

Universität Stuttgart
**Fakultät Elektrotechnik, Informatik,
Informationstechnik**

Studiengang: Softwaretechnik
Prüfer: Prof. Dr. rer. nat. Dr. h.c. Kurt Rothermel
Betreuer: Dipl. Inf. Frank Dürr
Dr. rer. nat. Alexander Leonhardi (DaimlerChrysler)
Beginn am: 01.07.2003
Beendet am: 31.12.2003

CR-Klassifikation: C.2.0, C.2.1, C.2.3, C.2.4, C.2.5, J.2

Diplomarbeit Nr. 2123

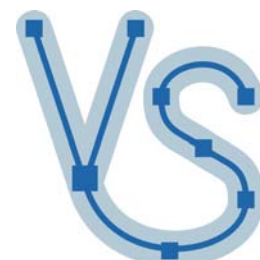
**Entwurf einer Beobachterkomponente für
die Diagnose eines Multimedianeitzwerks im
Fahrzeug**

Jan Geiger



Institut für Parallele und
Verteilte Systeme (IPVS)
Abteilung Verteilte Systeme

Universitätsstr. 38



Kurzfassung

Telematiksysteme in Kraftfahrzeugen bieten den Fahrzeuginsassen Navigations- Kommunikations- und Unterhaltungsfunktionen. Durch wachsende Ansprüche an Fahrzeugtelematiksysteme wachsen sowohl die angebotene Funktionsvielfalt als auch die Komplexität beständig an. Aus diesem Grund werden Fahrzeugtelematiksysteme zunehmend als verteilte Systeme realisiert, deren einzelne Komponenten spezialisierte Funktionen bereitstellen. Die einzelnen Komponenten sind über ein – für die Übertragung von Multimediadaten geeignetes - Kommunikationsnetz miteinander verbunden.

Eine im Automobilbereich eingesetzte Standardtechnologie für die Realisierung verteilter Telematiksysteme ist der MOST-Bus (Media Oriented Systems Transport). Die Verteilung der Telematikfunktionen auf verschiedene Geräte impliziert dabei eine Kooperation dieser Geräte.

Bislang fehlen allerdings noch systemweite Diagnosefunktionen für den MOST-Bus. Somit besteht ein Defizit bei der Überprüfung systemweiter Funktionen, welche die Kooperation von Komponenten des Telematiksystems beinhalten.

Ziel dieser Arbeit ist die Untersuchung von Mechanismen für die Beobachtung des MOST-Bus während des normalen Betriebs. Ausgehend von den Ergebnissen dieser Untersuchung wird eine generische Architektur für eine Beobachterkomponente in Form eines Frameworks vorgeschlagen. Das vorgeschlagene Framework ermöglicht eine flexible Konfiguration der Beobachterkomponente bezüglich der zu überwachenden Systemparameter. Die Funktionalität der Beobachterkomponente ist darüber hinaus um weitere Bausteine erweiterbar. Über eine definierte Schnittstelle können die aus der Beobachtung gewonnenen Daten an andere Programme zur weiteren Verarbeitung übertragen werden.

Als Nachweis für ihre Funktionsfähigkeit wird die vorgeschlagene Architektur in ein reales System umgesetzt. Bei der abschließenden Evaluation wird untersucht, inwiefern die realisierte Beobachterkomponente in der Lage ist, verschiedene Systemparameter zu beobachten, mit deren Hilfe ein mögliches Fehlverhalten entdeckt werden kann.

Inhalt

1	EINFÜHRUNG	1
1.1	ELEKTRONIK UND SOFTWARE IM FAHRZEUG	1
1.2	HINTERGRUND DIESER ARBEIT	4
1.2.1	Autonomic Computing.....	4
1.2.2	Autonomic Computing im Bereich der Fahrzeugtelematik.....	5
2	GRUNDLAGEN.....	8
2.1	DIE MOST-TECHNOLOGIE (MEDIA ORIENTED SYSTEMS TRANSPORT).....	8
2.1.1	Die MOST-Architektur	9
2.1.2	Applikationsschicht.....	13
2.1.3	NW-Schicht.....	16
2.1.4	Probleme beim Betrieb von MOST-Systemen	23
2.2	AUTONOMIC COMPUTING	24
2.2.1	Architecture Based Languages and Environments.....	24
2.2.2	IBM Autonomic Computing	26
2.2.3	Recovery-Oriented Computing	27
2.2.4	Self-healing In-vehicle Telematics Systems (SeHeT).....	28
2.3	MONITORING	31
2.3.1	Netzwerk-Management und Netzwerk-Monitoring	32
2.3.2	Die Rolle der Statistik im Netzwerk-Monitoring	39
2.3.3	Automatische Erkennung von Anomalien	42

2.3.4	Visualisierung der gewonnenen Daten.....	43
3	ANFORDERUNGSANALYSE	44
3.1	EINSATZSZENARIEN	44
3.2	MONITORING DES NACHRICHTENVERKEHRS AUF DEM MOST-BUS.....	45
3.3	KONFIGURIERBARKEIT.....	45
3.4	ANWENDUNGSSCHNITTSTELLE	46
3.5	PERFORMANZ.....	46
3.6	EINSATZ IN EINGEBETTETEN SYSTEMEN	46
3.7	GERINGE BEEINFLUSSUNG DES SYSTEMS DURCH DIE BEOBACHTUNG	47
4	EXISTIERENDE LÖSUNGEN FÜR DIE NETZWERKBEOBACHTUNG	48
4.1	SIMPLE NETWORK MANAGEMENT PROTOCOL - SNMP	48
4.1.1	Anwendbarkeit in Fahrzeugtelematiksystemen.....	50
4.2	NETWORK WEATHER SERVICE (NWS)	50
4.2.1	Anwendbarkeit der Mechanismen für die Beobachtung eines Fahrzeugtelematiksystems	52
4.3	MONITORING DES MOST-BUS.....	53
4.3.1	Möglichkeiten des Netzwerk-Monitoring	53
4.3.2	Umsetzung der Beobachtungsmechanismen in einem Fahrzeugtelematiksystem.....	58
4.3.3	OptoLyzer4MOST Software	59
4.3.4	MOST DataAnalyzer	60
5	KONZEPTE DER BEOBACHTUNG.....	63
5.1	ZU BEOBACHTENDE EREIGNISSE	63
5.1.1	Beobachtung auf der Netzwerkebene.....	64

5.1.2	Beobachtung auf der Anwendungsebene	65
5.2	VERARBEITUNGSMODELL	65
5.2.1	Das Architekturmuster Pipes and Filters.....	66
5.2.2	Anwendung des „Pipes and Filters“-Musters.....	67
5.2.3	Filter, Prozessoren und Datensenken	69
5.2.4	Verwaltung von Kontextinformationen.....	72
5.2.5	Verwendung eines einheitlichen Datenmodells	74
6	REALISIERUNG.....	76
6.1	ARCHITEKTUR DER BEOBACHTERKOMPONENTE	76
6.2	DATENQUELLEN.....	78
6.2.1	Auswertung von Tracedateien.....	79
6.2.2	Beobachtung während des laufenden Betriebs (Onlinebetrieb)	80
6.3	HAUPTPUFFER.....	81
6.4	FILTER UND PROZESSOREN	82
6.5	DATENSENKEN.....	85
6.6	KONFIGURATIONSKOMPONENTE	87
6.7	ANWENDUNGSSCHNITTSTELLE	87
6.7.1	Initialisierung der Beobachterkomponente	89
6.7.2	Zugriff auf die Beobachtungsdaten und Kontextinformationen	91
6.8	DAS DATENMODELL	93
6.9	DATENVERARBEITUNG WÄHREND DER BEOBACHTUNG	95
6.10	AUFBAU DER KONFIGURATIONSDATEI	96
6.11	FORMAT DER ZU VERARBEITENDEN TRACEDATEIEN	97
6.12	JAVA-COM BRIDGE	98

6.13	KOMPATIBILITÄT ZU J2ME UND OSGI	100
7	EVALUATION.....	101
7.1	EVALUATION DER FUNKTIONALITÄT	101
7.2	SPEICHERBEDARF UND PERFORMANZ DER BEOBACHTERKOMPONENTE.....	102
7.2.1	Speicherbedarf	103
7.2.2	Performanz.....	103
8	RESÜMEE.....	105
8.1	ZUSAMMENFASSUNG	105
8.2	AUSBLICK	107
A	BEOBACHTETE SYSTEMPARAMETER	109
B	SCHNITTSTELLEDEFINITIONEN.....	114
B.1	PACKAGE COM.DCX.MOST.MONITORING.APP	114
B.1.1	public interface Controller	114
B.1.2	public abstract class MonitoringFacade	114
B.1.3	public interface ProcessingCompleteHandler	117
B.2	PACKAGE COM.DCX.MOST.MONITORING.CAPTURE.....	117
B.2.1	public interface Buffer	117
B.2.2	public abstract class MOSTReader	118
B.3	PACKAGE COM.DCX.MOST.MONITORING.DATASINKS	119
B.3.1	public interface DataSink.....	119
B.3.2	public interface StatefulDataSink.....	119
B.4	PACKAGE COM.DCX.MOST.MONITORING.FILTERS.....	120
B.4.1	public abstract class AggregatingFilter	120

B.4.2	public interface Filter	121
B.4.3	public interface Processor	121
B.5	PACKAGE COM.DCX.MOST.MONITORING.IMPL	122
B.5.1	public abstract class ConfigurationManager	122
B.6	PACKAGE COM.DCX.MOST.OPTOCONTROL	123
B.6.1	public interface OptolyzerEventHandler	123
B.6.2	public interface OptolyzerWrapper	124
C	LITERATURVERZEICHNIS	127

1 Einführung

Dieses Kapitel beschreibt im ersten Teil die wachsende Bedeutung von elektronischen Systemen in Kraftfahrzeugen und die damit verbundenen Herausforderungen. Dabei wird zwischen Telematiksystemen, Komfortfunktionen und elektronisch gesteuerten Fahrzeugfunktionen differenziert. Im zweiten Teil wird der Ansatz beschrieben, ein Fahrzeugtelematiksystem als autonomes System zu betrachten. Dieser Ansatz bildet auch den Hintergrund für diese Diplomarbeit.

1.1 Elektronik und Software im Fahrzeug

Um das Autofahren komfortabler und sicherer zu machen, sind in modernen Automobilen immer mehr elektronische Systeme im Einsatz. Neue Komfortfunktionen wie die automatische Abstandsregelung (*adaptive cruise control*, ACC, [12]) oder Regensensoren machen das Autofahren immer komfortabler.

Den Funktionen und Anforderungen entsprechend werden bei der Fahrzeugelektronik drei Hauptbereiche unterschieden:

- **Elektronisch gesteuerte Fahrzeugfunktionen:** Diese Funktionen sind für den Betrieb des Fahrzeugs notwendig und meist sicherheitskritisch. Zu diesen Funktionen zählen das elektronisch gesteuerte Motormanagement, das elektronisch gesteuerte, aktive Fahrwerk, elektromechanische Bremsen (Brake-by-Wire) und die elektromechanische Lenkung ohne mechanische Lenksäule (Steer-by-Wire).

Die Ersetzung mechanischer und hydraulischer Systeme durch elektrische und elektronische Systeme bringt mehrere Vorteile mit sich: Die elektronische Motorsteuerung beispielsweise ermöglicht eine möglichst effiziente Kraftstoffnutzung, der Wegfall der mechanischen Lenksäule ergibt we-

sentlich mehr Freiheitsgrade bei der Anordnung der Fahrzeugkomponenten im Motorraum, und das Lenksystem kann in kritischen Situationen selbstständig gegenlenken um das Fahrzeug unter Kontrolle zu halten.

Allerdings ist gerade bei Funktionen wie Brake-by-Wire oder Steer-by-Wire absolute Zuverlässigkeit notwendig, so dass die elektronischen Systeme dann oftmals redundant ausgelegt werden müssen.

- Der Begriff **Telematik** beschreibt die Integration von Informations- und Kommunikationstechnologien. Auf den Automobilbereich übertragen (*Fahrzeugtelematik*) fallen darunter Kommunikations-, Navigations- und Unterhaltungsfunktionen. Konkrete Beispiele sind die satellitenbasierte Navigation, der automatische Notruf, Zugriff auf das Web oder auf Emails aus dem Fahrzeug, die Einbindung des Mobiltelefons, sowie DVD, Fernsehen und Videospiele.

Da ein Navigationssystem eine Schlüsselkomponente der Fahrzeugtelematik darstellt, besitzt das Telematiksystem Wissen über den Ortskontext (*location awareness*), mit dem zusätzliche Fahrzeugfunktionen wie Kurvenwarnsysteme realisiert werden können.

Die Anforderungen an ein Fahrzeugtelematiksystem unterscheiden sich von denen an elektronisch gesteuerte Fahrzeugfunktionen und sind eher mit den Anforderungen an moderne PCs oder Geräte der Unterhaltungselektronik zu vergleichen. Ein Telematiksystem muss über eine ansprechende und einfach zu handhabende Mensch-Maschine-Schnittstelle verfügen, welche die große Anzahl von Funktionen auf einfache Art und Weise zugänglich macht. Dadurch und durch die Verarbeitung von hochqualitativen Audio- und Videodaten müssen Telematikgeräte über große Rechen- und Speicherkapazitäten verfügen sowie über ein Multimedia-Netzwerk mit entsprechender Bandbreite vernetzt sein.

Darüber hinaus sind natürlich bei sicherheitskritischen Telematikfunktionen, wie beispielsweise einem Kurvenwarnsystem, zusätzlich Fehlertoleranz und Robustheit essentielle Anforderungen.

- Unter den Begriff **Komfortfunktionen** fallen elektronische Funktionen, die der Erhöhung des Komforts dienen, aber weder zu den Fahrzeugfunktionen im engeren Sinne noch zu den Telematikfunktionen zählen. Beispiele für Komfortfunktionen sind elektrische Fensterheber oder die Steuerung der Klimaanlage.

Die Mehrheit der wesentlichen Innovationen bei Automobilen entfällt heute auf den Bereich der Elektronik, so dass Funktionen, die durch elektronische Systeme realisiert werden, ein wesentliches Differenzierungsmerkmal bei

Kraftfahrzeugen darstellen. Allerdings führt der zunehmende Einsatz von Elektronik im Fahrzeug auch zu gewaltigen Steigerungen der Komplexität und der Kosten. Besonders anschaulich lässt sich die Komplexität anhand der Menge der Verkabelung quantifizieren: In dem Oberklassenfahrzeug Volkswagen Phaeton beispielsweise sind Kabel mit einer Gesamtlänge von nahezu vier Kilometern und einem Gesamtgewicht von über sechzig Kilogramm verbaut [12].

Es wird angestrebt, immer mehr Funktionalität durch Software statt durch Hardware zu realisieren, um Funktionsverbesserungen und Fehlerbereinigungen durch eine Reprogrammierung der Steuergeräte in ein Fahrzeug einzubringen (*Flashen*). Daher wird zukünftig der größte Anteil der Komplexität auf den Bereich der Software entfallen. Schon jetzt ist in einem modernen Kraftfahrzeug Software mit einem Umfang von über einer Million Zeilen Quellcode enthalten [32].

Eines der größten Probleme, das mit der steigenden Komplexität der Fahrzeugelektronik einhergeht, ist die Sicherung der Qualität, insbesondere in Bezug auf Robustheit und Fehlertoleranz, was in der Informationstechnik oftmals zugunsten der Innovationsgeschwindigkeit vernachlässigt wird ([43], [51]).

Ein Ansatz, um die zunehmenden Komplexitätsprobleme und die daraus resultierenden Qualitätsmängel zu entschärfen, besteht in der Einführung einer Standardplattform in der Fahrzeugelektronik. Eine Initiative, die in diese Richtung zielt, ist die *Automotive Open System Architecture* (AUTOSAR, [1]). Den Kern der AUTOSAR-Initiative bildet eine Middleware (*AUTOSAR RunTime Environment*), die von der zugrundeliegenden Hardware und den verschiedenen Betriebssystemen abstrahiert und einen einheitlichen Mechanismus für die Interprozesskommunikation definiert.

Die Verwendung einer solchen Architektur ermöglicht die Wiederverwendung zahlreicher Grundfunktionen in Form von Komponenten und erlaubt den Einsatz von Standardhardware. Die AUTOSAR-Architektur würde den Automobilherstellern aber weiterhin die Möglichkeit zur Produktdifferenzierung geben, indem sie bestimmte Funktionen wie z.B. die Mensch-Maschine-Schnittstelle unterschiedlich realisieren.

Es ist ein erklärtes Ziel von MOST, die Probleme im Bereich der Fahrzeugtelematik auf ähnliche Weise zu lösen, allerdings stellt sich bei näherer Betrachtung auch heraus, dass die MOST-Technologie an mehreren Stellen unnötig

komplex ist und wichtige Funktionen zur Unterstützung der Anwendungsentwicklung noch fehlen.

1.2 Hintergrund dieser Arbeit

Um die mit der Komplexität verbundenen Probleme zu überwinden, werden vor allem Techniken aus dem Bereich der Softwaretechnik, wie beispielsweise die Nutzung von Softwarekomponenten [54] oder die Einführung von Softwareproduktlinien [4] vorgeschlagen. Es gibt jedoch auch eine Reihe weiterer Ansätze, um die Komplexität moderner Softwaresysteme in den Griff zu bekommen. Ein Ansatz ist das so genannte *Autonomic Computing*, das im Folgenden kurz erläutert wird.

1.2.1 Autonomic Computing

Autonomic Computing zielt darauf ab, Softwaresysteme zu erhalten, die trotz ihrer Komplexität zuverlässig sind. Für ein System, das nach den Grundsätzen des *Autonomic Computing* entwickelt wurde (im weiteren Verlauf *autonomes System* genannt), ist keine dedizierte Systemadministration mehr nötig, da das System auf Fehler tolerant reagiert und sich selbständig rekonfiguriert. Das reicht bis zur Selbstreparatur, wenn das System eine Fehlfunktion bei sich beobachtet hat. Die Idee des *Autonomic Computing* geht insbesondere aus dem Bereich der Serveradministration hervor und bezeichnet ursprünglich eine Initiative von IBM [18], die Kosten für die Administration und für Systemausfälle zu senken, da sie diejenigen für die Hardware und die initiale Softwareentwicklung bei weitem übersteigen.

Laut IBM besitzt ein autonomes System die folgenden Haupteigenschaften:

- Ein autonomes System muss Wissen über sich selbst besitzen. Es muss also sowohl den aktuellen Zustand der Komponenten kennen, aus denen es besteht, als auch die Verbindungen zu anderen Systemen. Zu diesem Zweck muss sich ein autonomes System selbständig überwachen können: es muss über Monitoring-Funktionen verfügen.
- Es muss sich, unter Berücksichtigung der aktuellen Bedingungen, selbst konfigurieren können. Dazu gehört auch, dass ein autonomes System die Fähigkeit besitzt, sich selbst zu reparieren (*self-healing*), nachdem einzelne

Komponenten ausgefallen sind oder sich auf andere Weise fehlerhaft verhalten.

- Ein autonomes System muss vorhersagen können, welche Funktionen und Ressourcen seine Benutzer benötigen werden und sich auf diesen Bedarf einstellen. Die Performanz eines autonomen Systems wird daran gemessen, wie schnell es Dienste für seine Benutzer erbringen kann. Dabei ist es für die Benutzer vollkommen transparent, wie komplex das System ist.
- Da komplexe Systeme mit der Zeit aus den verschiedensten Gründen in ungünstige Zustände geraten können, führen autonome Systeme im Hintergrund ständig vorbeugende Rekonfigurationsmaßnahmen durch, z.B. Garbage Collection.
- Ein autonomes System muss sich zwar selbständig verwalten können, es wird aber nicht völlig autonom funktionieren, sondern immer mit anderen Systemen vernetzt sein und interagieren. Aus diesem Grund muss es auch die Fähigkeit mitbringen, seine Umwelt zu kennen und mit dieser möglichst optimal zu interagieren.

Im Rahmen dieser Arbeit, wird die Eigenschaft zur Selbstheilung bzw. der Selbstadaption als die zentrale Charakteristik eines autonomen Systems betrachtet.

Es gibt dabei verschiedene Ansätze, wie bestehende und noch zu entwickelnde Systeme mit Mechanismen zur Selbstadaption versehen werden können.

Abschnitt 2.1.4 stellt dazu mehrere Forschungsprojekte und deren Ergebnisse vor.

1.2.2 Autonomic Computing im Bereich der Fahrzeugtelematik

Im Serverbereich ist es vor allem der Kostendruck bezüglich der Administration, der Hersteller und Systemanbieter dazu bringt, autonomen Systemen ihre Aufmerksamkeit zu schenken. Im Bereich der Fahrzeugtelematik hingegen sind die Anforderungen anders gelagert:

- **Fehlende Administration:** Während des Betriebs gibt es bis jetzt keine Möglichkeit, das Telematiksystem innerhalb eines Fahrzeugs zu administrieren. Auch wenn das Auto in der Werkstatt steht, ist dort oftmals keine fachkundige Administration möglich, da das Werkstattpersonal häufig nicht über das nötige Fachwissen verfügt. Dies hängt vor allem mit der Entwicklung der Technik und der zunehmenden Komplexität der Kraftfahrzeugelektronik zusammen.

- **Robustheit:** An ein Fahrzeugtelematiksystem werden hohe Ansprüche in Bezug auf dessen Robustheit und Zuverlässigkeit gestellt. Dies hängt vor allem mit dem breiten Einsatz in zahlreichen Fahrzeugen zusammen. Damit verbunden sind auch hohe Kosten für Einzelreparaturen und insbesondere für Rückrufaktionen.
- **Geschäftsprozesse in der Automobilbranche:** Neben deutlich längeren Entwicklungszeiten weisen Fahrzeuge üblicherweise wesentlich längere Lebenszyklen als Serversysteme oder Arbeitsplatzrechner auf. Da ein Telematiksystem einen integralen Bestandteil eines Fahrzeugs darstellt, gilt dies auch für Fahrzeugtelematiksysteme. Somit muss ein Telematiksystem über eine sehr lange Zeit zuverlässig funktionieren. Wünschenswert ist ferner, dass ein Telematiksystem auf konsistente Art und Weise aktualisiert werden kann.

Die einzelnen Telematikgeräte werden nicht von den Automobilherstellern selbst, sondern von verschiedenen, eventuell wechselnden, Zulieferern entwickelt und hergestellt. Aus diesem Grund fehlen bislang systemweite, geräteübergreifende Diagnose-Mechanismen. Falls auf Systemebene Mechanismen für die Selbstadaption von Fahrzeugtelematiksystemen eingeführt werden, müssten diese erst zwischen den Automobilherstellern und den verschiedenen Zulieferern abgestimmt werden.

- **Kostendruck:** Im Gegensatz zum Serverbereich schließt der Kostendruck - vor allem auf Grund der großen Stückzahlen und der Konkurrenzsituation - die Verwendung redundanter Komponenten weitgehend aus. Lediglich für sicherheitskritische Funktionen ist der Einsatz redundanter Komponenten möglich.
- **Einsatzumgebung:** Ein Kraftfahrzeug ist eine vollständig andere Umgebung für ein elektronisches System als ein Büro oder ein Serverraum. Die Elektronik muss über einen großen Temperaturbereich funktionieren, ist während der Fahrt hohen mechanischen Belastungen ausgesetzt und die Spannungsversorgung ist größeren Schwankungen unterworfen.
- **Dynamik und Konnektivität:** Ein Fahrzeugtelematiksystem ist in seiner Dynamik und Konnektivität stark eingeschränkt. Die einzelnen Geräte sind fest in das Telematiksystem eingebaut und ihre Anzahl ist bereits im Voraus bekannt.

Die Konnektivität eines Fahrzeugtelematiksystems mit externen Systemen ist, zumindest beim derzeitigen Stand, noch stark eingeschränkt. Allerdings ist es möglich, dass sich zumindest beim letztgenannten Punkt in nächster Zeit etwas ändert, wenn beispielsweise benachbarte Autos Informationen austauschen.

Manche der Anforderungen, wie die fehlende Administration oder die lange Lebensspanne der Systeme machen die Verwendung von autonomen Systemen im Bereich der Fahrzeugtelematik besonders attraktiv. Die eingeschränkte Dynamik und Vernetzung mit anderen Systemen vereinfachen die Implementierung entsprechender Mechanismen. Andere Faktoren wie die Rolle der Zulieferer und der hohe Kostendruck erschweren wiederum die Realisierung von selbstheilenden Telematiksystemen.

Aus diesem Grund beschäftigt sich bei DaimlerChrysler eine Forschungsprojekt mit selbstheilenden Telematiksystemen [32]. Danach umfasst die Realisierung eines selbstheilenden Telematiksystems insgesamt vier Schritte. Der erste Schritt besteht in der Beschreibung des Systems. Diese Beschreibung wird bei der Selbstbeobachtung (Schritt 2) und der Diagnose des Systemverhaltens (Schritt 3) berücksichtigt. Wird dabei ein Fehler erkannt, führt das System Reparaturmaßnahmen durch (Schritt 4). Das Forschungsprojekt wird in Abschnitt 2.2.4 näher vorgestellt.

Die in der Diplomarbeit zu realisierende Beobachterkomponente soll dabei geeignet sein, in der Monitoringschicht eingesetzt zu werden und die Komponente(n) der Diagnoseschicht mit den benötigten Informationen versorgen. Daneben soll die Beobachterkomponente auch als eigenständiges Werkzeug im Bereich der Entwicklung von Telematiksystemen eingesetzt werden können, um Mechanismen für die Beobachtung und Diagnose zu untersuchen und zu bewerten.

2 Grundlagen

In diesem Kapitel werden die technischen Grundlagen erläutert, auf die im weiteren Verlauf der Arbeit Bezug genommen wird. Zunächst wird die Netzwerktechnologie MOST (Media Oriented Systems Transport) vorgestellt. Als nächstes werden einige konkrete Projekte im Bereich des Autonomic Computing beschrieben, um die allgemeinen Ausführungen des ersten Kapitels zu konkretisieren. Den Abschluss des Kapitels bildet eine Betrachtung über die grundlegenden Aspekte der System- und der Netzwerkbeobachtung.

2.1 Die MOST-Technologie (Media Oriented Systems Transport)

Die Initiative zur Entwicklung der MOST-Technologie geht auf die Automobilhersteller BMW und DaimlerChrysler zurück, die zusammen mit Zulieferern im Bereich der Automobilelektronik 1998 die MOST Cooperation [38] gegründet haben.

Der MOST-Bus [42] beginnt sich zumindest bei Fahrzeugen der Oberklasse als eine Standardtechnologie zur Vernetzung von Telematikgeräten zu etablieren. So haben nach BMW und Mercedes auch Audi [58], Porsche, Saab sowie weitere Hersteller Modelle mit auf MOST basierenden Telematiksystemen vorgestellt.

Ziel der MOST-Initiative ist es, nicht nur die reine Vernetzung von Telematikgeräten zu standardisieren, sondern darüber hinaus ein vollständiges Framework zur Entwicklung von Telematikanwendungen zur Verfügung zu stellen. Somit können sich die Automobilhersteller die Entwicklungskosten für die Vernetzungstechnologie teilen.

Durch die Verwendung eines gemeinsamen Mediums, über das alle Telematikgeräte miteinander verbunden sind (Bus), wird die Verkabelung wesentlich vereinfacht, da nicht mehr jedes Gerät mit jedem anderen (durch Kabel) verbunden werden muss. Als Übertragungsmedium kommen Lichtwellenleiter aus Kunststoff zum Einsatz, die leichter als herkömmliche Kupferkabel und unempfindlich gegen elektromagnetische Störungen sind. Die Übertragung erfolgt ausschließlich in digitaler Form, dadurch entfallen unnötige AD/DA-Wandlerbausteine in den Telematikgeräten.

Die Gesamtbandbreite des MOST-Bus von knapp 25 Megabit pro Sekunde wird durch *Time Division Multiplexing* in Zeitschlitze unterteilt. Für die Angabe des Zeittaktes ist ein einzelnes Gerät verantwortlich (*Master*).

Grundsätzlich überträgt der MOST-Bus die Daten mit einer definierten Dienstgüte: Da die einzelnen Zeitschlitze reserviert werden können, ist es möglich, kontinuierliche Daten wie Audio- und Videosignale mit garantierter Bandbreite und Verzögerung zu übertragen. Da über den MOST-Bus nicht nur Audio- und Videodaten übermittelt werden, sondern auch Steuerdaten oder Grafiken, werden aufbauend auf den synchronen Basismechanismus auch geeignete Dienste für die Übertragung dieser Daten angeboten.

MOST definiert über die Bitübertragungsschicht und die Zugriffsverfahren hinaus ein einheitliches Adressierungsverfahren für Geräte, Anwendungen bis hin zu einzelnen Operationen und Eigenschaften, die von Anwendungen angeboten werden. Schließlich definiert MOST nicht nur die Mechanismen für die Netzwerkverwaltung und den Informationsaustausch, sondern auch standardisierte Programmierschnittstellen (API) für eine Auswahl typischer Telematikanwendungen [39].

2.1.1 Die MOST-Architektur

Abbildung 1 zeigt den logischen Aufbau eines MOST-Gerätes. Ein MOST-Gerät ist nach dem Schichtenprinzip aufgebaut, wobei im Gegensatz etwa zum ISO-OSI-Referenzmodell [26] keine strikte, sondern eine eher lose Schichtung besteht. Im Vergleich zum OSI-Schichtenmodell verfügt ein MOST-Gerät über deutlich weniger Schichten. Die einzelnen Schichten werden im Folgenden kurz beschrieben:

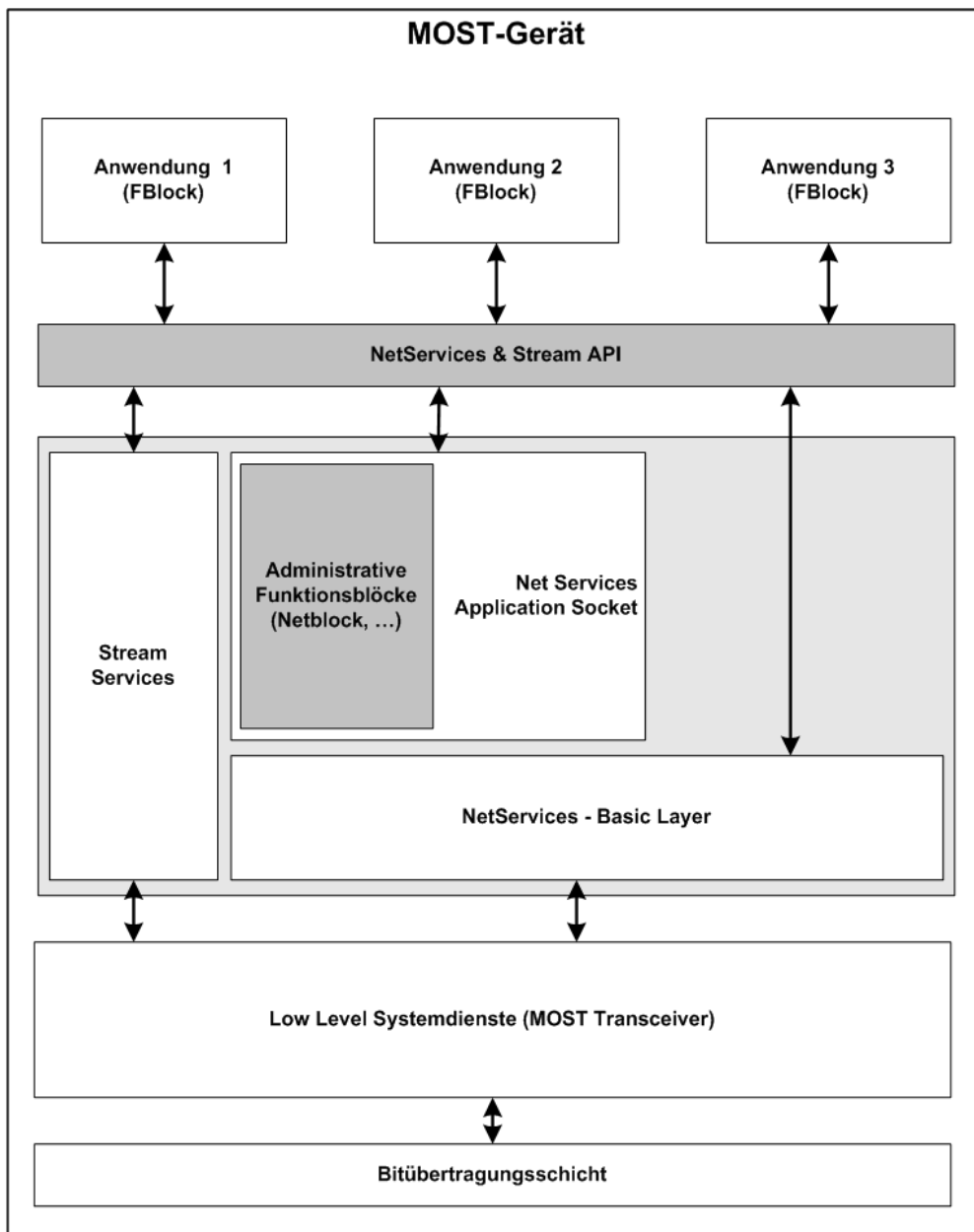


Abbildung 1: Struktureller Aufbau eines MOST-Gerätes

- **Bitübertragungsschicht** (*MOST Physical Layer*): Der MOST Physical Layer definiert die Eigenschaften des Übertragungsmediums, der Steckverbindungen zwischen Kabelsegmenten und Geräten, die Netzwerktopologie sowie die Kodierung der Daten zur Übertragung. Standardmäßig geht die MOST-Spezifikation von Lichtwellenleitern aus Kunststoff (*Plastics Optical Fiber – POF*) aus, da sich mit ihnen die geforderten Eigenschaften einfach

und kostengünstig realisieren lassen. Prinzipiell ist aber auch die Verwendung anderer Übertragungsmedien wie etwa Kupferkabeln möglich. Als Netzwerktopologie wird in der MOST-Spezifikation eine ringförmige Anordnung der MOST-Geräte favorisiert, aber die Spezifikation lässt auch andere Topologien zu.

- **Low-Level Systemdienste:** Zu der Funktionalität dieser Schicht gehört vor allem die physische Adressierung der MOST-Geräte. Die physische Adresse eines MOST-Gerätes wird aus der Position des Gerätes im MOST-Ring abgeleitet. Das taktgebende Mastergerät hat immer die Position 0, die nachfolgenden Geräte im Ring werden aufsteigend durchnummeriert. Weitere Funktionen, die diese Schicht bietet sind die Belegung und Freigabe physischer Kanäle, der paketweise Austausch von Daten einschließlich der Erkennung von Übertragungsfehlern sowie die Konvertierung und Übertragung serieller Daten, die in anderen Formaten vorliegen (z.B. Audiodaten von der S/PDIF-Schnittstelle). Die Funktionalität der Low-Level Systemdienste wird vom MOST-Transceiverchip und damit in Hardware realisiert.
- **Net Services Basic Layer:** Diese Schicht stellt vor allem die Schnittstelle zu den Low-Level Systemdiensten und damit zum Zugriff auf den MOST-Transceiverchip bereit. Die Kernfunktionalität stellen dabei die Dienste für die Datenübertragung dar: der Zugriff auf den Kontrollkanal über den Control Message Service (CMS) oder den Application Message Service (AMS), der Zugriff auf den asynchronen Kanal sowie die Reservierung, Nutzung und Freigabe von synchronen Kanälen. Neben den Kommunikationsfunktionen bietet diese Schicht der Applikationsschicht auch Zugriffsmöglichkeiten auf die Register des Transceiverchips und zeichnet sich für die korrekte Initialisierung der Netzwerkschicht verantwortlich. Zur Netzwerkschicht zählt die gesamte Funktionalität unterhalb der Programmierschnittstelle.
- **Net Services Application Socket:** Diese Schicht definiert den Aufbau und die Adressierung von Anwendungen in einem MOST-System. Dazu wird die gesamte Anwendungsfunktionalität in verschiedene so genannte Funktionsblöcke (FBlocks) unterteilt, die den Typ und damit die Schnittstelle der jeweiligen Anwendungen festlegen. Um verschiedene Instanzen desselben Anwendungstyps in einem MOST-System zu unterscheiden, bekommt jede Anwendung einen innerhalb des Anwendungstyps eindeutigen Bezeichner zugeordnet. Der Application Socket definiert auch spezielle Funktionsblöcke, die administrative Funktionen in MOST-Systemen übernehmen. So verfügt beispielsweise jedes Gerät über einen speziellen Funktionsblock, den so genannten *NetBlock*, der Informationen über die Eigenschaften des jeweili-

gen Gerätes und dessen Anwendungen bereitstellt. Das Mastergerät stellt systemweit relevante Daten wie etwa die Beschreibungen der Systemkonfiguration oder die Belegung der synchronen Kanäle durch bestimmte, wohlbekannte Funktionsblöcke zur Verfügung.

Zu den Bestandteilen des Application Socket gehören mit einem Ereignisdienst (*Notification Service*) und einem dem Transmission Control Protocol (TCP) nachempfundenen Transportdienst (*MOST High Protocol*, vgl. [40]) auch zusätzliche Kommunikationsdienste.

- **Stream Services:** Die Stream Services bieten einen Transportdienst für die Übertragung synchroner Daten in Echtzeit, der von Anwendungen genutzt werden kann.
- Das **Net Services und Stream API** stellt eine einheitliche Programmierschnittstelle zum Zugriff auf die Funktionalität der Stream Services und der Net Services bereit. Damit können MOST-Anwendungen Zugriff auf Funktionen zur Netzwerkverwaltung bekommen sowie anwendungsspezifische Daten versenden und empfangen.
- **Anwendungen:** Jede Anwendung wird, wie schon bei der Beschreibung des Application Socket erwähnt, durch einen FBlock repräsentiert, der instanziiert werden kann und über eine definierte Schnittstelle verfügt. Die Schnittstelle eines FBlocks beschreibt, welche Eigenschaften und Methoden eine Anwendung öffentlich zur Verfügung stellt. Unter Nutzung des Net Services & Stream API können Anwendungen auf die Funktionalität des MOST-Bus zugreifen und miteinander kommunizieren.

Insgesamt definiert die MOST-Architektur zwar mehrere Schichten. Oberhalb der vom Transceiverchip realisierten Low-Level Systemdienste wird das Schichtenprinzip aber nur mehr sehr lose verfolgt. So werden für die Netzwerkverwaltung mit der Nutzung von Funktionsblöcken und dem Mechanismus zur Adressierung von Anwendungen Konzepte aus der Applikationsschicht eingesetzt. Die Anwendungen selbst können über das NetServices-API bis auf die Register des Transceiverchips zugreifen. Im weiteren Verlauf dieses Kapitels wird daher nur noch zwischen der Netzwerk- und der Applikationsschicht unterschieden.

2.1.2 Applikationsschicht

Die Applikationsschicht (oder Anwendungsschicht) besteht aus verschiedenen Funktionsblöcken, die jeweils bestimmte Funktionen realisieren. Ein Funktionsblock kann dabei verschiedene Rollen einnehmen:

Ein *Controller* benutzt Funktionen von anderen Funktionsblöcken und verfügt in vielen Fällen über eine Schnittstelle zur Bedienung durch den Endnutzer (Mensch-Maschine-Schnittstelle, MMI). Funktionsblöcke, die lediglich Basisfunktionalität anbieten und nur von anderen Funktionsblöcken aufgerufen werden, bezeichnet man in der MOST-Terminologie als *Slaves*.

Jeder Funktionsblock hat einen Typ, der die Schnittstelle definiert. Jeder Typ wird durch einen eindeutigen Bezeichner (*FBlockID*) gekennzeichnet. Von einem Typ können dann in einem MOST-System mehrere Instanzen vorhanden sein. Jede FBlock-Instanz wird innerhalb des Typs durch eine Instanz-ID (*InstID*) eindeutig gekennzeichnet. Systemweit werden FBlock-Instanzen somit durch einen zusammengesetzten Bezeichner *FBlockID.InstID* unterschieden. Die verschiedenen InstIDs werden in flüchtigem Speicher gehalten und können sich daher mit jedem Neustart eines Gerätes ändern. Die einzige Zusage, die bezüglich der Instanz-IDs besteht, ist die Eindeutigkeit innerhalb des jeweiligen FBlock-Typs. Aus diesem Grund werden in der Praxis häufig die Geräteadresse und die FBlockID als identifizierendes Merkmal von FBlock-Instanzen verwendet. Dieses Vorgehen impliziert natürlich, dass die Geräte feste Positionen im Ring haben und jedes Gerät nur maximal eine Instanz eines FBlock-Typs implementiert.

Jeder Funktionsblock stellt eine bestimmte Menge von Funktionen öffentlich zur Verfügung. Dabei werden zwei Arten von Funktionen unterschieden:

Methoden sind Funktionen, die aufgerufen werden können und ein Resultat in definierter Zeit berechnen.

Eigenschaften (Properties) repräsentieren dagegen Gerätezustände und können entweder gesetzt oder abgefragt werden.

Alle Funktionsaufrufe werden für den Transport in entsprechende Kontrollnachrichten umgesetzt. Um die verschiedenen Funktionen in einem MOST-System zu adressieren definiert die MOST-Spezifikation so genannte *Protokolle*: Jede Funktion ist demnach durch folgende Eigenschaften gekennzeichnet, die sich so auch im Aufbau der Kontrollnachrichten widerspiegeln.

Ein vollständiges Protokoll besitzt das folgende Format:

DeviceID.FBlockID.InstID.FktId.OpType.Length

FktID gibt einen eindeutigen Bezeichner der Methode oder der Eigenschaft innerhalb eines Funktionsblocks an. *OpType* bezeichnet den Funktionstyp, also ob es sich beispielsweise um eine Fehlerbenachrichtigung oder die Anfrage nach dem Wert einer Eigenschaft handelt. *Length* beschreibt schließlich die Länge der enthaltenen Nutzdaten (Parameter, Rückgabewerte). Die in *DeviceID* enthaltene Geräteadresse wird dabei nicht von der Anwendung, sondern von den NetServices festgelegt.

Für die Beschreibung der Nutzdaten definiert MOST ein einfaches Typmodell, in dem Grundtypen wie Boolean, Aufzählungen, Ganzzahlen sowie Strings zur Verfügung stehen. Mit *Streams* existieren Container-Datentypen, die beliebige andere Datenstrukturen enthalten können und explizite Typinformationen über die enthaltenen Daten bereitstellen. Als zusammengesetzte Datentypen werden Arrays und Records angeboten.

Bei einem entfernten Aufruf werden die Typinformationen, außer bei Streams, nicht mitübertragen, sondern sind Bestandteil der Funktionsbeschreibung. Die Funktionen werden üblicherweise durch Spezifikationsdokumente (so genannte Funktionskataloge) beschrieben. Tabelle 1 listet die Funktionsblöcke aus dem Standard-Funktionskatalog von MOST [39] auf.

FBlockID	Name	Kommentar
0x00	GeneralFBlock	Legt die grundlegende Funktionalität aller Anwendungs-Funktionsblöcke fest (über Vererbung), etwa den Ereignisdienst oder den Introspektionsmechanismus.
0x01	NetBlock	Administrativer Funktionsblock, stellt Informationen über Gerät zur Verfügung
0x02	NetworkMaster	Administrativer Funktionsblock, verwaltet Central Registry

0x03	ConnectionMaster	Administrativer Funktionsblock, verwaltet synchrone Verbindungen
0x06	Diagnosis	Administrativer Funktionsblock, stellt Diagnosefunktionen für Gerät zur Verfügung
0x22	AudioAmplifier	
0x26	MicrophoneInput	
0x30	AudioTapePlayer	
0x31	AudioDiskPlayer	
0x40	AmFmTuner	
0x41	TMCTuner	Erweiterter Tuner, der Verkehrsnachrichten verarbeiten kann
0x42	TVTuner	
0x50	Telephone	
0x51	GeneralPhoneBook	
0xFF	UniqueFunction	Zeit- und Datumsfunktionen, können in anderem Funktionsblock implementiert werden

Tabelle 1: Im Standard-Funktionskatalog von MOST definierte Funktionsblöcke

Alternativ können Informationen über Geräte, Funktionsblöcke und einzelne Funktionen durch Introspektion zur Laufzeit abgefragt werden. Der Introspektionsmechanismus ist – ähnlich wie bei Java – selbst wiederum durch Funktionen realisiert.

Neben dem Aufruf entfernter Operationen steht bei MOST auch ein Mechanismus zur ereignisbasierten Kommunikation zur Verfügung, der so genannte *Notifikationsdienst*. Über den Notifikationsdienst können sich Geräte für die Benachrichtigung bei Änderungen von Eigenschaften registrieren. Dabei kön-

nen sie sich für einzelne Eigenschaften oder alle Eigenschaften eines Funktionsblocks anmelden. Die Benachrichtigung über die Änderungen erfolgt dann durch Kontrollnachrichten.

2.1.3 NW-Schicht

Bei der Datenübertragung wird eine gepulste Kodierung verwendet, wodurch geringere Bitfehlerraten und eine einfachere Synchronisation ermöglicht werden. Von der Verwendung von Lichtwellenleitern verspricht man sich ebenfalls geringe Bitfehlerraten.

Für die Synchronisation gibt das Mastergerät ein Synchronisationssignal in einem festen Zeittakt vor (Time Division Multiplexing, TDM), die anderen Geräte (Slaves) orientieren sich an diesem Signal. Die Frequenz des Zeittaktes beträgt standardmäßig 44,1 kHz. Damit wird die auf dem MOST-Bus vorhandene Gesamtbandbreite in Rahmen mit jeweils 64 Bytes unterteilt. Jeder Rahmen ist dabei in mehrere Sektionen unterteilt (Abbildung 2):

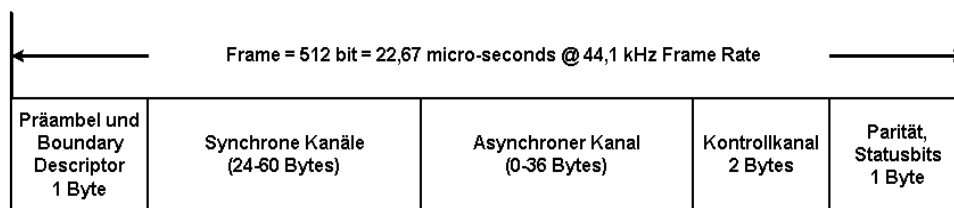


Abbildung 2: Aufbau eines MOST-Rahmens

- Die erste Sektion (Byte 0) enthält das Synchronisationssignal (*Präambel*) und den so genannten *Boundary Descriptor*, der die Größe des synchronen und des asynchronen Bereichs festlegt. Zusammen haben diese beiden Bereiche eine konstante Größe von 60 Bytes. Der Wert des Boundary Descriptors kann zur Laufzeit geändert werden.
- Für die Übertragung der Nutzdaten stehen insgesamt 62 Bytes in einem Rahmen zur Verfügung (Bytes 1 bis 62). Der Nutzdatenbereich ist dabei wiederum in drei Teile untergliedert: Es gibt einen Teil, der für die synchronen Kanäle reserviert ist, einen zweiten Bereich für den asynchronen Kanal und schließlich einen kleinen Teil, der für den Kontrollkanal vorgesehen ist.

- Den Abschluss (Byte 63) bilden wiederum administrative Daten. Dazu gehören Paritäts- und Statusinformationen.

2.1.3.1 Mechanismen zur Datenübertragung

Entsprechend den unterschiedlichen Anforderungen durch die zu übertragenden Daten haben die verschiedenen Übertragungskanäle auf dem MOST-Bus unterschiedliche Eigenschaften.

Der **Kontrollkanal** dient vorwiegend zur Übertragung administrativer Daten und bietet lediglich eine relativ kleine Bandbreite. Auf Grund der Größe des Kontrolldatenbereichs pro MOST-Rahmen steht eine Brutto-Bandbreite von 690 Kilobit pro Sekunde auf dem Kontrollkanal zur Verfügung, was einem Maximum von circa 3000 Kontrollnachrichten pro Sekunde entspricht.

Als Zugriffsverfahren auf den MOST-Kontrollkanal wird der wahlfreie Zugriff mit Abhören (CSMA) benutzt, um möglichst kleine Verzögerungen beim Buszugriff zu erzielen. Der Transport auf dem Kontrollkanal bietet unterschiedliche Prioritäten für die Kontrollnachrichten an. Dabei haben standardmäßig Systemnachrichten (z.B. zur Reservierung oder Freigabe von synchronen Kanälen) und vom Endbenutzer initiierte Kontrollnachrichten eine höhere Priorität. Somit kann eine vergleichsweise schnelle Reaktion auf Befehle des Benutzers erreicht werden.

Der grundlegende Transportdienst auf dem Kontrollkanal (*Control Message Service, CMS*) bietet den Transfer von Kontrollnachrichten mit einer festen Länge von 32 Bytes. Davon stehen maximal 12 Bytes für Nutzdaten wie Parameter oder Rückgabewerte zur Verfügung. Die Integrität der zu übertragenden Kontrolldaten wird durch Prüfsummen gesichert. Zusätzlich gibt es positive und negative Bestätigungen, um Nachrichtenempfänger vor Überlastung zu schützen (Flusskontrolle) und eine zuverlässige Übertragung sicherzustellen. Wurde eine Nachricht vom Empfänger mit einer negativen Bestätigung quittiert oder wurde anhand der Prüfsumme eine Verletzung der Integrität übertragener Daten festgestellt, so wird die Übertragung der entsprechenden Kontrollnachricht nach einem Timeout (Standardwert: fünf Millisekunden) automatisch wiederholt.

Da in einem einzelnen Telegramm des CMS lediglich 12 Bytes an Nutzdaten transportiert werden können, ergibt sich die Notwendigkeit, größere Mengen an Nutzdaten in segmentierten Nachrichten zu übertragen. Diese Funktionalität bietet der Application Message Service (AMS), der auf dem CMS aufbaut.

Mit Hilfe des AMS können Nachrichten mit bis zu 65535 Bytes an Nutzdaten verschickt werden.

Der **asynchrone Kanal** dient der Übertragung von Anwendungsdaten, für die zeitweise eine hohe Bandbreite benötigt wird (Bursts). Typische Beispiele für solche Daten stellen Grafiken oder Kartendaten vom Navigationssystem dar. Über den asynchronen Kanal werden Pakete mit einer festen Länge versendet. Die Integrität der zu übertragenden Daten wird dabei durch einen Prüfsummenmechanismus gesichert, aber im Gegensatz zum Kontrollkanal stellt der asynchrone Kanal keine Empfangsbestätigungen und damit auch keine automatische Wiederholung der Datenübertragung zur Verfügung. Stattdessen kann die Zuverlässigkeit der Datenübertragung erst durch die Verwendung eines entsprechenden Transportprotokolls auf dem asynchronen Kanal gewährleistet werden. Die MOST-Spezifikation schlägt das MOST High Protocol [40] für diesen Zweck vor. Zur Steuerung des Zugriffs auf den asynchronen Kanal wird ein Token-Verfahren verwendet, um einen guten Datendurchsatz bei einer hohen Auslastung zu gewährleisten.

Um kontinuierliche Datenströme zu übertragen, die eine beständig hohe Bandbreite benötigen, stehen bei MOST die **synchronen Kanäle** zur Verfügung. Typischerweise fallen Audio- und Videodaten in diese Kategorie. Die für die synchrone Datenübertragung zur Verfügung stehende Gesamtbandbreite wird dabei in mehrere (physische) Kanäle unterteilt. Jeder physische Kanal belegt dabei innerhalb eines Rahmens ein Byte, insgesamt sind so zwischen 24 und 60 physischen Kanälen verfügbar.

MOST-Funktionsblöcke können die Rolle einer Datenquelle einnehmen und die physischen Kanäle einzeln reservieren oder freigeben. Sie können dabei auch mehrere reservierte physische Kanäle zu einem logischen Kanal mit einem eindeutigen Bezeichner zusammenfassen. Ein logischer Kanal entspricht somit dem Konzept einer virtuellen Leitung. Andere Funktionsblöcke können als Datensinken fungieren und damit die auf einem logischen Kanal übertragenen Daten erhalten. Die Administration der synchronen Kanäle erfolgt über den Kontrollkanal.

Dadurch, dass bei der Übertragung eines kontinuierlichen Datenstroms immer jeweils ein logischer Kanal benutzt wird, ist es nicht nötig, diese Daten weiter in Pakete zu unterteilen und mit Adressierungsinformationen zu versehen. Somit können die Daten sehr bandbreiteneffizient über die synchronen Kanäle übertragen werden.

2.1.3.2 Adressierung

Für die nachrichtenorientierte Kommunikation über den asynchronen Kanal bzw. den Kontrollkanal stehen mehrere Adressierungsarten zur Verfügung.

Jedes Gerät kann einzeln über seine physische oder seine logische Adresse adressiert werden. Die physische Adresse errechnet sich dabei immer durch die Position im Ring relativ zum Mastergerät zuzüglich einem festen Offset ($0x0400$).

Die logische Adresse eines Gerätes kann prinzipiell beliebig aus einem bestimmten Wertebereich genommen werden, genauso, wie man jedem Gerät eine (innerhalb des gültigen Wertebereichs) beliebige Gruppenadresse zuordnen kann (vgl. Tabelle 2). Allerdings werden logische Geräteadressen und Gruppenadressen in MOST-Geräten gemäß der MOST-Spezifikation nur in flüchtigem Speicher abgelegt. Nach dem Reset wird einem Gerät daher eine neue logische Geräte- und eine neue Gruppenadresse zugeordnet und die Repräsentation der MOST-Konfiguration muss aktualisiert werden. Um diesen Aufwand zu vermeiden, wird zumindest das Konzept der logischen Geräteadressen aufgeweicht: Anstatt einer beliebigen Abbildung der physischen auf die logische Adresse wird für die logische Geräteadresse lediglich ein anderer Offset ($0x0100$) als bei der physischen verwendet.

Neben der Einzel- und Gruppenadressierung gibt es bei MOST auch die Möglichkeit, mittels eines Broadcasts eine Nachricht an alle Geräte im Ring zu verschicken.

Art der Adressen	Adressbereich	Anmerkung
Physische Geräteadressen	0x0400..0x04FF	Adressierung gemäß Position im MOST-Ring
Logische Geräteadressen	0x0001..0x02FF 0x0500..0xFFFF	Prinzipiell pro Gerät beliebig wählbar (solange eindeutig), aber normalerweise $0x0100 + \text{Ringposition}$
Gruppenadressen	0x0300..0x03C7 0x03C9..0x03FF	Gruppenadresse besteht aus Offset ($0x0300$) + Gruppennummer

Broadcastadresse	0x03C8	Spezielle Gruppenadresse, Gruppennummer C8

Tabelle 2: MOST-Adressmodi und -bereiche

MOST stellt auf der Ebene des Transceiverchips und in den Net Services Mechanismen bereit, um die physische Adresse automatisch zu bestimmen und um die Eindeutigkeit von Gruppenadressen und logischen Geräteadressen sicherzustellen.

Die Adressierung von Anwendungen und Bestandteilen der öffentlichen Schnittstellen von Anwendungen wird im Abschnitt 2.1.2 beschrieben.

2.1.3.3 Verwaltung der MOST-Konfiguration

Anwendungen in einem MOST-System (Funktionsblöcke) wissen im Idealfall nichts über die verschiedenen Geräte im System und deren Adressen. Sie orientieren sich stattdessen an den im System vorhandenen Anwendungen, die jeweils eine abgegrenzte Funktionalität anbieten. Da Anwendungen in einem MOST-System durch Funktionsblöcke repräsentiert werden, geben sie beim Aufruf entfernter Funktionen nur den Typ des FBlocks und die ID der gewünschten Instanz an, um diese zu adressieren.

Es ist Aufgabe der Net Services, diese Information (*FBlockID.InstID*) bei einem entfernten Aufruf um die Angabe der Geräteadresse zu erweitern.

Aus diesem Grund implementieren Geräte, die andere Geräte steuern (so genannte Controller) eine Tabelle, welche die Abbildung der Angabe eines Funktionsblocks auf die (logische) Geräteadresse enthält. Diese Tabelle wird in der MOST-Terminologie als **Decentral Registry** bezeichnet. Die Decentral Registry wird bei Bedarf durch einen Introspektionsmechanismus oder durch eine Abfrage der Central Registry initialisiert.

Der Begriff **Central Registry** bezeichnet die Beschreibung der MOST-Systemkonfiguration. Die Central Registry wird von einem dedizierten Funktionsblock des Mastergerätes (*NetworkMaster-FBlock*) während der Netzwerkinitialisierung aufgebaut, indem jeder Knoten im System mittels *FBlockIDs.Get*-Nachrichten nach den zur Verfügung gestellten Funktionsblö-

cken befragt wird. Für die Anfrage nach den Funktionsblöcken kommt dabei der ganz normale Anfragemechanismus zum Zuge, den MOST-Anwendungen zur Interaktion untereinander ebenfalls verwenden. Interessanterweise müssen die Anfragen der Central Registry aber innerhalb einer Zeitspanne von 50 Millisekunden beantwortet werden, wogegen für Anfragen im Kontext von Anwendungen jeweils ein Timeout von 200 Millisekunden gilt.

Der Inhalt der Central Registry besteht aus einer Tabelle, die beschreibt, welche logische Adresse und welche Funktionsblöcke jedes Gerät im Ring hat. Tabelle 3 zeigt den beispielhaften Aufbau eines Eintrags für ein Mastergerät in der Central Registry.

Sobald der NetworkMaster die Central Registry vollständig aufgebaut hat, schickt er eine Meldung *Configuration.Status.OK* an alle Knoten im Ring.

Falls er feststellt, dass sich etwas an der Konfiguration gegenüber dem letzten Stand geändert hat, sendet er die Meldung *Configuration.Status.NotOK* als Broadcast-Nachricht. Damit informiert er alle Geräte, dass die verschiedenen Decentral Registries ungültig sind und neu aufgebaut werden müssen.

Logische Adresse	Position im Ring	FBlockID	InstID
0x0100	0	AMFMTuner	1
		NetworkMaster	0
		Connection Master	0

Tabelle 3: Beispielhafte Gerätebeschreibung in der Central Registry

Im Rahmen der Netzwerkinitialisierung legen die Net Services jedes Gerätes die Instanz-IDs der zur Verfügung gestellten Funktionsblöcke selbst fest. Sie sind dabei auch dafür verantwortlich, die Eindeutigkeit der Bezeichner zu gewährleisten. Ebenso wie die Registries werden die Instanz-IDs in flüchtigem Speicher verwaltet, so dass sich nach jedem Neustart eines Gerätes der Bezeichner einer FBlock-Instanz ändern kann.

2.1.3.4 Weitere Funktionen der Netzwerkverwaltung

Neben dem *NetworkMaster*-Funktionsblock, der die Central Registry eines MOST-Systems verwaltet, gibt es weitere Funktionsblöcke, die an der Netzwerkverwaltung beteiligt sind.

So implementiert jedes Gerät einen Funktionsblock mit der Bezeichnung *NetBlock*. Dieser Funktionsblock stellt als grundlegende Informationen die verwendete Samplingfrequenz, die Geräteadressen und die implementierten Funktionsblöcke bereit. Daneben erlaubt dieser Funktionsblock das geordnete Herunterfahren eines Geräts durch den Aufruf von *NetBlock.Shutdown* und die Verwaltung der Ereignisbenachrichtigungen.

Das Mastergerät implementiert neben dem *NetworkMaster*-Funktionsblock noch zwei weitere obligatorische Funktionsblöcke für die Verwaltung des MOST-Netzwerks:

- Der *PowerMaster*-Funktionsblock kümmert sich um das Energiemanagement innerhalb des MOST-Systems. Zu diesem Zweck schaltet er beispielsweise das Lichtsignal an, wenn er von anderen Komponenten im Auto benachrichtigt wird. Um die Fahrzeugbatterie zu schonen, startet er nach einer gewissen Phase der Untätigkeit den Shutdown-Mechanismus, der den MOST-Bus und die daran angeschlossenen Geräte abschaltet. Unterstützt wird das Energiemanagement von der Fähigkeit des MOST-Transceiverchips, nicht benötigte Teile von sich selbst abzuschalten. Dabei können alle Teile des Chips bis auf eine kleine Wakeup-Logik abgeschaltet werden.
- Der *ConnectionMaster*-Funktionsblock verwaltet die Reservierung und die Freigabe synchroner Verbindungen auf einer Abstraktionsebene, die der Benutzung aus Anwendungen heraus angemessen ist. Dazu setzt er, transparent für die Anwendungen, beispielsweise den Wert des Boundary Descriptors so, dass genügend Bandbreite für die synchrone Datenübertragung zur Verfügung steht. Verbindungsanfragen von Datenquellen oder Datensinken setzt er in die entsprechenden Kontrollnachrichten zur Allokation bzw. Deallokation synchroner Kanäle um.

Für einen stabilen Betrieb definiert die MOST-Spezifikation [42] zahlreiche Timing-Bedingungen, die MOST-Geräte erfüllen müssen. So sind z.B. Timeouts für die Initialisierung der Netzwerk-Schnittstelle, die Fehlerbehandlung oder den Abschaltvorgang von Geräten vorgeschrieben.

Systemereignisse, die für administrative Funktionsblöcke oder für Anwendun-

gen interessant sein könnten, werden von der Netzwerk-Schnittstelle in die oberen Schichten propagiert. Dazu gehören insbesondere Ereignisse, die den Zustand des MOST-Busses, Änderungen an der MOST-Konfiguration oder den Zustand der Spannungsversorgung beschreiben.

2.1.4 Probleme beim Betrieb von MOST-Systemen

Insgesamt bestehen beim Betrieb von MOST-Systemen noch zahlreiche Probleme. Von diesen Problemen sollen an dieser Stelle nur die wichtigsten ange-rissen werden.

Ein Problem besteht darin, dass der MOST-Bus in Ringtopologie unidirektional ausgelegt ist. Falls sich ein MOST-Gerät mit aktiver Netzwerkschnittstelle fehlerhaft verhält, besteht die Möglichkeit, dass die gesamte Kommunikation auf dem Bus gestört wird. Jedes aktive Gerät stellt somit einen zentralen Ausfallpunkt (Single Point Of Failure) dar.

Wird das Mastergerät durch zahlreiche Nachrichten überlastet, kann es zum Ausfall dieses Gerätes und damit des gesamten MOST-Systems kommen.

Bei einer Unstabilität des Netzwerks kann es kurzzeitigen Verlusten des Licht- oder des Synchronisationssignals kommen. Dies ist durchaus ein praxisrelevantes Problem, da es im Laufe der Zeit auf Grund der mechanischen Beanspruchung im Fahrzeug zu defekten Kabeln oder Steckverbindungen kommen kann.

Ein weiteres Problem besteht in dem Auftreten von vorübergehenden Inkonsistenzen der Konfiguration - einzelne Geräte oder Funktionsblöcke fehlen beim Hochfahren und dem anschließenden Betrieb des Telematiksystems. Besonders störend sind Fehler beim Herunterfahren des Systems nach dem Abstellen des Motors. Kommt es dabei zu Störungen, werden in einigen Fällen der MOST-Bus und damit die Telematikgeräte nicht heruntergefahren und beziehen weiter Strom. Da die Lichtmaschine bei abgeschaltetem Motor nicht mehr läuft, kann es passieren, dass die Fahrzeugbatterie dabei vollständig entladen wird.

Bei der Kommunikation kann es ebenfalls zu Problemen kommen. Es sind Fehlerfälle bekannt, in denen der Kontrollkanal – über längere Zeiträume - vollständig ausgelastet war. Von der Startphase abgesehen wird die dort zur Verfügung stehende Bandbreite üblicherweise nur zu einem kleinen Bruchteil ausgelastet. Andere Kommunikationsprobleme bestehen in ausbleibenden, zu spät oder unerwartet auftretenden Antwort- und Notifikationsnachrichten.

Schließlich kommt es in der Praxis auch zu Konflikten bei der Administration

der synchronen Kanäle, wie z.B. dem Fehlschlagen der Reservierung freier synchroner Kanäle.

2.2 Autonomic Computing

In Abschnitt 1.2.1 wurde der Begriff des Autonomic Computing bereits eingeführt. An dieser Stelle werden nun Forschungsprojekte vorgestellt, die sich mit dem Thema beschäftigen. Den Abschluss bildet dabei eine Beschreibung des Projekts *Self-healing In-vehicle Telematics Systems*, das Anforderungen und Vorgehensweisen für autonome Fahrzeugtelematiksysteme untersucht.

2.2.1 Architecture Based Languages and Environments

Das Forschungsprojekt ABLE (Architecture Based Languages and Environments) an der Carnegie Mellon University befasst sich in einem Teilprojekt mit der Rolle von Softwarearchitekturen für autonome, adaptive Systeme. Dazu wird ein autonomes System formal beschrieben, indem die Komponenten, die Beziehungen zwischen den Komponenten und Regeln bezüglich der Komponenten und Beziehungen definiert werden. Anhand dieses Architekturmodells wird ein konkretes System instrumentiert, um bestimmte Systemparameter zu beobachten, auszuwerten und – wenn nötig – entsprechende Reparaturmaßnahmen einzuleiten. Das Architekturmodell wird dabei zur Laufzeit aktualisiert, wenn sich die Konfiguration des Systems ändert.

Um Systeme gemäss dem Architekturmodell einfacher zu realisieren, steht ein Framework für sich selbst adaptierende Systeme zur Verfügung. Dieses Framework besteht aus drei Schichten (vgl. Abbildung 3). Die unterste Schicht (*Runtime Layer*) stellt dabei die eigentliche laufende Anwendung dar, die mit Messsensoren (*Probes*) instrumentiert wurde. Die mittlere Schicht (*Model Layer*) verwaltet das Architekturmodell, während die oberste Schicht (*Task Layer*) benutzerdefinierte Anforderungen und Aufgaben verwaltet. Brücken zwischen den einzelnen Schichten (*Abstraction Bridges*) propagieren und übersetzen Statusinformationen zur nächst oberen Schicht beziehungsweise Anweisungen zur Adaption zur darunter liegenden Schicht.

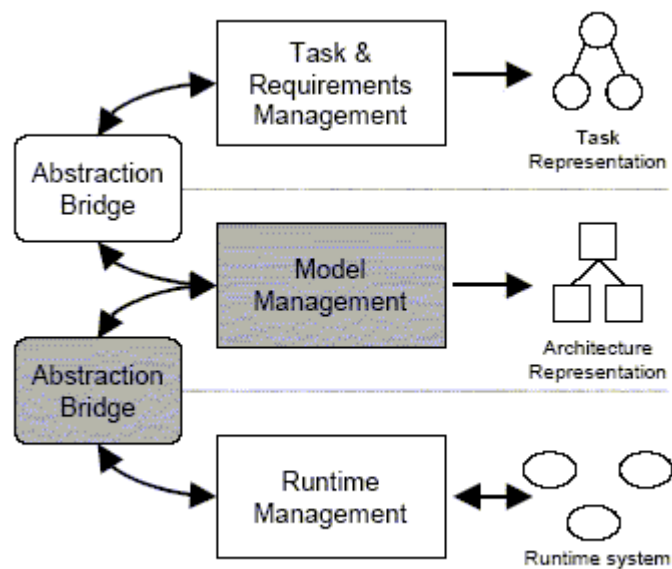


Abbildung 3: Framework für automatische Systemadaption [17]

Abbildung 4 zeigt das Zusammenspiel der Frameworkkomponenten und die zeitliche Abfolge bei der Selbstadaption:

1. Das Laufzeitverhalten des laufenden Systems (1) wird überwacht (2).
2. Die beobachteten Werte der Systemparameter werden abstrahiert und auf Architektureigenschaften abgebildet (3).
3. Wenn sich eine Architektureigenschaft ändert, wird geprüft, ob die Regeln des Architekturmodells verletzt werden (4).
4. Bei einer Regelverletzung wird eine Reparaturmaßnahme angestoßen, welche die Architektur anpasst (5).
5. Änderungen am Architekturmodell werden schließlich zum laufenden System propagiert (6).

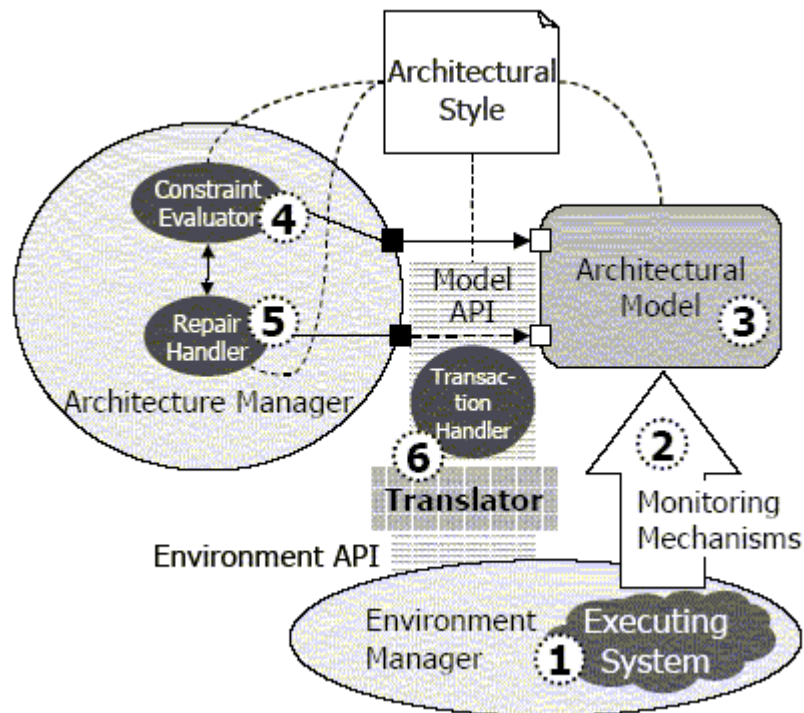


Abbildung 4: Zusammenspiel zwischen Framework und Anwendungssystem [9]

Dadurch, dass der Aspekt der Autonomie in der Architekturbeschreibung spezifiziert und das System entsprechend instrumentiert wird, kann die eigentliche Anwendungslogik davon freigehalten werden. Somit können die Mechanismen zur Selbstbeobachtung und Selbstadaption einfacher wiederverwendet werden.

2.2.2 IBM Autonomic Computing

IBM setzt bei dem Forschungsprojekt *Agent Building and Learning Environment* [2,3] auf ein Framework, mit dem sich intelligente, autonome Agenten erstellen lassen, die miteinander interagieren. Jeder Agent führt unterschiedliche Aufgaben durch, z.B. die Überwachung des Systems, die Aggregation und Interpretation der beobachteten Daten oder das Durchführen von entsprechenden Maßnahmen zur Systemadaption. Ein einzelner Agent (*AbleAgent*) ist wiederum aus mehreren Basiskomponenten, so genannten *AbleBeans* zusammen-

gesetzt. Da sich die Agenten autonom und intelligent verhalten sollen, liefert IBM im Rahmen des Frameworks zahlreiche AbleBeans mit, die Verfahren für maschinelles Lernen, die Verarbeitung von wissensbasierten Regeln und andere Algorithmen realisieren.

Um Entwickler bei der Benutzung des Frameworks zu unterstützen, stehen Entwicklungsumgebungen zur Verfügung, mit denen Agenten grafisch aus vorhandenen Komponenten zusammengestellt werden können.

Im Rahmen der *Autonomic Computing*-Initiative vermarktet IBM schon konkrete Ausprägungen des Frameworks, etwa als automatisches Administrationswerkzeug für die IBM eServer Produktlinie [20].

2.2.3 Recovery-Oriented Computing

Einen interessanten Ansatz verfolgt das Recovery Oriented Computing-Projekt (ROC) der Universitäten Berkeley und Stanford [48]. Beim ROC wird davon ausgegangen, dass Softwaresysteme in der Praxis häufig nur unzureichend spezifiziert sind, da es extrem aufwändig wäre, alle möglichen Zustände eines komplexen Systems zu dokumentieren und die Dokumentation immer auf dem aktuellen Stand zu halten. Dementsprechend schwierig gestaltet sich die vollständige Vermeidung von Fehlern in der Software. Daher soll der Fokus nicht ausschließlich auf die Fehlervermeidung, sondern auch auf die schnelle und sichere Wiederherstellung eines gültigen Zustands nach dem Auftreten eines Fehlers gelegt werden.

Ein gültiger Zustand lässt sich unter Umständen durch einen gezielten Crash und einem anschließenden Neustart (*Crash Restart*) schneller erreichen als durch ordnungsgemäßes Herunterfahren und Neustarten oder andere Reparaturmaßnahmen. Dadurch sinkt die Ausfallzeit eines Systems nach einem Fehler. Da bei einem Crash wichtige Daten verloren gehen können, muss ein System so entworfen werden, dass es Abstürze tolerieren kann. Dazu müssen wichtige Daten rechtzeitig persistent gespeichert werden, während bei anderen, weniger wichtigen, Daten auf eine persistente Speicherung verzichtet werden kann.

Weiterhin ist es nötig genau zu wissen, inwiefern Komponenten an der Verwaltung komponentenübergreifender Zustände beteiligt sind. Mit diesem Wissen können einzelne Komponenten bei Fehlfunktionen gezielt zum Absturz gebracht und neu gestartet werden. Es muss also sichergestellt werden, dass die Fehlerursache möglichst eng eingegrenzt und für die richtige (und mög-

lichst kleine) Menge von Komponenten die Wiederherstellung (*Recovery*) angestoßen wird.

In [8] beschreiben Candea et al. dazu eine Technik (*Automatic Failure Path Inference*, kurz AFPI), bei der Middleware für die Aufzeichnung von Fehlern und deren Ausbreitung instrumentiert wird. Durch die Aufzeichnung der Fehlerausbreitung lassen sich automatisch die Abhängigkeiten zwischen den Komponenten erkennen. Die Fehlerausbreitung AFPI arbeitet dabei mit zwei Phasen:

- In der **Trainingsphase** werden absichtlich Fehler in das System injiziert und durch die Beobachtung des Systemverhaltens ein *Fehlerausbreitungsgraph* (*F-map*) aufgebaut, der die Komponentenabhängigkeiten in Bezug auf die Fehlerausbreitung beschreibt.

In dem bisher realisierten Prototyp werden zur Beobachtung der Fehlerausbreitung Java-Exceptions verwendet. Zu Beginn der Trainingsphase wird mit Hilfe von Introspektionsmechanismen festgestellt, welche Exceptions bei der Ausführung welcher Methode auftreten können.

Diese Informationen werden dann zur Fehlerinjektion benutzt. Tritt eine Exception auf, so kann mit Hilfe von Stacktraces festgestellt werden, wie die entsprechende Exception propagiert wurde. Daraus lassen sich dann die Komponentenabhängigkeiten ableiten.

- In der **Produktivphase** werden Fehler anhand der Informationen in der F-map diagnostiziert. Falls neue Fehler auftreten, wird die F-map zur Laufzeit aktualisiert.

Durch die Verwendung eines automatischen Mechanismus werden die Probleme manuell erstellter und gepflegter Informationen über die einzelnen Komponenten und deren Abhängigkeiten umgangen. Es wird sichergestellt, dass das zur Verfügung stehende Wissen stets aktuell und vollständig ist.

2.2.4 Self-healing In-vehicle Telematics Systems (SeHeT)

Das SeHeT-Projekt [32] beschäftigt sich mit der Anwendung von Mechanismen zur Selbstheilung in Fahrzeugtelematiksystemen. Dabei wurden zunächst die Besonderheiten von Fahrzeugtelematiksystemen in Bezug auf Selbstheilungsmechanismen untersucht (vgl. Abschnitt 1.2.1). Darauf aufbauend wurden An-

forderungen an autonome Fahrzeugtelematiksysteme formuliert und Mechanismen zur Selbstheilung im Bereich der Fahrzeugtelematik vorgestellt.

Die vorgestellten Mechanismen und ihre vorgeschlagene Verwendung ergeben dabei eine Architektur wie in Abbildung 5 dargestellt.

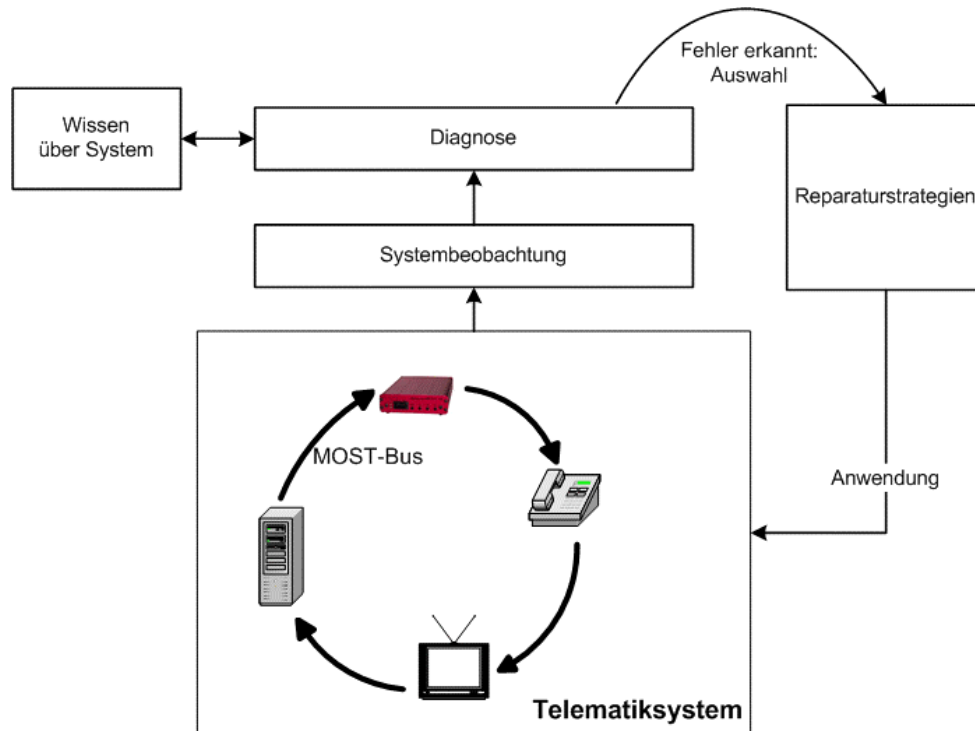


Abbildung 5: SeHeT-Architektur (nach [32])

Die vorgeschlagenen Schritte und Mechanismen sind dabei im Einzelnen:

1. Der erste Schritt besteht in der **Definition des Normalzustands** des Telematiksystems. Dabei bestehen zwei Möglichkeiten. Eine Möglichkeit ist die Spezifikation des Normalzustands und des gewünschten Verhaltens. Mit Hilfe von (gefärbten) Petrinetzen [10] können die internen Zustände von Komponenten und die Übergänge dazwischen spezifiziert werden. Mit Hilfe von Sequenzdiagrammen, die den Nachrichtenfluss zwischen verteilten Komponenten beschreiben (Message Sequence Charts, [27]), können die Interaktionen zwischen den einzelnen Komponenten vorgeschrieben werden. Bei beiden Verfahren besteht das Problem, dass die Systemspezifikation möglicherweise unvollständig ist oder sich nach Änderungen am System nicht mehr auf dem aktuellen Stand befindet. Als Alternative stellt sich die Gewinnung der Definition des Normalzu-

stands durch eine Trainingsphase an einem fehlerfreiem System dar. Die Schwierigkeit bei diesem Vorgehen besteht in der Sicherstellung eines fehlerfreien Betriebs während dieser Phase.

2. Im nächsten Schritt, dem **System-Monitoring**, werden das Systemverhalten und das applikationsspezifische Verhalten beobachtet. Die Beobachtung des Systemverhaltens bezieht sich dabei auf systemspezifische Parameter auf einer niedrigen Abstraktionsebene. Bei den in dieser Arbeit betrachteten MOST-Systemen sind das beispielsweise die Buslast oder die Anzahl von Low-Level Retries.

Für die Beobachtung des applikationsspezifischen Verhaltens wird die Kommunikation der verschiedenen Anwendungen mit den – durch Message Sequence Charts - spezifizierten Abläufen verglichen. Diese Funktion übernimmt eine zentrale Komponente. Die Beobachtung erfolgt dabei vorrangig passiv, um das System möglichst wenig zu beeinflussen. Es soll aber auch die Möglichkeit bestehen, den Zustand des Systems und der einzelnen Geräte aktiv in Erfahrung zu bringen. Zu diesem Zweck müssen die einzelnen Geräte und Komponenten spezielle Diagnoseschnittstellen implementieren, über die ihr Zustand abgefragt werden kann.

Signifikante Abweichungen vom definierten Normalzustand werden als Fehlersymptome aufgefasst. Die Fehlersymptome werden der Diagnosekomponente zur weiteren Untersuchung mitgeteilt. Alternativ können die Fehlersymptome auch durch einen Entwicklungsingenieur ausgewertet werden.

3. Die **Diagnose** soll, ausgehend von den Fehlersymptomen, die verursachende Komponente bzw. die verursachenden Komponenten finden und den Fehler genau identifizieren sowie in einer für Menschen verständlichen Form beschreiben. Realisiert werden soll die Diagnose durch die Verwendung von Regeln, die bestimmte Fehlersymptome mit der Fehlerursache verknüpfen (wissensbasiertes System). Falls die von der Beobachterkomponente bereitgestellten Informationen nicht ausreichen, muss die Diagnosekomponente darüber hinaus in der Lage sein, den Zustand einzelner Komponenten aktiv zu erfragen. Die einfachste Möglichkeit dazu wäre das „Anpingen“ der entsprechenden Komponenten, um festzustellen, ob sie noch lauffähig sind.

Die Diagnoseergebnisse können anschließend zur Selbstheilung des Systems verwendet werden, indem die passenden Reparaturmechanismen angestoßen werden. Daneben soll das Diagnoseergebnis auch direkt durch Wartungspersonal ausgewertet werden können.

4. An die **Reparaturmechanismen**, die in einem Fahrzeugtelematiksystem zum Einsatz kommen, werden die folgenden Anforderungen gestellt:
Sie sollen möglichst einfach und effizient zu realisieren sein und sie sollen

das Telematiksystem möglichst wenig beeinträchtigen. Als sinnvolle Reihenfolge bei der Anwendung von Reparaturmaßnahmen wird erachtet, zunächst die Maßnahmen durchzuführen, die das System am geringsten beeinträchtigen. Durch die Anwendung in dieser Reihenfolge können weiche, transiente Fehler (*soft faults*, vgl. [30]) toleriert werden, das System bleibt voll funktionsfähig.

Als Reparaturmechanismen kommen die Wiederherstellung eines gültigen Zustands (einer Komponente), der Neustart einer einzelnen Komponente, der Neustart des gesamten Telematiksystems oder die Abschaltung einzelner fehlerhafter Komponenten in Frage.

Die Wiederherstellung eines gültigen Komponentenzustands ist vermutlich die aufwändigste Reparaturstrategie, allerdings auch diejenige, welche das System am geringsten beeinflusst. Nach einem Neustart von Komponenten oder des Systems steht weiterhin die volle Funktionalität des Systems zur Verfügung. Allerdings kann insbesondere ein Systemneustart relativ viel Zeit in Anspruch nehmen, so dass er mit Sicherheit vom Kunden registriert wird. Falls die anderen Reparaturmaßnahmen keinen Erfolg bringen, bleibt als letzte Reparaturmaßnahme nur die Abschaltung von fehlerhaften Komponenten. Damit lassen sich dann auch so genannte harte Fehler (*hard faults*, vgl. [30]) beheben.

Die vorgestellten Konzepte wurden mit einem prototypischen Framework auf ihre Tauglichkeit hin überprüft. Dazu wurden in ein mit dem Framework erstelltes System Fehler injiziert, die erkannt und behoben wurden. Die injizierten Fehler waren allerdings noch recht einschneidender Natur, zum Beispiel Endlosschleifen oder unbehandelte Ausnahmesituationen. Es ist noch offen, wie gut weiche Fehler erkannt werden können, die das System nicht sofort offensichtlich beeinträchtigen.

2.3 Monitoring

Der Begriff *Monitoring* beschreibt allgemein die Beobachtung und Überwachung eines Systems oder einer Komponente innerhalb eines Systems. Die Abgrenzung zwischen Monitoring- und Diagnosefunktionen ist in der Literatur dabei fließend. In der vorliegenden Arbeit umfasst der Begriff Monitoring nur die Überwachung von direkt verfügbaren Systemparametern. Die Informationen, die dabei gewonnen werden, können gefiltert und zu statistischen Werten aggregiert werden.

Die Interpretation der beim Monitoring gewonnenen Daten auf einer höheren Abstraktionsebene fällt dagegen in den Bereich der *Diagnose*.

Eine Monitoring-Lösung muss aussagekräftige Informationen über den Zustand eines Systems liefern. Dabei ist zu beachten, dass beim Vorgang der Beobachtung sehr viele Daten anfallen können, die ausgewertet werden müssen. Dazu muss eine Monitoring-Lösung sehr effizient realisiert sein. Diese beiden Anforderungen beschreiben zusammengefasst die *Leistungsfähigkeit* einer Monitoring-Lösung. Ein weiterer wichtiger Gesichtspunkt ist die *Beeinflussung des Systems* durch das Monitoring. Damit der Betrieb eines Systems nicht eingeschränkt wird und die Ergebnisse durch die Messung selbst nicht verfälscht werden, ist es wichtig, dass die Beeinflussung möglichst gering ist.

Wird ein einzelner Rechner beobachtet, so spricht man von *Host-Monitoring*. Bei der Überwachung eines einzelnen Rechners werden typischerweise Dateizugriffe, Prozesse, Ereignislogs und Benutzerinteraktionen wie Logins, Tastatureingaben oder die Verwendung von Programmen überwacht und analysiert [33].

Unter dem Begriff *Netzwerk-Monitoring* versteht man die Beobachtung des Netzwerkverkehrs um daraus Rückschlüsse auf den Systemzustand zu ziehen [33].

System-Monitoring befasst sich, wie der Name schon sagt, mit der Überwachung von Systemen. Somit umfasst es sowohl Host- als auch Netzwerk-Monitoring. Im weiteren Verlauf dieser Arbeit wird im wesentlichen auf letzteres eingegangen.

2.3.1 Netzwerk-Management und Netzwerk-Monitoring

Mit der zunehmenden Komplexität und Verteilung von Systemen gewinnen Rechnernetze immer weiter an Bedeutung. Da sie heutzutage in vielen Systemen ein zentrales Element der Infrastruktur darstellen, wird es immer wichtiger, den genauen Zustand eines Rechnernetzes zu kennen und es nötigenfalls zu warten. Diese Aufgabe wird als *Netzwerk-Management* bezeichnet. Auf Grund der Komplexität moderner Rechnernetze ist es für Netzwerkadministratoren schwierig, den Überblick zu behalten und die richtigen Einschätzungen zu treffen. Daher nehmen Verfahren zur automatisierten Verwaltung von Netzen eine wichtige Rolle ein.

Nach [26] umfasst das Netzwerk-Management fünf Aspekte:

- **Performance Management:** Das Ziel beim Performance Management besteht in der Messung der Performanz und der Bereitstellung dieser Informationen. Da auf Ebene der Netzwerkschicht die vom Anwender wahrgenommene Performanz nicht direkt gemessen werden kann, werden Indikatoren wie Durchsatz, Auslastung, Verfügbarkeit oder Antwortzeiten gemessen. Anhand der Veränderung solcher Indikatoren lässt sich dann die Performanz auch vorhersagen (so genanntes *Proaktives Performance Monitoring*). Werden für die einzelnen Indikatoren Schwellwerte festgelegt, kann damit der Netzwerkadministrator von Performanz-Engpässen automatisch benachrichtigt werden.
Ferner können die Informationen über die Performanz eines Rechnernetzes zur Planung einer möglichen Netzwerkerweiterung genutzt werden.
Damit die Performanz und ihre Entwicklung sinnvoll abgeschätzt werden kann, ist es wichtig, die Beobachtung über einen ausreichend langen Zeitraum durchzuführen. Nur so können längerfristige Trends und periodische Entwicklungen erkannt werden. Damit ist es dann möglich, ein aussagekräftiges Verhaltensmodell für das jeweilige Rechnernetz aufzustellen.
- **Fault Management:** Dieser Aspekt beinhaltet die Erfassung und Analyse von Abweichungen gegenüber dem Normalzustand. Dazu ist es nötig, geeignete Metriken und dazugehörige Normalwerte zu ermitteln. Das *Fault Monitoring*, also die Überwachung des Systems in Bezug auf mögliche Fehler, ist dabei nicht nur auf die unteren Schichten beschränkt, sondern kann sich auch auf die Transport- und die Applikationsschicht erstrecken, da ja festgestellt werden muss, auf welcher Ebene sich möglicherweise ein Fehler befindet. Fault Management umfasst neben dem Fault Monitoring auch die Fehlerdiagnose und mögliche Reparaturmaßnahmen.
- **Accounting Management:** Ziel des Accounting Management ist die Feststellung, welche Benutzer welche Ressourcen im Netz benutzen. Basierend auf diesen Informationen kann die Nutzung von Ressourcen geplant und vorhergesagt werden. Ein anderes Einsatzgebiet besteht in der Berechnung von Gebühren bei der Nutzung bestimmter Ressourcen.
- **Configuration Management:** Kernelement des Configuration Management ist die Beobachtung der Netzwerk- und der Systemkonfiguration. Die dabei gesammelten Informationen liefern in erster Linie ein statisches Modell, das als Grundlage für andere Aspekte dient. Darüber hinaus ist es möglich, die Konfigurationsänderungen und ihre Auswirkungen auf den Systembetrieb aufzuzeichnen und zu analysieren.

- **Security Management:** Ziel des Security Management ist die Kontrolle der Zugriffe auf Netzwerkressourcen und sensible Daten. Wichtige Bestandteile sind dabei die Benutzerauthentifizierung und die Festlegung der Benutzerrechte. Ein weiterer Bestandteil ist die Aufzeichnung und Auswertung außergewöhnlicher Aktivitäten, die nicht den Sicherheitsrichtlinien entsprechen.

Wie man aus der Erläuterung der verschiedenen Aspekte des Netzwerk-Managements erkennen kann, stellt Netzwerk-Monitoring einen bedeutenden Querschnittsaspekt dar, da es in fast allen der fünf genannten Bereiche eine wichtige Rolle spielt.

Im Folgenden geht diese Arbeit auf unterschiedliche Gesichtspunkte beim Netzwerk-Monitoring ein. Dazu gehört die Betrachtung, welche grundsätzlichen Möglichkeiten es gibt, Informationen über den Zustand eines Netzwerks zu sammeln. Weiterhin von Bedeutung ist, welche Art von Rechnernetz überwacht werden soll. Es gibt dabei wesentliche Unterschiede, ob es sich um ein Direktverbindungsnetz oder ein Verbundnetz handelt. Abgeschlossen wird dieses Kapitel durch Beschreibungen über die Verwendung statistischer Verfahren beim Netzwerk-Monitoring und über die Visualisierung der beim Monitoring aufgezeichneten Daten.

2.3.1.1 Aktives Netzwerk-Monitoring

Unter aktivem Netzwerk-Monitoring versteht man die Gewinnung von Informationen durch gezieltes und explizites Anfragen. Ein Beispiel dafür ist die Verwendung des *Ping*-Programms, mit dem die Roundtrip-Zeit für Nachrichten zwischen zwei Maschinen gemessen werden kann. Dazu sendet eine Maschine spezielle *ICMP-Nachrichten* an die Gegenstelle und wartet auf die Antwort.

In einem MOST-System würden die aktive Abfrage von Diagnoseinformationen oder die Prüfung auf Verfügbarkeit von Geräten und Anwendungen mögliche aktive Messungen darstellen.

Der Vorteil bei aktivem Monitoring besteht in der sehr hohen Genauigkeit der Messergebnisse, da gezielt nach den Informationen gefragt wird, die von Interesse sind. Dabei muss beachtet werden, dass die Messungen wirklich repräsentativ sind und beispielsweise sowohl zu Zeiten mit hoher Last als auch zu Zeiten mit niedriger Last auf dem Netzwerk ausgeführt werden.

Den größten Nachteil beim aktiven Netzwerk-Monitoring stellt mit Sicherheit

die Beeinflussung des beobachteten Netzwerks durch die Messungen selbst dar. Dadurch kann einerseits das Messergebnis selbst verfälscht werden und andererseits der normale Betrieb, insbesondere in Zeiten hohen Verkehrsaufkommens, beeinträchtigt werden.

2.3.1.2 Passives Netzwerk-Monitoring

Unter passivem Netzwerk-Monitoring versteht man die Beobachtung des Netzwerkverkehrs ohne selbst aktiv Nachrichten zu Testzwecken zu versenden. Dadurch bleibt das System durch die Beobachtung selbst unbeeinflusst. Darüber hinaus ist es oft einfacher, den Netzwerkverkehr passiv zu beobachten, da keine Kooperation der beobachteten Komponenten nötig ist.

Der Hauptnachteil besteht darin, dass die Beobachtung weniger zielgerichtet ist und der gesamte Netzwerkverkehr aufgezeichnet und analysiert werden muss. Diesem Nachteil kann man nur teilweise entgegenwirken, indem für die Beobachtung uninteressante Nachrichten möglichst frühzeitig herausgefiltert werden. Selbst dann fallen noch sehr große Datenmengen zur Weiterverarbeitung an, insbesondere in Rechnernetzen mit hoher Bandbreite.

In diesem Fall bietet es sich an, bereits auf der Ebene des Netzwerk-Monitorings die Daten zu verdichten.

2.3.1.3 Monitoring in Direktverbindungsnetzen

Bei einem Direktverbindungsnetz handelt es sich um ein Rechnernetz, bei dem die einzelnen Stationen über ein gemeinsames Medium (*Bus*) miteinander verbunden sind. Durch die gegebene Bus-Charakteristik kann eine einzelne Station den gesamten Verkehr innerhalb des Direktverbindungsnetzes beobachten, sofern der Netzwerkadapter in den entsprechenden Betriebsmodus (*promiskutiver Modus*) gesetzt wurde.

Bei dieser Form der Beobachtung wird das Netzwerk nicht belastet und das Messergebnis somit auch nicht durch die Beobachtung selbst verfälscht.

Durch die Verwendung eines gemeinsamen Übertragungsmediums besitzt ein Direktverbindungsnetz im Vergleich zu einem Verbundnetz einen hohen Grad an Homogenität. Daher können die zu beobachtenden Metriken und Merkmale individuell auf das Netzwerk zugeschnitten sein.

Da viele Direktverbindungsnetze über hohe Bandbreiten für die Datenübertragung verfügen, müssen insbesondere bei der Beobachtung im promiskutiven Modus sehr hohe Datenraten bewältigt werden. Aus diesem Grund müssen in

solchen Fällen leistungsfähige Workstations oder Spezialhardware für das Netzwerk-Monitoring eingesetzt werden.

Für das Netzwerk-Monitoring wurde früher hauptsächlich Spezialhardware mit speziell darauf abgestimmter Software benutzt. Seit geraumer Zeit sind allerdings Arbeitsplatzrechner leistungsfähig genug für die Aufzeichnung des Netzwerkverkehrs auf Ebene der Sicherungsschicht und deren Auswertung in Echtzeit [36]. Die Benutzung von Arbeitsplatzrechnern ist auch deshalb besonders attraktiv, da sie flexible Auswertungsmöglichkeiten zur Verfügung stellen. So existieren inzwischen zahlreiche Werkzeuge für das Netzwerk-Monitoring, die unter gängigen Betriebssystemen für Arbeitsplatzrechner laufen [50].

Moderne Betriebssysteme wie Unix oder die neueren Varianten von Microsoft Windows (Win32) beinhalten von sich aus schon zahlreiche Funktionen zur Rechnervernetzung, sind aber nicht spezialisiert auf die effiziente Aufzeichnung und Auswertung des Datenverkehrs im Rahmen von Netzwerk-Monitoring. Von den Spezifika der Netzwerkhardware wird in diesen Betriebssystemen abstrahiert, stattdessen stellen Netzwerkgerätetreiber die empfangenen und noch unveränderten Daten (raw data) an einer einheitlichen Schnittstelle (*Paketschnittstelle*) zur Verfügung. Normalerweise werden die Daten an dieser Stelle an den Netzwerk-Protokollstack weitergegeben. Allerdings kann sich ein Zusatztreiber (*Paketfilter*) ebenfalls an dieser Stelle „einhängen“ um die Daten zur weiteren Verarbeitung entgegenzunehmen und sie schließlich Monitoring-Anwendungen zur Verfügung zu stellen (vgl. Abbildung 6). Solche Paketfilter stehen beispielsweise mit dem *BSD Packet Filter* (BPF) [34] für Unix sowie mit *Winpcap* [13] für Windows zur Verfügung.

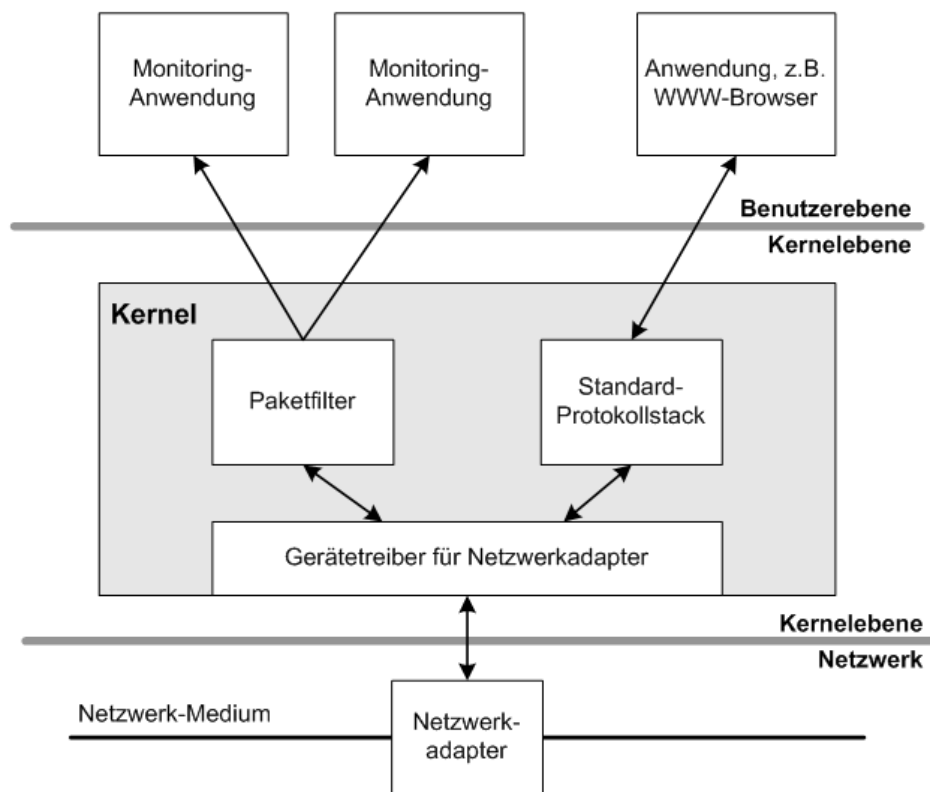


Abbildung 6: Beziehungen zwischen dem Paketfilter und anderen Systemkomponenten; die Pfeile zeigen den Datenfluss an

Zumindest bei den oben genannten Betriebssystemen muss der Paketfilter im Kontext des Betriebssystemkernels ausgeführt werden (privilegierter Modus), um auf die Paketschnittstelle zugreifen zu können. Monitoring-Anwendungen, welche die aufgezeichneten Daten auswerten, werden dagegen als Benutzerprozesse ausgeführt. Dies hängt vor allem damit zusammen, dass die Programmierung von Code, der in Prozessen auf der Benutzerebene ausgeführt wird, wesentlich einfacher ist. Damit der Paketfilter die aufgezeichneten Daten an eine Monitoring-Anwendung übergeben kann, müssen der Ausführungskontext vom Kernelprozess zu dem entsprechenden Prozess auf Benutzerebene umgeschaltet werden und die Daten in diesen Prozess kopiert werden. Sowohl beim Kontextwechsel als auch dem Kopiervorgang handelt es sich um teure Operationen. Daher muss die Anzahl der Wechsel zwischen dem Kernelprozess und Prozessen auf der Benutzerebene und der Kopieroperationen minimiert werden [37].

Die oben genannten Paketfilter führen daher eine erste Filterung durch, bei der – für die Monitoring-Anwendungen - uninteressante Pakete verworfen werden. Die Filter können dabei vom Benutzer spezifiziert werden und werden dann aus Effizienzgründen in einer virtuellen Maschine innerhalb des Kernel-codes interpretiert. Weitere Optimierungen bestehen darin, mehrere Pakete auf einmal an den Benutzerprozess der Monitoring-Anwendung zu übergeben (Batchübergabe) und nur diejenigen Teile der Pakete zu kopieren, die wirklich benötigt werden.

Bei dem MOST-Bus handelt es sich ebenfalls um ein Direktverbindungsnetz, das mit 25 Megabit pro Sekunde über eine vergleichsweise hohe Bandbreite verfügt. Die für Arbeitsplatzrechner verfügbaren Netzwerkkadaper (MOST Optolyzer) stellen aber lediglich diejenigen Nachrichten, die über den MOST-Kontrollkanal übertragen werden, für die weitere Auswertung zur Verfügung. Aufgrund der geringen Bandbreite des Kontrollkanals bietet es sich daher an, die Filterung und Auswertung des beobachteten Datenverkehrs vollständig innerhalb eines Prozesses auf der Benutzerebene durchzuführen.

2.3.1.4 Monitoring in Verbundnetzen

Verbundnetze zeichnen sich üblicherweise durch ihre Größe und ihre Heterogenität aus, da sie häufig aus mehreren Teilnetzen bestehen, in denen unterschiedliche Technologien zur Vernetzung zum Einsatz kommen. Ihre Größe bedingt dabei einen hohen Kommunikationsaufwand für den Austausch der Messdaten. Ihre Heterogenität führt zwangsläufig zu einer Beschränkung auf wenige allgemeingültige Metriken, die beobachtet werden können. Beispiele für solche Metriken stellen der Anteil verlorener Pakete oder die Antwortzeiten bestimmter Hostrechner dar.

Durch den Zusammenschluss von Teilnetzen bedingt besitzen die Verbindungen eine Punkt-zu-Punkt-Charakteristik im Gegensatz zur Buscharakteristik von Direktverbindungsnetzen. Daher muss die Beobachtung in einem Verbundnetz verteilt erfolgen.

Bei der passiven Beobachtung fallen in jedem Teilnetz große Datenmengen an. Diese Daten müssen ausgetauscht und integriert werden. Durch den Austausch der Daten besteht die Gefahr, dass der Nutzdatenverkehr in einem Verbundnetz beeinträchtigt und das Messergebnis selbst verfälscht wird. Zusätzlich besteht das Problem der Zeitsynchronisation, so dass in verschiede-

nen Teilnetzen aufgezeichnete Daten nicht ohne weiteres mittels Zeitstempeln in Bezug gesetzt werden können.

Aus den genannten Gründen werden daher häufig gezielte aktive Messungen durchgeführt. Damit fallen bei der Beobachtung geringere Datenmengen an. Allerdings hat die aktive Beobachtung den Nachteil, dass durch die Beobachtung selbst die Messergebnisse verfälscht und der normale Betrieb beeinträchtigt werden können. Es muss daher sehr sorgfältig geplant werden, welche Messdaten an welchen Stellen und zu welchen Zeitpunkten erhoben werden.

2.3.2 Die Rolle der Statistik im Netzwerk-Monitoring

Statistische Verfahren spielen beim Netzwerk-Monitoring aus mehreren Gründen eine wichtige Rolle. Ein Grund besteht darin, dass gewisse Einschränkungen beim Netzwerk-Monitoring bestehen [33]. Sofern es sich bei dem zu überwachenden Netzwerk nicht um ein einzelnes Direktverbindungsnetz handelt, muss die Überwachung mit mehreren Sensoren erfolgen. Aus Kostengesichtspunkten und um die Beeinflussung des Betriebs durch den Datenaustausch zwischen den einzelnen Sensoren gering zu halten, versucht man, die Anzahl der Sensoren zu minimieren. Ist die maximale Anzahl von Sensoren festgelegt, so ist es wichtig, die Sensoren an den richtigen Knoten einzusetzen. Hier können statistische Modelle des Netzwerkverkehrs helfen, die richtigen Knoten zu bestimmen.

Eine ähnliche Einschränkung besteht bezüglich der Beeinflussung des Netzwerkbetriebs durch aktive Messungen. Somit muss es ein Ziel sein, die Anzahl der Messungen zu minimieren und dennoch ein genaues und aussagekräftiges Messergebnis zu erhalten. Auch hier können statistische Modelle des Netzwerkverkehrs bei der Festlegung helfen, wann welche aktiven Messungen sinnvoll sind.

Ein weiterer Grund, weshalb der Einsatz statistischer Methoden beim Netzwerk-Monitoring sinnvoll ist, besteht darin, dass bei der Beobachtung des Netzwerkverkehrs sehr große Datenmengen anfallen können, die sich allein auf Grund ihrer Größe nicht sinnvoll in Logdateien aufzeichnen und anschließend durch einen Netzwerkadministrator auswerten lassen. Auch die aussagekräftige Visualisierung solcher Datenmengen stellt ein Problem dar, so dass es für einen Netzwerkadministrator schwierig ist, den Überblick zu behalten und wichtige Entwicklungen von unwichtigen zu unterscheiden. Mit Hilfe von

geeigneten statistischen Verfahren können die gesammelten Daten verdichtet werden, so dass wesentliche Entwicklungen stärker zum Vorschein kommen.

Ein einfacher, aber Erfolg versprechender Ansatz besteht dabei in der Auffassung der gesammelten Daten als Zeitreihen. Darauf aufbauend können einfache statistische Metriken erhoben werden, die zur Definition des Normalzustand dienen. Mögliche Metriken sind das Zählen bestimmter Ereignisse innerhalb eines Zeitintervalls, sowie die Bildung des Medians, des Durchschnitts oder des gleitenden Durchschnitts über einer Zeitreihe. Wurde ein Normalwert für eine bestimmte Metrik festgelegt, können mit der Berechnung der Standardabweichung oder der Varianz schon Abweichungen vom Normalzustand erkannt werden. Wenn es sinnvoll erscheint, können zwei Zeitreihen auf mögliche Abhängigkeiten überprüft werden, indem beispielsweise die Korrelation zwischen den beiden Zeitreihen berechnet wird.

In manchen Fällen genügen die Verwendung einfacher statistischer Grundwerkzeuge und fester Schwellwerte zur Erkennung anomalen Verhaltens nicht, insbesondere wenn die automatische Erkennung von Anomalien das Ziel ist. Hier kommen dann fortgeschrittenere Verfahren zum Einsatz. Dabei ist zu beachten, dass die berechneten Metriken von den Netzwerkverantwortlichen verstanden werden und in Echtzeit berechenbar sein müssen. Brutlag beschreibt in [6] ein statistisches Modell, das zur automatischen Erkennung von Anomalien dient, die auf mögliche Fehler oder Angriffe hindeuten. Die Schwellwerte zur Erkennung von Anomalien werden dabei dynamisch berechnet und an die Entwicklung der Zeitreihen angepasst. Das verwendete Verfahren führt dabei drei Hauptschritte durch:

1. Die Vorhersage des jeweils nächsten Wertes in der Zeitreihe. Dabei kommt das Verfahren des *exponentiellen Glättens* [5] zum Zuge. Exponentielles Glätten verwendet als Vorhersagewert \hat{y}_{t+1} einen gewichteten Durchschnitt über die bisherigen Werte der betrachteten Zeitreihe:

$$\hat{y}_{t+1} = \alpha \cdot y_t + (1 - \alpha) \cdot \hat{y}_t$$

α ($0 < \alpha < 1$) ist dabei der Modellparameter, der angibt, welches Gewicht der aktuelle Wert und welches Gewicht Werte aus der Vergangenheit besitzen. Durch die inkrementelle Natur des Verfahrens haben ältere Werte ein exponentiell geringeres Gewicht bei der Berechnung des Vorhersagewertes.

2. Die Messung der Abweichung zwischen der Vorhersage und dem tatsächlich beobachteten Wert. Die Abweichung berechnet sich als Differenz

$$\Delta_{t+1} = y_{t+1} - \hat{y}_{t+1}.$$

3. Die Entscheidung, ob und wann ein beobachteter Wert zu weit von dem vorhergesagten Normalverhalten abweicht. Hier wird die Verwendung von Konfidenzbereichen statt eines statischen Schwellwertes vorgeschlagen. Die Grenzen eines Konfidenzbereiches werden ebenfalls durch exponentielles Glätten festgelegt.

Eine Verfeinerung des Verfahrens besteht in der Verwendung des *Vorhersagealgorithmus nach Holt-Winters* ([5], [6]). Dieser Algorithmus legt als Prämisse zugrunde, dass eine Zeitreihe aus drei Komponenten mit jeweils unabhängigen Anpassungsparametern besteht:

- Einem langfristigen Grundtrend (entspricht beispielsweise einer graduellen Zunahme der Netzwerkauslastung)
- einem periodischen Trend (z.B. tagsüber eine höhere Auslastung des Netzwerks als in den Nachtstunden)
- sowie kurzfristigen Schwankungen innerhalb des periodischen Trends

Wie bei anderen Modellen auch ist natürlich zu prüfen, inwiefern das statistische Modell mit der Wirklichkeit übereinstimmt.

Mit Hilfe von Konfidenzbereichen sollen bei dieser Variante außergewöhnliche kurzfristige Schwankungen innerhalb eines periodischen Zyklus erkannt werden. Die Anpassung der Konfidenzbereiche muss daher denselben Anpassungsparameter verwenden wie die Vorhersagekomponente für kurzfristige Schwankungen.

Für die automatische Erkennung von Anomalien in der Zeitreihenentwicklung gibt es schließlich mehrere Möglichkeiten. Einzelne Werte außerhalb der Konfidenzbereiche dafür heranzuziehen, ist zwar naheliegend, führt aber zu häufigen Fehlalarmen. Um dies zu vermeiden, ist es sinnvoller, eine Warnung anzuzeigen, wenn innerhalb eines definierten Zeitintervalls die Anzahl der Werte außerhalb der Konfidenzbereiche eine bestimmte Anzahl überschreitet.

Insgesamt muss man bei der Anwendung statistischer Verfahren eine gewisse Vorsicht walten lassen. Statistische Verfahren legen immer ein bestimmtes Modell zu Grunde, das die Wirklichkeit repräsentieren soll. Damit man nicht von falschen Voraussetzungen beim Netzwerk-Monitoring ausgeht, muss getestet werden, ob das zugrunde gelegte Modell auch Gültigkeit für die Wirklichkeit besitzt.

Ein Fahrzeugtelematiksystem hat eine ganz andere Charakteristik als ein Rechnernetz in einem Unternehmen. Es ist relativ klein, abgeschlossen und die

Betriebszeiten sind vergleichsweise kurz. Daher liefert ein Betriebszyklus eines Fahrzeugtelematiksystems eine sehr geringe Datenmenge, die ausgewertet werden kann. Damit ist es auch schwierig, statistische Modelle des Netzwerkverkehrs sinnvoll zu verwenden, etwa zur Vorhersage des Systemverhaltens. Andererseits bieten statistische Verfahren eine Möglichkeit, die anfallenden Datenmengen zu reduzieren, was insbesondere bei den eingeschränkten Speicherkapazitäten, über die Telematikgeräte verfügen, sinnvoll ist.

2.3.3 Automatische Erkennung von Anomalien

Die automatische Erkennung von Anomalien wird im Bereich des Netzwerk-Monitoring hauptsächlich zur Erkennung von Fehlern oder Angriffen eingesetzt. Ausgehend von einer Definition des Normalverhaltens werden Abweichungen, die eine bestimmte Grenze überschreiten, als Fehler oder Angriff interpretiert.

An die automatische Erkennung von Anomalien werden dabei mehrere Anforderungen gestellt. Eine Anforderung besteht in der zuverlässigen Erkennung von wirklichen Anomalien. Dabei ist auch zu unterscheiden, ob lediglich bekannte Muster erkannt werden müssen oder ob auch Fehler und Angriffe mit unbekanntem Mustern erkannt werden sollen. Eine gegensätzliche Anforderung besteht in der Vermeidung von Fehlalarmen. Die Leistungsfähigkeit von Verfahren zur automatischen Erkennung von Anomalien lässt sich somit anhand der Erkennungsrate und der Häufigkeit von Fehlalarmen charakterisieren.

Für die automatische Erkennung von Anomalien gibt es verschiedene Verfahren. Ein Ansatz besteht in der Verwendung von Zustandsautomaten, welche die Zustände von Netzwerkprotokollen repräsentieren. Hierbei wird das Auftreten außergewöhnlicher Zustandsübergänge als Anomalie gewertet [49]. Ein anderer Ansatz besteht in der Bestimmung von Wahrscheinlichkeiten, mit der die verschiedenen Arten von Datenpaketen im Netzwerkverkehr auftauchen ([53], [29]). Das Auftreten von „unwahrscheinlichen“ Paketen wird hierbei als Anomalie behandelt. Andere Ansätze benutzen Verfahren zur Analyse von Clustern [57] oder aus dem Bereich des Data Mining [31].

2.3.4 Visualisierung der gewonnenen Daten

Selbst aggregierte Daten sind für menschliche Beobachter schwer zu erfassen, wenn sie sich beispielsweise in einer Logdatei befinden. Aus diesem Grund ist es naheliegend, die aus der Beobachtung gewonnenen Daten in grafischer Form zu visualisieren.

Dafür gibt es zahlreiche Varianten, die sich je nach Charakteristik der Daten anbieten. So eignen sich beispielsweise Sequenzdiagramme zur Visualisierung einzelner Abläufe, während sich Punktwolken (Scatterplots) für die Darstellung großer Mengen zweidimensionaler Daten eignen. Wenn die Anzahl der Dimensionen zwei übersteigt, können mehrere Punktwolken, die jeweils verschiedene Dimensionen derselben Daten darstellen zu einer Matrix zusammengefasst werden (so genannte Pair Plots). Wenn die Anzahl der Dimensionen sehr groß wird, verlieren allerdings auch Pair Plots schnell an Übersichtlichkeit.

Alternativen für die Darstellung mehrdimensionaler Daten bestehen in der Verwendung von Liniendiagrammen mit Parallelkoordinaten oder von Farbhistogrammen. Damit können auch Daten, die sehr viele Dimensionen umfassen, noch übersichtlich angezeigt werden.

Für die Visualisierung der Netzwerktopologie bieten sich Graphen an. Neben der reinen Topologieinformation können solche Graphen noch weitere Informationen enthalten: Beispielsweise können die Kanten unterschiedlich breit oder in unterschiedlichen Farben eingezeichnet werden, um die Verkehrsdichten zwischen den verschiedenen Stationen anzuzeigen.

Für menschliche Benutzer ist es von großem Nutzen, wenn sich mehrere Diagramme miteinander in Bezug setzen lassen [62]. Damit lässt sich dann manuell relativ einfach erkennen, ob eine bestimmte Entwicklung mit einer anderen Entwicklung oder einem bestimmten Ereignis zusammenhängt. Dies bedingt natürlich, dass die zu verknüpfenden Daten eine gemeinsame Dimension wie beispielsweise eine Zeitangabe besitzen und sie bezüglich dieser Dimension normiert sind.

Eine solche Möglichkeit ist auch hilfreich bei der Auswertung des beobachteten Datenverkehrs auf dem MOST-Kontrollkanal. So lassen sich beispielsweise die Auswirkungen von einzelnen Ereignissen wie einer Konfigurationsänderung oder dem Verlust des Synchronisationssignals auf die Buslast oder das Antwortzeitverhalten einzelner Geräte einfach erkennen.

3 Anforderungsanalyse

Telematiksysteme in modernen Kraftfahrzeugen, insbesondere denen der Oberklasse, entwickeln sich immer mehr zu komplexen verteilten Systemen. Ziel dieser Arbeit ist die Entwicklung einer Komponente, die das Verhalten eines Telematiksystems beobachtet, das MOST als Vernetzungstechnologie benutzt. Somit müssen bei der Entwicklung der Beobachterkomponente die speziellen Eigenschaften des MOST-Bus ebenso berücksichtigt werden wie der mögliche Einsatz in einem Steuergerät im Fahrzeug. Diese und weitere Anforderungen sollen in diesem Kapitel untersucht werden.

3.1 Einsatzszenarien

Die zu entwickelnde Komponente soll in mehreren möglichen Szenarien zum Einsatz kommen. Ein Szenario sieht den Einsatz als Beobachterkomponente im Rahmen der SeHeT-Architektur vor (vgl. Abschnitt 2.2.4). Ein weiteres Szenario geht davon aus, die Komponente zur Evaluierung konkreter Monitoring- und Diagnosemechanismen zu benutzen. Diejenigen Mechanismen, die sich bei der Evaluierung als geeignet erwiesen haben, können dann in den entsprechenden Telematik-Steuergeräten in Hardware realisiert werden.

3.2 Monitoring des Nachrichtenverkehrs auf dem MOST-Bus

Bei einem MOST-System handelt es sich um ein geschlossenes System mit eingeschränkter Dynamik, da die Steuergeräte des Telematiksystems fest miteinander vernetzt sind. Für die Auswertung kommt hauptsächlich der Kontrollkanal in Betracht, da die verfügbare Hardware zur Netzwerkanalyse nicht in der Lage ist, einer Monitoring-Anwendung den Inhalt der synchronen und asynchronen Kanäle zur Verfügung zu stellen. Die zu realisierenden Sensoren müssen dabei auf die Charakteristika des MOST-Bus abgestimmt sein und sowohl die Systemereignisse als auch den Nachrichtenverkehr auswerten können.

3.3 Konfigurierbarkeit

Um für die Evaluierung unterschiedlicher Monitoring- und Diagnosemechanismen benutzt werden zu können, soll die Beobachterkomponente so weit konfigurierbar sein, dass die entsprechenden Mechanismen aus vorhandenen Bausteinen in kurzer Zeit zusammengestellt werden können. Es soll dabei auch möglich sein, die Beobachterkomponente um weitere Bausteine, die Filter- und Auswertungsmöglichkeiten realisieren zu erweitern. Die Konfiguration soll dabei von außen festgelegt werden können.

Sehr viele Informationen über das Verhalten von konkreten MOST-Systemen in Fahrzeugen steht bislang in Form von Tracedateien zur Verfügung, in denen der Nachrichtenverkehr und wichtige Systemereignisse über einen begrenzten Zeitraum aufgezeichnet wurden. Daher soll die zu realisierende Komponente in der Lage sein, Tracedateien auszuwerten, als auch ein MOST-System während des laufenden Betriebs zu beobachten.

3.4 Anwendungsschnittstelle

Für den Einsatz im Rahmen der SeHeT-Architektur (vgl. Abschnitt 2.2.4) müssen die Informationen über den Systemzustand in einer Form verfügbar sein, die maschinell auswertbar ist. Da die Beobachterkomponente allerdings auch in anderen Szenarien eingesetzt werden soll, muss die entsprechende Schnittstelle so allgemein gehalten werden, dass auch andere Anwendungen, beispielsweise zur Visualisierung der aufgezeichneten Daten, darauf aufsetzen können.

3.5 Performanz

Eine generelle Anforderung an ein Monitoring-Werkzeug ist die Verarbeitung in Echtzeit, da sonst wichtige Informationen über den Systemzustand verloren gehen können. Auch die Anwendungen, die auf die Beobachterkomponente aufbauen, sollen die von der Beobachterkomponente aufgezeichneten und ausgewerteten Daten in Echtzeit erhalten.

3.6 Einsatz in eingebetteten Systemen

Bei den Telematikgeräten im Fahrzeug handelt es sich um eingebettete Systeme, die jeweils eine spezielle Funktionalität realisieren. Diese Geräte sind in Gegensatz zu Arbeitsplatzrechnern in Bezug auf die Kosten und den Stromverbrauch eingeschränkt. Diese Einschränkungen schlagen sich in eingeschränkten Speicher- und Rechenkapazitäten nieder und sind bei der Realisierung der Beobachterkomponente zu berücksichtigen.

Durch die verschiedenen Szenarien bedingt soll die Beobachterkomponente sowohl auf leistungsfähigen Arbeitsplatzrechnern als auch in eingebetteten Systemen lauffähig sein. Zu diesem Zweck bietet sich die Java 2 Platform Micro Edition (J2ME, [55]) als Ablaufumgebung in beiden Fällen an.

Um die Konfiguration der Software auf den verschiedenen Steuergeräten in Zukunft durch einen einheitlichen Mechanismus zu verwalten, soll das Kom-

ponentenmodell der Open Service Gateway Initiative (OSGi, vgl. [47] und [11]) eingesetzt werden. Das von der OSGi definierte Komponentenmodell ist im Vergleich zu den auf Arbeitsplatzrechnern und Servern verwendeten Komponentenmodellen relativ leichtgewichtig und auf eingebettete Geräte zugeschnitten. Jede Komponente verfügt über eine definierte öffentliche Dienstschnittstelle, Informationen zur Versionierung und eine explizite Beschreibung der Abhängigkeiten. Dadurch und durch die Möglichkeit der Remote-Administration der Komponenten wird das Konfigurationsmanagement der Software in den verschiedenen Steuergeräten wesentlich erleichtert. Um den Einsatz in Steuergeräten zu erleichtern, soll die Beobachterkomponente ebenfalls als OSGi-Bundle realisiert werden.

3.7 Geringe Beeinflussung des Systems durch die Beobachtung

Da das mit dem MOST-Kontrollkanal zur Verfügung stehende Medium eine sehr kleine Bandbreite besitzt, muss bei der Beobachtung besonders auf eine geringe Beeinflussung des Systems geachtet werden. Daher soll auf aktive Messungen und auf die Benutzung von mehreren, verteilt installierten, Sensoren verzichtet werden.

4 Existierende Lösungen für die Netzwerkbeobachtung

Im bisherigen Verlauf dieser Arbeit wurden die Grundlagen der Netzwerkbeobachtung und die Anforderungen an eine Beobachterkomponente in einem MOST-basierten Fahrzeugtelematiksystem erörtert. Dieses Kapitel geht auf existierende Standards und Werkzeuge für die Netzwerkbeobachtung ein und untersucht dabei, inwiefern sie den genannten Anforderungen entsprechen und inwiefern sie davon zu Gunsten anderer Anforderungen abweichen. Abgeschlossen wird dieses Kapitel mit der Vorstellung bestehender Lösungen für die Netzwerkbeobachtung in MOST-Systemen.

4.1 Simple Network Management Protocol - SNMP

Bei dem Simple Network Management Protocol (SNMP) handelt es sich um den De facto-Standard für die Beobachtung und Verwaltung von IP-basierten Rechnernetzen ([22], [24],[25] und [52]). Somit ist zumindest eine teilweise Interoperabilität zwischen verschiedenen Beobachterkomponenten gegeben.

SNMP besteht aus zwei Hauptbestandteilen:

- Der Beschreibung eines allgemeinen Datenmodells für Informationen über Geräte und andere Entitäten in einem Rechnernetz.
- Einem Kommunikationsprotokoll, das den Austausch der genannten Informationen zwischen den verschiedenen, am Monitoring beteiligten Entitäten festlegt.

Auch wenn der Name nahe legt, dass SNMP den gesamten Bereich der Netzwerkverwaltung standardisiert, beschränkt sich SNMP dennoch weitestgehend auf die Netzwerkbeobachtung.

Bei SNMP werden drei Arten von Entitäten unterschieden:

- Der Begriff **verwaltete Objekte** (Managed Objects) bezeichnet die verwalteten Geräte oder spezifische Eigenschaften dieser Geräte.
- Bei **Agenten** handelt es sich um Softwareeinheiten, die Informationen über Netzwerkelemente sammeln und so genannten Monitoringstationen bereitstellen.
- **Monitoringstationen** übernehmen die Verwaltung überwachter Netzwerkelemente. Um die nötigen Informationen über diese Netzwerkelemente zu erhalten, fragen sie die zugeordneten Agenten periodisch nach neuen Daten ab (Polling). Um Monitoringstationen über besondere Vorkommnisse, insbesondere Fehler, sofort zu benachrichtigen, können Agenten ihre Monitoringstation auch direkt über Ereignisse benachrichtigen.

SNMP definiert ein generisches hierarchisches Datenmodell, die so genannte *Management Information Base (MIB)*. Eine MIB beinhaltet eine Menge von Informationen über verwaltete Objekte, jedes verwaltete Objekt wird dabei durch einen numerischen Bezeichner aus einem hierarchischen Namensraum (Object Identifier – OID) beschrieben. Ein OID beinhaltet Informationen über den Typ des Objekts (z.B. ein Router, der Pakete des AppleTalk-Protokolls verarbeiten kann) sowie einen Unterbezeichner zur Unterscheidung verschiedener Instanzen eines Typs. Jedes Managed Object stellt Informationen über sich durch eine Menge von skalaren Eigenschaftsvariablen bereit.

Sowohl das Datenmodell von SNMP als auch die Nachrichtenformate sind bei SNMP durch eine Teilmenge der Abstract Syntax Notation One (ASN.1) beschrieben.

Bis einschließlich der Version 2 (SNMPv2) waren in SNMP keinerlei Sicherheitsfunktionen definiert, jeder konnte ohne Authentifizierung die Agenten nach Informationen abfragen oder die unverschlüsselten SNMP-Nachrichten mithören. Erst SNMPv3 führt ein benutzerbasiertes Sicherheitsmodell ein [25]. Das Sicherheitsmodell beinhaltet die Authentifizierung von Absendern, die Sicherung der Nachrichtenintegrität und die Geheimhaltung von Nachrichteninhalten durch Anwendung von Verschlüsselung.

Während das von SNMP definierte Datenmodell (MIB) noch sehr generischer Natur ist, gibt es mit dem Remote Monitoring-Standard (RMON) eine konkrete Ausprägung der SNMP MIB für Ethernet- und Token Ring-Netzwerke [23]. RMON definiert dabei konkrete Typen von verwalteten Objekten mit bestimmten Eigenschaftsvariablen. Damit sorgt RMON für eine weiterreichende Interoperabilität von verschiedenen Beobachterkomponenten in einem Netzwerk.

4.1.1 Anwendbarkeit in Fahrzeugtelematiksystemen

Die von SNMP und RMON verwendeten Mechanismen eignen sich für lokale Netzwerke oder Verbundnetze, wofür sie ja entworfen wurden. Die Anforderungen an die Beobachtung eines MOST-basierten Fahrzeugtelematiksystems sind jedoch unterschiedlich. So wird der Normalbetrieb des Fahrzeugtelematiksystems durch den Austausch der Beobachtungsdaten beeinträchtigt. Die Verwendung von Polling als primäres Kommunikationsmuster verschärft die Beeinträchtigung. Wenig überraschend ist, dass SNMP und RMON nicht auf charakteristische Eigenschaften von MOST-Systemen eingehen.

4.2 Network Weather Service (NWS)

Der Network Weather Service [61] ist ein Dienst für die Beobachtung und kurzfristige Vorhersage der Performanz in einem verteilten System. Vorrangiger Einsatzbereich sind Systeme für verteiltes Rechnen, in denen der Network Weather Service Abschätzungen für die Systemperformanz an einen Scheduler auf Applikationsebene liefert. Dabei wird die kurzfristige Vorhersage im Vergleich zur aktuellen Performanz als aussagekräftiger bewertet, da der Scheduler oder die Benutzer des Systems entsprechend darauf reagieren können, beispielsweise durch automatische Lastverteilung.

Der NWS ist ein schönes Beispiel für einen Dienst, bei dessen Entwurf auf Erweiterbarkeit, Skalierbarkeit, Fehlertoleranz sowie effiziente Algorithmen (zur Messung und Vorhersage der Performanz) geachtet wurde.

Der NWS besteht aus vier verschiedenen Komponenten, die jeweils einen Aspekt des NWS behandeln:

- **Sensor:** Ein Sensor sammelt Messwerte bezüglich der Performanz einer spezifischen Ressource. Diese Daten werden zusammen mit einem aussagekräftigen Bezeichner (vgl. Namensdienst) an den Persistenzdienst gesendet.
Die Beobachtung der Performanz durch Sensoren erfolgt hauptsächlich passiv. Allerdings wird die passive Beobachtung durch aktive Messungen unterstützt, um die Genauigkeit der Messungen zu erhöhen. Um den Konflikt zwischen hoher Messgenauigkeit und einer geringen Beeinflussung zu entschärfen, werden Techniken der adaptiven Programmierung eingesetzt:
Wenn die vorhergesagte Performanz einer Ressource mit der später gemessenen Performanz übereinstimmt, werden in Zukunft weniger aktive Messungen zur Korrektur durchgeführt und umgekehrt.
- **Namensdienst:** Im Network Weather Service steht ein einfacher Namensdienst zur Verfügung, der die Adressierungsinformationen des eingesetzten Netzwerkprotokolls – in der bisherigen Realisierung sind das IP-Adressen und Portnummern – an aussagekräftige Bezeichner von Daten und Prozessen bindet. Der Namensdienst selbst ist über eine wohlbekannte Adresse erreichbar.
- **Persistenzdienst:** Diese Komponente ist dafür zuständig, Messergebnisse zusammen mit den zugehörigen Zeitstempeln unter aussagekräftigen Bezeichnern persistent zu speichern und für Abfragen zur Verfügung zu halten.
- **Vorhersage:** Die Vorhersagekomponente fragt den Persistenzdienst periodisch nach neuen Messwerten, um daraus sowie aus historischen Messergebnissen Vorhersagen der Performanz zu berechnen. Dabei werden, wie bei der Beobachtung, Techniken der adaptiven Programmierung benutzt, um die Genauigkeit der Vorhersagen zu erhöhen:
Es werden verschiedene einfache statistische Modelle (Mittelwert, Median, Autoregression) benutzt um die Vorhersage einer Zeitreihe zu berechnen. Die Vorhersagen werden vom Vorhersageprozess (nicht-persistent) gespeichert, um sie zu einem späteren Zeitpunkt mit dem tatsächlichen Performanzverlauf zu vergleichen. Aus dem Vergleich resultierend erfolgt dann die Anpassung der Gewichtungsfaktoren für die einzelnen statistischen Modelle.

Die Messungen der Netzwerkperformanz erfolgen verteilt durch eine Vielzahl von Sensoren. Dabei wird eine hohe Skalierbarkeit bezüglich der Systemgröße

erreicht, indem zwischen dem gesamten Rechnernetz und einzelnen Stationen eine weitere Hierarchiestufe eingeführt wird, so genannte *Cliquen*. Eine Clique ist eine Gruppe von Sensoren, die benachbart sind. Jeder Sensor in einer Clique führt dabei Messungen mit jedem anderen Sensor in derselben Clique durch. Jede Clique verfügt dabei über einen so genannten *Grenzsensoren*, so dass Messungen zwischen je zwei Cliquen mittels der Grenzsensoren durchgeführt werden können. Um die Messungen innerhalb einer Clique zu koordinieren und Konflikte zu vermeiden, wird ein *Token-Protokoll* zur Zugriffssteuerung eingesetzt. Die Cliquen-Mitglieder bilden einen logischen Ring, entlang dem das Token weitergereicht wird. Da das Messsystem lange in Betrieb sein muss, trifft das NWS Vorkehrungen, um die Stabilität des Messprotokolls zu gewährleisten. So gibt es einen *Token-Recovery-Mechanismus*, der im Falle von Sensor-Ausfällen oder Netzwerkpartitionierungen greift. Die Wiederherstellung des Tokens erfolgt dabei nach einem bestimmten Timeout. Der Timeout-Wert wird ebenfalls durch adaptive Techniken (basierend auf der vorhergesagten Performanz) angepasst.

Nach den Angaben in [59] und [60] wird durch die adaptiven Vorhersagetechniken eine, auch im Vergleich zu wesentlich komplexeren statistischen Verfahren, sehr hohe Genauigkeit erzielt. Die Effizienz der verwendeten Vorhersage- und Beobachtungsmechanismen ist dabei sehr hoch und in dem getesteten Fall wurde das Gesamtsystem nur geringfügig beeinflusst. Die hohe Effizienz erlaubt dabei auch die Heranziehung großer Datenmengen zur Berechnung der Vorhersage.

4.2.1 Anwendbarkeit der Mechanismen für die Beobachtung eines Fahrzeugtelematiksystems

Die im NWS verwendeten Mechanismen behandeln viele Problemstellungen, die in einem Grid, aber nicht in einem Fahrzeugtelematiksystem bestehen. Bei einem Grid handelt es sich zumindest in vielen Fällen um ein großes, dynamisches System, dessen Ressourcen weit verteilt sind. Typische Problemstellungen bestehen dabei in der Skalierbarkeit solcher Systeme, der eindeutigen Benennung von Ressourcen oder dem Umgang mit Netzwerkpartitionierungen.

Bei der Beobachtung eines solchen Systems fallen große Datenmengen an. Neben der Größe der Systeme ist dies auch dadurch bedingt, dass solche Systeme über sehr lange Zeiträume in Betrieb sind. Dagegen ist es sehr aufwändig,

ein Fahrzeug über einen sehr langen Zeitraum zu beobachten. Während des Normalbetriebs finden nur relativ kurze Betriebszeiten statt, die Dauer einzelner Fahrten ist üblicherweise in Stunden oder gar nur in Minuten bemessen. Durch die wesentlich kleineren Datenmengen und Zeiträume, die bei einem Fahrzeugtelematiksystem ausgewertet werden können, ist es schwierig, statistische Modelle für die Vorhersage der Performanz oder des Kommunikationsverhaltens allgemein sinnvoll anzuwenden.

Die Verteilung der Sensoren beim NWS impliziert eine Übertragung der Beobachtungsdaten über das Netzwerk, was im Fall des MOST Kontrollkanals zu einer Beeinflussung des Systems führen würde. Die Durchführung aktiver Messungen hätte denselben Effekt.

Ein weiteres Problem besteht in der persistenten Speicherung der Messdaten, da dafür zumindest ein Steuergerät des Telematiksystems über stabilen, wiederbeschreibbaren Speicher verfügen muss.

4.3 Monitoring des MOST-Bus

Für den MOST-Bus existieren bereits Lösungen für die Analyse des Netzwerkverkehrs. Die bestehenden Lösungen werden vorgestellt und daraufhin untersucht, inwieweit sie den in Abschnitt 3 formulierten Anforderungen gerecht werden.

4.3.1 Möglichkeiten des Netzwerk-Monitoring

Das grundlegende Werkzeug für die Analyse ist die so genannte *Optolyzer Interface Box*, im Folgenden auch mit der Kurzform *Optolyzer* bezeichnet (siehe Abbildung 7). Dabei handelt es sich um einen MOST-Netzwerkadapter, der mit einem Windows-PC über die serielle Schnittstelle verbunden wird. Der Optolyzer besitzt den in Abbildung 8 skizzierten Aufbau. Der Network Interface Controller bezeichnet den MOST Transceiverchip. Über die S/PDIF-Schnittstelle oder den Line-Eingang können Audiodaten auf einen synchronen Kanal des MOST-Bus eingespeist werden. Über den Line-Ausgang kann ein beliebiger synchroner Kanal abgehört werden. Über die serielle Schnittstelle (RS232 Interface) werden empfangene Kontrollnachrichten und MOST-Systemereignisse zum angeschlossenen Arbeitsplatzrechner übertragen.



Abbildung 7: Optolyzer Interface Box [45]

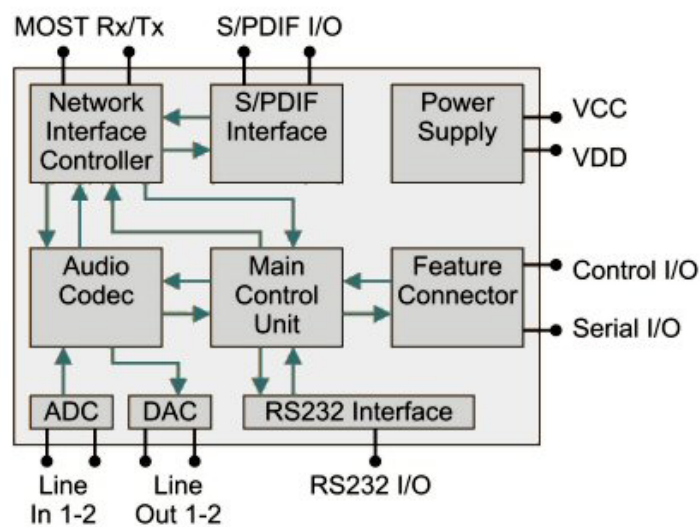


Abbildung 8: Blockschaltbild des Optolyzers [45]

Ein Optolyzer kann in einer der drei folgenden Betriebsarten exklusiv betrieben werden:

- Als **Master-Gerät**.
- Als **Slave-Gerät**. In dieser Betriebsart ist der Optolyzer ebenso wie als Master-Gerät im Netzwerk sichtbar und kann adressiert werden. Ebenso wie

im Master-Modus kann der Optolyzer nur diejenigen Kontrollnachrichten empfangen, die direkt an ihn adressiert sind.

- Im **Spy-Modus**: Im Spy-Modus ist der Optolyzer für die anderen Geräte im Ring nicht sichtbar und kann selbst keine Nachrichten versenden, in dieser Betriebsart kann somit nur eine passive Beobachtung erfolgen. Wird ein Gerät in diesem Modus betrieben, so erzeugt es auch keine Verzögerung im Ring.

Ein Vorteil bei dieser Betriebsart ist, dass der Optolyzer den gesamten Netzwerkverkehr auf dem Kontrollkanal beobachten kann (*promiskutiver Modus*).

Unter Verwendung von einem oder mehreren Optolyzern ergeben sich daraus die folgenden Konfigurationsmöglichkeiten für die Beobachtung eines MOST-Systems:

Eine mögliche Konfiguration besteht in der Verwendung eines Optolyzers, der im Spy-Modus betrieben wird (vgl. Abbildung 9). Damit können dann der gesamte Netzwerkverkehr und alle wichtigen Systemereignisse passiv beobachtet werden. Allerdings sind aktive Messungen oder das Abfragen von Diagnoseschnittstellen nicht möglich.

Eine wichtige Rolle spielt hierbei die Position des Optolyzers im Ring. Dies liegt daran, dass MOST-Geräte die Eigenschaft haben, bestimmte Systemereignisse nicht weiterzupropagieren oder verloren gegangene Signale wieder aufzubereiten. Ein Beispiel dafür ist der kurzzeitige Verlust des Synchronisationssignals. Bei diesem Ereignis wird das entsprechende Fehlerflag erst ab einer bestimmten Mindestzeitdauer des Verlustes weiterpropagiert.

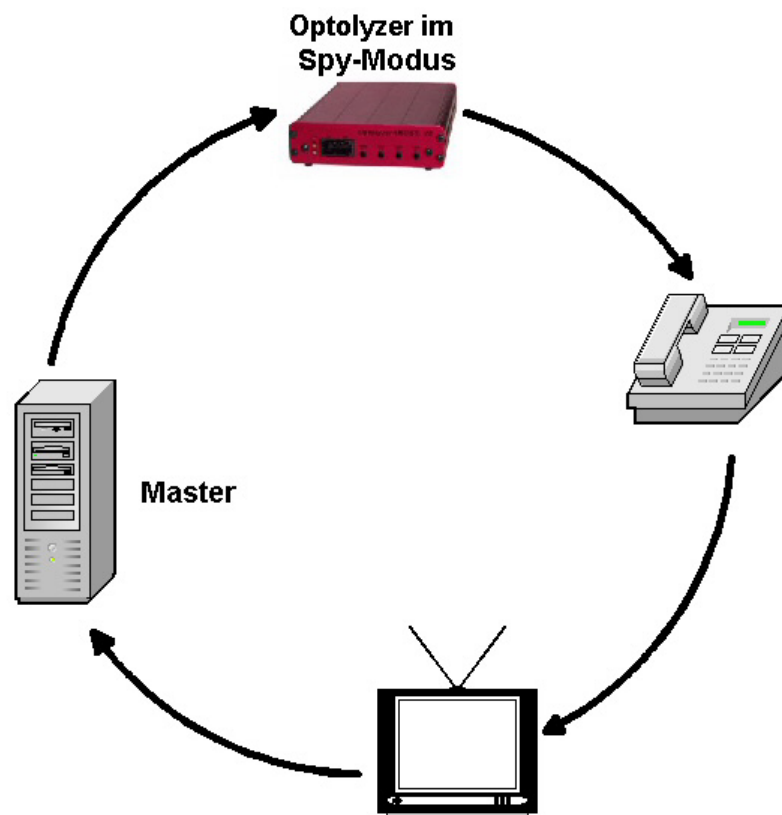


Abbildung 9: Ringkonfiguration mit Optolyzer im Spy-Modus

Werden, wie in Abbildung 10 dargestellt, zwei Optolyzer verwendet, so kann einer davon weiterhin die passive Beobachtung des Kontrollkanals im Spy-Modus übernehmen. Der zweite Optolyzer wird als Slave-Gerät betrieben und kann beispielsweise mit anderen Geräten kommunizieren, einfache Netzwerkfehler mit Hilfe der Optolyzer-Software erzeugen (vgl. Abschnitt 4.3.3) oder eine Diagnoseschnittstelle der verschiedenen Geräte im Ring abfragen.

Tests, die mit dieser Konfiguration durchgeführt wurden umfassen beispielsweise die künstliche Erzeugung von Buslast, Licht- und Synchronisationsfehlern oder Lasttests einzelner Geräte.

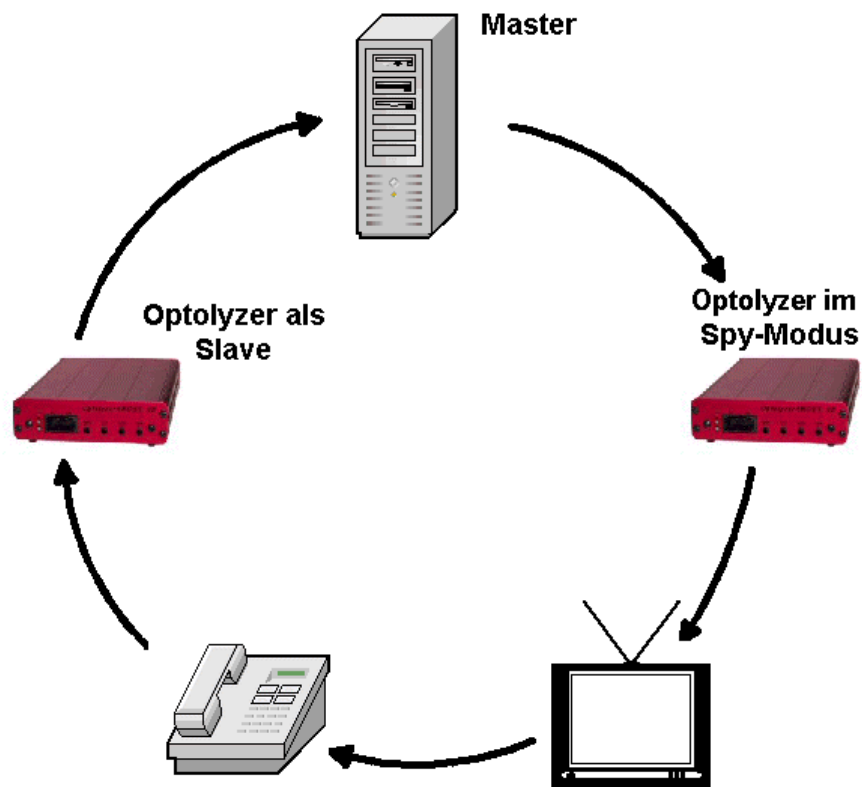


Abbildung 10: Ringkonfiguration mit zwei Optolyzern

Prinzipiell bestünde auch die Möglichkeit, einen Optolyzer alternierend im Slave- und im Spy-Modus zu betreiben. Da die Umschaltung eines angeschlossenen Gerätes jedoch einer Rekonfiguration des Netzwerks entspricht – im Slave-Modus ist der Optolyzer im Netzwerk sichtbar, im Spy-Modus dagegen nicht – ist die Beeinflussung des Systems dabei sehr hoch, da die Adressen der nachfolgenden Geräte und andere Konfigurationsparameter bei jeder Umschaltung neu festgelegt werden müssten.

Für die konkrete Durchführung der Beobachtungsversuche stehen für diese Arbeit zwei Umgebungen zur Verfügung:

- **MOST-Board:** Bei dieser Versuchsumgebung handelt es sich um einen Aufbau, der dieselben Telematikgeräte wie ein gut ausgestattetes Serienfahrzeug enthält. Die einzelnen Telematikgeräte sind dabei über Lichtwellenleiter in Ring-Topologie vernetzt. Im Gegensatz zu einem Fahrzeug können die einzelnen Geräte aber einfach aus dem Ring entfernt, wieder hinzugefügt oder in der Reihenfolge vertauscht werden.

In einem in der aktuellen Mercedes E-Klasse eingebauten Fahrzeugtelematiksystem hat das Master-Gerät zusätzlich die Funktion eines *Gateways* zwischen dem MOST-Bus und einem anderen Bussystem im Fahrzeug, dem *CAN-Bus*. Ein Teil der Nachrichten auf dem CAN-Bus wird dabei über das Master-Gerät auf den Kontrollkanal des MOST-Bus weitergeleitet. Mit Hilfe eines Notebooks, das eine CAN-Karte enthält, kann der CAN-Bus eines Fahrzeugs simuliert werden, so dass dieser Aufbau durchaus als realistische Versuchsumgebung betrachtet werden kann.

Aus diesem Grund werden ähnliche Aufbauten auch in der Elektronikentwicklung bei DaimlerChrysler zu Testzwecken verwendet.

- **Versuchsfahrzeug:** Als eine weitere Versuchsumgebung steht eine aktuelle Mercedes Benz E-Klasse mit Vollausstattung in Bezug auf die Telematikfunktionen zur Verfügung.

Im Vergleich zum MOST-Board ist auf dem MOST-Bus im Fahrzeug eine wesentlich höhere Buslast zu verzeichnen, was damit zusammenhängt, dass deutlich mehr Nachrichten aus dem „Restfahrzeug“ auf den MOST-Bus weitergeleitet werden und außerdem zusätzliche Daten wie beispielsweise die Navigationsdaten während der Fahrt über den MOST-Kontrollkanal ausgetauscht werden.

Als Einschränkung muss allerdings in Kauf genommen werden, dass die Ringkonfiguration im Gegensatz zum MOST-Board fest ist und die Optolyzer-Geräte nicht in beliebiger Anzahl und an beliebiger Position in den Ring eingehängt werden können.

4.3.2 Umsetzung der Beobachtungsmechanismen in einem Fahrzeugtelematiksystem

Soll die Beobachterkomponente in das Fahrzeugtelematiksystem integriert werden, so bietet sich dafür das Master-Gerät an. Das Master-Gerät verfügt über die Kenntnisse der meisten Systemfunktionen und kann daher sehr viel vom Systemverhalten beobachten, auch wenn der Transceiverchip im Master-Modus und nicht im Spy-Modus betrieben wird. Dies wäre die preisgünstigste Lösung, da keinerlei Zusatzgeräte in Fahrzeug für die Beobachtung nötig sind. Ferner sind bei dieser Lösung aktive Messungen wie z.B. das Abfragen von Diagnoseschnittstellen möglich.

Alternativ könnte man das Mastergerät um einen zweiten Transceiverchip, einem so genannten *Secondary Node*, erweitern, der im promiskutiven Modus betrieben wird und daher den gesamten Netzwerkverkehr beobachten kann.

Eine weitergehende Möglichkeit besteht in der Realisierung einer verteilten Monitoring-Lösung. Dabei müsste jedes Gerät Beobachtungsfunktionen implementieren, was die Kosten für das Telematiksystem signifikant erhöhen würde. Außerdem müssten die aufgezeichneten Daten ausgetauscht werden, um ein Gesamtbild des Systemzustands zu bekommen. Dabei bleibt festzustellen, wie der Austausch der Beobachtungsdaten das Systemverhalten beeinflussen würde.

4.3.3 OptoLyzer4MOST Software

Die Optolyzer Interface Box bietet zusammen mit der mitgelieferten Software zahlreiche Funktionen zur Netzwerkanalyse. So kann zwischen den verschiedenen Betriebsarten der am Arbeitsplatzrechner angeschlossenen Optolyzer umgeschaltet werden. Mit Hilfe der Softwaremodule *Viewer* und *Recorder* ist es möglich, den Netzwerkverkehr auf dem Kontrollkanal sowie MOST-Systemereignisse aufzuzeichnen.

Die aufzuzeichnenden Daten können dabei durch einfache Filter eingeschränkt werden. Die Filter beziehen sich dabei auf Werte einzelner Datenfelder der Kontrollnachrichten (Test auf Gleichheit des Feldinhaltes mit dem spezifizierten Referenzwert). Es können mehrere Filter simultan benutzt werden, welche durch die Operatoren logisches *ODER* sowie logisches *UND* verknüpft werden können.

Die Konfigurationsmöglichkeiten der Filter sind dabei sehr eingeschränkt: So lassen sich nur eine beschränkte Anzahl von Filtern verwenden, die dabei nicht über eine Programmierschnittstelle von außen konfiguriert und verknüpft werden können. Es gibt auch keine Möglichkeit, die Analysesoftware um neue Arten von Filtern zu erweitern, um beispielsweise auch statistische Auswertungen durchführen zu können.

Die Software bietet auch Möglichkeiten, die Belegung der synchronen Kanäle in Erfahrung zu bringen und sie zu verändern. Der Inhalt einzelner synchroner Kanäle kann durch die Ausgabe auf einen der Audioausgänge des Optolyzers ausgewertet werden. Über die Audioeingänge können Inhalte auf die synchronen Kanäle eingespeist werden.

Weitere Funktionen umfassen das Lesen und Setzen von Transceiverregistern der angeschlossenen Optolyzer, das Versenden einzelner MOST-Kontrollnachrichten, die Simulation von MOST-Geräten für den Test von MOST-Anwendungen sowie Mechanismen zur Injektion von Fehlern. Die Funktionen zur

Fehlerinjektion umfassen die periodische Generierung von Kontrollnachrichten, Licht- und Synchronisationsfehlern sowie die Simulation eines vollen Empfangspuffers des Optolyzers.

Die OptoLyzer4MOST Software kann mittels des Dynamic Data Exchange-Protokolls (DDE), einem Windows-Mechanismus zur Interprozesskommunikation auf einem Rechner, um weitere Funktionen erweitert werden, die durch externe Programme bereitgestellt werden.

Als Zusatzmodul zur OptoLyzer4MOST Software kann eine ActiveX-Komponente erworben werden (*OptoControl*), die einen Teil der Funktionalität auf einfache Weise mit einer Programmierschnittstelle verfügbar macht. Die im Rahmen dieser Arbeit zu entwickelnde Beobachterkomponente baut darauf auf.

Insgesamt fehlt eine Schnittstelle, die den Zustand beobachteter Systemparameter auf einer höheren Abstraktionsebene anbietet als auf der Ebene einzelner Kontrollnachrichten und Systemereignisse. Somit ist es vergleichsweise aufwändig, Anwendungen für die Visualisierung und Diagnose des beobachteten Systemverhaltens zu realisieren, die direkt auf die OptoLyzer4MOST Software aufbaut.

4.3.4 MOST DataAnalyzer

Mit dem *MOST DataAnalyzer* [44] steht eine kombinierte Hardware- und Software-Lösung zur Analyse des Netzwerkverkehrs zur Verfügung. Der MOST DataAnalyzer kann dabei den Netzwerkverkehr auf dem Kontrollkanal und dem asynchronen Kanal sowie Systemereignisse auswerten. Um mit der höheren Bandbreite auf dem asynchronen Kanal umgehen zu können, verfügt der DataAnalyzer über eine *MOST PCI-Karte* anstelle einer Optolyzer Interface Box.

Die Nachrichten und Ereignisse können dabei aus verschiedenen Datenquellen stammen. So ist es möglich, mit dem DataAnalyzer Benutzereingaben, MOST-Systemereignisse, den MOST-Nachrichtenverkehr und Tracedateien zu verarbeiten. Die verschiedenen Daten werden dabei auf ein einheitliches Typmodell abgebildet.

Mit Hilfe von Filtern können die ankommenden Daten ausgewertet werden. Dazu stehen mehrere vordefinierte Filter zur Verfügung:

- Filter, die nur bestimmte Ereignisse blockieren oder zur weiteren Auswertung passieren lassen. Die einzelnen Ereignisse können dabei nach ihrem jeweiligen Typ oder nach ihren Feldinhalten gefiltert werden.
- Filter, die in der Lage sind, segmentierte Nachrichten zusammensetzen.
- So genannte Trigger, die für eine bestimmte Zeitdauer oder eine bestimmte Anzahl von Ereignissen nach dem Auftreten eines definierten Ereignisses geöffnet oder geschlossen sind.

Die zu verwendenden Datenquellen, Filter und Datensenken können grafisch verknüpft und konfiguriert werden.

Die Gesamtkonfiguration ist dabei in zwei so genannte Pfade unterteilt. Als Verbindung zwischen beiden Pfaden existiert ein *Puffer* mit begrenzter Kapazität, der Schwankungen bei der Anzahl der auftretenden Ereignisse bis zu einem gewissen Grad ausgleichen kann, ohne dass dabei Daten verloren gehen. Der Pfad, der von den Datenquellen bis zum Hauptpuffer reicht, wird als *Storage-Pfad* bezeichnet, derjenige Pfad zwischen dem Hauptpuffer und der Ausgabe der Analyseergebnisse als *Analyse-Pfad*. Im Storage-Pfad sollten daher nur Auswertungen stattfinden, die wenig rechenintensiv sind.

Die Ergebnisse der Datenauswertung werden so genannten *Analyzern* zugeführt. Bei Analyzern handelt es sich also um Datensenken. Dabei stehen drei Arten von Datensenken zur Verfügung:

- **Graphen:** Bei einem Graph handelt es sich beim DataAnalyzer um eine grafische Anzeige der Buslast über der Zeitachse. Die angezeigte Buslast kann dabei nach verschiedenen Kriterien differenziert werden. So kann die Buslast z.B. für jeden Nachrichtentyp, jedes Quell- oder jedes Zielgerät getrennt dargestellt werden.
- **Recorder:** Ein Recorder dient der Aufzeichnung der gefilterten Ereignisse in Tracedateien, die zu einem späteren Zeitpunkt ausgewertet werden können. Diese Auswertung kann dann ebenfalls durch die DataAnalyzer-Software erfolgen, wobei eine andere Filterkonfiguration benutzt werden kann. Dieses Vorgehen empfiehlt sich bei umfangreichen und rechenintensiven Auswertungen, die nicht in Echtzeit ausgeführt werden können.
- **DataGrid:** Ein DataGrid bezeichnet die tabellarische Darstellung der gefilterten Ereignisse. Da die Tabelle entsprechend der Anzahl der Ereignisse sehr groß werden kann, besitzt ein DataGrid Möglichkeiten, die Daten zu sortieren, nur ausgewählte Datenfelder der einzelnen Er-

eignisse anzuzeigen oder die verschiedenen Ereignisse farblich zu differenzieren. Pro Filter im vorangehenden Analysepfad kann bei einem DataGrid eingestellt werden, ob die Ereignisse, die einen einzelnen Filter passiert haben, angezeigt oder ausgeblendet werden.

Die Möglichkeiten zur Auswertung des beobachteten Systemverhaltens gehen beim DataAnalyzer klar über diejenigen hinaus, welche die OptoLyzer4MOST Software bietet. Dafür muss man Einschränkungen bei den Möglichkeiten zur aktiven Beobachtung wie der Fehlerinjektion oder der Allokation synchroner Kanäle in Kauf nehmen.

Ähnlich wie bei der OptoLyzer4MOST Software auch besteht allerdings ein Mangel an Möglichkeiten zur weiteren Verdichtung der Daten und zur Visualisierung des beobachteten Systemverhaltens. Es gibt auch keine einfache Möglichkeit, den DataAnalyzer um solche Funktionen zu erweitern. Ferner bietet der DataAnalyzer außer der Verwendung von Tracedateien keine Schnittstelle für Anwendungen, welche die Auswertungsergebnisse weiterverarbeiten möchten. Somit ist es schwierig, direkt auf der DataAnalyzer Software eine Diagnosekomponente aufzusetzen.

5 Konzepte der Beobachtung

Nachdem bereits die Anforderungen an die Beobachterkomponente formuliert wurden, werden in diesem Abschnitt zunächst die Systemparameter beschrieben, die es zu beobachten gilt, bevor die Konzepte erläutert werden, welche zur Umsetzung der Anforderungen dienen.

5.1 Zu beobachtende Ereignisse

Die Beobachterkomponente soll sich lediglich mit der Beobachtung von *Systemparametern* befassen. Unter den Begriff Systemparameter werden im Rahmen dieser Arbeit alle Nachrichten und Systemereignisse, die in einem MOST-System direkt beobachtet werden können, verstanden.

Die Tabellen in Anhang A geben eine Übersicht, welche Systemparameter in dieser Arbeit zur Beobachtung ausgewählt wurden und welche Aussagekraft sie jeweils besitzen.

Die sich aus der Beobachtung ergebenden Informationen werden als Symptome für den Zustand des Telematiksystems aufgefasst. Es ist nicht Aufgabe der Beobachterkomponente, diese Symptome zu interpretieren, um beispielsweise fehlerhafte Komponenten zu identifizieren. Diese Aufgabe wird von einer Diagnosekomponente übernommen, welche die Symptome weiter auswertet.

Durch diese Trennung der Verantwortlichkeiten ist es möglich, die Beobachterkomponente im Rahmen der SeHeT-Architektur oder als eigenständige Monitoringlösung zu verwenden. Dieser Vorteil würde bei einer monolithischen Komponente, die sowohl Beobachtungs- als auch Diagnosefunktionen umfasste, entfallen.

Um verschiedene Mechanismen zur Beobachtung auf ihre Praxistauglichkeit und Systemparameter auf ihre Aussagekraft hin untersuchen zu können, soll die Beobachterkomponente in einem hohen Maß konfigurierbar und erweiterbar sein.

Wie in Abschnitt 2.1.1 beschrieben umfasst die MOST-Technologie mehr als nur die reine Vernetzung von Telematikgeräten. MOST definiert auch den Rahmen für die Entwicklung von Telematikanwendungen. Analog dazu findet die Beobachtung ebenfalls auf der Netzwerkebene und der Anwendungsebene statt. Der auf dem Kontrollkanal beobachtete Nachrichtenverkehr wird daher vollständig ausgewertet, d.h. nicht nur die Header-Informationen werden ausgewertet, sondern auch die enthaltenen Nutzdaten.

5.1.1 Beobachtung auf der Netzwerkebene

Auf dieser Ebene werden bei der Beobachtung verschiedene Arten von Daten erhoben:

- Der Netzwerkverkehr als ganzes wird mit allgemeinen Metriken wie der Anzahl der Nachrichten pro Zeitintervall (Buslast), dem Anteil von Fehlernachrichten oder von Low-Level Retries beschrieben. Dabei wird von den Nutzdaten der Nachrichten abgesehen und lediglich die Protokolldaten ausgewertet. Die Auswertung der Protokolldaten erlaubt dabei auch eine Differenzierung der Metriken auf die unterschiedlichen Systembestandteile wie Geräte, Netzwerksegmente oder Funktionsblöcke. Um die zugehörigen Werte zu den einzelnen Metriken zu erhalten, muss der beobachtete Netzwerkverkehr zu statistischen Informationen aggregiert werden. Tabelle 8 in Anhang A gibt einen Überblick über die statistischen Daten, die im Rahmen dieser Arbeit gewonnen wurden.
- Systemspezifische Nachrichten und Ereignisse werden aufgezeichnet. Dazu gehören beispielsweise der Verlust der Synchronisation (Unlock), Änderungen an der Systemkonfiguration oder Nachrichten zur Administration der synchronen Kanäle.

5.1.2 Beobachtung auf der Anwendungsebene

Von Interesse auf dieser Ebene ist das allgemeine und das spezifische Verhalten von Telematikanwendungen. Die MOST-Spezifikation definiert durch ihr Anwendungsmodell dabei das allgemeine Anwendungsverhalten. Darunter fallen Zeitschranken für die Beantwortung von Anfragen an Anwendungen oder auch standardisierte Kommunikationsabläufe. Die standardisierten Kommunikationsabläufe legen dabei fest, welcher Nachrichtentyp als Antwort auf eine Anfrage eines bestimmten Typs erwartet wird. Indikatoren für das allgemeine Anwendungsverhalten sind beispielsweise Antwortzeiten, das Ausbleiben von Antwortnachrichten oder das Auftreten von unerwarteten Antworten oder Notifikationsnachrichten.

Die Auswertung von Nachrichten, die von Steuergeräten außerhalb des Telematiksystems auf den MOST-Bus weitergeleitet werden fällt unter die Beobachtung spezifischen Anwendungsverhaltens. Beispiele für solche Nachrichten sind CAN-Nachrichten, die über die Stellung des Zündschlosses, die Fahrzeuggeschwindigkeit oder den Kilometerstand Auskunft geben.

Eine weitere Möglichkeit besteht in der Beobachtung anwendungsspezifischer Kommunikationsabläufe, wie sie durch die Spezifikation der einzelnen Telematikgeräte definiert sind, beispielsweise in Form von Sequenzdiagrammen, die den Ablauf von Kommunikationsprotokollen beschreiben (Message Sequence Charts).

5.2 Verarbeitungsmodell

Die (geringe) Häufigkeit des Auftretens von Systemereignissen und die vorhandene Bandbreite auf dem MOST-Kontrollkanal legen es nahe, die Verarbeitung vollständig innerhalb eines Benutzerprozesses abzuwickeln. Um mit größeren Schwankungen in der Ereignisdichte und der Auslastung des Kontrollkanals umgehen zu können, ohne dabei potentiell wichtige Daten zu verlieren, wird ein Puffer eingesetzt. Damit können die Aufzeichnung der Ereignisse und des Netzwerkverkehrs und deren Auswertung bis zu einem gewissen Grad zeitlich entkoppelt werden.

Die Beobachterkomponente soll sich auf die Beobachtung von Systemparametern beschränken und keine Diagnoseaufgaben übernehmen. Dadurch kann die Verarbeitung weitestgehend zustandslos erfolgen, der zu verwaltende

globale Zustand während der Beobachtung ist vergleichsweise klein. Der nächste Abschnitt beschreibt dazu das Architekturmuster *Pipes and Filters* [7], mit dem Anwendungen zur zustandslosen Datenverarbeitung aus mehreren Bausteinen aufgebaut werden können. Um die Erweiterbarkeit und Konfigurierbarkeit der Beobachterkomponente sicherzustellen, wird dieses Architekturmuster in einer angepassten Form verwendet.

Gemäß den Anforderungen sollen mit Tracedateien und der Beobachtung während dem Betrieb von Telematiksystemen verschiedene Datenquellen für die Beobachtung verwendet werden. Um die Verwendung der verschiedenen Datenquellen transparent zu gestalten, kapselt eine Schnittstelle die unterschiedlichen Realisierungen.

Neben der Schnittstelle für die Datenquellen stellt die Beobachterkomponente anderen Anwendungen eine Schnittstelle zur Verfügung, mit der sie die Beobachtungsdaten verfügbar macht und über die sie konfiguriert werden kann (*Anwendungsschnittstelle*).

5.2.1 Das Architekturmuster Pipes and Filters

Die bei der Beobachtung des Telematiksystems aufgezeichneten Daten werden innerhalb der Beobachterkomponente grundsätzlich gemäß dem Architekturmuster *Pipes and Filters* verarbeitet.

Dieses Architekturmuster eignet sich insbesondere für die Verarbeitung von Datenströmen, die in einzelne diskrete Einheiten unterteilt sind. Die einzelnen Dateneinheiten sind dabei unabhängig voneinander. Damit lassen sie sich auch unabhängig voneinander verarbeiten, wodurch kein globaler Kontext dafür verwaltet werden muss.

Aus diesem Grund lässt sich eine Anwendung, die diesem Verarbeitungsparadigma folgt, aus mehreren Bausteinen aufbauen, die unabhängig voneinander agieren. Die einzelnen Bausteine werden dabei als *Filter* bezeichnet. Ein Filter besitzt jeweils einen Eingang, an dem er Daten zur Verarbeitung entgegennehmen kann und einen Ausgang, an dem die Ergebnisse der Verarbeitung bereitgestellt werden. Ein einzelner Filter kann dabei durchaus einen Zustand verwalten, der über eine einzelne Dateneinheit hinausgeht. Allerdings ist die Verwaltung von Zustandsinformationen, die sich über mehrere Filter erstrecken, schwierig zu realisieren.

Die einzelnen Filter werden durch eine zweite Art von Bausteinen, den so ge-

nannten *Pipes*, verbunden. Eine Pipe transformiert Daten nicht, sondern realisiert nur die Weiterleitung der Daten von einem Filter zum nächsten. Pipes besitzen üblicherweise einen Puffer und sind für die Flusskontrolle bei der Weiterleitung der Daten verantwortlich, um die Verarbeitungsgeschwindigkeiten unterschiedlicher Filter anzugleichen. Pipes besitzen innerhalb einer Anwendung bzw. eines Systems eine einheitliche Schnittstelle. Somit ist sichergestellt, dass sich die verschiedenen Filter beliebig miteinander kombinieren lassen.

Mehrere durch Pipes verbundene Filter bilden eine so genannte *Filterkette*. Da jeder Filter genau je einen Ein- und Ausgang besitzt, sind Filterketten immer linear aufgebaut (vgl. Abbildung 11).

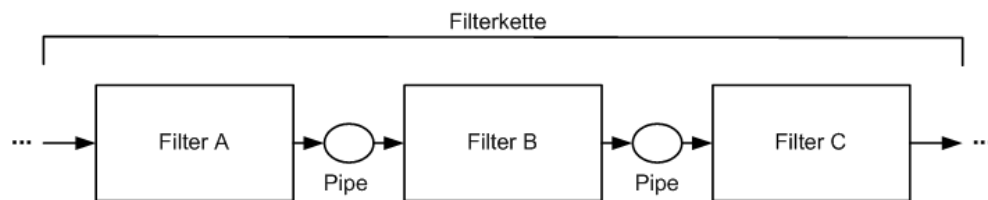


Abbildung 11: Aufbau einer (linearen) Filterkette

Die Verwendung mehrerer paralleler Filterketten würde auch eine parallele Verarbeitung von Daten auf unterschiedlichen Prozessoren. Aufgrund der Zustandslosigkeit ist eine solche Parallelverarbeitung relativ einfach zu realisieren. Die längste Filterkette bestimmt dabei die Skalierbarkeit des Systems.

5.2.2 Anwendung des „Pipes and Filters“-Musters

Grundsätzlich geht das „Pipes and Filters“-Muster davon aus, dass sich die einzelnen Filterinstanzen in verschiedenen Prozessen befinden und sie daher über Nachrichten kommunizieren. Die Beobachterkomponente wird jedoch innerhalb *eines* Prozesses ausgeführt. Die einzelnen Filter sind dabei durch die einfachste Realisierung von Pipes miteinander verbunden - synchronen, lokalen Methodenaufrufen. Aufgrund der Synchronizität entfällt die Implementierung einer Flusskontrolle und des Pufferspeichers. Innerhalb der Beobachterkomponente wird daher das Konzept der Pipes nicht explizit implementiert, sondern durch eine einheitliche Schnittstelle der verschiedenen Filterbausteine umgesetzt.

Besonders um die Effizienz zu erhöhen kann ein Filter mehrere Nachfolger und mehrere Vorgänger besitzen. Damit können Teilaufgaben, die wiederholt in den unterschiedlichen Filterketten eines Systems auftreten, innerhalb einer Filterinstanz durchgeführt werden, wodurch unnötige Mehrfachberechnungen entfallen. Durch diese Erweiterung können nicht nur lineare Filterketten, sondern graphenförmige Filterkonfigurationen erstellt werden. In diesem Fall muss natürlich darauf geachtet werden, dass der Graph zyklensfrei ist, da die Verarbeitung in der Filterkonfiguration sonst möglicherweise in eine Endlosschleife läuft.

Jeder Filter besitzt ferner einen positiven und einen negativen Ausgang. An den beiden Ausgängen werden jeweils komplementäre Dateneinheiten zur Weiterverarbeitung zur Verfügung gestellt. Die Semantik des Komplements ist dabei von Filter zu Filter unterschiedlich. In einem Filter, der nur Systemereignisse, welche Änderungen am Zustand des optischen Signals repräsentieren, am positiven Ausgang passieren lässt, verhalten sich Lichtereignisse und alle anderen Ereignisse komplementär zueinander. Für den Fall, dass ein Filter keine „Filterung“ der Eingangsdaten im engeren Sinn durchführt, also einen Teil der ankommenden Daten weiterleitet und den anderen Teil blockiert, kann der negative Ausgang auch unbeschaltet bleiben.

Damit können sich Filterkonfigurationen wie in Abbildung 12 dargestellt ergeben. Im weiteren Verlauf dieser Arbeit wird aber trotz der fehlenden Linearität von Filterketten gesprochen.

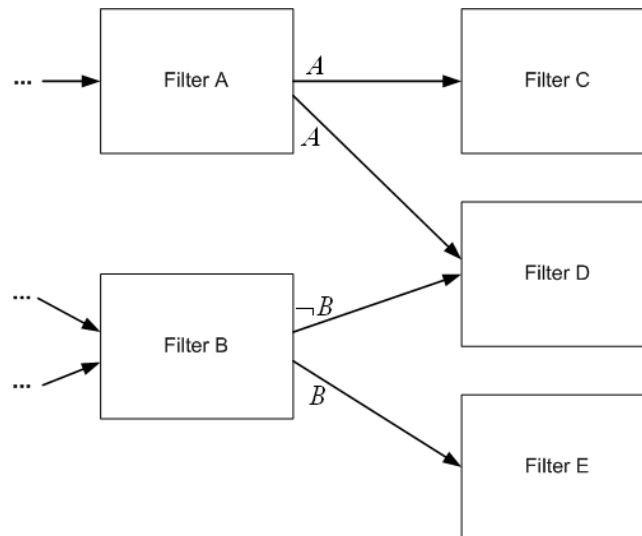


Abbildung 12: Verschiedene Ausgänge und mehrere Nachfolger ermöglichen Filtergraphen

Die Propagierung der einzelnen Dateneinheiten erfolgt durch Push-Kommunikation, d.h. der jeweils vorangehende Filter ist für die Weiterleitung der ausgehenden Daten verantwortlich.

Die Filterketten sind von außen über eine programmatische Schnittstelle konfigurierbar. Über diese Schnittstelle können neue Filterinstanzen erstellt und mit bestehenden Instanzen verbunden werden. Darüber hinaus können die Typinformation und die Eigenschaften der Filterinstanzen abgefragt und geändert werden.

5.2.3 Filter, Prozessoren und Datensenken

Im Rahmen einer Filterkonfiguration werden insgesamt drei Arten von Bausteinen unterschieden: Prozessoren, Filter und Datensenken. Jede Art von Baustein ist dabei durch ihre spezifische Schnittstelle gekennzeichnet, über die Implementierungsinterna nicht ausgesagt wird. Abbildung 13 veranschaulicht die Beziehungen (Schnittstellenvererbung) zwischen den einzelnen Typen von Bausteinen.

Jede Bausteininstanz verfügt dabei über so genannte *Eigenschaften (Properties)*. Eine Eigenschaft ist ein Name-Wert-Paar, der Wert einer Eigenschaft steuert

das Verhalten der entsprechenden Instanz oder dient zu ihrer Beschreibung. Beispiele für Eigenschaften von Bausteinen sind etwa der Bezeichner eines Prozessors oder die Angabe der Intervalldauer eines Bausteins, der als Taktgeber für die Beobachterkomponente fungiert.

Bei einem *Prozessor* handelt es sich um den grundlegendsten Baustein innerhalb einer Filterkonfiguration. Die anderen Bausteine, *Filter* und *Datensenken*, besitzen eine erweiterte Schnittstelle und damit eine erweiterte Funktionalität gegenüber Prozessoren. Ein Prozessor besitzt einen Eingang, um Daten von einem vorangehenden Filter entgegenzunehmen, aber er hat keine Ausgänge um die verarbeiteten Daten weiter zu leiten und kann demnach auch über keine Nachfolger verfügen. Damit terminiert ein Prozessor eine Filterkette.

Ein Prozessor ist durch einen – innerhalb der Filterkonfiguration – eindeutigen Bezeichner gekennzeichnet. Er besitzt lediglich Operationen für seine Aktivierung und Deaktivierung, das Setzen oder Abfragen seines Bezeichners, sowie eine Operation **process**, um Dateneinheiten am Eingang in Empfang zu nehmen. Die Operation **process** definiert somit eine uniforme Kommunikationsschnittstelle (die Signatur ist eindeutig festgelegt, siehe Abschnitt 6.4 im Entwurf). Damit realisiert ein Prozessor implizit die Funktionalität einer Pipe.

Ein *Filter* kann gegenüber einem Prozessor die verarbeiteten Daten weiterleiten. Dazu besitzt er zwei Ausgänge, einen positiven und einen negativen. Jeder Ausgang ist durch einen Bezeichner gekennzeichnet, der innerhalb einer Filterkonfiguration eindeutig sein muss. Die Schnittstelle eines Filterbausteins ist um Operationen zum Setzen und Abfragen der Bezeichner sowie für die Verbindung eines Filters mit Nachfolgerbausteinen bzw. der Lösung von solchen Verbindungen erweitert. Intern besitzt er zusätzlich eine Operation, die für die Weiterleitung der Daten an die Nachfolger verantwortlich ist.

Eine besondere Art von Filter stellt ein *AggregatingFilter* dar. Wie der Name schon verrät, aggregiert ein solcher Baustein eingehende Informationen zu statistischen Werten. Wird die Aggregation dabei relativ zu Zeitintervallen durchgeführt, so ergeben die ausgehenden Daten eine Zeitreihe. Die Aggregation kann dabei auch relativ zum gesamten Beobachtungszeitraum, zu einer Fahrt oder zu einem Betriebszyklus des Telematiksystems erfolgen. Darüber hinaus können die Ergebnisse der statistischen Auswertung auch nach einzelnen Komponenten des Telematiksystems differenziert werden, sofern die zu aggregierenden Daten über Adressierungsinformationen verfügen (was bei Kontrollnachrichten der Fall ist). Als Differenzierungsmerkmale können die FBlock-Instanz, das Quellgerät, das Zielgerät, ein Paar von MOST-Geräten oder ein Abschnitt des MOST-Rings (MOST-Link) verwendet werden.

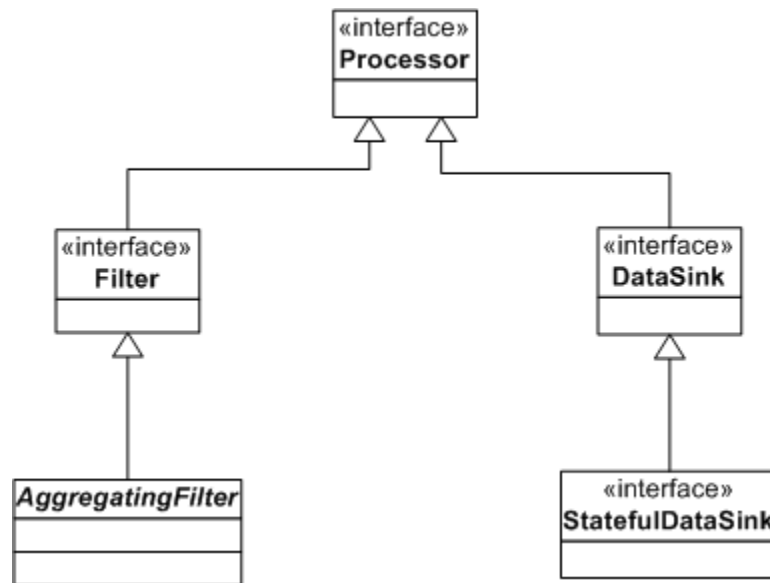


Abbildung 13: Die verschiedenen Typen von Bausteinen und ihre Beziehungen

Eine *Datensenke* terminiert analog zu einem Prozessor eine Filterkette, da sie ebenfalls nicht mit Nachfolgerbausteinen verbunden werden kann. Allerdings stellt eine Datensenke die verarbeiteten Daten über eine Ereignisschnittstelle externen Komponenten zur Verfügung. Dazu definiert eine Datensenke in ihrer Schnittstelle Operationen, mit denen externe Komponenten *Callback-Schnittstellen* zur Ereignisbenachrichtigung registrieren oder deregistrieren können.

Als *Ereignis* wird hierbei das Eintreffen einer Dateneinheit bei einer Datensenke betrachtet. Die Registrierung erfolgt dabei immer für Dateneinheiten eines bestimmten Typs. Der Typ einer Dateneinheit stimmt dabei nicht mit dem programmiersprachlichen Typ (z.B. einer Klasse in Java) überein, sondern wird durch ein Bezeichnerfeld in der jeweiligen Dateneinheit beschrieben. Das Bezeichnerfeld hat dabei den Bezeichner des zuletzt passierten Filterausgangs als Wert.

Die an einer Datensenke verfügbaren Ereignistypen können durch eine entsprechende Operation abgefragt werden. Durch die Verwendung von Schnittstellen bleibt die konkrete Implementierung der Ereignisbenachrichtigung offen. So ist es beispielsweise möglich, dass sich die zu benachrichtigende externe Komponente im selben Prozess und damit im gleichen Adressraum befindet. In diesem Fall würde die Benachrichtigung durch lokale Methodenaufrufe erfolgen. Falls sich die externe Komponente in einem ande-

ren Adressraum befindet, könnte die Benachrichtigung über entfernte Methodenaufrufe (*Remote Method Invocation, RMI*) erfolgen.

Datensenken sind standardmäßig zustandslos, d.h. nach der erfolgten Ereignisbenachrichtigung werden die Ereignisse nicht weiter gespeichert. *Zustandsverwaltende Datensenken* verwalten auch nach der Benachrichtigung die eingetroffenen Dateneinheiten. Externe Komponenten können auf die verwalteten Informationen dabei mit Hilfe von Abfrageoperationen zugreifen.

5.2.4 Verwaltung von Kontextinformationen

Bei Kontextinformationen handelt es sich um Zusatzinformationen zu den einzelnen beobachteten Kontrollnachrichten und Systemereignissen. Diese Kontextinformationen sind vor allem für die interne Verarbeitung innerhalb einer Filterkonfiguration von Bedeutung, in den Fällen der MOST-Konfiguration (Inhalt der CentralRegistry) und der Allokation synchroner Kanäle sind diese Informationen jedoch möglicherweise auch für externe Anwendungen, welche die Beobachterkomponente benutzen, von Interesse. Bei den in der Beobachterkomponente benutzten Kontextinformationen handelt es sich um die folgenden Umgebungsdaten für einzelne Kontrollnachrichten oder Systemereignisse :

- Belegung synchroner Kanäle (Allokationstabelle)
- Beschreibung der MOST-Konfiguration (CentralRegistry)
- Fahrtenzähler (bezieht sich auf Fahrzeug)
- Nummer des Startup-/Shutdown-Zyklus (Betriebszyklus) des Fahrzeugtelematiksystems
- km-Stand
- Zeitintervall, in dem eine Nachricht verschickt wurde oder ein Systemereignis auftrat
- Registrierungen für Notifikationsnachrichten

Die hier genannten Kontextinformationen sind von sehr unterschiedlicher Natur. Daher kommen in der Beobachterkomponente zwei unterschiedliche Mechanismen für die Verwaltung und Bekanntmachung der Kontextinformationen zum Einsatz:

- **Die Verwaltung der jeweiligen Kontextinformation an einer zentralen Stelle innerhalb der Beobachterkomponente.** Interessierte Bausteine aus

der Filterkonfiguration können dabei über eine Programmierschnittstelle auf diese Informationen zugreifen.

- **Die Propagierung der jeweiligen Kontextinformation an alle Bausteine innerhalb der Filterkonfiguration.** Diese Informationen werden nicht zusätzlich an einer zentralen Stelle gespeichert. Bei den zu propagierenden Kontextinformationen handelt es sich um Dateneinheiten, die durch die Verwendung eines speziellen Typs (vgl. die Beschreibung des Datenmodells in Abschnitt 6.8) gekennzeichnet sind. Externen Komponenten werden die propagierten Kontextinformationen dabei nicht über Datenscken zur Verfügung gestellt.

Die Entscheidung, welche Möglichkeit für welche Art von Kontextinformationen benutzt wird, hängt dabei von mehreren Faktoren ab:

- Sind bei der jeweiligen Kontextinformation Zustände aus der Vergangenheit mit zu berücksichtigen? (Verwaltung an zentraler Stelle geeigneter)
- Wie umfangreich sind die jeweiligen Informationen? (Bei großem Umfang der Informationen ist die Verwaltung an zentraler Stelle vorzuziehen)
- Wie häufig und von wie vielen Filterbausteinen werden die jeweiligen Informationen benötigt? (Informationen, die häufig benötigt werden, bieten sich zur Propagierung an)
- Sind die jeweiligen Informationen für eine externe Komponente von Interesse?

Die folgende Tabelle soll verdeutlichen, warum welcher Mechanismus für die Bereitstellung der unterschiedlichen Kontextinformationen verwendet wurde:

Kontextinformation (verwendete Bausteinart)	Vergangenheitswerte zu berücksichtigen?	Umfang	Häufigkeit der internen Nutzung	Interessant für externe Komponente	Verwendeter Mechanismus
Allokationstabelle (Filter)	ja	hoch	selten	evtl. ja	Zentrale Verwaltung, aber Propagierung von Änderungsereignissen

Kontextinformation (verwendete Bausteinarart)	Vergangenheitswerte zu berücksichtigen	Umfang	Häufigkeit der internen	Interessant für externe Kompo-	Verwendeter Mechanismus
CentralRegistry (Filter)	ja	hoch	selten	evtl. ja	Zentrale Verwaltung, aber Propagierung von Änderungsereignissen
Fahrtennummer (Filter)	nein	gering	eher hoch	nein	Propagierung
Zyklennummer (Filter)	nein	gering	eher hoch	nein	Propagierung
Km-Stand (Filter)	nein	gering	eher hoch	eher nein	Propagierung
Zeitintervall (Filter)	nein	gering	hoch	nein	Propagierung
Notifikationstabelle (Prozessor)	nein	hoch	selten	eher nein	Zentrale Verwaltung

Tabelle 4: Verwendung von Mechanismen zur Bereitstellung der Kontextinformationen

Wie aus Tabelle 4 ersichtlich, ist die Verwaltung der Kontextinformationen ebenfalls in das Filtermodell integriert. Es gibt jeweils einen Filter- bzw. Prozessorbaustein, der für die Verwaltung einer Kontextinformation verantwortlich ist. Im Gegensatz zu den übrigen Bausteinen, sind diese Bausteine, von der Beobachterkomponente weitestgehend fest vorkonfiguriert. Nur ihre Eigenschaften, wie beispielsweise die Dauer von Zeitintervallen, sind nachträglich konfigurierbar.

5.2.5 Verwendung eines einheitlichen Datenmodells

Die Verwendung einer uniformen Schnittstelle für die Verbindung der verschiedenen Bausteine der Filterkonfiguration ist nutzlos, wenn über diese Schnittstelle keine Daten oder völlig beliebige Daten übermittelt werden können.

Aus diesem Grund wird in dieser Arbeit ein einheitliches Datenmodell erarbeitet, das sowohl alle beobachteten Nachrichten und Systemereignisse als auch die Kontextinformationen abbildet.

Der konkrete Aufbau des Datenmodells wird in Abschnitt 6.8 vorgestellt.

6 Realisierung

Dieses Kapitel beschreibt die Realisierung der Beobachterkomponente. Dabei wird zunächst ein Überblick über die Gesamtarchitektur gegeben. Anschließend werden die einzelnen Teile und ihr Zusammenspiel genauer beschrieben.

6.1 Architektur der Beobachterkomponente

Im vorherigen Kapitel wurden die Konzepte vorgestellt, welche die Realisierung der Beobachterkomponente prägen. Das Hauptaugenmerk lag dabei auf der Vorstellung eines Verarbeitungsmodells, das eine Variante des Architekturmodells Pipes and Filters verwendet. Die Verwendung dieses Modells ermöglicht die flexible Konfiguration der verschiedenen Bausteine um verschiedene Systemparameter beobachten zu können.

Neben den Filterketten, in denen der größte Teil der Datenverarbeitung stattfindet, besteht die Beobachterkomponente aus den folgenden Teilkomponenten:

- Datenquelle
- Puffer
- Datensenken
- Konfigurationskomponente

Abbildung 14 zeigt das Zusammenspiel der verschiedenen Komponenten.

Eine *Datenquelle* zeichnet den Netzwerkverkehr und die auftretenden Systemereignisse auf. Dabei werden die aufgezeichneten Daten auf das Datenmodell der Beobachterkomponente abgebildet bevor sie zur weiteren Verarbeitung in den *Hauptpuffer* gestellt werden.

Die konfigurierten *Filterketten* holen die Daten sukzessive aus dem Hauptpuffer und verarbeiten sie inkrementell. Die Endpunkte der Filterketten bilden die *Datensenken*. Dort angelangt, werden die Daten an registrierte externe Komponenten (*Monitoring-Anwendungen*) weitergeleitet. Die externen Komponenten können diese Daten dann weiter verarbeiten, z.B. zu Diagnosezwecken oder sie einfach nur mitschreiben oder grafisch aufbereitet ausgeben.

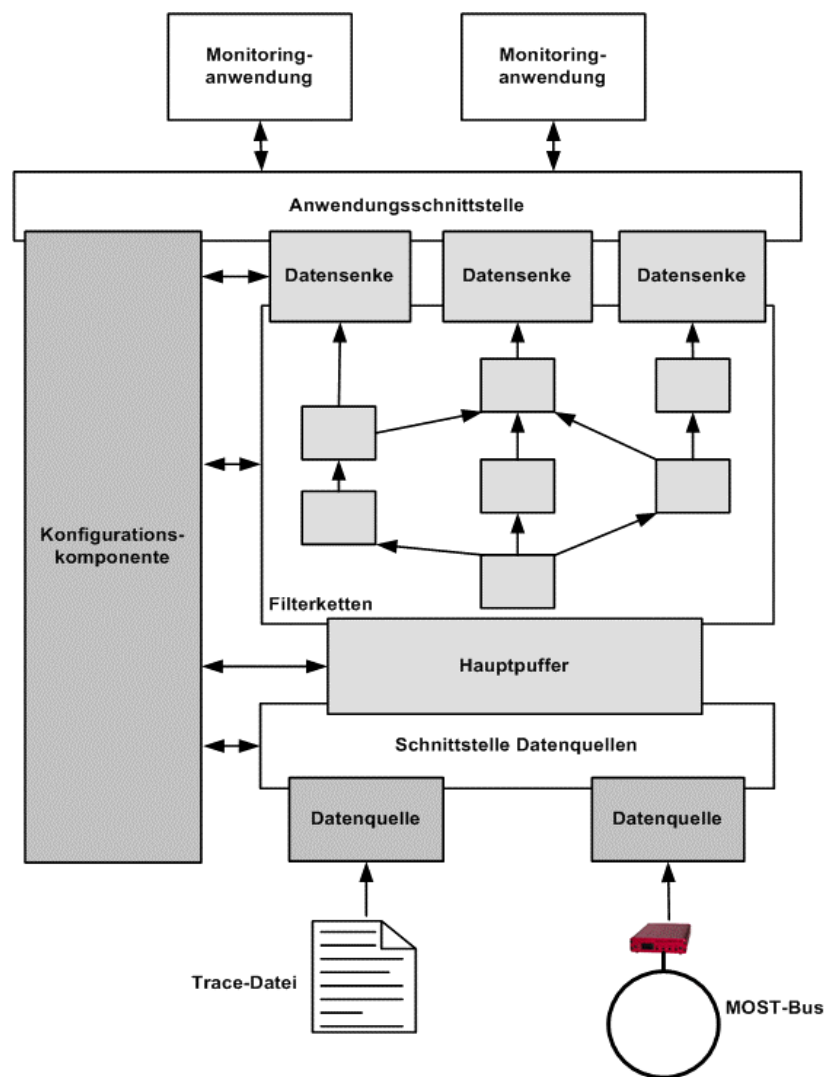


Abbildung 14: Architektur der Beobachterkomponente

Eine Monitoring-Anwendung, die das von der Beobachterkomponente bereitgestellte Framework benutzt, kann dabei über die *Anwendungsschnittstelle* die

Konfiguration der Filterketten steuern oder die Beobachterkomponente starten und anhalten.

Die *Konfigurationskomponente* ist dafür verantwortlich, dass die einzelnen Teile der Beobachterkomponente miteinander verbunden und konfiguriert werden. So sorgt die Konfigurationskomponente beispielsweise dafür, dass die Datenquellen und die Filterketten über den Hauptpuffer miteinander verknüpft sind.

Bei der Realisierung wurde darauf geachtet, dass die Beobachterkomponente mit möglichst geringem Aufwand für den Einsatz in eingebetteten Systemen angepasst werden kann.

Die vorgestellten Teilkomponenten werden in den folgenden Abschnitten im Einzelnen beschrieben.

Die vollständigen Schnittstellendefinitionen der einzelnen Teilkomponenten und der Anwendungsschnittstelle können in Anhang B gefunden werden.

6.2 Datenquellen

Der erste Schritt bei der Beobachtung besteht in der Überwachung bestimmter Systemparameter wie dem Netzwerkverkehr durch geeignete *Sensoren*. Somit stellt ein Sensor eine *Datenquelle* dar, die Daten zur weiteren Auswertung im Rahmen der Beobachterkomponente liefert. Gemäß den Anforderungen an die Beobachterkomponente sollen sowohl ein MOST-basiertes Telematiksystem im laufenden Betrieb als auch in Tracedateien aufgezeichnete Nachrichten und Systemereignisse ausgewertet werden können.

Dementsprechend wurden zwei unterschiedliche Datenquellen realisiert, die Tracedateien einlesen bzw. den laufenden Betrieb über eine Optolyzer Interface Box beobachten können. Um die beiden genannten Arten von Sensoren transparent einsetzen zu können, definiert diese Arbeit die abstrakte Klasse **MOSTReader**, die eine einheitliche Schnittstelle für Datenquellen anbietet.

Grundsätzlich kann die Aufzeichnung durch eine Datenquelle gestartet und angehalten werden. Dazu definiert **MOSTReader** die folgenden Operationen:

start(): Aktiviert die Datenquelle, um mit der Überwachung der Systemparameter zu beginnen.

stop(): Deaktiviert die Datenquelle.

Durch den Aufruf der Operation

setBuffer(Buffer buffer)

erhält eine Datenquelle die Referenz auf den Hauptpuffer. Die Datenquelle nutzt diese Referenz, um ein neu aufgetretenes Ereignis im Hauptpuffer abzulegen.

Schließlich kann die Lokation der Ereignisquelle festgelegt werden. Dies geschieht durch den Aufruf von

setSource(String sourceURL).

Der Parameter **sourceURL** kann dabei auf den MOST-Bus (Wert: `online://`) oder auf eine Tracedatei im lokalen Dateisystemen (Wert: `file://<Pfad>`) verweisen.

6.2.1 Auswertung von Tracedateien

Bei Tracedateien handelt es sich um Textdateien, in denen vorwiegend Kontrollnachrichten, aber auch bestimmte Systemereignisse wie Unlocks oder das Ein- und Ausschalten des optischen Signals auf dem MOST-Bus aufgezeichnet sind. Die Aufzeichnung der Tracedateien erfolgt jeweils mit der Optolyzer4MOST Software. Die Tracedateien werden zeilenweise eingelesen, wobei davon ausgegangen wird, dass der Aufbau jeder Zeile einem bestimmten, festgelegten Format entspricht. Dieses Format ist in Abschnitt 6.11 beschrieben.

Unabhängig von der Art des Ereignisses, das von einer Zeile in der Tracedatei repräsentiert wird, wird beim Parsen zunächst die Information über das optische Signal und das Synchronisationssignal ausgewertet, gefolgt von einem Zeitstempel.

Der restliche Teil einer Zeile unterscheidet sich dann für jede Art von Ereignis. Bei Änderungen des optischen Signals oder des Synchronisationssignals liegen keine weiteren Informationen mehr vor.

Bei normalen Kontrollnachrichten folgen zunächst die Quell- und Zielgeräteadressen vor dem Nachrichteninhalt.

Die Auswertung der Informationen zur Geräteadressierung erfolgt dabei innerhalb der Klasse **MOSTTraceReader**, die für das Einlesen der Tracedatei

verantwortlich zeichnet. Die Auswertung der übrigen Informationen wird durch einen Konstruktoraufruf an die entsprechende Klasse des Datenmodells delegiert. Die Auswahl der entsprechenden Klasse des Datenmodells ist allerdings noch Aufgabe von **MOSTTraceReader**. Nachdem die Abbildung auf das interne Datenmodell erfolgt ist, ruft **MOSTTraceReader** die **put**-Operation der Pufferschnittstelle auf, um das Datum in den Hauptpuffer zu stellen.

6.2.2 Beobachtung während des laufenden Betriebs (Onlinebetrieb)

Beim Onlinebetrieb wird direkt der MOST-Bus abgehört. Wie in Abbildung 15 dargestellt, sind dabei zahlreiche verschiedene Softwareeinheiten beteiligt. Der in der Optolyzer Interface Box befindliche Transceiverchip propagiert geänderte Registerinhalte über den entsprechenden Gerätetreiber (dort sind die *MOST NetServices* realisiert) wahlweise zur Optolyzer4MOST Software oder an eine *ActiveX*-Komponente (*OptoControl*), welche die NetServices-Schnittstelle kapselt.

Eingegangene Nachrichten oder aufgetretene Systemereignisse werden vom OptoControl als COM-Ereignisse (*Component Object Model* [35]) zur Verfügung gestellt.

Das OptoControl stellt zwar eine *ActiveX*- bzw. *COM*-Komponente dar, verfügt aber über keinerlei Serverfunktionalität. Zu diesem Zweck wurde ein *COM*-Server in Visual Basic geschrieben, der nichts anderes macht, als die Ereignisse des OptoControls ohne jegliche Transformation durchzuleiten. Aus Gründen der Leistungsfähigkeit wurde der *COM*-Server als so genannter *In-Process Server* realisiert, der während der Laufzeit an denjenigen Client-Prozess gebunden wird, der das OptoControl nutzen möchte. Somit entfallen zumindest an dieser Stelle aufwändige prozessübergreifende Prozeduraufrufe.

Um die *COM*-Ereignisse konsumieren zu können, muss sich ein entsprechen-

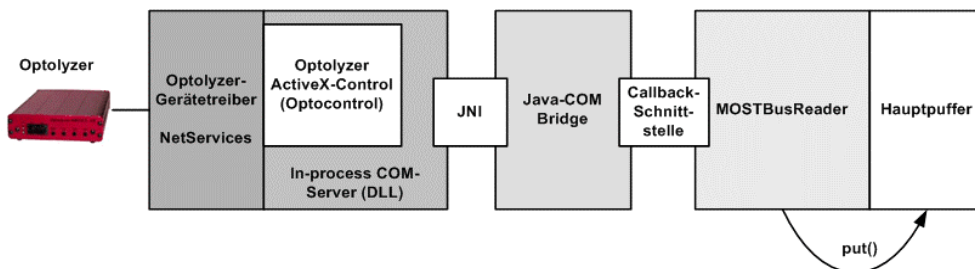


Abbildung 15: An der Aufzeichnung beteiligte Softwareeinheiten

der Client bei der COM-Komponente mit Hilfe einer Callback-Schnittstelle registrieren. Die Beobachterkomponente ist in Java realisiert, das in einer speziellen Ablaufumgebung, der so genannten *Java Virtual Machine (JVM)*, abläuft. Die JVM definiert dabei eine spezielle Schnittstelle, mit der Java-Anwendungen auf nativen Code zugreifen können, das *Java Native Interface (JNI)* [56].

Um die Verwendung des Java Native Interface zu vereinfachen und um von den COM-Interaktionen zu abstrahieren, kommt eine generische *Java-COM Bridge (JACOB)*, vgl. [28]) zum Einsatz, die das Java Native Interface kapselt. JACOB bietet zu diesem Zweck ein Java-API, mit dem COM-Operationen aufgerufen werden können und mit dem sich in Java formulierte Callback-Schnittstellen an der COM-Komponente registrieren lassen.

Generiert die OptoControl-Komponente ein Ereignis, so wird der entsprechende Callback-Aufruf innerhalb der Beobachterkomponente weiter an eine Instanz von **MOSTBusReader** delegiert. **MOSTBusReader** implementiert ebenfalls die genannte Callback-Schnittstelle.

Die Klasse **MOSTBusReader** ist analog zur Klasse **MOSTTraceReader** verantwortlich für das Parsen der empfangenen Systemereignisse und die Abbildung auf das interne Datenmodell. Nach der Abbildung in das interne Datenmodell werden die Daten ebenfalls durch den Aufruf der **put**-Operation an der Pufferschnittstelle im Hauptpuffer abgelegt.

6.3 Hauptpuffer

Die von einer Datenquelle gelieferten Informationen werden im Hauptpuffer abgelegt, bevor sie von einer Filterkette zur weiteren Auswertung aus dem Puffer geholt werden. Der Hauptpuffer arbeitet dabei nach dem FIFO-Prinzip (First In First Out). Um unterschiedlichen Ressourcensituationen und Anforderungen an die Leistungsfähigkeit gerecht zu werden, kann die Kapazität des Puffers wahlweise beschränkt werden.

Der Puffer ist typisiert und nimmt nur Daten gemäß dem in Abschnitt 6.8 beschriebenen Datenmodell auf. Dazu definiert die Pufferschnittstelle die beiden Operationen

- **put(DataItem item)**
- **get():DataItem**

Um die Leistungsfähigkeit der Beobachterkomponente bei beschränkter Pufferkapazität einschätzen zu können (d.h., ob Ereignisse wegen zu langsamer Verarbeitung verloren gehen), können an der Pufferschnittstelle sowohl die Anzahl aller `put()`-Aufrufe als auch die Anzahl derjenigen Ereignisse, die wegen eines vollen Puffers abgewiesen werden mussten, abgefragt werden.

6.4 Filter und Prozessoren

In Abschnitt 5.2.3 wurden bereits die verschiedenen Arten von Bausteinen aus konzeptioneller Sicht beschrieben. Die einfachsten Bausteine stellen demnach die *Prozessoren* dar. Da ein Prozessor über keine Ausgänge verfügt und somit auch keine Nachfolgerbausteine in einer Filterkonfiguration zu verwalten hat, besteht seine Schnittstelle nach außen hin lediglich aus Operationen, die seine Aktivierung und Deaktivierung steuern sowie zur Festlegung und Abfrage seines Bezeichners.

Die Operation

`open()`

aktiviert die Prozessorinstanz, damit sie an ihrem Eingang Dateneinheiten entgegennehmen kann. Die komplementäre Operation hat die Signatur

`close()`.

Sie dient dementsprechend zur Deaktivierung der Prozessorinstanz.

Mittels der Operation

`setID(String ID)`

kann der Bezeichner einer Prozessorinstanz festgelegt werden. Diese Methode ist dabei **nicht** für die Prüfung der Eindeutigkeit des Bezeichners verantwortlich. Die Verantwortung dafür hat die Konfigurationskomponente (vgl. Abschnitt 6.6), welche die Filterkonfiguration kapselt. Der Wert des Bezeichners einer Prozessorinstanz kann durch den Aufruf von

`getID():String`

festgestellt werden.

Ein *Filter* ist gegenüber einem Prozessor durch das Vorhandensein zweier Ausgänge erweitert und zeichnet auch für die Verwaltung eventueller Nachfolgerbausteine verantwortlich. Ein Filter besitzt im Gegensatz zu einem Prozessor keinen eindeutigen Bezeichner, sondern die beiden Filterausgänge sind durch Bezeichner gekennzeichnet. Seine vom Java-Interface **Filter** definierte Schnittstelle wurde daher um die folgenden Operationen erweitert:

addSuccessor(Processor successor, boolean atPositiveOut)

fügt einen Baustein an den durch das Flag **atPositiveOut** angegebenen Ausgang an.

setNegativeID(String negativeID)

legt den Bezeichner des negativen Ausgangs fest, der durch einen Aufruf von

getNegativeID() : String

abgefragt werden kann.

Wie in Abschnitt 5.2.3 bereits erläutert, sollen nicht nur Filter möglich sein, die einzelne Dateneinheiten weiterleiten oder blockieren, sondern auch solche, die statistische Auswertungen bieten. Jeder Filter, der statistische Auswertungen durchführt, muss in der Lage sein, die statistischen Auswertungen über den gesamten Beobachtungszeitraum oder relativ zu definierten, zeitlich begrenzten Phasen zu berechnen. Ferner muss ein solcher Filter dabei die Differenzierung zwischen verschiedenen Einheiten eines MOST-Systems erlauben. Daher definiert die abstrakte Klasse **AggregatingFilter** die folgenden Operationen, um diese Optionen festzulegen. Mittels eines Aufrufs von

setAggrBase(byte base)

kann festgelegt werden, ob sich die statistischen Auswertungen auf den gesamten Beobachtungszeitraum, eine einzelne Fahrt, einen einzelnen Betriebszyklus des Telematiksystems oder ein Zeitintervall beziehen soll. Mit der Operation

setDifferentiateBy(byte discriminator)

kann bestimmt werden, ob sich die statistische Auswertung auf das Gesamtsystem, ein einzelnes Gerät, einen einzelnen Funktionsblock, ein Paar von MOST-Geräten oder einen MOST-Netzwerkabschnitt beziehen soll.

Basierend auf den beschriebenen Schnittstellen wurden einige konkrete Bausteine realisiert. Ziel war es dabei, möglichst einfache und möglichst allgemeingültige Filter zu erhalten. Die folgende Tabelle gibt einen Überblick über die realisierten Bausteine:

Name (verwendete Bausteин- art)	Beschreibung
AMSMessagesAssembler (Filter)	Setzt segmentierte Nachrichten zusammen
AverageFilter (AggregatingFilter)	Berechnet den Durchschnitt einer einzelnen Zeitreihe
BasicFilter (Filter)	Leitet alle Dateneinheiten unverändert an den positiven Ausgang weiter; dient damit als Verbindungselement zwischen mehreren Filterketten
CentralRegMsgsFilter (Filter)	Wertet Nachrichten vom Protokolltyp <i>CentralRegistry.Status</i> aus und interpretiert ihren Inhalt
CounterFilter (AggregatingFilter)	Zählt die ankommenden Dateneinheiten
DifferenceFilter (AggregatingFilter)	Berechnet die Differenz zweier Zeitreihen pro Paar zeitlich korrelierter Dateneinheiten
ErrorFilter (Filter)	Leitet Fehlermeldungen in aufbereiteter Form weiter
FieldValueFilter (Filter)	Vergleicht den Feldinhalt einer Dateneinheit mit einem Referenzwert. Dabei sind verschiedene Vergleichsoperatoren möglich: =, <, >= und <=
LightLockFilter (Filter)	Wertet Ereignisse bezüglich Licht und Synchronisationssignal aus; andere Ereignisse werden zu negativem Ausgang weitergeleitet
MaximumFilter (AggregatingFilter)	Berechnet das Maximum einer Zeitreihe
MinimumFilter (AggregatingFilter)	Berechnet das Minimum einer Zeitreihe
MovingAverageFilter (AggregatingFilter)	Berechnet den gleitenden Durchschnitt einer Zeitreihe

Name (verwendete Baustein-)	Beschreibung
MovingCorrelationFilter (AggregatingFilter)	Berechnet den Korrelationskoeffizient zweier Zeitreihen
RatioFilter (AggregatingFilter)	Berechnet das Verhältnis zweier Zeitreihen pro Paar zeitlich korrelierter Dateneinheiten
RequestReplyTableFilter (Filter)	Wertet Request-Response Interaktionen und Notifikationsnachrichten aus
RetryFilter (Filter)	Erkennt Low-Level Retries auf der Ebene des CMS
TriggerFilter (Filter)	Generiert Ereignisse, welche die Über- oder Unterschreitung eines Schwellwerts anzeigen und leitet sie an die Nachfolgerbausteine weiter
TypeFilter (Filter)	Lässt nur Ereignisse eines bestimmten (Java-)Typs passieren, alle anderen werden an den negativen Ausgang weitergeleitet.

Tabelle 5: Für die Auswertung realisierte Bausteine

6.5 Datensenken

Eine *Datensenke* hat die Aufgabe, die während der Beobachtung anfallenden Daten, den Anwendungen, die sich dafür interessieren, zur Verfügung zu stellen. Wie schon in Abschnitt 5.2.3 erläutert, sind Datensenken standardmäßig zustandslos, um die Verwaltung der Daten möglichst einfach zu halten.

Damit Anwendungen die Beobachtungsdaten erhalten können, müssen sie sich zuvor mit Hilfe von Callback-Schnittstellen (definiert durch das Java-Interface **Controller**) bei einer oder mehreren Datensenken registrieren. Dazu definiert das Java-Interface **DataSink** die folgenden Operationen:

- **registerController(Controller controller, String dataSeriesID):boolean**: Registriert eine Implementierung der Callback-Schnittstelle für den Konsum bestimmter Beobachtungsdaten.
- **removeController(Controller controller, String dataSeriesID):boolean**: Deregistriert eine Implementierung der

Callback-Schnittstelle von dem Konsum bestimmter Beobachtungsdaten.

Die Übergabe der Daten erfolgt dann durch eine ereignisbasierte Benachrichtigung über die Callback-Schnittstelle. Dazu wird an der Callback-Schnittstelle die folgende Operation aufgerufen:

```
notify(String dataSeriesID, DataItem item).
```

Neben den Funktionen zur (De-)Registrierung von anwendungsseitigen Callback-Schnittstellen kann bei einer Datensenke auch abgefragt werden, welche Daten zur Verfügung stehen. Die Auskunft erfolgt dabei durch die Angabe einer Menge von Bezeichnern, welche die verfügbaren Daten repräsentieren. Diese Abfrage erfolgt durch den Aufruf der Operation

```
getAvailableIDs():Collection
```

bei einer Datensenke. Als Ergebnis wird eine Menge von Bezeichnern zurückgeliefert, die alle an dieser Datensenke verfügbaren Dateneinheiten repräsentiert.

Das API für Datensenken, die einen Zustand verwalten, ist noch nicht endgültig festgelegt. Im Rahmen dieser Arbeit entstand eine prototypische Abfrageschnittstelle für zustandsverwaltende Datensenken, die auf den Verwendungszweck innerhalb der Beobachterkomponente spezialisiert ist. Diese Abfrageschnittstelle wird durch das Java-Interface **StatefulDataSink** definiert.

Sie stellt dabei Abfrageoperationen zur Verfügung, um alle Daten mit einem bestimmten Bezeichner, alle Einzelereignisse oder lediglich die aggregierten Daten zu erhalten. Optional kann dabei die Größe der Abfrageergebnisse durch die Angabe einer Ganzzahl **count** auf die jeweils **count** aktuellsten Dateneinheiten begrenzt werden.

Es ist noch offen, inwiefern diese Schnittstelle (zukünftigen) Anforderungen genügen wird. Es sind auf jeden Fall alternative und flexiblere Schnittstellen denkbar:

Falls zur Verwaltung des Zustands einer Datensenke beispielsweise eine relationale Datenbank zum Einsatz kommt, würde sich eine SQL-Schnittstelle anbieten. Der Einsatz in eingebetteten Geräten stellt dabei nicht unbedingt einen Hinderungsgrund dar, da schon seit geraumer Zeit relationale Datenbanken verfügbar sind, die speziell für die effiziente und sparsame Nutzung von Ressourcen entworfen wurden, so z.B. DB2 Everyplace von IBM [19].

6.6 Konfigurationskomponente

Die Konfigurationskomponente enthält das Wissen über die Konfiguration der Beobachterkomponente und sorgt für den Zusammenschluss der einzelnen Teilkomponenten. Die Konfiguration der Beobachterkomponente erfolgt dabei teilweise durch die Schnittstelle der Konfigurationskomponente, teilweise durch eine Konfigurationsdatei, welche durch die Konfigurationskomponente ausgewertet wird (vgl. Abschnitt 6.10). Intern besitzt sie daher auch das Wissen über den Aufbau der Konfigurationsdatei.

Die Schnittstelle der Konfigurationskomponente ermöglicht es, die Beobachtung zu starten, sie anzuhalten, die Konfigurationsdatei neu einzulesen und die Filterketten zu konfigurieren. Für den internen Gebrauch steht ferner eine Operation zur Verfügung, um eine Referenz auf den Hauptpuffer zu bekommen. Die von der Konfigurationskomponente zur Verfügung gestellten Operationen sind im Anhang B.5.1 aufgelistet.

Die Operationen zur Abfrage und Modifikation der Filterkonfiguration benutzen dabei die in Java ab der Version 1.3 eingebauten Reflektionsmechanismen (*Java Reflection*). Diese Mechanismen sind auch in der *Connected Device Configuration (CDC)* der *Java 2 Platform Micro Edition (J2ME)* verfügbar.

Da die Beobachterkomponente von außen konfiguriert werden soll, wurde die Schnittstelle der Konfigurationskomponente nahezu vollständig auf die Anwendungsschnittstelle abgebildet.

6.7 Anwendungsschnittstelle

Bei der Beobachterkomponente handelt es sich im Wesentlichen um ein Framework, das erweiterbar und von außen konfigurierbar ist. Eine externe Komponente oder Anwendung besitzt die folgenden Möglichkeiten zur Interaktion mit der Beobachterkomponente:

- Starten und Anhalten der Beobachtung.
- Veranlassen der Beobachterkomponente, ihre Konfigurationsdatei neu einzulesen.

- Festlegung der Filterkonfiguration.
- Registrierung für die Benachrichtigung, wenn die Beobachterkomponente die Datenverarbeitung abgeschlossen hat (z.B. bei der Auswertung von Tracedateien).
- Registrierung für die Benachrichtigung über neue Beobachtungsdaten.
- Abfrage von aktuellen und zurückliegenden Kontextinformationen, um die Beobachtungsdaten selbst weiter auszuwerten. Dies umfasst beispielsweise die Abfrage der aktuellen MOST-Konfiguration, der Belegung synchroner Kanäle oder des Inhalts der Notifikationstabelle.

Im Folgenden werden einige der gerade genannten Interaktionen exemplarisch erläutert. Die Anwendungsschnittstelle wird durch die abstrakte Klasse **MonitoringFacade** definiert. Eine vollständige Beschreibung aller von der Anwendungsschnittstelle angebotenen Operationen findet sich in Anhang B.1.2.

6.7.1 Initialisierung der Beobachterkomponente

Damit eine Monitoring-Anwendung die Beobachterkomponente nutzen kann, muss diese zunächst initialisiert werden. Abbildung 16 zeigt den allgemeinen Ablauf bei der Initialisierung.

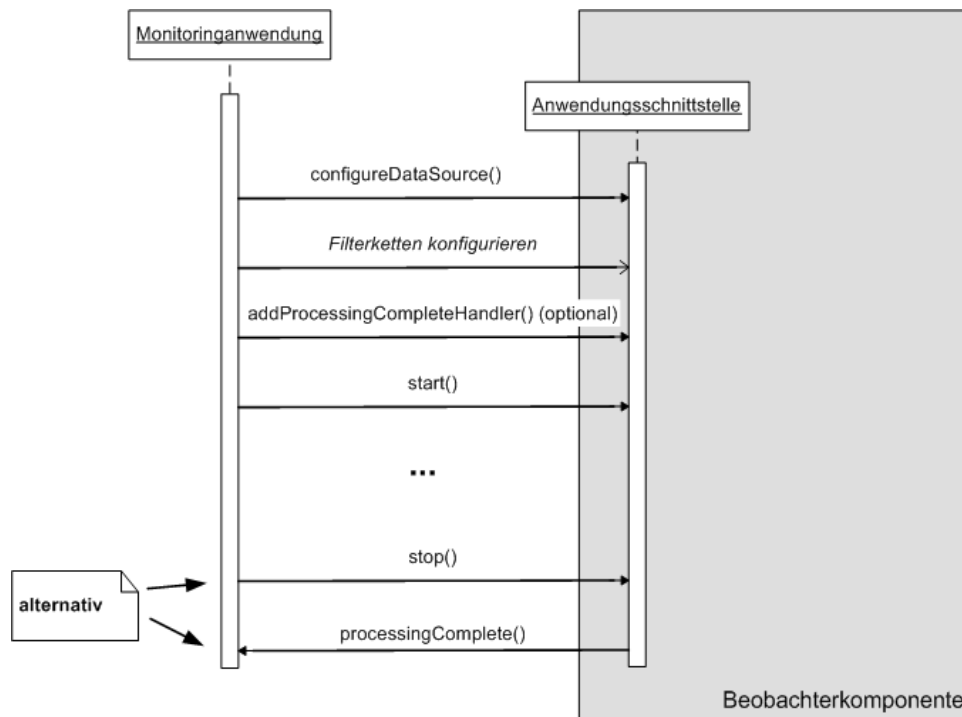


Abbildung 16: Ablauf der Initialisierung

Der erste Schritt besteht dabei in einem Aufruf von `configureDataSource()`. Diese Operation ist verantwortlich für das Einlesen und die Auswertung der Konfigurationsdatei.

Die Konfigurationsdatei muss sich dabei im Java-Klassenpfad der Beobachterkomponente befinden. Sie enthält Angaben zu der zu verwendenden Datenquelle, der Kapazität des Hauptpuffers sowie eine Auflistung aller verfügbaren Bausteinklassen.

Der nächste Schritt besteht in der Konfiguration der Filterketten. Eine Anwendung kann dabei neue Bausteininstanzen anlegen, die Ein- und Ausgänge jeweils zweier Bausteine verknüpfen und die Werte der Eigenschaften einzelner Bausteininstanzen festlegen.

Um die Konfiguration der Filterketten zu vereinfachen, gibt es vielfältige Mög-

lichkeiten zur Introspektion der Filterkonfiguration: So kann abgefragt werden, welche Bausteintypen verfügbar sind, welche Bausteininstanzen existieren und welche Eigenschaften einzelne Bausteininstanzen besitzen.

Es steht einer Monitoring-Anwendung damit offen, ob sie eine hart codierte Filterkonfiguration verwendet, die Filterkonfiguration in einer Konfigurationsdatei beschrieben ist oder ob der Benutzer der Anwendung die Filterkonfiguration dynamisch zur Laufzeit festlegt.

Das folgende Listing zeigt einen kurzen Ausschnitt aus einer hart codierten Filterkonfiguration. Die Operationen werden alle auf der Anwendungsschnittstelle aufgerufen:

```
// Definition eines Filters, der nur Änderungen der
// MOST-Konfiguration (am positiven Ausgang) weiterleitet
createProcessor("com.dcx.most.monitoring.filters.TypeFilter",
               "ET_Configuration");

setProperty("ET_Configuration", "negativeID",
           "obsolete");

setProperty("ET_Configuration", "typeName",
           "com.dcx.most.monitoring.data.config.ConfigStatusChangedEvent");

connectProcessors(
    MonitoringFacade.getInstance().getBaseProcessorID(),
    "ET_Configuration", true);

// Definition der Datensenke
createProcessor(
    "com.dcx.most.monitoring.datasinks.StatelessDataSink",
    "Demo data sink");

// Verbindung des Filters mit der Datensenke
connectProcessors("ET_Configuration", "Demo data sink", true);
```

Listing 1: Ausschnitt aus Filterkonfiguration

Mittels `createProcessor` wird eine neue Instanz des gegebenen Bausteintyps angelegt. Der erste Parameter gibt den qualifizierten Namen des Bausteintyps an, der zweite enthält den Bezeichner für die Instanz.

Durch den Aufruf von `setProperty` kann der Wert der Eigenschaft einer Instanz festgelegt werden. Der erste Parameter bei einem Aufruf von `setProperty` entspricht dem Bezeichner der Instanz, der zweite bezeichnet die zu verändernde Eigenschaft und der dritte Parameter schließlich den neuen Wert der Eigenschaft.

`connectProcessors` dient der Verbindung zweier Bausteininstanzen. Der

erste Parameter bezeichnet den Baustein in der Vorgängerrolle, der zweite Parameter denjenigen in der Nachfolgerrolle. Der letzte Parameter gibt schließlich den Ausgang des Vorgängerbausteins an, der zur Verbindung benutzt werden soll.

In dem obigen Beispiel wurde ein Filter konfiguriert, der nur Ereignisse passieren lässt, die Änderungen an dem Zustand der MOST-Konfiguration repräsentieren. Dieser Filter wurde dann mit einer Datensenke verbunden.

Werden zur Beobachtung Tracedateien verarbeitet, so terminiert die Beobachtung, nachdem die Tracedatei vollständig eingelesen und gemäß der Filterkonfiguration ausgewertet wurde. Die Beobachterkomponente besitzt die Fähigkeit, eine Monitoring-Anwendung über das Ende einer Beobachtung zu benachrichtigen. Um diese Funktionalität in Anspruch zu nehmen, muss die Monitoring-Anwendung eine Callback-Schnittstelle vom Typ **ProcessingCompleteHandler** bei der Beobachterkomponente registrieren. Dies geschieht durch den Aufruf der Operation **addProcessingCompleteHandler** an der Anwendungsschnittstelle. Die Referenz auf die zu benachrichtigende Callback-Schnittstelle wird als Parameter übergeben. **ProcessingCompleteHandler** definiert eine Operation **processingComplete**, die von der Beobachterkomponente im Falle des Endes einer Beobachtung aufgerufen wird.

Damit ist die Initialisierung abgeschlossen. Die Beobachtung kann nun durch einen Aufruf von **start** an der Anwendungsschnittstelle gestartet werden.

Falls die Monitoring-Anwendung von sich aus die Beobachtung abbrechen möchte, muss sie **stop** an der Anwendungsschnittstelle aufrufen.

6.7.2 Zugriff auf die Beobachtungsdaten und Kontextinformationen

Es wird grundsätzlich zwischen Kontextinformationen und Beobachtungsdaten unterschieden, die eher den Charakter von Ereignissen haben.

Da die Bausteine für die Verwaltung und die Bereitstellung der Kontextinformationen weitestgehend statisch konfiguriert werden und nur in geringem Maße von außen beeinflussbar sind, sieht die Anwendungsschnittstelle eine feste Programmierschnittstelle für den Zugriff auf diese Informationen vor. So kann die Monitoring-Anwendung beispielsweise durch den Aufruf von **getCurrentConfigInfo** eine Beschreibung der aktuellen MOST-Konfiguration erhalten oder durch den Aufruf von **getAllocTable** die Belegung der

synchronen Kanäle erfahren. Eine vollständige Beschreibung der Operationen für den Zugriff auf die Kontextinformationen findet sich in Anhang B.1.2.

Welche weiteren Beobachtungsdaten die Beobachterkomponente bereitstellt hängt von der Konfiguration der Filterketten ab. Dies kann von Fall zu Fall unterschiedlich sein. Aus diesem Grund bietet die Anwendungsschnittstelle eine Operation an, mit der abgefragt werden kann, welche Beobachtungsdaten verfügbar sind. Diese Operation heißt **getDataSeriesIDs** und liefert eine Auflistung von Bezeichnern verfügbarer Dateneinheiten zurück.

Ausgehend von diesen Informationen kann sich eine Monitoring-Anwendung für den Konsum der Beobachtungsdaten durch Angabe einer Callback-Schnittstelle registrieren. Zur Registrierung wird die Operation

```
registerController(Controller controller, String dataID):  
boolean
```

aufgerufen. Der Parameter **controller** beinhaltet dabei eine Referenz auf die Callback-Schnittstelle, über welche eine externe Komponente (Monitoring-Anwendung) über neu verfügbare Beobachtungsdaten informiert wird. Mit Hilfe des Parameters **dataID** kann die Monitoring-Anwendung steuern, welche Daten sie konsumieren möchte. Möchte die Anwendung bestimmte Beobachtungsinformationen nicht mehr beziehen, so kann sie sich durch den Aufruf von

```
removeController(Controller controller, String dataID):  
boolean
```

explizit deregistrieren. Die Parameter besitzen bei dieser Operation analoge Bedeutung.

Die Definition der Callback-Schnittstelle zur Benachrichtigung über neu vorliegende Beobachtungsdaten liefert das Java-Interface **Controller**. Wenn neue Beobachtungsdaten verfügbar sind, wird in den registrierten Callback-Schnittstellen die Operation

```
notify(String dataID, DataItem item)
```

aufgerufen. **dataID** gibt dabei an, um welche Art von Beobachtungsdaten es sich handelt, während **item** die Daten selbst enthält.

Darüber hinaus bietet die Anwendungsschnittstelle den direkten Zugriff auf einzelne Datensinken an. Damit lassen sich die erweiterten Funktionen nut-

zen, welche die Schnittstellen einzelner Datensenken bieten. Somit können Datensenken, die erhaltene Beobachtungsdaten selbst verwalten, über entsprechende Operationen abgefragt werden (vgl. Abschnitt 6.5).

Der Zugriff auf die konfigurierten Datensenken erfolgt durch die Operation **getDataSinks** der Anwendungsschnittstelle. Diese Operation liefert eine Liste von Referenzen auf die verfügbaren Datensenken zurück. Durch explizite Typkonvertierung der in der Liste enthaltenen Referenzen auf die entsprechenden spezialisierten Schnittstellen von Datensenken können dann die erweiterten Operationen genutzt werden.

6.8 Das Datenmodell

Um die einheitliche Verarbeitung der Daten durch die autonomen Bausteine zu ermöglichen, wurde im Rahmen dieser Arbeit ein einheitliches Datenmodell entwickelt. Die wichtigsten Klassen und Schnittstellen dieses Datenmodells sowie ihr Zusammenhang werden dabei in dem Klassendiagramm aus Abbildung 17 gezeigt.

Die Basisklasse des Datenmodells ist **DataItem**. **DataItem** definiert die Informationen, die jedes Ereignis und jede Nachricht besitzen. Dazu gehören ein eindeutiger Bezeichner, ein Zeitstempel sowie Informationen, die Hinweise auf den Kontext eines Datums geben. Die Hinweise auf den Kontext beziehen sich jeweils auf den Zeitpunkt, an dem das entsprechende Ereignis oder die entsprechende Nachricht aufgezeichnet wurden. Sie umfassen dabei:

- den Stand des Intervallzählers (numerisch)
- den Stand des Betriebszyklenzählers
- den Stand des Fahrtenzählers
- den Bezeichner der Beschreibung der damals gültigen MOST-Konfiguration
- den Zustand der Konfiguration
- den Kilometerstand

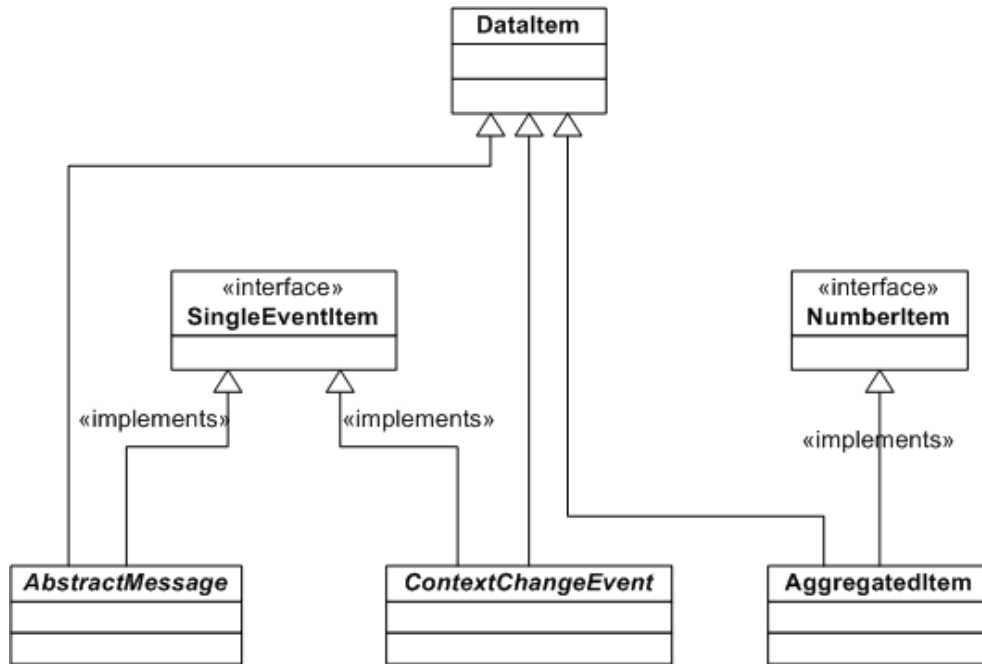


Abbildung 17: Hauptstrukturen des Datenmodells (Klassendiagramm)

Die Schnittstellendefinition **SingleEventItem** dient zur Kennzeichnung von einzelnen Ereignissen und Nachrichten. Sie definiert keinerlei Operationen, es handelt sich um ein so genanntes *Tagged Interface* [46]. Sämtliche Klassen, die Systemereignisse oder Nachrichten repräsentieren implementieren **SingleEventItem**.

Die abstrakte Klasse **AbstractMessage** dient als Basis für Subklassen, die verschiedene Arten von Nachrichten repräsentieren. **AbstractMessage** erweitert **DataItem** um Adressierungsinformationen (Quelladresse, Zieladresse, FBlockID und InstID), die Angabe des Nachrichtentyps und einem Feld für die Nutzdaten.

Die Klassen, die Systemereignisse repräsentieren, sind direkt von **DataItem** abgeleitet und implementieren ebenfalls die Schnittstelle **SingleEventItem**. Aufgrund ihrer Diversität gibt es keine gemeinsame Oberklasse für die verschiedenen Systemereignisse.

Die Schnittstellendefinition **NumberItem** dient zur Kennzeichnung von Daten, die numerische Werte repräsentieren wie z.B. die Buslast oder die Antwortzeit für eine Anfrage. Auf den numerischen Wert kann dabei einheitlich mittels der Operation **extractValue** zugegriffen werden.

Die Klasse **AggregatedItem** repräsentiert Ergebnisse von statistischen Aus-

wertungen. Bei einem solchen Ergebnis handelt es sich immer um einen numerischen Wert, weshalb **AggregatedItem** die Schnittstelle **NumberItem** implementiert. Zusätzlich definiert **AggregatedItem** Metainformationen darüber, auf welche Auswertungsphase (gesamter Beobachtungszeitraum, Zeitintervall, Fahrt oder Betriebszyklus) und welche Teile des MOST-Systems (Gesamtsystem, MOST-Gerät, FBlock, Gerätepaar oder MOST-Link) sich der statistische Wert bezieht.

Ereignisse, die Änderungen an der Kontextinformation beschreiben, sind vom Typ **ContextChangeEvent** oder, um genauer zu sein, von Subtypen dieser Klasse. Damit sind sie bei der Auswertung von anderen Ereignissen zu unterscheiden und können speziell behandelt werden. Bei den Filtern, die statistische Auswertungen durchführen, können die Beobachtungsdaten mit Hilfe dieser Ereignisse je Zeitintervall, Fahrt oder Betriebszyklus aggregiert werden.

6.9 Datenverarbeitung während der Beobachtung

Die Datenverarbeitung während der Beobachtung läuft in zwei nebenläufigen Threads ab:

- Eine Datenquelle läuft dabei im ersten Thread, dem *Aufzeichnungsthread*, ab. Die Aufgabe dieses Threads besteht in der Aufzeichnung des Nachrichtenverkehrs und der Systemereignisse und ihrer Abbildung auf das interne Datenmodell. Nach der erfolgten Abbildung werden die aufgezeichneten Daten im Hauptpuffer abgelegt. Die in diesem Thread durchzuführende Verarbeitung ist für einzelne Nachrichten und Ereignisse vergleichsweise kurz und jeweils ununterbrechbar (die entsprechenden Methoden sind mit dem Java-Schlüsselwort *synchronized* gekennzeichnet).
- Innerhalb des zweiten Threads, dem *Auswertungsthread*, läuft die Auswertung durch die Filterketten und die Benachrichtigung von Monitoringanwendungen über neue Beobachtungsdaten ab. Je nach Konfiguration der Filterketten kann diese Auswertung sehr aufwändig ausfallen und daher pro Ereignis viel Zeit in Anspruch nehmen. Damit auftretende Ereignisse nicht ausgelassen werden müssen, sind die Auswertungsfunktionen bis auf wenige Ausnahmen unterbrechbar.

Das Sequenzdiagramm in Abbildung 18 veranschaulicht den Ablauf der Datenverarbeitung während der Beobachtung.

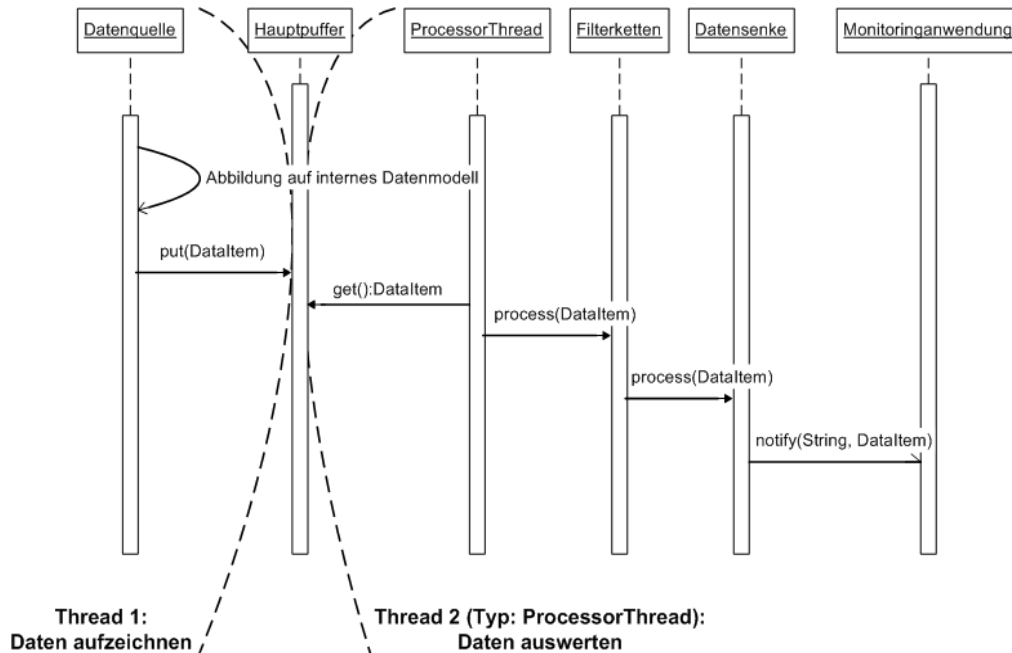


Abbildung 18: Ablauf der Datenverarbeitung während der Beobachtung

6.10 Aufbau der Konfigurationsdatei

Die Konfigurationsdatei der Beobachterkomponente dient zur Festlegung der Puffergröße und zur Konfiguration der Datenquelle. Sie muss sich im Java-Klassenpfad (*Java Classpath*) der Beobachterkomponente befinden. Der Aufbau der Konfigurationsdatei ist identisch mit dem von *.properties-Dateien* unter Java. Die Konfigurationsdatei enthält dabei die folgenden Name-Wert-Paare (*properties*):

Name	Wertebereich	Bedeutung
reader.source	<i>online://</i> oder <i>file://<Pfad in lokalem Dateisystem></i>	Objekt der Beobachtung; <i>online://</i> bezeichnet dabei die Beobachtung eines MOST-Systems während des laufenden Betriebs

buffer.capacity	-1 oder positive Ganzzahlen	Kapazität des Hauptpuffers, -1 bezeichnet unbeschränkte Kapazität
reader.licenseNo	String mit festem Aufbau: XXXX-XXXX-XXXX-XXXX	Lizenznummer für die ActiveX-Komponente
reader.comPort	positive Ganzzahl	Nummer der seriellen Schnittstelle, an welcher der Optolyzer angeschlossen ist
processor.class	Liste qualifizierter Java-Typbezeichner, jeweils durch Semikolon getrennt	Klassen, die als Typen für Bausteine zur Verfügung stehen

Tabelle 6: Einträge in der Konfigurationsdatei

Die Reihenfolge der einzelnen Properties innerhalb der Konfigurationsdatei ist dabei unerheblich.

6.11 Format der zu verarbeitenden Tracedateien

Mit Hilfe der Optolyzer4MOST Software (vgl. Abschnitt 4.3.3) können die auf dem MOST-Bus verschickten Kontrollnachrichten und die aufgetretenen Systemereignisse in *Tracedateien* aufgezeichnet werden. Die Aufzeichnung erfolgt dabei wahlweise im Binär- oder im Textformat, wobei Tracedateien im Binärformat nachträglich ins Textformat konvertiert werden können.

Die Beobachterkomponente ist in der Lage, Tracedateien im Textformat mit einem bestimmten Aufbau zu verarbeiten. Tracedateien im Textformat sind zeilenorientiert aufgebaut, wobei eine Zeile jeweils einer Kontrollnachricht oder einem Systemereignis entspricht. Innerhalb der Optolyzer4MOST Software kann dabei festgelegt werden, welche Informationen für ein Ereignis aufgezeichnet werden. Somit ist der Aufbau einer Zeile in einer Tracedatei

nicht festgelegt.

Da eine Tracedatei keine Metainformationen über ihren Aufbau enthält, wurde ein einheitliches Format festgelegt, in welchem Tracedateien für die Auswertung mit der Beobachterkomponente vorliegen müssen. Dieses Format orientiert sich an den bereits verfügbaren Tracedateien, mit denen das Verhalten von MOST-Systemen in der Praxis aufgezeichnet wurde. Dazu können in der Optolyzer4MOST Software die Felder ausgewählt werden, welche für jedes Ereignis in der Tracedatei enthalten sein sollen. Die einzelnen Felder besitzen dabei relativ zueinander eine feste Reihenfolge.

Das für die Auswertung durch die Beobachterkomponente festgelegte Zeilenformat besitzt den folgenden Aufbau:

Zustand des optischen Signals	Lock- Zustand	Zeit- stempel	Quell- adresse	Ziel- adresse	Nach- richten- typ	Nach- richten- inhalt
----------------------------------------	------------------	------------------	-------------------	------------------	--------------------------	-----------------------------

Die einzelnen Felder innerhalb einer Zeile besitzen dabei eine feste Anzahl von Stellen. Außer zwischen den Feldern, die den *Zustand des optischen Signals* und den *Lock-Zustand* beinhalten, befindet sich zwischen den einzelnen Feldern jeweils ein Leerzeichen. Nach dem Nachrichteninhalte können weitere Felder folgen, so z.B. der Nachrichteninhalte in menschenlesbarer Form (*Disassembly*). Diese zusätzlichen Felder werden bei der Verarbeitung ignoriert.

6.12 Java-COM Bridge

Wie in Abschnitt 6.2.2 erwähnt, benutzt die Beobachterkomponente die Software *Java-COM Bridge* [28], um mit COM-Komponenten zu interagieren. JACOB ist dabei lediglich in der Lage, die generischen Schnittstellen von COM-Komponenten anzusprechen. Damit gestaltet sich die Benutzung ähnlich wie der Aufruf von Methoden über die Reflektionsmechanismen von Java:

```
public void setMostReg(short page, short map, short newValue)
{
    Variant[] argv = new Variant[3];
    argv[0] = new Variant(page);
```

```
    argv[1] = new Variant(map);
    argv[2] = new Variant(newValue);

    wrapperCls.invoke("SetMostReg", argv);
}
```

Listing 2: Interaktion mit einer COM-Komponente über die generische Schnittstelle

Wie man anhand von Listing 2 sehen kann, müssen die (typisierten) Parameter zuerst in ein Array untypisierter Elemente eingepackt werden, bevor dann die entsprechende Operation der COM-Komponente unter Angabe des Parameterarrays und des Namens der Operation aufgerufen werden kann. Diese Art der Interaktion ist – im Vergleich zu lokalen Methodenaufrufen – aufwändig und, da der Compiler keine Überprüfung durchführen kann, fehlerträchtig. Ferner geht bei dieser Art von Aufrufen jegliche Typisierung der Schnittstelle verloren.

Um diese Art von Operationsaufrufen an einer Stelle zu konzentrieren und um der Datenquelle für die Beobachtung während des laufenden Betriebs eine typisierte Schnittstelle für die Interaktion mit der COM-Komponente anbieten zu können, kapseln die Klasse **OptolyzerWrapperImpl** und **OptolyzerEventWrapper** die Interaktion mit der COM-Komponente. Anhang B.6 enthält eine Beschreibung der von diesen Klassen realisierten Schnittstellen **OptolyzerWrapper** und **OptolyzerEventHandler**.

Als MOST-Systeme während des laufenden Betriebs beobachtet wurden, konnte mit Hilfe des Windows Taskmanagers bei der Beobachterkomponente ein beständig anwachsender Speicherbedarf festgestellt werden.

Als Ursache dafür stellte sich ein Speicherleck in der Java-COM Bridge heraus. Das Speicherleck hatte seinen Ursprung in einer Hashtabelle, in der die Parameter für jeden COM-Aufruf und für jede Ereignisbenachrichtigung von der COM-Komponente eingetragen wurden. Diese Einträge blieben auch in der Hashtabelle bestehen, nachdem die Aufrufe abgeschlossen waren und konnten deswegen nicht vom Garbage Collector freigegeben werden, obwohl diese Objekte an keiner anderen Stelle mehr referenziert wurden. Damit sank im Laufe der Zeit auch die Performanz der Hashtabelle und der Beobachterkomponente insgesamt stark ab.

Die Lösung für dieses Speicherleck bestand in der Ersetzung der Implementierung der Hashtabelle durch die von Java bereitgestellte Klasse **WeakHashMap**.

Instanzen von **WeakHashMap** referenzieren die von ihnen verwalteten Objekte durch so genannte *schwache Referenzen* (*Weak References*). Wird auf ein bestehendes Objekt nur noch durch eine schwache Referenz verwiesen, so kann der Garbage Collector von Java dieses Objekt dennoch aufräumen und den von ihm belegten Speicherplatz wieder freigeben.

6.13 Kompatibilität zu J2ME und OSGi

Um die Beobachterkomponente für den Einsatz in eingebetteten Systemen vorzubereiten, wurde bei der Realisierung der Beobachterkomponente gemäß der in Abschnitt 3.6 definierten Anforderungen auf die Kompatibilität zur Micro Edition der Java 2 Platform (J2ME) sowie zum OSGi-Komponentenmodell geachtet.

Die Sicherstellung der Kompatibilität mit J2ME gestaltete sich problemlos, da die verwendete J2ME-Konfiguration CDC (Connected Device Configuration) nahezu alle Klassen bereitstellt, welche die Java 2 Standard Edition in der Version 1.3 auch bietet. Die einzigen Abstriche müssten bei der Unterstützung der Entwicklung von grafischen Bedienoberflächen gemacht werden, da J2ME auch in der umfangreichsten Konfiguration lediglich AWT-Oberflächenkomponenten (*Abstract Window Toolkit*) anbietet und auch davon nur eine Teilmenge. Dieser Teil der Klassenbibliothek war für die Entwicklung der Beobachterkomponente jedoch irrelevant.

Die Kompatibilität wurde überprüft, indem die Beobachterkomponente für die J2ME-VM J9 [21] von IBM kompiliert und auch unter dieser Ablaufumgebung ausgeführt wurde.

Die realisierte Beobachterkomponente wurde für den Einsatz als OSGi-Komponente vorbereitet. Da sich eine Komponente in OSGi im Wesentlichen durch die explizite Angabe der Komponentenschnittstelle(n) und die sich daraus ergebende Trennung zwischen Schnittstelle und Implementierung auszeichnet, kann die Beobachterkomponente mit der beschriebenen Realisierung schon per se als OSGi-Komponente aufgefasst werden.

Die Schnittstelle einer entsprechenden OSGi-Komponente entspricht dabei der Anwendungsschnittstelle der Beobachterkomponente (vgl. Abschnitt 6.7 und Anhang B.1.2).

7 Evaluation

In diesem Kapitel werden die Ergebnisse des Testbetriebs der Beobachterkomponente zusammengefasst. Dazu wird zunächst auf die Erfüllung der funktionalen Anforderungen – die Konfigurierbarkeit und die Eignung für die Beobachtung von MOST-Systemen – eingegangen. Der zweite Teil beschäftigt sich damit, welchen Ressourcenbedarf die prototypische Implementierung im Testbetrieb aufwies.

7.1 Evaluation der Funktionalität

Am Anfang dieser Arbeit entstand eine Liste der zu beobachtenden Systemparameter (vgl. Anhang A). Um die Funktionalität der Beobachterkomponente zu überprüfen, wurden nach und nach alle der aufgelisteten Systemparameter mit Hilfe der Beobachterkomponente überwacht. Es konnte dabei verifiziert werden, dass sich die Beobachterkomponente vergleichsweise einfach anpassen lässt. Daraus kann man schließen, dass das angewendete Architekturmuster „Pipes and Filters“ und die prototypisch realisierten Bausteine geeignet sind, um die Beobachterkomponente an verschiedene Anforderungen bezüglich der zu beobachtenden Systemparameter anzupassen.

Schließlich wurde für Demonstrationszwecke eine Monitoring-Anwendung mit grafischer Oberfläche realisiert, die über die folgende Funktionalität verfügt:

- **Visualisierung der statistischen Informationen über der Zeitachse.** Dazu werden Liniendiagramme verwendet. So können beispielsweise die aktuelle Buslast und ihr zeitlicher Verlauf angezeigt werden.

- **Anzeige von Ereignissen.** Es ist möglich, Ereignisse in Form einer Liste anzuzeigen und diese Ereignisse auch in den oben erwähnten Linien-
diagrammen einzublenden. Letztere Funktion erlaubt die einfache Kor-
relation von einzelnen Ereignissen mit dem zeitlichen Verlauf
ausgewählter Systemparameter.
- **Anzeige der Konfiguration des MOST-Systems und der Belegung
synchroner Kanäle.** Mit der Demoanwendung können sowohl die
Systemkonfiguration (Inhalt der Central Registry) als auch die Bele-
gung der synchronen Kanäle, die zu einem beliebigen Zeitpunkt der
Beobachtung gültig waren, in einem separaten Fenster angezeigt wer-
den.

Die Realisierung der Demoanwendung mit der genannten Funktionalität war dabei relativ kurzfristig zu bewerkstelligen. Der Code für die Interaktion mit der Beobachterkomponente hat – inklusive dem Code für die Konfiguration für die Filterketten – einen Umfang von etwa 50 Codezeilen.

7.2 Speicherbedarf und Performanz der Beobachterkomponente

Um die Performanz der Beobachterkomponente beurteilen zu können und um abzuschätzen, ob der Ressourcenbedarf für einen späteren Einsatz in einem Fahrzeugsteuergerät grundsätzlich angemessen ist, wurden die benötigte Rechenzeit und der Speicherbedarf der Beobachterkomponente gemessen.

Die Messungen wurden sowohl während der Auswertung von Tracedateien als auch während der Beobachtung eines MOST-Systems im laufenden Betrieb (*Onlinebetrieb*) durchgeführt.

Die Tests wurden dabei auf mehreren üblichen Arbeitsplatzrechnern mit x86-Architektur und Microsoft Windows 2000 als Betriebssystem durchgeführt. Als Java VM kam das Java Runtime Environment (*JRE*) von Sun für die Java 2 Standard Edition (*J2SE*) in der Version 1.4.2 zum Einsatz.

Die Monitoring-Anwendung, welche die Beobachterkomponente bei diesen Tests gesteuert und benutzt hat, kam ohne grafische Visualisierung und ohne Verwaltung eines Zustands aus. Auch auf Datei- oder Konsolenausgaben wurde während der Tests verzichtet.

Die für die Tests verwendete Filterkonfiguration war relativ komplex, sie deckte alle in Anhang A aufgelisteten Systemparameter mit Ausnahme der MSC-Beobachtung (Anwendungsverwendung) und der Anzahl von Gruppen- und Broadcastnachrichten ab.

7.2.1 Speicherbedarf

Der Speicherbedarf der Beobachterkomponente wurde überwiegend mit Hilfe des *Windows Taskmanagers* bestimmt, in dem der Speicherbedarf des entsprechenden Java-Prozesses beobachtet wurde. Um die Messergebnisse zu bestätigen, wurde dieselbe Messung zusätzlich mit dem Java-Profiler *JProfiler* [14] durchgeführt.

Die Menge des von der Beobachterkomponente benötigten Speichers ist insgesamt sehr gering; auf den getesteten Plattformen betrug der Speicherbedarf bei der Auswertung von Tracedateien weniger als 3000 Kilobytes, unabhängig von der Charakteristik der verwendeten Tracedatei (auch der Buslast).

Beim Onlinebetrieb wurde auf denselben Plattformen ein ungleich höherer Speicherbedarf festgestellt. Der Speicherbedarf betrug mindestens 20 Megabytes, konnte in der Spitze aber auch bis auf über 30 Megabytes steigen. Auch mit Hilfe von *JProfiler* konnten in dem Java-seitigen Code keine Stellen gefunden werden, von denen der zusätzliche Speicherbedarf herrühren könnte. Daher liegt die Vermutung nahe, dass die Ursache für den zusätzlichen Verbrauch an Speicher in der Verwendung der ActiveX-Komponente liegt, da sich diese zur Laufzeit im selben Prozess befindet. Es wurde dabei aber nicht weiter evaluiert, ob dies an der ActiveX-Komponente selbst, an dem selbst implementierten COM-Server oder der Verwendung des Java Native Interface liegt.

7.2.2 Performanz

Die Performanz wurde ebenfalls mit Hilfe des *Windows Taskmanagers* gemessen. Als Kriterium für die Performanz diente dabei die verbrauchte CPU-Zeit des Prozesses, an den die Beobachterkomponente während des Betriebs gebunden war.

Im Vergleich zum Onlinebetrieb, bei dem lediglich die verbrauchte CPU-Zeit des Gesamtprozesses gemessen werden konnte, gab es während der Verarbeitung von Tracedateien die Möglichkeit, die benötigte CPU-Zeit in Relation zur Buslast auf dem Kontrollkanal und dem von der Tracedatei abgedeckten Zeitrahmen zu setzen. Tracedateien, die während der Fahrt mit dem Versuchsfahrzeug aufgezeichnet wurden, weisen dabei eine durchschnittliche Buslast von circa 100 - 150 Kontrollnachrichten pro Sekunde auf, wohingegen am MOST-Board aufgezeichnete Tracedateien typischerweise durch eine Buslast von deutlich weniger als 50 Kontrollnachrichten pro Sekunde gekennzeichnet sind.

Im Fahrzeug aufgezeichnete Tracedateien konnten auf der Testplattform (einem PentiumIII-Rechner mit 933 MHz Taktfrequenz und 512 Megabyte Arbeitsspeicher) mit der ungefähr zehnfachen Aufzeichnungsgeschwindigkeit ausgewertet werden. Die dabei benötigte CPU-Zeit war sehr gering, sie bewegte sich dauerhaft im einstelligen Prozentbereich.

Trotz der wesentlich geringeren „Ereignisfrequenz“ im Vergleich zur Verarbeitung von Tracedateien, war die benötigte CPU-Zeit im Onlinebetrieb deutlich höher und bewegte sich im mittleren zweistelligen Prozentbereich. Eine weitere Evaluation steht noch aus, allerdings ist auch in diesem Fall davon auszugehen, dass die Interaktion mit der ActiveX-Komponente unter Verwendung des Java Native Interface für diesen Bedarf an Rechenkapazität verantwortlich ist.

Da die Lösung für den Zugriff auf den MOST-Bus beim Betrieb der Beobachterkomponente in einem Fahrzeugsteuergerät aber mit sehr hoher Wahrscheinlichkeit anders aussehen dürfte, ist die hohe CPU-Belastung beim Onlinebetrieb kein Aspekt, dessen Untersuchung im Rahmen dieser Arbeit statt fand.

Zu erwähnen bleibt noch, dass die gemessenen Werte für die Performanz praktisch unabhängig von der verwendeten Konfiguration der Filterketten waren. Bei der Verwendung einer „Minimalkonfiguration“, bei der nur die Buslast gemessen wurde, waren zumindest mit Hilfe des Windows Taskmanagers keine Unterschiede in der Performanz feststellbar.

8 Resümee

In diesem Kapitel werden noch einmal die Ergebnisse dieser Arbeit zusammengefasst. Zum Abschluss folgt eine kurze Beschreibung möglicher Erweiterungen der realisierten Beobachterkomponente.

8.1 Zusammenfassung

Gegenstand dieser Arbeit ist die Entwicklung einer Beobachterkomponente für ein verteiltes Fahrzeugtelematiksystem.

Dazu wurden zunächst die Besonderheiten moderner, verteilter Fahrzeugtelematiksysteme betrachtet. Sie zeichnen sich durch einen großen Funktionsreichtum und eine hohe Leistungsfähigkeit aus, besitzen allerdings auch eine große und wachsende Komplexität. Sofern keine besonderen Vorkehrungen getroffen werden, sind sie daher anfällig für Fehler, was dem Qualitätsanspruch der Automobilindustrie und der Fahrzeugkäufer zuwiderläuft.

Die Technologie Media Oriented Systems Transport (MOST) dient als Basistechnologie für die Realisierung von Fahrzeugtelematiksystemen und soll helfen, deren Komplexität zu reduzieren. Dieses Ziel kann die MOST-Technologie jedoch nicht vollständig erreichen. Sie zeichnet sich stattdessen an einigen Stellen durch hohe Komplexität und Inkonsistenz aus.

Da die Komplexität von Fahrzeugtelematiksystemen in näherer Zukunft wohl bestehen bleiben wird, muss sich die Automobilindustrie damit arrangieren. Eine Möglichkeit dazu zeigen die Autonomic Computing-Initiative von IBM und thematisch verwandte Forschungsprojekte auf. Da wegen der großen Komplexität der Telematiksysteme, ihrer Konfigurationsvielfalt und der Vielzahl von Fahrzeugen keine hundertprozentige Fehlerfreiheit gewährleistet werden kann, sollen autonome Systeme fehlertolerant arbeiten, indem sie sich

im Fehlerfall selbst reparieren.

Diese Arbeit fand ebenfalls im Rahmen eines Forschungsprojekts statt, bei dem die Anwendbarkeit von Ideen des Autonomic Computing auf Fahrzeugtelematiksysteme untersucht werden.

Der erste Schritt hin zu autonomen Systemen besteht darin, dass sich die entsprechenden Systeme selbst beobachten. Dazu wurde untersucht, welche Anforderungen an eine Beobachterkomponente in einem MOST-basierten Fahrzeugtelematiksystem bestehen und ob bestehende Monitoringlösungen diesen Anforderungen genügen. Die Beobachterkomponente soll konfigurierbar und erweiterbar sein, sowie in der Lage, MOST-spezifische Systemparameter zu beobachten und auszuwerten. Dabei wurde festgestellt, dass bestehende Lösungen sehr viel eher für die Beobachtung großer und dynamischer Systeme konzipiert wurden, die durch verteilte Sensoren beobachtet werden. Ferner sind sie nicht dafür ausgelegt, MOST-spezifische Systemparameter zu beobachten. Die für die MOST-Technologie verfügbaren Lösungen können zwar die MOST-spezifischen Systemparameter beobachten, allerdings besitzen sie nur sehr eingeschränkte Auswertungsmöglichkeiten und sie sind nicht ausreichend erweiterbar.

Aus diesen Gründen wurde in dieser Arbeit eine Architektur für eine Beobachterkomponente vorgeschlagen, die dafür ausgelegt ist, das System sowohl auf der Ebene der Netzwerkschicht als auch auf der Anwendungsebene zu beobachten.

Im Mittelpunkt der vorgeschlagenen Architektur steht die Verwendung des Architekturmusters „Pipes and Filters“ in einer erweiterten Fassung. Die Funktionen zur Auswertung der aufgezeichneten Nachrichten und Systemereignisse werden dabei aus einzelnen Bausteinen zusammengesetzt. Durch eine geeignete Auswahl der zur Verfügung stehenden Bausteine und die Gewährleistung, dass sich die Bausteine vielfältig kombinieren lassen, kann mit dieser Architektur die Mehrzahl interessanter Systemparameter beobachtet werden. Durch die Verwendung des Bausteinprinzips ist die vorgeschlagene Architektur vielfältig konfigurierbar- und nicht zuletzt erweiterbar.

Um die im Rahmen dieser Arbeit entstandenen Konzepte zu bewerten, wurde die Architektur prototypisch implementiert. Die Evaluierung bestand in der Durchführung verschiedener Beobachtungsszenarien, bei denen verschiedenste Systemparameter ausgewertet wurden. Dabei konnte gezeigt werden, dass die Architektur den unterschiedlichen Anforderungen gewachsen ist.

8.2 Ausblick

Im Rahmen dieser Arbeit konnten nicht alle Bausteine realisiert werden, die nötig wären, um alle interessanten Systemparameter zu beobachten.

Besonders bei der Beobachtung von Systemparametern auf der Anwendungsebene besteht noch Handlungsbedarf. So wurde bislang kein Baustein realisiert, der Nachrichten über den Zustand der verschiedenen Fahrzeugfunktionen auswertet. Diese Nachrichten werden üblicherweise über andere Bussysteme im Fahrzeug (CAN-Nachrichten) versendet. Ein Teil dieser Nachrichten wird über das Gateway-Gerät auf den MOST-Bus weitergeleitet und könnte daher direkt von der Beobachterkomponente ausgewertet werden. Andere Nachrichten, die nicht auf den MOST-Bus weitergeleitet werden, müssten der Beobachterkomponente durch das Abhören der verschiedenen CAN-Busse zugänglich gemacht werden.

Durch die Auswertung dieser Nachrichten könnten wertvolle Informationen gewonnen werden, die Aufschluss über die Ursachen der Startup- und Shutdownvorgänge des Telematiksystems geben. Diese Nachrichten umfassen beispielsweise die folgenden Informationen:

- Stellung des Zündschlosses
- Zustand der Fahrzeurtüren (offen/geschlossen)
- Motorstarts und -stopps

Von Interesse wäre auch die allgemeine Beobachtung von Kommunikationsabläufen der Telematikanwendungen. Dadurch könnte die Ausführung von Anwendungsfällen mit dem Systemverhalten korreliert werden. Viele dieser Abläufe sind durch Message Sequence Charts spezifiziert. Allerdings sind nicht alle Abläufe spezifiziert und die spezifizierten Abläufe überschneiden sich teilweise, was die Auswertung erschwert. Bei der schrittweisen Verfolgung von Abläufen muss der Zustand verwaltet werden, in dem sich der jeweilige Ablauf befindet. Dabei stellt sich die Frage, ob ein einzelner Baustein die richtige Granularität bietet, um so einen Zustand geschlossen zu verwalten oder ob für die Beobachtung von Abläufen das Zusammenspiel mehrerer Bausteine sinnvoller wäre. Im letzteren Fall wäre zu klären, inwiefern sich das mit der vorgeschlagenen Architektur verträgt und welche Erweiterungen dafür nötig wären.

Eine Erweiterungsmöglichkeit in ganz anderer Richtung besteht in der Vereinfachung der Filterkonfiguration. Die bisherige Realisierung ermöglicht zwar die Nutzung aller Freiheitsgrade bei der Konfiguration der Filterketten, ist bei der Benutzung allerdings auch fehlerträchtig und aufwändig. Eine mögliche Lösung würde in der Realisierung eines grafischen Editors bestehen, in dem die Filterketten konfiguriert werden könnten. Ein solcher Editor könnte auch die Korrektheit einer Filterkonfiguration überprüfen. Da eine Filterkonfiguration als ein Graph aufgefasst werden kann, müsste der Graph beispielsweise auf Zyklensfreiheit oder auf seinen Zusammenhang geprüft werden.

Anhang

A Beobachtete Systemparameter

In den folgenden Tabellen sind alle im Rahmen dieser Arbeit beobachteten Systemparameter aufgeführt.

Beobachteter Systemparameter	Ebene (Netzwerk/Anwendung/Fahrzeug)	In Relation zu	Aussagekraft und Anmerkungen
Aktuelle MOST-Konfiguration	Netzwerk und Anwendung	-	Kann auf Fehler bei Systemkonfiguration hinweisen z.B., dass bestimmte Anwendungen oder Geräte nicht verfügbar sind
Zustand der aktuellen MOST-Konfiguration	Netzwerk und Anwendung	-	Kann darauf hinweisen, dass die Initialisierung des Gesamtsystem fehlgeschlagen und die Konfiguration unvollständig ist

Tabelle 7: Aus der Beobachtung gewonnene Konfigurationsinformationen

Die folgende Tabelle enthält eine Beschreibung aller statistischen Informationen, die bei der Beobachtung erhoben wurden. Die statistische Auswertung beschränkt sich dabei auf Zählungen, wie häufig bestimmte Nachrichten oder Systemereignisse aufgetreten sind. Gezählt werden kann über die Gesamtlebensdauer des Telematiksystems, über eine einzelne Fahrt oder über definierte Zeitintervalle. Letzteres erlaubt die zeitliche Korrelation von Ereignissen.

Beobachteter Systemparameter	Ebene (Netzwerk/ Anwendung/ Fahrzeug)	In Relation zu	Aussagekraft
MOST Systemereignisse			
Startvorgang inkl. Ursache	Netzwerk	insgesamt	Kann auf Fehler bei Systemstart oder auf unbeabsichtigten Start hinweisen; Referenz für weitere Zähler
Konfigurationsänderungen zählen, nach Typ (OK, NotOK, ...) unterschieden	Netzwerk und Anwendung	insgesamt	Kann auf Probleme in der Startphase oder auf Resets des MOST-Systems hinweisen
MOST Systemereignisse zählen (Lock, Unlock, stabiles Lock, kritisches Unlock)	Netzwerk	insgesamt	Eine hohe Anzahl oder eine zeitliche Korrelation von Ereignissen zeigt ein Problem des Telematiksystems an
Fehlersituationen			
Fehlermeldungen (OpType Error) zählen	Netzwerk und Anwendung	insgesamt	Eine hohe Anzahl von Fehlermeldungen zeigt ein Problem des Telematiksystems an
	Netzwerk und Anwendung	pro Gerät	Softwareproblem eines bestimmten Geräts
	Anwendung	pro FBlock	Softwareproblem einer bestimmten Anwendung

Beobachteter Systemparameter	Ebene (Netzwerk/Anwendung/Fahrzeug)	In Relation zu	Aussagekraft
	Netzwerk und Anwendung	pro Gerätepaar	Kommunikationsschwierigkeiten auf Anwendungsebene, mindestens eines der Geräte weicht vermutlich von Spezifikation ab
Timeouts bei Funktionsaufrufen	Anwendung	insgesamt	Überlastung des Gesamtsystems
	Anwendung	pro Gerät	Überlastung oder Problem eines bestimmten Geräts
	Anwendung	pro FBlock	Überlastung oder Problem einer bestimmten Anwendung
Netzwerkauslastung			
Anzahl von Kontrollnachrichten	Netzwerk	insgesamt	Netzwerkbelastung insgesamt
	Netzwerk	pro Gerät	Netzwerkbelastung durch ein bestimmtes Gerät
	Netzwerk	pro FBlock	Netzwerkbelastung durch eine bestimmte Anwendung
	Netzwerk	pro Gerätepaar	Netzwerklast durch Interaktion zweier Geräte
Anzahl von Gruppen- und Broadcastnachrichten	Netzwerk	insgesamt	Hoher Aufwand für die Systemverwaltung und Gesamtkommunikationscharakteristik
Anzahl von bestimmten Nachrichtentypen (Unterscheidung nach OpType, z.B. Get, Set, Status oder Error)	Anwendung	insgesamt	Gesamtkommunikationscharakteristik
	Anwendung	pro Gerät	Kommunikationscharakteristik eines Geräts
	Anwendung	pro FBlock	Kommunikationscharakteristik einer Anwendung
	Anwendung	pro Gerätepaar	Charakteristik der Kommunikation zwischen zwei Geräten
Anzahl von Low-Level Retries	Netzwerk	insgesamt	Überlastung oder Probleme des Netzwerks insgesamt

Beobachteter Systemparameter	Ebene (Netzwerk/ Anwendung/ Fahrzeug)	In Relation zu	Aussagekraft
	Netzwerk	pro Gerät	Überlastung oder Problem eines bestimmten Geräts
	Netzwerk	pro FBlock	Überlastung einer bestimmten Anwendung
	Netzwerk	pro Netzwerksegment (MOST-Link)	Übertragungsprobleme auf einem bestimmten MOST-Link
Anwendungsverhalten			
Verfügbarkeit von Funktionsblöcken	Netzwerk und Anwendung	insgesamt	Überlastung oder Problem des gesamten Netzwerks
Antwortzeit auf Kontrollnachricht	Netzwerk und Anwendung	insgesamt	Überlastung oder Problem des gesamten Netzwerks
	Netzwerk und Anwendung	pro Gerät	Überlastung oder Problem eines bestimmten Geräts
	Netzwerk und Anwendung	pro FBlock	Überlastung einer bestimmten Anwendung
	Netzwerk und Anwendung	pro Gerätepaar	Kommunikationsprobleme zwischen den Anwendungen zweier Geräte

Tabelle 8: Statistische Informationen über das Systemverhalten

Beobachteter Systemparameter	Ebene (Netzwerk/Anwendung/Fahrzeug)	Umgebungsdaten	Anmerkungen
Loggen von Fehler- nachrichten	Anwendung	Zeitstempel, Gerät, FBlock, Fehlertyp (Timeout, CRC, ...)	
Loggen von MOST- Systemereignissen	Netzwerk	Zeitstempel	
Loggen von Fahr- zeugereignissen	Anwendung und Fahr- zeug	Zeitstempel, Typ	
Loggen von Span- nungsänderungen	Fahrzeug	Zeitstempel, Gerät	Konnte in den verwendeten Ver- suchsumgebungen nicht beobach- tet werden
Konfigurationsände- rung	Netzwerk und Anwen- dung	Zeitstempel, Inhalt der Central- Registry	
Anwendungsver- wendung (MSC- Beobachtung)	Anwendung		Hierbei werden bestimmte MOST- Kontrollbefehle und –befehlsfolgen bestimmten Anwendungs- ereignissen zugeordnet; wurde in dieser Arbeit nicht realisiert

Tabelle 9: Beobachtete Systemereignisse (Ereignislog)

B Schnittstellendefinitionen

Dieser Abschnitt enthält die Definition der wesentlichen Schnittstellen der Beobachterkomponente. Die Unterteilung erfolgt gemäß der Java-Packages, wobei jedes Package eine Teilkomponente oder einen Teilaspekt der Beobachterkomponente abdeckt.

B.1 Package com.dcx.most.monitoring.app

Dieses Java-Package enthält die Schnittstellen- und Klassendefinitionen, welche die Anwendungsschnittstelle bilden.

B.1.1 public interface Controller

Callback-Schnittstelle für die Benachrichtigung einer Anwendung über neue Beobachtungsdaten.

public notify(String dataSeriesID, DataItem item)

Wird zur Benachrichtigung einer Anwendung über ein neu verfügbares Datum aufgerufen. **item** enthält dabei die neu verfügbare Dateneinheit.

public getRegistrationIDs():List

Über diese Operation kann eine Realisierung der Callback-Schnittstelle abgefragt werden, für den Bezug welcher Daten sie sich registriert hat.

public reset()

Setzt den (optional verwalteten) Zustand einer Realisierung der Callback-Schnittstelle zurück.

B.1.2 public abstract class MonitoringFacade

Definition der Anwendungsschnittstelle

public createProcessor(String type, String processorId):

Erstellt eine Instanz des angegebenen Bausteintyps mit dem angegebenen Bezeichner **processorId**.

public connectProcessors(String predecessorId, String successorId, boolean attachToPositiveOutput):

Verbindet die beiden angegebenen Bausteininstanzen.

public getAvailProcessorIDs():String[]:

Gibt eine Auflistung der Bezeichner aller Bausteine zurück.

```
public getAvailProcessorTypes () :String[]:
```

Gibt eine Auflistung der verfügbaren Bausteintypen zurück. Jeder Typ wird durch den qualifizierten Namen der entsprechenden Java-Klasse repräsentiert.

```
public getProcessorType (String processorId) :String:
```

Liefert den Typ eines Bausteins als qualifizierten Java-Typnamen zurück.

```
public getProcessorProperties (String type) :String[]:
```

Liefert die Namen aller konfigurier- und abfragbaren Eigenschaften eines Bausteintyps zurück. Der Typname wird als qualifizierter Java-Typbezeichner angegeben.

```
public getPropertyValue (String processorId, String  
propertyName) :String:
```

Gibt den Wert der angegebenen Eigenschaft der Bausteininstanz mit dem Bezeichner `processorId` zurück.

```
public setPropertyValue (String processorId, String  
propertyName, String value):
```

Legt den Wert der angegebenen Eigenschaft der Bausteininstanz mit dem Bezeichner `processorId` fest.

```
public getBaseProcessorID () :String:
```

Gibt den Bezeichner desjenigen Filters zurück, der den Einsprungpunkt in die Filterkonfiguration darstellt. Dieser Filter ist der Vorgängerbaustein aller konfigurierten Filterketten. Der Filter ist vom Typ `BasicFilter` und wird von der Konfigurationskomponente bereits fest vorkonfiguriert.

```
public String[] getSuccessors (String processorId):
```

Liefert eine Auflistung der Bezeichner aller Nachfolgerbausteine des angegebenen Bausteins zurück.

```
public configureDataSource ():
```

Liest die Konfigurationsdatei (erneut) ein und konfiguriert Datenquelle und Puffer gemäß den Angaben in der Konfigurationsdatei.

```
public abstract void addProcessingCompleteHandler (  
ProcessingCompleteHandler handler):
```

Registriert die angegebene Callback-Schnittstelle, um eine Anwendung über das Ende der Beobachtung zu informieren.

```
public abstract void removeProcessingCompleteHandler (  
ProcessingCompleteHandler handler):
```

Deregistriert die angegebene Callback-Schnittstelle zur Benachrichtigung über das Beobachtungsende.

```
public start ():
```

Startet die Beobachtung. Die einzelnen Teilkomponenten (Datenquelle, Puffer und Filterkonfiguration) müssen vor dem Aufruf dieser Methode konfiguriert worden sein.

public stop(): Stoppt die Beobachtung.

public abstract Collection getDataSeriesIDs():

Liefert die Menge der Bezeichner aller verfügbaren Arten von Beobachtungsdaten zurück.

public abstract void registerController(Controller controller, String dataSeriesID):

Registriert die angegebene Callback-Schnittstelle zur Benachrichtigung über neue Beobachtungsdaten für Daten des angegebenen Typs.

public abstract void removeController(Controller controller, String dataSeriesID):

Deregistriert die angegebene Callback-Schnittstelle für die Benachrichtigung über neue Beobachtungsdaten für Daten des angegebenen Typs.

public abstract void performWait():

Wird von Anwendung zum Warten auf neue Beobachtungsdaten aufgerufen.

public abstract TimeStamp getFirstTS():

Liefert den Zeitstempel des während der Beobachtung zuerst aufgetretenen Ereignisses zurück.

public abstract TimeStamp getLastTS():

Liefert den Zeitstempel des während der Beobachtung zuletzt aufgetretenen Ereignisses zurück.

public abstract MOSTConfig getConfigInfo(int configID):

Liefert die Beschreibung der MOST-Konfiguration mit dem angegebenen Bezeichner zurück.

public abstract MOSTConfig getConfigInfo(TimeStamp ts):

Liefert die Beschreibung der MOST-Konfiguration zurück, die zu dem angegebenen Zeitpunkt gültig war.

public abstract MOSTConfig getCurrentConfigInfo():

Liefert die Beschreibung der aktuell gültigen MOST-Konfiguration zurück.

public abstract short getCurrentMaxNodePos():

Liefert Position des aktuell letzten Geräts .im Ring zurück. Dieser Wert entspricht der Geräteanzahl – 1.

public abstract MOSTConfig getMaxConfig():

Liefert Maximalkonfiguration zurück.

public abstract List getMissingFBlocks(MOSTConfig config):

Liefert Liste mit FBlocks zurück, die in der angegebenen Konfiguration gegenüber der Maximalkonfiguration fehlen.

public abstract ChannelAllocationEntry[] getAllocTable(TimeStamp ts):

Liefert den Inhalt der Tabelle mit der Belegung der synchronen Kanäle zurück.

```
public abstract int getTripCount():
```

Liefert den aktuellen Wert des Fahrtenzählers zurück.

```
public abstract CycleInfo getCurrentCycle():
```

Liefert Informationen über den aktuellen Betriebszyklus zurück.

```
public abstract SortedSet getCycleHistory():
```

Liefert eine Historie der zurückliegenden Betriebszyklen zurück.

```
public abstract boolean[] getNotificationsForDevice(short  
fBlockID, short instID, int deviceID):
```

Liefert die Tabelle von Funktionen zurück, bei denen sich ein Gerät für Notifikationsnachrichten registriert hat.

```
public abstract int getCurrentMileage():
```

Liefert den aktuellen Kilometerstand zurück.

```
public abstract Collection getDataSinks():
```

Liefert die Menge von Referenzen auf die konfigurierten Datensinken zurück.

B.1.3 public interface ProcessingCompleteHandler

Callback-Schnittstelle für die Benachrichtigung über die Beendigung von Beobachtungsvorgängen.

```
public void processingComplete():
```

Callback-Methode, die aufgerufen wird, wenn ein Beobachtungsvorgang abgeschlossen ist.

B.2 Package com.dcx.most.monitoring.capture

Enthält die Schnittstellen des Hauptpuffers und der Datenquellen

B.2.1 public interface Buffer

Schnittstellendefinition für Hauptpuffer mit FIFO-Eigenschaft.

```
public int getMsgCount():
```

Liefert die Anzahl aller put-Aufrufe zurück.

```
public int getDroppedMsgCount():
```

Liefert die Anzahl abgewiesener Ereignisse zurück (Puffer war voll).

```
public boolean put(DataItem item):
```

Stellt das angegebene Datum in den Puffer, sofern dieser nicht voll ist.

public DataItem get():

Returns the least recent buffer entry and then removes it from the buffer (FIFO buffer).

public int getCapacity():

Liefert die aktuelle Kapazität des Puffers zurück..

public void setCapacity(int newCapacity):

Setzt die Pufferkapazität auf den angegebenen Wert.

public boolean isEmpty():

Gibt Auskunft darüber, ob der Puffer leer ist.

public void performWait():

Ein Aufruf dieser Operation legt den aufrufenden Thread in die Warteschlange des Pufferobjekts.

B.2.2 public abstract class MOSTReader

Allgemeine Schnittstellendefinition für Datenquellen.

start()

Aktiviert die Datenquelle, um mit der Überwachung der Systemparameter zu beginnen.

stop()

Deaktiviert die Datenquelle.

setBuffer(Buffer buffer)

Setzt die Referenz auf den Hauptpuffer. Die Datenquelle nutzt diese Referenz, um auf der dahinterliegenden Instanz die **put**-Operation aufzurufen.

setSource(String sourceURL)

Setzt die Lokation der Ereignisquelle. **sourceURL** kann dabei auf den MOST-Bus (`online://`) oder auf eine Tracedatei in lokalen Dateisystemen (`file://...`) verweisen.

setLicenceNumber(String licenseNo)

Legt die Lizenznummer für die verwendete OptoControl ActiveX-Komponente fest.

setComPort(short comPort)

Legt die Nummer der seriellen Schnittstelle (COM Port) fest, an welcher der Optolyzer angeschlossen ist.

getFirstEventTS()

Liefert den Zeitstempel des während der Beobachtung zuerst aufgetretenen Ereignisses zurück.

getLastEventTS ()

Liefert den Zeitstempel des während der Beobachtung zuletzt aufgetretenen Ereignisses zurück.

public String getLicenseNumber ()

Liefert die aktuell gesetzte OptoControl-Lizenznummer zurück.

public short getComPort ()

Liefert die Nummer des COM Ports zurück, an den der Optolyzer angeschlossen ist.

B.3 Package com.dcx.most.monitoring.datasinks

Enthält die Schnittstellen und die Implementierung der Datensinken

B.3.1 public interface DataSink

Schnittstellendefinition einer zustandslosen Datensenke.

public addDataID (String ID)

Fügt einen Bezeichner für Dateneinheiten zu der Menge von Bezeichnern verfügbarer Daten dieser Datensenke hinzu. Diese Operation wird von einem Filter aus derjenigen Filterkette aufgerufen, die durch diese Datensenke terminiert wird.

public removeDataID (String ID)

Entfernt den angegebenen Bezeichner aus der Menge von Bezeichnern für verfügbare Daten. Auch diese Operation wird von einem Filter aus der Filterkette aufgerufen, die durch diese Datensenke terminiert wird.

public getAvailableIDs () :Collection

Liefert eine Auflistung aller an dieser Datensenke verfügbaren Bezeichner für Dateneinheiten zurück.

public abstract boolean registerController (Controller controller, String dataSeriesID)

Registriert die angegebene Callback-Schnittstelle zur Benachrichtigung über neue Beobachtungsdaten für Daten des angegebenen Typs.

public abstract boolean removeController (Controller controller, String dataSeriesID)

Deregistriert die angegebene Callback-Schnittstelle für die Benachrichtigung über neue Beobachtungsdaten für Daten des angegebenen Typs.

B.3.2 public interface StatefulDataSink

Schnittstellendefinition einer Datensenke, die einen Zustand verwaltet. Legt das Abfrage-API fest.

```
public getDataByID(String ID):List
```

Liefert eine Liste aller Dateneinheiten zurück, die **ID** als Bezeichner haben.

```
public getDataByID(String ID, int count):List
```

Liefert die **count** aktuellsten Dateneinheiten mit dem gegebenen Bezeichner zurück.

```
public abstract List getDataByID(String ID, byte diffType,  
long value)
```

Liefert alle Daten einer Datenreihe für eine bestimmte Entität des MOST-Systems zurück. Die Art der Entität wird durch **diffType** angegeben, die konkrete Instanz durch **value**.

```
public abstract List getDataByID(String ID, byte aggrBase,  
int value)
```

Liefert alle Daten einer Datenreihe für die angegebene Beobachtungsphase (Zeitintervall, Fahrt, Betriebszyklus) zurück.

```
public getEvents():List
```

Liefert alle an der Datensenke verfügbaren Einzelereignisse zurück.

```
public getEvents(int count):List
```

Gibt die **count** aktuellsten an dieser Datensenke verfügbaren Einzelereignisse zurück.

```
public getStatistics():List
```

Liefert alle an der Datensenke verfügbaren statistischen Daten zurück.

```
public getStatistics(byte aggrBase):List
```

Liefert alle an der Datensenke verfügbaren statistischen Daten zurück, die relativ zu dem durch **aggrBase** definierten Beobachtungsintervall erhoben wurden.

B.4 Package com.dcx.most.monitoring.filters

Enthält die Schnittstellen und die Realisierung der verschiedenen Bausteine für die Auswertung der Daten.

B.4.1 public abstract class AggregatingFilter

implements Filter

Erweiterte Schnittstelle und Implementierung für Filterbausteine, die statistische Auswertungen durchführen.

```
public byte getAggrBase()
```

Liefert zurück, ob sich die statistischen Auswertungen auf den gesamten Beobachtungszeitraum, eine einzelne Fahrt, einen einzelnen Betriebszyklus des Telematiksystems oder ein Zeitintervall beziehen.

```
public byte getDifferentiateBy()
```

Liefert zurück, ob sich die statistische Auswertung auf das Gesamtsystem, ein einzelnes Gerät, einen einzelnen Funktionsblock, ein Paar von MOST-Geräten oder einen MOST-Netzwerkabschnitt beziehen soll.

```
public setAggrBase(byte base)
```

Der Wert von **base** legt fest, ob sich die statistischen Auswertungen auf den gesamten Beobachtungszeitraum, eine einzelne Fahrt, einen einzelnen Betriebszyklus des Telematiksystems oder ein Zeitintervall beziehen soll.

```
public setDifferentiateBy(byte discriminator)
```

Legt fest, ob sich die statistische Auswertung auf das Gesamtsystem, ein einzelnes Gerät, einen einzelnen Funktionsblock, ein Paar von MOST-Geräten oder einen MOST-Netzwerkabschnitt beziehen soll.

B.4.2 public interface Filter

extends Processor

Schnittstellendefinition für Filterbausteine.

```
public addSuccessor(Processor successor, boolean atPosOut)
```

Fügt einen Baustein an den durch das Flag **atPosOut** angegebenen Ausgang an.

```
public setNegativeID(String negativeID)
```

Legt den Bezeichner des negativen Ausgangs fest.

```
public getNegativeID():String
```

Liefert den Bezeichner für den negativen Ausgang zurück.

```
public getSuccessors():Collection
```

Liefert die Menge aller Nachfolgerbausteine in einer Instanz der Java-Sammelklasse `Collection` zurück.

```
public removeSuccessor(removeSuccessorString ID)
```

Entfernt einen Nachfolgerbaustein mit dem gegebenen Bezeichner. Falls es sich bei dem entsprechenden Nachfolgerbaustein um einen Filter handelt, so bezieht es sich auf den Bezeichner des positiven Filterausgangs.

B.4.3 public interface Processor

Schnittstellendefinition für die einfachsten Bausteine (Prozessoren).

public open()

Aktiviert die Prozessorinstanz, damit kann sie an ihrem Eingang Dateneinheiten entgegennehmen.

public close()

Deaktiviert die Prozessorinstanz.

public reset()

Kann bei einer Prozessorinstanz nur in aktivem Zustand aufgerufen werden. Führt zunächst eine Deaktivierung und anschließend eine Aktivierung durch.

public setID(String ID)

Legt den Bezeichner einer Prozessorinstanz fest. Diese Methode prüft nicht die Eindeutigkeit des Bezeichners. Die Verantwortung dafür hat die Konfigurationskomponente (vgl. Abschnitt 6.6), welche die Filterkonfiguration kapselt.

public getID():String

Liefert den aktuellen Bezeichner einer Prozessorinstanz zurück.

public equals(Object obj):boolean

Vergleicht zwei Prozessorinstanzen auf Gleichheit.

public process(DataItem item)

Nimmt eine Dateneinheit zur weiteren Verarbeitung entgegen.

B.5 Package com.dcx.most.monitoring.impl

Dieses Package enthält die Implementierung der Konfigurationskomponente, der Datenquellen und des Hauptpuffers.

B.5.1 public abstract class ConfigurationManager

implements MonitoringFacade

Schnittstelle der Konfigurationskomponente. Da ConfigurationManager die Schnittstellendefinition MonitoringFacade (vgl. B.1.2) implementiert, sind an dieser Stelle nur die zusätzlichen Operationen aufgeführt.

public getBuffer():Buffer

Liefert eine Referenz auf den Hauptpuffer zurück. Diese Operation ist für den internen Gebrauch, etwa durch eine Datenquelle, gedacht

synchronized public void processingComplete()

Callback-Methode, die aufgerufen wird, wenn die Filterketten die Auswertung abgeschlossen haben und der Hauptpuffer leer ist..

B.6 Package *com.dcx.most.optocontrol*

Enthält die Schnittstellen, welche die Interaktion mit der OptoControl-Komponente kapseln.

B.6.1 public interface `OptolyzerEventHandler`

Eventhandler-Interface fuer die Ereignisse des Optolyzer COM-Wrappers.

```
public void OnSpyMessage(int srcAdr, int tgtAdr, short  
msgType, String msgData, String spyData, short status, int  
timeStamp)
```

Wird aufgerufen, wenn eine Kontrollnachricht im Spymodus eintrifft.

```
public void OnReceiveMessage(int srcAdr, int tgtAdr, short  
msgType, String msgData, short status, int timeStamp)
```

Wird aufgerufen, wenn eine Kontrollnachricht im Master- oder Slavemodus eintrifft.

```
public void OnUndefined(short cmd)
```

Wird aufgerufen, wenn ein Property gesetzt wird, das der Optolyzer nicht unterstützt.

```
public void OnChangeOptoMode(short newOptoMode)
```

Wird aufgerufen, wenn der Gerätemodus des Optolyzers geändert wird (z.B. Master -> Spy).

```
public void OnChangeOwnAdr(short nodePos, short deviceAdr,  
short groupAdr)
```

Wird aufgerufen, wenn die Geräte oder Gruppenadresse des Optolyzers geändert wurde.

```
public void OnBusLoadStop()
```

Wird aufgerufen, wenn die Generierung von Last auf dem MOST-Bus beendet wurde.

```
public void OnChangeAllocScanMode(short newScanMode)
```

Wird aufgerufen, wenn die Beobachtung der Allokation synchroner Kanäle an- oder abgeschaltet wurde.

```
public void OnChangeAllocTable(short map, String data)
```

Wird aufgerufen, wenn die ein synchroner Kanal (de-)allokiert wurde.

```
public void OnChangeLightLock(short status, int timeStamp)
```

Wird aufgerufen, wenn sich am Licht- oder Lockzustand des MOST-Rings etwas geändert hat.

```
public void OnChangeMPR(short mprValue)
```

Wird aufgerufen, wenn sich der Wert des MPR-Registers geändert hat.

public void OnChangeSpyTransMode(short newTransMode)

Wird aufgerufen, der Modus zur Übertragung von Spynachrichten zwischen Optolyzer und serieller Schnittstelle geändert wurde.

public void OnRemoteResult(short handle, short msgResult, short type, short map, String data)

Wird aufgerufen, wenn das Ergebnis einer Remote-Zugriffsnachricht empfangen wurde.

public void OnSendResult(short handle, short msgResult):

Wird aufgerufen, wenn das Ergebnis einer 'normalen' Kontrollnachricht empfangen wurde.

public void OnSpyInfo(short bufferCount, short msgCountSinceLast)

Wird aufgerufen, wenn eine Nachricht vom Optolyzers bezueglich des Spy-Puffers empfangen wurde.

public void OnStressStop()

Wird aufgerufen, wenn die Generierung von maximaler Last auf dem MOST-Bus beendet wurde.

public void OnChangeRegs(short page, short map, String regData)

Wird aufgerufen, wenn sich der Inhalt eines MOST-Transceiver-Registers geändert hat.

B.6.2 public interface OptolyzerWrapper

Schnittstelle, welche die Interaktion mit der OptoControl-Komponente kapselt.

public Collection getEventHandlers()

public void registerEventHandler(OptolyzerEventHandler handler)

public void releaseCOMPtr()

Gibt die Referenz auf die OptoControl-Komponente frei. Wenn keine weiteren Referenzen vorhanden sind, kann sie aus dem Speicher entfernt werden.

public short getComPort()

Liefert die Portnummer der seriellen Schnittstelle zurück, an welcher der Optolyzer angeschlossen ist.

public void setComPort(short newComPort)

Legt die Portnummer der seriellen Schnittstelle fest, an welcher der Optolyzer angeschlossen ist.

```
public String getLicenseNumber()
```

Liefert die aktuell benutzte Lizenznummer für die OptoControl-Komponente zurück.

```
public void setLicenseNumber(String newLicNo)
```

Legt die Lizenznummer für die OptoControl-Komponente fest.

```
public void open()
```

Öffnet die COM-Schnittstelle zum Optolyzer für den exklusiven Zugriff durch diesen Prozess.

```
public void close()
```

Schließt den COM Port zum Optolyzer. Komplementäre Operation zu open().

```
public short getDeviceAdr()
```

Liefert die logische Adresse des angeschlossenen Optolyzers zurück.

```
public short setOwnAdr(int deviceAdr, short groupAdr)
```

Legt die logische Geräteadresse und die Gruppenadresse des angeschlossenen Optolyzers fest.

```
public short getGroupAdr()
```

Liefert die Gruppenadresse des angeschlossenen Optolyzers zurück.

```
public short getErrorCode()
```

Liefert den Fehlercode zurück, falls einer bei der Verarbeitung im Optolyzer aufgetreten ist.

```
public short getOptoMode()
```

Liefert den aktuellen Betriebsmodus des Optolyzers zurück.

```
public void setOptoMode(short newMode)
```

Legt den Betriebsmodus des Optolyzers fest.

```
public short getAllocScanMode()
```

Gibt Auskunft darüber, ob die OptoControl-Komponente über Änderungen der synchronen Kanalbelegung informiert.

```
public void setAllocScanMode(short newMode)
```

Legt fest, ob die OptoControl-Komponente für Änderungen bei der Belegung der synchronen Kanäle ein Ereignis erzeugen soll.

```
public void connect()
```

Öffnet die serielle Schnittstelle zum Optolyzer für die gemeinsame Nutzung mit einem anderen Prozess.

```
public void disconnect()
```

Schließt die serielle Schnittstelle für diesen Prozess. Komplementäre Operation zu connect().

```
public boolean getRingLock()
```

Liefert zurück, ob das Synchronisationssignal auf dem MOST-Bus (Lock) fehlerhaft ist oder nicht.

```
public boolean getUseShadow()
```

Gibt Auskunft darüber, ob ein Schattenspeicher zum Zugriff auf die Transceiver-Register verwendet wird.

```
public void setUseShadow(boolean newValue)
```

Aktiviert oder deaktiviert den Schattenspeicher, der die Inhalte der Transceiver-Register enthält.

```
public short getMostReg(short page, short map)
```

Liest das angegebene Transceiver-Register aus.

```
public void setMostReg(short page, short map, short newValue)
```

Setzt den Inhalt des angegebenen Transceiver-Registers auf den Wert von `newValue`.

```
public short reset()
```

Setzt den Zustand des Optolyzers zurück.

```
public short sendNodeMsg(short handle, short destAdr, short msgType, String msgData)
```

Sendet eine MOST-Kontrollnachricht

```
public short flushSpyBuffer()
```

Leert den Puffer des Optolyzers, der bei Betrieb im Spy-Modus verwendet wird.

```
public short startBusLoad(short delay)
```

Startet die Erzeugung von Buslast auf dem Kontrollkanal.

```
public short stopBusLoad()
```

Beendet die Erzeugung von Buslast.

```
public short startStress(short errorMode, short errorCount, short length, short pause)
```

Startet den Stresstest (periodische Fehlererzeugung auf dem Bus)

```
public short stopStress()
```

Beenden des Stresstests.

```
public short updateMostReg(short page, short map, short numBytes)
```

Aktualisiert die Inhalte der angegebenen Transceiver-Register mit den entsprechenden Inhalten aus dem Schattenspeicher (vgl. `UseShadow`).

```
public short updateByMask(int mask)
```

Stößt die Aktualisierung der Informationen über die Optolyzer-Eigenschaften (Version, Lichtsignal, Geräteadresse, ...) an.

C Literaturverzeichnis

- [1] AUTOSAR: *Automotive Open System Architecture (AUTOSAR)*.
URL: <<http://www.autosar.org>>.
- [2] J.P. Bigus: *The Agent Building and Learning Environment*. Proceedings of the Fourth International Conference on Autonomous Agents, Seiten 108–109. ACM Press, 2000.
- [3] J. P. Bigus et al.: *ABLE: A toolkit for building multiagent autonomic systems*. IBM Systems Journal, 41(3), 2002.
URL: <<http://www.research.ibm.com/journal/sj/413/bigus.pdf>>.
- [4] J. Bosch: *Design and Use of Software Architectures*, Addison Wesley, 2000.
- [5] P.J. Brockwell and R.A. Davis: *Introduction to Time Series and Forecasting*, Springer, 1996.
- [6] J. Brutlag: *Aberrant Behavior Detection in Time Series for Network Monitoring*. Proceedings of the 14th Systems Administration Conference, New Orleans & Los Angeles, USA, 2000.
- [7] F. Buschmann et al.: *A System of Patterns. Pattern-Oriented Software Architecture*. Wiley, 1996.
- [8] G. Candea, M. Delgado, M. Chen, und A. Fox: *Automatic Failure-Path Inference: A Generic Introspection Technique for Internet Applications*. Proceedings of the 3rd IEEE Workshop on Internet Applications (WIAPP), San Jose, USA, 2003.
- [9] S.-W. Cheng et al.: *Exploiting Architectural Style for Self-repairing Systems*. Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA), Montreal, Kanada, 2002.
- [10] S. Christensen, L. Petrucci: *Towards a Modular Analysis of Coloured Petri Nets*. Edited by K. Jensen. Proceedings of the 13th Int. Conf. on Application and Theory of Petri Nets. Springer-Verlag, Lecture Notes in Computer Science, Vol. 616, Seiten 113-133, Sheffield, UK, 1992.
- [11] Crnkovic, Larsson: *Building Reliable Component-based Software Systems*. S. 79-83, Artech House, 2002.

- [12] C't (2003): *Rad am Draht*. c't – Magazin für Computertechnik Ausgabe 14/2003, Seiten 170-183, Heise
- [13] L. Degioanni: *Development of an Architecture for Packet Capture and Network Traffic Analysis*. Graduation Thesis, Politecnico Di Torino, 2000.
URL: <<http://winpcap.polito.it/docs/default.htm>>.
- [14] ej-Technologies: *JProfiler*.
URL: <<http://www.ej-technologies.com>>.
- [15] F. Feather, R. Maxion, D. Siewiorek: *Fault Detection in an Ethernet Network Using Anomaly Signature Matching*. ACM SIGCOMM Conference Proceedings On Communications architectures, protocols and applications, San Francisco, USA, 1993.
- [16] D. Garlan, S.-W. Cheng, B. Schmerl. *Increasing System Dependability through Architecture-based Self-repair*. In *Architecting Dependable Systems*, Springer, 2003.
- [17] D. Garlan, B. Schmerl und J. Chang: *Using Gauges for Architecture-Based Monitoring and Adaptation*. In *Working Conference on Complex and Dynamic Systems Architecture*, Brisbane, Australien, 2001.
- [18] P. Horn: *Autonomic Computing, IBM's Perspective on the State of Information Technology*. White Paper, 2001.
URL:<http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf>.
- [19] IBM: *IBM DB2 Everyplace*.
URL: <<http://www.ibm.com/software/data/db2/everyplace/index.html>>.
- [20] IBM: *eServer Produktlinie*.
URL: <<http://www.ibm.com/autonomic/eserver.shtml>>.
- [21] IBM: *IBM Websphere Studio Device Developer*. Enthält unter anderem die J2ME-VM J9.
URL:<<http://www-3.ibm.com/software/wireless/wsdd/>>.
- [22] IETF: *RFC 1157 - Simple Network Management Protocol (SNMP)*, Internet Engineering Task Force, 1990.
- [23] IETF: *RFC 1757 – Remote Network Monitoring Management Information Base (RMON MIB)*, Internet Engineering Task Force, 1995.
- [24] IETF: *RFCs 1902-1907 – Version 2 of the Simple Network Management Protocol (SNMPv2)*, Internet Engineering Task Force, 1996.

- [25] IETF: *RFCs 2271-2275 – Version 3 of the Simple Network Management Protocol (SNMPv3)*, Internet Engineering Task Force, 1998.
- [26] ISO: *OSI, ISO/IEC 7498-1:1994: Open Systems Interconnection Reference Model*. International Organization for Standardization, 1994.
- [27] ITU: *MSC2000, Recommendation Z.120: Message Sequence Charts (MSC)*. ITU General Secretariat, 1999.
- [28] JACOB: *Java-COM Bridge*.
URL: <<http://www.danadler.com/jacob/>>.
- [29] C. Krugel, T. Toth und E. Kirda. *Service Specific Anomaly Detection for Network Intrusion Detection*. Proceedings of the ACM Symposium on Applied Computing, 2002.
- [30] J.C. Laprie: *Dependable Computing and Fault Tolerance: Concepts and Terminology*. 15th Symposium on Fault Tolerant Computing (FCTS), 1985, Ann-Arbor, USA, 1985.
- [31] W.Lee, S.J. Stolfo, K.W.Mok: *Mining in a Data-Flow Environment*. Proceedings of the Fifth International Conference on Knowledge Discovery and Data Mining (KDD-99), San Diego, USA, 1999.
- [32] A. Leonhardi et. al.: *Towards Self-healing In-vehicle Telematics Systems*. VDI Tagung Elektronik im Kraftfahrzeug, Baden Baden, September 2003.
- [33] D.J. Marchette: *Computer Intrusion Detection and Network Monitoring. A Statistical Viewpoint*. Springer, 2001.
- [34] S. McCanne, V. Jacobson: *The BSD Packet Filter: A New Architecture for User-level Packet Capture*. 1993 Winter USENIX Conference, San Diego, USA, 1993.
- [35] Microsoft: *Component Object Model (COM)*.
URL: <<http://www.microsoft.com/com/>>
- [36] J. Mogul: *Efficient Use of Workstation for Passive Monitoring of Local Area Networks*. Proceedings of the ACM Symposium on Communications Architectures and Protocols, Seiten 253-263, Philadelphia, USA, 1990.
- [37] J. Mogul, R.F. Rashid, M.J. Accetta: *The Packet Filter: An Efficient Mechanism for User-level Network Code*. ACM Operating Systems Review, SIGOPS, 1987.
- [38] MOST Cooperation. URL: <<http://www.mostnet.org>>.

- [39] MOST Cooperation: *MOST Function Catalog*. MOST Cooperation, 2003.
URL: <[http://www.mostnet.org/Specifications/MOST Function Catalog/](http://www.mostnet.org/Specifications/MOST_Function_Catalog/)>.
- [40] MOST Cooperation: *MOST High Protocol Specification Rev. 2.1*. MOST Cooperation, 2003.
URL: <<http://www.mostnet.org/downloads/Specifications/MOSTHigh-ProtocolSpec.pdf>>.
- [41] MOST Cooperation: *MOST Specification Framework Rev 1.1*. MOST Cooperation, 2001.
URL: <[http://www.mostnet.org/downloads/Specifications/MOST Framework/](http://www.mostnet.org/downloads/Specifications/MOST_Framework/)>.
- [42] MOST Cooperation : *MOST Specification Rev 2.2*. MOST Cooperation, 2002.
URL:<<http://www.mostnet.org/downloads/Specifications/MOSTSpecification.pdf>>.
- [43] C. J. Murray: 'Cold War' hits telematics. EE Times, October 25, 2002,
URL: <<http://www.eetimes.com/issue/fp/OEG20021025S0041>>.
- [44] Oasis Silicon Systems : *MOST Data Analyzer System User Manual for Version 2.00*. Oasis Silicon Systems AG, 2003.
URL: <<http://www.oasis.de>>.
- [45] Oasis Silicon Systems : *Optolyter4MOST Professional Edition User Manual for Version 1.42*. Oasis Silicon Systems AG, 2003.
URL: <<http://www.oasis.de>>.
- [46] OMG: Unified Modeling Language (UML), Version 1.5. Object Management Group, 2003.
URL: <<http://www.omg.org/technology/documents/formal/uml.htm>>.
- [47] Open Service Gateway Initiative (OSGI) : *OSGI Service Platform Specification Release 3*. Open Service Gateway Initiative, 2003.
URL: <<http://www.osgi.org>>.
- [48] ROC: *The Berkeley/Stanford Recovery-Oriented Computing (ROC) Project*.
URL: <<http://roc.cs.berkeley.edu/>>.
- [49] R. Sekar et. al.: *Specification-based Anomaly Detection: A New Approach for Detecting Network Intrusions*. Proceedings of the ACM Conference on Computer and Communications Security 2002, November 2002.

- [50] SLAC: *Network Monitoring Tools*.
URL:< <http://www.slac.stanford.edu/xorg/nmtf/nmtf-tools.html>>.
- [51] Der Spiegel (2003): *Fahrzeug steht, Kunde läuft*. Ausgabe 40/2003.
URL: <<http://www.spiegel.de/spiegel/0,1518,267507,00.html>>.
- [52] W. Stallings: *SNMP, SNMPv2 and RMON Practical Network Management*. Addison-Wesley Professional Computing and Engineering, 1996.
- [53] S. Staniford, J.A. Hoagland und J.M. McAlerney. *Practical Automated Detection of Stealthy Portscans*. Proceedings of the IDS Workshop of the 7th Computer and Communications Security Conference, 2000.
- [54] M. Stümpfle, et. al.: *COSIMA - A Component System Information and Management Architecture*. Proceedings of the IEEE Intelligent Vehicles Symposium 2000, Dearborn, USA, 2000.
- [55] Sun Microsystems: *Java 2 Platform, Micro Edition (J2ME)* .
URL: <<http://java.sun.com/j2me>>.
- [56] Sun Microsystems: *Java Native Interface Specification 1.1*.
URL: < <http://java.sun.com/j2se/1.3/docs/guide/jni/spec/jniTOC.doc.html>>
- [57] C. Taylor und J. Alves-Foss. *NATE - Network analysis of anomalous traffic events, a low-cost approach*. Proceedings of New Paradigms in Security Workshop, Cloudcroft, USA, Sept. 2001.
- [58] C. Thiel: *Optische Datenübertragung mit MOST*. In *Elektronik Sonderheft Automotive* 04/2002.
- [59] R. Wolski, N. Spring, C.Peterson: *Implementing a Performance Monitoring System for Metacomputing: The Network Weather Service*. Proceedings of the 1997 ACM/IEEE Conference on Supercomputing, San Jose, USA, 1997.
- [60] R. Wolski : *Dynamically Forecasting Network Performance Using the Network Weather Service*. UCSD Technical Report TR-CS96-494, Computer Science and Engineering Department University of California San Diego, USA, 1998.
- [61] R. Wolski, N. Spring, J. Hayes: *The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing*. *Journal of Future Generation Computing Systems*, Volume 15, Numbers 5-6, Seiten 757-768, October, 1999.

- [62] J.A. Zinky, F.M. White: *Visualizing Packet Traces*. ACM SIGCOMM Conference Proceedings on Communication architectures and protocols, Seiten 293-304, 1992.

Erklärung

Hiermit versichere ich, diese Arbeit
selbständig verfasst und nur die
angegebenen Quellen benutzt zu haben.

(Jan Geiger)