»
5  «EXPAND defSNPSVersion FOR sys-»
6  «EXPAND defMaxDR FOR this-»
7  «EXPAND defFlagTopicHasPub FOR this-»
8  «EXPAND defTopicMaxCount FOR this-»
9  «makeFooter("CONSTANTS")»
10 «ENDFILE»
11 «ENDDEFINE»
12
13 «DEFINE defSNPSVersion FOR System-»
14 #define SDDS_NET_VERSION 0x01
15 «ENDDEFINE»
16
17 «DEFINE defMaxDR FOR Application-»
18 #define sDDS_MAX_DR_APP «this.eAllContents.typeSelect(DataReader).size»
19 «ENDDEFINE»
20
21 «DEFINE defFlagTopicHasSub FOR Application-»
22 «IF !this.eAllContents.typeSelect(DataReader).isEmpty-»
23 #define sDDS_TOPIC_HAS_SUB
24 «ENDIF-»
```

```
25 «ENDDEFINE»
26
27 «DEFINE defTopicMaxCount FOR Application-»
28 #define sDDS_TOPIC_MAX_COUNT «returnTopicList(this).size»
29 «ENDDEFINE»
```

Quellcode 5.18: Xpand-Template für die sDDS-Konfigurationsdatei

```
1  /* generated file ... */
2  /* filename: CONSTANTS.h */
3
4  #ifndef _CONSTANTS_H
5  #define _CONSTANTS_H
6
7  #define SDDS_NET_VERSION 0x01
8  #define sDDS_MAX_DR_APP 2
9  #define sDDS_TOPIC_HAS_PUB
10 #define sDDS_TOPIC_MAX_COUNT 2
11
12 #endif /* CONSTANTS_H */
```

Quellcode 5.19: Ein Beispiel für eine generierte sDDS-Konfigurationsdatei

Die Funktion `createConstantsFile`, die die Konfigurationsdatei erzeugt, bekommt als Parameter den String für den Namensraum, den Dateipfad, in dem die erzeugte Datei gespeichert werden soll und das Modell in der Ecore-Repräsentierung übergeben.

Das Beispiel verwendet die Xtend-Hilfsfunktionen `makeHeader`, `makeFooter` und `returnTopicList`. Die ersten beiden Funktionen erzeugen den Text am Anfang und Ende der C-Header-Datei im Listing 5.19. `returnTopicList` hat eine komplexere Funktionalität und gibt eine Liste mit Referenzen auf alle Topics zurück, die für einen sDDS-Middleware-Knoten relevant sind.

Mit dem Xpand-Schlüsselwort `FILE` wird eine neue Datei angelegt und mit `EXPAND` wird eine Xpand-Funktion aufgerufen. In Zeile 5 wird die Funktion in Zeile 13 aufgerufen, die statisch den String mit der Präprozessordirektive für die Version der sDDS-Implementierung in die Datei schreibt. Das Ergebnis ist im Listing 5.19 in Zeile 7 zu sehen. Dieses Schema gilt für die gesamte Programmcodegenerierung.

Für die Übersetzung des universellen Teils der sDDS-Middleware ist es notwendig zu wissen, wie viele DataReader auf einem Middleware-Knoten existieren, damit die entsprechende Menge an Datenstrukturen allokiert werden kann. In Zeile 18 des Listings 5.18 werden alle Instanzen der Metaklasse `DataReader`

des Teilmodells für einen Middleware-Knoten ausgewählt und ihre Anzahl als String in die Datei geschrieben. Zeile 8 des Listings 5.19 zeigt das Ergebnis, wenn für den Knoten zwei DataReader modelliert sind. Analog wird für die Anzahl der Topics verfahren, nur mit dem Unterschied, dass hierfür die Xtend-Funktion `returnTopicList` verwendet wird. Dies ist notwendig, da auf einem Knoten mehrere DataReader oder DataWriter dasselbe Topic bedienen können und bei der vorgestellten Methode für das Zählen der DataReader diese Topics sonst mehrfach gezählt würden.

5.6.4.3 Erzeugung der sDDS-Header-Datei

Für die Erzeugung der sDDS-Header-Datei soll auf zwei Aspekte eingegangen werden. Zum einen die Deklaration der Topic-Datentypen, da diese ein wiederkehrendes Thema dieser Arbeit ist. Zum anderen, wie die DDS-Schnittstellen für die Topic-Datentypen-spezifischen DataWriter erzeugt werden.

Deklaration der Topic-Datentypen

Das Listing 5.20 gibt den Ausschnitt aus dem Xpand-Template für die Generierung der sDDS-Header-Datei wieder, der die Deklaration der Topic-Datentypen generiert. Dies geschieht mit der Funktion `writeTopicStructDecl`.

```
1  ...
2  «EXPAND writeTopicStructDecl(namespace) FOREACH
3          returnTopicList(this).topicContents.typeSelect(DataType).dataType»
4  ...
5
6  «DEFINE writeTopicStructDecl(String ns) FOR TopicType»
7  struct «ns»_«this.name» {
8  «FOREACH this.fields.typeSelect(SimpleType) AS e-»
9  «returnDataTypeAsString(e)» «e.name»;
10 «ENDFOREACH-»
11 };
12 typedef struct «ns»_«this.name» «ns»_«this.name»;
13 «ENDDEFINE»
```

Quellcode 5.20: Darstellung der Xpand-Funktion für die Generierung der Topic-Datentyp-Deklaration

In Zeile 2 ist der Aufruf der Funktion `code:writeTopicStructDecl` in der Funktion für die Erzeugung der gesamten Header-Datei dargestellt. Die bekannte Xtend-Hilfsfunktion `returnTopicList` gibt eine Liste mit allen Topics für den

Middleware-Knoten zurück. Auf dieser Liste werden dann Auswahloperationen für jedes Topic durchgeführt, die auf der Struktur des in Abschnitt 4.2.3 definierten Metamodells für die DDS-Domains basieren. Aus den möglichen Inhalten einer Topic-Definition werden die Datentypen (`DataType`) ausgewählt und über die im Attribut *dataType* gespeicherte Assoziation die jeweilige Topic-Datentyp-Instanz (`TopicType`) ausgewählt und der Funktion `writeTopicStructDecl` übergeben.

In der `writeTopicStructDecl`-Funktion wird die `C-struct` Struktur mit dem Namen des Topic-Datentyps als Bezeichner generiert. In der Struktur wird jeder enthaltene Datentyp deklariert. Die Xtend-Hilfsfunktion `returnDataTypeAs String` gibt für jede Datentyp-Instanz des Metamodells den plattformspezifischen Namen bzw. Bezeichner zurück. In der Linux-Version von sDDS wäre dies für die Metaklasse `ULong_t` (siehe Abschnitt 4.2.2 für die Definition) der String „uint32_t", für einen vorzeichenlosen Ganzzahlentyp mit einer Länge von 32 Bit.

Deklaration der Topic-Datentyp-spezifischen DDS-Schnittstellen

Das Listing 5.21 zeigt die Teile des Xpand-Templates, die die Deklaration der Topic-Datentyp-spezifischen DDS-Schnittstelle für den DataWriter generieren. Wie schon bei der Beschreibung des vorherigen Template-Ausschnittes ist in Zeile 2 der Aufruf der spezialisierten Funktion und danach die Funktion selber abgebildet.

```
1  ...
2  «EXPAND writeDWDDSAPI(namespace)
3      FOREACH this.eAllContents.typeSelect(DataWriter)»
4  ...
5  «DEFINE writeDWDDSAPI (String ns) FOR DataWriter»
6  DDS_ReturnCode_t «ns»_«returnDataTypeName(this)»DataWriter_write(
7          «ns»_«returnDataTypeName(this)»DataWriter _this,
8          const «ns»_«returnDataTypeName(this)»* instance_data,
9          const DDS_InstanceHandle_t handle);
10 «ENDDEFINE»
```

Quellcode 5.21: Darstellung der Xpand-Funktion für die Generierung der Deklaration der Topic-Datentyp-spezifischen DDS-Schnittstelle des DataWriters

Die Funktion `writeDWDDSAPI` wird für jeden DataWriter in dem Modell für einen Middleware-Knoten aufgerufen. Sie erzeugt die Deklaration der `write()`-Methode des `DataWriter` für einen Topic-Datentyp. Die Signatur dieser Methode

enthält dementsprechend den Namen des Topic-Datentyps als Bestandteil des Bezeichners der Klasse und den plattformspezifischen Bezeichner des Datentyps als Parameter. Die Xtend-Hilfsfunktion `returnDataTypeName` gibt den Bezeichner bzw. Namen des Topic-Datentypen zurück, mit dem der assoziierte DataWriter verbunden ist.

5.6.5 Software-Entwicklungsprozess mit dem sDDS-Middleware-Framework

Im Abschnitt 4.3 des Entwurfs der Architektur des sDDS-Middleware-Frameworks wurde der Prozess festgelegt, der aus der Modellierung der Middleware und den Anforderungen der Anwendung bzw. aus deren Konfiguration den angepassten und plattformspezifischen Programmcode der sDDS-Middleware-Implementierung erzeugt.

Im Rahmen dieser Arbeit und der prototypischen Implementierung des Middleware-Frameworks wurden nicht alle Schritte des festgelegten Prozesses realisiert bzw. nicht im vollem Umfang. Der entwickelte Verarbeitungsprozess selber wurde primär durch die Vorgaben oder Standardeinstellungen der verwendeten Eclipse-Werkzeuge bestimmt.

Das mit Xtext erzeugte Grundgerüst für die Programmcodegenerierung enthält auch einen „Workflow", der in einer XML-Datei definiert wird. Dieser legt die Reihenfolge fest, in der Komponenten aufgerufen werden und welche Daten sie als Eingangsparameter bekommen bzw. wie deren Ergebnisse an andere Komponenten weitergeleitet werden. Das erzeugte Grundgerüst besteht aus zwei getrennten Teilen, die als eigenständige Projekte in der Eclipse-Entwicklungsumgebung angelegt werden. Das erste ist ein in Eclipse integrierter Editor für die jeweilige DSL und das andere erzeugt basierend auf dem Workflow den Programmcode. Im Rahmen dieser Arbeit wurden die beiden Projekte nicht verbunden, so dass für die Erzeugung der sDDS-Middleware-Implementierung manuelle Schritte notwendig sind.

Der primäre Teil des Middleware-Framework-Prototyps ist das Projekt für die Programmcodegenerierung. Der generierte Workflow wurde übernommen, so dass das Modell und die Konfiguration der zu generierenden Middleware-Implementierung in einer im Workflow festgelegten Textdatei bereitgestellt werden muss. Das Modell wird dann von dem von Xtext generierten Parser eingelesen und in einen

Ecore-Objektbaum überführt. Dieser Objektbaum wird dem Programmcodegenerator übergeben, der als erstes die in Abschnitt 5.6.3.2 beschriebene Modelltransformation durchführt und plattformspezifische Informationen in das Modell einfügt. Danach wird der Objektbaum von den in Abschnitt 5.6.4 beschriebenen Xpand-Templates in Programmcode übersetzt.

Der nächste Schritt für die Weiterentwicklung des Middleware-Frameworks sollte die Integration des Editors mit der Programmcodegenerierung sein. Damit kann die Middleware direkt modelliert und konfiguriert und im nächsten Arbeitsschritt direkt generiert werden. Des Weiteren sollte die Modelltransformation in einem eigenen Arbeitsschritt des Workflows ausgeführt werden. Im Kontext der Xtext bzw. Xpand-Werkzeuge von Eclipse gibt es eine eigene Constraint-Sprache, die OCL für UML ähnelt. Mit dieser Sprache ließe sich das Modell auf Konsistenz und die Einhaltung von plattformspezifischen Bedingungen überprüfen und könnte ebenfalls als eigenständiger Arbeitsschritt im Workflow ausgeführt werden. Darauf aufbauend kann der generierte Editor für die sDDS-DSL erweitert werden, so dass die Rückmeldungen der Überprüfung des Modells dem Entwickler direkt angezeigt werden.

Kapitel 6

Evaluation

Dieses Kapitel befasst sich mit der Evaluation des in dieser Arbeit entworfenen und prototypisch implementierten MDSD-basieren Generierungsprozesses für eine anwendunsspezfisch angepasste sDDS-Middleware. Das übergeordnete Ziel dieser Arbeit ist es, wie in der Einleitung (Kapitel 1) bzw. der Aufgabenstellung in Abschnitt 3.1.1 beschrieben, die Software-Entwicklung für Sensornetze zu vereinfachen. Die Überprüfung, inwieweit dieses Ziel erreicht wurde, ist die in diesem Kapitel betrachtete Fragestellung.

6.1 Ablauf

Die Evaluation wird anhand eines konkreten Anwendungsfalls durchgeführt, der einen Teilaspekt des in Abschnitt 3.2.4 beschriebenen Anwendungsfalls darstellt. Dort wird das Sensornetz unter anderem für eine Anwendung verwendet, die die Temperatur eines Raumes nach den Vorgaben der Bewohner regelt. An einer solchen Anwendung soll hier die Evaluation durchgeführt werden.

Der Ablauf der Evaluation erfolgt in den folgenden Schritten:

1. Entwurf und Modellierung eines DDS-Systems für den Anwendungsfall
2. Verwendung von OpenSplice DDS für die Implementierung der Anwendung auf einer Linux-PC-Plattform
3. Definition des Entwurfs und Modells des DDS-Systems mit der DSL von sDDS

4. Erzeugung einer sDDS-Middleware-Implementierung für eine Linux-PC-Plattform
5. Implementierung der Anwendung für das Linux-PC-sDDS und Vergleich des Vorgangs mit OpenSplice DDS
6. Erzeugung einer sDDS-Middleware-Implementierung für die CC2430-Sensorknoten-Hardware-Plattform
7. Anpassung der Anwendung für die Sensorknotenplattform CC2430
8. Bewertung der Evaluation und des gewählten Ansatzes in dieser Arbeit

6.2 Evaluationsanwendung

6.2.1 Anwendungsfall

Aus dem in Abschnitt 3.2.4 beschriebenen Anwendungsfall für eine AAL-Umgebung dient der Teil der Temperaturregelung von Räumen hier als Vorlage für die Evaluation von sDDS.

Der hier betrachtete Anwendungsfall geht von einem Sensornetz mit zwei Sensorknoten aus. Ein Sensorknoten verfügt über einen Temperatursensor und misst kontinuierlich die Raumtemperatur. Diese Werte werden von dem zweitem Sensorknoten benötigt, der über einen Aktor mit einer Heizung verbunden ist. Er verwendet die Temperaturwerte, um über einen Regelungsalgorithmus die Raumtemperatur der Soll-Temperatur anzunähern. Es wird vereinfachend angenommen, dass die Heizung nur ein- oder ausgeschaltet werden kann, daher wird als Regelungsalorithmus ein Zweipunktregler verwendet, um die Temperatur auf dem Soll-Wert zu halten.

In Abbildung 6.1 ist das UML-Composite-Diagramm für diesen Anwendungsfall dargestellt. Der Sensorknoten *TemperaturSensor* veröffentlicht die Temperaturdaten. Der zweite Sensorknoten *TemperaturRegler* abonniert diese Daten, und das Sensornetz verbindet diese beiden Komponenten.

Es soll im Weiteren angenommen werden, dass auf jedem Sensorknoten eine Anwendung läuft, die die jeweilige Aufgabe durchführt und denselben Namen verwendet wie die Sensorknoten.

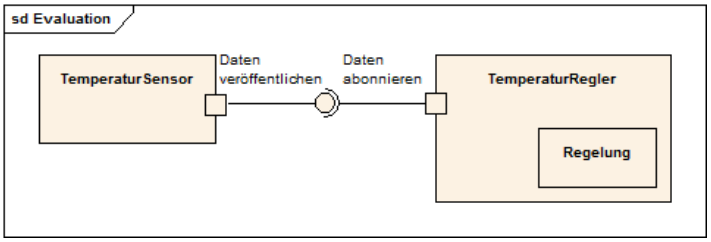

Abbildung 6.1: UML-Composite-Diagramm des Anwendungsfalls zur Temperaturregelung

6.2.2 DDS-Modell

Das DDS-System wird dazu verwendet, die Temperaturdaten zwischen den Anwendungen der Sensorknoten auszutauschen. Da dies ein Teil des AAL-Anwendungsfalls aus Abschnitt 3.2.4, ist wird als Name der DDS-Domain *AAL* verwendet. Die Struktur der auszutauschenden Temperaturdaten wird mit dem Topic-Datentyp *Temperatur* festgelegt und die beiden Sensorknoten des Anwendungsfalls verwenden das DDS-Topic *TempControl*, um die Daten zu referenzieren.

Die Anwendung *TemperaturSensor* benötigt eine DomainParticipant-, Publisher- und DataWriter-Instanz, um Temperaturdaten im sDDS-System veröffentlichen zu können. Der DataWriter ist dementsprechend mit dem Topic *TempControl* verbunden. Auf dem zweiten Sensorknoten läuft die Anwendung *TemperaturRegler*, die eine DomainParticipant-, Subscriber-, und DataReader-Instanz benötigt. Analog ist der DataReader ebenfalls dem Topic *TempControl* zugeordnet.

6.3 OpenSplice DDS-basierte Implementierung

Für die Implementierung der Anwendung wird das in Abschnitt 5.2 beschriebene Entwicklungssystem auf der Linux-PC-Plattform verwendet und OpenSplice DDS in der Version 5.1 für die AMD64-Architektur eingesetzt. Für den Anwendungsfall der Temperaturregelung wurden zwei Anwendungen mit ANSI-C implementiert, die jeweils die in Abschnitt 6.2.1 beschriebenen Aufgaben übernehmen. Als Temperatursensor wird ein USB-Thermometer verwendet und die Steuerung der Heizung erfolgt virtuell über eine Textausgabe.

Die DDS-Anwendungen müssen, bevor sie Daten über OpenSplice austauschen können, die DDS-Struktur über die Instanziierung und die Konfiguration der DDS-

Objekte aufbauen. Dieser Vorgang ist für die Temperatursensor-Anwendung im Listing 6.1 dargestellt.

```
1
2  static DDS_DomainId_t aalDomain = DDS_OBJECT_NIL;
3  static DDS_DomainParticipantFactory dpf = DDS_OBJECT_NIL;
4  static DDS_DomainParticipant dp = DDS_OBJECT_NIL;
5  static DDS_Publisher pub = DDS_OBJECT_NIL;
6  static DDS_Topic topic = DDS_OBJECT_NIL;
7  static Eval_TemperaturDataWriter dw = DDS_OBJECT_NIL;
8
9  static Eval_TemperaturTypeSupport tempTypeSup = DDS_OBJECT_NIL;
10 static DDS_string tempTypeSupName;
11
12
13 int initDDS(){
14
15     dpf = DDS_DomainParticipantFactory_get_instance();
16
17     dp  = DDS_DomainParticipantFactory_create_participant(dpf, aalDomain,
18                 DDS_PARTICIPANT_QOS_DEFAULT, NULL, DDS_STATUS_MASK_NONE);
19
20     pub = DDS_DomainParticipant_create_publisher(dp, DDS_PUBLISHER_QOS_DEFAULT,
21             NULL, DDS_ANY_STATUS);
22
23     tempTypeSup = Eval_TemperaturTypeSupport__alloc();
24
25     tempTypeSupName = Eval_TemperaturTypeSupport_get_type_name(tempTypeSup);
26
27     Eval_TemperaturTypeSupport_register_type(tempTypeSup, dp, tempTypeSupName);
28
29     topic = DDS_DomainParticipant_create_topic(dp, "TOPICNAME", tempTypeSupName,
30             DDS_TOPIC_QOS_DEFAULT, NULL, DDS_ANY_STATUS);
31
32     dw = DDS_Publisher_create_datawriter(pub, topic, DDS_DATAWRITER_QOS_DEFAULT,
33             NULL, DDS_ANY_STATUS);
34 }
```

Quellcode 6.1: Initialisierung des OpenSplice DDS-Systems für die Temperatursensor-Anwendung der Evaluation

Der gezeigte Programmcode enthält nur die Aufrufe der DDS-Funktionen; die Überprüfung auf Fehler wurde zur besseren Übersichtlichkeit weggelassen. Des Weiteren wird die Standardkonfiguration der QoS-Richtlinien der einzelnen DDS-Objekte verwendet, wie sie im DDS-Standard definiert ist.

Die Konfiguration der DDS-Domain erfolgt bei OpenSplice DDS über eine XML-Datei, die den im Hintergrund laufenden OpenSplice-Dienst initialisiert. Für diese Evaluation wird die Standard-DDS-Domain von OpenSplice verwendet, die nicht

konfiguriert werden muss, daher wird der Methode `DDS_DomainParticipant Factory_create_participant()` ein gekapselter Void-Pointer übergeben. Nach dieser Initialisierung kann die Anwendung auf den angelegten DataWriter *dw* zugreifen.

6.4 sDDS-basierte Implementierung

6.4.1 Modellierung der Anwendungsanforderungen

Die Temperaturkontrollanwendungen, die für OpenSplice DDS implementiert wurden, bauen das DDS-System zur Laufzeit auf und konfigurieren es. Der Programmcode, der dies umsetzt, musste manuell implementiert werden. Ein Anspruch dieser Arbeit ist es, diesen Vorgang automatisch über die Generierung des Programmcodes auszuführen.

Der erste Schritt für die Entwicklung einer Anwendung mit sDDS ist es, das DDS-System mit der sDDS-DSL zu definieren. Hierfür wurde der von Xtext generierte Eclipse-Editor verwendet, der während der Eingabe die Syntax der DSL überprüft und Syntax-Fehler mit Fehlerbeschreibungen meldet, die sich auf das Metamodell beziehen. Das erstellte Modell ist in Listing 6.2 dargestellt.

```
1  namespace Eval
2
3  IDL {
4    struct Temperatur {
5      short temp;
6      octet ID;
7      unsigned short position;
8    };
9    #pragma keylist
10 }
11
12 domain AAL {
13   topic TempControl {
14     dataType Temperatur
15   }
16 }
17
18 application TemperaturSensor {
19   domain AAL {
20     publisher pub {
21       datawriter dw {
22         topic TempControl
```

```
23        }
24      }
25    }
26  }
27
28  application TemperaturRegler {
29    domain AAL {
30      subscriber sub {
31        datareader dr {
32          topic TempControl
33        }
34      }
35    }
36  }
```

Quellcode 6.2: Modell der Evaluationsanwendungen mit der sDDS-DSL

Das Modell entspricht dem in Abschnitt 6.2.2 beschriebenen System. Es gibt eine Domain „AAL" mit dem Topic *TempControl*, das den in OMG IDL definierten Topic-Datentyp *Temperatur* referenziert. Die beiden Anwendungen des Anwendungsfalls sind separat beschrieben und entsprechen in ihrem Aufbau dem in Abschnitt 6.2.2 definierten Modell. Da die sDDS-Implementierung bisher QoS-Richtlinien nicht vollständig unterstützt, wurden diese nicht modelliert. Damit entspricht die Konfiguration ungefähr der Grundkonfiguration, die der DDS-Standard vorsieht.

Es zeigt sich, dass die Modellierung des Systems direkt unter Verwendung der DSL durchgeführt werden kann. Wenn zukünftige Erweiterungen von sDDS als DSL auch das UML-Profil von DDS unterstützen, kann dieser Teil der Modellierung auch mit grafischen UML-Werkzeugen durchgeführt werden. Zumindest ist es wünschenswert, dass der generierte Editor um weitere Rückmeldungen über die Ergebnisse einer semantischen Überprüfung des Modells erwartet wird und somit den Modellierungs- und Entwicklungsvorgang von sDDS-Anwendungen weiter vereinfachen kann.

6.4.2 Linux-PC-Version

Nach der Modellierung des sDDS-Systems für die Anwendungen kann der Programmcode für die angepassten anwendungsspezifischen Teile generiert werden. Dieser Vorgang erfordert manuelle Eingriffe, da der Editor für die sDDS-DSL noch nicht mit dem Generierungs-Framework verbunden ist. Die Datei, in der das Modell gespeichert ist, muss hierfür in dem „Workflow" des Codegenerators angegeben werden (Siehe Abschnitt 5.6.5 der Implementierung).

Für die Linux-PC-Version wird durch das Framework ein Anwendungsskelett generiert, in dem an geschützten Stellen, die nicht von einer erneuten Generierung auf Grund einer ggf. iterativen Anpassung des Modells überschrieben werden können, die Anwendungslogik eingefügt werden kann.

Anders als bei OpenSplice DDS kann sofort begonnen werden, die Anwendungs-

da hierbei noch die Bibliotheken für die Ansteuerung des USB-Thermometers hinzukommen. Die verwendeten POSIX- und Linux-spezifischen Funktionen werden bei beiden über dynamische Bibliotheken eingebunden.

6.4.3 CC2430-Version

Eine erneute Generierung der sDDS-Middleware-Implementierung für die Plattform des CC2430 ist nicht notwendig, da alle plattformspezifischen Komponenten von sDDS in eigenen Implementierungsdateien gekapselt sind. Da der Generierungsprozess noch nicht in die verschiedenen Entwicklungsumgebungen eingebunden ist, ist es allerdings notwendig, die plattformspezifischen Quellcode-Dateien manuell für die richtige Plattform auszuwählen.

Die implementierte Anwendungslogik kann blockweise in die CC2430-Version übernommen werden, da die instantiierten DDS-Schnittstellen identisch sind. Die Teile, die den plattformspezifischen Zugriff auf die Sensoren und Aktoren realisieren muss aber angepasst werden.

Die Implementierung von Anwendungen für die CC2430-Version erfolgt innerhalb der IAR-Entwicklungsumgebung. Der erste Schritt dazu ist es, ein existierendes sDDS-Projekt zu kopieren und umzubenennen. Danach werden die generierten Dateien, mit Ausnahme des Anwendungsskeletts, in das Projekt importiert.

In einem IAR-Projekt können verschiedene Ziele für die Anwendung getrennt verwaltet werden. Damit ist es möglich, die beiden Anwendungen in dasselbe Projekt einzufügen, jeweils für einen Sensorknoten zu erzeugen und zu installieren.

Als Temperatursensor wird der intern in der CC2430-Plattform eingebaute verwendet und der Aktor zur Heizungssteuerung über eine LED simuliert.

Die Implementierung der Anwendungen für diese Plattform ist auf Grund der schlechten Integration in die IAR-Entwicklungsplattform noch sehr mühsam. Es sind viele manuelle Anpassungen notwendig. Dennoch zeigt sich, dass die Portierung einer sDDS-Anwendung zwischen verschiedenen Plattformen auf Grund der identisch generierten Schnittstelle einfacher möglich ist, als zum Beispiel die Portierung von OpenSplice DDS-Anwendungen nach sDDS.

Mit Ausnahme der auf Grund der unvollständigen, prototypischen Implementierung etwas umständlichen Einrichtung eines neuen sDDS-Projekts in der IAR-Entwicklungsumgebung ist die Realisierung einer Anwendung mit sDDS mit

überschaubarem Aufwand verbunden. Eine bessere Integration in den MDSD-basierten Generierungsprozess kann dies noch weiter vereinfachen.

6.4.4 Erweiterung der Anwendungen

Als nächstes soll betrachtet werden, wie hoch der Aufwand für eine Erweiterung der bestehenden Anwendungen wäre. Im Lebenszyklus einer Anwendungen sind Erweiterung häufig erforderlich, wenn sich mit der Zeit neue Anforderungen ergeben. Dies soll lediglich theoretisch betrachtet und nicht über eine reale Implementierung validiert werden.

Es wird daher den OpenSplice DDS- und sDDS-Anwendungen für den „TemperaturSensor"-Knoten jeweils ein neuer DataWriter hinzugefügt, der die Temperaturwerte für eine Branderkennungsanwendung bereitstellt. Diese Branderkennungsanwendung ist nicht Teil des AAL-Kontextes und verwendet daher die DDS-Domain „Sicherheit", um sicherheitskritische Sensordaten von normalen zu trennen. Auf Grund der neuen Domain benötigt jede „TemperaturSensor"-Anwendung einen neuen DomainParticipant und Publisher.

6.4.4.1 OpenSplice DDS

Für OpenSplice bedeutet die Notwendigkeit, zwei DDS-Domains in einer Anwendung verwenden zu müssen, dass zwei getrennte XML-Konfigurationsdateien für die Domains erstellt und angepasst werden müssen. Die Pfade zu diesen Dateien müssen in der Anwendung bekannt sein und bei der Initialisierung des DomainParticipant mit übergeben werden. Darüber hinaus ist es notwendig, dass für jede Domain ein eigener OpenSplice-Daemon läuft.

Die Verwendung des zusätzlichen DataWriter macht es notwendig, dass der Initialisierungscode der OpenSplice DDS-Anwendung um Methodenaufrufe für das Anlegen eines Publishers und DataWriters erweitert wird. Da die Temperaturdaten für eine sicherheitskritische Anwendung verwendet werden sollen ist es wahrscheinlich, dass diese um weitere Konfiguration für spezifische QoS-Richtlinien erweitert werden muss.

Der Aufwand für eine Erweiterung ist im Bereich der Initialisierung des DDS-Systems relativ hoch. Dafür ist es möglich, die Domain für sicherheitskritische Auf-

gaben stärker von dem AAL-System zu trennen. Es könnten auch verschiedene Netzwerke für die Übertragung der Daten verwendet werden.

6.4.4.2 sDDS

Eine Erweiterung des DDS-Systems bedeutet für eine Anwendung, dass das DDS-Modell erweitert werden muss. Hierfür würde in diesem Fall der Anwendung „TemperaturSensor" eine weitere Domain mit einem Publisher und DataWriter hinzugefügt. Die Programmcodegenerierung modifiziert den Programmcode der ursprünglichen sDDS-Implementierung und fügt ihm die neue Funktionalität hinzu. Sofern die Anwendung in das generierte Anwendungsskelett eingefügt wurde, wird dieses so angepasst, dass der neue DataWriter zur Verfügung steht. Der bisherige Programmcode wird nicht verändert und kann einfach über das Einfügen der Anwendungslogik für den zweiten DataWriter angepasst werden.

Wenn sDDS QoS-Richtlinien unterstützen würde, die von einer sicherheitskritischen Anwendung benötigt würden, dann würde ihre Konfiguration im DDS-Modell geschehen. Der Generierungsprozess würde die entsprechende QoS-Implementierung in die sDDS-Middleware-Implementierung einfügen und direkt konfigurieren. Für den Anwendungsentwickler wäre dieser Vorgang transparent, sofern er die QoS-Konfiguration nicht zur Laufzeit verändern will.

Für sDDS ist eine Domain nur ein einfacher Namensraum; der Datenverkehr wird nicht vollständig getrennt wie bei OpenSplice DDS. Dies hat den Vorteil, dass die Umsetzung leichtgewichtig möglich ist, aber es ist in dieser Ausbaustufe von sDDS nicht möglich, sicherheitskritische Daten auf der Netzwerkebene von normalen Daten zu trennen.

6.5 Zusammenfassung und Bewertung

Für die Evaluation anhand eines Teil des AAL-Anwendungsfalls wurde der Software-Entwicklungsprozess für die Implementierung einer Anwendung auf Basis einer DDS-Middleware dreimal durchlaufen. Als erstes wurden mit OpenSplice DDS die beiden Anwendungen implementiert und danach dieser Prozess mit sDDS für die Linux-PC- und 2430-Sensorknotenplattform wiederholt. Im Anschluss wurde theoretisch betrachtet, welcher Implementierungsaufwand für eine spätere Erweiterung einer der Anwendungen entstehen würde.

Es zeigt sich, dass die Generierung der anwendungsspezifischen sDDS-Implementierung den generellen Vorteil hat, dass es nicht notwendig ist, das DDS-System manuell über das Anlegen von DDS-Objekten aufzubauen. Auch Änderungen im DDS-Modell können direkt in angepassten Programmcode umgesetzt werden, die der Anwendungsentwickler sofort verwenden kann.

Die Portierung von sDDS-Anwendungen auf eine neue Plattform ist relativ einfach, sofern dasselbe DDS-Model verwendet wird. Dies wird in der aktuellen prototypischen Implementierung dadurch eingeschränkt, dass die plattformspezifischen Entwicklungsumgebungen nicht in den Generierungsprozess integriert sind, und daher manuelle Anpassungen notwendig sind.

Während der Durchführung der Evaluation zeigten sich an vielen Stellen Programmfehler und nicht berücksichtigte Sonderfälle. Da das MDSD-basierte Middleware-Framework selber modellgetrieben entwickelt wird, war es sehr schnell möglich, notwendige Änderungen zu modellieren, das Framework anzupassen und die Änderungen zu testen. Dieser MDSD-Ansatz hat sicherlich maßgeblich dazu beigetragen, dass die Generierung einer angepassten sDDS-Middleware in diesem Umfang umgesetzt werden konnte.

Die Evaluation zeigt, dass der gewählte Ansatz grundsätzlich geeignet ist und helfen kann, die Anforderungen, die in Abschnitt 3.1.6 aufgestellt wurden, zu erfüllen. Obwohl die Implementierung von sDDS in dieser Arbeit nur fragmentarisch durchgeführt werden konnte, ist es mit kleineren Einschränkungen möglich, den vollständigen Prozess für die Generierung einer anwendungsspezifisch erzeugten sDDS-Middleware zu durchlaufen.

Kapitel 7

Zusammenfassung und Ausblick

7.1 Erreichte Ziele

In dieser Arbeit wurde untersucht, wie Software-Entwicklung auf der Basis einer leichtgewichtigen, datenzentrierten Middleware für drahtlose Sensornetze vereinfacht werden kann. Anwendungen für drahtlose Sensornetze sind auf Grund der Ressoucenbeschränktheit stark an die jeweilige Zielplattform, die Middleware und das Kommuniktionssystem angepasst, was einen hohen Aufwand für die Entwicklung, Portierung und Erweiterung verursacht. Daher wurde hier ein modellgetriebener Software-Entwicklungsansatz (MDSD) konzipiert und prototypisch realisiert, mit dem eine datenzentrierte Middleware an Anwendungsanforderungen und Sensornetz-Plattformen angepasst werden kann und der in dieser Weise noch nicht in der Literatur beschrieben wurde.

In dem modellgetriebenen Software-Entwicklungsprozess werden die Anforderungen der Anwendungen an die Middleware, die benötigte Funktionalität und Eigenschaften der Zielplattform als Vorgabe modelliert und daraus eine Middleware-Implementierung generiert, die nur die benötigte Funktionalität enthält, effizient auf die jeweilige Plattform angepasst ist und den besonderen Ressourcenanforderungen von drahtlosen Sensornetzen gerecht wird. Der Anwendungsentwickler wird in diesem Prozess durch Rückmeldungen bezüglich der Konsistenz seines Modells unterstützt.

Damit Anwendungen auch außerhalb der hier entwickelten Umgebung portabel sein können, wurde der Data Distribution Service der OMG als Schnittstellenstandard für eine datenzentrierte Middleware ausgewählt, der auch eine Reihe

von QoS-Richtlinien definiert, die für das Sensornetz-Umfeld interessant sind. Der DDS-Standard wurde allerdings nicht für drahtlose Sensornetze spezifiziert; existierende vollständige DDS-Implementierungen haben daher einen für eingebettete Anwendungen verhältnismäßig hohen Speicherbedarf, basieren auf IP als Kommunikationsschnittstelle und sind daher nicht für drahtlose Sensornetze geeignet. Für diese Arbeit wurde deshalb eine neue DDS-Implementierung für drahtlose Sensornetze entworfen, für die mit dem in dieser Arbeit entwickelten MDSD-Prozess in anwendungs- und plattformspezifischer Weise Code generiert werden kann.

Für die in dieser Arbeit entworfene und prototypisch realisierte Middleware, die den Namen sDDS (sensor network DDS) hat, wurde ein geringer Ressourcenverbauch durch einen reduzierten Funktionsumfang erreicht, der modular gestaltet wurde, so dass die Funktionalität auswählbar ist. Des Weiteren wurde ein neues DDS-Kommunikationsprotokoll (SNPS) für diese Arbeit entworfen, auf dem die sDDS-Middleware aufsetzt. SNPS ist für die Verwendung in drahtlosen Sensornetzen und damit auf kleine Netzwerkpaketgrößen für häufig vorkommende Operationen der Middleware optimiert. Funktionalität in den SNPS-Nachrichten, die ein Knoten auf Grund des modularen Charakters der einzelnen sDDS-Implementierungen nicht versteht, ist für ihn transparent. Das SNPS-Protokoll definiert eine abstrakte Kommunikationsschnittstelle für ihm unterlagerte Kommunikationssysteme, so dass es selber plattformunabhängig ist.

Die Ausgangsbasis für den MDSD-Prozess zur Generierung der angepassten sDDS-Middleware-Implementierung sind die in dieser Arbeit entworfenen Metamodelle für das DDS-System und die plattform- und anwendungsspezifische Konfiguration, die mittels einer sDDS-spezifischen „Domain Specific Language" für die Modellierung der Anwendungsanforderungen verwendet werden. Diese Modelle bieten das Potenzial, mit weiterem Domänenwissen über die sDDS-Funktionalität und die Zielplattform auf ihre Konsistenz überprüft und über Transformationen auf geringen Speicherverbrauch optimiert und in plattformspezifische Modelle überführt zu werden. Die Programmcodegenerierung verwendet diese Modelle, um eine angepasste sDDS-Middleware-Implementierung zu generieren, wofür im Rahmen der vorliegenden Arbeit Programmcode-Templates verwendet werden, die aus der prototypischen Implementierung von sDDS entstanden sind. Die vollständige Umsetzung der Prüf- und Transformationsschritte des Prozesses war zu aufwendig, um im Rahmen dieser Arbeit implementiert werden zu können, daher wurde er auf die grundlegenden Schritte reduziert.

Die Programmiersprache ANSI-C wurde für die prototypische Implementierung verwendet und ermöglichte es, die zu implementierenden Komponenten in universellen, anwendungs- und plattformspezifischen Teilen zu realisieren. Dies vereinfacht eine Implementierung der sDDS-Middleware für verschiedene Plattformen, die prototypisch für zwei sehr unterschiedliche Systemplattformen durchgeführt wurde und die Basis für die Evaluation der in dieser Arbeit gewählten Ansätze bildet. Die erste Version von sDDS wurde für Linux-PC-Systeme entwickelt und verwendet UDP/IP als unterlagerte Kommunikationsschnittstelle. Als Sensorknotenplattform für die zweite Version von sDDS wurde die System-on-a-Chip-Lösung CC2430 von TI verwendet sowie ZigBee, um im drahtlosen Sensornetz Daten auszutauschen.

Die Evaluation der hier durchgeführten Arbeiten wurde an einem einfachen Anwendungsfall aus dem „Ambient Assisted Living"-Bereich (AAL) durchgeführt. Dieses Forschungfeld wird aktuell im „Labor für Verteilte Systeme" der Hochschule RheinMain mit unterschiedlichen Projekten bearbeitet und stellt den konkreten Kontext dieser Arbeit dar. Für die Evaluation wurde der Software-Entwicklungsprozess mit den prototypisch realisierten Werkzeugen und Komponenten durchlaufen und auch die Portierbarkeit von Anwendungen untersucht, die auf Basis einer anderen DDS-Implementierung entwickelt wurden. Als Ergebnis lässt sich feststellen, dass der gewählte Ansatz vielversprechend erscheint und Potenzial für zukünftige Erweiterungen und Projekte bietet.

7.2 Vorgehen / Durchgeführte Arbeiten

Um das Ziel dieser Arbeit zu erreichen, wurde zuerst in Abschnitt 3.1 anhand eines generischen Anwendungsfalls für drahtlosen Sensornetze (Abschnitt 3.1.2) untersucht, welche Anforderungen an eine Middleware gestellt werden können. Diese Anforderungen wurden anhand aktueller Übersichtsliteratur in 3.1.4 überprüft, in 3.1.5 erweitert und in Abschnitt 3.1.6 eine für diese Arbeit relevante Auswahl festgelegt. Eine wichtige Anforderung ist es dabei, dass das Verfahren der modellgetriebenen Software-Entwicklung eingesetzt werden soll. Der DDS-Standard wurde in Abschnitt 3.2 auf seine Eignung für diese Arbeit untersucht und als Ergebnis seine grundsätzliche Eignung festgestellt (Abschnitt 3.2.1), aber es war notwendig, für die weitere Analyse in 3.2.3 zusätzliche Fragestellungen zu

definieren und zu untersuchen, wie der Ressourcenverbrauch reduziert und nur die benötigte Funktionalität in einer Implementierung eingebunden werden kann.

Für die weitergehende Analyse von DDS wurde in Abschnitt 3.2.4 ein DDS-spezifischer Anwendungsfall aus dem AAL-Kontext beschrieben, der die Basis für die weiteren Untersuchungen bildete. Dazu wurde in Abschnitt 3.3 das Datenmodell und in 3.4 die für diese Arbeit relevante DDS-Funktionalität daraufhin untersucht, wie der Speicherverbrauch reduziert und wie die Funktionalität von DDS für den MDSD-Prozess modularisiert werden kann. Da für diese Arbeit das ZigBee-Protokoll verwendet werden sollte, wurde in Abschnitt 3.5.2 analysiert, wie DDS auf ZigBee abbildbar ist und welche Anforderungen ein Kommunikationsprotokoll für sDDS erfüllen sollte (in 3.5.1). Für den zu verwendenden MDSD-Prozess wurden in Abschnitt 3.6.3 State-of-the-Art-Werkzeuge des Eclipse-Projekts untersucht und nach Beispielen für eine übliche Vorgehensweise eines MDSD-Prozesses für einen Anwendungsfall wie in dieser Arbeit gesucht.

Die Untersuchungen zeigten generell, dass der gewählte Ansatz eine Vielzahl von Möglichkeiten für die modellgetriebene Entwicklung einer DDS-Implementierung bietet, aber es war aus Zeitgründen nicht möglich, alle Aspekte in dieser Arbeit zu betrachten. Daher war es notwendig, sich auf die Bereiche zu konzentrieren, die notwendig sind, um den gewählten Ansatz auf seine Tauglichkeit evaluieren zu können (siehe Abschnitt 3.7 über die getroffenen Designentscheidungen).

In Kapitel 4 wurden die sDDS-Middleware und das MDSD-basierte Framework, das sie erzeugen soll, entworfen. Die in Abschnitt 4.1.1 dargestellte Architektur des sDDS-Middleware-Frameworks basiert auf den untersuchten State-of-the-Art-Werkzeugen des Eclipse-Projektes, die damit den weiteren Ablauf der Implementierung des Frameworks vorgaben. Die benötigten Metamodelle für die Modellierung des DDS-Systems und der anwendungs- bzw. plattformspezifischen Konfiguration wurden in Abschnitt 4.2 und der Software-Entwicklungsprozess in Abschnitt 4.3 beschrieben.

Die sDDS-Middleware wurde unter dem Aspekt der Tauglichkeit für Sensornetze und der Vorgabe konzipiert, dass ihre Implementierung anwendungs- und plattformspezifisch generiert werden soll. Die in Abschnitt 4.5 beschriebene Architektur verwendet nicht die Struktur des DDS-Standards, sondern orientiert sich in ihrem Aufbau an den typischen Rollen in einem Sensornetzwerk. In Abschnitt 4.4 wurde das SNPS-Protokoll für die sDDS-Middleware definiert, das unabhän-

gig von dem unterlagerten Kommunikationssystem ist, sofern es die in Abschnitt 3.5.4 der Analyse beschriebene Schnittstelle implementieren kann. Es zeigte sich auch in der Entwurfsphase, dass in dem zeitlichen Rahmen dieser Arbeit nur ein Ausschnitt, der die Basisfunktionalität von DDS umfasst, realisiert werden kann.

Die Umsetzung des Entwurfs in eine prototypische Implementierung von sDDS-Middleware und dem MDSD-basierten Generierungs-Framework wird in Kapitel 5 beschrieben. Als Programmiersprache wurde ANSI-C verwendet, da es hierfür eine standardisierte Abbildung der DDS-Schnittstelle gibt. Für die Implementierung der sDDS-Middleware wurde eine Trennung der Komponenten von sDDS in universelle, plattform- und anwendungsspezifische Teile vorgenommen. Der in Abschnitt 5.3 beschriebene universelle Teil von sDDS verwendet nur den C-Sprachumfang und wird über C-Präprozessordirektiven mit anwendungsspezifischer Konfiguration angepasst. Die plattformspezifischen Teile implementieren abstrakte Schnittstellen und sind damit unabhängig von dem universellen Teil. Es wurden zwei sDDS-Versionen realisiert: eine für die Linux-PC-Plattform beschrieben in Abschnitt 5.4 und eine für die CC2430-Plattform von TI (in Abschnitt 5.5). Die Implementierung des MDSD-Prozesses wird in Abschnitt 5.6 beschrieben. Abgeschlossen wird die Implementierung mit der Evaluation des gewählten Ansatzes in Abschnitt 6 anhand eines Anwendungsfalls aus dem AAL-Umfeld.

7.3 Ausblick

Die Evaluation hat gezeigt, dass der gewählte Ansatz für das Ziel, die Software-Entwicklung in Sensornetzen zu vereinfachen und Anwendungen unabhängiger von der jeweiligen Zielplattform implementieren zu können, geeignet ist. Während der Analyse-, Entwurfs- und Implementierungsphasen musste der Umfang aber kontinuierlich reduziert werden, da die Bearbeitungszeit, der Master-Thesis nicht ausreichend war, um alle Aspekte, die sich aus dem gewählten Ansatz ergaben, weiterverfolgen zu können. Damit verbleiben entsprechend viele „weiße Flecken”, die mit zukünftigen Arbeiten gefüllt werden können.

Der erste Schritt für eine Erweiterung des sDDS-Middleware-Prototypen sollte es sein, die im Entwurf definierten QoS-Richtlinien und die Unterstützung von Gruppenkommunikation zu realisieren. Dies kann zum einem in Form von eigenständigen, netzwerkunabhängigen Komponenten geschehen. Eine weitere Option ist

es zu untersuchen, wie diese Eigenschaften auf andere Kommunikationssysteme abgebildet werden können oder wie ihre Funktionalität für die Realisierung der QoS-Komponenten verwendet werden kann.

Die Evaluation des prototypischen, MDSD-basierten Middleware-Frameworks hat gezeigt, dass dieser Prozess großes Potential hat, die Entwicklung von Software unter Verwendung von DDS nicht nur für Sensornetze zu vereinfachen. Die nächsten Arbeiten sollten sich daher mit der weitergehenden Integration der bisher noch alleinstehenden Prozessschritte befassen und die Entwicklungsumgebungen der verschiedenen Plattformen direkt in den Generierungsprozess einbinden. Auch der im Entwurf des Frameworks beschriebene interaktive Entwicklungsprozess sollte über die Erweiterung des von Xtext generierten Editors umgesetzt werden.

Die Analyse in Abschnitt 3.4.1 hat gezeigt, dass eine tiefere Analyse der DDS-Funktionalität und der Abhängigkeit von DDS-Komponenten untereinander oder von einer spezifischer Konfiguration notwendig ist, damit sich die Generierung einer DDS-Implementierung auf die von Anwendungen benötigte Funktionalität beschränken kann. Die Ergebnisse dieser Untersuchung können direkt in das sDDS-Middleware-Framework übernommen werden.

Langfristiges Thema für die Erweiterung von sDDS wird der schrittweise Ausbau der Unterstützung der in Abschnitt 3.4.1 als für Sensornetze relevant angesehenen DDS-Funktionalität sein. Die bisher umgesetzte Teilmenge dieser DDS-Funktionalität bietet ein gutes Grundgerüst, damit weitere Funktionen iterativ hinzugefügt werden können.

Um dem Anwendungsentwickler zusätzliche Flexibilität für die Modellierung des sDDS-Systems bieten zu können, sollten zukünftige Erweiterungen weitere DSLs unterstützen, wie es in Abschnitt 4.1.1 entworfen wurde. Mit der entsprechenden Wahl der DSLs kann auch ermöglicht werden, Modelle mit anderen Anwendungen auszutauschen. Auch die vorgesehenen Modelltransformationen sollten ausgebaut und in separate Verarbeitungsschritte untergliedert werden. Die Entwicklung eines Metamodells für plattformspezifische Informationen kann mit generischen Anforderungen des sDDS-Systems in einer Contraint-Sprache dazu verwendet werden, Modelle auf Konsistenz und Einhaltung der aus der Zielplattform gegebenen Bedingungen zu überprüfen. Die plattformspezifischen Modelle können darüber hinaus für eine generische PIM2PSM-Transformation eingesetzt werden.

Das SNPS-Protokoll wurde von Beginn an auf Erweiterbarkeit ausgelegt, und der definierte Funktionsumfang sollte auch für Erweiterungen der DDS-Funktionalität ausreichend sein. In Abschnitt 4.4 wurde die Möglichkeit angesprochen, das SNPS-Protokoll um Routing-Funktionalität zu erweitern. Dies würde es der sDDS-Middleware erlauben, viel umfassender die Verteilung von Daten und die Durchsetzung von QoS-Richtlinien zu optimieren. Da dieses Thema eine wichtige Position in der aktuellen Forschung im Bereich von Sensornetzen einnimmt, ist die Erweiterung von sDDS um diesen Aspekt sehr interessant.

Die Entwicklung der beiden Versionen des sDDS-Prototypen hat gezeigt, dass neue Hardware- und Kommunikationssystemplattformen mit überschaubarem Aufwand unterstützt werden können. Daher sollte ein zukünftiger Fokus die Unterstützung weiterer Plattformen sein. Diese größere Basis an Systemen könnte dann für weitergehende Untersuchungen und Tests des Ansatzes verwendet werden. Das Protokoll SNPS ermöglicht kompaktere Nachrichten als für ZigBee notwendig. Damit ist es möglich, auch weitere in Abschnitt 3.3.2.3 aufgeführte Netzwerksysteme zu unterstützen, wie beispielsweise KNX oder ggf. auch CAN.

Offensichtlich bietet diese Arbeit eine große Anzahl an Ansatzpunkten für zukünftige Arbeiten und Erweiterungen. Die Fortführung des gewählten Ansatzes erscheint insofern lohnenswert, als das die zunehmende Verbreitung von heterogener Sensornetztechnologie in immer mehr Anwendungsbereichen, wie dem AAL-Umfeld, den Bedarf nach einem ganzheitlichen Software-Entwicklungsprozess erzeugt, wie er in dieser Arbeit erarbeitet wurde, und es ermöglicht, verschiedene Systeme über eine einheitliche Anwendungsschnittstelle über Plattformgrenzen zu verbinden, und die Software-Entwicklung von angepasster Software zu vereinfachen.

Kapitel 8

Literaturverzeichnis

[Ada06] ADAMS, Jon T.: An introduction to IEEE STD 802.15.4. In: *Aerospace Conference, 2006 IEEE*, 2006, S. 8 pp.

[ADBS09] AKBAL-DELIBAS, Bahar ; BOONMA, Pruet ; SUZUKI, Junichi: Extensible and Precise Modeling for Wireless Sensor Networks. In: *UNISCON*, 2009, S. 551–562

[All08] ALLIANCE, ZigBee: *ZigBee Specification*. Document 053474r17. Sam Ramon, USA: ZigBee Standards Organization, jan 2008

[Ass09] ASSOCIATION, KNX: *KNX System Specification Architecture*. v3.0. Brüssel, Belgien: KNX Association, jun 2009

[ASSC02] AKYILDIZ, I. F. ; SU, W. ; SANKARASUBRAMANIAM, Y. ; CAYIRCI, E.: Wireless sensor networks: a survey. In: *Computer Networks* 38 (2002), Nr. 4, 393 - 422. – ISSN 1389–1286

[Atm09a] ATMEL: *ATMEL ARM920T-based Microcontroller AR91RM9200*. 1768I-ATARM–09-Jul-09. San Jose, USA: Atmel Corporation, jul 2009

[Atm09b] ATMEL: *Datenblatt ATmega128 und ATmega128L*. Rev. 2467S-AVR-07/09. San Jose, California, USA: Atmel Corporation, 07 2009

[BAB+07] BAKER, Chris R. ; ARMIJO, Kenneth ; BELKA, Simon ; BENHABIB, Merwan ; BHARGAVA, Vikas ; BURKHART, Nathan ; MINASSIANS, Artin D. ; DERVISOGLU, Gunes ; GUTNIK, Lilia ; HAICK, M. B. ; HO, Christine

; KOPLOW, Mike ; MANGOLD, Jennifer ; ROBINSON, Stefanie ; ROSA, Matt ; SCHWARTZ, Miclas ; SIMS, Christo ; STOFFREGEN, Hanns ; WATERBURY, Andrew ; LELAND, Eli S. ; PERING, Trevor ; WRIGHT, Paul K.: Wireless Sensor Networks for Home Health Care. In: *AINAW '07: Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops*. Washington, DC, USA : IEEE Computer Society, 2007. – ISBN 0–7695–2847–3, S. 832–837

[BPC+07] BARONTI, Paolo ; PILLAI, Prashant ; CHOOK, Vince W. C. ; CHESSA, Stefano ; GOTTA, Alberto ; HU, Y. F.: Wireless sensor networks: A survey on the state of the art and the 802.15.4 and ZigBee standards. In: *Comput. Commun.* 30 (2007), Nr. 7, S. 1655–1695. – ISSN 0140–3664

[Bro04] BROWN, Alan W.: Model driven architecture: Principles and practice. In: *Software and Systems Modeling* 3 (2004), December, Nr. 4, 314–327. – ISSN 1619–1366 (Print) 1619–1374 (Online)

[BS08] BOONMA, Pruet ; SUZUKI, Junichi: Middleware Support for Pluggable Non-Functional Properties in Wireless Sensor Networks. In: *SERVICES '08: Proceedings of the 2008 IEEE Congress on Services - Part I*. Washington, DC, USA : IEEE Computer Society, July 2008. – ISBN 978–0–7695–3286–8, S. 360–367

[BS09] BOONMA, Pruet ; SUZUKI, Junichi: Self-Configuring Publish/Subscribe Middleware for Wireless Sensor Networks. In: *Consumer Communications and Networking Conference, 2009. CCNC 2009. 6th IEEE*, 2009, S. 1–8

[CES04] CULLER, D. ; ESTRIN, D. ; SRIVASTAVA, M.: Guest Editors' Introduction: Overview of Sensor Networks. In: *Computer* 37 (2004), Aug., Nr. 8, S. 41–49. – ISSN 0018–9162

[CF09] CORRADI, Antonio ; FOSCHINI, Luca: A DDS-compliant P2P infrastructure for reliable and QoS-enabled data dissemination. In: *Parallel and Distributed Processing Symposium, International* 0 (2009), S. 1–8. ISBN 978–1–4244–3751–1

[Chi08] CHIPCON: *CC2430 A True System-on-Chip solution for 2.4 GHz IEEE 802.15.4 / ZigBee*. rev. 2.1 SWRS036F. Dallas, Texas, USA: Chipcon / Texas Instruments Inc., Nov 2008

[CV04] CHEN, Dazhi ; VARSHNEY, Pramod K.: QoS Support in Wireless Sensor Networks: A Survey. In: *Proc. of the 2004 International Conference on Wireless Networks (ICWN 2004), Las Vegas, Nevada, USA*, 2004

[DBM+09] DUBOIS, Philippe ; BOTTERON, Cyril ; MITEV, Valentin ; MENON, Carlo ; FARINE, Pierre-André ; DAINESI, Paolo ; IONESCU, Adrian ; SHEA, Herbert: Ad hoc wireless sensor networks for exploration of Solar-system bodies. In: *Acta Astronautica* 64 (2009), Nr. 5-6, 626 - 643. – ISSN 0094–5765

[Eic10] EICHELBERG, Marco: *Interoperabilität von AAL-Systemkomponenten 1: Stand der Technik*. Vde-Verlag, 2010. – ISBN 9783800731961

[Ets94] ETSCHBERGER, Konrad: *CAN Controller-Area-Network*. Carl Hanser Verlag, 1994

[Far08] FARAHANI, Shahin: *ZigBee Wireless Networks and Transceivers*. Newnes Elsevier, 2008. – ISBN 978–0–7506–8393–7

[FFM+09] FLAMMINI, Alessandra ; FERRARI, Paolo ; MARIOLI, Daniele ; SISINNI, Emiliano ; TARONI, Andrea: Wired and wireless sensor networks for industrial applications. In: *Microelectronics Journal* 40 (2009), Nr. 9, 1322 - 1336. – Quality in Electronic Design; 2nd IEEE International Workshop on Advances in Sensors and Interfaces; Thermal Investigations of ICs and Systems. – ISSN 0026–2692

[Fou] FOUNDATION, Eclipse: *Eclipse Modeling Framework.* Webseite. http://www.eclipse.org/emf/

[Her05] *Kapitel* Sensoren. In: HERING, Gutekunst: *Elektronik für Ingenieure und Naturwissenschaftler*. 5. Auflage. Springer Berlin Heidelberg, 2005, S. 298 – 319

[HM06] HADIM, Salem ; MOHAMED, Nader: Middleware: Middleware Challenges and Approaches for Wireless Sensor Networks. In: *IEEE Distributed Systems Online* 7 (2006), Nr. 3. – ISSN 1541–4922

[HSI+01] HEIDEMANN, John ; SILVA, Fabio ; INTANAGONWIWAT, Chalermek ; GOVINDAN, Ramesh ; ESTRIN, Deborah ; GANESAN, Deepak: Building efficient wireless sensor networks with low-level naming. In: *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*. New York, NY, USA : ACM, 2001. – ISBN 1–58113–389–8, S. 146–159

[Ins09] INSTRUMENTS, Texas: *MSP430 Ultra-Low-Power Microcontrollers*. Produktbeschreibung, 2009

[Int05] INTERSEMA: *MS5534A (RoHS) Barometer Module*. DA5534A_001. Bevaix, Schweitz: Intersema Sensoric SA, Jun 2005

[KMSB08] KURSCHL, Werner ; MITSCH, Stefan ; SCHÖNBÖCK, Johannes ; BEER, Wolfgang: Modeling wireless sensor networks based context-aware emergency coordination systems. In: *iiWAS '08: Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services*. New York, NY, USA : ACM, 2008. – ISBN 978–1–60558–349–5, S. 117–122

[Kno09] KNOLL, Andreas: Software-Stecker als Schlüssel zur Flexibilität. In: *Markt & Technik SPS/IPC/Drives 2009* Sonderheft 7 (2009), S. 18–21

[KPJ06] KABADAYI, Sanem ; PRIDGEN, Adam ; JULIEN, Christine: Virtual Sensors: Abstracting Data from Physical Sensors. In: *WOWMOM '06: Proceedings of the 2006 International Symposium on on World of Wireless, Mobile and Multimedia Networks*. Washington, DC, USA : IEEE Computer Society, 2006. – ISBN 0–7695–2593–8, S. 587–592

[KW05] KARL, Holger ; WILLIG, Andreas: *Protocols and Architectures for Wireless Sensor Networks*. John Wiley & Sons, 2005. – ISBN 0470095105

[LVCA+07] LOSILLA, Fernando ; VICENTE-CHICOTE, Cristina ; ÁLVAREZ, Bárbara ; IBORRA, Andrés ; SÁNCHEZ, Pedro: Wireless Sensor Network Application Development: An Architecture-Centric MDE Approach. In: OQUENDO, Flávio (Hrsg.): *ECSA* Bd. 4758, Springer, 179–194

[MCP+02] MAINWARING, Alan ; CULLER, David ; POLASTRE, Joseph ; SZEWCZYK, Robert ; ANDERSON, John: Wireless sensor networks for habitat

monitoring. In: *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*. New York, NY, USA : ACM, 2002. – ISBN 1–58113–589–0, S. 88–97

[Mic05] MICROCHIP: *PIC16F87/88 Data Sheet*. DS30487C. Chandler, USA: Microchip Technology Inc., 2005

[MM03] MILLER, Joaquin ; MUKERJI, Jishnu: MDA Guide Version 1.0.1 / Object Management Group (OMG). 2003. – Forschungsbericht

[MM07] MASRI, W. ; MAMMERI, Z.: Middleware for Wireless Sensor Networks: A Comparative Analysis. In: *Network and Parallel Computing Workshops, 2007. NPC Workshops. IFIP International Conference on*, 2007, S. 349–356

[OMG07] OMG: *Data Distribution Service for Real-Time Systems Version 1.2*. formal/07-01-01. Needham MA, USA: Object Management Group, jan 2007

[OMG08a] OMG: *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1 Part 1: CORBA Interfaces*. Version 3.1 formal/2008-01-04. Needham MA, USA: Object Management Group, Jan 2008

[OMG08b] OMG: *UML Profile for Data Distribution Specification FTF Beta 1*. ptc/2008-07-02. Needham MA, USA: Object Management Group, jul 2008

[OMG09] OMG: *The Real-time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol Specification*. Version 2.1. Needham MA, USA: Object Management Group, Jan 2009

[Pri09] PRISMTECH: *OpenSplice DDS Version 4.x IDL Pre-processor Guide*. Doc Issue 14. Dunfermline, UK: PrismTech Limited, Nov 2009

[PSC09] POTDAR, V. ; SHARIF, A. ; CHANG, E.: Wireless Sensor Networks: A Survey. In: *Advanced Information Networking and Applications Workshops, 2009. WAINA '09. International Conference on*, 2009, S. 636–641

[RKM02] RÖMER, Kay ; KASTEN, Oliver ; MATTERN, Friedemann: Middleware challenges for wireless sensor networks. In: *SIGMOBILE Mob. Comput. Commun. Rev.* 6 (2002), Nr. 4, S. 59–61. – ISSN 1559–1662

[SCv08] SCHMIDT, Douglas C. ; CORSARO, Angelo ; VAN'T HAG, Hans: Addressing the Challenges of Tactical Information Management in Net-Centric Systems With DDS. In: *Crosstalk-The Journal of Defense Software Engineering* Mar 2008 Issue (2008), S. 24–29

[sen09] SENSIRION: *Preliminary Datasheer SHT21*. Version 0.5. Staefa, Schweitz: SENSIRION AG, Sep 2009

[SET09] SEAH, W.K.G. ; EU, Zhi A. ; TAN, Hwee-Pink: Wireless sensor networks powered by ambient energy harvesting (WSN-HEAP) - Survey and challenges. In: *Wireless Communication, Vehicular Technology, Information Theory and Aerospace Electronic Systems Technology, 2009. Wireless VITAE 2009. 1st International Conference on*, 2009, S. 1 –5

[SH08] SCHMIDT, D.C. ; HAG, H. van't: Addressing the challenges of mission-critical information management in next-generation net-centric pub/-sub systems with OpenSplice DDS. In: *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, 2008. – ISSN 1530–2075, S. 1–8

[STM05] STMICROELECTRONICS: *LIS3LV02DQ MEMS INERTIAL SENSOR 3-Axis - ±2g/±6g Digital Output Low Voltage Linear Accelerometer*. CD00047926 Rev 1. Genf, Schweitz: STMicroelectronics, Oct 2005

[SVEH07] STAHL, Thomas ; VÖLTER, Markus ; EFFTINGE, Sven ; HAASE, Arno: *Modellgetriebene Software-Entwicklung*. dpunkt.verlag GmbH, 2007. – ISBN 978–3–89864–448–8

[TAO03] TAOS: *TSL2550 AMBIENT LIGHT SENSOR WITH SMBus INTERFACE*. TAOS029A. Plano, USA: TAOS Texas Advanced Optoelectronic Solutions, Dec 2003

[TI04] TI: *CC2420 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver*. SWRS041B. Dallas, USA: Texas Instruments Incorporated, jun 2004

[TI07] TI: *CC1000 Single Chip Very Low Power RF Transceiver*. SWRS048A. Dallas, USA: Texas Instruments Incorporated, jan 2007

[TI09] TI: *CC430F613x MSP430 SoC with RF Core*. SLAS554. Dallas, USA: Texas Instruments Incorporated, may 2009

[TS03] TANENBAUM, Andrew S. ; STEEN, Maarten van: *Verteilte Systeme. Grundlagen und Paradigmen*. Pearson Studium, 2003. – ISBN 3827370574

[VCSM03] VIEIRA, M.A.M. ; COELHO, Jr. ; SILVA, Jr. da ; MATA, J.M. da: Survey on wireless sensor network devices. In: *Emerging Technologies and Factory Automation, 2003. Proceedings. ETFA '03. IEEE Conference* Bd. 1, 2003, S. 537–544 vol.1

[Vdd06] VIEIRA, Marcos A. M. ; DA CUNHA, Adriano B. ; DA SILVA, Diógenes C.: Designing Wireless Sensor Nodes. In: *Embedded Computer Systems: Architectures, Modeling, and Simulation* Bd. 4017/2006, Springer Berlin / Heidelberg, 2006. – ISBN 978–3–540–36410–8, S. 99 – 108

[VSK05] VOELTER, Markus ; SALZMANN, Christian ; KIRCHER, Michael: Model Driven Software Development in the Context of Embedded Component Infrastructures. In: *Component-Based Software Development for Embedded Systems* Bd. 3778, Springer Berlin / Heidelberg. – ISBN 978–3–540–30644–3, 143–163

[YMG08] YICK, Jennifer ; MUKHERJEE, Biswanath ; GHOSAL, Dipak: Wireless sensor network survey. In: *Comput. Netw.* 52 (2008), Nr. 12, S. 2292–2330. – ISSN 1389–1286

[ZSLM04] ZHANG, Pei ; SADLER, Christopher M. ; LYON, Stephen A. ; MARTONOSI, Margaret: Hardware design experiences in ZebraNet. In: *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*. New York, NY, USA : ACM, 2004. – ISBN 1–58113–879–2, S. 227–238