DIPLOMARBEIT

Middleware-Konzepte für verteilte Automatisierungssysteme basierend auf IEC 61499

ausgeführt zur Erlangung des akademischen Grades eines Diplom-Ingenieurs

unter der Leitung von

Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Markus Vincze
und
Projektass. Dipl.-Ing. Oliver Hummer-Koppendorfer

E376
Institut für Automatisierungs- und Regelungstechnik

eingereicht an der Technischen Universität Wien
Fakultät für Elektrotechnik und Informationstechnik

von

Bernhard Voglmayr
Matr.-Nr.: 0125685
Boltzmanngasse 10, 1090 Wien
Tel.: 0650 / 25 24 022
Email: berni.voglmayr@gmx.at

Wien, am 13.09.2007 _____

Kurzfassung

Durch die Verwendung von Middleware wird die Entwicklung von verteilten Systemen und Anwendungen wesentlich vereinfacht. Hardware- und Software-spezifische Eigenschaften, sowie die physikalische Verteilung werden vor der Applikation verborgen, und sämtliche Kommunikationsabläufe durch die Middleware koordiniert. Zusätzlich werden diverse Services, wie z.B. Echtzeitüberwachung des Programm- und Systemzustandes, angeboten und mit einer definierten Quality-of-Service (QoS) ausgeführt. Vor allem im IT-Bereich haben sich Middleware-Konzepte beispielsweise zur Abwicklung von Geschäfts- oder Business-to-Business-Prozessen (B2B) bewährt, und auch die verteilte Automation könnte von besagter Middleware-Funktionalität profitieren.
Um eine Umsetzung in der Automatisierungstechnik, speziell für IEC 61499-Systeme, zu ermöglichen, wird in dieser Arbeit zuerst der Stand der Technik identifiziert, und die Voraussetzungen und Anforderungen für eine Implementierung von Middleware-Funktionalität in Automatisierungssystemen betrachtet. Neben der industriellen Kommunikation werden die relevanten QoS-Parameter sowie eventuelle Ressourceneinschränkungen analysiert. Gängige Middleware-Produkte (CORBA bzw. RT-CORBA, DCOM, .NET, Java EE, DDS und OSA+) werden auf diese Punkte im Hinblick auf einen direkten Einsatz innerhalb von Automatisierungssystemen untersucht. Da - wie sich zeigt - keines der Produkte allen Anforderungen genügt, wird auch die Entwicklung bzw. der Einsatz von Protokollen abgewogen. Auch hier zeigen sich einige Einschränkungen, weshalb letztendlich die Nachbildung von Middleware-Funktionalität mit IEC 61499-Mitteln favorisiert wird.
Ausgewählte Mechanismen werden mithilfe diverser Diagramme rein funktional beschrieben und für eine Implementierung aufbereitet. Diese Ergebnisse können mit dem IEC 61499-konformen FBDK (Function Block Development Kit) der Firma HOLOBLOC Inc. umgesetzt werden. Als Endresultat präsentiert sich eine Bibliothek an Kommunikationsfunktionsblöcken mit unterschiedlichen QoS-Möglichkeiten, die für einen konkreten Einsatz bereitsteht.
Eine Einschätzung der Ergebnisse sowie ein Ausblick auf mögliche Weiterentwicklungen runden die Diplomarbeit ab.

Abstract

The usage of middleware makes the development of distributed systems and applications much easier. Hardware- and software-specific features, as well as the physical distribution, are hidden from the application, and all communication activities are coordinated by the middleware. Additionally, various services such as real-time monitoring of the program and system status are offered and executed with a defined quality of service (QoS). Middleware concepts are proven mainly in the IT-domain, e.g. for handling business, specifically business-to-business (B2B) processes. Distributed automation could benefit from middleware functionality as well.
In order to enable a realisation in automation, especially for IEC 61499-systems, the state of the art is identified and the preconditions and requirements for an implementation of middleware functionality in automation systems are considered. Apart from industrial communication all relevant QoS-parameters and potential resource constraints are analysed. Concentrating on these points prevalent middleware-products (CORBA and RT-CORBA, DCOM, .NET, Java EE, DDS and OSA+) are observed in terms of a direct insertion into an automation system. As no product can fulfil all requirements, the development, and deployment of protocols is weighed. In this case some constraints eventually appear for which the emulation of middleware functionality with means of IEC 61499 is favoured.
Selected mechanisms are described with the help of different diagrams in a functional way and pre-processed for implementation. These results can be implemented by dint of the IEC 61499-compliant FBDK (Function Block Development Kit) from HOLOBLOC Inc., resulting in a library of communication function blocks with different QoS-possibilities that are available for a concrete application.
An evaluation of the achievements as well as future prospects regarding the continuation of the results top off this diploma thesis.

Danksagung

Für seine umfassende Unterstützung danke ich meinem Betreuer Oliver. Auch bedanken möchte ich mich bei Christoph, der durch seine (Diskussions-) Beiträge stets hilfreich zur Seite stand.
Meiner Familie danke ich für ihren Beistand während meines Studiums - unsagbar dankbar bin ich meinen Eltern, die mir so vieles ermöglicht haben.
Ganz besonderer Dank gilt meiner Freundin, die mir fortwährend ein offenes Ohr geschenkt und mich ermutigt hat.

Inhaltsverzeichnis

1 Einleitung

Die zunehmend komplexer werdenden Anlagen und Steuerungsprogramme verlangen nach Maßnahmen zur Strukturierung, Kapselung und Wiederverwendung von Funktionalität [FB04]. Zu diesem Zweck wurde innerhalb des SPS-Standards IEC 61131-3 [Int03] der Funktionsblock (engl. Function Block, FB) eingeführt, der mittlerweile zum Stand der Technik gehört.
Das Konzept des Funktionsblocks hilft dabei, komplexe Systeme übersichtlich zu halten und unterstützt gleichzeitig das Prinzip der Wiederverwendung. Durch die Kapselung der Funktionalität in einem Funktionsblock entsteht eine Black Box, die von anderen FBs weitestgehend unabhängig ist, wodurch eine robustere, weniger fehleranfällige Gesamtleistung erzielt wird (im Verbund mit anderen FBs).
Parallel dazu zeichnet sich ein Übergang von dezentralen Architekturen hin zu verteilten Systemen ab. In diesem Zusammenhang nimmt der Standard IEC 61499 [Lew01] eine zentrale Rolle ein, der in Erweiterung zu IEC 61131 auch Verteilung unterstützt und Interoperabilität, Portabilität und Konfigurierbarkeit garantieren soll.

Innerhalb verteilter Anwendungen bestehen oft komplexe Kommunikationsbeziehungen, für die das Engineering mit den bisherigen Mitteln sehr aufwendig werden kann, zumal die Konfiguration durch die verschiedenen Busse und ihre vielen Parameter kompliziert wird.
Bei der Applikationserstellung mittels FBs geht man beispielsweise folgendermaßen vor: Zuerst verschaltet man mehrere FBs miteinander, d.h. man verbindet ihre Ein- und Ausgänge entsprechend der gewünschten Funktionalität miteinander. Von diesem resultierenden Funktionsblocknetzwerk (Applikation) ausgehend überlegt man sich, welche FBs welchen Geräten (engl. devices) zugeordnet sind, also auf welchen Geräten die einzelnen FBs ausgeführt werden sollen. Die Zuordnung von FBs zu Devices bzw. Ressourcen bezeichnet man als Mapping.
Durch diese Verteilung werden bestehende Verbindungen zwischen den FBs unterbrochen, nämlich jene, die nun über die jeweilige Gerätegrenze hinweg existieren müssten. Als Ausweg können so genannte Service Interface Function Blocks (SIFBs) eingesetzt werden, die eine Kommunikation zwischen FBs

auf unterschiedlichen Geräten erlauben. Zu diesem Zweck müssen besagte SIFBs für gewöhnlich „per Hand“ gesetzt werden, was bedeutet dass für jede aufgebrochene Verbindung jeweils zwei SIFBs in das Funktionsblocknetzwerk aufgenommen werden müssen, nämlich pro kommunizierendem Device einer. Es ist einsichtig, dass dieser Prozess mitunter sehr mühsam sein kann.

Mit der Einführung einer Kommunikation zwischen den FBs stellt sich nun auch die Frage nach der Qualität der wiederhergestellten Verbindungen.
Aus der Informatik sind in diesem Zusammenhang Middleware-Konzepte bekannt, wobei der Middleware die Rolle eines Kommunikations-Managements zwischen unterschiedlichen Rechnern innerhalb einer verteilten Anwendung zukommt. Neben den eigentlichen Aufgaben, wie der Verbergung der Heterogenität oder der Vermittlung zwischen den Rechnern, bieten diese Middleware-Lösungen (z.B. RT-CORBA) zusätzliche Services an, wie z.B. Quality-of-Service (QoS) - Überwachung oder die Einhaltung von Echtzeit-Bedingungen. Diese Middleware-Konzepte sind aber innerhalb der verteilten Automation nur bedingt einsetzbar - mitunter sind sie für die Zielsysteme und ihre Embedded Devices meist zu umfangreich und groß.

Im Zusammenhang mit der Norm IEC 61499 und der Zusammenschaltung mehrerer FBs über Gerätegrenzen hinweg ermöglichen sich durch die Anlehnung an bekannte Middleware-Konzepte bequeme Erweiterungen: Durch die Bereitstellung diverser Middleware-Funktionalität, im Speziellen unterschiedlicher QoS-Mechanismen, für die verteilte Automation soll die Kommunikationskonfiguration in verteilten Applikationen nach IEC 61499 vereinfacht werden. Als Basis sollen hierbei besagte Middleware-Konzepte aus der Informatik dienen, welche eingehend analysiert und auf ihre Tauglichkeit für die industrielle Automation (IAT) untersucht werden sollen.

1.1 Zielsetzung und Motivation

Das Ziel dieser Diplomarbeit ist die einfache Konfiguration der Kommunikation in verteilten Applikationen nach IEC 61499. Dadurch soll erreicht werden, dass der Anwender bzw. Applikationsentwickler nicht notwendigerweise ein Programmierer oder IT-Spezialist sein muss. Vielmehr soll der jeweilige Spezialist für einen bestimmten Prozess (engl. domain expert) in der Lage sein können, diverse Änderungen oder Neuprogrammierungen selbst durchzuführen.

Als Ansatz hierzu sollen Middleware-Services mit definierbarer QoS dienen. Konkret sollen die unterschiedlichen Anforderungen der Anwendung durch die QoS beschrieben und die Kommunikation über die QoS parametriert werden. Zu diesem Zweck muss u.a. untersucht werden, ob ein direkter Einsatz von Middleware möglich ist, oder eine Adaption bzw. Imitation nötig ist.
Für die Umsetzung ist es wichtig, Middleware-Konzepte aus der Informatik sowie die industrielle Kommunikation und relevante QoS-Parameter zu analysieren, wobei dem Problem der begrenzten Ressourcen der Embedded Devices entgegnet werden muss, um diese Konzepte für die Automatisierung nutzbar machen zu können.
Eine zusätzliche Motivation dieser Arbeit ist sicherlich die Verbergung von Komplexität sowie die Erhöhung der Funktionalität und Zuverlässigkeit. Durch eine Erweiterung der Funktionalität kann in der Folge die Usability gesteigert, sowie das Monitoring und die Fehlersuche verbessert werden.

1.2 Hintergrund

Die Idee zu dieser Diplomarbeit wurde im Rahmen des Projekts εCEDAC (Evolution Control Environment for Distributed Automation Components) geboren [Evo]. Das übergeordnete Ziel dieses Projekts ist es, Änderungen von Steuerungsprogrammen im laufenden Betrieb zu ermöglichen. Der Vorteil eines derartigen Eingriffs liegt auf der Hand: Ohne die komplette Anlage herunterfahren zu müssen, könnten Software, Hardware oder sogar ganze Produktionsabläufe umgestellt werden. Die Zeit- und damit verbundene Kostenersparnis wäre beträchtlich.
Ein Teilbereich dieses Projekts ist die Entwicklung eines Engineering-Tools zur rechnergestützten Softwareentwicklung. Die Applikationserstellung soll hierbei in Form von Funktionsblöcken erfolgen. Um die Usability dieser Software zu steigern, ist geplant, die beim Mapping entstehenden Unterbrechungen toolgestützt wiederherzustellen. Der Benutzer soll die Möglichkeit haben die Kommunikation nach seinen Anforderungen und Wünschen zu konfigurieren, z.B. indem er durch Doppelklick auf eine aufgetrennte Linie einen Dialog öffnet, in dem er aus allen verfügbaren Kommunikationswegen den gewünschten auswählt. Um eine derartige Funktionserweiterung durchführen zu können, ist es zuerst erforderlich, entsprechende SIFBs zur Verfügung zu haben. Hier setzt diese Diplomarbeit an, die somit auch die Grundlage bzw. Vorarbeit zur besagten Software-Erweiterung darstellt: der Anwender soll in die Lage versetzt werden, direkt die Kommunikation auf

Applikationsebene zu parametrieren anstatt eines speziellen Bussystems. Als Mittel zur Beschreibung und Spezifikation von Kommunikationseigenschaften auf einem abstrakten Niveau dienen dabei QoS-Parameter.

1.3 Leitfaden durch die Arbeit

Der Rest der Diplomarbeit ist wie folgt aufgebaut:
Kapitel 2 dient der Einführung in die Thematik und liefert die nötigen Grundlagen zum Verständnis der folgenden Kapitel. Es soll den aktuellen Stand der Technik umreißen und den Ausgangspunkt für diese Arbeit darstellen. Die Schwerpunkte liegen auf allgemeinen und konkreten Middleware-Konzepten sowie auf der Kommunikation in der IAT. Außerdem wird der Begriff der QoS näher erläutert und eine Einführung in die Norm IEC 61499 gegeben.
Kapitel 3 beschäftigt sich mit der Analyse der industriellen Kommunikation und den daraus resultierenden Anforderungen. Ebenso werden die relevanten QoS-Parameter und die vorhandenen Middleware-Konzepte analysiert. Neben einer Untersuchung des Einsatzes von Protokollen werden auch gängige QoS-Mechanismen durchleuchtet und im Hinblick auf eine spätere Implementierung genau (funktional) beschrieben. Der Konzeptteil beschreibt somit die Möglichkeit einerseits einer direkten Umsetzung von Middleware-Funktionalität, andererseits einer Adaptierung bzw. Nachbildung dieser. Durch eine Bewertung der Möglichkeiten kann die Auswahl des bestgeeignetsten Ansatzes für die Automatisierungstechnik, speziell für IEC 61499, getroffen werden.
Im folgenden Kapitel steht die Implementierung von Middleware-Funktionalität mit Mitteln der IEC 61499 (z.B. in Form von Service Interface FBs) im Brennpunkt. Es werden nur die wichtigsten Ergebnisse herausgegriffen und näher beschrieben.
Die Diskussion der Erkenntnisse und Errungenschaften ist Thema von Kapitel 5.
Die Diplomarbeit schließt mit einer Schlussfolgerung (Kapitel 6) und einem Ausblick (Kapitel 7).
Im Anhang finden sich neben den einzelnen Verzeichnissen (Glossar, Index, Abbildungen, Tabellen und Literatur) ergänzende Erläuterungen.

2 Stand der Technik

Grundsätzlich muss in dieser Arbeit eine Brücke zwischen Middleware-Systemen und IEC 61499-Systemen geschlagen werden, wobei gewisse Anforderungen bzw. Randbedingungen einzuhalten sind. Diesem einfachen Schema entspricht auch der Aufbau dieses Kapitels: Die Anforderungen leiten sich aus der Kommunikation in der IAT (unter Einbeziehung gängiger Modelle bzw. Mechanismen, siehe Kapitel 2.1) sowie aus der QoS (Kapitel 2.3) ab. Eine allgemeine Darstellung von Middleware sowie kurze Beschreibungen bekannter Konzepte ist Inhalt von Kapitel 2.2. Dem IEC 61499-Standard ist Kapitel 2.4 gewidmet.

2.1 Kommunikation (in der IAT)

Die Koordination der Kommunikation zwischen Geräten in industriellen Anlagen übernahm bis Anfang der 90-er Jahre eine zentrale Steuerung [SFS05]. Die schwere Wartbarkeit derartiger Systeme sowie der enorme Verkabelungsaufwand waren u.a. Antriebsfaktoren für die Verwendung und Verbreitung von Feldbussen. Mittlerweile haben sich Feldbussysteme im Bereich der Automation etabliert. Eine Übersicht über die branchenspezifischen Anforderungen an die Kommunikation innerhalb der Automation gibt Tabelle 2.1.

Als neuer Trend ist das Vordringen von Ethernet in Kombination mit TCP/IP vor allem im Bereich der Fertigungsautomation zu bemerken [SFS05], [PB07]. In diesem Zusammenhang ist oft von Industrial Ethernet die Rede. Die Vorteile dieser Technologie sind unbestritten. Zum einen rechtfertigen die enormen Kostenvorteile (bedingt durch die großen Stückzahlen) einen Einsatz, andererseits garantiert eine große Bandbreite (derzeit sind 100 MBit/s üblich, Ausweitung auf 1 - 10 GBit/s möglich) die Übertragung von ebenso großen Datenmengen. Die einfache Integration in das Firmennetzwerk muss mitunter auch genannt werden.
Als wesentlicher Nachteil des Standard-Ethernets ist das nicht-deterministische Verhalten anzuführen, d.h. es kann zu unvorhersehbaren Verzögerungen kommen, die strenge Echtzeitanforderungen uneinhaltbar machen. Diese

	Fertigungsautomation (Roboter)	Prozessautomation (Chemie/Food)	Gebäudeautomation (Heizung/Lüftung)
Zykluszeiten	1 ms	100 ms	100 ms
Anzahl I/O	< 100	> 100	100…1000
Kabellängen	< 100 m	> 1000 m	> 1000 m
Störsicherheit	hoch	hoch	mittel
Aufgaben	Ablauf steuern, regeln, überwachen	Ablauf steuern, regeln, überwachen	Überwachen, parametrieren, Service
Beispiele	Profibus DP, Interbus, ControlNet, CAN	Profibus PA, Foundation Fieldbus	LON, EIB, Modbus-TCP (Ethernet)

Tabelle 2.1: Branchen der Automation und ihre Kenngrößen [SFS05]

Tatsache ist auf das Zugriffsprotokoll von Ethernet (nämlich CSMA/CD[1]) zurückzuführen, bei dem jeder Teilnehmer zu jeder Zeit Daten versenden kann, was zwangsläufig zu Kollisionen führen muss. Wenn keine harte Echtzeit erreicht werden muss, wird Ethernet aber mit seiner großen Bandbreite kaum ausgelastet und folglich schnell genug sein. Eine Kombination des CSMA-Verfahrens mit einem Master/Slave-Verfahren kann ebenso dazu beitragen, Kollisionen zu vermeiden, genauso wie der Einsatz von Switches, der allerdings einen größeren Verdrahtungsaufwand bedingt und die Möglichkeit der Uhrensynchronisation zunichte macht [Fel00]. Zum Thema „Ethernet und Echtzeit“ gibt [RTIa] Auskunft.
Weiters sind beim Ethernet die Bereiche Störsicherheit, Zuverlässigkeit und Einsatz in explosionsgefährdeten Bereichen problematisch. Eine Energieversorgung der Feldgeräte über die Feldbusleitung ist bisweilen auch nicht möglich und in punkto Sicherheit (illegaler Zugriff) existieren einige Bedenken. Besagte Nachteile sind auch der Grund dafür, weshalb Ethernet in der Prozessautomation bis jetzt nur sehr eingeschränkt zum Einsatz kommt.

Bezüglich des Kommunikationsmodells wird bei Feldbussen oft zwischen einem Client-Server-Modell und einem Publish-Subscribe-Modell unterschieden. Diese Modelle werden im Folgenden kurz vorgestellt.

[1] CSMA/CD steht für „Carrier Sense Multiple Access/Collision Detection“ und ist ein Buszugriffsverfahren, bei dem die Teilnehmer Kollisionen erkennen und durch erneutes, zeitversetztes Senden korrigieren können [Wika]. CSMA/CD kommt, wie bereits erwähnt, bei Ethernet zum Einsatz und ist dort als IEEE 802.3 standardisiert.

Client-Server-Modell
Bei dieser Netzwerkstruktur bietet ein Server über das Netz seine Dienste (bzw. Daten) an, die von mehreren Clients über so genannte Requests angefordert und somit genutzt werden können [Wikj]. Der Server fungiert als Dienstleister oder Slave, der auf Anforderungen wartet, diese verarbeitet und das entsprechende Ergebnis an den Client retourniert. Der Client kann als Dienstnehmer oder Master bezeichnet werden, dem im Gegensatz zum Server eine aktive Rolle zukommt.
Beim Client-Server-Prinzip können mehrere Clients gleichzeitig mit einem Server kommunizieren, weshalb man in diesem Zusammenhang auch von einer n-zu-1-Kommunikation spricht. An der eigentlichen Kommunikation sind allerdings nur zwei Teilnehmern beteiligt (ein Client und ein Server). Aus diesem Grund kann es Probleme mit der örtlichen Konsistenz geben, wenn ein Client bei mehreren Servern Daten anfordert, da die Konsistenz immer nur zwischen dem Client und einem Server gewährleistet ist [Fel00].
Bei vielen gleichzeitigen Anfragen an einen Server kann es zu Überlastungen kommen, da der Server in diesem Fall zum Flaschenhals wird. Diesem Nachteil kann nur mit mehreren Servern begegnet werden. Ein Serverausfall stellt ein weiteres Problem dar, da dann die Requests der Clients unbeantwortet bleiben.
Bei einem Peer-to-Peer (P2P) - Netzwerk sind die Rollen von Client und Server vertauschbar, d.h. ein Client kann auch die Funktion eines Server übernehmen und umgekehrt.

Publish-Subscribe-Modell
Bei dieser Kommunikationsform veröffentlicht (engl. to publish) ein Publisher Daten, indem er diese auf das Übertragungsmedium legt [PCSH99] (vgl. auch [KDL00]). Diese Daten haben typischerweise einen Titel (engl. topic), unter dem sie veröffentlicht werden und einen Typ (engl. type), also ein gewisses Datenformat. Ein Subscriber, der an einer Publikation (engl. publication) interessiert ist, kann diese unter dem jeweiligen Titel abonnieren (siehe Abbildung 2.1).

Beispielsweise existiert in einem verteilten System ein Drucksensor, der in regelmäßigen Abständen den Druck einer Turbine erfasst. Diese Werte können anderen Komponenten zur Verfügung gestellt werden, indem sie ein Publisher unter dem Titel „Druck“ auf den Bus legt. Ein Subscriber kann sein Interesse an diesen Daten kundtun, indem er ein „Druck“-Abo anmeldet, d.h. er hört den Bus auf Druckwerte ab, wodurch er sämtliche Druckwerte des Publishers empfangen kann.

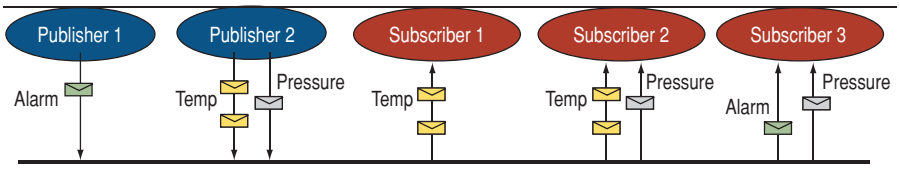

Abbildung 2.1: Publish/Subscribe-Prinzip [PC05]

Beim Publish-Subscribe-Modell kommt es zu einer Entkopplung von Sender und Empfänger, da der Sender keine Information über die Anzahl der Empfänger braucht und vice versa (siehe Abbildung 2.2). Dieser Anonymität verdankt das Publish-Subscribe-Prinzip auch seinen Namen „Fire and Forget“, denn der Publisher braucht sich nach Abschicken seiner Nachrichten nicht mehr weiter um sie zu kümmern.

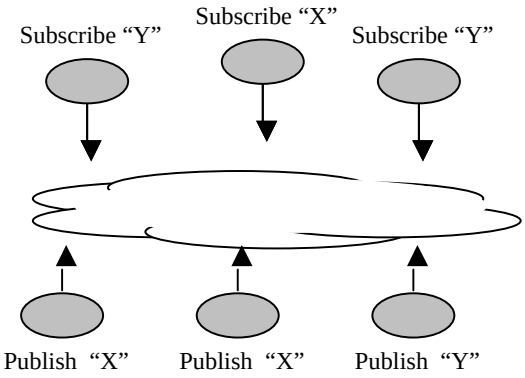

Abbildung 2.2: Entkopplung von Publishern und Subscribern [PCSH99]

Im Gegensatz zum Client-Server-Modell spricht man hier auch von einer 1-zu-n-Kommunikation, wobei auch eine n-zu-n-Kommunikation realisiert werden kann. Im obigen Beispiel könnten mehrere Drucksensoren installiert sein, die ihre Werte jeweils über einen eigenen Publisher veröffentlichen - jedoch alle unter dem Titel „Druck“. Mehrere Subscriber könnten in der Folge von mehreren Publishern Daten empfangen.
Ein weiterer Unterschied zum Client-Server-Modell ist die ereignisgesteuerte Form der Kommunikation, d.h. der Publisher sendet die Daten, sobald sie zur Verfügung stehen, der Subscriber reagiert durch entsprechende Behandlung der erhaltenen Daten.
Das einfache Prinzip von Publish/Subscribe unterstützt auch die Skalierbar-

keit und Wiederverwendbarkeit, wodurch Systemupgrades erleichtert werden.

Des Weiteren werden zwei wichtige Möglichkeiten der Interaktion basierend auf dem Client-Server-Modell beschrieben. Diese Kommunikationstechniken sind auf der fünften (zum Teil auch auf der sechsten Schicht) des ISO/OSI-Modells[2] angesiedelt und kommen u.a. bei diversen Middleware-Ansätzen (z.B. CORBA, DCOM oder Java RMI) zum Einsatz.

RPC

Beim entfernten Prozeduraufruf (engl. Remote Procedure Call) kann ein Client eine Prozedur auf einem entfernten Rechner (Server) aufrufen [Ham05]. Eine sogenannte Client-Stub-Prozedur erzeugt den Eindruck eines lokalen Prozeduraufrufs - in Wirklichkeit wird der Aufruf zum Server weitergeleitet, wo er von einer Server-Stub-Prozedur an die eigentliche Prozedur übergeben wird. Die Antwort des Servers geht über die gleichen Stationen zum Client retour.
Während beim herkömmlichen RPC der Client bis zum Erhalt der Server-Antwort blockiert (siehe Abbildung 2.3), sendet der Server beim asynchronen RPC schon beim Erhalt der Anforderung eine Antwort an den Client zurück. Erst dann kümmert sich der Server um den Prozeduraufruf; der Client kann schon nach Erhalt der Empfangsbestätigung seine Arbeit fortsetzen und braucht nicht bis auf das endgültige Ergebnis warten.

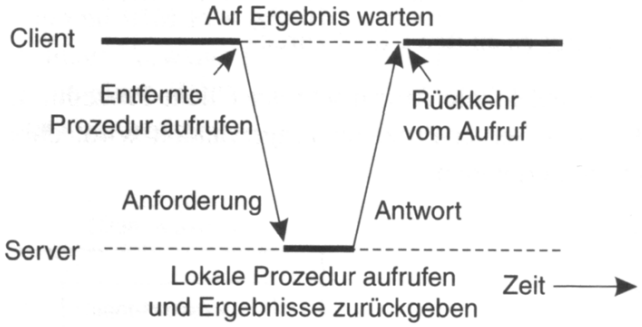

Abbildung 2.3: Remote Procedure Call (RPC) [TvS03]

[2]Das OSI-Modell (Open Systems Interconnection) der ISO (International Standards Organization) organisiert die Kommunikation in sieben Schichten und soll bei der Entwicklung von Kommunikationsprotokollen helfen, indem es jeder Schicht definierte Aufgaben zuweist [Wikf], [Int96].

RMI
Der entfernte Methodenaufruf (engl. Remote Method Invocation) ist das objektorientierte Pendant zum RPC, mit dessen Hilfe Methoden von entfernten Objekten aufgerufen werden können. Hier wird der Eindruck eines lokalen Methodenaufrufs durch ein sogenanntes Proxy-Objekt im Client-Adressraum erweckt, das dem Client das eigentliche Serverobjekt vortäuscht. Auch beim RMI dient ein Client-Stub dem Datenverpacken (engl. Marshalling) vor bzw. dem Datenauspacken (engl. Unmarshalling) nach der Übertragung sowie dem Verbindungsaufbau. Auf Server-Seite leitet der Server-Stub den Aufruf an das Serverobjekt weiter. Der Rückweg ist der gleiche. Der Client blockiert bis zum Erhalt des Ergebnisses.

2.2 Middleware

Der Begriff Middleware wird oft in einem Atemzug mit verteilten Systemen genannt, wobei ein verteiltes System laut Tanenbaum als
„*eine Menge voneinander unabhängiger Computer, die dem Benutzer wie ein einzelnes, kohärentes System erscheinen*“
angesehen werden kann [TvS03]. Neben dieser Transparenz will ein verteiltes System Benutzer und Ressourcen möglichst einfach miteinander in Verbindung bringen und sowohl Offenheit als auch Skalierbarkeit ermöglichen.
Die Verbergung der Heterogenität (sowohl hinsichtlich Hardware, aber auch die unterschiedlichen Betriebssysteme, Software und Programmiersprachen betreffend) und das Kommunikations-Management werden dabei häufig von einer so genannten Middleware erledigt. Es handelt sich hierbei um
„*eine Softwareschicht über der Betriebssystemebene, welche die Entwicklung verteilter Systeme unterstützt*“ [WB05] bzw. wenn man das Betriebssystem als jene Software ansieht, die die Hardware nutzbar macht, so kann man Middleware auch als eine Software definieren, die ein verteiltes System programmierbar macht [Pic04], [Bak01].

Der Name Middleware leitet sich aus der logischen Positionierung dieser Softwareschicht zwischen einer verteilten Anwendung und der Betriebssystem-Schicht ab (siehe Abbildung 2.4), wobei die Middleware ebenso verteilt ist wie die Applikation.
(Grundsätzlich ist zwischen einem verteilten System und einer verteilten Anwendung zu unterscheiden [Ham05]. Die verteilte Anwendung besteht aus mehreren Komponenten, die in ihrer Gesamtheit eine bestimmte Funktionali-

tät erfüllen. Diese Komponenten befinden sich auf den Rechnern des verteilten Systems, das der verteilten Anwendung seine Kommunikationsinfrastruktur zur Verfügung stellt.)

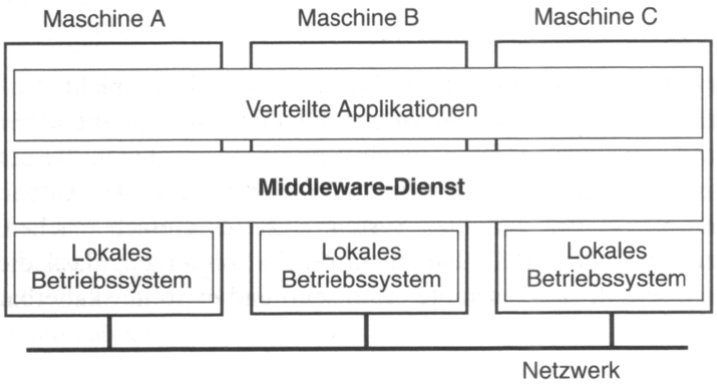

Abbildung 2.4: Logische Positionierung von Middleware [TvS03]

Um eine bessere Vorstellung vom Begriff Middleware zu bekommen, sei eine letzte, etwas ausführlichere Definition angeführt:
„*Middleware is the software that assists an application to interact or communicate with other applications, networks, hardware, and/or operating systems. This software assists programmers by relieving them of complex connections needed in a distributed system. It provides tools for improving quality of service (QoS), security, message passing, directory services, file services, etc. that can be invisible to the user.*“ [BK03]

Man merkt hier schon, dass die Verwendung von (standardisierter) Middleware dem Programmierer eine Menge an Arbeit (Netzwerkprogrammierung) erspart [WB05]. Typische Aufgaben der Middleware wie die Identifikation und Lokalisierung von Objekten innerhalb des verteilten Systems, sowie die Koordination der Interaktion zwischen den Objekten und eine eventuell nötige Fehlerbehandlung müssen somit nicht gesondert implementiert werden. Der Preis dafür ist ein zusätzlicher Overhead in zweierlei Hinsicht. Einerseits bedeutet ein allgemeiner Lösungsweg mittels Middleware im Gegensatz zu einer maßgeschneiderten Lösung mehr Speicherbedarf. Andererseits werden durch die Middleware mehr Ressourcen bei der Verwaltung und Kommunikation benötigt, da hier zusätzliche Informationen (z.B. Protokollinformationen) verarbeitet bzw. transportiert werden müssen.

Ein wesentlicher Bestandteil jeder Middleware sind die der Anwendung zur Verfügung gestellten Dienste (engl. services) [Ham05]. Jeder dieser Dienste hat seine eigene Funktionalität, die jeweils über ein so genanntes API (Application Programming Interface) der Applikation zugänglich ist. Im Gegensatz zu den Programmierschnittstellen der Netzwerkdienste oder der Systemdienste eines Betriebssystems sind die hier verwendeten APIs abstrakter und zugleich funktioneller.
Dienste sind anwendungsunabhängig und somit leicht wieder verwendbar. Nicht jede Middleware bietet die gleichen Dienste an. Deshalb sollen hier nur kurz einige wichtige Dienste vorgestellt werden.

Namensdienst
Dieser Dienst kann mit einem Telefonbuch verglichen werden, denn er erlaubt das Suchen, Finden und Nutzen von Ressourcen (Hardware oder Software) innerhalb einer verteilten Anwendung. Dazu wird jeder Ressource vom Namensdienst ein eindeutiger Name verpasst. Bei Kenntnis dieses Namens erhält ein interessierter Client vom Namensdienst die Adresse (auch Referenz genannt) der gesuchten Ressource. Somit steht einer Kommunikation zwischen Client und Ressource nichts mehr im Weg.

Transaktionsverwaltung
Transaktionen spielen bei der Wahrung der Datenkonsistenz eine wichtige Rolle. Beispielsweise muss beim gleichzeitigen Zugriff mehrerer Anwender auf die gleichen Daten (z.B. bei einem Datenbankzugriff), das Ändern oder Speichern dieser Daten geregelt werden.
Deshalb werden derartige Aktionen innerhalb einer Transaktion erledigt, denn diese garantiert:

- Atomarität: Entweder alle ihre Aktionen sind erfolgreich, ansonsten keine („ganz oder gar nicht“).
- Konsistenz: Es werden nur konsistente Zustände hinterlassen.
- Isolation: Jede Transaktion ist von anderen Transaktionen abgeschirmt, d.h. sie behindern sich nicht gegenseitig.
- Dauerhaftigkeit: Das Resultat einer Transaktion ist dauerhaft.

Bei verteilten Transaktionen sind die Daten über mehrere Rechner verteilt, sodass eine verteilte Transaktionsverwaltung eingesetzt wird. Es existiert

hierzu ein Modell der Open Group, auf das in dieser Arbeit aber nicht eingegangen wird (siehe z.B. [Ham05]).

Persistenz
Unter Persistenz versteht man Funktionen zum Speichern von Daten auf nicht-flüchtigen Speichermedien (z.B. in Datenbanken oder Dateisystemen), sodass die Daten auch nach einer Programmbeendigung (und einem erneuten Programmstart) noch zur Verfügung stehen. Die Abbildung der Daten (Objekte) auf ein passendes Speicherformat kann auf unterschiedliche Weise erfolgen (z.B. durch objekt-relationale Mapper).

Sicherheit
Für die Einhaltung der Sicherheit innerhalb einer verteilten Anwendung kann sich die Middleware nicht auf das ihr zugrunde liegende Betriebssystem verlassen, d.h. die Middleware muss selbst Sicherheitsmechanismen zur Verfügung stellen [TvS03]. Gängige Mechanismen sind die Verschlüsselung, Authentifizierung, Autorisierung und das Auditing.

Die Programmierung von Middleware kann prinzipiell auf drei Arten erfolgen [Bak01]: Die erste Möglichkeit besteht darin, auf eine Funktionsbibliothek der Middleware zurückzugreifen. Bei der zweiten Variante wird eine so genannte Interface Definition Language (IDL) verwendet, mit der die Schnittstelle zur entfernten Komponente beschrieben wird, wobei das IDL-File noch auf eine gängige Programmiersprache (z.B. C oder Java) abgebildet werden muss (siehe auch Kapitel 2.2.1). Als letzte Möglichkeit kann die Verteilung durch die Programmiersprache und Laufzeitumgebung unterstützt werden (z.B. durch Javas Remote Method Invocation (RMI)).

Die Zahl der verfügbaren Middleware-Produkte ist mittlerweile sehr groß. Je nach Blickwinkel gibt es unterschiedliche Einteilungsmöglichkeiten von Middleware:
Hammerschall unterscheidet beispielsweise zwischen kommunikationsorientierter Middleware und anwendungsorientierter Middleware [Ham05]. Die Hauptaufgabe ersterer liegt in der Offerierung von Kommunikationsinfrastruktur. Hier spielen entfernte Methodenaufrufe oder das nachrichtenorientierte Modell zur asynchronen Kommunikation (nachrichtenorientierte Middleware) eine Rolle. Die zweite Kategorie erweitert die kommunikationsorientierte Middleware hinsichtlich Laufzeitumgebung und Diensten (eventuell auch um ein Komponentenmodell) und hat die Unterstützung der verteilten Anwendung zur Aufgabe.

In [Sie03] erfolgt eine Einteilung in peer-to-peer-Architekturen und zentralisierte Architekturen. Kennzeichen der P2P-Architektur ist das Vorhandensein von Diensten auf jedem Knoten, was den Ausfall eines einzelnen Knoten weniger problematisch macht. Die zentrale Architektur zeichnet sich durch einen zentralen Server aus, wobei Probleme bezüglich der Effizienz (Flaschenhals) und Fehlertoleranz auftreten können. Bei dieser Struktur gibt es keine direkte Kommunikation zwischen Sender und Empfänger.
Je nachdem ob die angebotenen Dienste im Mittelpunkt stehen oder aber die ausgetauschte Information ist von einer servicezentrierten (z.B. CORBA) oder einer informationszentrierten Middleware (siehe DDS) die Rede [Pri], [Kar05].
Weitere Beispiele für Middleware-Kategorien sind (z.T. schon erwähnt): transaktionale Middleware, nachrichtenorientierte Middleware, prozedurale Middleware, Objekt-Middleware, Komponenten-Middleware, Publish-Subscribe-Middleware, service-orientierte Middleware etc. [Pic04], [Bak01].

In den folgenden Kapiteln werden einige bekannte Middleware-Ansätze näher unter die Lupe genommen.

2.2.1 CORBA

Der Standard CORBA (Common Object Request Broker Architecture) wurde von der OMG (Object Management Group) entwickelt, um Interoperabilität, Portabilität und Unabhängigkeit bei der Implementation von verteilten Anwendungen und Systemen zu gewährleisten [TvS03], [OMGa], [COR].
Die Architektur (siehe Abbildung 2.5) ist objektorientiert ausgelegt, woher sich auch ihr Name OMA (Object Management Architecture) ableitet. Als Hauptbestandteil kann der so genannte ORB (Object Request Broker) angesehen werden, der dieser Middleware auch seinen Namen gibt. Der ORB koordiniert die Kommunikation zwischen Servern und Clients und verbirgt gleichzeitig die Verteilung und Heterogenität. Neben dem ORB enthält die Architektur vier weitere Elemente:

- Objektdienste (Services)
- Applikationsobjekte
- CORBA-Facilities (Sie enthalten mehrere Services und sind von keinem speziellen Bereich abhängig.)

- CORBA-Domains (Diese enthalten Services für bestimmte Bereiche, z.B. für Finanzen, Gesundheit etc.)

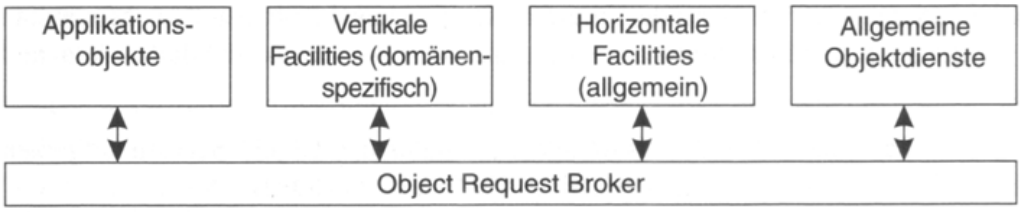

Abbildung 2.5: Die Architektur von CORBA [TvS03]

Die Interoperabilität zwischen CORBA-basierten Programmen von unterschiedlichen Herstellern wird durch das IIOP (Internet Inter-ORB-Protokoll) erreicht. Es handelt sich hierbei um ein Standardprotokoll für die Kommunikation zwischen Servern und Clients namens GIOP (General Inter-ORB-Protocol), das in Verbindung mit TCP/IP verwendet wird, woraus sich der Name IIOP ableitet. Ein IOP (Inter-ORB-Protokoll) besteht immer aus einem ORB-Protokoll und einem Transport-Protokoll (bzw. aus einer Abbildung auf ein Transport-Protokoll).
Für jeden Objekttyp und jeden Dienst muss bei CORBA eine Schnittstelle in OMG IDL (Object Management Group Interface Definition Language) definiert werden, wodurch eine Trennung der Schnittstelle von der Implementierung erreicht wird. Die IDL legt die Syntax für Methoden und deren Parameter fest. Für einen Methodenaufruf (am Server) muss ein Client eine entsprechende IDL-Schnittstelle verwenden, der Server verwendet die gleiche Schnittstelle. IDL-Spezifikationen müssen auf unterschiedliche Programmiersprachen abgebildet werden. Derzeit existieren Regeln für C, C++, Java, COBOL, Smalltalk, Ada, Lisp, Python und IDLscript.

Abbildung 2.6 zeigt den allgemeinen Aufbau eines CORBA-Systems, das in der Regel aus mehreren Clients und Servern besteht.
Die zentrale Rolle (sowohl auf Server- als auch auf Client-Seite) kommt - wie schon erwähnt - dem ORB zu, der durch eine Schnittstelle vom Client bzw. Server abgetrennt ist. Um einen entfernten Prozeduraufruf wie einen lokalen erscheinen zu lassen, gibt es sogenannte Stubs, die das Verpacken bzw. Entpacken sowie das Senden von Nachrichten (transparent) erledigen (siehe Kapitel 2.1). Dabei ist ein Proxy ein Stub auf Client-Seite, ein Skeleton ist ein serverseitiger Stub. Das Dynamic Invocation Interface (DII)

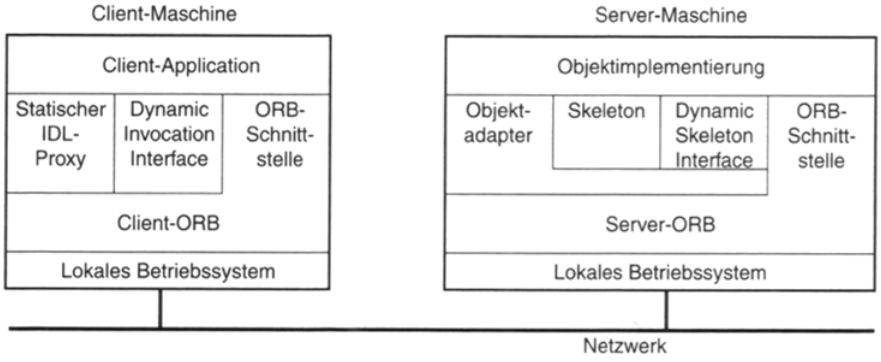

Abbildung 2.6: Beispiel eines CORBA-Systems [TvS03]

erlaubt eine dynamische Erkennung von Objekt-Schnittstellen sowie eine entsprechende Aufrufanforderung zur Laufzeit. Das Dynamic Skeleton Interface (DSI) ist das serverseitige Äquivalent. Der Objektadapter sorgt dafür, dass ankommende Anforderungen an das richtige Objekt weitergeleitet werden.

CORBA stellt eine Vielzahl an Diensten (Services) zur Verfügung:

- Namensdienst (INS ... Interoperable Naming Service): übernimmt die Identifikation der Objekte über ihre Namen
- Transaktionsdienst (OTS ... Object Transaction Service): ermöglicht eine verteilte Transaktionsverwaltung (auch verschachtelte Transaktionen)
- Handelsdienst (Trading-Service): Identifikation der Objekte über ihre Eigenschaften
- Ereignisdienst (Event-Service)
- Notification Service
- Etc.

Neben dem allgemeinen Standard sind auch spezielle CORBA-Versionen für eingebettete Systeme (minimumCorba bzw. CORBA/e, siehe Kapitel 3.3) und Echtzeitfähigkeit (RT-CORBA, siehe nächstes Kapitel) verfügbar.

2.2.2 RT-CORBA (Real-Time CORBA)

In Erweiterung zu CORBA bietet RT-CORBA (aktuelle Version: 1.2) Unterstützung bei der Ressourcenverwaltung und der Ende-zu-Ende-Vorhersagbarkeit an, was einer Ausweitung der QoS-Kontrolle gleichkommt [TAO], [OMGa] (bezüglich der RT-CORBA Spezifikation siehe [OMGg]). Es existieren Mechanismen zur Konfiguration folgender Ressourcen bzw. zur Einflussnahme darauf:

- Prozessorressourcen (CPU-Ressourcen): durch Verwendung von Thread Pools, Prioritäten-Modellen und Mutexes (Abkürzung für engl. Mutual Exclusion ...„gegenseitiger Ausschluss“: dienen der Vermeidung von Zugriffskonflikten und Inkonsistenzen)
- Kommunikationsressourcen: Definition von Protokollrichtlinien, Explizites Binden (engl. Explicit Binding), d.h. die Applikation bestimmt den Zeitpunkt des Bindens
- Speicherressourcen: Puffern von Requests, Festlegung der Größe von Thread Pools

Es werden zwei Arten von Prioritäten unterschieden: CORBA-interne (von 0 bis 32767) und Endsystem-spezifische (z.B. unterschiedlicher Prioritätstyp zwischen Windows XP und Solaris) [SK00].
Außerdem gibt es zwei unterschiedliche Prioritätsmodelle (siehe Abbildung 2.7): Beim server declared Modell wird die Priorität von vornherein durch den Server festgelegt und in der Objektreferenz vermerkt. Das client propagated Modell lässt den Client die Aufrufpriorität festlegen und damit - im Falle einer Rückmeldung - auch die Priorität der Antwort. Jedes Endsystem zwischen Client und Server bildet dazu die Priorität auf den eigenen Prioritätstyp ab.

Ein Server kann die Priorität auch dynamisch festlegen (Prioritätstransformation, engl. priority transform), wodurch die Priorität von der Art des Aufrufs abhängig wird.
Generell können Zeiteinschränkungen der Anwendung auf Prioritäten abgebildet werden (Offline-Analyse).
Um auch multithreaded Systeme entwickeln zu können wurde bei RT-CORBA ein Threadpool-Modell eingeführt. Der Entwickler kann Threadpools vorallozieren und bestimmte Attribute definieren (z.B. vorgegebene Priorität, Puffer für Requests: ja/nein). Es werden zwei Modelle unterschieden: ein Threadpool-Modell mit und ohne lanes: Beim Modell ohne lanes wird die Anzahl der Threads zu Beginn festgelegt, weshalb man hier auch von

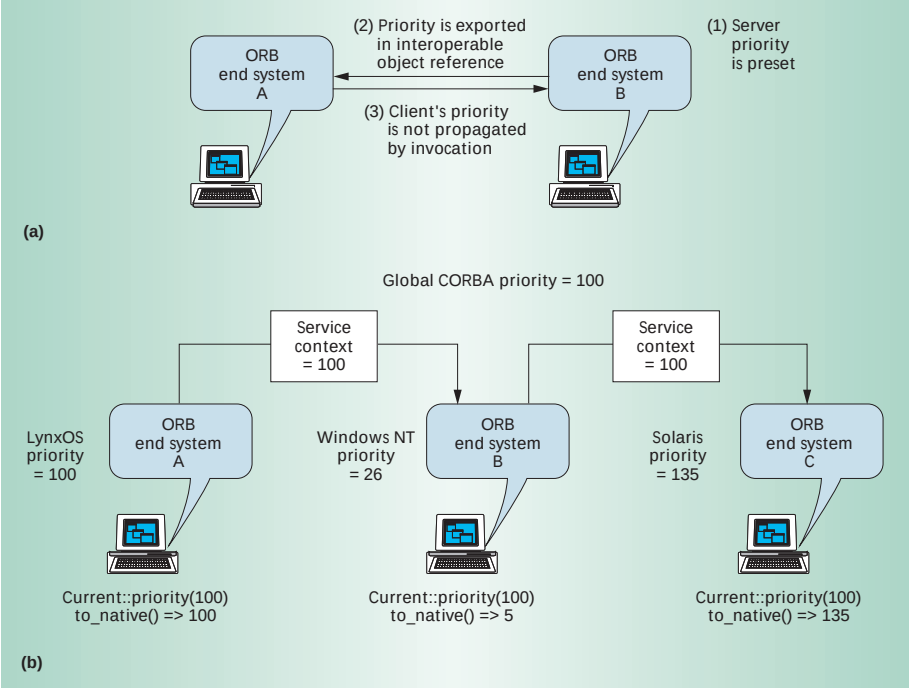

Abbildung 2.7: Prioritätenmodelle bei RT-CORBA [SK00]

statischen Threads spricht. Außerdem wird die maximale Anzahl an Threads, die dynamisch erzeugt werden können, sowie die Priorität sämtlicher Threads fixiert. Diese Prioritäten können je nach Priorität des anfragenden Clients geändert werden.
Beim Modell mit lanes kommt es zu einer Gruppierung von Threads innerhalb des Pools. Alle Threads innerhalb einer lane haben die gleiche Priorität, wodurch eine Prioritäteninversion vermieden werden kann (siehe Abb. 2.8). Jede lane hat eine unterschiedliche Anzahl an statischen und dynamischen Threads. Das Ausborgen von Threads aus anderen lanes mit niedrigerer Priorität ist möglich, wobei die Priorität für die Dauer des Threads angehoben wird.

Durch ein Scheduling Service auf Applikationsebene, das so genannte Global Scheduling Service, können Anforderungen betreffend die Abarbeitung von Operationen (z.B. worst-case execution time) angegeben werden.

RT-CORBA erlaubt die Wahl des unterliegenden Kommunikations-Protokolls. Es spezifiziert ein Interface zur Festlegung von Eigenschaften

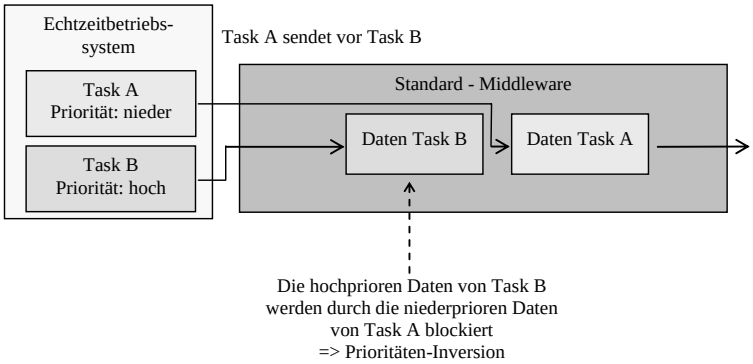

Abbildung 2.8: Prioritäteninversion [WB05]

des ORB- und Transportprotokolls, z.B. kann die Pufferlänge im Sender und Empfänger bei TCP geändert werden, was einen direkten Einfluss auf den Durchsatz hat. (TCP ist momentan das einzige Protokoll, dessen Protokolleigenschaften spezifiziert sind.) Die Auswahl und Konfiguration von Protokollen kann auf Server-Seite und auf Client-Seite erfolgen (Unterscheidung zwischen ServerProtocolPolicy und ClientProtocolPolicy).
Beim Expliziten Binden gibt es zwei Arten von Verbindungen: Priority-Banded Connections und Private Connections: Bei ersterem gibt der Client die Priorität der Verbindung an, bei den Private Connections kann eine Verbindung erst nach Beantwortung einer Anfrage für eine neue Anfrage benutzt werden (keine Multiplex-Verbindungen).

RT-CORBA baut auf CORBA 2.2 und der Messaging Specification auf.
Es existieren mehrere Implementierungen, die bekannteste ist wohl TAO. Für eingebettete Systeme eignen sich u.a. OpenFusion e*ORB, ORBExpress RT, RTZen oder ROFES.
Eine auf Java und dessen Echtzeiterweiterung (Real-Time Specification for Java, RTSJ) basierende Implementierung ist Zen bzw. RTZen [Zen], [RTZ], [TDP04].

2.2.3 DCOM

Das Distributed Component Object Model (DCOM) wurde von Microsoft entwickelt und durch die Active Group spezifiziert [Micc], [Micd]. Obwohl es nicht mehr weiterentwickelt wird, wird es nach wie vor unterstützt (z.B. Feature von Windows Vista). DCOM ist von der Programmiersprache

unabhängig.
Es existieren Versionen für fast alle Windowsanwendungen (z.B. DCOM98 Version 1.3 für Microsoft Windows® 98, oder Versionen für Windows NT® 4.0 und Windows® 95). Neben diesen gibt es aber auch Implementierungen für UNIX-Plattformen (siehe [Sof]).
Das DCOM-Konzept basiert auf dem COM-Modell, indem es eine Erweiterung von diesem darstellt (siehe Abbildung 2.9).

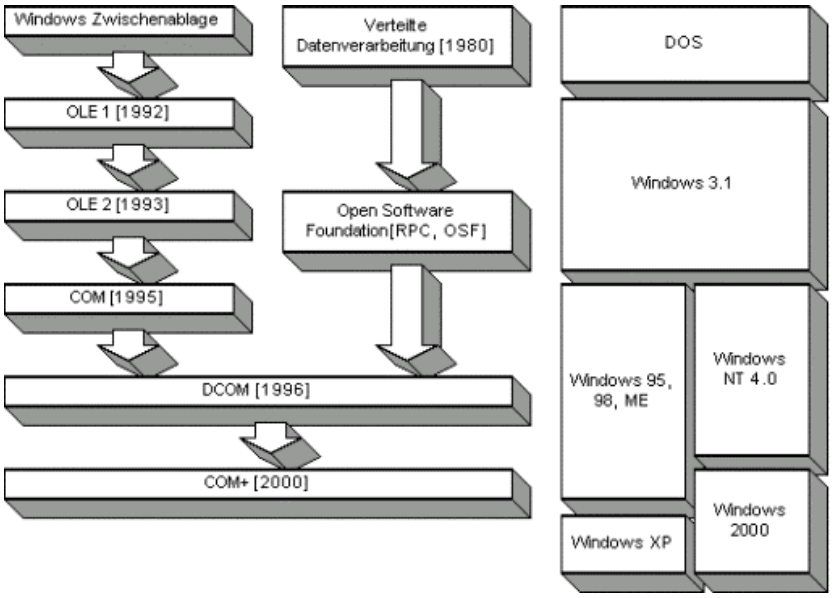

Abbildung 2.9: DCOM als Middleware [Rei04]

In COM wird definiert, wie ein Client eine Methode innerhalb einer Komponente aufruft. DCOM ermöglicht außerdem die Kommunikation zwischen Objekten auf verschiedenen Maschinen. Das beinhaltet u.a. den Verbindungsaufbau und die Erzeugung neuer Instanzen (Aktivierungsmechanismen).
Der Zugriff auf Objekte erfolgt über Interfaces. Neben den Standard-Interfaces gibt es auch so genannte Custom-Interfaces.
Die Identifikation von Objektklassen erfolgt allgemein über globally unique identifiers (GUIDs), bei bestimmten Klassen spricht man auch von Class IDs (CLSIDs).
DCOM verwendet für die Kommunikation zwischen DCOM-Komponenten ORPC (Object RPC), eine Erweiterung des RPC-Protokolls. Die Kommunikation ist in beide Richtungen möglich (peer-to-peer oder Client/Server).

Die COM Run-Time auf Client- und Komponentenseite stellt objektorientierte Dienste zur Verfügung und benutzt einen Security-Provider sowie DCE RPC. DCE (Distributed Computing Environment) ist ein Standard der Open Software Foundation für verteilte Anwendungen.
Jede Komponente besitzt einen Referenzzähler, der die Anzahl an Verbindungen zu unterschiedlichen Clients erfasst. Wenn der Zähler den Wert 0 erreicht hat, wird die Komponente freigegeben.
Eine Besonderheit von DCOM ist das Pinging-Protokoll, das Fehlertoleranz erlaubt: jeder Client sendet in periodischen Abständen Ping-Nachrichten. Wenn eine Komponente in drei aufeinander folgenden Perioden keine Ping-Nachricht erhält (numPingsToTimeOut = 3), gilt die Verbindung zwischen Client und Komponente als unterbrochen und der Referenzzähler der Komponente wird um eins reduziert. Gängige DCOM-Implementierungen verwenden als Ping-Periode (pingPeriod) zwei Minuten. Die Parameter pingPeriod und numPingsToTimeOut können allerdings nicht geändert werden.
DCOM unterstützt Thread-Pools. Es wird zwischen Single-Threaded Apartments (STA) und Multithreaded Apartments (MTA) unterschieden. Bei STA befindet sich jedes Objekt in einem Thread. Neben einer Nachrichten-Pumpe ist auch ein verstecktes Fenster für die Kommunikation (Synchronisation) notwendig.
Bei MTA verwenden die RPCs den von der RPC-runtime zugeteilten Thread. In diesem Fall sind keine Nachrichten-Pumpe und kein verstecktes Fenster notwendig, dafür aber entsprechende Synchronisationsmechanismen.
Es gibt mehrere Mechanismen zur Last-Balancierung (engl. load balancing). Sicherheitseinstellungen werden entweder über einen Registry-Eintrag oder durch die Applikation durchgeführt.
DCOM unterstützt jedes Transport-Protokoll (z.B. TCP/IP, UDP, IPX/SPX oder NetBIOS) und ist eventfähig.
DCOM findet in der Automatisierungstechnik eine weite Verbreitung, zumal neben dem OPC[3]-Standard auch PROFINET[4] DCOM verwendet.
Für DCOM sprechen auch die geringen Kosten der benötigten Entwicklungstools (z.B. Visual C++ oder Visual Basic). Allerdings ist der eigentliche Preis für DCOM nur für Windows-Plattformen niedrig.

[3] OPC stand früher für OLE (Object Linking and Embedding) for Process Control und ist heute die Abkürzung für Openness, Productivity, Collaboration [Wike]. Es handelt sich hierbei um eine standardisierte Software-Schnittstelle, die vorwiegend in der Automatisierungstechnik für eine herstellerunabhängige Kommunikation eingesetzt wird.

[4] PROFINET ist ein Industrial Ethernet Standard, der von PROFIBUS International und dem INTERBUS Club unterstützt wird [Wikg].

2.2.4 .NET

So wie DCOM wurde auch die .NET-Plattform von Microsoft entwickelt [Ham05]. (COM bzw. DCOM ist allerdings nicht die Basis, sondern existiert parallel.) Es handelt sich hierbei nur um einen teilweise offenen Standard.
Die Plattform, die Interoperabilität mit anderen Plattformen ermöglicht (durch offene Webservice-Standards), besteht aus dem .NET-Framework inklusive Laufzeitumgebung (CLR, Common Language Runtime = virtuelle Maschine, VM) und einer Klassenbibliothek (FCL, Framework Class Library), sowie Diensten und Servern (siehe Abbildung 2.10). .NET setzt auf dem Betriebssystem auf, wobei dies nicht notwendigerweise Windows sein muss (wie jedoch in den meisten Fällen üblich), sondern auch Linux sein kann.

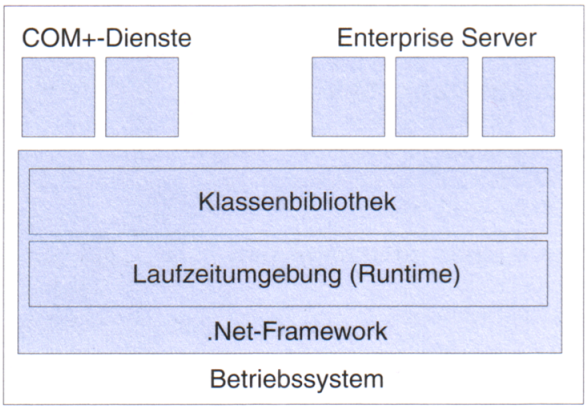

Abbildung 2.10: .NET-Plattform [Ham05]

Das derzeit aktuelle .NET Framework 3.0 erweitert das .NET Framework 2.0 um folgende Komponenten: Windows Workflow Foundation, Windows Communication Foundation (WCF), Windows CardSpace und Windows Presentation Foundation (siehe Abbildung 2.11).

Wesentliche Komponenten von .NET Framework 2.0 sind ASP.NET, ADO.NET, Windows Forms und System.XML.
Durch die WCF (auch bekannt unter Indigo) werden die in .NET Framework 2.0 existierenden Kommunikationstechnologien (ASP.NET Web Services, .NET Remoting, Enterprise Services, System.Messaging, Web Services Enhancements (WSE)) kombiniert bzw. unter einem gemeinsamen API

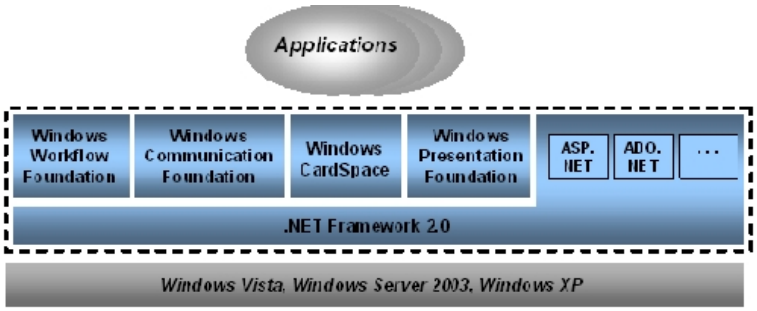

Abbildung 2.11: .NET Framework 3.0 [Mice]

zusammengefasst [Mick], [Micf], [Micb], [Mich].
Damit ein Client Zugriff auf einen Dienst hat, muss dieser Dienst eine Schnittstelle anbieten. Ein Dienst hat mehrere Endpunkte, die jeweils einen Vertrag (engl. contract) zur Beschreibung der möglichen Operationen, eine Bindung (engl. binding), die angibt, wie die Operationen „konsumiert“ werden können (Art des Protokolls, Art der Sicherheit, Encoder: z.B. Text oder binär), und eine Adresse (engl. address) für die Position des Endpunkts enthalten.
Für eine Kommunikation stehen folgende Verfahren zur Verfügung:

- RPCs in Form synchroner Aufrufe, bei denen auf die Antwort gewartet wird
- OneWay-Aufrufe, bei denen nicht auf eine Antwort gewartet wird (z.B. für die Ereignisweitergabe)
- Asynchrone Kommunikation mit XML-Meldungen
- SOAP-Meldungen: bei SOAP (Simple Object Access Protocol) handelt es sich um ein Netzwerk-Standardprotokoll basierend auf HTTP und TCP

Es stehen eine Reihe von WS-*-Spezifikationen zur Verfügung: Für eine zuverlässige Übertragung gibt es beispielsweise die Option WS-ReliableMessaging, die Spezifikation WS-AtomicTransaction ermöglicht verteilte, herstellerübergreifend interoperable Transaktionen, WS-Security sorgt für Sicherheit.

Durch die CLR werden Anwendungen in beliebigen Programmiersprachen möglich (Programmiersprachenunabhängigkeit). Die Sprache wird durch den jeweiligen Compiler, der Bestandteil des Frameworks ist, auf die Common Intermediate Language (CIL) (früher Microsoft Intermediate Language,

MSIL) - eine binäre objektorientierte Sprache - abgebildet, wodurch Interoperabilität ermöglicht wird (siehe Abbildung 2.12).
Unter einer Assembly versteht man in .NET eine Datei mit ausführbarem Code (im Intermediate Language Format) und ihrem Manifest (das sind zusätzliche Infos bezüglich der enthaltenen Klassen, der Anwendung etc.). Diese Assemblies werden durch den JIT-Compiler (Just-in-Time-Compiler) in Maschinencode übersetzt, wodurch die CLR von der Hardware entkoppelt wird. Der von der CLR ausgeführte Code wird als Managed Code bezeichnet. Wichtige Aufgaben der CLR sind die Speicher-, Ressourcen und Threadverwaltung (z.B. Garbage Collection = automatische Speicherbereinigung) sowie die Fehlerbehandlung (Exception-Handling). Außerdem enthält sie ein einheitliches Typsystem (Common Type System, CTS), das somit nicht mehr im Compiler enthalten sein muss (Interoperabilität).
Da durch die CLR eine Vorhersagbarkeit der WCET (Worst-Case Execution Time) nicht mehr möglich ist, ist mit .NET die Echtzeitfähigkeit nur schwer garantierbar.

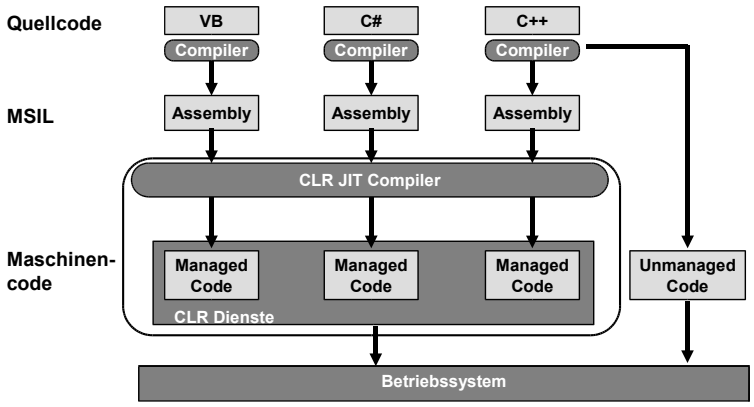

Abbildung 2.12: Sprachintegration in .NET [Str]

Die Klassenbibliothek enthält eine Fülle an Klassen und ist durch Namensräume gegliedert. Diese Namensräume sind wiederum in Bäumen organisiert, wobei die Namensräume „System“ und „Microsoft“ die Wurzeln darstellen. Der Namensraum *System* enthält Microsoft-unabhängige Dienste, z.B. Events, Exceptions etc.

Dienste bestehen in Form von Enterprise Servern (z.B. SQL-Server, BizTalk Server), COM-Diensten (z.B. Microsoft Transaction Server, MTS) und

Subsystemen (z.B. ADO.Net für Persistenz).

Für die Applikationsentwicklung steht Visual Studio .NET zur Verfügung.

2.2.5 Java Platform, Enterprise Edition (Java EE)

Bei der Java Platform, Enterprise Edition - früher als J2EE (Java 2 Platform, Enterprise Edition) bezeichnet - handelt es sich um einen offenen Standard einer Middleware-Plattform (Softwarearchitektur), der die Erstellung von verteilten Anwendungen ermöglicht [Ham05], [Jav]. Die aktuelle Version ist die Java EE 5 Plattform.
Der Standard verwendet einen Java- und komponentenbasierten Ansatz. Das Komponentenmodell trägt den Namen EJB (Enterprise JavaBeans), wobei mit Bean eine serverseitige Komponente gemeint ist, die mehrere Objekte beinhalten kann. Ein so genannter Container stellt die Laufzeitumgebung für die Komponenten dar. Die Kommunikation funktioniert über Java RMIs und das CORBA-Protokoll IIOP.
Es gibt vier verschiedene Container, die unterschiedliche Komponenten zur Realisierung von Anwendungen beinhalten:

- EJB-Container: beinhaltet EJBs, laufen auf Server-Seite
- Web-Container: mit Servlets für Webanwendung, auf Server-Seite
- Application-Client-Container: unterstützt Clients bei der Kommunikation mit EJBs, auf Client-Seite
- Applet-Container: mit Applets, auf Client-Seite

Die Grundlage von Java EE ist Java Platform, Standard Edition (Java SE) - früher als Java 2 Platform Standard Edition (J2SE) bekannt - die als Basislaufzeitumgebung agiert.
Bei den Servern wird zwischen Webservern und Anwendungsservern, bei den Java EE Clients wird zwischen Webclients und Anwendungsclients unterschieden
Bezüglich der EJBs wird zwischen vier verschiedenen Typen unterschieden: Es gibt Stateless Session Beans (diese haben keinen Zustand), Stateful Session Beans (sind über mehrere Methodenaufrufe innerhalb einer Sitzung aktiv), Message Driven Beans (besitzen keinen Zustand und dienen der asynchronen Kommunikation) und Entity Beans (für Persistenz, wobei man je nach Verantwortung über die Datenhaltung zwischen Container-managed

Persistence und Bean-managed Persistence unterscheidet). Mehrere Beans ergeben eine Anwendung, d.h. die Beans besorgen die Abläufe innerhalb einer Anwendung bzw. innerhalb einer Sitzung. Message Driven Beans und Session Beans sind Schnittstellen, die Aufrufe entgegennehmen (sowohl synchrone als auch asynchrone). Entity Beans sind für die Datenhaltung verantwortlich.

Dienste werden in Form von Schnittstellen zur Verfügung gestellt. Einige Beispiele sind:

- JNDI (Java Naming and Directory Interface): Namensdienst
- JTS (Java Transaction Service): Unterscheidung zwischen Container- und Bean-managed Transaktionen
- JCA (J2EE Connector Architecture): Kopplung der J2EE-Plattform mit Informationssystemen (z.B. SAP), durch unterschiedliche Kontrakte und speziellen Adapter möglich
- JMX (Java Management Extension): Architektur zum Ressourcen-Management

Für die Anwendungsentwicklung existieren ein Musterkatalog (mit 21 Mustern) sowie eine Musterarchitektur (J2EE BluePrint)
Grundsätzlich ist Java EE für Real-Time-Anwendungen nicht geeignet, da der Garbage Collector kein deterministisches Verhalten erlaubt, und es an einer adäquaten Thread-Unterstützung mangelt (mehr dazu in Kapitel 3.3).

2.2.6 DDS

Der „Data Distribution Service for Real-Time Systems“ (DDS) ist eine datenzentrierte, auf Publish-Subscribe beruhende Echtzeit-Infrastruktur, die von verteilten Applikationen zur Datenkommunikation verwendet werden kann. Es handelt sich dabei um einen Standard der OMG (Object Management Group), der unter der URL `http://www.omg.org/` heruntergeladen werden kann [Sie06b], [PC05].
Die im Mittelpunkt der Übertragung stehenden Datenobjekte werden mittels Topics identifiziert und in einem gemeinsamen virtuellen globalen Datenraum abgelegt (siehe Abbildung 2.13). Topics stellen die Verbindungseinheit zwischen Publishern und Subscribern dar: Publisher und Subscriber geben ihre Sende- bzw. Empfangsabsicht durch Registrierung der gewünschten Topics bei der Middleware bekannt. Erst dann können die einzelnen Knoten

auf die Datenobjekte aus dem globalem Datenraum durch entsprechende Lese- und Schreiboperationen zugreifen, bzw. können sich die Datenabnehmer auch per Rückruf über bereitliegende Daten verständigen lassen. Diese Trennung der Absicht (der Applikation) des Informationszugangs und des eigentlichen Informationsaustausches (durch read- und write-Operationen) ermöglicht es der Middleware, den Informationsaustausch bestmöglich vorzubereiten.

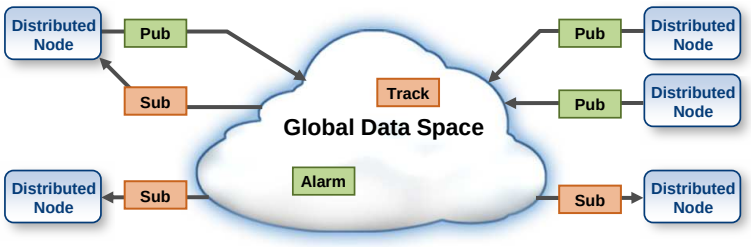

Abbildung 2.13: Publish/Subscribe bei DDS [PCH05]

Der Datentyp des Topics wird mittels IDL spezifiziert. Während der Titel für mehrere Instanzen gleich sein kann, um eine gleiche Behandlung dieser Instanzen zu ermöglichen, spezifiziert das DDS auch einen sogenannten Key (das sind mehrere Datenfelder), der für jede Instanz einzigartig ist, wodurch auch unterschiedliche Instanzen mit gleichem Topic voneinander unterschieden werden können. Dies ist u.a. im Hinblick auf die Skalierbarkeit interessant.
Wie bereits angedeutet macht DDS die Programmierung der Kommunikation (Datenverpacken und -auspacken, Adressierung etc.) überflüssig. Ganz anders verhält es sich mit der QoS der Kommunikation. Hier hat der Applikationsentwickler viele Möglichkeiten der Einflussnahme, was als weiterer ganz wesentlicher Vorteil genannt werden muss. Beispielsweise können die maximale Sende- und Empfangsrate des Publishers bzw. Subscribers oder aber auch die Gültigkeit von Nachrichten spezifiziert werden. Eine genauere Beschreibung der QoS beim DDS folgt in Kapitel 3.3.

Die DDS-Spezifikation beschreibt zwei unterschiedliche APIs [PCFW05], [OMGa]: die niedere DCPS (Data-Centric Publish-Subscribe) - Schicht dient der Kommunikation und beinhaltet auch die oben erwähnte QoS-Unterstützung. Die höhere und optionale DLRL (Data Local Reconstruction Layer) - Schicht soll den Zugang zur DCPS-Schicht ermöglichen und gleichzeitig erleichtern.

Zur Verdeutlichung der wesentlichen DDS-Konzepte soll Abbildung 2.14 dienen. Zu sehen sind die primären Entities der DCPS-Schicht, nämlich:

- Domain: Sie dient der Verbindung von miteinander kommunizierenden Applikationen, d.h. nur die Publisher und Subscriber innerhalb einer Domäne können miteinander interagieren.
- Domain Participant: Durch diese Entity wird die Zugehörigkeit einer Applikation zu einer Domäne angedeutet. Mit ihr können allgemeine Einstellung (z.B. QoS-Parameter) bezüglich aller Data Writers, Data Readers, Publisher und Subscriber innerhalb einer Domäne vorgenommen werden.
- Data Writer: Er dient der Applikation für Schreiboperationen, wobei er einen bestimmten Datentyp repräsentiert.
- Publisher: Er verteilt die Daten (unterschiedlichen Datentyps), die er von den ihm zugeordneten Data Writers bekommt - er ist somit ein Container für Data Writers, über den der Applikationsentwickler allgemeine Einstellungen, die alle Data Writers betreffen, tätigen kann.
- Data Reader: Ihm ist ein bestimmter Datentyp zugeordnet. Über ihn hat die Applikation Zugriff auf Daten.
- Subscriber: Er empfängt die publizierten Daten (unterschiedlichen Typs) und stellt sie der Applikation über die ihm zugeordneten Data Readers zur Verfügung.
- Topic: siehe oben

Ein DDS-Standardprotokoll, das auf dem Real-Time Publish-Subscribe (RTPS) Wire-Protocol (IEC PAS 62030) basiert, soll in Zukunft Interoperabilität zwischen unterschiedlichen DDS-Implementierungen garantieren (siehe auch [THJ+05] und [PCB02]).
Es existieren einige Implementierungen des DDS-Standards. Die bekanntesten sind RTI (Real-Time Innovations) DDS (früher NDDS, Network Data Distribution Service) und Splice von Thales, wobei beide kommerziell erhältlich sind. Außerdem stehen einige Open-Source-Produkte zur Verfügung (z.B. DDS for TAO). Eine Open Source Implementierung des RTPS-Protokolls mit dem Namen ORTE (OCERA Real-Time Ethernet) ist unter `http://www.ocera.org/download/components/WP7/ethdev-0.2.2.html` erhältlich.

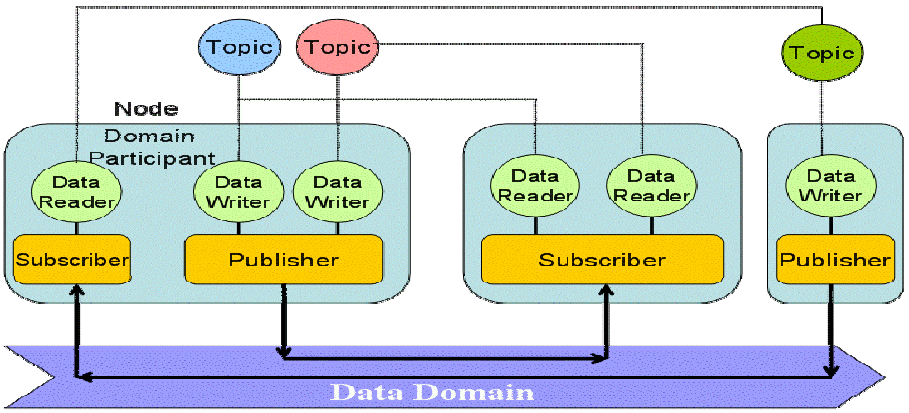

Abbildung 2.14: DDS Entities [PCFW05]

2.2.7 OSA+

OSA+ steht für Open Service Architecture Platform for Universal Services und ist eine service-orientierte Middleware, die an der Universität Karlsruhe entwickelt wurde [PB06], [Pic04], [WB05].
Die Funktionalität wird durch Services zur Verfügung gestellt, die über Interfaces genutzt werden können. Der Zugriff auf die Schnittstellen ist plattform- und sprachenunabhängig und erfolgt über so genannte Jobs, welche aus einer Anforderung oder Auftrag (engl. order) und einem Ergebnis (engl. result) bestehen (siehe Abbildung 2.15). Die mittels Jobs realisierte Kommunikation zwischen den Diensten kann sowohl synchron als auch asynchron sein. Die Verbindung zwischen den einzelnen Services wird über die Plattform hergestellt, wobei ein Service die kleinste Einheit und zugleich die einzige Einheit zur Ausführung von Aufgaben darstellt. Services werden über einen Servicenamen und eine Serviceversion identifiziert. Zu ihrer Speicherung existiert ein zentraler Service Speicher (Repository).

OSA+ präsentiert sich in Form einer Mikrokern-Architektur. Die Basis ist die Plattform (Mikrokern), welche ein Minimum an Funktionalität beinhaltet und schrittweise um Services erweitert wird (siehe Abbildung 2.16).

Die Kern-Plattform ist sowohl hardware- als auch betriebssystemunabhängig (umgebungsunabhängig) und für sich alleine ablauffähig. Zu ihren Aufgaben zählen:

- Service-Management: Hinzufügen und Entfernen von Services

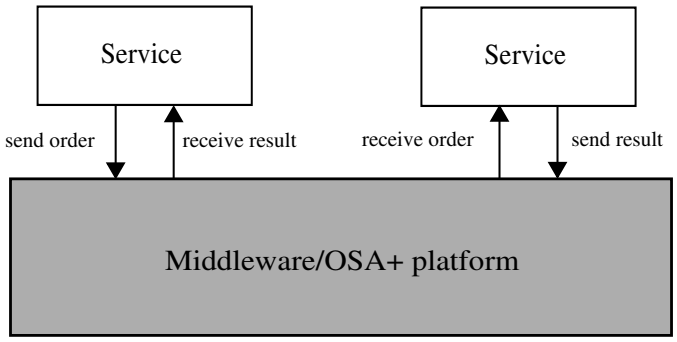

Abbildung 2.15: OSA+ Plattform [Pic04]

- Lokales Job-Management: Interaktion zwischen lokalen Services über Jobs
- QoS-Unterstützung
- Bereitstellung eines API zur Manipulation von Services, Jobs und der QoS.

Wie aus Abbildung 2.16 ersichtlich gibt es neben den Anwendungsdiensten noch zwei weitere Arten von Services: die Basisdienste (engl. basic services) und die Erweiterungsdienste (engl. extension services). Die Basisdienste stellen eine Verbindung zu Hardware, Betriebssystem und Kommunikationssystem her. Diese Anpassung an die Umgebung wird u.a. durch folgende Dienste erreicht: Prozessdienste (Umsetzung von Services mittels Threads und Prozessen), Ereignisdienste, Speicherdienste (Speicherallozierung für Echtzeit), Kommunikationsdienste (Verwendung von Protokollen). Außerdem stellt das ARS (Address Resolution Service) einen Basisdienst dar, der es dem Benutzer erlaubt, lokale als auch entfernte Services in gleicher Weise zu lokalisieren (Ortstransparenz) [Pic04].
Die Erweiterungsdienste unterstützen funktionale Erweiterungen (Logging Service, Rekonfigurationsdienst, Verschlüsselungsdienst).

Durch Kombination des Mikrokerns mit unterschiedlichen Services können mannigfache Konfigurationen realisiert werden. Die alleinige Verwendung des Mikrokerns stellt eine minimale Funktionalität (Beschränkung auf lokale, sequentielle Dienste) und maximale Ressourcenbeschränkung dar. Die Kombination des Mikrokerns mit dem Prozessdienst schafft eine Verbindung zum Betriebssystem und erlaubt lokale sequentielle und parallele

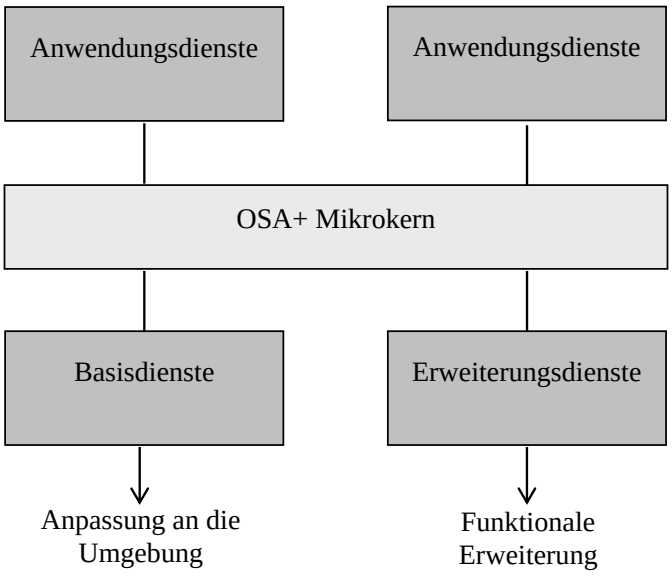

Abbildung 2.16: OSA+ Dienste [WB05]

Dienste. Bei Verknüpfung des Mikrokerns mit dem Prozessdienst und dem Kommunikationsdienst können sowohl lokale als auch globale Dienste (sequentielle und parallele) genutzt werden.

Um mit OSA+ Echtzeit erzielen zu können, müssen die unterliegende Hardware, sowie das Betriebs- und Kommunikationssystem echtzeitfähig sein. Dabei nutzen die Services die Echtzeit der Umgebung (z.B. der Prozessdienst nutzt die Scheduling Mechanismen des Betriebssystems, oder der Speicherdienst nutzt die Speicherallozierung usw.).

Es muss darauf hingewiesen werden, dass im Rahmen dieser Arbeit nur eine sehr eingeschränkte Auswahl an Middleware-Konzepten abgehandelt werden kann. Es handelt sich bei diesen in erster Linie um bekannte bzw. weit verbreitete Lösungen, die somit einigermaßen gut erprobt sind und meist ein weites Spektrum an Anwendungen bedienen.
Bezüglich weiterer Middleware-Produkte muss auf die Literatur verwiesen werden: MiRPA beispielsweise wird innerhalb von Robotersteuerungen eingesetzt und ist eine leichtgewichtige, echtzeitfähige Middleware [DFK04]. Bei Mires handelt es sich um eine nachrichtenorientierte Middleware, die auf dem Publish-Subscribe-Prinzip beruht und für einen Einsatz in

Sensornetzwerken konzipiert wurde [SGV+05]. OSACA wiederum ist ein gemeinsames europäisches Projekt industrieller und öffentlicher Einrichtungen zur Entwicklung einer offenen, herstellerunabhängigen Systemarchitektur [OSA], [SL96]. In [RDN06], [RDN05] und [RDNS04] wird die Middleware DOME (Distributed Object Model Environment) vorgestellt, die speziell für automationsspezifische, objektorientierte Applikationen ausgelegt und eng mit IEC 61499 verbunden ist.[5]
Schließlich sei auf das europäische Forschungs- und Entwicklungsprojekt SIRENA (Service Infrastructure for Real Time Embedded Networked Applications) hingewiesen, das die Schaffung eines serviceorientierten Frameworks zur Entwicklung von verteilten Anwendungen innerhalb eingebetteter, echtzeitfähiger Systeme zum Ziel hatte [Inf], [GD].

2.3 Quality of Service (QoS)

Der Begriff Quality of Service (QoS) ist ein sehr strapazierter [McK03]. Im Zusammenhang mit Netzwerken versteht man darunter Mechanismen, die die Nutzung von Netzwerkressourcen regeln. Beispielsweise können durch die Einführung eines Prioritätenmodells die Zuteilung von Bandbreite oder aber auch die Zeitverzögerungen von Datenströmen beeinflusst werden. Eine derartige Einflussnahme ist etwa bei der Multimediakommunikation essentiell, da bei einem mangelnden Ressourcen-Management die Performance und damit die Benutzerfreundlichkeit leiden.

Die ISO (International Standards Organization) definiert QoS wie folgt:
QoS is a „*set of qualities related to the collective behavior of one or more objects*“ [Int98].
Bei dieser Definition steht nicht die Art der Funktionalität im Vordergrund, sondern wie sie zur Verfügung gestellt wird. Das heißt, die Frage nach dem

[5] DOME mag zwar sehr vielversprechend klingen, es muss jedoch betont werden, dass bisher keine QoS-Mechanismen implementiert wurden (eine Priorisierung der Nachrichten bzw. Threads ist geplant), und auch der Speicherbedarf erscheint für die IAT zu hoch (ca. 4 MB für die Runtime, mit applikationsabhängigem RAM für dynamische Anforderungen ca. 16 MB, vgl. Kapitel 3.2.3). Außerdem ist DOME zum momentanen Zeitpunkt nicht verfügbar - mit einer Veröffentlichung ist frühestens Ende 2008 zu rechnen, wobei die Lizenzbedingungen noch nicht ganz geklärt sind (eine Version wird unter der GNU General Public License (GPL) stehen, welche dem universitären Umfeld bzw. der Evaluierung dient). Da DOME mit einer eigenen Runtime ausgestattet ist, würde die Verwendung dieser Middleware auch den Einsatz ihrer Runtime bedingen.

wie überwiegt das *was*.
Eine derartige Unterscheidung wird auch durch die Bezeichnungen QoS-Eigenschaften und QoS-Mechanismen getroffen. Erster Begriff steht für das *was* und betrifft konkrete Eigenschaften, die von einer (verteilten) Anwendung erwartet werden (z.B. Sicherheit). Diese Eigenschaften können letztendlich auf unterschiedliche Weise realisiert werden (z.B. über ein bestimmtes Codierungsverfahren).

QoS Kategorien
Bezüglich der Modellierung von QoS (und fehlertoleranten Systemen) sei eine Spezifikation der OMG (Object Management Group) erwähnt [OMGh]. Als Modellierungssprache wird die - ebenfalls von der OMG entwickelte und spezifizierte - UML (Unified Modelling Language) verwendet. Das Kapitel „QoS Catalog“ dieser Spezifikation definiert einige anwendungsunabhängige QoS Charakteristika und enthält eine Einteilung von QoS. Abbildung 2.17 gibt einen Überblick über die QoS Kategorien.

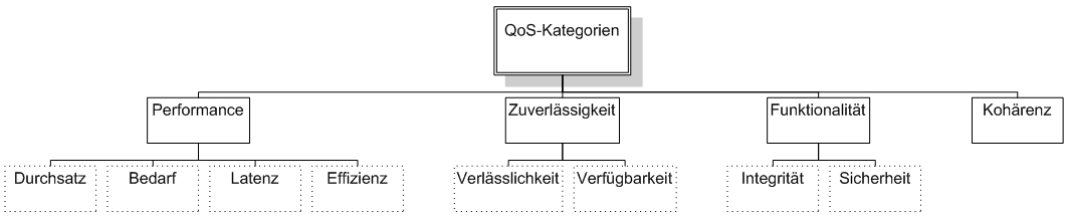

Abbildung 2.17: QoS-Kategorien nach [OMGh]

Zur Illustration werden im Folgenden einige QoS-Parameter aufgelistet und kurz beschrieben (für eine genauere Darstellung siehe auch [OMGh]):

- Unter Latenz versteht man allgemein eine Verzögerungszeit bzw. ein Zeitintervall, in dem eine Antwort auf ein Ereignis gefordert wird. Im Zusammenhang mit Kommunikation ist damit das Zeitintervall zwischen dem Senden und Empfangen von Daten gemeint. Die Dauer der Übertragung von Daten wird dabei durch die geographische Distanz sowie die Anzahl an Netzwerkteilnehmern und Infrastrukturkomponenten (Router) beeinflusst. Die Art der Vernetzung der Teilnehmer kann ebenso maßgebend sein.

- Mit Jitter wird die Abweichung oder Schwankung (Varianz) der Latenz bezeichnet. Bei der Ermittlung des Jitters bedient man sich oft statistischer Methoden.

- Die Verlustrate gibt die Wahrscheinlichkeit für den Verlust von Daten an. Daten können an den unterschiedlichsten Stellen der Übertragung verloren gehen. So ist es z.B. denkbar, dass ein Router mit vollem Buffer Daten verwirft. Es können aber auch bei der Datenverarbeitung oder aufgrund des Übertragungsmediums Fehler auftreten.

- Die Zuverlässigkeit (engl. dependability) bezieht sich auf den Grad der Erfüllung der geforderten Aufgabe. Auch hier werden statistische Aussagen herangezogen. Wichtige in diesem Begriff beinhaltete Kenngrößen sind die Verlässlichkeit (engl. reliability) und die Verfügbarkeit (engl. availability).

- Bei der Sicherheit spielen Maßnahmen zur Vermeidung von illegalem Zugriff oder nicht erlaubter Verwendung von Daten eine entscheidende Rolle.

- Der Netzwerkdurchsatz ist ein (gemitteltes) Maß für die pro Zeiteinheit übertragene Datenmenge.

- Um von Kohärenz[6] sprechen zu können, muss Konsistenz[7] gewährleistet werden.

- Effizienz bezieht sich auf die Erfüllung von Aufgaben mit minimalen Mitteln (Ressourcen).

- Fehlertoleranz kann mit Redundanz, Fehlererkennung und Wiederherstellungsmechanismen erzielt werden. In diesem Zusammenhang ist auch oft von Robustheit die Rede, die einen zuverlässigen Betrieb selbst unter ungünstig(st)en Bedingungen meint.

- Der Begriff Skalierbarkeit bezieht sich auf die Performance bei steigender Komplexität.

Bezüglich der auf hoher Ebene angesiedelten Darstellung nicht-funktionaler Anforderungen, nämlich der Beschreibung von QoS mittels XML sei noch auf [PDC04] verwiesen. Die dort beschriebene Real Time Markup Language (RTML) ermöglicht die Transformation von QoS-Daten in ein

[6] In [BU] wird Kohärenz als „*das korrekte Voranschreiten des Systemzustands durch ein abgestimmtes Zusammenwirken der Einzelzustände*“ beschrieben. Bei der Cache-Kohärenz muss dafür Sorge getragen werden, dass der Cache immer die aktuellen Werte zurück liefert.

[7] Konsistenz meint die Widerspruchsfreiheit gemeinsam genutzter Daten und spielt im Zusammenhang mit Lese- und Schreiboperationen eine wesentliche Rolle [Wikc].

XML-Format für Echtzeit-Systeme, wodurch der Datenaustausch und die Interoperabilität zwischen Internet- und Business-to-Business (B2B) Anwendungen ermöglicht werden soll. Die RTML baut unter anderem auf oben erwähntem UML-Profil auf und kann als Alternative zum CORBA IIOP (siehe Kapitel 2.2.1) oder Java Messaging Service (Kapitel 3.3) gesehen werden.

2.4 IEC 61499

Wie eingangs erwähnt kann der steigenden Komplexität in der Automatisierungstechnik, die u.a. aus einer verstärkten Verteilung der Intelligenz innerhalb des Systems resultiert, mit Hilfe von (wieder verwendbaren) Funktionsblöcken begegnet werden [Lew01]. Der Standard IEC 61499 - oft auch als „Nachfolge-Norm“ von IEC 61131 bezeichnet - baut auf dem FB-Konzept auf, indem er die Modellierung (jedoch nicht die Programmierung) von verteilten Systemen mittels Funktionsblöcken beschreibt. Da es sich hierbei um einen offenen Standard handelt, wird auch eine Möglichkeit geschaffen, Systeme unterschiedlicher Hersteller kompatibel miteinander zu vernetzen.
Als wichtigste Ziele der Norm IEC 61499 sind die Interoperabilität, Portabilität, Konfigurierbarkeit bzw. Rekonfiguration und ein verbessertes Engineering zu nennen.

Das zentrale Element der IEC 61499, nämlich der elementare FB (engl. basic function block), ist in Abbildung 2.18 zu sehen.

Durch die Ausführungssteuerung wird festgelegt, welcher Algorithmus durch welches Ereignis ausgelöst wird, und welches Ereignis letztendlich am Ausgang generiert wird. Die Regeln dieser ereignisgesteuerten Zustandsmaschine werden für den Basic FB normativ in einem Zustandsgraphen (ECC, Execution Control Chart) veranschaulicht. Über das WITH-Konstrukt (eine vertikale Verbindung zwischen einem Event und einem oder mehreren Dateneingängen) wird angezeigt, welche Daten mit welchem Event anliegen müssen.
Die internen Daten sowie Algorithmen bleiben durch den FB verborgen.

Neben dem elementaren FB, der durch das Zusammenspiel von Eingangsevent, Ausführungssteuerung und Ausgangsevent gekennzeichnet ist, definiert der Standard außerdem einen zusammengesetzten FB (Composite FB) und einen SIFB (Service Interface Function Block). Der Composite FB

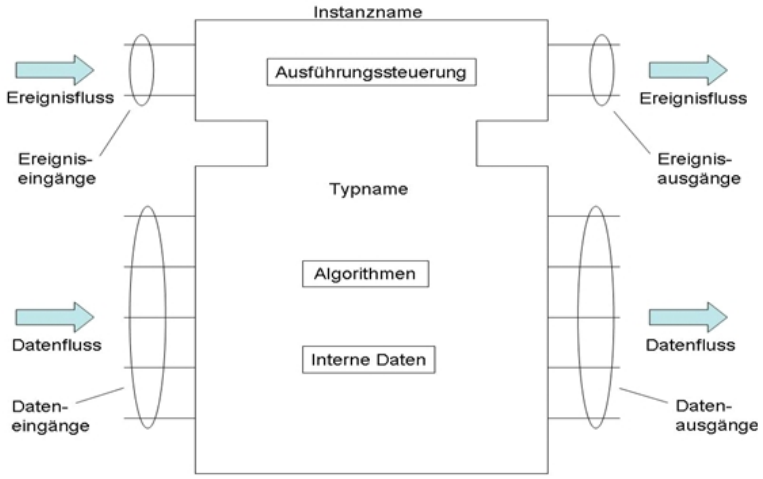

Abbildung 2.18: Basic FB nach IEC 61499, in Anlehnung an [Lew01]

beinhaltet in seinem Inneren ein Netzwerk an FB-Instanzen, die entweder untereinander oder mit den Ein- und Ausgängen des Composite FB verbunden sind. Durch diese Kapselung erscheint der Composite FB nach außen hin von seiner Funktionsweise her wie ein Basic FB.
Dem SIFB kommt bei der Verteilung (Mapping) eine zentrale Rolle zu, denn er übernimmt die wichtige Aufgabe der Kommunikation. Sobald eine Interaktion zwischen FBs innerhalb einer Ressource und der Außenwelt (externe Ressource oder Prozess) stattfinden soll, kommt ein SIFB zum Einsatz.
Obwohl die Norm IEC 61499 keine Vorgaben für spezielle SIFBs macht, so definiert sie doch ein generisches Interface. Dieses schließt sowohl Eingangs-Events und Ausgangs-Events als auch Eingangs- und Ausgangs-Variablen ein. Diese Inputs und Outputs müssen zwar nicht verwendet werden, falls sie allerdings eingesetzt werden, sollten sie der IEC-Semantik gehorchen. Zu den Event Inputs zählen:

- INIT: dient der Initialisierung eines FB-Dienstes
- REQ: wird zur Anforderung von Daten von einer externen Quelle verwendet
- RSP: bewirkt die Beantwortung einer externen Anfrage

Die Event Outputs sind

- INITO: zeigt eine abgeschlossene Initialisierung an (als Folge eines INIT-Events), sagt aber nichts über deren Erfolg aus

- CNF: dient der Signalisierung des abgeschlossenen Anfrageversendens (vgl. REQ)
- IND: zeigt den Erhalt einer Antwort an (korrespondiert mit RSP)

Bei den Dateninputs werden folgende Variablen vorgeschrieben:

- QI: BOOL (wird mit dem INIT-Ereignis verwendet und sagt aus, ob das FB-Service initialisiert werden soll)
- PARAMS: ANY (Datenstruktur, die beim INIT-Event benötigt wird, z.B. ID zur Kommunikation)
- SD_1, ..., SD_N: ANY (diese Daten werden bei Anfragen bzw. Antworten verschickt)

Bei den Datenoutputs sollten nachstehende Variablen nicht fehlen:

- QO: BOOL (zeigt an, ob die Initialisierung erfolgreich war)
- STATUS: ANY (kann von jedem Eingangsereignis gesetzt werden und enthüllt etwas über deren Abarbeitung)
- RD_1, ..., RD_N: ANY (Daten, die mit CNF- und IND-Events anliegen)

Bezüglich des SIFB-Typnamen ist zu sagen, dass er die Art des Services beschreiben soll (z.B. HMIWriter).
Für eine anschauliche Darstellung der Interaktion hat sich innerhalb der Norm das Zeit-Sequenz-Diagramm (ZSD) durchgesetzt. Darin kann die Reihenfolge bei der Interaktion, d.h. wann welcher Akteur aktiv ist, erkannt werden. Der bidirektionale Datentransfer eines Client/Server-Paars als ZSD ist in Abbildung 2.19 zu sehen.

Schließlich sollen einige Modellarten von IEC 61499 kurz vorgestellt werden:

Systemmodell
Das Systemmodell kann als höchste Abstraktionsebene gesehen werden. Hier wird ein verteiltes System durch seine über mehrere Kommunkationsnetzwerke verbundenen Geräte dargestellt, wobei diesen Geräten unterschiedliche Anwendungen (bzw. Teile von verteilten Anwendungen) zugeordnet sind (siehe Abbildung 2.20).

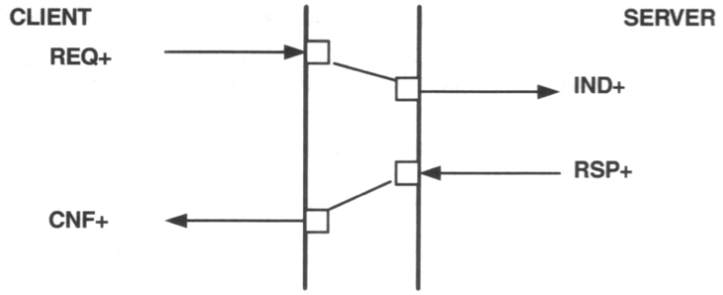

Abbildung 2.19: Mögliches Szenario bei der Kommunikation zwischen Client und Server [Lew01]

Gerätemodell
Ein Gerät besteht immer aus einer oder mehreren Ressourcen, die ganze Anwendungen oder Teile davon unterhalten (siehe Abbildung 2.20). Konkret werden in den Ressourcen FBs ausgeführt. Um mit externen Ressourcen (aus anderen Geräten) kommunizieren zu können, gibt es ein Kommunikationsinterface. Das Prozessinterface stellt eine Verbindung zwischen den Ressourcen und den Ein- und Ausgängen des Geräts dar.

Ressourcenmodell
Eine Ressource ist eine unabhängige Einheit, d.h. sie stellt eine entsprechende Infrastruktur zur Verfügung, damit die in ihr enthaltenen FB-Netzwerke (bzw. Fragmente davon) für sich alleine ausführbar sind (siehe Abbildung 2.20). Eine Scheduling-Funktion sorgt dafür, dass die FB-Algorithmen in der richtigen Reihenfolge ausgeführt werden. Außerdem werden Schnittstellen zu den Kommunikationssystemen und zum gerätespezifischen Prozessinterface angeboten (vgl. Gerätemodell).

Bezüglich anderer Modelle innerhalb des Standards IEC 61499 (z.B. Applikationsmodell, Managementmodell) muss auf die Literatur (z.B. [Lew01]) verwiesen werden.

Die Applikationserstellung mittels FBs ist mittlerweile Stand der Technik, und es existieren auch schon einige Referenzimplementierungen bezüglich IEC 61499-basierter Verteilung.
Besonders erwähnenswert ist beispielsweise der TORERO-Ansatz - ein EU-Projekt, das die Entwicklung eines selbst-konfigurierenden, selbst-wartenden und automatisch verteilten Steuersystems zum Ziel hatte [TOR].

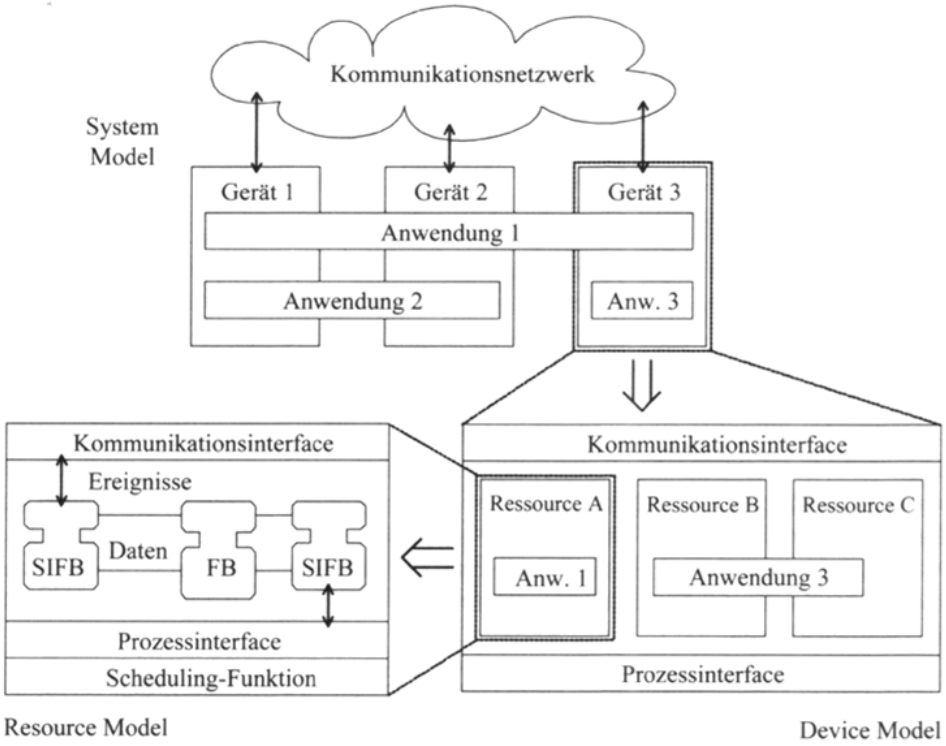

Abbildung 2.20: Zusammenhang zwischen System-, Geräte- und Ressourcenmodell in IEC 61499 [FB04]

In [STL+04] wird ein Ansatz innerhalb des TORERO-Projekts präsentiert, der das Mapping von IEC 61499 FBs sowohl auf zeitgesteuerte als auch ereignisgesteuerte Protokolle behandelt.
Thramboulidis et al. widmen sich in ihrer Arbeit u.a. der Entwicklung eines Engineering-Tools für verteilte Steuerungsanwendungen [TT04], [TT03]. Durch eine Kombination des FB-Konzepts mit der UML (Unified Modelling Language) soll die Applikationsentwicklung in allen Phasen unterstützt werden. In [Thr05] wird im Besonderen auf die Diskrepanz zwischen IEC 61499-Systemen und der gesamten Life Cycle-Unterstützung beim Software Engineering innerhalb der IAT eingegangen.

3 Konzept

Aus dem IT-Bereich sind uns verteilte Anwendungen schon lange bekannt, und sie betreffen mittlerweile unser tägliches Leben. Bei der Geldbehebung am Bankautomaten oder bei einer Buchbestellung im Online-Store nutzen wir die Dienste einer verteilten Anwendung bzw. erachten diese als selbstverständlich. Der Verteilung wird dabei meist mit einer Middleware begegnet, die neben ihrer Grundfunktionalität zusätzliche Services (z.B. QoS-Überwachung) zur Verfügung stellen muss, um den Anforderungen des Benutzers bzw. der Applikation gerecht zu werden.
Die starke Durchtränkung des Marktes mit verteilten Systemen bzw. deren große Verbreitung spiegelt sich in einer Fülle an (frei) verfügbaren Middleware-Lösungen wider, die für die unterschiedlichsten Einsatzgebiete ausgelegt und demzufolge durch jeweils andere Features gekennzeichnet sind.

Die Entwicklung in der Automation und ihren vernetzten Gerätearchitekturen geht einen ähnlichen Weg [JSAD04]. Während heutzutage noch häufig zentralisierte Systeme mit großen „intelligenten“ Servern und „dummen“ Geräten anzutreffen sind, in denen Client-Server- und Punkt-zu-Punkt-Strukturen gängig sind, so geht der Trend in Richtung dezentralisierte bzw. verteilte Systeme mit ereignisgesteuerter und serviceorientierter Kommunikation. Publish-Subscribe-, Multicast- und P2P-Kommunikation sollen in Zukunft in ihrer aktiven Ausprägung Effizienz, Skalierbarkeit, Konfigurierbarkeit und Ausfallsicherheit ermöglichen bzw. diese erhöhen und in Kombination mit entsprechenden Protokollen und Services Interoperabilität und Plattformunabhängigkeit gewährleisten. Diese Tendenzen sind in Tabelle 3.1[1] nochmals zusammengefasst.

Es ist daher naheliegend, sich des einen oder anderen Konzepts aus der Informatik zu bedienen und es für die Automatisierungstechnik zu übernehmen bzw. zu adaptieren, wobei hier die Norm IEC 61499 im Zusammenhang mit verteilten Systemen einen besonderen Stellenwert einnimmt. Beispielsweise wäre die Umsetzung von ausgereiften QoS-Mechanismen für verteilte, echtzeitfähige und eingebettete Systeme äußerst erstrebenswert, um auch wirklich

[1] Dezentrale Systeme (engl. decentralized systems) sind durch das Zusammenspiel von SPS und Feldbus gekennzeichnet und gehören zum Stand der Technik. In Zukunft werden hingegen vor allem verteilte Architekturen eine besondere Rolle spielen, weshalb der Begriff „decentralized“ in Tabelle 3.1 durch „distributed“ ersetzt werden sollte.

	Today	Tomorrow
System	Centralized: large intelligent "servers", dumb devices	Decentralized: intelligent, autonomous devices
Comms	Polling, client-server, point-to-point	Event-driven, publish-subscribe, multicast, peer-to-peer
Set-up	Long and difficult, manual programming, tedious debugging	No programming, plug-and-play, context-aware configuration

Tabelle 3.1: Tendenzen in der Automation [JSAD04]

die Zielsetzungen für die Zukunft (siehe oben) erreichen zu können. Wichtig bei der Bereitstellung diverser Services für IEC 61499-Systeme ist es, die begrenzten Ressourcen der Zielsysteme entsprechend zu berücksichtigen; die Konzepte aus dem IT-Bereich werden mitunter für die verteilte Automation mit Embedded Systemen zu umfangreich und groß sein.

3.1 Umsetzung der Konzepte aus dem IT-Bereich

Um die besagte Funktionalität der IT-Konzepte für die Automatisierungstechnik nutzbar zu machen, bieten sich im Wesentlichen drei Möglichkeiten an:

- Direkter Einsatz einer vorhandenen Middleware
  Falls eine entsprechende Middleware sämtliche Anforderungen bezüglich QoS-Unterstützung, Kommunikationsmechanismen und Ressourcenbeschränkung erfüllt, kann die Middleware auf jeden Knoten im verteilten System gespielt werden, um sodann die Kommunikation innerhalb der verteilten Anwendung zu koordinieren.

- Verwendung von Protokollen mit QoS-Unterstützung
  Durch die Verwendung bzw. Entwicklung eines Protokolls, das die höheren Schichten des OSI-Referenzmodells funktional unterstützt, könnten der verteilten Anwendung die benötigten Services zur Verfügung gestellt werden.

- Nachbilden der Middleware-Funktionalität mit Mitteln der IAT
  Als letzte Möglichkeit bietet sich eine Adaption der Middleware-Funktionalität mit Mitteln der IEC 61499 an. Diverse (QoS-) Mechanismen könnten beispielsweise mittels SIFBs abgesetzt werden.

Um nun eine dieser Möglichkeiten favorisieren zu können, müssen zuerst die Anforderungen von Automatisierungssystemen analysiert werden. Erst dann kann geprüft werden, ob bzw. in welchem Ausmaß die Forderungen mit den erwähnten Mitteln und Möglichkeiten erfüllt werden können.

3.2 Anforderungen der IAT

Die Trends in der IAT wurden bereits oben genannt (siehe Tabelle 3.1) und sind eng mit den Anforderungen verbunden. Natürlich sind diese Anforderungen unzählig und vom jeweiligen Anwendungsfall abhängig. Doch eine vollständige Abhandlung ist nicht notwendig, da eine Beschränkung auf einige wenige Anforderungen die Möglichkeiten bereits deutlich verringert und somit den Lösungsweg vorgibt.

3.2.1 Kommunikation

Um ein passendes Kommunikationsmodell aussuchen zu können, ist es wichtig, die Voraussetzungen bzw. Anforderungen für eine Kommunikation in verteilten, echtzeitfähigen, eingebetteten Automatisierungssystemen zu kennen.
Die Herleitung der Kommunikationsanforderungen kann über eine Betrachtung der möglichen „Kommunikationsfälle“ geschehen, wie dies in [SPK05] gezeigt wird. Die Ergebnisse werden im Folgenden kurz zusammengefasst und beziehen sich in erster Linie auf die Prozessautomation, wobei eine Ausweitung auf andere Bereiche der industriellen Automation durchaus legitim ist:

- Prozesskontrolle: Die Prozesskontrolle erfordert das Zusammenspiel von Controllern, Sensoren und Aktoren, zwischen denen entweder zyklische Informationen ausgetauscht werden oder Ereignisse. Bei periodischen Daten ist es meist nicht unbedingt erforderlich, dass alle Werte an ihrem Ziel ankommen. Ereignisse (z.B. Parameter-Änderungen) hingegen sind unregelmäßiger Natur; bei ihnen ist eine sichere Übermittlung wichtig.

- Produktionskontrolle: Hier müssen (große Mengen an) Produktionsdaten sowie Parameter verlässlich übertragen bzw. geändert werden können.

Eine etwaige Latenz kann meist vernachlässigt werden - ein spätes Empfangen von Produktionsdaten strapaziert letztendlich „nur“ die Geduld des Benutzers.

- Benutzerschnittstellen: An Benutzerschnittstellen müssen sowohl zyklische Informationen (Trends) als auch Events und Alarme angezeigt werden. Je nach Art des Datenstroms resultieren unterschiedliche Anforderungen an die Sicherheit und Latenz. Als wesentliche Akteure sind Kontrollräume, Terminals und menschliche Benutzer zu nennen.

- Management der Produktionsdaten: Es müssen große Datenmengen (sowohl zyklische Daten als auch Events) gespeichert und daraus Berichte generiert werden. Auch wenn keine hohen Anforderungen an die Latenz gestellt werden, kann eine Zeitauflösung bis in den Millisekundenbereich durchaus nötig sein; die Generierung von Berichten darf mitunter wenige Sekunden dauern. Die Akteure sind Geräte auf Feldebene, Controller und Datenbanken.

- Systempflege: Diese beinhaltet das Laden, Starten und Updaten von Komponenten.

Bei den Verbindungstypen kann - je nach Beziehung zwischen Quelle und Ziel - zwischen point-to-point (PTP), point-to-multipoint, multipoint-to-point und multipoint-to-multipoint unterschieden werden [KDL00]. Die letzten beiden Typen sind für fehlertolerante Systeme wichtig, da sie eine gewisse Redundanz ermöglichen (Duplikate in unterschiedlichen Prozessoren).

Der Verbindungsmodus kann entweder synchron oder asynchron sein. Bei einer asynchronen Übertragung blockiert der Sender im Gegensatz zur synchronen Übertragung nicht bis zum Erhalt einer Bestätigung.

Bezüglich der Interaktion zwischen den einzelnen Komponenten existieren mehrere Mechanismen. Die wichtigsten Kommunikationsmechanismen (bzw. Arten von Datenströmen) sind (siehe auch [PCSH99], [THJ+05]):

- Kontinuierliche Datenverteilung: Dieser Mechanismus entspricht der „Verdrahtung“ von Funktionsblöcken zur Datenweitergabe in zyklischen Systemen (vgl. IEC 61131-3). Die Ausgänge eines FBs werden mit den Eingängen eines anderen FBs verbunden, wodurch aus Sicht der Anwendung der Eindruck eines gleichen Wertes an beiden Enden der Verbindung entsteht (siehe Abbildung 3.1). Dieser Eindruck wird durch

eine kurze Zykluszeit bei der Programmabarbeitung und Datenübertragung erzielt. Neben einer periodischen (zyklischen) Aktualisierung der Daten bietet sich auch eine Übertragung in Abhängigkeit von der Änderung der Daten an (Event- bzw. Interrupt-getrieben).
Prinzipiell gelten strenge Anforderungen sowohl hinsichtlich Latenz und Jitter als auch die Verlässlichkeit bzw. Sicherheit der Übertragung betreffend. Bei der Übermittlung von Sensorwerten ist es allerdings meist besser, einen Wert zu versäumen, als eine Verzögerung durch ein erneutes Versenden zu riskieren.

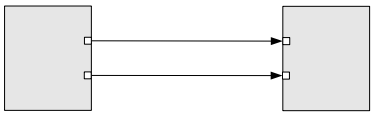

Abbildung 3.1: Kontinuierliche Datenverteilung [THJ+05]

Der Datenstrom besteht aus Signalen. Signale stellen einen unidirektionalen Datenfluss dar und beinhalten kontinuierliche Information (z.B. Sensordaten). Ihre Übertragung kann als zeitkritisch bezeichnet werden, woraus eine last-is-best Strategie resultiert, d.h. die aktuellsten Daten werden bevorzugt, alte Daten oder wiederholtes Senden von verloren gegangenen Daten sind meist sinnlos.

- Ereignisbasierte Verteilung: Dieses Konzept findet seine Umsetzung in der Norm IEC 61499 (siehe Abbildung 3.2). Eine Datenübertragung bzw. eine Algorithmus-Auslösung im empfangenden FB (und die damit eventuell verbundenen weiteren Zustandsänderungen) sind immer an ein Ereignis (Zustandsänderung des erzeugenden Blocks) gebunden. Es darf kein Event verloren gehen, d.h. eine sichere Verteilung muss garantiert sein.

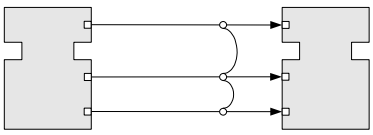

Abbildung 3.2: Ereignisbasierte Verteilung [THJ+05]

Die Übertragung von Nachrichten erfolgt über Ereignisse, d.h. die Daten

sind an Ereignisse geknüpft. Durch diese Zuordnung eines Sets an Daten zu einem bestimmten Ereignis soll Konsistenz gewährleistet werden.

- Ereignisbenachrichtigung und -bestätigung: Während die zuvor erwähnte ereignisbasierte Verteilung die normale Programmabarbeitung betrifft, werden hier zusätzlich Alarme und Benachrichtigungen an interessierte Empfänger geschickt (siehe Abbildung 3.3). Der Empfang dieser Daten erfordert ein entsprechendes Reagieren bzw. Eingreifen in die Applikation durch den Benutzer; beispielsweise müssen Benachrichtigungen bestätigt werden, bevor sie in einer Datenbank abgelegt werden. Es existieren meist keine Einzelverbindungen, sondern es wird eine große Menge an Events von unterschiedlichen Ressourcen angefordert.
  Die verlässliche Übertragung der Ereignisse ist besonders wichtig.

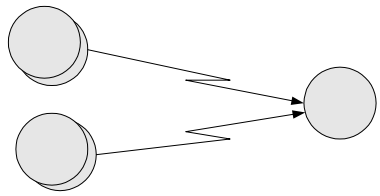

Abbildung 3.3: Ereignisbenachrichtigung und -bestätigung [THJ+05]

- Request/reply (vgl. requester/responder SIFBs in IEC 61499): Damit eine Applikation an gewünschte Daten kommt, kann sie dies über eine Anfrage (request) machen. Bei dieser Form der Informationsbeschaffung kommt das Client/Server-Prinzip (siehe Kapitel 2.1) zum Einsatz. Die Dringlichkeit ergibt sich aus der Situation heraus, wobei eine verlässliche Kommunikation wünschenswert ist.

- Andere: Remote Read/Write (vereinfachte Form von Request/Reply, da ausschließlich Lese- und Schreiboperationen durchgeführt werden, siehe Abbildung 3.4), nachrichtenorientierte Kommunikation (Entkopplung von Sender und Empfänger durch Verwendung von Mailboxen bzw. Kanälen, siehe Abbildung 3.5) u.a. [Sie03]

Neben den erwähnten Datenstromarten (Signale, Ereignisse, Anfragen) kommen bei der Kommunikation in der IAT auch Kommandos und Statusabfragen zum Einsatz. Kommandos dürfen nicht verpasst oder unabsichtlich zweimal ausgeführt werden. Durch Statusabfragen können aktuelle Zustände oder gewisse Ziele in Erfahrung gebracht werden.

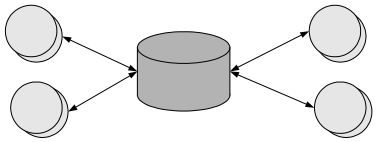

Abbildung 3.4: Remote Read/Write [THJ+05]

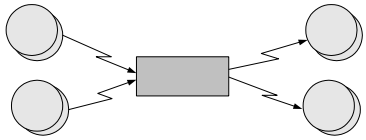

Abbildung 3.5: Nachrichtenorientierte Kommunikation [THJ+05]

Sämtliche der erwähnten Datenstromarten stellen unterschiedliche Anforderungen an die Geschwindigkeit, Verlässlichkeit etc., die entsprechend berücksichtigt und bedient werden müssen.

3.2.2 QoS

Um eine Konzentration auf die aus industrieller Sicht relevanten QoS-Parameter zu ermöglichen, wurde unter den εCEDAC-Projekt-Partnern eine Umfrage durchgeführt. Die Umfrage-Teilnehmer (insgesamt zehn Personen) wurden nach ihrem Verständnis für QoS-Parameter in der industriellen Kommunikation auf Ebene der Steuerungsapplikation befragt, sowie zur Nennung einiger Beispiele aufgefordert.
Die Befragten stammten aus den unterschiedlichsten Bereichen: neben Steuerungsherstellern und Herstellern von Software-Tools waren auch Endanwender aus der Prozesstechnik sowie der Gebäudeautomation vertreten. Insofern war ein breit gefächertes Spektrum an Antworten zu erwarten, aus dem letztendlich eine Auswahl an Parametern getroffen werden konnte.
Die meisten der angegebenen Parameter sind zeitbezogen. Zu ihnen zählen:

- Antwortzeit auf Anforderung (response time)
- Deadlines: absolute und relative Zeitbedingungen (in Bezug auf andere Nachrichten), die Synchronisation notwendig machen
- Latenzzeit (Determinismus)

- Jitter
- Gültigkeitsdauer (Erkennen von veralteten Daten)
- Zykluszeiten (cycle time)
- Real Time (hard/soft, absolut/relativ)
- MTBF (Mean Time Between Failures)
- MTTF (Mean Time To Failure)
- Rekonfigurationsdauer (Zeiten für Verbindungsaufbau)

Die genannten kapazitätsbezogenen Parameter sind:

- Datendurchsatz (minimale Sendefrequenz), Systembelastung durch Kommunikation (Benchmarks), Reservierung von Bandbreite
- Fehlerrate (absolut/relativ)
- Prioritäten (können auf Buszeiten abgebildet werden)
- Verfügbarkeit (Laut [Wiki] ist sie durch $\frac{Gesamtzeit - Gesamtausfallzeit}{Gesamtzeit}$ definiert und ein Maß für die Funktionsfähigkeit eines Systems.)

Bei den sicherheitsbezogenen Einflussgrößen wurden folgende Parameter angeführt:

- Datensicherheit (Manipulierbarkeit, Abhörsicherheit)
- Übertragungssicherheit (Bestätigung des Erhalts durch Acknowledgements, Redundanz), Reliability
- Fehlererkennung (Paketverlust, Erkennen von out-of-sequence Paketen, sicheres Erkennen von Verbindungsabbrüchen, Datenfehler: Wahrscheinlichkeit, Erkennbarkeit, Korrigierbarkeit)
- Reihenfolge bei der Initialisierung (Server vor Clients) → Rekonfigurierbarkeit (Hinzukommen bzw. Verschwinden von Teilnehmern)

Die oben getroffene Einteilung in zeitbezogene, kapazitätsbezogene und sicherheitsbezogene Parameter ist nur eine Möglichkeit der Kategorisierung (vgl. [Bla00]). Eine andere Art der Einteilung wurde bereits in Kapitel 2.3 vorgestellt (UML-Profil).

Bei der Bereitstellung von QoS muss zwischen einer statischen und einer dynamischen Bereitstellung unterschieden werden [WSG+03]: Während die statische QoS-Bereitstellung eine Vorkonfiguration der QoS-Ressourcen (zur Design-Zeit) erfordert, werden bei der dynamischen Bereitstellung die benötigten Ressourcen in Abhängigkeit des Runtime-Status bereitgestellt (Einflussnahme zur Laufzeit).
Des Weiteren muss bei der QoS-Implementierung auf den alle Ebenen übergreifenden Charakter der QoS geachtet werden, d.h. die QoS-Implementierung innerhalb einer alleine stehenden Komponente ist meist nicht möglich, vielmehr ist eine Kooperation zwischen allen beteiligten Komponenten wichtig. Manche QoS-Parameter betreffen nur den Sender oder nur den Empfänger, es gibt aber auch Größen, die beide Seiten betreffen.
Die Definition der einzelnen Parameter auf Anwendungsschicht birgt zwar den Vorteil einer größeren Aussagekraft für den Benutzer. Viele Parameter müssen aber letztendlich zwecks Umsetzung auf ein Bussystem abgebildet werden.
Bei der Implementierung von QoS kann zwischen Parametern, die nur auf Applikationsebene und jenen, die in den unterliegenden Schichten umgesetzt werden, unterschieden werden. Meist verhält es sich so, dass für die Festlegung und Kontrolle der QoS Schicht 7 verantwortlich ist, während eine konkrete Realisierung Aufgabe anderer Schichten ist. Beispielsweise können Gültigkeitsdauern von Nachrichten oder Bestätigungen über den Erhalt oder die Verarbeitung von Nachrichten auf Applikationsebene definiert und überprüft werden, ohne in andere Schichten eingreifen zu müssen.
Bezüglich der konkreten Umsetzung einer vorgegebenen Latenzzeit oder eines Jitters, oder aber die Einhaltung von Echtzeit- und Bandbreitevorgaben betreffend, muss die Netzwerkebene unausweichlich einbezogen werden. Ebenso verhält es sich mit den in der Umfrage genannten Prioritäten: Diese werden vom Bussystem entweder unterstützt oder auch nicht und folglich auch in der Netzwerkebene umgesetzt.

Im Zusammenhang mit der QoS-Beeinflussung auf Netzwerkebene existieren einige Techniken, die Ende-zu-Ende-QoS in IP-Netzen garantieren sollen [Rod02]: Dazu zählen spezielle auf Ressourcenreservierung basierende Protokolle wie RSVP[2] (Resource reSerVation Protocol), Systeme zur Klassifizierung von Paketen (Diff-Serv[3]) und hybride Routing/Switching

[2] Das RSVP ist ein Protokoll im IP-Stack, das sowohl Hosts als auch Routern zur Umsetzung ihrer QoS-Anforderungen dient [Wikh].
[3] DiffServ steht für Differentiated Services und erlaubt dem Sender, eine Priorität für seine IP-Datenpakete anzugeben [Wikb].

Systeme wie z.B. MPLS[4] (Multiprotocol Label Switching).

3.2.3 DRE-Systeme

In der Literatur wird oft der Begriff „DRE-System“ erwähnt. DRE steht hier für distributed (verteilt), real-time (Echtzeit) und embedded (eingebettet). Der Begriff des verteilten Systems bzw. der verteilten Anwendung wurde bereits in Kapitel 2.2 erklärt. Ganz allgemein ist damit das Zusammenspiel von unterschiedlichen Maschinen gemeint, wobei durch den Fortschritt in den Kommunikationstechnologien immer öfter auch drahtlose Verbindungen zum Einsatz kommen.
Echtzeit bzw. Echtzeitfähigkeit wurde bereits im Zusammenhang mit QoS erwähnt und stellt eine wichtige Anforderung im Bereich der Automatisierungstechnik dar [FB04]. Während bei herkömmlichen Aufgaben innerhalb der Technik alleine die Richtigkeit des Ergebnisses ausreichend sein kann, spielt bei Echtzeitsystemen auch der Zeitpunkt der Ergebnisbereitstellung eine entscheidende Rolle. Zu beachten ist allerdings, dass Echtzeit nicht unmittelbar mit Schnelligkeit zu tun hat, sondern eher mit Rechtzeitigkeit (bzw. Gleichzeitigkeit oder Pünktlichkeit) und in weiterer Folge mit Determiniertheit und Verfügbarkeit. D.h. die voraussehbare Steuerung der Ende-zu-Ende-Eigenschaften eines Systems stellt eine wichtige Voraussetzung für Echtzeitsysteme dar. Eine mögliche Einteilung in harte und weiche Echtzeitanforderung gibt Aufschluss darüber, wie kritisch die Einhaltung von Zeitvorgaben ist[5]. Für die Modellierung von Echtzeitsystemen sei auf [OMGi] verwiesen.
Als letzter Begriff soll der des eingebetteten Systems (engl. Embedded System) erklärt werden [McK03]: Ein eingebettetes System dient meist einem einzigen spezifischen Zweck, wobei das eingebettete System als Bestandteil eines größeren Systems oft nicht direkt erkennbar ist (sowohl in physischer als auch in funktioneller Hinsicht) aber das Verhalten des übergeordneten Systems oft maßgeblich beeinflusst. Zu beachten ist, dass ein eingebettetes System nicht unbedingt klein oder leistungsarm sein muss. Z.B. kann die Autopilot-Funktion bei einem Flugzeug mit all den beteiligten (und untereinander kommunizierenden) Komponenten und den dafür nötigen

[4] Die MPLS-Schicht ist zwischen den OSI-Schichten zwei und drei angesiedelt und erlaubt eine verbindungsorientierte Übertragung in verbindungslosen Netzen [Wikd].

[5] Während bei weichen Echtzeitanforderungen Zeitlimits nur im Mittel erreicht werden müssen, so erfordern harte Echtzeitanforderungen das Einhalten von Zeitlimits unter allen Umständen.

komplexen Regelungen als eingebettetes System innerhalb des Systems Flugzeug betrachtet werden.
Bei den in der Automatisierungstechnik eingesetzten Systemen (Aktoren, Sensoren etc.) handelt es sich jedoch oft um Microcontroller, die sowohl Einschränkungen bezüglich Größe als auch Ressourcen unterworfen sind [Pic04]. Neben Kosten und Gewichtsüberlegungen spielt die eingeschränkte Leistung eine entscheidende Rolle. Prozessoren ab 8 Bit, mit 20 MHz Taktfrequenz und einigen 100 KB Speicher sind durchaus üblich. Die verwendeten Betriebssysteme (z.B. eCos (embedded Configurable operating system), eRTOS (embedded Real-Time Operating System), ThreadX etc.) sind nicht mit herkömmlichen Desktop- oder PC-Betriebssystemen zu vergleichen. Es handelt sich bei ihnen oftmals um echtzeitfähige Systeme (RTOS, Real-Time Operating System), deren Funktionalität auf ein Minimum beschränkt ist. Beispielsweise können oft keine herkömmlichen Speichertechniken verwendet werden, und die Software muss auf kleinen Speichermedien wie EEPROMs oder NVRAMs Platz finden.
Auf einen Umstand sei noch hingewiesen: Eine notwendige (nicht hinreichende) Bedingung, um mit Middleware überhaupt Echtzeitverhalten erzielen zu können, ist die Verwendung von RTOSs.
Eine zusätzliche und wichtige Einschränkung in der IAT ist das Vorhandensein heterogener Umgebungen.

3.3 Middleware-Analyse

In diesem Kapitel sollen die in Kapitel 2.2 kurz vorgestellten Middleware-Produkte im Hinblick auf einen direkten Einsatz in der IAT untersucht werden. Es werden besondere Features der einzelnen Produkte hervorgehoben. Außerdem erfolgt eine Beurteilung der jeweiligen Middleware bezüglich Kommunikation, QoS und Ressourcenbeschränkung.
Prinzipiell existiert eine Vielzahl an Kriterien, die beim Einsatz von Middleware bedacht werden muss. Tabelle 3.2 gibt dazu einen Überblick. Es handelt sich hierbei um eine mögliche Taxonomie von Middleware, die u.U. sehr hilfreich bei der Auswahl von Middleware sein kann. Eine genaue Erklärung dieser Tabelle findet sich in [McK03].

(RT-) CORBA
CORBA verwendet für die Kommunikation RMIs, die in ihrer Grundform jedoch sehr zeitaufwendig und unflexibel sind [Sie03]. Vor allem bei großen

Embedded Hardware	Roles (Client/Server/Peer)			Software Input/Output	IDL Subsets
	Control Flow	Data Flow	Interaction Style		
System Composition • Homogenous • Asymmetric ***Hardware I/O Support*** • Serial, Parallel, 1-wire, Ethernet, IrDA, Bluetooth, GSM, GPRS ***Resources*** • Memory • Power ***Processing Capabilities*** • 8-bit, 16-bit, 32-bit, ...	***Connection Setup*** • Initiate setup • Receive setup requests ***Service Location*** • Hardwired-logic • Config. file • Name service • Other	***Data Direction*** • Bits in • Bits out • Bits in/out ***Parallelism*** • 1 message in transit • N messages in transit	***Sync*** • (Send/ Receive) ***Async*** • (One-way msgs) ***Msg. Push*** ***Msg. Pull*** ***Passive*** ***Pro-Active*** --- ***Event & Notification Services*** ***Publish / Subscribe***	***Data Representation*** • CORBA CDR • MQC CDR • ... ***Protocols*** • TCP/IP • UDP • PPP • 1-wire ***Gateways*** • Data representation • Transports • Protocols	***Message Types*** • Request • Reply • Locate ***Parameter Types*** • CORBA in, out, inout ***Data Types*** • char, short, long, float, double, ... ***Exceptions*** • System • User ***Message Payload*** • Fixed length • Variable length

Tabelle 3.2: Middleware Taxonomie [McK03]

Datenmengen wird die Ineffizienz bedingt durch den vielen Ballast deutlich.
Durch den Ereignisdienst und dessen zentrales Element den Ereigniskanal wird die Kommunikation um einen wichtigen Mechanismus erweitert. (Das Client-Server-Modell, auf dem CORBA fußt, ist nicht ereignisbasiert.) Der Ereigniskanal bewirkt eine Entkopplung von Erzeuger und Empfänger, indem diese die Ereignisse und die damit verbundenen Daten dem Kanal übergeben bzw. von diesem beziehen, ohne über das andere Ende (über den Ort und die Anzahl der Bereitsteller bzw. Verbraucher, oder ob die Daten sicher empfangen wurden) Bescheid wissen zu müssen.
Ein wesentlicher Nachteil des Ereignisdienstes ist seine Unzuverlässigkeit beim Weiterleiten der Ereignisse [TvS03]. Falls die Verbindung des Erzeugers oder Verbrauchers zum Ereigniskanal unterbrochen wird (bzw. der Verbraucher erst nach Ereignisauftritt eine Verbindung hergestellt hat), können Ereignisse verloren gehen, da keine Persistenz garantiert ist.
Das Notification Service kann als eine Erweiterung des Event-Services angesehen werden [ACW+], [TDP04]. Zum einen erlaubt es die Filterung von Events, d.h. es werden nicht mehr alle Events zu allen Verbrauchern geschickt, sondern nur bestimmte Event-Typen werden über den Notification Channel an interessierte Verbraucher weitergeleitet. Zum anderen beinhaltet das Notification Service Erweiterungen in Bezug auf die QoS, z.B. bietet das CORBA Notification Service Unterstützung für Verlässlichkeit (sowohl bezüglich Verbindungen als auch Events –> persistente Events). Die zentrale Architektur birgt jedoch auch einige Nachteile (siehe Kapitel 2.2).

Bezüglich der Kommunikationsmechanismen wird sowohl die kontinuierliche Datenverteilung als auch die ereignisbasierte Verteilung unterstützt [Sie03]. Beide Mechanismen werden durch den Event-Kanal ermöglicht, der im Wesentlichen dem in Kapitel 2.1 beschriebenen Publish-Subscribe-Mechanismus entspricht. Die Ineffizienz bedingt durch die Realisierung mittels RMIs wurde bereits erwähnt. Des Weiteren kann es zu Integritätsproblemen kommen: die dem Event anhaftenden Daten müssen für den verarbeitenden Algorithmus von derselben Quelle stammen.
Eine Ereignisbestätigung durch den Benutzer ist nicht vorgesehen. Ereignisbenachrichtigungen hingegen werden unterstützt.

CORBA an sich bietet nur sehr eingeschränkte Möglichkeiten der Ende-zu-Ende-Unterstützung von QoS [TDP04]. Beispielsweise können keine Anforderungen bezüglich Latenz, Jitter oder Bandbreite berücksichtigt werden.
RT-CORBA hingegen als Erweiterung von CORBA kommt für DRE-

Anwendungen prinzipiell in Frage, da eine Ende-zu-Ende-Voraussagbarkeit durch das Prioritäten-Modell gewährleistet wird.
Es können verschiedene Scheduling-Strategien verarbeitet werden: Neben einer Priorität (z.B. first-come-first-serve oder die höchste Priorität „siegt“) können Nachrichten auch mit einer Deadline versehen werden, d.h. wenn die zur Verfügung stehenden Ressourcen knapp werden, können Nachrichten mit kritischer Deadline zuerst versendet werden [Sie03]. Dennoch findet sich derzeit keine CORBA-Implementierung, die sowohl funktionale als auch nicht-funktionale Eigenschaften ausreichend unterstützt (z.B. Sicherheit und Fehlertoleranz). Über die Diskrepanz zwischen FT-CORBA (Fault-Tolerant CORBA) und RT-CORBA kann man beispielsweise in [NDP+05] lesen. (Eine Einführung in FT-CORBA gibt [MT]; der FT-CORBA Standard ist auf `http://www.omg.org/cgi-bin/doc?formal/01-09-29` erhältlich.)

Im Zusammenhang mit QoS muss noch ein interessanter Ansatz von BBN Technologies erwähnt werden [Rod02], [BBN]: Das QuO (Quality Objects) - Framework ermöglicht es dem Entwickler von verteilten Anwendungen seine QoS-Anforderungen in Form von Kontrakten zwischen Clients und Serviceanbietern (in einer aspektorientierten Sprache) zu spezifizieren. Die Umsetzung erfolgt letztendlich mittels (RT-) CORBA (oder Java RMI) unter Zuhilfenahme von Netzwerktechnologien wie z.B. RSVP oder DiffServ.
Eine CORBA-Applikation wird durch QuO um folgende Komponenten erweitert [LSZB98]:

- Delegierte: Dieser stellt das gleiche Interface wie das entfernte Objekt zur Verfügung und kann zusätzlich eine Vertragsbeurteilung bei jedem Methodenaufruf einleiten. Er muss auf den aktuellen QoS-Zustand entsprechend reagieren.
- QoS Kontrakt: Darin sind die QoS-Erwartungen des Clients und des Objekts sowie Verhaltensregeln für entsprechende Zustände (bzw. Zustandsänderungen) spezifiziert.
- Systemzustands-Objekte: Sie dienen der Messung und Kontrolle der QoS.

Abbildung 3.6 zeigt ein Beispiel eines RMI unter Verwendung von QuO. Der QuO-Kernel ist multi-threaded ausgeführt.
Für eine weiterführende Darlegung sei z.B. auf [ZBS97] verwiesen.

In [HRHS01] wird die sogenannte CQoS (Configurable QoS) Architektur beschrieben, die eine QoS-Implementierung unterstützt (siehe Abbildung 3.7).

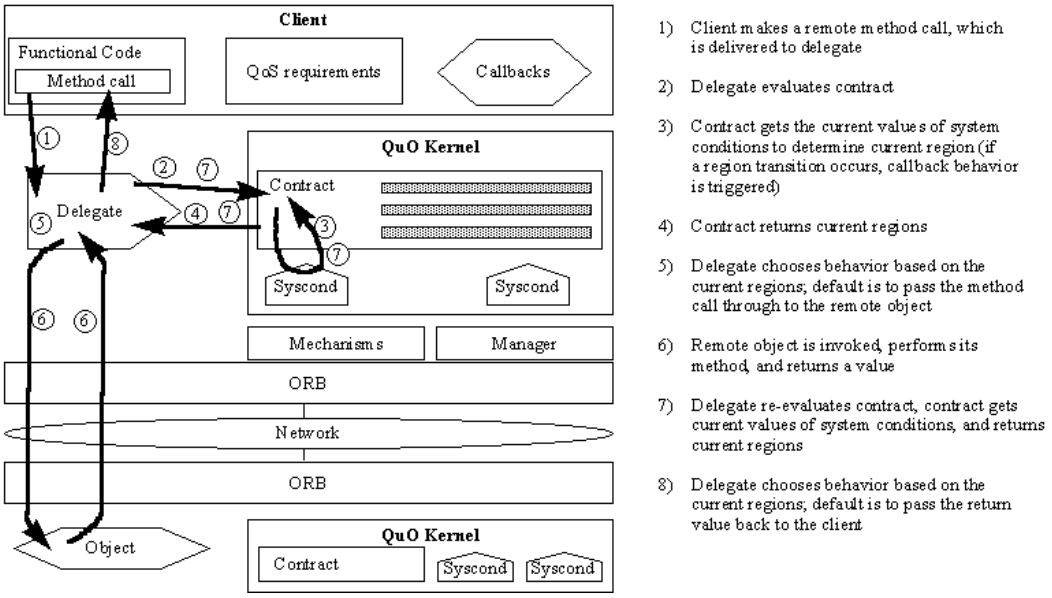

Abbildung 3.6: RMI und QuO [LSZB98]

Das Prinzip basiert auf middleware- und anwendungsspezifischen Interceptors und generischen QoS-Komponenten, die mittels Cactus - einem System für die Erzeugung von Protokollen und Diensten - implementiert werden können und kann auf sämtliche Plattformen, die einen Request-Reply-Mechanismus unterstützen (z.B. CORBA oder Java RMI), angewendet werden. Der Interceptor stellt der CQoS-Service-Komponente Interfaces für die Beeinflussung der Anfragen und Antworten zur Verfügung.
Eine genaue Darstellung der CQoS Architektur und der Umsetzung von QoS mittels Cactus findet sich in [HRHS01].

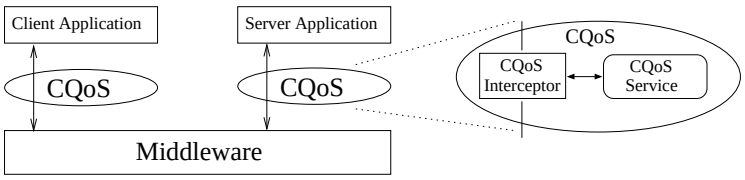

Abbildung 3.7: Prinzip der CQoS Architektur [HRHS01]

Schließlich sei auf eine OMG-Spezifikation bezüglich der QoS-

Implementierung für CORBA-Komponenten hingewiesen [OMGf]. Darin werden Konzepte beschrieben, die eine Trennung von funktionalen und nicht-funktionalen Aspekten ermöglichen sollen.

Zum Thema Ressourcenbeschränkung ist zu sagen, dass sämtliche CORBA und RT-CORBA Implementierungen einen zu hohen Speicherbedarf für eingebettete Systeme haben (deutlich über 1 MByte, vgl. TAO) [MDD+03]. Auch die Minimum-CORBA-Spezifikation der OMG[6], die viele Features (u.a. dynamische Interfaces oder den dynamischen Datentyp Any) weglässt und somit den Speicherbedarf (im Vergleich zu CORBA) um ca. die Hälfte reduziert, ändert an dieser Tatsache nur wenig. Entweder der Speicherbedarf ist zu hoch für DRE-Systeme oder aber es gibt keine Echtzeitunterstützung.
Der Mitte 2006 eingeführte Standard CORBA/e (e steht für embedded) ist ein neuer Akzent der OMG in Richtung Unterstützung von DRE-Systemen [Sie06a], [OMGa]. Darin werden das Kompakt-CORBA/e- und das Micro-CORBA/e-Profil definiert, die beide Echtzeit-Aspekte berücksichtigen und jeweils für unterschiedliche Endsysteme geeignet sind. Kompakt-CORBA/e ist für Systeme mit geringen Ressourcen (beispielsweise für 32-Bit Prozessoren mit RTOS) geeignet, während Micro-CORBA/e für besonders ressourcenarme Geräte (z.B. für Handys oder Schwachstrom-Mikroprozessoren) ausgelegt ist. Auch wenn dieser Standard vielversprechend erscheint, so sind zu diesem Zeitpunkt keine Implementierungen bekannt, bzw. ist abzuwarten, ob den Echtzeitanforderungen auch wirklich genüge getragen werden kann.

Um einen Eindruck über die Vielfalt an CORBA-Anbietern bzw. die verfügbaren CORBA-Implementierungen (z.B. e*ORB, JacORB, legORB, MICO, OpenFusion TCS, TAO, VBOrb etc.) zu bekommen, sei auf folgende URLs verwiesen: [OMGb], [OMGe], [Val].

ROFES

Eine besondere Erwähnung verdient die an der RWTH (Rheinisch-Westfälische Technische Hochschule) Aachen am Lehrstuhl für Betriebssysteme entwickelt Middleware ROFES (Real-Time CORBA for embedded Systems) [ROF]. Es handelt sich hierbei um einen RT-CORBA Prototypen unter gleichzeitiger Berücksichtigung der minimumCORBA-Spezifikation. Der ORB ist wie eine Mikrokern-Architektur ausgeführt - nur die benötigten Komponenten werden geladen. Laut [Pic04] beträgt der Speicherbedarf

[6] Eine gute Übersicht über verfügbare OMG-Standards im Zusammenhang mit DRE-Systemen gibt [Sie07].

von ROFES (C++ Library) rund 324 KB. Weitere Informationen über die Verfügbarkeit, benötigte Tools und Compilers, unterstützte Plattformen und Netzwerk-Architekturen etc. finden sich auf der Projekt-Homepage [ROF].

MicroQoSCORBA (MQC)
Auch die an der School of Electrical Engineering & Computer Science an der Washington State University entwickelte Middleware MicroQoSCORBA (MQC) scheint sehr vielversprechend zu sein [MDD+03]. Es handelt sich hierbei um ein Middleware Framework, das sowohl an Applikations- als auch an Geräteanforderungen angepasst werden kann und speziell für speicherarme Anwendungen konzipiert wurde. Es werden eine Reihe an QoS-Eigenschaften unterstützt, z.B. Fehlertoleranz, Sicherheit und Echtzeit, und auch das Hinzufügen weiterer QoS-Mechanismen ist möglich.
MQC wurde ursprünglich in Java entwickelt, wobei mittlerweile auch eine (bezüglich der QoS) eingeschränkte C++ Version existiert. Obwohl MQC keinen harten Echtzeitanforderungen genügt, so ist der Jitter im Vergleich zu anderen RT-CORBA-Produkten doch sehr gering (siehe „MW Comparator“ auf `http://www.atl.external.lmco.com/projects/QoS/`). Momentan läuft MQC unter Linux, das Portieren auf Windows oder andere Unix-Systeme sollte jedoch auch möglich sein.
Mehr Informationen zu MQC finden sich neben [MDD+03] auch in [McK03]. Unter [Mica] wird in Zukunft der aktuelle Projektstatus abzufragen sein; mit einer Open Source Version inklusive ausführlicher Dokumentation kann jedoch frühestens im Herbst 2007 gerechnet werden.

DDS
Durch das verwendete Publish-Subscribe-Prinzip wird eine kontinuierliche Datenverteilung, die Echtzeit-Anforderungen genügt, ermöglicht [Sie03]. Ebenso wird durch den Publish-Subscribe-Mechanismus eine ereignisbasierte Verteilung unterstützt. Durch diverse QoS-Mechanismen (Sende- und Empfangsschlangen, Senden von Bestätigungen bzw. erneutes Senden) kann obendrein Verlässlichkeit garantiert werden (siehe unten). Allerdings kann es ähnlich wie bei CORBA zu Problemen mit der Konsistenz der mit einem Event verbundenen Daten kommen.
Während DDS auch für Ereignisbenachrichtigung verwendet werden kann, gibt es bezüglich der Ereignisbestätigung keine Unterstützung auf Applikationsebene, dafür ist ein Request/Reply-Mechanismus vorgesehen.

Die große Stärke des DDS liegt in einer möglichen QoS-Kontrolle [PCFW05], [OMGc]. Die einzelnen QoS-Parameter können alle Entities betreffen (Topic,

	QoS Policy	QoS Policy	
Volatility	DURABILITY	USER DATA	User QoS
	HISTORY	TOPIC DATA	
	READER DATA LIFECYCLE	GROUP DATA	
	WRITER DATA LIFECYCLE	PARTITION	Presentation
Infrastructure	LIFESPAN	PRESENTATION	
	ENTITY FACTORY	DESTINATION ORDER	
	RESOURCE LIMITS	OWNERSHIP	Redundancy
	RELIABILITY	OWNERSHIP STRENGTH	
Delivery	TIME BASED FILTER	LIVELINESS	
	DEADLINE	LATENCY BUDGET	Transport
	CONTENT FILTERS	TRANSPORT PRIORITY	

Tabelle 3.3: QoS beim DDS [PCH05]

Data Writer, Publisher, Data Reader, Subscriber und Domain Participant), d.h. es kann eventuell zu Widersprechungen zwischen der geforderten QoS auf Subscriber- und Publisher-Seite kommen. Dies ist beispielsweise dann der Fall, wenn der Subscriber eine verlässliche Kommunikation fordert, der Publisher aber nur eine best-effort-Kommunikation zur Verfügung stellt. Nur wenn die auf Subscriber-Seite geforderte und die auf Publisher-Seite offerierte QoS kompatibel sind, kommt eine Kommunikation zustande.
Diese Tatsache wird durch das RxO (Requested / Offered) Property berücksichtigt: wenn RxO „Yes“ ist, dann kann die QoS auf beiden Seiten angegeben werden, muss aber übereinstimmen. Bei RxO = „No“, kann die QoS auf beiden Seiten unabhängig voneinander angegeben werden. RxO = „N/A“ bedeutet, dass eine QoS-Angabe nur auf einer Seite möglich ist.
Die einzelnen QoS-Parameter sind in Tabelle 3.3 zusammengefasst und werden im Folgenden kurz erklärt. Auf etwaige Abhängigkeiten zwischen den Parametern wird an dieser Stelle nicht eingegangen.

- Durability: Durch diesen Parameter wird eine Möglichkeit geschaffen, vergangene Nachrichten (samples) für neu hinzukommende Data Readers (im System) zu speichern. Es gibt drei mögliche Einstellungen: Volatile (keine Speicherung), Transient (Ablegen im Speicher) und Persistent (nicht-flüchtige Speicherung z.B. auf der Hard Disk).

- History: dient der Speicherung von Samples in der History Queue des Data Writers für die spätere Versendung. Es können entweder die letzten N samples gespeichert werden oder alle („Keep All“). Dies kann dann nützlich sein, wenn eine neue Applikation zur Domäne hinzukommt und einen eventuellen „Rückstand“ aufholen muss.

- Reader Data Lifecycle: Es kann angegeben werden, wie lange ein Data Reader seine Daten für eine Topic-Instanz behält, wenn kein Data Writer mehr Daten von dieser Instanz sendet. Dadurch können zusätzliche Ressourcen geschaffen werden.

- Writer Data Lifecycle: Daten für Topic-Instanzen, die nicht mehr registriert sind, werden aus der Queue des Data Writers entfernt.

- Lifespan: gibt die Gültigkeitsdauer einer Nachricht an. Nach Ablauf dieser werden die Daten gelöscht (sowohl in den Data Reader Caches, als auch in den transienten und persistenten History Queues).

- Entity Factory: Alle Factory-Entities können entweder gemeinsam bei ihrer Erzeugung durch die Factory enabled werden (sie werden damit im Netzwerk sichtbar und können an der Kommunikation teilnehmen) oder zu einem späteren Zeitpunkt.

- Resource Limits: erlaubt die lokale Ressourcenbeschränkung durch das Setzen der maximalen Anzahl an Instanzen für ein Topic, der maximalen Data Samples für ein Topic und der maximalen Data Samples für eine Instanz. Dies kann nützlich sein, wenn mehrere Applikationen die Ressourcen eines Knoten beanspruchen.

- Reliability: Mit diesem Parameter kann über ein verlässliches Senden und Empfangen von Daten entschieden werden. Das Setzen auf RELIABLE bedeutet, dass keine Nachrichten versäumt werden dürfen - versäumte Nachrichten werden erneut versendet. Bei einer BEST-EFFORT-Konfiguration unterbleibt ein erneutes Senden. Zusätzlich (in Verbindung mit RELIABLE) kann eine maximale Wartezeit (max_blocking_time) angegeben werden, die anzeigt, wie lange ein Data Writer im Falle einer vollen Queue blockiert. In dieser Zeit sendet der Data Writer keine Daten, da kein Platz für deren Zwischenspeicherung (für den Fall eines erneuten Sendens) existiert. Nach Ablauf von max_blocking_time gilt die write-Operation als erfolglos, und es erfolgt eine TIMEOUT-Meldung.

- Time Based Filter: Falls ein Data Reader nur an einer bestimmten Anzahl an Samples interessiert ist, kann eine Dauer (minimum_separation) spezifiziert werden, in der er nicht mehr als einen Wert erhält.

- Deadline: Diese soll eine periodische Aktualisierung auf Publisher-Seite garantieren, d.h. sie stellt eine Anforderung an die Applikation, indem sie eine minimale Senderate des Data Writers angibt. Auf Subscriber-Seite stellt dieser Parameter eine Anforderung für den Publisher dar, da er festlegt, wie lange der Data Reader auf ein Update wartet. Eine Kompatibilität zwischen Subscriber und Publisher resultiert nur, wenn gilt: offerierte Deadline $\leq$ angeforderte Deadline.

- Content Filters: Es werden nur Werte eines Topics empfangen und verarbeitet, die innerhalb eines festgelegten Bereichs liegen (z.B. Temperaturwerte oberhalb eines vorgegebenen Limits).

- User Data, Topic Data, Group Data: Mit diesen Parametern werden ähnliche Funktionen zur Verfügung gestellt wie durch den Parameter Partition (siehe unten). Damit wird beispielsweise eine zusätzliche Möglichkeit der Segmentation zwischen Topics oder der Authentifizierung und Identifikation geschaffen. User Data ist eine zusätzliche Information, die der erzeugten Entity von der Applikation angefügt werden kann, Topic Data wird dem erzeugten Topic angehängt und Group Data bezieht sich entweder auf den Subscriber oder den Publisher.

- Partition: Damit können Topics innerhalb einer Domäne unterschieden werden. Sowohl Publisher als auch Subscriber können diesen Wert in Form eines Strings setzen. Ein Subscriber erhält nur Nachrichten von jenen Publishern, die den gleichen Partition-String haben. Auf diese Art kann Skalierbarkeit innerhalb einer Domäne erreicht werden.

- Presentation: Hier geht es darum, wie die Daten von mehreren Data Readers eines Subscribers, sowie Änderungen dieser Daten und ihre gegenseitigen Abhängigkeiten der Applikation präsentiert werden.

- Destination Order: erlaubt der Applikation, die Reihenfolge der empfangenen Daten festzulegen. Es kann entweder nach dem Zeitpunkt des Sendens oder des Empfangens sortiert werden.

- Ownership: Damit kann bestimmt werden, ob mehrere Data Writers eine Instanz eines Datenobjekts aktualisieren dürfen. Es sind zwei mögliche Einstellungen vorgesehen: Shared (alle Data Writers dürfen updaten), Exclusive (nur ein Data Writer darf updaten, wobei der Data Writer über den Ownership Strength Parameter festgelegt wird). Durch die Parameter Ownership und Ownership Strength kann Redundanz bzw. Fehlertoleranz eingeführt werden.

- Ownership Strength: siehe Ownership

- Liveliness: legt fest, wie festgestellt wird, ob Entities noch aktiv sind. Dazu sind drei Möglichkeiten vorgesehen: Automatic (ein Lebenszeichen wird automatisch in einer frei wählbaren Frequenz gesendet), Manual by Topic (der Data Writer gilt als aktiv, wenn zumindest eine seiner Topic-Instanzen von der Applikation benötigt wird) und Manual by Participant (eine Entity ist aktiv, sobald eine andere Entity innerhalb des Domain Participant aktiv ist)

- Latency Budget: Beim Wartezeitbudget handelt es sich nur um einen Hinweis für das Service. Es ist damit die maximal tolerierbare Verzögerung zwischen dem Schreiben des Datums und seinem Einfügen im Applikations-Cache des Empfängers gemeint. Wenn man einem Topic mit niederem Latency Budget eine höhere Transport-Priorität zuweist, kann man u.U. Echtzeit gewährleisten.

- Transport Priority: erlaubt das Setzen der relativen Priorität eines Topics, wobei das Prioritätenmodell auch vom unterliegenden Transportmedium unterstützt werden muss.

Wie man sieht, stellt das DDS eine breite Palette an QoS zur Verfügung, mit der neben Fehlertoleranz und Skalierbarkeit auch andere Bereiche wie etwa eine dynamische Rekonfigurierbarkeit abgedeckt sind. (Um einen Publisher auszutauschen, muss lediglich der neue Publisher (mit gleichem Topic) in das System aufgenommen werden, worauf der alte Publisher entfernt werden kann.) Bei Blick auf Abbildung 3.8 sieht man, dass mit DDS-Middleware das ganze Echtzeit-Spektrum abgedeckt wird.

Bezüglich verfügbarer Implementierungen hat man beim DDS nicht so große Auswahlmöglichkeiten wie etwa bei CORBA. Die bekanntesten Produkte sind RTI DDS (vormals NDDS), OpenSplice von PrismTech und

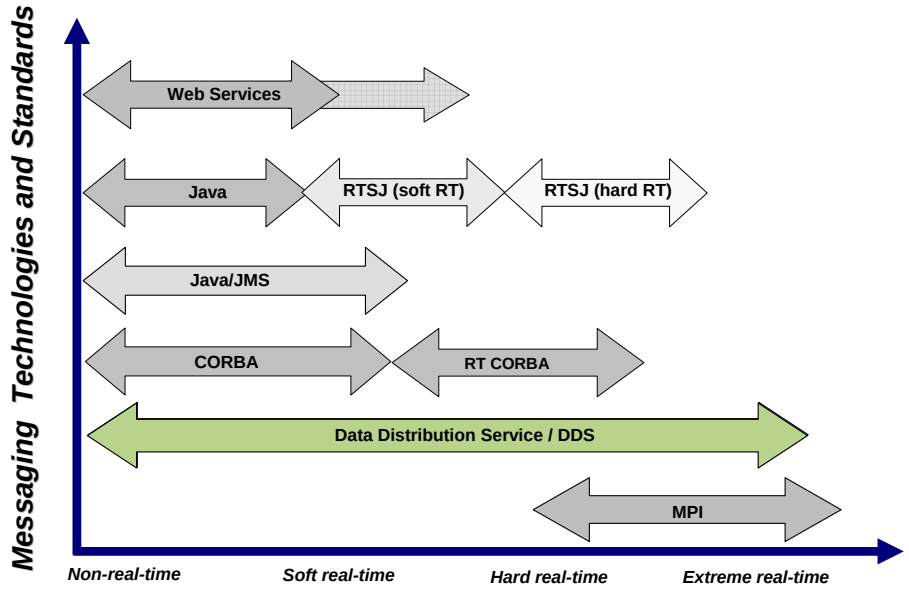

Abbildung 3.8: Real-Time unter DDS im Vergleich [PCH05]

das Open Source Produkt TAO DDS der Object Computing Inc. Weitere Anbieter finden sich beispielsweise unter [OMGd].
Obwohl TAO DDS viele Plattformen unterstützt (siehe [OCb]), ist es für die Automatisierungstechnik nicht geeignet, da für die Installation TAO (Version 1.4a oder höher) benötigt wird, das im Hinblick auf die Ressourceneinschränkungen zu umfangreich ist. (Außerdem stellt das DCPS Information Repository - das ist ein Server zur Vermittlung zwischen Publishern und Subscribern - eine bedenkliche Einschränkung („Single Point of Failure“) dar, denn ohne diesen Server gibt es kein DDS [OCa].)
Über den Ressourcenbedarf der anderen Produkte ist wenig bekannt. In einer RTI-Ausschreibung vom 23.01.2006 ist von einer speicherarmen NDDS-Version die Rede, die mit weniger als 100KB auskommt und in einer Reihe von SPSen zum Einsatz kommt [RTIb]. Es geht allerdings nicht hervor, ob der ganze DDS-Standard (und somit die ganze QoS-Palette) unterstützt wird. Außerdem ist der Preis beachtlich (ab $46,920 USD).

Java EE

Ein wesentlicher Bestandteil von Java EE ist das JMS (Java Message Service), welches ein API für die asynchrone Kommunikation darstellt [Lin06], [Sun02], [Jav]. Das JMS wird u.a. von den Message Driven Beans (siehe Kapitel

2.2.5) verwendet und ermöglicht neben einer Persistenz auch das Filtern von Nachrichten.
Grundsätzlich unterstützt der Dienst zwei Nachrichtenmodelle: Beim Versenden von Nachrichten kommen entweder Schlangen zum Einsatz, wobei hier eine Nachricht immer nur für einen Empfänger bestimmt ist, und zwischen Sender und Empfänger eine PTP-Verbindung besteht. Anderseits können Nachrichten (Ereignisse) über den Publish-Subscribe-Mechanismus mittels Topics versendet werden; in diesem Fall können mehrere Empfänger ein und dieselbe Nachricht entgegennehmen.

Das JMS-API kann auf viele nachrichtenorientierte Middleware-Produkte aufgesetzt werden, so auch auf das CORBA Notification Service. (Eine Anbindung von Java an CORBA ist ab CORBA 2.2 möglich.) Das im Zusammenhang mit dem CORBA Notification Service über die unterstützten Kommunikationsmechanismen Gesagte gilt daher sinngemäß auch für das JMS.

Das JMS bietet mit seinem speziellen Nachrichten-Aufbau auch eine (eingeschränkte) QoS-Unterstützung an. Im Nachrichten-Header sind einige Felder vorgesehen, mit denen Einfluss auf die QoS genommen werden kann. Beispielsweise kann eine Priorität zwischen 0 und 9 vergeben werden (0 ist die niedrigste, 9 die höchste Priorität), wobei Nachrichten mit einer Priorität zwischen 0 und 4 als „normal“ und solche mit höherer Priorität als dringlich gelten. Ebenso ist das Thema Verlässlichkeit berücksichtigt: Neben dem Senden von Bestätigungen (ein Client kann eine Destination für die Rückmeldung angegeben) ist auch ein Erkennungsmechanismus für wiederverschickte Nachrichten vorgesehen (zur Duplikatvermeidung). Außerdem können Nachrichten mit einem „Ablaufdatum“ (Gültigkeitsdauer) versehen und miteinander verlinkt (korreliert) werden. Beispielsweise kann eine Verlinkung von einer Anforderungsnachricht mit ihrer Antwortnachricht sinnvoll sein.

Ein großes Manko von Java bezüglich Echtzeitfähigkeit ist die Verwendung der Garbage Collection. Darunter versteht man eine automatische Speicherbereinigung, die in regelmäßigen Abständen den normalen Programmablauf unterbricht, um nicht mehr benötigten Speicher freizugeben.
Dieser Umstand hat zur Einführung der Realtime Specification for Java (RTSJ) beigetragen, die sowohl weiche als auch harte Echtzeit ermöglichen soll (vgl. Abbildung 3.8) [Sun], [RTS]. Die RTSJ führt zwei zusätzliche Thread-Arten ein, nämlich Real-Time Threads und No-Heap Real-Time Threads, welche nicht von der Garbage Collection unterbrochen werden kön-

nen. Eine bessere Zeitplanung wird auch durch die 28 Prioritätslevel erreicht. Durch eine Synchronisation der RT-Threads soll eine Prioritätsinversion (= Blockieren höherer Prioritäten durch niedrigere Prioritäten) vermieden werden.
Zwei zusätzliche Speicherplatzarten, welche nicht von der Garbage Collection beeinflusst werden, sind der Immortal Memory (Objekte werden bis zum Programmende verwaltet) und der Scoped Memory (wird nur in einem bestimmten Bereich innerhalb eines Prozesses verwaltet). Des Weiteren sind zwei Arten der asynchronen Kommunikation vorgesehen: asynchrones Event-Handling (Reaktion auf externe Events mit einstellbarer Antwortzeit) und ein asynchroner Kontroll-Transfer (ATC ... Asynchronous Transfer of Control, definiertes Unterbrechen eines Threads durch einen anderen).
Die Definition von Zeitgebern (für absolute und relative Zeitangaben), sowie die Möglichkeit eines direkten Zugriffs auf den Hardware-Speicher runden die Spezifikation ab.

Leider sind die Systemvoraussetzungen für die RTSJ-Implementierung Java Real-Time System 1.0 (nämlich ein Dual UltraSparc III Prozessor mit Solaris 10 und J2SE Version 1.4.1) viel zu hoch für die Automatisierungstechnik. (Die Verwendung eines Single-Prozessors verursacht größere Latenzen und Jitter.)

In [KDL00] wird ein auf Java basierendes Publish-Subscribe-Modell für DRE-Systeme vorgestellt. Die zugehörige Middleware-Implementierung trägt den Namen Java Embedded Bus (JEB) und ist zwischen der Java Virtual Machine (JVM) und den Java-Anwendungen angesiedelt. Der Heterogenität wird mit einem Middleware-unabhängigen Plug-and-Play JEB Gerätetreiber-Modell begegnet. Neben einem Prioritätenmodell zur Echtzeitunterstützung, sind auch Mechanismen zur Unterstützung von synchroner Kommunikation (für Kommandos und Anfragen) und Fehlertoleranz (multipoint-to-point und multipoint-to-multipoint Verbindungen sowie ein Master/Shadow-Modell, bei dem immer nur ein Publisher aktiv ist, während die anderen Publisher im „Schatten“ bleiben und nur im Fehlerfall aktiv werden) vorgesehen. Außerdem kann die Senderate zwischen Publishern und Subscribern vereinbart werden. Ansonsten finden sich keine Angaben bezüglich QoS und Ressourcenbeschränkung.

.NET
Wie schon in Kapitel 2.2.4 erwähnt wird mit der WCF neben dem herkömmlichen (synchronen) RPC-Mechanismus (in .NET Framework 2.0

auch als Remoting[7] bezeichnet) ebenso eine Kommunikation nach dem Fire-and-Forget-Prinzip sowie ein Rückruf (engl. Callback) des Clients durch das Service ermöglicht [Low].
Bei der Fire-and-Forget-Kommunikation handelt es sich um eine Einweg-Operation, d.h. es gibt nur eine Request-Nachricht, aber keine Antwort-Nachricht. Das empfangende Service kann die ankommenden Nachrichten in einer Schlange puffern. Sobald diese Schlange aber voll ist, blockiert der anfragende Client. Erst wenn die Nachricht in die Schlange aufgenommen wurde, wird die Blockierung des Clients aufgehoben.
Beim Rückruf durch das Service sind die Rollen von Client und Service vertauscht (siehe Abbildung 3.9). Rückruf-Aktionen sind sowohl innerhalb einer Maschine (NetNamedPipeBinding) als auch zwischen unterschiedlichen Maschinen (NetTcpBinding) möglich. Da sich das verbindungslose HTTP (BasicHttpBinding oder WSHttpBinding) nicht für Rückrufe eignet, bietet WCF die Möglichkeit einer HTTP-basierenden Kommunikation über zwei Kanäle (WSDualHttpBinding) - ein Kanal für die Client-Aufrufe, der andere für die Service-Aufrufe.

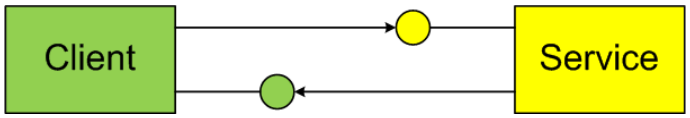

Abbildung 3.9: Callback-Aktion [Low]

Durch das Rückruf-Prinzip können relativ einfach Ereignisse implementiert werden. Sobald sich auf Service-Seite etwas tut, wird der Client verständigt, wobei das Service in diesem Zusammenhang als Publisher und der Client als Subscriber bezeichnet werden.
Allerdings wird durch den herkömmlichen Rückruf eine zu starke Bindung zwischen Publisher und Subscriber erzeugt. Jede Seite muss über den anderen Bescheid wissen, wodurch es kein anonymes Senden (auch keine Broadcasts)

[7] Das Remoting-Konzept aus dem .NET Framework 2.0 basiert auf Formatters, die die Nachrichten in ein unabhängiges Format (Bytestrom) um- bzw. zurückwandeln, und Channels, die auf unterschiedlichen Schichten aufsetzen und als Transportmedium dienen [Str], [Mar]. Neben einem HTTP-Channel für die Übertragung von SOAP-Nachrichten über das HTTP-Protokoll bietet Microsoft auch einen TCP-Channel an, bei dem die zu übertragenden Nachrichten in ein binäres Format übersetzt werden. Während der HTTP-Channel für eine Einweg- oder Request/Response-Kommunikation im Internet gedacht ist, aber keine permanente Verbindung vorsieht, kann der schnellere TCP-Channel kontinuierlich verwendet werden.

und Empfangen geben kann. Das Hinzufügen bzw. Entfernen neuer Subscriber innerhalb einer laufenden Anwendung ist schwer möglich etc.
Erst durch die Einführung eines Subscription Services und eines Publishing Services gelingt eine Entkopplung zwischen Publisher und Subscriber (siehe Abbildung 3.10).

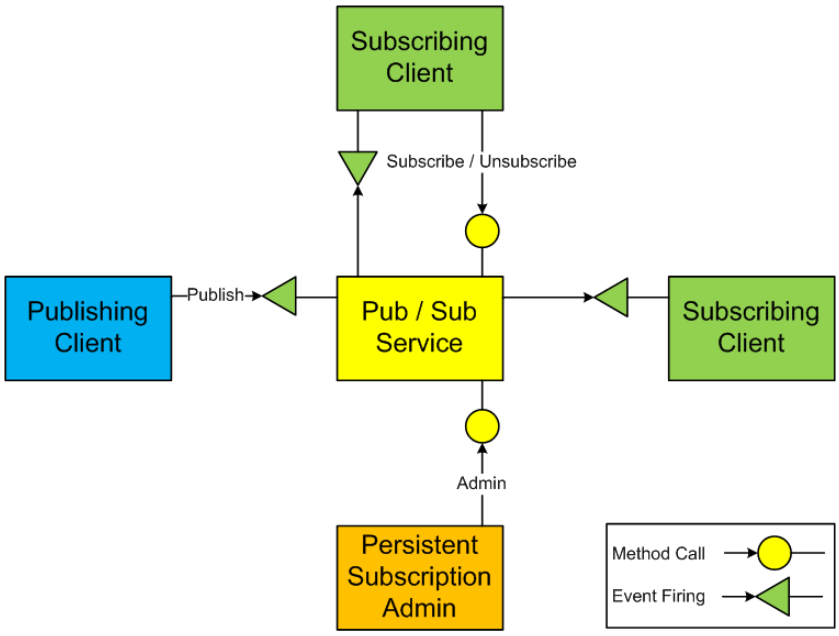

Abbildung 3.10: Publishing/Subscription Service [Low]

Jede Kommunikation läuft sodann über diese beiden Services ab. Subscriber registrieren ihr Interesse an einem bestimmten Ereignis beim Subscription Service (bzw. melden es wieder ab), Publisher übergeben ihre Ereignisse dem Publishing Service, welches die Zustellung an die interessierten Subscriber übernimmt.
Überdies können sowohl transiente als auch persistente Subscriber definiert werden. Während eine transiente Subskription beim Herunterfahren des Systems bzw. dessen Absturz verloren geht, bleibt die persistente Subskription (auf der Festplatte bzw. in einer Datenbank) bestehen und kann auch entsprechend konfiguriert werden.
Als mögliche Erweiterung können auch Schlangen bei den Publishern und Subscribern implementiert werden, wobei dann keine transienten Subskriptionen mehr möglich sind (keine Unterstützung durch MSMQ).

Auf die QoS kann teilweise Einfluss genommen werden [Micj]. Mit MSMQ[8] (Microsoft Message Queuing) sind beispielsweise Mechanismen zur Unterstützung von Sicherheit und Verlässlichkeit vorgesehen. Wenn die kommunizierenden Rechner nicht verbunden sind, erfolgt die Zustellung zu einem späteren Zeitpunkt.
MSMQ erlaubt nicht nur das gepufferte Senden und Empfangen von Nachrichten, sondern auch die Berücksichtigung von zeitkritischen Nachrichten [Dev]. Neben der Zeit, die bis zum Erreichen der Empfängerschlange verstreichen darf (TimeToReachQueue), kann für eine Nachricht auch die maximale Dauer bis zur Verarbeitung durch die Applikation (TimeToBeReceived) bestimmt werden. Wenn die Gültigkeitsdauer einer Nachricht abgelaufen ist, wird sie (je nach Einstellung) entweder verworfen oder in der sogenannten Dead-Letter Queue (diese administrative Schlange wird von MSMQ automatisch generiert) gespeichert.
Des Weiteren ist mit MSMQ eine Prioritäten-basierte Kommunikation möglich. Nachrichten können mit einer Priorität zwischen 0 (die niedrigste Priorität) und 7 (die höchste Priorität) versehen werden.
Mit WCF ist auch ein gefilterter Empfang von Nachrichten (Events) möglich.
Für ein deterministisches Verhalten unter .NET sind der Garbage Collector und der JIT-Compiler besonders hinderlich. KW-Software bietet dieses Problem betreffend eine eigene Embedded CLR (ProConOS embedded CLR) an [KS], [Pet03]. Statt des JIT-Compilers wird ein sogenannter Ahead-Of-Time (AOT) Compiler verwendet, wodurch der Zwischencode (CIL) nicht erst zur Laufzeit sondern schon vorzeitig in Maschinencode übersetzt wird.

Bezüglich Ressourcenbedarf und Systemanforderungen wird durch .NET ein zu hoher Anspruch gesetzt. Prozessoren mit mehreren Hundert MHz sowie Speicher im MB-Bereich sind hier Voraussetzung.
Auch das .NET Compact Framework (CF), das eigens für eingebettete Systeme mit beschränkten Ressourcen (speziell für mobile Geräte) entwickelt wurde, ändert an dieser Tatsache nur wenig [Micg], [Mici].
Selbst bei einer Größe von nur 8% des gesamten .NET Frameworks nimmt das .NET CF 2.0 noch immer 4,5 MB ein.
Auch wenn mit der Funktion „P/Invoke“ (Platform Invoke) der Aufruf von nicht verwaltetem Code möglich wird, empfiehlt es sich über-

[8] MSMQ ist seit Windows NT 4.0 in allen Windows Betriebssystemen enthalten und auch Bestandteil von WCF. Bei MSMQ handelt es sich um ein Nachrichtenprotokoll, das auf der Verwendung von Schlangen basiert.

dies nicht, .NET CF in Echtzeitszenarios einzusetzen (vgl. `http://www.microsoft.com/germany/msdn/library/net/compactframework/EchtzeitverhaltenVonNETCompactFramework.mspx?mfr=true`).

DCOM
Die Standard-Kommunikation in DCOM basiert auf RPCs und ist somit synchron [TvS03]. Zusätzlich werden aber auch asynchrone Aufrufe in Form einer transienten Kommunikation (d.h. Client und Objekt müssen aktiv sein) sowie Nachrichtenwarteschlange für eine persistente (asynchrone) Kommunikation unterstützt. Letztere ist mittels sogenannter Queued Components (QC) möglich, die eine Schnittstelle zu MSMQ (siehe .NET) bilden und nur einen Ein-Wege-Methodenaufruf erlauben.
Verbindungsfähige Objekte, die Callback-Schnittstellen anbieten, bieten sich auch für eine Ereignisweitergabe an (siehe .NET). In DCOM existiert zu diesem Zweck eine eigene Ereignisklasse. Objekte dieser Klasse werden über ihre CLSID identifiziert und beinhalten Methoden zur Ereigniserzeugung.

Leider gibt es in DCOM kaum Möglichkeiten der QoS-Unterstützung. Zwar sind bezüglich Sicherheit Maßnahmen für die Authentisierung, Autorisierung und Verschlüsselung vorgesehen, und auch Fehlertoleranz wird mithilfe von Transaktionen unterstützt, doch es gibt weder eine Unterstützung für Echtzeitfähigkeit noch für eine hohe Verlässlichkeit.

Nebenbei sei erwähnt, dass es auch Einschränkungen bezüglich folgender Punkte gibt (vgl. [CD01]):

- Plattform-Unterstützung (beste Unterstützung für Windows 95- und NT-Plattformen)
- Keine Plattform-Unabhängigkeit
- Sicherheit
- Hoher Einfluss von Microsoft
- Entwicklung eingestellt, derzeit noch unterstützt
- Aufwendige Entwicklung (erfordert viel Wissen)

Der Ressourcenverbrauch von DCOM ist eher gering. In diesem Zusammenhang ist die modulare Systemarchitektur MMLite zu nennen, die speziell für eingebettete Systeme entwickelt wurde, und mit der ein System mit

einem Speicherabdruck von nur 10KB realisiert werden konnte [HA98] bzw. [MDD+03]. Eine Applikation setzt sich bei MMLite aus objektorientierten Komponenten zusammen, wobei je nach Zusammensetzung unterschiedliche QoS-Eigenschaften realisiert werden können. Das Hauptaugenmerk wird dabei jedoch auf Echtzeit gelegt.

OSA+
Wie bereits in Kapitel 2.2.7 erwähnt unterstützt OSA+ sowohl eine synchrone als auch asynchrone Kommunikation, wobei diese beiden Varianten auch problemlos nebeneinander verwendet werden können [WB05].
Dafür stehen folgende Funktionen zur Option:

- SendOrder: mit diesem Befehl wird eine Job-Anforderung verschickt
- AwaitOrder: wartet auf den Eingang eines Auftrages eines anderen Dienstes; wird für eine synchrone Kommunikation verwendet, da der Dienst bis zum Auftragerhalt blockiert
- ExistOrder: fragt nach einem neuen Auftrag, der mit AwaitOrder abgeholt werden kann; dient der asynchronen Kommunikation, da der Dienst nicht blockiert
- ReturnResult: retourniert das Ergebnis einer Anforderung
- AwaitResult: Pendant zu AwaitOrder und damit für eine synchrone Kommunikation geeignet; erwartet das Ergebnis eines Auftrages
- ExistResult: falls ein Ergebnis vorliegt, kann dieses mit AwaitResult abgeholt werden, wobei dieser Vorgang asynchron abläuft

Im Zusammenhang mit der Kommunikation sind vor allem der Kommunikationsdienst und der Eventdienst wichtig. Der Kommunikationsdienst stellt eine Verbindung zwischen verschiedenen Plattformen her, indem er unterschiedliche Protokolle und Kommunikationsmedien verwendet. Dazu wird der Kommunikationsdienst in einen HLC (High Level Communication Service) und mehrere LLCs (Low Level Communication Service) unterteilt. Während die LLCs einfache Protokolle (z.B. CAN, TCP/IP etc.) zur Verfügung stellen, übernimmt der HLC das Routen von Jobs, wodurch ein Rechenknoten mehrere Kommunikationssysteme (z.B. Ethernet und Feldbus) miteinander verbinden kann.
Der Ereignisdienst ist für sämtliche zeit- und ereignisgesteuerten Aufgaben zuständig. Neben einer Job-Auslieferung zu einem bestimmten Zeitpunkt (es

können sowohl Anfangs- als auch Endzeiten definiert werden) ist auch eine periodische Abarbeitung möglich (Timer-Events). Der Ereignisdienst kann sowohl auf Events und Interrupts der Hardware als auch des Betriebssystems reagieren. Außerdem kann er für Monitoring-Zwecke (z.B. Einhaltung von Echtzeitanforderungen) herangezogen werden. Beispielsweise kann bei Nichteinhaltung einer Zeitschranke eine Fehlermeldung generiert werden.
Somit werden durch OSA+ die wichtigsten Datenstromarten (Signale, Ereignisse, Anfragen etc.) bzw. Kommunikationsmechanismen unterstützt. Das Job-Prinzip entspricht in seiner einfachen Ausprägung einem Request/Reply-Mechanismus, während durch den Ereignisdienst eine (annähernd) kontinuierliche Verteilung sowie fast alle ereignisbasierten Mechanismen abgedeckt werden. Lediglich eine Ereignisbestätigung ist nicht ohne weiteres möglich.

Bezüglich QoS wurde bei OSA+ die Möglichkeit einer flexiblen Erweiterung geschaffen [PB06].
Anstatt einer QoS-Anforderung über ein spezielles API kann bei OSA+ jedem Job eine QoS-Information angehängt werden (siehe Abbildung 3.11), wodurch die Middleware-Implementierung von der QoS-Angabe entkoppelt wird. Wenn der Benutzer einen Job angibt, definiert er gleichzeitig die QoS durch ein (key,value)-Dupel, wobei der erste Wert für eine konkrete QoS steht, während der zweite den dazugehörigen Wert angibt. Das Paar (10, 100) könnte beispielsweise eine Deadline von 100ms bedeuten (10 steht für Deadline, die zweite Zahl definiert einen konkreten Wert in ms). Für ressourcenstärkere Plattformen kann des Weiteren eine zusätzliche XML-Schicht vorgesehen werden, die die QoS-Anfragen etwas anschaulicher darstellt.
Die QoS-Information wird in Klassen eingeteilt (z.B. zeitabhängige oder kommunikationsabhängige). Jede Klasse wird von einem Dienst verwaltet und über einen eindeutigen Integer-Wert angesprochen. Der Plattform können beliebig viele QoS-Klassen hinzugefügt werden.
Für die Umsetzung der QoS ist ein QoSHandler verantwortlich, der von dem für eine QoS-Klasse verantwortlichen Service bei der Plattform registriert wird und eine Verbindung zwischen Middleware und Service darstellt. Der QoSHandler wird bei jedem QoS-behaftetem Job ausgeführt (vorausgesetzt seine Klasse entspricht der QoS-Anfrage; z.B. ist das Event-Service für die zeitabhängige QoS-Klasse verantwortlich).
QoS-Anfragen können entweder auf Sitzungsbasis (einmal zu Beginn der Sitzung bearbeitet) oder auf Nachrichtenbasis (für jede Nachricht neu bearbeitet) definiert werden.

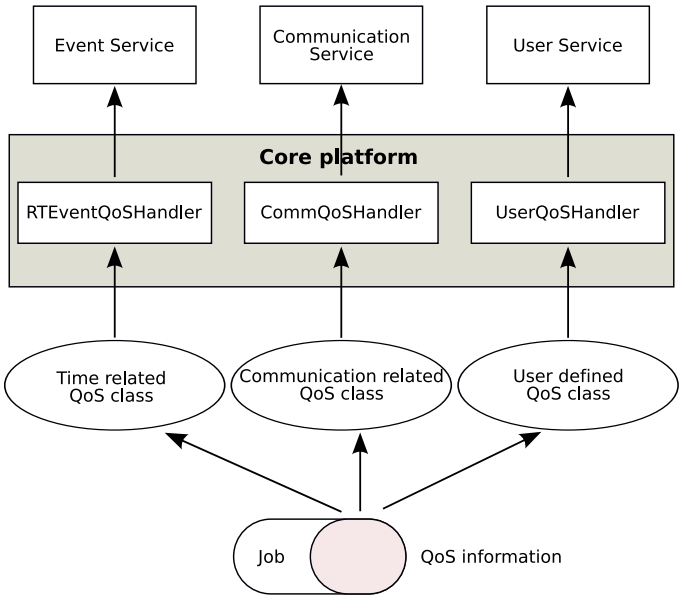

Abbildung 3.11: QoS bei OSA+ [PB06]

Die bei OSA+ vordefinierten QoS-Parameter finden sich in [Pic04]:
Die von den Kommunikationsdiensten unterstützten Parameter betreffen Echtzeit, Bandbreite, Prioritäten, Buffergrößen von Sender und Empfänger sowie Timeouts (vorausgesetzt das unterlagerte Kommunikationssystem bietet entsprechende Unterstützungen an).
Der Prozessdienst hat Einfluss auf die Prozess- bzw. Thread-Stackgröße sowie auf RT-Scheduling-Techniken und deren Parameter.
Durch den Ereignisdienst können Deadlines überwacht bzw. eingehalten werden.
Es sei darauf hingewiesen, dass OSA+ harten Echtzeitanforderungen genügt, wenn es in einer harten Echtzeitumgebung eingesetzt wird (für genaue Ergebnisse siehe [PBBS04]). D.h. für die verfügbare Java-Version ist lediglich ein RTOS mit JVM notwendig. (Es existiert auch eine C-Version von OSA+.)

Tabelle 3.4 gibt schließlich einen Überblick über den Speicherbedarf von OSA+ für die Java-Version, wobei der Sun Java Compiler für Linux, Version 1.4.2-05 verwendet wurde. Der normale Speicherabdruck ergibt sich nach Kompilieren der Source-Files und Entfernen der Debug-Information. Die Minimums-Konfiguration kann mit einem speziellen Tool zur Code-Optimierung erreicht werden.

	Minimum	Normal
OSA+ Core	28628	52705
Process Service	3316	5084
HLC Service	5695	9561
TCPIP Service	5992	7573
Init Service	880	1046
Total	44511	75969

Tabelle 3.4: Ressourcenverbrauch von OSA+ in Bytes (Das HLCService und TCPIPService sind Teil der Kommunikationsdienste) [PB06]

Fazit der Middleware-Analyse

Bezüglich der Kommunikation ergeben sich durch die Verwendung von Middleware keine größeren Einschränkungen, wenngleich keines der analysierten Produkte sämtliche Mechanismen unterstützt.
Die meisten Middleware-Produkte bestechen durch eine Vielzahl an Diensten und Möglichkeiten. Daraus resultiert jedoch unmittelbar ein erster, wesentlicher Nachteil für die IAT, nämlich der zu hohe Ressourcenverbrauch. Von den verfügbaren Produkten kann lediglich von OSA+ mit Sicherheit behauptet werden, diesbezüglich eine Ausnahme zu bilden.
Auch beim Thema Echtzeitfähigkeit gibt es bei einigen Produkten Mängel. Java EE (ohne RTSJ), DCOM und .NET scheiden diesen Punkt betreffend aus.
Letztendlich sind auch die von der Middleware unterstützten Plattformen kritisch. Diese Tatsache gilt für alle der untersuchten Produkte und stellt eine wesentliche Einschränkung dar.
Das Fazit muss somit lauten, dass kein Produkt den Anforderungen vollständig genügt, weshalb die erste Möglichkeit zur Umsetzung der IT-Konzepte (siehe Kapitel 3.1) ausgeschlossen wird.

Als Favorit der Middleware-Analyse geht DDS hervor. Im Hinblick auf eine einfache Konfiguration der Kommunikation ist vor allem die umfangreiche QoS-Unterstützung (siehe Tabelle 3.3) hervorzuheben. Diese ist bei der Nachbildung von Middleware-Funktionalität von großem Interesse (siehe Kapitel 3.5).

3.4 Verwendung von Protokollen mit QoS-Unterstützung

Protokolle dienen allgemein der Festlegung von Regeln für den Datenaustausch zwischen unterschiedlichen Geräten. Meist sind Protokolle in Schichten organisiert (vgl. OSI-Modell). Die Gesamtheit aller Schichten wird als Protokollstapel bezeichnet. Für eine Kommunikation zwischen zwei Rechnern muss dieser Stapel auf der einen Seite von oben nach unten und auf der anderen Seite von unten nach oben abgearbeitet werden. Jede Protokoll-Schicht hat eine genau definierte Aufgabe, die sie unter Verwendung der von der ihr unterliegenden Schicht zur Verfügung gestellten Dienste erfüllen muss. (Die Antwortzeit auf eine Anforderung setzt sich somit aus der benötigten Zeit am Bus und der zweifachen Zeit für die Protokollstapelabarbeitung zusammen.)
Die große Anzahl an Protokollen ist mittlerweile nicht mehr überschaubar [THJ+05]. Die Umsetzung diverser Services kann in den unterschiedlichsten Schichten erfolgen: Beispielsweise kann die Echtzeitkommunikation basierend auf Ethernet entweder über der Transportschicht implementiert sein. Als Alternative bietet sich aber auch eine Umsetzung auf den Ebenen drei und vier an (siehe Abbildung 3.12). Mit der QoS generell verhält es sich ähnlich. Eine Betrachtung auf Netzwerkebene ist ebenso möglich wie eine Implementierung auf Anwendungsebene (vgl. Kapitel 3.2.2).

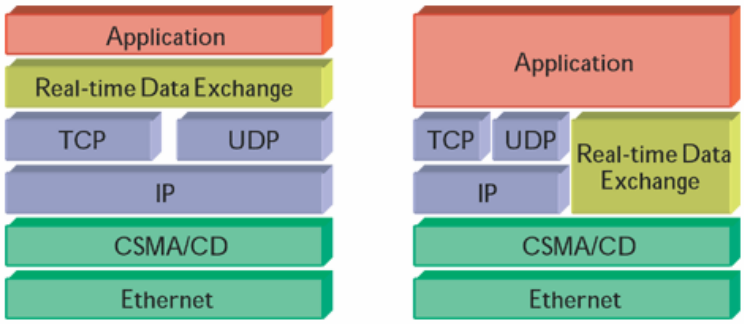

Abbildung 3.12: Echtzeit-Umsetzung im OSI-Modell [THJ+05]

In diesem Zusammenhang treten unmittelbar die Schwierigkeiten bei der Implementierung von Services kraft Protokollen in den Vordergrund: Protokolle sind häufig herstellerspezifisch bzw. an ein bestimmtes Kommunikationsmedium gebunden. Zum einen kann oft nicht die geforderte Portabilität (Plattformunabhängigkeit) innerhalb verteilter Systeme gewährleistet werden,

andererseits mangelt es den Anwendungen häufig an Interoperabilität - eine Zusammenarbeit heterogener Rechner ist schwer bis gar nicht möglich. D.h. Protokolle schränken die möglichen Zielsysteme ein, indem sie nicht überall verfügbar sind, bzw. stehen durch die Verwendung mehrerer unterschiedlicher Protokolle nicht allerorts die gleichen Features zur Verfügung.

Die Umsetzung von Services bzw. von QoS mittels Protokollen ist nicht Hauptaugenmerk dieser Arbeit und wurde daher nicht im Detail untersucht. Eine genauere Auseinandersetzung ist jedoch angesichts der besagten Schwierigkeiten vermeintlich nicht sinnvoll, bzw. scheint kein Produkt den Anforderungen vollständig zu genügen, weshalb diese Lösungsvariante prinzipiell ausscheidet.

Es sei darauf hingewiesen, dass der DDS-Standard sehr eng mit dem RTPS-Protokoll verbunden ist. (Die RTPS-Spezifikationen wurden ursprünglich von RTIs NDDS implementiert.) DDS ist jedoch umfangreicher als RTPS (bessere Interoperabilität, mehr Kommunikationsmechanismen etc.).

3.5 Nachbilden von Middleware-Funktionalität

Wie wir gesehen haben, bieten die einzelnen Middleware-Konzepte sehr nützliche und hilfreiche Eigenschaften, auf die man auch im Bereich der Automatisierungstechnik zugreifen bzw. über die man verfügen möchte. Allerdings bietet keine der beschriebenen Lösungen eine vollständige Befriedigung bezüglich der Anforderungen. Entweder gibt es Mängel im Bereich der offerierten Kommunikationsmechanismen oder aber den geforderten QoS- und Ressourcenansprüchen kann nicht begegnet werden.
Auch wenn der unmittelbare Einsatz von Middleware verlockend erscheint, so resultiert dennoch ein gewisser Aufwand (nicht nur ein Arbeitsaufwand sondern eventuell auch ein finanzieller Aufwand, vgl. RTI DDS). Eine u.U. zeitaufwendige Einarbeitung bzw. ein Erlernen der Programmierung (vgl. Kapitel 2.2) der jeweiligen Middleware ist in diesem Zusammenhang unausweichlich.

Um nicht gänzlich auf die Vorteile von Middleware verzichten zu müssen, ist eine Imitation von Middleware-Funktionalität denkbar. Bevor jedoch mit irgendeiner Implementierung begonnen werden kann, sollten die interessierenden Mechanismen losgelöst von einem bestimmten Produkt rein

funktional und gleichzeitig in abstrakter Form beschrieben werden. Erst dann kann man sich über eine genaue Umsetzung der QoS-Parameter mit Mitteln der IEC 61499 den Kopf zerbrechen (siehe Kapitel 4).
Für eine allgemeine Darstellung von Interaktionen und Zustandswechsel eignen sich besonders Sequenz- bzw. Zustandsdiagramme. Einen besonderen Stellenwert nimmt hier die UML (Unified Modelling Language) ein, die im Folgenden als hilfreiches und zugleich anschauliches Visualisierungswerkzeug herangezogen wird. (Die nachstehenden Diagramme sind nur an die UML angelehnt, d.h. dass weniger auf eine korrekte UML-Notation, dafür aber auf Logik und leichtes Verständnis geachtet wurde. Die Diagramme sind in den meisten Fällen selbstsprechend.)
Auch auf die SDL (Specification and Description Language) wird zurückgegriffen, da sie vor allem für die Kommunikationsbeschreibung (auch für die Spezifizierung von Protokollen) bestens geeignet ist. Im Anhang findet sich eine kurze Erläuterung zur SDL-Notation.

Bei Blick auf die bei der QoS-Umfrage genannten Parameter und der vom DDS zur Verfügung gestellten QoS sieht man, dass viele der Größen übereinstimmen. Bei den folgenden funktionalen Beschreibungen der QoS-Mechanismen erfolgt in erster Linie eine Konzentration auf die (auf Publish/Subscribe basierenden) DDS-Größen (auch in Bezug auf die Benennungen). Natürlich wird auch auf die Umfrage-Parameter eingegangen.

Auf einen Umstand sei noch hingewiesen: Die nachstehenden Diagramme beinhalten des Öfteren beispielsweise zwei Clients oder zwei Publisher und zwar in folgender Notation: CLIENT_QoS und CLIENT (bzw. PUBLISH_QoS und PUBLISH). Das heißt jedoch nicht, dass an dieser Kommunikation zwei Clients (im eigentlichen Sinn) teilnehmen, sondern vielmehr, dass ein Objekt (CLIENT_QoS) benötigt wird, das sämtliche Anfragen der Applikation entgegen nimmt und diese dann an den eigentlichen Client (CLIENT) oder andere Objekte delegiert. Diese Vorgangsweise entspricht der in Kapitel 4 beschriebenen Wiederverwendung vorhandener SIFBs, die um QoS-Möglichkeiten auf Applikationsebene ergänzt in neuen „Composite SIFBs“ gekapselt werden.

3.5.1 Allgemeiner Initialisierungsprozess

Zunächst sei der - jeglicher Kommunikation vorausgehende - Initialisierungsprozess erklärt (siehe Abbildung 3.13).

Dieser wird von der Applikation durch Aufruf eines entsprechenden vom Kommunikationsteilnehmer (Server, Client, Publisher, Subscriber) implementierten Algorithmus (INIT) angestoßen. Zusätzlich müssen die dafür benötigten Parameter übergeben werden. Dazu gehören u.a. eine ID, eventuell Informationen bezüglich der QoS bzw. andere Informationen das Service betreffend. Der eigentlichen Initialisierung folgt ein Rückgabewert (STATUS) an die Applikation. Dieser teilt der Anwendung mit, ob der Initialisierungsvorgang erfolgreich war oder nicht. Je nach Ergebnis kann entweder mit der Kommunikation begonnen werden, oder der Vorgang muss wiederholt werden.

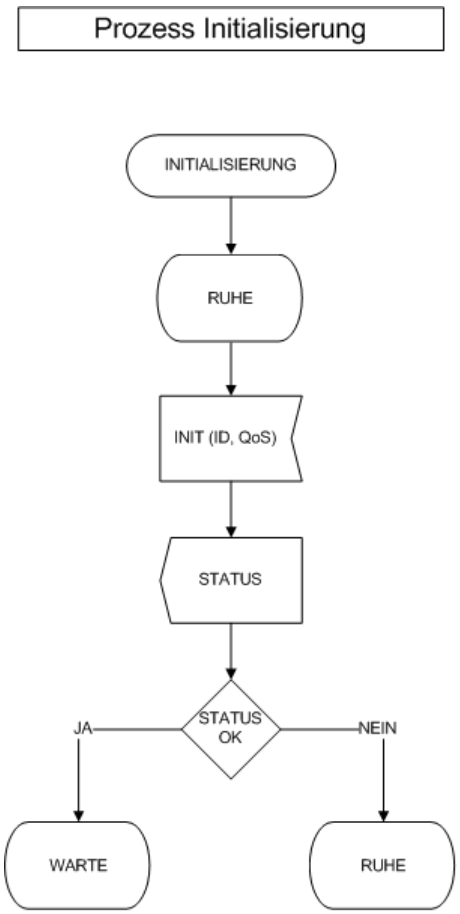

Abbildung 3.13: Allgemeiner Initialisierungsprozess

3.5.2 Berücksichtigung der Reihenfolge bei der Initialisierung (Server vor Clients)

Bei der Initialisierung von Clients und Servern ist die Reihenfolge der Anmeldung wichtig. Bevor eine Kommunikation stattfinden kann, muss sich zuerst der Server initialisieren. Erst nach einer erfolgreichen Initialisierung des Servers ist eine Anmeldung des Clients beim Server über dessen ID möglich.
Die Initialisierung der einzelnen FBs wird oft als Kette durchlaufen, d.h. das INITO-Ereignis eines FBs (vgl. Kapitel 2.4) dient als Auslöser einer weiteren FB-Initialisierung. Beim Laden bzw. Starten einer verteilten Applikation werden oft zig solcher Ketten gleichzeitig durchlaufen, sodass der genaue Initialisierungszeitpunkt eines FBs im Allgemeinen nicht bestimmt werden kann. Daraus resultieren aber Schwierigkeiten bei der Client- und Serveranmeldung (bezüglich der korrekten Reihenfolge).
Als Ausweg kann bei einer erfolglosen Client-Initialisierung ein neuerlicher Initialisierungsversuch unternommen werden - dies solange, bis der Vorgang erfolgreich war. Konkret wird der Applikationsentwickler festlegen wollen, wie oft ein neuerlicher Versuch gestartet werden soll (um eine Endlosschleife zu vermeiden). Außerdem wird er über die Dauer zwischen zwei Initialisierungsversuchen entscheiden wollen.
Abbildung 3.14 zeigt ein mögliches Szenario bei der Client-Initialisierung: Beim ersten Versuch der Applikation, den Client zu initialisieren, wird eine Negativmeldung retourniert, d.h. der Initialisierungsstatus (INIT_STATUS) hat den Wert „INIT NOT SUCCESSFUL“. Sodann versucht der FB CLIENT_INIT eine neuerliche Initialisierung - wieder ohne Erfolg. Bevor der dritte Versuch unternommen wird, kommt es zu einer erfolgreichen Server-Initialisierung, worauf auch die Anmeldung des Clients funktioniert (INIT_STATUS = INIT SUCCESSFUL).
Wie bereits erwähnt wird diese Prozedere durch zusätzliche benutzerdefinierte Parameter (Anzahl der Versuche und Dauer zwischen den Versuchen) beeinflusst werden. Diese Parameter müssten beim ersten INIT-Aufruf durch die Applikation zusätzlich zur ID mitgegeben werden.

3.5.3 Antwortzeiten auf Anforderungen

Oft ist es wichtig, dass gewisse Daten innerhalb einer bestimmten Zeit an ihr Ziel gelangen.
Für einen Client, der von einem Server Daten anfordert, ist es u.U. notwendig, dass seine Anfrage innerhalb einer definierten Zeit beantwortet wird. Um

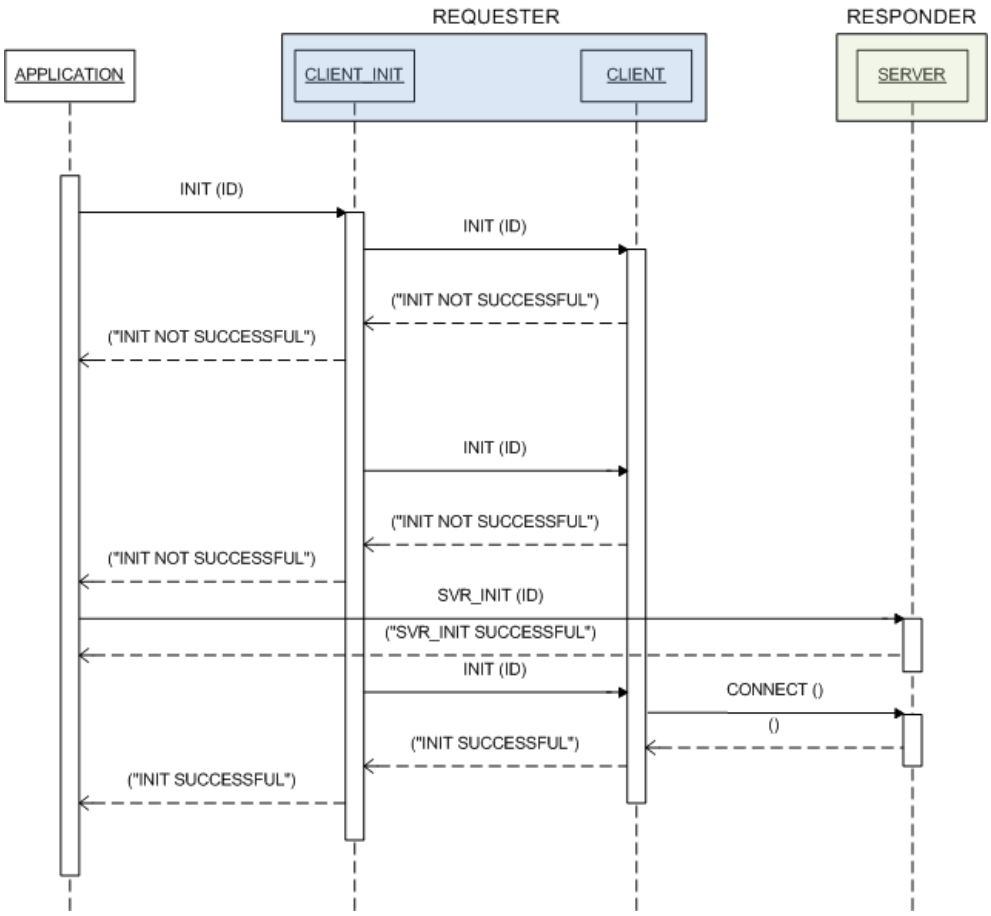

Abbildung 3.14: Berücksichtigung der Reihenfolge bei der Initialisierung

überprüfen zu können, ob die Antwort rechtzeitig einlangt, muss der Client unmittelbar vor Versenden der Anfrage eine Stoppuhr starten, die er mit Erhalt der Antwort wieder anhält[9]. Sodann muss er nur noch die mit der Stoppuhr ermittelte, verstrichene Zeit mit seiner Zeitanforderung vergleichen und kann sogleich auf die Rechtzeitigkeit und damit auf die Gültigkeit (bzw. Verwertbarkeit) der Daten schließen. Der Vorgang zur Ermittlung der Rechtzeitigkeit der angeforderten Daten ist in Abbildung 3.15 dargestellt.

[9]Eigentlich müsste die Stoppuhr genau *zum Zeitpunkt des Absendens* gestartet werden. Durch Starten der Stoppuhr *vor* der Anfrage ist man jedoch immer auf der „sicheren Seite“, da dann die gestoppte Zeit größer ist als die tatsächlich verstrichene. Analog dazu sollte die Stoppuhr erst *nach* Eingang der Antwort gestoppt werden.

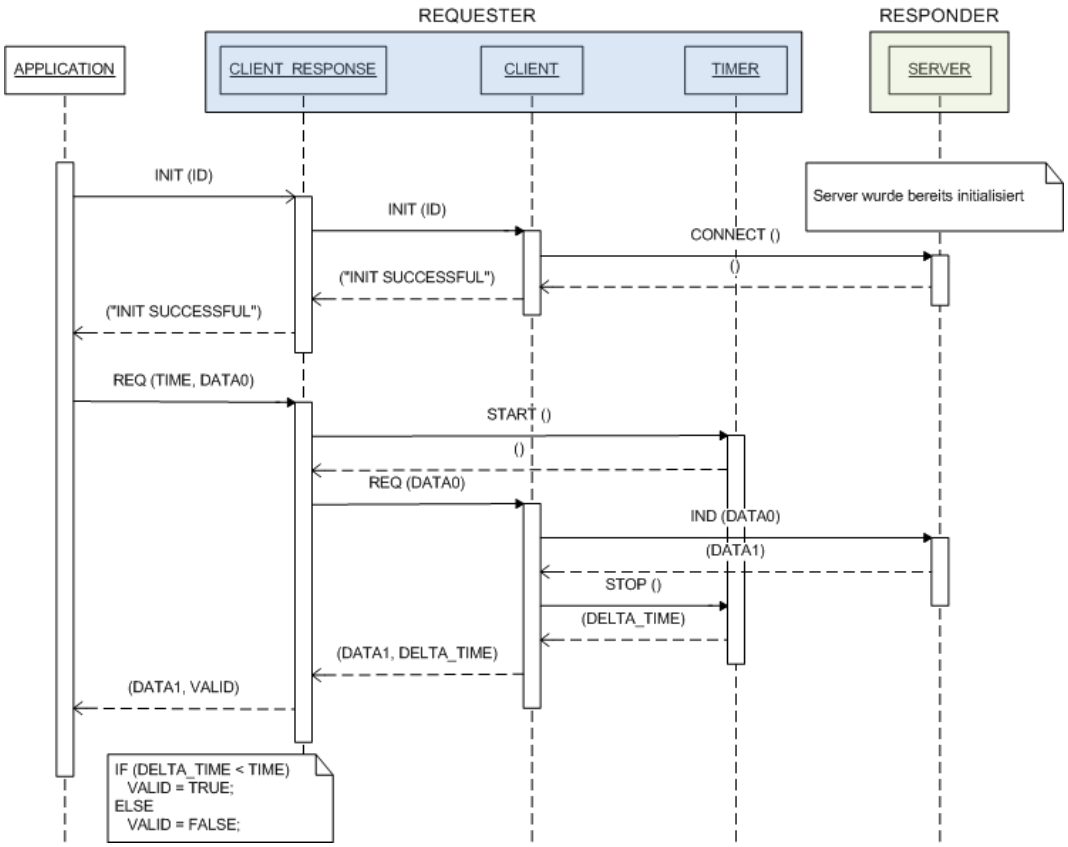

Abbildung 3.15: Maximale Antwortzeiten

3.5.4 Berücksichtigung von Deadlines

Eine Berücksichtigung von Deadlines kann sowohl auf Publisher- als auch auf Subscriber-Seite erfolgen. (Der Begriff Deadline ist hier wie im Kapitel 3.3 erklärt zu verstehen.)
Beim Publisher muss zu diesem Zweck lediglich die Zeit zwischen zwei aufeinanderfolgenden Sende-Ereignissen gemessen werden. Wenn diese gemessene Zeit größer als die durch die Applikation spezifizierte Zeit (Deadline) ist, erfolgt eine Benachrichtigung an die Applikation (beispielsweise in Form eines Flags).
Auf Subscriber-Seite wird analog verfahren. Die Applikation hat im Falle einer nichteingehaltenen Deadline dafür Sorge zu tragen, dass der Publisher in schnelleren Abständen sendet.

3.5.5 Einführung einer Sendeschlange

Durch die Verwendung einer Warteschlange im Publisher wird eine Pufferung der zu sendenden Daten ermöglicht. Dadurch kann der Publisher von der Applikation insofern entkoppelt werden, dass er nicht mehr im gleichen Tempo publizieren muss, wie Daten von der Applikation kommen (man spricht in diesem Zusammenhang auch von Asynchronität). Vielmehr kann nun der Benutzer festlegen, mit welcher Rate gesendet wird.
Eine Schwierigkeit bei der Implementierung kann u.U sein, dass die Schlange nicht beliebig viele Elemente aufnehmen kann. Im Falle eines „Überlaufens“ kann entweder das neue oder das älteste Element verworfen werden.

3.5.6 Mitverschicken der Historie

Für eine Applikation, die erst zu einem späteren Zeitpunkt (im Vergleich zu anderen Applikationen) im verteilten System gestartet wird, aber auch beim Hinzufügen und Austausch von Devices kann es nützlich sein, vergangene (alte) Nachrichten diverser Publisher zu empfangen. Zu diesem Zweck müssen besagte Publisher mit der aktuellen Nachricht auch die bereits versendeten Daten mitverschicken. D.h. der Publisher hat dafür Sorge zu tragen, seine Historie in geeigneter Form zu speichern. Die dafür notwendige Datenstruktur muss in der Lage sein, die letzten N Samples inklusive Zeitstempel zu speichern und bei Bedarf alle gleichzeitig (z.B. in Array-Form) auszugeben. (Es wäre wünschenswert, wenn der Benutzer die Zahl N festlegen kann.) Für eine kurzzeitige Speicherung, die automatisch die Reihenfolge des Einlangens berücksichtigt, eignet sich eine - bei der nachrichtenorientierten Kommunikation oft verwendete - (Warte-) Schlange (engl. Queue). Lediglich auf das FIFO-Prinzip muss verzichtet werden, da ja nicht ein Element entnommen werden soll, sondern alle auf einmal.
Das Prinzip des Mitverschickens der Historie durch einen Publisher ist in Abbildung 3.16 veranschaulicht: Bevor irgendwelche Daten veröffentlicht werden können, muss die Schlange in einen definierten Ausgangszustand gebracht (RESET) und der Publisher initialisiert (INIT) werden. Sodann kann mit dem Methodenaufruf DEQ () (engl. dequeue, zu Deutsch „aus der Schlange entnehmen“) eine Ausgabe der gesamten Historie (der Schlange) bewirkt werden, welche anschließend mit dem aktuellen Datum (DATA) veröffentlicht werden kann. Erst dann darf die Schlange durch den Befehl ENQ (DATA) (engl. enqueue, zu deutsch „einreihen“) mit dem aktuellen Element befüllt werden.

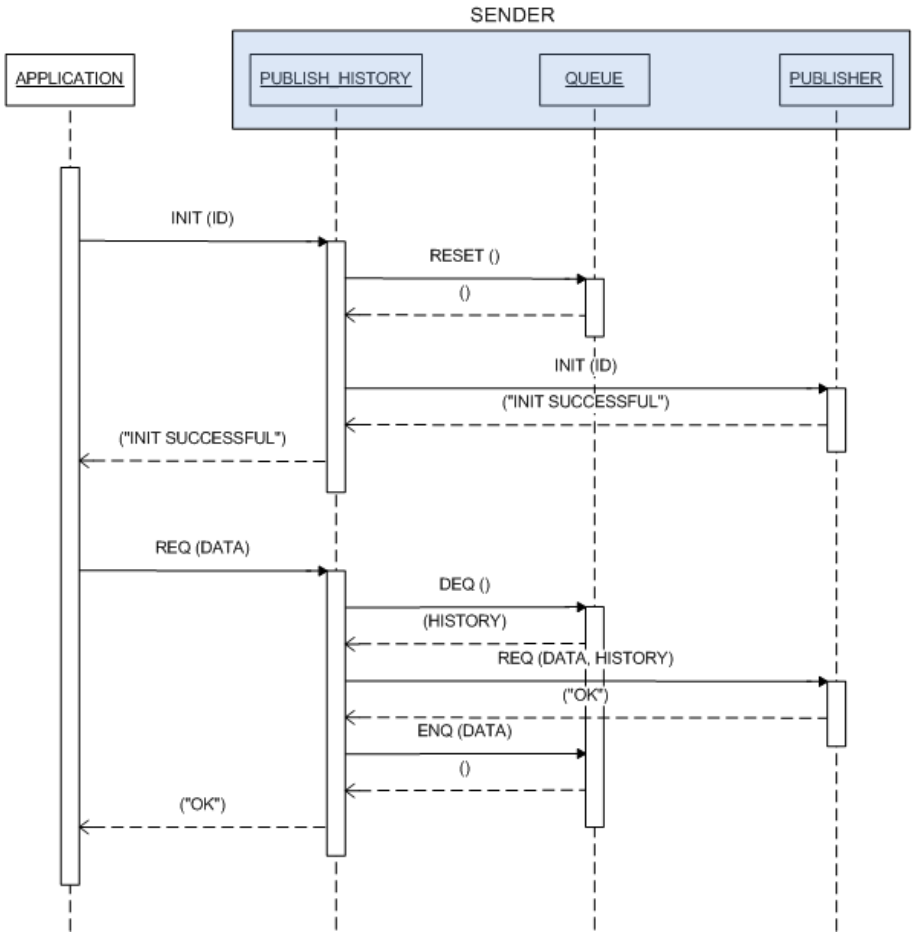

Abbildung 3.16: Mitverschicken der Historie

Auf Subscriber-Seite wird also das aktuelle Datum inklusive der Historie empfangen. Falls die Historie beispielsweise in Array-Form vorliegt, kann es hilfreiche sein, einen Mechanismus vorzusehen, der das Auslesen eines Elements (aus dem Array) ermöglicht. D.h. sobald die Historie empfangen wurde, kann der Subscriber durch Angabe eines Array-Indexes auf ein bestimmtes Element des Arrays zugreifen.

3.5.7 Einführung einer Segmentierung

Oft kommt es vor, dass ein publiziertes Datum (für ein gewisses Topic) nicht alle Subscriber interessiert. Um einen Mechanismus einzuführen, der zwischen Publishern mit gleichem Topic unterscheidet, bietet sich das Prinzip der Segmentierung an: Dazu muss ein Publisher eine Kennung in Form einer Partition definieren (z.B. durch einen String). Ein Subscriber, der die gleiche Partition hat, empfängt sodann nur von diesem Publisher (vorausgesetzt er hat das Topic des Publishers abonniert). Falls ein Subscriber keine Partition angibt, empfängt er von allen Publishern (des gleichen Topics), auch wenn ein Publisher eine Partition angegeben hat.
Zur Illustration: Ein Publisher versendet Temperatur-Werte unter dem Topic „TEMP“. Zusätzlich definiert er eine Partition „FLOOR1“. Ein zweiter Publisher versendet ebenfalls Werte zum Topic „TEMP“, allerdings unter der Partition „FLOOR2“. Innerhalb der verteilten Anwendung existieren drei Subscriber die das Topic „TEMP“ abonniert haben, einer von ihnen unter Angabe der Partition „FLOOR1“, einer mit der Partition „FLOOR2“, der dritte hat keine Partition angegeben. Folglich empfängt der erste nur von jenem Publisher mit der Partition „FLOOR1“, der zweite empfängt nur „FLOOR2“-Werte, und der dritte erhält sämtliche Temperaturwerte („FLOOR1“ und „FLOOR2“).

3.5.8 Verlässliches Senden bzw. Empfangen

Das verlässliche Senden und Empfangen von Daten und Ereignissen kann für eine Applikation essentiell sein. So ist etwa das Versäumen eines Not-Aus-Signals in den meisten Fällen verhängnisvoll.
Falls kein zuverlässiges, verbindungsorientiertes Protokoll (z.B. TCP/IP) verwendet wird, muss ein Mechanismus vorgesehen sein, der Übertragungsfehler erkennt und eventuell korrigiert. Prinzipiell hilft das Versenden von Empfangs-Bestätigungen, die den korrekten Erhalt einer Nachricht anzeigen sollen, Übertragungsfehler zu erkennen.
Während dieser Vorgang mit Clients und Servern relativ leicht bewerkstelligt werden kann, so hat ein Subscriber keine Möglichkeit, dem Publisher zu antworten. Eine Bestätigung würde ja das „Fire and Forget“-Prinzip untergraben.
Um dennoch innerhalb des Publish-Subscribe-Modells eine Aussicht auf eine Bestätigungsnachricht zu haben, kann man den Publisher mit einem Hilfs-Subscriber und den Subscriber mit einem Hilfs-Publisher ausstatten. D.h. sobald der eigentliche Subscriber eine Nachricht erhalten hat, kann er

diese mit seinem Hilfs-Publisher bestätigen. Diese Bestätigung wird schließlich vom Hilfs-Subscriber des Publishers empfangen.
Wenn der Publisher keine Bestätigung erhält, muss er seine Daten erneut versenden. Dabei muss der Applikationsentwickler festlegen, nach welcher Zeit eine Nachricht als verlorengegangen gilt, und wie oft der Publisher einen neuerlichen Versuch unternehmen soll.
In der Zwischenzeit des wiederholenden Sendens müssen neu hinzukommende Nachrichten in einer Schlange gepuffert werden. Erst wenn eine erfolgreiche Zustellung möglich war, wird mit dem nächsten Element in der Schlange fortgefahren.
Falls keine Zustellung der Daten möglich ist, muss die Applikation benachrichtigt werden.
Die beiden Abbildungen 3.17 und 3.18 zeigen zwei unterschiedliche Szenarien. Im ersten Fall muss eine Nachricht zweimal gesendet werden, bis sie vom Subscriber fehlerfrei empfangen werden kann. Das zweite Beispiel veranschaulicht den Fall, dass die Nachricht nicht verlässlich übertragen werden kann. Hier erfolgt eine Benachrichtigung („ERROR“) an die Applikation.

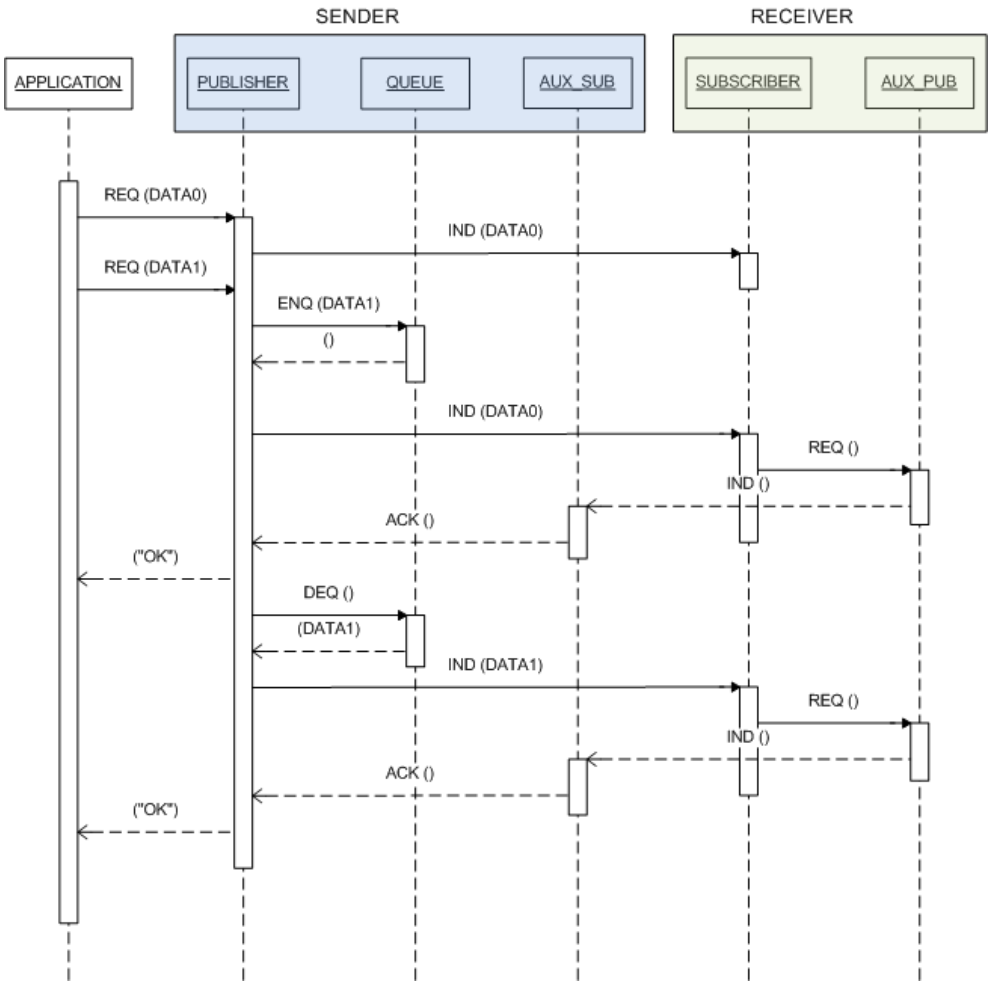

Abbildung 3.17: Verlässliche Kommunikation

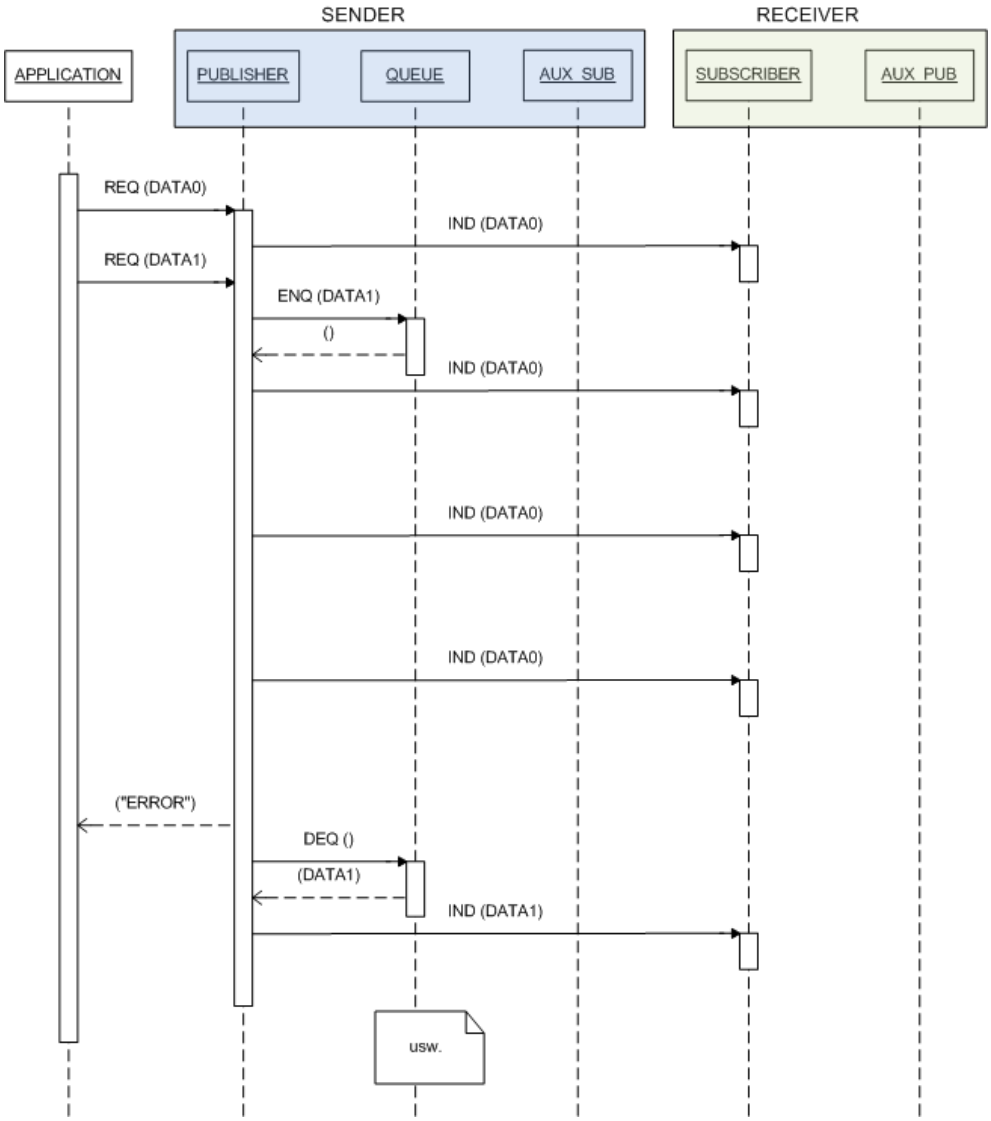

Abbildung 3.18: Unbestätigte Nachricht

3.5.9 Festlegen der Sende- bzw. Empfangsrate

Um eine definierte Senderate realisieren zu können, muss der Publisher nach Versenden der Daten für eine bestimmte Zeit blockieren. Diese Verzögerung kann leicht mit einem Timer umgesetzt werden. In der Zwischenzeit des Blockierens werden keine Daten von der Applikation entgegengenommen, d.h. eventuell anfallende Daten werden verworfen[10].
Während bei der Berücksichtigung von Deadlines (vgl. Kapitel 3.5.4) die Einhaltung von Zeitlimits nur überprüft (und durch entsprechende Benachrichtigungen angezeigt) wird, können mit dem hier beschriebenen Mechanismus Zeitvorgaben tatsächlich umgesetzt werden. Natürlich bietet sich eine Kombination beider Mechanismen insofern an, dass bei zu wenig empfangenen Samples auf Subscriber-Seite (entspricht einer Nicht-Einhaltung der Subscriber-Deadline) die Senderate auf Publisher-Seite angehoben wird.
Als zusätzliche Option bietet sich ein Mechanismus an, der die von der Applikation an den Publisher übergebenen Daten auf Neuheit überprüft und im Falle einer Änderung unter Verletzung der Senderate dennoch versendet. D.h. falls beispielsweise eine Senderate von 3 Sekunden spezifiziert ist und innerhalb einer Sekunde zweimal der gleiche Wert an den Publisher übergeben wird, so wird der erste Wert veröffentlicht und der zweite verworfen. Würde sich jedoch der zweite Wert vom ersten unterscheiden, so würde sowohl der erste als auch der zweite Wert innerhalb der einen Sekunde versendet werden, obwohl eine Senderate von 3 Sekunden angegeben ist.
Auf Subscriber-Seite kann eine bestimmte Empfangsrate analog zur Publisher-Seite realisiert werden, d.h. sobald ein Wert empfangen wurde, muss der Subscriber für die Dauer der Empfangsrate blockieren.
Zu beachten ist, dass eine gleichzeitige Zeit-Festlegung auf Publisher- und Subscriber-Seite eventuell zu Inkompatibilitäten führen kann. So bringt es beispielsweise relativ wenig, wenn ein Publisher alle 2 Sekunden sendet, der korrespondierende Subscriber allerdings nur alle 5 Sekunden empfängt.

3.5.10 Berücksichtigung von Gültigkeiten

Ähnlich der Angabe von maximalen Antwortzeiten für einen Client (siehe oben) kann man für die von einem Publisher veröffentlichten Daten eine Gültigkeitsdauer festlegen. Dazu muss unmittelbar vor dem Senden die

[10] Eine Alternative wäre es, die Daten immer von der Applikation anzunehmen und zu puffern. Der zuletzt gespeicherte (aktuelle) Wert könnte mit der vorgegebenen Rate versendet und der Puffer mit jedem Absenden gelöscht werden

aktuelle Systemzeit ermittelt, diese mit der spezifizierten Zeit addiert, und die Summe schließlich gemeinsam mit dem eigentlichen Datum veröffentlicht werden.
Ein korrespondierender Subscriber muss nach dem Erhalt dieser Daten ebenfalls die Systemzeit feststellen und diese mit der empfangenen Zeit vergleichen. Die empfangenen Daten sind nur dann gültig, wenn gilt:
Zeit des Empfangens $<$ Zeit des Sendens $+$ Gültigkeitsdauer.
Abbildung 3.19 soll die Verhältnisse veranschaulichen.

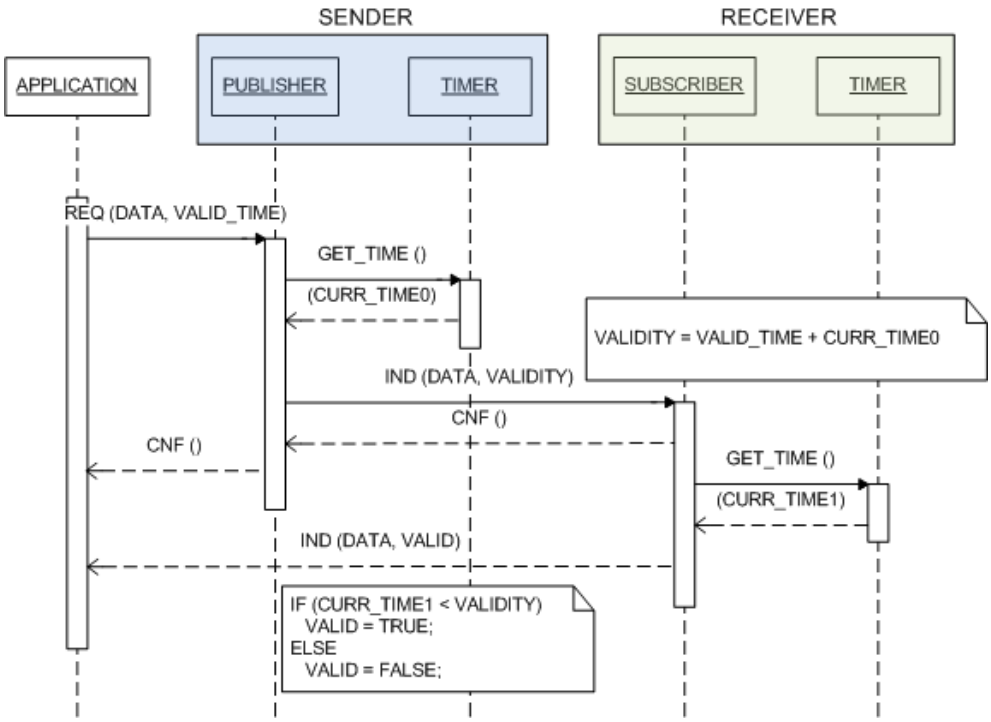

Abbildung 3.19: Berücksichtigung einer Gültigkeitsdauer

Eine wichtige Voraussetzung für die Realisierung dieses Mechanismus ist das Vorhandensein synchronisierter Uhren. Diesbezüglich sei auf den Standard IEEE 61588 verwiesen, der das PTP (Precision Time Protocol) definiert und dadurch einen Uhrenabgleich im Nanosekunden-Bereich in Multicast-fähigen Netzwerken ermöglicht [Wei04], [Ins04].

3.5.11 Gefilterter Empfang

Vor allem im Zusammenhang mit Sensorwerten werden oft Bereiche definiert, die einen regulären Betrieb kennzeichnen bzw. welche, die als kritisch bezeichnet werden können. Es kann manchmal sinnvoll sein, gleich beim Empfang von Daten zu überprüfen, ob diese Daten innerhalb eines spezifizierten Fensters liegen oder nicht. Beispielsweise können Temperaturwerte zwischen -78°C und -90°C kennzeichnend für einen normalen Betrieb sein. Sobald dieser Bereich über- oder unterschritten wird, muss eine Meldung an die Applikation erfolgen. Abbildung 3.20 zeigt den Ablauf.

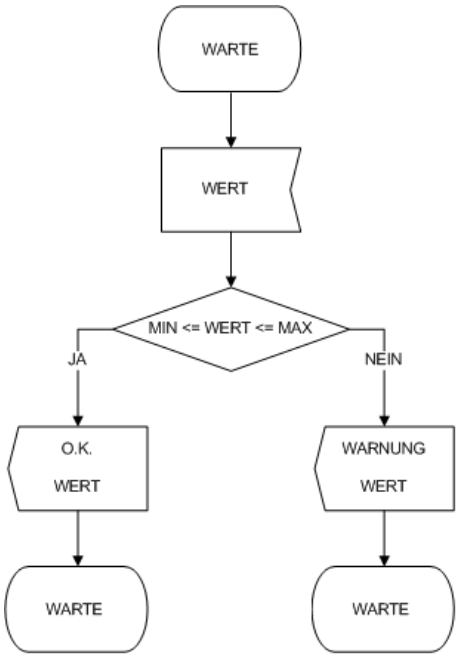

Abbildung 3.20: Gefilterter Empfang

3.5.12 Konsistentes Empfangen

Eventuell kann es für eine Applikation wichtig sein, dass Daten von unterschiedlichen Stellen an einer Stelle empfangen werden und zwar innerhalb gewisser Zeiten, da ansonsten keine Konsistenz gewährleistet ist. Konkret wird ein Subscriber, der sowohl Temperatur- als auch Druckwerte empfängt und diese für weitere Berechnungen bereitstellt, prüfen müssen, ob das 2-Tupel brauchbar ist, d.h. ob beide Werte innerhalb eines spezifizierten

Zeitfensters empfangen wurden. Ist dies nicht der Fall, so muss er jenes 2-Tupel entsprechend kennzeichnen (z.B. mithilfe eines Flags).
Auch hier ist eine Realisierung mittels Zeitgebern möglich. Sobald ein Wert von Stelle A empfangen wurde, wird die aktuelle Systemzeit erfasst. Wenn ein Wert von Stelle B eintrifft, wird abermals die Systemzeit abgefragt, und die beiden Zeiten subtrahiert. Ist die Differenz kleiner als das spezifizierte Zeitfenster, so ist das 2-Tupel gültig und kann weiterverarbeitet werden. Falls mehrere Werte von Stelle A eintreffen, bevor ein Wert von Stelle B ankommt, so muss immer der aktuellste Wert von Stelle A und dessen Ankunftszeit gespeichert werden. Das Ganze muss natürlich auch funktionieren, wenn zuerst von Stelle B empfangen wird.
Als Alternative kann anstatt der Empfangszeit auch die Sendezeit zur Ermittlung der Konsistenz herangezogen werden. Dazu muss jeder Sender (A und B) neben dem eigentlichen Datum auch den Zeitpunkt des Versendens, also einen Zeitstempel (engl. timestamp), verschicken. Für Konsistenz muss die Differenz von Zeitstempel A und Zeitstempel B innerhalb des auf Empfängerseite spezifizierten Fensters (von z.B. 10 ms) liegen.
Zusätzlich kann auf Senderseite für jedes Datum eine Gültigkeitsdauer festgelegt und verschickt werden, die auf Empfängerseite mit dem Zeitstempel addiert und zur Ermittlung der Gültigkeit schließlich mit der Zeit des Empfangens verglichen wird (vgl. Kapitel 3.5.10).

3.5.13 Entity Factory

Die Entity Factory soll eine gruppenweise Anmeldung von Kommunikationsteilnehmer ermöglichen. Dazu ist es erforderlich, ein entsprechendes Flag (das anzeigt, ob die Anmeldung gleich oder zu einem späteren Zeitpunkt erfolgen soll) an die jeweiligen Teilnehmer zu senden. D.h. die Entity Factory muss lediglich einen booleschen Wert generieren, die Versendung dessen kann separat durch einen Publisher geschehen.

4 Implementierung

In diesem Kapitel soll die prototypische Implementierung der im Abschnitt 3.5 beschriebenen Mechanismen dargestellt werden, die später beim Engineering evaluiert werden soll.
Für eine Umsetzung wurde der weit verbreitete, IEC 61499-konforme FBDK (Function Block Development Kit) der Firma HOLOBLOC Inc. ausgewählt [Hol]. HOLOBLOC Inc. ist ein gewinnorientiertes Unternehmen in Ohio, U.S.A., das von Dr. James H. Christensen gegründet wurde und geleitet wird. Die Software kann inklusive Java-basierter Runtime (FBRT, Function Block Runtime) unter `www.holobloc.com` kostenlos heruntergeladen werden und ermöglicht die Entwicklung von Datentypen, Funktionsblock-Typen, Resource-Typen, Device-Typen und System-Konfigurationen. Die aktuelle Version des FBDKs ist Version 20061017, die auch für die folgenden Implementierungen verwendet wurde.

Die Kommunikation wird durch das „Compliance Profile for Feasibility Demonstrations“ festgelegt, aus dem gewisse Einschränkungen resultieren: Als unterlagertes Kommunikationsmedium wird Standard-Ethernet vorausgesetzt, welches bekanntlich nicht echtzeitfähig ist. Ebenso ist keine Einflussnahme auf die Busparameter möglich, wobei diese für den Anwendungsfall nicht relevant sind. Dafür schafft die Funktionalitäts-Kapselung Transparenz, denn die Änderung der internen Implementierung der FBs bei gleich bleibendem Interface macht deren Fähigkeiten auch für andere Medien nutzbar.
Letztendlich wurden keine QoS-Mechanismen implementiert, die dezidiert einen Echtzeitbus fordern.

Die Benutzeroberfläche des FBDK ist in Abbildung 4.1 zu sehen. Neben einer graphischen Darstellung von FBs (Worksheet) (mit Drag & Drop - Funktionalität bei FB-Netzwerken) ist auch deren Beschreibung in XML möglich. Im Navigation-Tree können unterschiedliche Ansichten ausgewählt werden. Neben der FB-Ansicht (Interface) können bei einem Basic-FB auch der ECC sowie die einzelnen Algorithmen des FBs (im Bild INIT und REQ) angezeigt werden. Bei einem Composite-FB kann zwischen seinem externem Interface und seinem Innenleben (internes FB-Netzwerk) gewechselt werden.

Für das Testen der erstellten FB-Typen (und Systemkonfigurationen) ist der Launch-Button besonders nützlich (in Abbildung 4.1 zwischen Run- und Help-Menü zu sehen). Durch Klick auf diesen Button wird eine Instanz des Funktionsblocks (bzw. ein Systemabbild) in einem Testfenster mit entsprechenden Eingabefeldern und Anzeigen erzeugt.
Eine genauere Beschreibung des FB-Editors ist in der Software inkludiert.

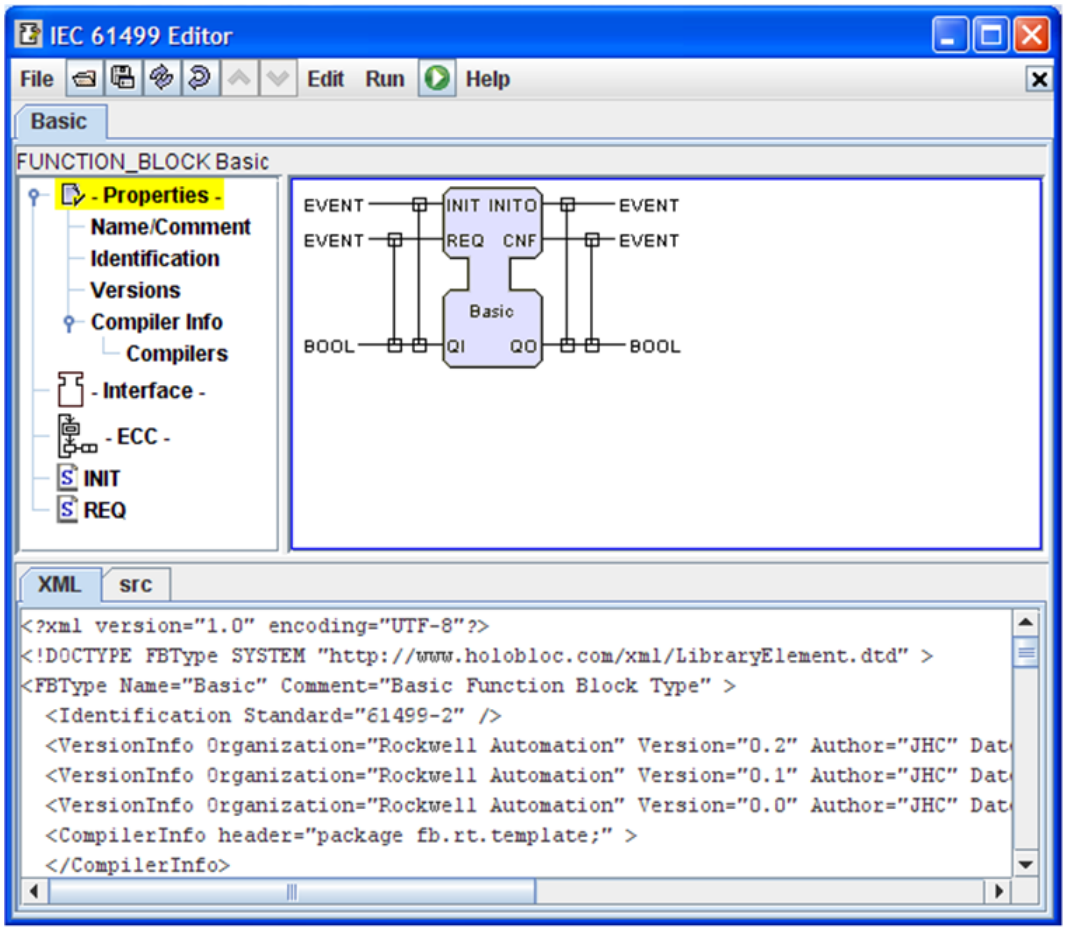

Abbildung 4.1: Benutzeroberfläche des FBDKs der Firma Holobloc, Inc.

Eine Realisierung der Middleware-Funktionalität erfolgt unter Verwendung von Composite-FBs, wobei u.a. auf vorgefertigte (d.h. im Programmumfang enthaltene) Client-, Server-, Publish- und Subscribe-SIFBs sowie einige Basic-FBs zurückgegriffen wird.
Die meisten der entworfenen Kommunikations-FBs orientieren sich an

dem aus der Middleware-Analyse hervorgegangenen Favoriten DDS und wurden deshalb als Publish- bzw. Subscribe-FBs realisiert. Zwei Services wurden mittels Client/Server implementiert, um bekannten Schwierigkeiten beim Einsatz von im FBDK-Umfang enthaltenen Client- und Server-FBs vorzubeugen, bzw. um die Thematik abzurunden.
Es werden im Folgenden nur die implementierten Kommunikationsblöcke (bzw. FBs mit QoS-Unterstützung) beschrieben. Die darin enthaltenen, zusätzlich entworfenen FBs werden im Anhang erläutert.

Bezüglich der durch die Kommunikationsblöcke verschickten Daten wäre die Verwendung des generischen Datentyps ANY sinnvoll, welcher die Versendung von Daten beliebigen Typs erlaubt. Allerdings ist IEC 61499 eine streng typisierte Sprache, d.h. mathematische Operationen wie Vergleich, Addition und Subtraktion sind nur für konkrete Datentypen verfügbar (es dürfen auch keine unterschiedlichen Datentypen gemischt werden, beispielsweise ist die Addition eines UINTs mit einem REAL-Wert ohne explizite Typumwandlung nicht erlaubt). Verbindungen von ANY zu ANY sind grundsätzlich unzulässig (der Datentyp muss mindestens auf einer Seite der Verbindung eindeutig festgelegt sein). Beim Erstellen generischer FBs muss man also immer aufpassen, dass man dabei die Typsicherheit nicht aushebelt. Die entworfenen Kommunikations-FBs unterstützen daher - wenn die Verwendung von ANY problematisch ist - die gängigen Datentypen REAL bzw. WSTRING, da die Verwendung dieser Datentypen eindeutig ist, und durch sie fast alle relevanten Daten übertragen werden können. Implementierungen für andere Datentypen sind analog möglich.

4.1 Berücksichtigung der Reihenfolge bei der Initialisierung (CLIENT_INIT)

Um einem Server mehr Zeit für seine Initialisierung zu geben, kann der FB aus Abbildung 4.2 verwendet werden, der bei einer gescheiterten Client-Initialisierung neuerliche Anmeldungsversuche (des Clients) vorsieht (vgl. Kapitel 3.5.2). Neben den üblichen Dateneingängen eines Clients (QI, ID und SD_1) sind zwei weitere Eingänge definiert: INIT_TRIES und RETRY_INTERVAL. Durch INIT_TRIES wird die maximale Anzahl an Initialisierungsversuchen festgelegt. Die Dauer, die zwischen jedem neuen Initialisierungsversuch gewartet wird, ist durch den Parameter

RETRY_INTERVAL einstellbar.
Der Initialisierungsausgang INIT_STATUS kann einen der drei folgenden Werte einnehmen:

- INIT NOT SUCCESSFUL: wird nach dem letzten erfolglosen Initialisierungs-Versuch angezeigt.
- INIT SUCCESSFUL: Erfolgsmeldung für den Fall, dass sich während der durch INIT_TRIES angegebenen Versuche der Server anmeldet.
- WAITING FOR INIT: wird während der Initialisierungsversuche oder nach Abmeldung des Clients angezeigt. (Nach Client-Abmeldung hat STATUS den Wert TERMINATED.)

Als Pendant zu CLIENT_INIT eignet sich ein gewöhnlicher SERVER_1-SIFB.

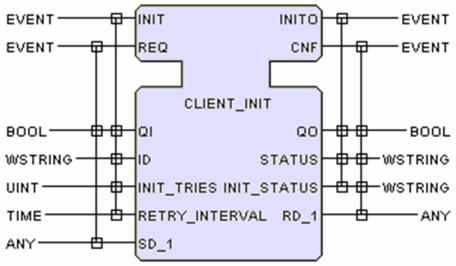

Abbildung 4.2: CLIENT_INIT FB

Das interne FB-Netzwerk von CLIENT_INIT ist in Abbildung 4.3 dargestellt. Solange der Initialisierungsprozess noch nicht abgeschlossen ist, was durch die FBs FB_CAST, FB_EQUAL und FB_E_SWITCH_3 festgestellt wird, wird am Ausgang WAITING FOR INIT ausgegeben. Außerdem wird ein Zähler (FB_E_CTD) - von seinem Startwert INIT_TRIES beginnend - mit jedem Initialisierungsversuch um eins verringert, wobei eine Verzögerung zwischen jedem neuerlichen Versuch durch FB_E_DELAY realisiert wird. Je nachdem ob innerhalb der definierten Versuche eine Initialisierung möglich ist oder nicht, erfolgt eine entsprechende Ausgabe. Bei Erfolg wird der Zähler zusätzlich in seinen Ausgangszustand versetzt.
Die drei FBs nach FB_SELECT (FB_CAST_2, FB_EQUAL_2, FB_E_PERMIT) werden für den Fall benötigt, dass sich der Server nach erfolgreicher Initialisierung wieder deinitialisiert. Denn dann probiert der

Client sich wieder anzumelden und zwar genau so oft und lange, wie durch INIT_TRIES bzw. RETRY_INTERVAL vorgegeben. Durch besagte drei FBs wird der INIT_TRIES-Wert nach jeder erfolgreichen Client-Initialisierung wieder neu in den Zähler geladen.

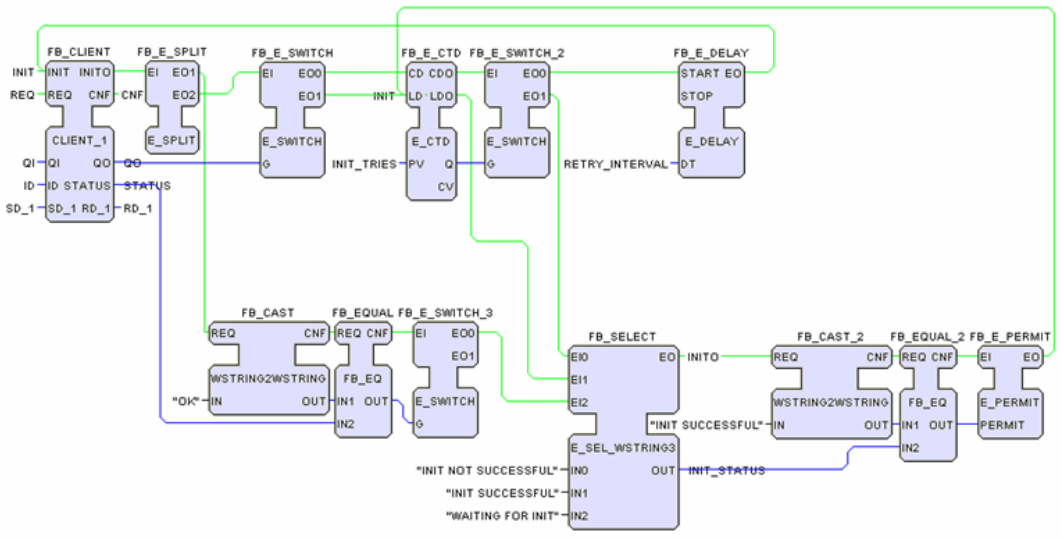

Abbildung 4.3: CLIENT_INIT Netzwerk

4.2 Antwortzeiten auf Anforderungen (CLIENT_RESPONSE_TIME)

Zur Angabe von maximal tolerierbaren Antwortzeiten eignet sich der Client aus Abbildung 4.4, der zusammen mit einem SERVER_1-FB verwendet werden kann (vgl. Kapitel 3.5.3).[1]
Bei jedem Request wird ein Datum (RD_1) angefordert und zusätzlich angegeben, wann der Client spätestens eine Antwort erwartet (MAX_RSP_TIME). Wenn gilt: verstrichene Zeit bis zur Antwort < MAX_RSP_TIME, so wird das VALID-Flag am Ausgang auf TRUE gesetzt, d.h. die empfangenen Daten

[1] Eine Implementierung mittels Publish/Subscribe ist insofern nicht sinnvoll, da das Publish/Subscribe-Prinzip keine Antwortmöglichkeit des Subscribers vorsieht. Das Publish/Subscribe-Äquivalent zu diesem Mechanismus kann am ehesten in der in Kapitel 4.10 beschriebenen Berücksichtigung von Gültigkeitsdauern gesehen werden.

sind gültig.
Die empfangenen Daten werden immer ausgegeben, sodass das SIFB-Paar (CLIENT_RESPONSE_TIME und SERVER_1) auch wie ein gewöhnliches Client/Server-Paar verwendet werden kann.

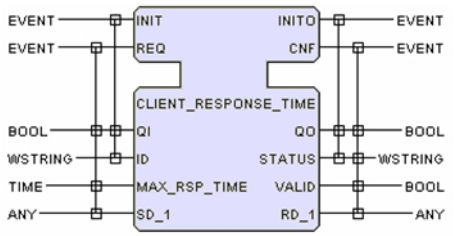

Abbildung 4.4: CLIENT_RESPONSE_TIME FB

Die interne Realisierung von CLIENT_RESPONSE_TIME ist in Abbildung 4.5 zu sehen. Mit Verschicken des Requests wird ein Timer (E_CHRON2, siehe Anhang) gestartet. Bei Erhalt der Antwort wird der Timer wieder gestoppt, und die bis zu diesem Zeitpunkt verstrichene Zeit mit der angegebenen QoS (MAX_RSP_TIME) verglichen (VALIDATE_TIME ist im Anhang beschrieben). VALID wird auf TRUE gesetzt, wenn gilt: ET < MAX_RSP_TIME.

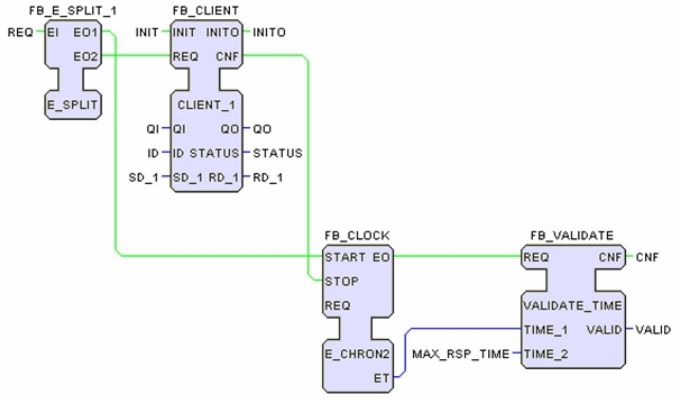

Abbildung 4.5: CLIENT_RESPONSE_TIME Netzwerk

4.3 Entity Factory

Wie bereits in Kapitel 3.5.13 erwähnt, dient die Entity Factory der Anmeldung (bzw. Abmeldung) von SIFBs (siehe Abbildung 4.6). Je nach Eingangs-Event (ENABLE oder DISABLE) wird das Ausgangs-Flag auf True (STATUS := TRUE;) oder False (STATUS := FALSE;) gesetzt. Dieses Flag kann letztendlich mittels eines Publishers und mehrerer Subscriber (1-zu-n-Kommunikation) an sämtliche QI-Eingänge der beteiligten Kommunikations-FBs gesendet werden.
Es handelt sich bei ENTITY_FACTORY um einen Basic-FB, der zwei Algorithmen (für jedes Event einen) beinhaltet, die das entsprechende Setzen des Ausgangs-Flags erledigen (siehe oben).

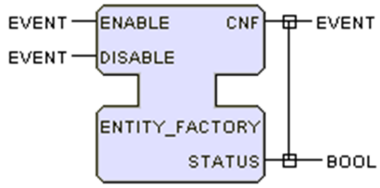

Abbildung 4.6: ENTITY_FACTORY FB

4.4 Berücksichtigung von Deadlines

Der Mechanismus zur Berücksichtigung von Deadlines wurde bereits in Kapitel 3.5.4 erörtert. Er soll feststellen helfen, ob der Publisher genügend Daten versendet, bzw. ob der Subscriber ausreichend Daten empfängt.

PUBLISH_DEADLINE
Durch das Ausgangsflag IN_TIME wird angezeigt, ob der Publisher innerhalb der vorgegebenen Zeit DEADLINE von der Applikation Daten zum Versenden bekommt (siehe Abbildung 4.7). War der letzte Request (REQ) länger aus als durch DEADLINE angegeben, so wird IN_TIME auf FALSE gesetzt.

FB_E_TIME (siehe Anhang) misst die Dauer zwischen zwei aufeinander folgenden Events (siehe Abbildung 4.8). Die resultierende Zeitdifferenz DELTA_T wird durch FB_VALIDATE_TIME (siehe Anhang) mit DEADLINE

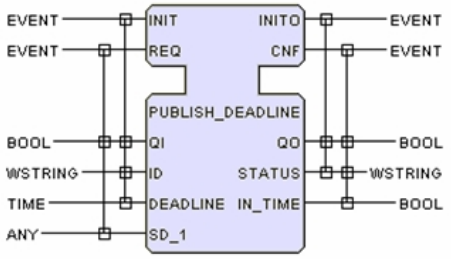

Abbildung 4.7: PUBLISH_DEADLINE FB

verglichen, und das Ergebnis in Form des VALID-Flags ausgegeben.

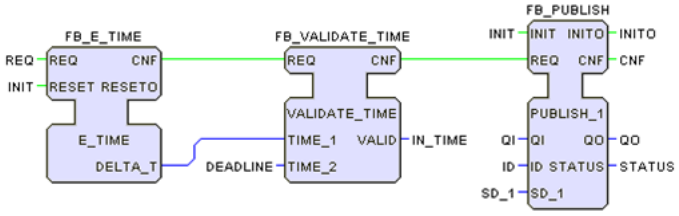

Abbildung 4.8: PUBLISH_DEADLINE Netzwerk

SUBSCRIBE_DEADLINE
Wenn nicht oft genug Daten empfangen werden (durch DEADLINE angegeben) bzw. der korrespondierende Publisher nicht oft genug sendet, so wird ENOUGH auf False gesetzt (siehe Abbildung 4.9).
Analog zu PUBLISH_DEADLINE (siehe oben) misst FB_E_TIME (siehe Anhang) jeweils die Dauer zwischen zwei aufeinander folgenden IND-Events und übergibt diese Zeitdifferenz für eine Auswertung an FB_VALIDATE (siehe Abbildung 4.10). Beim ersten IND-Event ist DELTA_T = t#0s, weshalb das erste Datum immer ENOUGH = TRUE zur Folge hat.

4.5 Einführung einer Sendeschlange (PUBLISH_FIFO)

Um eine Pufferung der zu sendenden Daten zu erreichen, kann ein Publisher, wie in Kapitel 3.5.5 beschrieben, verwendet werden. Bei diesem

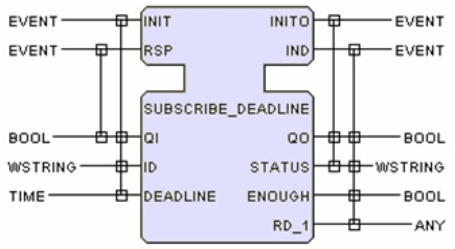

Abbildung 4.9: SUBSCRIBE_DEADLINE FB

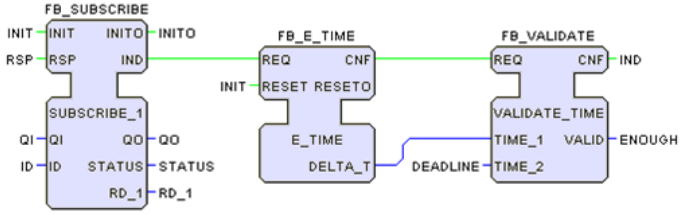

Abbildung 4.10: SUBSCRIBE_DEADLINE Netzwerk

Publisher werden die Daten vor ihrer Veröffentlichung in eine Schlange (mit FIFO-Prinzip) eingereiht (siehe Abbildung 4.11 und Abbildung 4.12).
Die Senderate (RATE) muss schon bei der Initialisierung angegeben werden, denn schon mit dem INIT-Event beginnt der Publisher damit, Daten mit vorgegebener Rate aus der Schlange zu nehmen und zu verschicken. Natürlich sind anfangs noch keine Elemente in der Schlange (N = 0), d.h. es wird zwar nichts versendet, aber das zyklische PULL wird mit der Initialisierung gestartet und erst bei der Abmeldung des Publishers, wenn gilt STATUS = TERMINATED und QO = FALSE, gestoppt. Bei einem Request (REQ) werden die Daten lediglich in die Schlange eingereiht.
FB_E_SWITCH erkennt, wenn sich der Publisher deinitialisiert und stoppt in diesem Fall FB_E_CYCLE (und somit das zyklische PULL).
Der Composite FB PUBLISH_FIFO kann 128 Elemente in seine Schlange (FB_FIFO, siehe Anhang) aufnehmen. Sobald die Schlange voll ist, können keine weiteren Elemente mehr eingereiht werden, d.h. die neuen Werte werden verworfen.
Auf Subscriber-Seite kann ein gewöhnlicher SUBSCRIBE_1-FB verwendet werden.

Nach Abmeldung des Publishers können zwar noch Daten in die Schlange gestellt werden, bei einer Neuinitialisierung gehen diese Daten allerdings verloren. Abhilfe (bezüglich der Einreihung der Daten) könnte (ähnlich wie beim Stoppen der zyklischen Pull-Generierung) die Kombination des REQ-Events mit dem QO-Flag und einem E_SWITCH-FB schaffen.

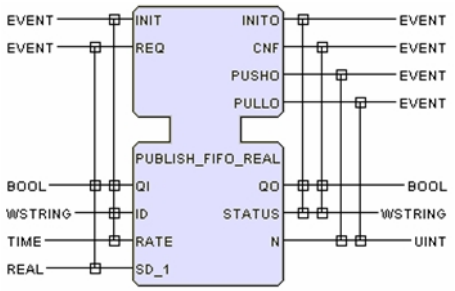

Abbildung 4.11: PUBLISH_FIFO FB

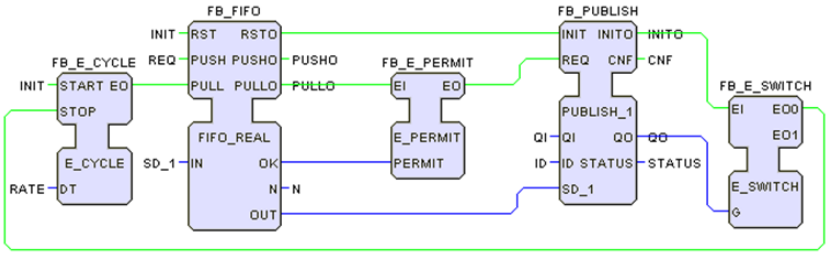

Abbildung 4.12: PUBLISH_FIFO Netzwerk

4.6 Mitverschicken der Historie

Für zum verteilten System neu hinzukommende Applikationen und Devices kann die Speicherung vergangener Nachrichten und deren Mitversendung mit dem aktuellen Datum sinnvoll sein (vgl. Kapitel 3.5.6). Diese Aufgaben werden vom Publisher erfüllt, während der Subscriber neben dem einzelnen, aktuellen Datum auch die Sammlung an älteren Werten empfangen können muss.

PUBLISH_HISTORY

PUBLISH_HISTORY sieht von seinem externen Interface her wie ein PUBLISH_1-FB aus, weshalb im Folgenden nur das interne Netzwerk dargestellt wird (siehe Abbildung 4.13).
Neben dem aktuellen Datum (SD_1) werden auch die letzten 20 (REAL- bzw. WSTRING-) Samples in Array-Form (SD_2) versendet. Dazu wird der FB QUEUE_REAL (bzw. _WSTRING, siehe Anhang) benötigt, wobei zu beachten ist, dass das REQ-Event zuerst mit DEQ und dann mit ENQ verbunden werden muss. Ansonsten würde das aktuelle Datum auch in der Historie (QU) aufscheinen.
Auf Subscriber-Seite muss SUBSCRIBE_HISTORY (_REAL bzw. _WSTRING) verwendet werden (siehe unten).

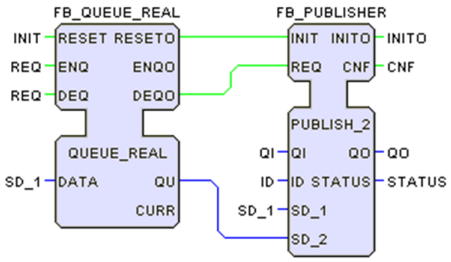

Abbildung 4.13: PUBLISH_HISTORY Netzwerk

SUBSCRIBE_HISTORY

SUBSCRIBE_HISTORY (siehe Abbildung 4.14) ist der „Partner“ von PUBLISH_HISTORY (_REAL bzw. _WSTRING, siehe oben).
Neben dem eigentlichen (aktuellen) Datum empfängt dieser Subscriber auch die letzten 20 Samples des Publishers als (REAL- bzw. WSTRING-) Array (HISTORY).
Das interne Netzwerk besteht aus einem SUBSCRIBE_2-FB, der unter RD_1 das aktuelle Datum und unter RD_2 die Historie empfängt. Damit die Historie am Ausgang (HISTORY) ausgegeben werden kann, muss sie zuvor mit dem FB WSTRING2WSTRING_ARRAY bzw. REAL2REAL_ARRAY (siehe Anhang) entsprechend aufbereitet werden.

SUBSCRIBE_HISTORY_READ_OUT

Dieser FB stellt eine Erweiterung von SUBSCRIBE_HISTORY (_REAL bzw. _WSTRING) dar (siehe oben).

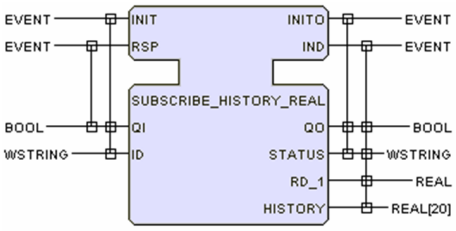

Abbildung 4.14: SUBSCRIBE_HISTORY FB

Hier gibt es die zusätzliche Möglichkeit, Elemente aus dem HISTORY-Array über das Event READ_OUT und unter Angabe eines Indexes (INDEX, zwischen 1 und 20) auszulesen (siehe Abbildung 4.15). Während das aktuelle Datum (RD_1) und die Historie (HISTORY) mit jedem IND-Event ausgegeben werden, wird die Ausgabe eines bestimmten Array-Elements (RD_2) mit dem ROO-Event angezeigt.
Erst nachdem das erste Mal ein Wert (und damit eine Historie) empfangen wurde, können die Daten aus dem Array ausgelesen werden. Dies wird durch FB_E_SR und FB_E_PERMIT garantiert (siehe Abbildung 4.16). Es ist wichtig, das Flip-Flop (FB_E_SR) mit der Initialisierung (INIT-Event) zurückzusetzen (R-Eingang), da ansonsten ein unzulässiger Wert gespeichert bleiben kann.
Im Anhang findet sich eine Erläuterung zu READ_OUT_REAL_ARRAY (bzw. READ_OUT_WSTRING_ARRAY).

Je nach Anwendungsfall kann es sinnvoll sein, nicht SUBSCRIBE_READ_OUT_HISTORY zu verwenden sondern SUBSCRIBE_HISTORY in Kombination mit einem READ_OUT_ARRAY-FB, d.h. das Auslesen nicht im Subscriber auszuführen sondern im Anschluss an diesen.
Eine mögliche Abwandlung wäre außerdem, den Array-Index schon bei der Initialisierung anzugeben, um dadurch auf das READ_OUT-Event verzichten, aber dafür immer nur eine Element-Position ansprechen zu können.

4.7 Einführung einer Segmentierung

Das Prinzip der Segmentierung wurde in Kapitel 3.5.7 beschrieben und muss sowohl im Publisher als auch Subscriber berücksichtigt werden.

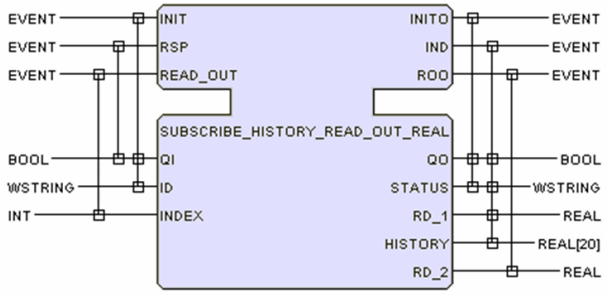

Abbildung 4.15: SUBSCRIBE_HISTORY_READ_OUT FB

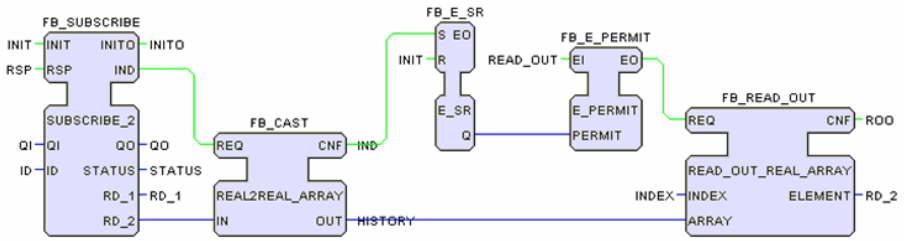

Abbildung 4.16: SUBSCRIBE_HISTORY_READ_OUT Netzwerk

PUBLISH_PARTITION

Durch Eingabe einer PARTITION kann sichergestellt werden, dass nur ein Subscriber mit gleicher PARTITION (oder keiner Partition) die versendeten Nachrichten erhält (siehe Abbildung 4.17).
Eine aufwendige Implementierung ist nicht nötig, zumal intern lediglich ein PUBLISH_2-FB verwendet werden muss. Das Datum (SD_1) wird an den SD_1-Dateneingang des PUBLISH_2-FBs, die PARTITION an den SD_2-Eingang gelegt.

SUBSCRIBE_PARTITION

Dieser FB korrespondiert mit PUBLISH_PARTITION (_REAL bzw. _WSTRING, siehe oben). Sein Interface ist in Abbildung 4.18, sein interner Aufbau in Abbildung 4.19 zu sehen.
Es werden nur dann Daten empfangen, wenn der Publisher die gleiche PARTITION hat. Wenn bei PARTITION nichts eingegeben wird, so empfängt der Subscriber alle Daten des Publishers, egal ob dieser eine PARTITION verwendet oder nicht.

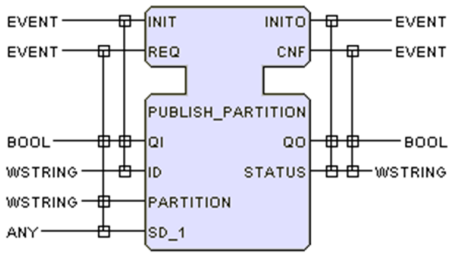

Abbildung 4.17: PUBLISH_PARTITION FB

Korrespondierend mit PUBLISH_PARTITION empfängt FB_SUBSCRIBE unter RD_1 das Datum und unter RD_2 die Partition. Je nach Datentyp sind eine Reihe an Konvertierungen (FB_CAST, FB_CAST_2, FB_CAST_3) notwendig. FB_COMPARE_2 überprüft, ob eine Partition auf Empfänger-Seite spezifiziert wurde. Falls nicht (EO1-Event bei FB_E_SWITCH), so wird ein IND-Event generiert, und das Datum unter RD_1 ausgegeben. Falls ja (EO0-Event bei FB_E_SWITCH), so wird die Subscriber-Partition mit der Publisher-Partition verglichen (FB_COMPARE). Bei einer Übereinstimmung wird das Datum ausgegeben, ansonsten unterbleibt die Ausgabe.

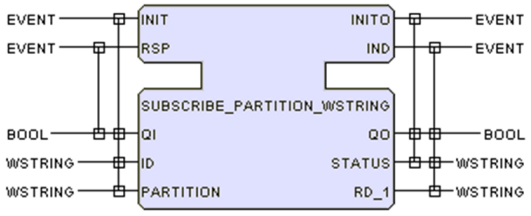

Abbildung 4.18: SUBSCRIBE_PARTITION FB

4.8 Verlässliches Senden bzw. Empfangen

Bei einer verlässlichen Kommunikation (siehe Kapitel 3.5.8) müssen alle gesendeten Daten ihren Empfänger erreichen. Damit der Sender weiß, ob seine Nachricht angekommen ist, muss der Empfänger eine Empfangs-Bestätigung an ihn retournieren. Falls innerhalb einer definierten Zeit keine Bestätigung ankommt, kann der Sender einen erneuten Sendeversuch unternehmen.

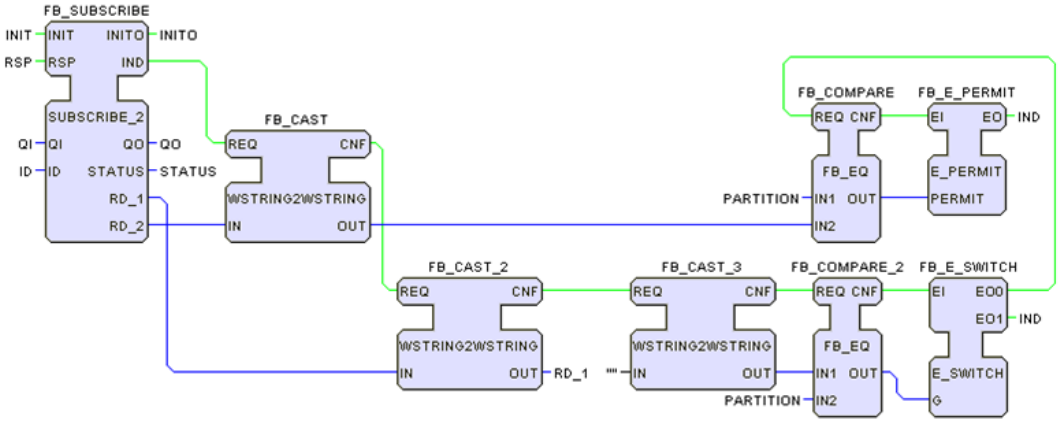

Abbildung 4.19: SUBSCRIBE_PARTITION Netzwerk

Nach mehreren unbestätigten Versuchen muss der Sender die Applikation alarmieren.

PUBLISH_RELIABILITY

Das externe Interface für einen „verlässlichen“ Publisher ist in Abbildung 4.20 zu sehen.
Neben der Zeit, die gewartet wird, bevor eine Nachricht als verloren gilt (RETRY_INTERVAL), ist auch die Anzahl der Sende-Wiederholungen (RETRIES) durch den Benutzer einstellbar. Außerdem ist die Rate, mit der die Elemente aus der Schlange genommen werden (PULL_RATE), frei wählbar.
Es werden keine weiteren Nachrichten verschickt, solange bis eine Bestätigung (ACK) empfangen wurde. Bei erfolgloser Versendung (bzw. Nicht-Bestätigung) wird am Ausgang ein ERR-Event generiert und mit dem nächsten Element aus der Queue fortgefahren.
PUBLISH_RELIABILTY passt mit SUBSCRIBE_RELIABILITY zusammen.

Abbildung 4.21 zeigt das FB-Netzwerk. Mit Initialisierung des Publishers wird gleichzeitig ein Hilfs-Subscriber (FB_SUBSCRIBE) initialisiert, der bei erfolgreichem Empfang durch den eigentlichen Subscriber das Bestätigungs-Event (ACK) empfängt (auf Subscriber-Seite wird die Bestätigung durch einen Hilfs-Publisher verschickt, siehe SUBSCRIBE_RELIABILITY).
Der Hilfs-Subscriber verwendet für seine Kommunikation mit dem Hilfs-

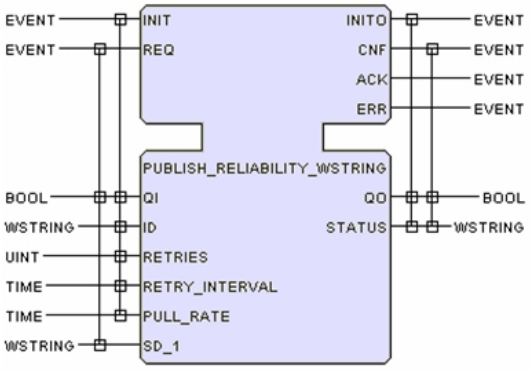

Abbildung 4.20: PUBLISH_RELIABILITY FB

Publisher den nächst höheren Kanal (Port) des Publishers (INCREASE_ID, siehe Anhang).
Beim Hilfs-Subscriber ist auf eine saubere Initialisierung (gleichzeitiges An- und Abmelden mit dem Publisher) zu achten, damit bei einer Deinitialisierung und neuerlichen Initialisierung sowohl Publisher als auch Subscriber wieder einsatzbereit sind.
Wie bereits erwähnt werden neu anfallende Daten (in der Zeit des Blockierens) in einer Schlange (FB_FIFO) zwischengespeichert. Wenn die Schlange voll ist, werden sämtliche weiteren Anfragen (REQ) verworfen. Sobald eine Bestätigung (ACK) hereinkommt, wird der Schlange das nächste Element nach dem FIFO-Prinzip entnommen und verschickt. Die zyklische PULL-Operation wird mit dem ersten REQ über die FBs FB_INIT und FB_E_PERMIT eingeleitet. Eine Rückkopplung über ein Delay-Element (FB_E_DELAY) sorgt dafür, dass der PULL-Prozess bei einer leeren Schlange nicht zum Erliegen kommt.
Damit die PULL_RATE auch bei einer erfolgreichen Versendung und Bestätigung nicht verletzt wird, muss FB_E_DELAY_2 ins Netzwerk aufgenommen werden. Ansonsten könnte es passieren, dass mehrere Elemente hintereinander aus der Schlange entnommen, verschickt und bestätigt werden - und dies ohne Verzögerung.
Sobald ein Element der Schlange entnommen wurde, wird das zyklische PULL angehalten und alles für ein wiederholtes Versenden einer eventuell versäumten Nachricht vorbereitet: ein Zähler (FB_E_CTD) wird mit dem RETRIES-Wert initialisiert und beginnt (falls keine Bestätigung vom Subscriber kommt) mit der durch RETRY_INTERVAL vorgegebenen Periode herunterzuzählen (FB_E_CYCLE). Wenn der Zähler den Wert Null

erreicht hat, wird ein ERR-Event erzeugt und mit dem nächsten Element aus der Schlange fortgefahren. Falls schon vorher eine Bestätigung eingeht, wird der Zählvorgang abgebrochen und dann wieder mit dem zyklischen PULL fortgesetzt.
Es sei darauf hingewiesen, dass kein Mechanismus (weder auf Publisher- noch auf Subscriber-Seite) zur Duplikat-Vermeidung implementiert wurde, d.h. eine Bestätigung, die erst nach Ablauf von RETRY_INTERVAL ankommt, müsste auf beiden Seiten entsprechend behandelt werden. Es wird empfohlen, im Falle eines ERR-Events die Applikation anzuhalten. Auch die in Kapitel 3.3 erwähnte max_blocking_time wurde nicht implementiert.

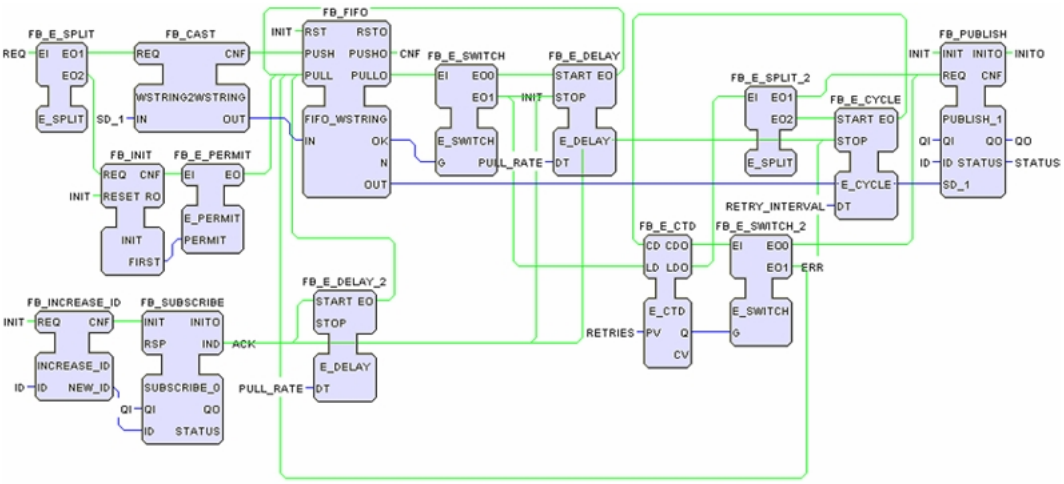

Abbildung 4.21: PUBLISH_RELIABILITY Netzwerk

SUBSCRIBE_RELIABILITY

SUBSCRIBE_RELIABILITY muss zusammen mit PUBLISH_RELIABILITY (_REAL bzw. _WSTRING) (siehe oben) verwendet werden und besitzt die gleichen Eingänge und Ausgänge wie ein SUBSCRIBE_1-FB.
Analog zu PUBLISH_RELIABILITY wird hier bei der Initialisierung ein Hilfs-Publisher (FB_PUBLISH) im Hintergrund initialisiert (siehe Abbildung 4.22). Dieser verwendet den nächst höheren Kanal (Port) (in Bezug auf den Subscriber) für seine Kommunikation mit dem Hilfs-Subscriber (auf Publisher-Seite). Bei Empfang einer Nachricht wird über den Hilfs-Publisher der Erhalt dieser Nachricht bestätigt.

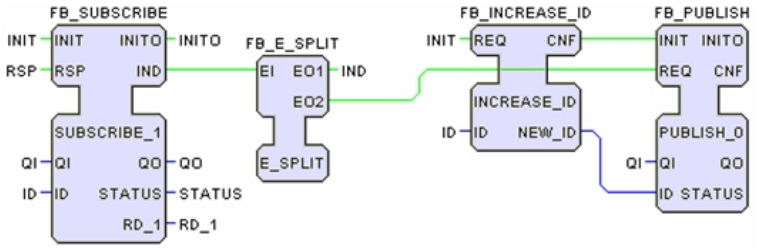

Abbildung 4.22: SUBSCRIBE_RELIABILITY Netzwerk

4.9 Festlegen der Sende- bzw. Empfangsrate

Die Umsetzung von definierten Zeitvorgaben kann durch das Festlegen der Sende- bzw. Empfangsrate geschehen (vgl. Kapitel 3.5.9). Dadurch wird erreicht, dass Daten nur so oft wie verlangt gesendet bzw. empfangen werden.

PUBLISH_TIME_BASED_FILTER
Um nicht mehr Daten als angegeben zu verschicken, muss der Publisher für die Dauer der Senderate (SEND_RATE) blockieren, was relativ einfach mit einem Verzögerungsglied (FB_E_DELAY) gelöst werden kann (siehe Abbildung 4.23 und 4.24). Eine Überprüfung auf Neuheit der Daten erfolgt mit dem Basic-FB EQUAL_REAL bzw. EQUAL_WSTRING (siehe Anhang). Wenn sich das Datum ändert, so wird es unter Verletzung von SEND_RATE trotzdem gesendet, ansonsten wird es zumindest mit der durch den Applikationsentwickler frei wählbaren Rate verschickt.
Zu beachten ist, dass das zyklische Versenden der Daten nicht durch das INIT-Event angestoßen wird sondern durch ein REQ-Event, wobei das erste Datum sofort und nicht verzögert verschickt wird.
PUBLISH_TIME_BASED_FILTER (_REAL und _WSTRING) kann zusammen mit einem SUBSCRIBE_1-FB verwendet werden, funktioniert aber auch in Kombination mit SUBSCRIBE_TIME_BASED_FILTER (siehe unten). Die zweite Variante ist jedoch nur dann sinnvoll, wenn gilt: Empfangsrate $\leq$ Senderate.

SUBSCRIBE_TIME_BASED_FILTER
Der Subscriber blockiert für die Dauer der MIN_SEP, d.h. die Daten werden dadurch im Takt der MIN_SEP (oder langsamer) ausgegeben (siehe Abbildung 4.25 und 4.26). Diese Funktionsweise wird im Wesentlichen durch ein Flip-Flop (FB_E_SR), das nach Ablauf der MIN_SEP (siehe FB_DLY)

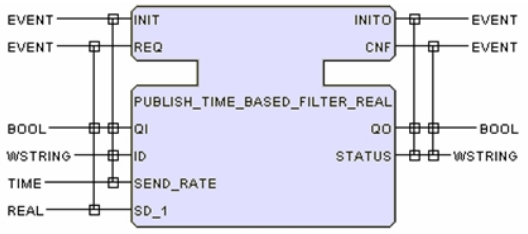

Abbildung 4.23: PUBLISH_TIME_BASED_FILTER FB

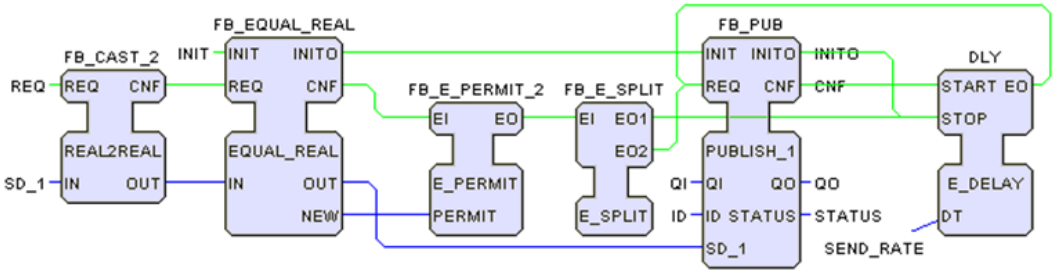

Abbildung 4.24: PUBLISH_TIME_BASED_FILTER Netzwerk

zurückgesetzt wird, erreicht.
Falls der Publisher in der Zeit des Blockierens Daten verschickt, gehen diese nicht verloren: FB_E_PERMIT_2 erkennt, wenn während MIN_SEP Events auftreten; über FB_E_REND wird der aktuelle Wert nach Ablauf von MIN_SEP ausgegeben.
Als Partner-FB von SUBSCRIBE_TIME_BASED_FILTER kann ein gewöhnlicher PUBLISH_1 (oder PUBLISH_TIME_BASED_FILTER, siehe oben) genommen werden.

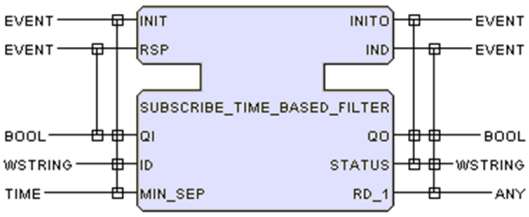

Abbildung 4.25: SUBSCRIBE_TIME_BASED_FILTER FB

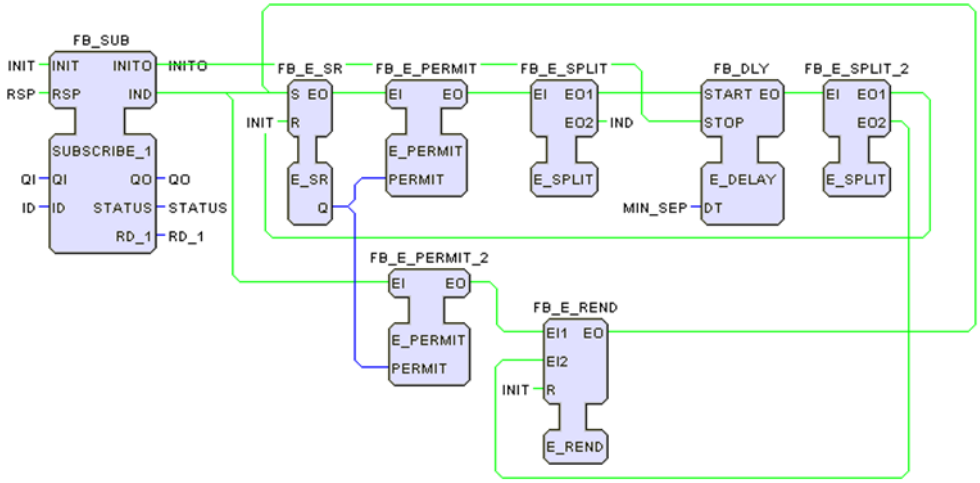

Abbildung 4.26: SUBSCRIBE_TIME_BASED_FILTER Netzwerk

4.10 Berücksichtigung von Gültigkeiten

Daten sind oft nur für eine bestimmte Zeit lang gültig (siehe Kapitel 3.5.10). Die Festlegung einer Gültigkeitsdauer erfolgt auf Publisher-Seite, die Überprüfung auf Gültigkeit auf Subscriber-Seite.

PUBLISH_VALID_TIME
Der Benutzer hat die Möglichkeit, eine Gültigkeitsdauer (VALID_TIME) für die zu versendenden Daten (SD_1) anzugeben (siehe Abbildung 4.27). Mit Anforderung des REQ-Events wird die momentane Systemzeit (über FB_CURR_TIME) ermittelt und mit der Gültigkeitsdauer addiert (FB_ADD, siehe Abbildung 4.28). Dieses „Ablaufsdatum“ (momentane Systemzeit + Gültigkeitsdauer) wird schließlich mit dem eigentlichen Datum mitverschickt. Zur Auswertung auf Subscriber-Seite wird ein SUBSCRIBE_VALID_TIME-FB (siehe unten) benötigt.

SUBSCRIBE_VALID_TIME
Dieser FB korrespondiert mit PUBLISH_VALID_TIME (siehe oben).
Das VALID-Flag am Ausgang zeigt an, ob die empfangenen Daten innerhalb der auf Publisher-Seite spezifizierten Gültigkeitsdauer angekommen sind (siehe Abbildung 4.29 und 4.30). Die empfangenen Daten (RD_1) werden immer ausgegeben, egal ob gültig oder nicht.
Nach Erhalt der Daten wird die aktuelle Systemzeit ermittelt

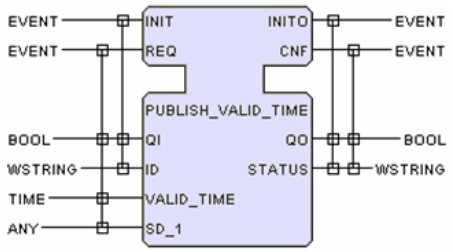

Abbildung 4.27: PUBLISH_VALID_TIME FB

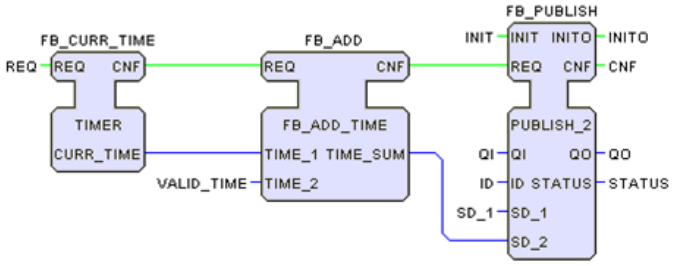

Abbildung 4.28: PUBLISH_VALID_TIME Netzwerk

(FB_CURR_TIME) und mit dem unter RD_2 empfangenen „Ablaufdatum“ verglichen (FB_VALIDATE_TIME). Wenn das Ablaufdatum noch in der Zukunft liegt, also wenn gilt: aktuelle Zeit < Ablaufdatum, dann wird VALID auf TRUE gesetzt.

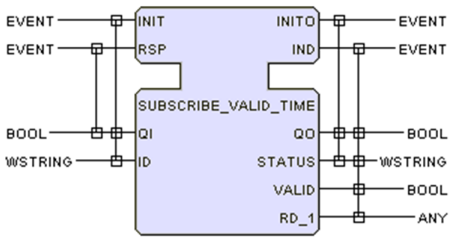

Abbildung 4.29: SUBSCRIBE_VALID_TIME FB

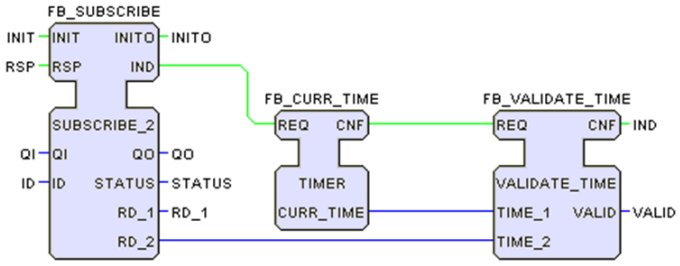

Abbildung 4.30: SUBSCRIBE_VALID_TIME Netzwerk

4.11 Empfang von unterschiedlichen Stellen (SUBSCRIBE_2_DATA_WRITERS)

In Kapitel 3.5.7 bzw. 4.7 wurde bereits das Prinzip der Segmentierung zur Unterscheidung zwischen Publishern mit gleichem Topic vorgestellt. Es muss allerdings festgehalten werden, dass beim FBDK eine Topic-basierte Unterscheidung der Nachrichten unabhängig von den IDs nicht möglich ist (es gibt beispielsweise auch keinen globalen Broadcast sondern nur einen ID-basierten), bzw. ist im Programmumfang des FBDKs kein Mechanismus zum Empfang von mehreren IDs vorgesehen. Mit den mitgelieferten Publish- und Subscribe-FBs sind somit sämtliche Kommunikationsformen (1-zu-1, 1-zu-n, n-zu-1 und n-zu-n) nur bei gleicher ID möglich. Darum soll in diesem Kapitel als Vorstufe für einen eingeschränkten Empfang von mehreren Stellen mit unterschiedlicher ID unter Verwendung einer Partition (siehe Kapitel 4.12) lediglich die Implementierung für den Empfang von zwei Stellen mit unterschiedlicher ID dargelegt werden.
Ein derartiger FB ist insofern wichtig, als dass durch ihn erst eine dem in Kapitel 2.1 beschriebenen Publish-Subscribe-Prinzip annähernd ähnliche Kommunikation möglich wird. Allerdings wird auch durch SUBSCRIBE_2_DATA_WRITERS keine Topic-basierte Kommunikation erreicht, da für den Empfang von einem Publisher dessen ID bekannt sein muss.[2]

Der implementierte Composite-FB (siehe Abbildung 4.31) empfängt REAL- bzw. WSTRING-Werte von zwei unterschiedlichen Data Writers (Publisher mit unterschiedlicher ID) und schreibt sie auf einen Ausgang (RD_1). Dafür

[2]Eine direkte Entsprechung von SUBSCRIBE_2_DATA_WRITERS mit einem in Kapitel 3.5 beschriebenen Mechanismus gibt es nicht. Vielmehr handelt es sich hier um eine durch SUBSCRIBE_PARTITION implizierte Ergänzung.

wird u.a. ein Multiplexer (FB_MUX_REAL bzw. FB_MUX_WSTRING, siehe Abbildung 4.32) benötigt, dem ein spezieller Konverter vorgeschaltet sein muss: FB_E_TO_UINT erkennt, welcher Subscriber gerade empfangen hat, und gibt diese Information an den Multiplexer weiter, welcher das jeweilige Datum am Ausgang (OUT) ausgibt.
Die Anzahl der Data Writers von SUBSCRIBE_2_DATA_WRITERS könnte problemlos angehoben werden. Mit den aus der Bibliothek entnommenen FBs E_TO_UINT und FB_MUX_REAL (bzw. _WSTRING) wäre eine Ausdehnung auf vier Data Writers möglich (siehe Abbildung 4.32).

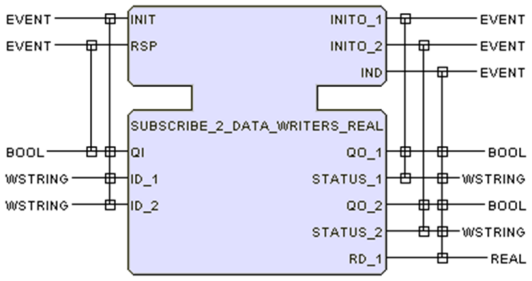

Abbildung 4.31: SUBSCRIBE_2_DWs FB

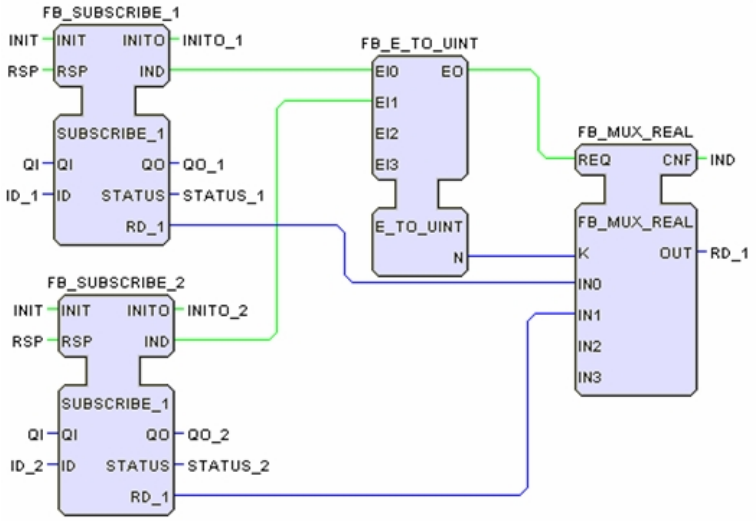

Abbildung 4.32: SUBSCRIBE_2_DWs Netzwerk

4.12 Eingeschränkter Empfang von unterschiedlichen Stellen (SUBSCRIBE_3_DW_PARTITION)

Der in diesem Kapitel vorgestellte FB (siehe Abbildung 4.33) kombiniert den Mechanismus der Segmentierung (vgl. Kapitel 3.5.7 und 4.7) mit dem des Empfangens von mehreren Publishern mit unterschiedlicher ID (siehe Kapitel 4.11).[3]
Es werden Daten von drei unterschiedlichen Stellen (ID_1, ID_2 und ID_3) empfangen (vgl. SUBSCRIBE_2_DATA_WRITERS). Zusätzlich kann eine PARTITION angegeben werden (vgl. SUBSCRIBE_PARTITION).
Das Funktionsblocknetzwerk ist in Abbildung 4.34 zu sehen. Der Aufbau ähnelt stark dem in Abbildung 4.32 gezeigten; der wesentliche Unterschied besteht in den Subscriber-FBs: Während in Abbildung 4.32 gewöhnliche SUBSCRIBE_1-FBs verwendet werden, wird hier auf den in Kapitel 4.7 beschriebenen SUBSCRIBE_PARTITION-FB und dessen Segmentierungs-Funktionalität zurückgegriffen.
Eine Einschränkung existiert bezüglich der Initialisierung (bzw. Deinitialisierung), da diese nicht für jeden Subscriber einzeln erfolgen kann. Eine eventuelle Verbesserungen bzw. Erweiterung wäre die Möglichkeit des voneinander unabhängigen Anmeldens (bzw. Abmeldens) bei einem oder zwei Data Writers.

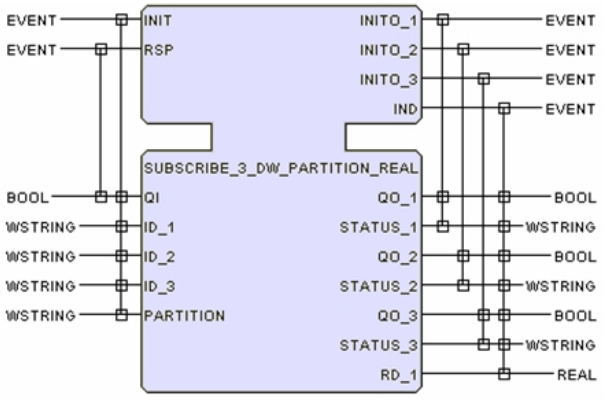

Abbildung 4.33: SUBSCRIBE_3_DW_PARTITION FB

[3] vgl. Fußnote 2 auf Seite 110

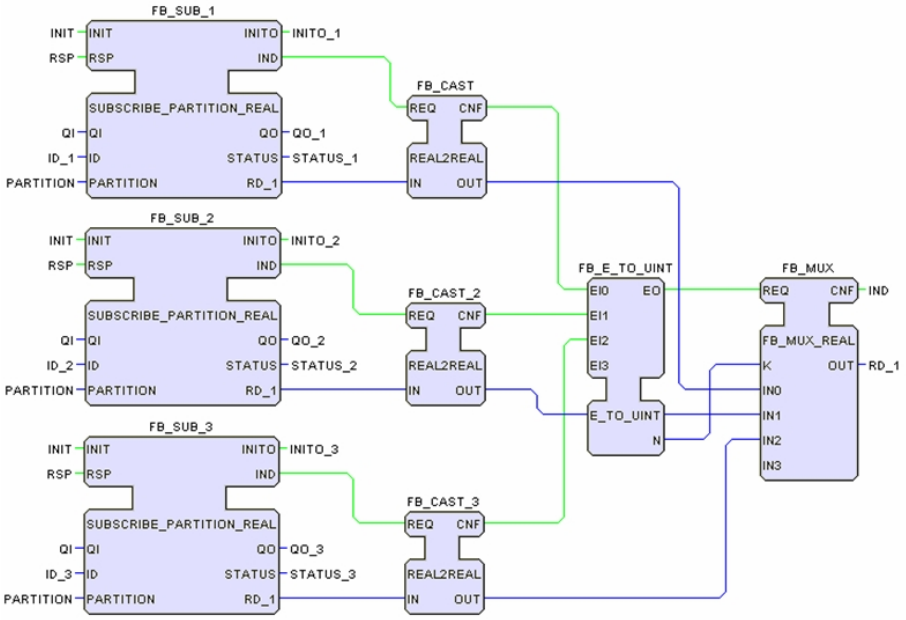

Abbildung 4.34: SUBSCRIBE_3_DW_PARTITION Netzwerk

4.13 Gefilterter Empfang (SUBSCRIBE_CONTENT_FILTER)

Ein gefilterter Empfang meint den selektiven Erhalt von Daten, d.h. es werden nur Daten empfangen, die gewisse Kriterien erfüllen (vgl. Kapitel 3.5.11). Die vorliegende Implementierung beschränkt sich auf den Empfang von REAL-Werten, die in einem spezifizierten Wertebereich liegen müssen.
Beim FB SUBSCRIBE_CONTENT_FILTER (siehe Abbildung 4.35) kann über die Dateneingänge MIN und MAX ein Wertebereich definiert werden. Wenn die empfangenen Daten (RD_1) in diesem Gültigkeitsbereich liegen, d.h. wenn MIN $\leq$ RD_1 $\leq$ MAX gilt, so wird das Event IND_1 generiert, andernfalls IND_2. Somit wird das Datum immer am Ausgang ausgegeben. Eine Entscheidung über dessen Gültigkeit erfolgt über die entsprechenden Events.
Der u.a. für dieses Verhalten verantwortliche, intern verwendete FB REAL_FILTER (siehe Abbildung 4.36) ist im Anhang beschrieben.

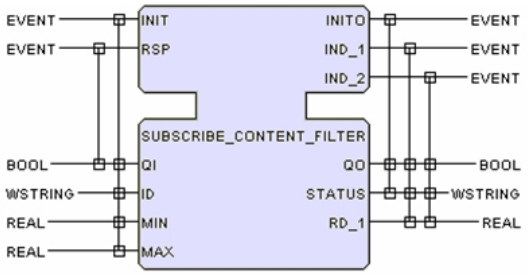

Abbildung 4.35: SUBSCRIBE_CONTENT_FILTER FB

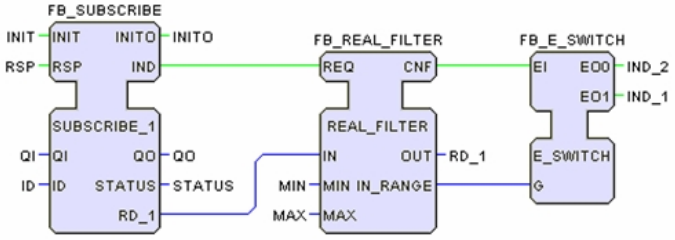

Abbildung 4.36: SUBSCRIBE_CONTENT_FILTER Netzwerk

4.14 Konsistentes Empfangen (SUBSCRIBE_PRESENTATION)

Kapitel 3.5.12 beschreibt den Mechanismus des konsistenten Empfangens. Dieser soll sicherstellen, dass Daten von unterschiedlichen Stellen gemeinsam genutzt werden können. Zu diesem Zweck müssen - je nach Implementierung - entweder die unterschiedlichen Sendezeiten der Daten, oder aber deren Empfangszeiten innerhalb eines definierten Zeitfensters liegen.
SUBSCRIBE_PRESENTATION baut auf dem zweiten Prinzip auf. Es werden die Daten von zwei Publishern empfangen (vgl. SUBSCRIBE_2_DATA_WRITERS, Kapitel 4.11). Beide Daten werden zusammen benötigt, d.h. es ist auf Konsistenz zu achten.
Der Benutzer hat die Möglichkeit, ein Zeitfenster (SLOT) anzugeben, in dem beide Daten empfangen werden müssen (siehe Abbildung 4.37 und 4.38). Sobald eines der beiden Daten ankommt, wird die aktuelle Systemzeit abgefragt (z.B. durch FB_TIMER_2). Wenn das zweite Datum ankommt, wird wiederum die Systemzeit abgefragt (durch FB_TIMER). Erst wenn von beiden Sendestellen Daten vorliegen (durch FB_E_REND sichergestellt), wird die Differenz gebildet (FB_SUB) und entsprechend ausgewertet (FB_VALIDATE, siehe Anhang), worauf auf die Konsistenz geschlossen und diese in Form des CONSISTENT-Flags angezeigt werden kann.
Ein Manko bezüglich der vorliegenden Implementierung ist, dass nicht garantiert ist, ob die beiden Daten wirklich zusammenpassen sondern nur, ob sie in passenden Abständen eintreffen.

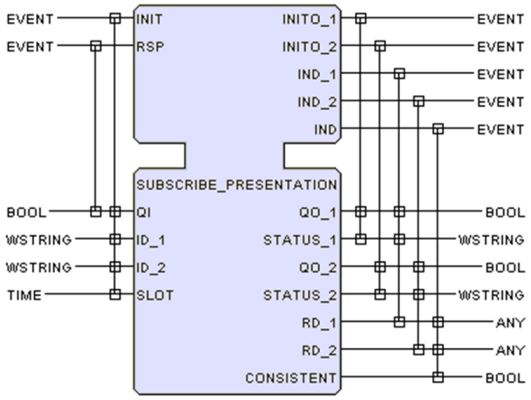

Abbildung 4.37: SUBSCRIBE_PRESENTATION FB

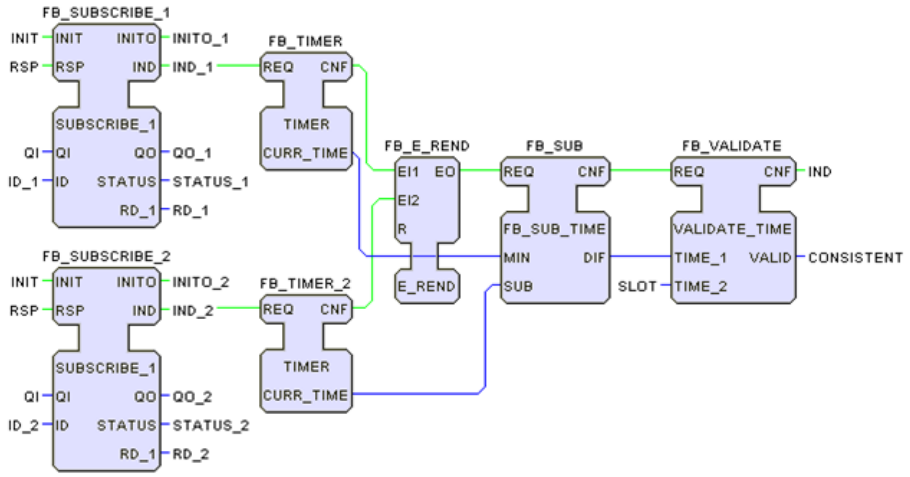

Abbildung 4.38: SUBSCRIBE_PRESENTATION Netzwerk

Bei der zweiten Implementierung das konsistente Empfangen betreffend (SUBSCRIBE_PRESENTATION2) wird nicht aufgrund der Empfangszeiten sondern aufgrund der Sendezeiten auf Konsistenz geschlossen (vgl. Kapitel 3.5.12). Hierzu muss aber auch die Publisher-Seite (siehe PUBLISH_TIMESTAMP) miteinbezogen werden. Außerdem wird die Gültigkeit der Daten überprüft.

PUBLISH_TIMESTAMP
Ähnlich PUBLISH_VALID_TIME (siehe Kapitel 4.10) - das Interface ist überhaupt gleich und wird hier daher nicht abgebildet - wird vorm Versenden der Daten die aktuelle Systemzeit (FB_CURR_TIME) ermittelt (siehe Abbildung 4.39). Außerdem kann der Benutzer für jedes Datum eine Gültigkeitsdauer (VALID_TIME) spezifizieren. Im Unterschied zu PUBLISH_VALID_TIME werden diese beiden Zeiten jedoch nicht addiert, sondern zusätzlich zum eigentlichen Datum separat mitverschickt. Somit hat jedes gesendete Datum (SD_1) einen Zeitstempel (SD_3) und eine Gültigkeit (SD_2).

SUBSCRIBE_PRESENTATION2
Wie bereits erwähnt wird bei dieser Implementierung (siehe Abbildung 4.40 und 4.41) nicht aufgrund der Empfangszeiten (siehe SUBSCRIBE_PRESENTATION) sondern aufgrund der Sendezeiten auf Konsistenz geschlossen, d.h. die Differenz (siehe FB_SUB) der beiden Zeitstempel der

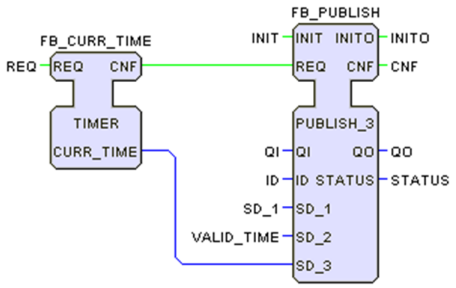

Abbildung 4.39: PUBLISH_TIMESTAMP Netzwerk

empfangenen Daten (jeweils RD_3 der beiden Subscriber) muss innerhalb des definierten Slots (SLOT) liegen (siehe FB_VALIDATE). Dieser Zeitvergleich kann erst durchgeführt werden, wenn von beiden Publishern Daten empfangen wurden (durch FB_E_REND gewährleistet); das Ergebnis wird in Form des CONSISTENT-Flags angezeigt.
Zusätzlich kann auch die Gültigkeit der empfangenen Daten (jeweils RD_2 der beiden Subscriber) berücksichtigt werden, wobei das VALID-Flag nur dann TRUE ist, wenn noch beide Daten gültig sind (siehe FB_AND).
Erst wenn das Ergebnis der Konsistenzüberprüfung feststeht, sowie beide Gültigkeiten ermittelt wurden (siehe FB_E_REND_2 und FB_E_REND_3), wird das Ausgangs-Event CV_IND generiert.
Die Implementierung bezüglich der Gültigkeits-Überprüfung ist analog zu der aus Kapitel 4.10 und wird hier nicht mehr erörtert.

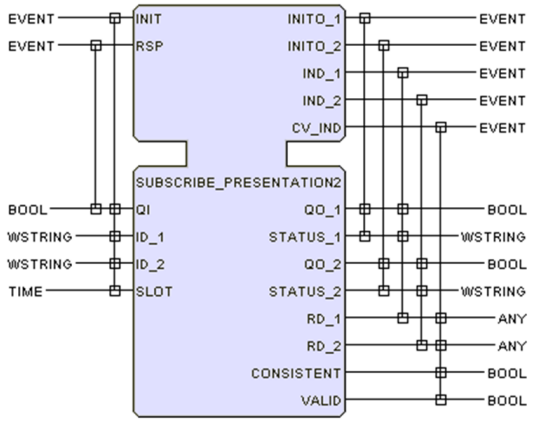

Abbildung 4.40: SUBSCRIBE_PRESENTATION2 FB

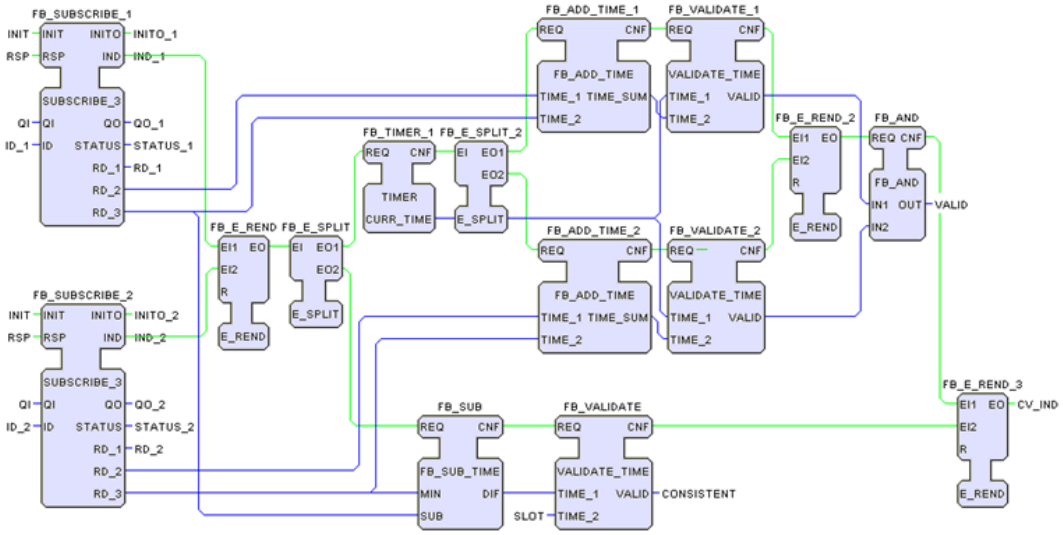

Abbildung 4.41: SUBSCRIBE_PRESENTATION2 Netzwerk

5 Diskussion der Ergebnisse

Die am Markt verfügbaren Middleware-Produkte und ihre Einsatzgebiete bzw. Einsatzmöglichkeiten sind unzählig. Oft wird durch die Middleware ein sehr breites Spektrum an Wünschen und Bedürfnissen des Benutzers bedient, um so in möglichst vielen Anwendungen zum Einsatz kommen zu können. Diese Allround-Produkte sind allerdings in erster Linie für den IT-Bereich (für herkömmliche Geschäftsprozesse) konzipiert, denn hier kann im Allgemeinen mit einer ausreichend leistungsfähigen Infrastruktur gerechnet werden.
Um innerhalb einer verteilten Anwendung nach IEC 61499 die Konfiguration der Kommunikation zu erleichtern, wurde versucht, diese Middleware-Funktionalität für die IAT nutzbar zu machen. Dabei können die Anforderungen der Applikation durch unterschiedliche QoS-Parameter erfasst, und in Folge die Kommunikation entsprechend konfiguriert werden.
Besagter Overhead der Middleware-Produkte ist jedoch, wie die in dieser Arbeit durchgeführte (Middleware-) Analyse gezeigt hat, für die IAT und ihre eingebetteten Systeme meist nicht leistbar. Ebenso können oft nicht alle Kommunikationsszenarien abgedeckt werden, bzw. reichen die von der Middleware zur Verfügung gestellten Services nicht aus, um die von der verteilten Anwendung gestellten Anforderungen zu beschreiben und somit zu bedienen.
Auch der Einsatz von Protokollen, welche QoS unterstützen, scheint nicht sehr vielversprechend, zumal man sich dadurch in seinen weiteren Möglichkeiten stark einschränkt.
Letztendlich bleibt die „Option“, Middleware-Funktionalität mit Mitteln der Steuerungstechnik (IEC 61499) nachzubilden. Die in dieser Arbeit geschaffenen, funktionalen Beschreibungen konzentrieren sich in erster Linie auf den aus der Analyse hervorgegangen, am besten geeigneten Ansatz, wobei die unter Experten durchgeführte Umfrage eine weitere Orientierungshilfe darstellt.
Die einzelnen Mechanismen konnten mit dem IEC 61499-konformen FBDK (Function Block Development Kit) der Firma Holobloc Inc., welcher für den privaten Gebrauch kostenlos unter `www.holobloc.com` heruntergeladen werden kann, umgesetzt werden. Konkret wurde eine Bibliothek an Kommunikations-FBs entwickelt, welche folgende QoS unterstützt:

Berücksichtigung der richtigen Reihenfolge bei der Initialisierung der Kommunikationsteilnehmer (Client/Server), Berücksichtigung von Antwortzeiten auf Anforderungen, Berücksichtigung von Deadlines, Einführung einer Sendeschlange, Mitverschicken der Historie, Einführung einer Segmentierung, verlässliches Senden bzw. Empfangen, Festlegen der Sende- bzw. Empfangsrate, Berücksichtigung einer Gültigkeitsdauer, Empfang von unterschiedlichen Stellen, gefilterter Empfang, konsistentes Empfangen sowie gruppenweise An- und Abmeldung der Kommunikations-FBs (Entity Factory).
Diese FBs versprechen einerseits einen erweiterten Funktionsumfang (beispielsweise ein verbessertes Monitoring oder eine leichtere Fehlersuche), auf der anderen Seite garantieren sie eine universelle Einsetzbarkeit, da nur geringe Anforderungen an die unterlagerte Kommunikation gestellt werden. Die gesamte Komplexität ist in den FBs (und somit in der Applikationsebene) enthalten, wodurch die untergeordneten Ebenen lediglich einfachste Aufgaben zu erfüllen haben (z.B. Bereitstellung von Protokollen für das Versenden von einfachen Datentypen).
Auch wenn die Ergebnisse speziell auf dem FBDK und somit auf Ethernet basieren, ist eine Übertragung auf andere Kommunikationsmedien einfach und problemlos möglich, denn die FBs an sich sind von keinem Hersteller abhängig.
Was die Funktionalität betrifft so resultieren im Vergleich zu Middleware-Lösungen nur eingeschränkte Möglichkeiten (beispielsweise konnte die Verwaltung von Instanzen nicht berücksichtigt werden). Im Hinblick auf einen sehr breit gefächerten Einsatz (Realisierung auf Anwendungsebene) wurden durch die Umsetzung auch keine busnahen Eigenschaften (z.B. Berücksichtigung von Prioritäten) abgedeckt. Deren Berücksichtigung ist jedoch durch den gewählten Lösungsweg prinzipiell nicht ausgeschlossen.
Letztendlich konnte die Umsetzung von Middleware-Funktionalität mit IEC 61499-Mitteln nicht nur generell demonstriert, sondern konkret eine QoS-Bibliothek für die unterschiedlichsten Einsatzbereiche entwickelt werden.

6 Schlussfolgerung

In diesem Kapitel sollen die Ergebnisse und die Aufgabenstellung (Ziele) gegenübergestellt werden.
Dazu lässt sich sagen, dass dem Wunsch nach einer einfachen Konfiguration der Kommunikation innerhalb einer verteilten Anwendung nach IEC 61499 entsprochen wurde. Durch die Erstellung mehrerer Kommunikations-FBs mit unterschiedlichen Eigenschaften (QoS) stehen dem Applikationsentwickler viele Möglichkeiten der Konfiguration offen. Es wurde bei den einzelnen Implementierungen großer Wert auf Praxisnähe und Anschaulichkeit (bezüglich der Benennungen) gelegt, wodurch ein Einsatz diverser FBs (innerhalb konkreter Anwendungen) unkompliziert bzw. deren Funktionen und Verhalten selbst sprechend werden.
Für die Middleware-Analyse wurde insgesamt viel Zeit aufgewendet, da im Vorhinein noch nicht klar war, ob eine bestimmte Middleware direkt eingesetzt werden kann. Ein derartiger Einsatz hätte u.U. den Vorteil eines erweiterten Funktionsumfangs gehabt, wobei abgeschätzt hätte werden müssen, inwieweit dies auf Kosten der Performance gegangen wäre. Denn die Nachbildung besagter Funktionalität mit Mitteln der Steuerungstechnik ist auch nur eingeschränkt möglich. Damit ist gemeint, dass eine Überladung von FBs mit vielen Features zwar prinzipiell durchführbar ist, in Anbetracht des dadurch entstehenden Overheads aber eher davon abzuraten ist.
Letztendlich wurde durch diese Arbeit auch der Grundstein für eine Tool-Erweiterung innerhalb des εCEDAC-Projekts gelegt. Für ein automatisiertes Einfügen von Kommunikations-FBs steht nun eine Auswahl an FBs für die unterschiedlichsten Anforderungen bereit.

7 Aussichten und Perspektiven

Die im Zuge dieser Arbeit erstellte Kommunikationsfunktionsblock-Bibliothek wurde nach bestem Wissen und Gewissen auf Fehler untersucht und korrigiert und steht somit für einen konkreten Einsatz bereit.
Natürlich ist kein Programm perfekt, d.h. eventuelle Verbesserungen und Erweiterungen der einzelnen Implementierungen sind durchaus möglich und wurden auch schon zum Teil im Text angedeutet.
Besonders interessantes und nützliches Erweiterungspotential findet sich im Hinblick auf die Kombination mehrerer unterschiedlicher QoS-Eigenschaften. So steht jeder der entwickelten FBs meist für einen konkreten QoS-Parameter; beispielsweise existiert ein FB für verlässliche Kommunikation und ein anderer wiederum, der gewisse Zeitbedingungen garantiert. Eine Vereinigung dieser beiden Funktionalitäten in einem FB wäre jedoch durchaus denkbar und sinnvoll.

Leider konnte aus zeitlichen sowie inhaltlichen Gründen der Einsatz diverser FBs innerhalb einer konkreten Applikation nicht untersucht werden. D.h. das Zusammenspiel mehrerer (ressourcenarmer) Prozessoren zur Erzielung einer automatisierungstechnischen Aufgabe unter Nutzung der entwickelten FBs wurde nicht betrachtet. Eine Abschätzung der für einen Einsatz nötigen Voraussetzungen (Aufwand) bzw. des durch die FBs eingeführten Overheads (beispielsweise unter Verwendung des implementierten Timers zur Ausgabe der Systemzeit) wäre auf jeden Fall notwendig. Spezielle Benchmarks müssten überlegt und für eine Evaluierung durchgeführt werden.
In diesem Zusammenhang wäre auch die in Kapitel 1.2 erwähnte Tool-Erweiterung eine mögliche Fortführung dieser Arbeit.

Letztendlich ist auch der Einsatz einer vorhandenen Middleware überlegenswert. Die Voraussetzungen hierfür wurden durch die in dieser Arbeit durchgeführte Middleware-Analyse geschaffen. Auch wenn momentan keines der untersuchten Produkte für einen Einsatz in der IAT in Frage zu kommen scheint, so ist zumindest die Abänderung einer Middleware zu einer abgespeckten Version eine Option.

A Anhang

Die folgenden Ergänzungen betreffen einerseits die SDL, andererseits die Implementierungen gewisser FBs, die in Kapitel 4 nicht näher erklärt wurden.

SDL

Die Specification and Description Language (SDL) wurde von der ITU-T (International Telecommunication Union - Telecommunication Standardization Sector) in der Empfehlung Z.100 definiert und ist eine formale Sprache in graphischer Ausprägung. Sie wird gerne im Bereich der Telekommunikation zur Beschreibung und Modellierung von (verteilten) Systemen eingesetzt, u.a. für echtzeitfähige, reaktive (ereignisgesteuerte) Applikationen und die darin vorkommenden Kommunikationsabläufe.

Die Notation sowie die Regeln von SDL sind einigermaßen komplex und auf Anhieb nicht unbedingt leicht verständlich. Doch sie garantieren eindeutiges Verhalten. Die wichtigsten Symbole sollen im Folgenden kurz erklärt werden (siehe Abbildung A.1):

Abbildung A.1: Grundsymbole der SDL

Das Startsymbol präsentiert sich als Rechteck mit zwei Halbkreisen links und rechts. Es kennzeichnet den Anfang eines Prozesses und trägt manchmal die gleiche Bezeichnung wie der Prozess.
Durch das Stopp-Element, dargestellt durch ein x, wird der Prozess beendet.
Ein Zustand wird innerhalb der SDL durch ein Rechteck mit zwei abgeflachten Halbkreisen markiert (nicht zu verwechseln mit dem Startsymbol).
Ein Input wird durch ein Rechteck mit ausgeschnittenem Dreieck, ein Output durch ein Rechteck mit seitlich aufgesetztem Dreieck dargestellt.

Entscheidungen (auch informelle Abfragen genannt) werden durch eine Raute gekennzeichnet und können beliebig viele Verzweigung besitzen.
Die einzelnen Elemente sind durch Pfeile verbunden.

Implementierung

Die in Kapitel 4 beschriebenen FBs basieren zum Teil auf im Programmumfang enthaltenen, zum anderen auf selbst entwickelten FBs. Zweitere Kategorie soll nun umrissen werden:

CONVERT_UINT_TO_WSTRING
Dieser FB wird von INCREASE_ID benötigt und wandelt einen UINT in einen WSTRING um.

CONVERT_WSTRING_TO_UINT
Es handelt sich hierbei um das Pendant zu CONVERT_UINT_TO_WSTRING, das einen WSTRING in einen UINT umwandelt.

E_CHRON2 (vgl. mit dem im FBDK enthaltenen E_CHRON)
Durch E_CHRON2 wird eine Stoppuhr implementiert. Beim Start dieser Stoppuhr gibt es (im Gegensatz zu E_CHRON) keine Bestätigung (Event) am Ausgang (und insofern auch keine Zeitausgabe). Mit dem STOP-Ereignis wird die Uhr gestoppt und kann sodann nur mehr über einen neuen START-Request gestartet werden. REQ liefert die momentane Zwischenzeit, welche beliebig oft angefordert werden kann, ohne die Stoppuhr neu starten zu müssen.

E_SEL_WSTRING3
Je nach Event am Eingang wird ein entsprechender WSTRING am Ausgang ausgegeben (EI0 für IN0, EI1 für IN1 und EI2 für IN2). Im Prinzip wird die gleiche Funktionalität wie durch E_SELECT_WSTRING zur Verfügung gestellt, nur hat man mit E_SEL_WSTRING3 drei Auswahlmöglichkeiten. (E_SEL_WSTRING3 wird innerhalb CLIENT_INIT verwendet.)

E_SPLIT3 (vgl. E_SPLIT)
Ein Event wird auf drei Events aufgeteilt.

E_TIME
Es wird die Zeit zwischen zwei aufeinander folgenden Events gemessen. Beim ersten Request beträgt die Zeitdifferenz 0 Sekunden (weil ja noch kein anderes Ereignis vorangegangen ist). Ein Reset versetzt den FB wieder in den Anfangszustand (d.h. der erste Request hat wieder eine Zeitdifferenz von 0 Sekunden zur Folge).

EQUAL_REAL
EQUAL_REAL ist die analoge Implementierung zu EQUAL_WSTRING (siehe unten) für REAL-Werte.

EQUAL_WSTRING
Der WSTRING am Eingang wird an den Ausgang weitergegeben. Wenn sich der WSTRING im Vergleich zum vorhergehenden WSTRING geändert hat, wird das Ausgangs-Flag NEW auf True gesetzt, ansonsten auf False.
Der WSTRING wird für einen Vergleich unter der internen Variablen PREVIOUS abgespeichert. Der Initialisierungswert von PREVIOUS ist „InitialValue“, d.h. falls es vorkommt, dass beim ersten Request (nach dem INIT-Event) der WSTRING „InitialValue“ am Eingang anliegt, so liefert der FB ein falsches Ausgangsflag (nämlich False).

FB_ADD_TIME
Dieser FB addiert zwei Werte vom Typ TIME.

FB_ADD_UINT
FB_ADD_UINT wird beispielsweise in INCREASE_ID benötigt und dient der Addition zweier UINT-Werte.

FB_SUB_TIME
Es wird die Differenz zweier Zeiten berechnet, wobei der Betrag der Differenz ausgegeben wird.

FB_SUB_UINT
FB_SUB_UINT bildet die Differenz zweier UINTS (DIF = MIN - SUB).

FIFO_REAL
Durch FIFO_REAL wird eine Schlange nach dem FIFO-Prinzip für REAL-Werte realisiert. Eine genauere Darstellung (allerdings für WSTRINGs) findet sich unter FIFO_WSTRING (siehe unten).

FIFO_WSTRING
FIFO_WSTRING ist die Implementierung einer Schlange (Queue) nach dem FIFO-Prinzip und ist nur für WSTRINGs geeignet. Es können maximal 128 Elemente gespeichert werden, alle weiteren WSTRINGs werden verworfen.
Am Ausgang wird durch das OK-Flag angezeigt, ob unter OUT ein gültiger Wert anliegt. Außerdem wird die Anzahl der Elemente in der Schlange angezeigt (N).

INCREASE_ID
Durch diesen FB wird die (Portnumber einer) ID um eins erhöht, wobei die ID folgende Form haben muss: 255.0.0.1:1234 (die Zahl nach dem Doppelpunkt entspricht der Portnumber).
Es wird nicht überprüft, ob die resultierende ID gültig ist, d.h. es kann passieren, dass die höchst mögliche Port-Nummer überschritten wird.

INIT
Ein Ausgangsflag wird beim ersten Request (nach dem Reset-Ereignis) auf True gesetzt und bei allen weiteren Requests auf False.

PUBLISH_REAL
PUBLISH_REAL dient lediglich Testzwecken und kann innerhalb einer Applikation (System) durch einen herkömmlichen Publisher (z.B. PUBLISH_1), dem an seinen ANY-Eingang (SD_1) explizit REALs zum Versenden übergeben werden, ersetzt werden. Im Launch-Modus können mit dem vorgefertigten Publisher (PUBLISH_1) keine REALs versendet werden.

QUEUE (_REAL und _WSTRING)
Anliegende REAL- bzw. WSTRING-Elemente werden mit dem ENQ-Ereignis in einem (REAL- bzw. WSTRING-) ARRAY abgelegt. Die maximale Anzahl an speicherbaren Elementen ist auf 20 beschränkt. Wenn die Schlange voll ist, wird das älteste Element verworfen, und das neue Element aufgenommen.
Auch wenn die Speicherung der Elemente an das Warteschlangen-Prinzip erinnert, so wird dennoch kein FIFO-Prinzip realisiert, weil bei einem DEQ-Event kein einzelnes Element entnommen, sondern das gesamte Array ausgegeben wird. Außerdem wird durch DEQ kein Element aus der Schlange entfernt.
Das Reset-Ereignis ist bei REAL-Werten einigermaßen problematisch: sämtliche Elemente werden nämlich auf -1 initialisiert.

READ_OUT_ARRAY (_REAL und _WSTRING)
Dieser FB liest aus einem Array mit 20 Elementen ein bestimmtes Element, welches durch INDEX angegeben werden muss, aus. Die Bedingung für ein REQ-Event ist $1 \leq \text{INDEX} \leq 20$ und ist aus dem ECC ersichtlich.

REAL2REAL_ARRAY
Ein Array mit 20 REALs wird auf ein Array mit 20 REALs gecastet.

REAL_FILTER
Der Eingangswert (IN) wird immer an den Ausgang übergeben. Wenn gilt, $\text{MIN} \leq \text{IN} \leq \text{MAX}$, so wird das IN_RANGE-Flag auf True gesetzt.

SUBSCRIBE_REAL
Mit SUBSCRIBE_REAL können explizit REAL-Werte empfangen werden, was für das Testen einzelner FBs sinnvoll ist. Innerhalb einer Anwendung ist dieser FB allerdings überflüssig, weil die gleiche Wirkung mit einem herkömmlichen SUBSCRIBE-FB (mit ANY-Ausgang) erzielt werden kann.

TIMER
Es wird die momentane Systemzeit ausgegeben, nämlich als Differenz zwischen der aktuellen Zeit und dem Zeitpunkt „Mitternacht, 1. Jänner 1970, UTC“). Die Einheit des Rückgabewertes ist Millisekunden, die tatsächliche Auflösung hängt jedoch vom zugrunde liegenden Betriebssystem ab.

VALIDATE_TIME
Bei diesem FB müssen bei einem Request zwei Zeiten an den Dateneingängen anliegen. Diese Zeiten werden miteinander verglichen. Wenn gilt: $\text{TIME\_1} < \text{TIME\_2}$, so wird das Ausgangsflag VALID auf True gesetzt, ansonsten auf False.

WSTRING2WSTRING_ARRAY
Durch WSTRING2WSTRING_ARRAY wird ein Array mit 20 WSTRINGs auf ein Array mit 20 WSTRINGs gecastet.

Abkürzungsverzeichnis

εCEDAC	Evolution Control Environment for Distributed Automation Components
ACE	ADAPTIVE Communication Environment
ADAPTIVE	A Dynamically Assembled Protocol Transformation, Integration, and eValuation Environment
ADO	ActiveX Data Objects
AOT	Ahead-Of-Time
API	Application Programming Interface
ARS	Address Resolution Service
ASP	Active Server Pages
ATC	Asynchronous Transfer of Control
B2B	Business-to-Business
BBN	Bolt Beranek and Newman
CAN	Controller Area Network
CDR	Common Data Representation
CF	Compact Framework
CIL	Common Intermediate Language
CLR	Common Language Runtime
CLSID	Class ID
CNF	Confirmation
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
CQoS	Configurable QoS
CSMA/CD	Carrier Sense Multiple Access/Collision Detection
CTS	Common Type System
DCE	Distributed Computing Environment
DCOM	Distributed Component Object Model
DCPS	Data-Centric Publish-Subscribe
DDS	Data Distribution Service
DiffServ	Differentiated Services
DII	Dynamic Invocation Interface
DLRL	Data Local Reconstruction Layer

DOME	Distributed Object Model Environment
DOS	Disk Operating System
DP	Dezentrale Peripherie
DRE	Distributed, Real-Time, Embedded
DSI	Dynamic Skeleton Interface
ECC	Execution Control Chart
eCos	embedded Configurable operating system
EE	Enterprise Edition
EEPROM	Electrically Erasable Programmable Read-Only Memory
EIB	Europäischer Installationsbus
EJB	Enterprise JavaBeans
eRTOS	embedded Real-Time Operating System
FB	Function Block
FBDK	Function Block Development Kit
FBRT	Function Block Runtime
FCL	Framework Class Library
FIFO	First In - First Out
FT	Fault-Tolerant
GBit	Gigabit
GIOP	General Inter-ORB-Protocol
GNU	GNU is not Unix
GPL	General Public License
GPRS	General Packet Radio Service
GSM	Global System for Mobile Communications
GUID	Globally Unique Identifiers
HLC	High Level Communication Service
HMI	Human-Machine Interface
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
IAT	Industrielle Automatisierungstechnik
ID	Identity
IDL	Interface Definition Language
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IIOP	Internet Inter-ORB-Protocol
IND	Indication
INIT	Initialisation
INITO	Initialisation Confirm
INS	Interoperable Naming Service

IOP	Inter-ORB-Protocol
IP	Internet Protocol
IPX	Internetwork Packet eXchange
IrDA	Infrared Data Association
ISO	International Standards Organization
IT	Informationstechnik
ITEA	Information Technology for European Advancement
ITU-T	International Telecommunication Union - Telecommunication Standardization Sector
J2EE	Java 2 Platform, Enterprise Edition
J2SE	Java 2 Platform, Standard Edition
JCA	J2EE Connector Architecture
JEB	Java Embedded Bus
JIT	Just-in-Time
JMS	Java Message Service
JMX	Java Management Extension
JNDI	Java Naming and Directory Interface
JTS	Java Transaction Service
JVM	Java Virtual Machine
KB	Kilobyte
KW	Klöpper und Wiege
LLC	Low Level Communication Service
LON	Local Operating Network
m	Meter
MB, MByte	Megabyte
MBit	Megabit
ME	Millennium Edition
MHz	Megahertz
MiRPA	Middleware for Robotic and Process Control Applications
MPI	Message Passing Interface
MPLS	Multiprotocol Label Switching
MQC	MicroQoSCORBA
ms	Millisekunden
MSIL	Microsoft Intermediate Language
MSMQ	Microsoft Message Queuing
MTA	Multithreaded Apartment
MTBF	Mean Time between Failures
MTS	Microsoft Transaction Server
MTTF	Mean Time to Failure

Mutex	Mutual Exclusion
MW	Middleware
NDDS	Network Data Distribution Service
NT	New Technology
NVRAM	Non Volatile Random Access Memory
OCERA	Open Components for Embedded Real-time Applications
OLE	Object Linking and Embedding
OMA	Object Management Architecture
OMG	Object Management Group
OO	objektorientiert
OPC	Openness, Productivity, Collaboration (früher: OLE for Process Control)
ORB	Object Request Broker
ORPC	Object RPC
ORTE	OCERA Real-Time Ethernet
OS	Operating System
OSA+	Open Service Architecture Platform for Universal Services
OSACA	Open System Architecture For Controls Within Automation Systems
OSF	Open Software Foundation
OSI	Open Systems Interconnection
OTS	Object Transaction Service
P2P	Peer-to-Peer
PA	Prozess-Automation
PAS	Publicly Available Specification
PC	Personal Computer
POA	Portable Object Adapter
PPP	Point-to-Point Protocol
PTP	Point-to-Point
PTP	Precision Time Protocol
Pub	Publisher
QC	Queued Component
QI	Event Input Qualifier
QO	Event Output Qualifier
QoS	Quality of Service
QuO	Quality Objects
RAM	Random Access Memory
RD	Received Data

REQ	Request
RMI	Remote Method Invocation
ROFES	Real-Time CORBA for Embedded Systems
RPC	Remote Procedure Call
RSP	Response
RSVP	Resource reSerVation Protocol
RT	Real-Time
RTI	Real-Time Innovations, Inc.
RTML	Real Time Markup Language
RTOS	Real-Time Operating System
RTPS	Real-Time Publish-Subscribe
RTSJ	Real Time Specification for Java
RWTH	Rheinisch-Westfälische Technische Hochschule
RxO	Requested/Offered
s	Sekunde
SAP	Systemanalyse und Programmentwicklung
SD	Data to Send bzw. Sent Data
SDL	Specification and Description Language
SE	Standard Edition
SIFB	Service Interface Function Block
SIRENA	Service Infrastructure for Real Time Embedded Networked Applications
SOAP	Simple Object Access Protocol
SPS	Speicherprogrammierbare Steuerung
SPX	Sequenced Packet Exchange
SQL	Structured Query Language
STA	Single-Threaded Apartment
Sub	Subscriber
TAO	The ACE ORB
TCP	Transmission Control Protocol
TCS	Total CORBA Solution
TORERO	Total Life Cycle Web-integrated Control
UDP	User Datagram Protocol
UML	Unified Modelling Language
URL	Uniform Resource Locator
USD	United States Dollar
UTC	Coordinated Universal Time
VB	Visual Basic
VM	Virtual Machine
WCET	Worst-Case Execution Time

WCF	Windows Communication Foundation
WS	Web Service
WSE	Web Services Enhancements
XML	Extensible Markup Language
XP	eXPerience
ZSD	Zeit-Sequenz-Diagramm
°C	Grad Celsius

Index

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[ACW+] Appelbaum, R., L. Claudio, J. Watson, M. Cline und M. Girou: *Notification Service*. http://www4.informatik.uni-erlangen.de/~geier/corba-faq/notification-service.html, zugegriffen am 19.06.2007.

[Bak01] Bakken, D. E.: *Middleware*, 2001. http://www.eecs.wsu.edu/~bakken/middleware.pdf, zugegriffen am 31.05.2007.

[BBN] BBN Technologies: *Quality Objects (QuO)*. http://quo.bbn.com/, zugegriffen am 19.06.2007.

[BK03] Bishop, T. A. und R. K. Karne (Herausgeber): *A survey of middleware*, Proceedings of ISCA 18th International Conference on Computers and Their Applications, Seiten: 254-258, USA, März 2003.

[Bla00] Blair, G. S.: *QoS in Middleware*. In: dependability.org, Workshop on „Time and Dependability“, 2000. http://www.dependability.org/wg10.4/timedepend/10-Blair.pdf, zugegriffen am 31.05.2007.

[BU] Brinkschulte, U. und T. Ungerer: *Homepage zum Buch „Mikrocontroller und Mikroprozessoren“ (Kapitel 8)*. Springer-Verlag, September 2002, http://www.informatik.uni-augsburg.de/~ungerer/books/microcontroller/, zugegriffen am 05.07.2007.

[CD01] Comella-Dorda, S.: *Component Object Model (COM), DCOM, and Related Capabilities*. Carnegie Mellon® Software Engineering Institute (SEI), März 2001. http://www.sei.cmu.edu/str/descriptions/com.html, zugegriffen am 31.05.2007.

[COR] CORBA. http://corba.org/, zugegriffen am 31.05.2007.

[Dev] DevX: *MSMQ for .NET Developers (Part 1)*. http://www.devx.com/dotnet/Article/27560/1954?pf=true, zugegriffen am 19.06.2007.

[DFK04] Diethers, K., B. Finkemeyer und N. Kohn: *Middleware zur Realisierung offener Steuerungssoftware für hochdynamische Prozesse*. it - Information Technology, 01:39–47, 2004. Oldenbourg Verlag.

[Evo] Evolution Control Environment for Distributed Automation Components (εCEDAC). `http://www.easydac.org/`, zugegriffen am 31.05.2007.

[FB04] Favre-Bulle, B.: *Automatisierung komplexer Industrieprozesse. Systeme, Verfahren und Informationsmanagement*. Springer, Wien, 1. Auflage, September 2004.

[Fel00] Felser, M.: *Ethernet als Feldbus? Kommunikationsmodelle für Industrielle Netzwerke*. Infobit, 3:21–24, 2000. `http://felser.ch/download/FE-TR-0005.PDF`, zugegriffen am 31.05.2007.

[GD] Golatowski, F. und M. Ditze: *SIRENA, Service Infrastructure for Real Time Embedded Networked Applications*. `http://www.softwarefoerderung.de/projekte/sirena/beitrag_SIRENA.pdf`, zugegriffen am 02.08.2007.

[HA98] Helander, J. und Forin A. (Herausgeber): *MMLite: A Highly Componentized System Architecture*, Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications, Seiten: 96-103, Jänner 1998. `http://research.microsoft.com/~jvh/MMLite-sigops.pdf`, zugegriffen am 31.05.2007.

[Ham05] Hammerschall, U.: *Verteilte Systeme und Anwendungen: Architekturkonzepte, Standards und Middleware-Technologien*. Paerson Studium, 2005.

[Hol] Holobloc Inc. `www.holobloc.com`, zugegriffen am 31.05.2007.

[HRHS01] He, J., M. Rajagopalan, M. A. Hiltunen und R. D. Schlichting (Herausgeber): *Providing QoS Customization in Distributed Object Systems*, Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms, November 2001. `http://www.cs.arizona.edu/~hejun/papers/middleware01.pdf`, zugegriffen am 31.05.2007.

[Inf] Information Technology for European Advancement (ITEA): *SIRENA*. `http://www.sirena-itea.org/`, zugegriffen am 02.08.2007.

[Ins04] Institute of Electrical and Electronics Engineers (IEEE): *Precision clock synchronization protocol for networked measurement and control systems (IEEE 61588: 2004 (1588-2002))*, September 2004. `http://www.ieee.org`, zugegriffen am 07.07.2007.

[Int96] International Standards Organization (ISO): *Information technology - Open Systems Interconnection - Basic Reference Model: The Basic Model (ISO/IEC 7498-1:1994(E))*, 1996. `http://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994(E).zip`, zugegriffen am 21.08.2007.

[Int98] International Standards Organization (ISO): *Information technology - Quality of service: Framework (ISO/IEC 13236:1998)*, 1998.

[Int03] International Electrotechnical Commission (IEC): *IEC 61131-3: Programmable controllers - Part 3: Programming languages*, Jänner 2003. `http://www.iec.ch`, zugegriffen am 12.08.2007.

[Jav] Java. `http://java.sun.com/`, zugegriffen am 31.05.2007.

[JSAD04] Jammes, F., H. Smit, C. Arandyelovitch und F. Depeisses (Herausgeber): *Intelligent Device Networking in Industrial Automation*, Proceedings of the 2nd IEEE International Conference on Industrial Informatics 2004 (INDIN '04), Seiten: 449-456, Berlin, Juni 2004. `http://www.sirena-itea.org/NR/rdonlyres/1BAD6918-CFEB-4B47-9844-6F1685BC2EC0/319/INDINarticleV1.pdf`, zugegriffen am 31.05.2007.

[Kar05] Karoui, R.: *CORBA/DDS, Competing or Complementing Technologies?* OMG Real-time and Embedded Systems Workshop, Juli 2005. `http://www.omg.org/news/meetings/workshops/RT_2005/03-1_Karoui.pdf`, zugegriffen am 19.06.2007.

[KDL00] Kim, D., Y. Doh und Y.-H. Lee (Herausgeber): *Java Real-Time Publish-Subscribe Middleware for Distributed Embedded Systems*, Band 189 der Reihe *IFIP Conference Proceedings*, 2000.

[KS] KW-Software. `www.kw-software.de`, zugegriffen am 31.05.2007.

[Lew01] Lewis, R.: *Modelling control systems using IEC 61499: Applying function blocks to distributed systems.* IEE Control Engineering Series 59, 2001.

[Lin06] Linstaedt, S.: *Integration von Agentenplattformen in Middleware - am Beispiel von Jadex und Java EE.* Diplomarbeit, Universität Hamburg, Fakultät für Mathematik, Informatik und Naturwissenschaften, Verteilte Systeme und Informationssysteme, April 2006.

[Low] Lowy, J.: *WCF Essentials. What You Need To Know About One-Way Calls, Callbacks, And Events.* `http://msdn.microsoft.com/msdnmag/issues/06/10/wcfessentials/default.aspx`, zugegriffen am 19.06.2007.

[LSZB98] Loyall, J. P., R. E. Schantz, J. A. Zinky und D. E. Bakken (Herausgeber): *Specifying and Measuring Quality of Service in Distributed Object Systems*, Proceedings of ISORC'98, Japan, 1998.

[Mar] Martin, P.: *.NET Remoting - Architekturbewertung.* `http://www.microsoft.com/germany/msdn/library/net/NETRemotingArchitekturbewertung.mspx?mfr=true`, zugegriffen am 19.06.2007.

[McK03] McKinnon, A. D.: *Supporting fine grained configurability with multiple quality of service properties in middleware for embedded systems.* Dissertation, Washington State University, School of Electrical Engineering and Computer Science, Dezember 2003.

[MDD+03] McKinnon, A. D., K. E. Dorow, T. R. Damania, O. Haugan, W. E. Lawrence, D. E. Bakken und J. C. Shovic (Herausgeber): *A Configurable Middleware Framework with Multiple Quality of Service Properties for Small Embedded Systems*, Proceedings of the Second IEEE International Symposium on Network Computing and Applications (NCA'03), Seiten: 197ff, 2003.

[Mica] MicroQoSCORBA. `http://www.microqoscorba.net/`, zugegriffen am 31.05.2007.

[Micb] MICROSOFT. http://www.microsoft.com, zugegriffen am 19.06.2007.

[Micc] MICROSOFT: *COM: Component Object Model Technologies.* http://www.microsoft.com/com/default.mspx, zugegriffen am 19.06.2007.

[Micd] MICROSOFT: *DCOM.* http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnanchor/html/dcom.asp, zugegriffen am 19.06.2007.

[Mice] MICROSOFT: *Einführung in .NET Framework 3.0.* http://www.microsoft.com/germany/msdn/library/net/EinfuehrungInNETFramework30.mspx?mfr=true, zugegriffen am 29.07.2007.

[Micf] MICROSOFT: *Microsoft Developer Network.* http://msdn2.microsoft.com/de-de/default.aspx, zugegriffen am 19.06.2007.

[Micg] MICROSOFT: *.NET Compact Framework.* http://www.microsoft.com/germany/msdn/library/net/compactframework/default.mspx?mfr=true, zugegriffen am 19.06.2007.

[Mich] MICROSOFT: *.NET Framework.* http://www.microsoft.com/germany/msdn/library/net/framework/default.mspx?mfr=true, zugegriffen am 19.06.2007.

[Mici] MICROSOFT: *.NET Framework-Entwicklerhandbuch: .NET Compact Framework.* http://msdn2.microsoft.com/de-de/library/f44bbwa1(VS.80).aspx, zugegriffen am 19.06.2007.

[Micj] MICROSOFT: *Windows Communication Foundation.* http://msdn2.microsoft.com/en-us/library/ms735119.aspx, zugegriffen am 19.06.2007.

[Mick] MICROSOFT: *Windows Communication Foundation Website.* http://wcf.netfx3.com/, zugegriffen am 01.06.2007.

[MT] MARTIN, R. und S. TOTTEN: *Introduction to Fault Tolerant CORBA.* In: Object Computing, Inc. http://www.ociweb.com/cnb/CORBANewsBrief-200301.html, zugegriffen am 31.05.2007.

[NDP+05] Narasimhan, P., T. A. Dumitras, A. M. Paulos, S. M. Pertet, C. F. Reverte, J. G. Slember und D. Srivastava: *MEAD: support for Real-Time Fault-Tolerant CORBA*. Concurrency and Computation: Practice & Experience, 17(12):1527–1545, 2005. `http://www.ece.cmu.edu/~mead/ccpe-2005.pdf`, zugegriffen am 31.05.2007.

[OCa] Object Computing, Inc.: *TAO Developer's Guide Excerpt, Chapter 31: Data Distribution Service*. `http://download.ociweb.com/DDS/DDS-0.12.pdf`, zugegriffen am 01.06.2007.

[OCb] Object Computing, Inc.: *TAO Product Information*. `http://www.theaceorb.com/product/index.html`, zugegriffen am 19.06.2007.

[OMGa] OMG. `http://www.omg.org/`, zugegriffen am 31.05.2007.

[OMGb] OMG: *CORBA Vendor Directory Listing*. `http://corba-directory.omg.org/vendor/list.htm`, zugegriffen am 19.06.2007.

[OMGc] OMG: *Data Distribution Service for Real-Time Systems Specification*. `http://www.omg.org/docs/formal/05-12-04`, zugegriffen am 19.06.2007.

[OMGd] OMG: *DDS VendorsPage*. `http://portals.omg.org/dds/VendorsPage`, zugegriffen am 19.06.2007.

[OMGe] OMG: *Free CORBA® Downloads*. `http://www.omg.org/technology/corba/corbadownloads.htm`, zugegriffen am 19.06.2007.

[OMGf] OMG: *Quality of Service for CORBA Components Specification*. Stand: April 2006.

[OMGg] OMG: *Real-time CORBA Specification*. `http://www.omg.org/docs/formal/05-01-04.pdf`, zugegriffen am 21.06.2007.

[OMGh] OMG: *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms*. `http://www.omg.org/docs/formal/06-05-02.pdf`, zugegriffen am 01.06.2007.

[OMGi] OMG: *UML Profile for Schedulability, Performance, and Time Specification*. `http://www.omg.org/docs/formal/05-01-02.pdf`, zugegriffen am 01.06.2007.

[OSA] OSACA. `http://www.osaca.org/`, zugegriffen am 05.06.2007.

[PB06] Picioroaga, F. und U. Brinkschulte: *Flexible QoS management and real-time in OSA+ middleware*. In: *The 2006 International Conference on Real-Time Computing Systems and Applications (RTCOMP´06), USA*, Juni 2006. `http://ww1.ucmss.com/books/LFS/CSREA2006/PDP3992.pdf`, zugegriffen am 01.06.2007.

[PB07] Pelz, M. und R. Birkhofer: *Vielfalt der Feldbusse (NAMUR-Beitrag)*. atp - Automatisierungstechnische Praxis, 49(2):67–70, Februar 2007.

[PBBS04] Picioroaga, F., A. Bechina, U. Brinkschulte und E. Schneider: *OSA+ Real-Time Middleware. Results and Perspectives*. In: *Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04)*, Seiten 147–154, 2004.

[PC05] Pardo-Castellote, G.: *OMG Data Distribution Service: Real-Time Publish/Subscribe Becomes a Standard*. Neuauflage des RTC-Artikels durch RTI, Jänner 2005. `http://www.rti.com/docs/reprint_rti.pdf`, zugegriffen am 01.06.2007.

[PCB02] Pardo-Castellote, G. und P. Bolton: *Distributed Real-Time Applications Now Have a Data Distribution Protocol*. Neuauflage des RTC-Artikels durch RTI, Februar 2002. `http://www.rti.com/docs/RTC_Feb02.pdf`, zugegriffen am 30.07.2007.

[PCFW05] Pardo-Castellote, G., B. Farabaugh und R. Warren: *An Introduction to DDS and Data-Centric Communications*, August 2005. `www.omg.org/news/whitepapers/Intro_To_DDS.pdf`, zugegriffen am 01.06.2007.

[PCH05] Pardo-Castellote, G. und G. A. Hunt: *DDS Enabling Global Data*. In: *OMG Burlingame Technical Conference*, Dezember 2005. `http://www.rti.com/docs/DDS_Enabling_Global_Data.pdf`, zugegriffen am 01.06.2007.

[PCSH99] Pardo-Castellote, G., S. Schneider und M. Hamilton: *NDDS: The Real-Time Publish-Subscribe Middleware.* In: Real-Time Innovations, Inc., August 1999.

[PDC04] Poon, P. M. S., T. S. Dillon und E. Chang (Herausgeber): *Transformation of QoS data into XML characterising data communication in Real Time Distributed Systems*, Proceedings of the 2nd IEEE International Conference on Industrial Informatics 2004 (INDIN '04), Seiten: 204-209, Berlin, Juni 2004.

[Pet03] Petig, M.: *.net und Echtzeit - kein Widerspruch!* Computer&Automation, 11, 2003. `http://www.kw-software.com/global_download_de/FA_NET_und_Echtzeit-kein_Widerspruch.pdf`, zugegriffen am 01.06.2007.

[Pic04] Picioroaga, F.: *Scalable and Efficient Middleware for Real-Time Embedded Systems. A Uniform Open Service Oriented, Microkernel Based Architecture.* Doktorarbeit, Université Louis Pasteur, Straßburg, 2004. `http://eprints-scd-ulp.u-strasbg.fr:8080/269/01/FPicioroaga_PhDThesis.pdf`, zugegriffen am 01.06.2007.

[Pri] PrismTech: *DDS and CORBA.* `http://www.prismtechnologies.com/section-item.asp?id=563&sid4=123&sid=4`, zugegriffen am 19.06.2007.

[RDN05] Riedl, M., C. Diedrich und F. Naumann (Herausgeber): *Function Block Applications Based on an Object Oriented Middleware*, Proceedings of the 3rd IEEE International Conference on Industrial Informatics 2005 (INDIN '05), Seiten: 19-24, Perth, August 2005.

[RDN06] Riedl, M., C. Diedrich und F. Naumann: *SFC inside IEC 61499.* In: *IEEE Conference on Emerging Technologies and Factory Automation 2006 (ETFA '06)*, Prag, September 2006.

[RDNS04] Riedl, M., C. Diedrich, F. Naumann und R. Simon: *An Object Based Approach for Distributed Automation.* In: *IEEE AFRICON 2004, 7th AFRICON Conference in Africa*, Band 2, Seiten 1253–1260, September 2004.

[Rei04] Reinecke, G.: *COM als Middleware.* In: ActiveVB, Februar 2004. `http://www.activevb.de/tutorials/tut_middleware/middleware.html`, zugegriffen am 31.05.2007.

[Rod02] Rodrigues, C.: *Using Quality Objects (QuO) Middleware for QoS Control of Video Streams.* OMG's Third Workshop on Real-Time and Embedded Distributed Object Computing, USA, Jänner 2002.

[ROF] ROFES. `http://www.lfbs.rwth-aachen.de/content/20`, zugegriffen am 01.06.2007.

[RTIa] Real-Time Innovations, Inc.: *Can Ethernet Be Real Time?* `http://www.rti.com/resources/whitepapers.html`, zugegriffen am 01.06.2007.

[RTIb] RTI: *Real-Time Innovations Creates Standards-Compliant Real-Time Networking Solution for Industrial Automation Applications.* `http://www.rti.com/corporate/news/industrial_automation.html`, zugegriffen am 19.06.2007.

[RTS] RTSJ: *Real Time Specification for Java.* `http://www.rtsj.org/`, zugegriffen am 19.06.2007.

[RTZ] RTZen. `http://doc.ece.uci.edu/rtzen`, zugegriffen am 01.06.2007.

[SFS05] Santner, G., M. Felser und H. Scheitlin: *Feldbusse ersetzen Kabelsalat.* Bulletin SEV/VSE, Artikelserie Automation: Kommunikation (3), Seiten 19–25, Juli 2005. `http://felser.ch/download/FE-TR-0502.pdf`, zugegriffen am 01.06.2007.

[SGV+05] Souto, E., G. Guimarães, G. Vasconcelos, M. Vieira, N. Rosa, C. Ferraz und J. Kelner: *Mires: a publish/subscribe middleware for sensor networks.* Personal and Ubiquitous Computing, 10(1):37–44, 2005.

[Sie03] Sierla, S. A.: *Middleware Solutions for Automation Applications - Case RTPS.* Diplomarbeit, Technische Universität Helsinki, Espoo, 2003. `http://www.rti.com/docs/RTPS_Thesis_HUT.pdf`, zugegriffen am 01.06.2007.

[Sie06a] Siegel, J.: *CORBA/E: CORBA für eingebettete Systeme.* OMG-Kolumne, März 2006. `http://www.sigs.de/publications/os/2006/03/OMG_OS_03_06.pdf`, zugegriffen am 01.06.2007.

[Sie06b] SIEGEL, J.: *Der Data Distribution Service der OMG.* OMG-Kolumne, Juni 2006. `http://www.sigs.de/publications/os/2006/06/OMG_OS_06_06.pdf`, zugegriffen am 01.06.2007.

[Sie07] SIEGEL, J.: *Modellierungs- und Middleware-Standards für verteilte Echtzeit- und eingebettete Systeme.* OMG-Kolumne, Jänner 2007. `http://www.sigs.de/publications/os/2007/01/OMG_OS_01_07.pdf`, zugegriffen am 19.06.2007.

[SK00] SCHMIDT, D. C. und F. KUHNS: *An Overview of the Real-Time CORBA Specification.* IEEE Computer Magazine, 33(6):56–63, Juni 2000.

[SL96] SPERLING, W. und P. LUTZ: *Enabling Open Control Systems - An Introduction to the OSACA System Platform.* Robotics and Manufacturing, 6, 1996. ISRAM'97 Montpellier/France, ASME Press, New York 1996. `http://www.osaca.org/_pdf/wac96.pdf`, zugegriffen am 05.06.2007.

[Sof] SOFTWARE, AG. `http://www.softwareag.com/us/`, zugegriffen am 21.06.2007.

[SPK05] SIERLA, S. A., J. P. PELTOLA und K. O. KOSKINEN (Herausgeber): *Real-Time Middleware for the Requirements of Distributed Process Control*, Proceedings of the 3rd IEEE International Conference on Industrial Informatics 2005 (INDIN '05), Seiten: 1-6, Perth, 2005.

[STL+04] SCHWAB, C., M. TANGERMANN, A. LÜDER, A. KALOGERAS und L. FERRARINI (Herausgeber): *Mapping of IEC 61499 Function Blocks to Automation Protocols within the TORERO Approach*, Proceedings of the 2nd IEEE International Conference on Industrial Informatics 2004 (INDIN '04), Seiten: 149-154, Berlin, Juni 2004.

[Str] STRUFE, T.: *Mircosoft DotNot. Verteilte Anwendungen „the microsoft(cc) way".* `http://eris.prakinf.tu-ilmenau.de/edu/va/folien/VL5.pdf`, zugegriffen am 01.06.2007.

[Sun] SUN MICROSYSTEMS: *Java Real-Time System (Java RTS).* `http://java.sun.com/javase/technologies/realtime/index.jsp`, zugegriffen am 19.06.2007.

[Sun02] Sun Microsystems: *Java Message Service Specification, Version 1.1*, April 2002. `http://java.sun.com/products/jms/docs.html`, zugegriffen am 19.06.2007.

[TAO] TAO. `http://www.cs.wustl.edu/~schmidt/TAO.html`, zugegriffen am 01.06.2007.

[TDP04] Thramboulidis, K. C., G. S. Doukas und T. G. Psegiannakis (Herausgeber): *An IEC-Compliant Field Device Model for Distributed Control Applications*, Proceedings of the 2nd IEEE International Conference on Industrial Informatics 2004 (INDIN '04), Seiten: 277-282, Perth, Juni 2004.

[THJ+05] Tommila, T., J. Hirvonen, L. Jaakkola, J. Peltoniemi, J. P. Peltola, S. A. Sierla und K. O. Koskinen: *Next generation of industrial automation. Concepts and architecture of a component-based control system.* In: VTT Industrial Systems, Espoo, 2005. `http://www.vtt.fi/inf/pdf/tiedotteet/2005/T2303.pdf`, zugegriffen am 01.06.2007.

[Thr05] Thramboulidis, K. C.: *IEC 61499 in Factory Automation.* In: *International Conference on Industrial Electronics, Technology & Automation (CISSE-IETA 05)*, Dezember 2005. `http://seg.ee.upatras.gr/thrambo/dev/Papers/IETA-05paper.pdf`, zugegriffen am 01.06.2007.

[TOR] TORERO. `http://www.uni-magdeburg.de/iaf/cvs/torero/`, zugegriffen am 02.07.2007.

[TT03] Tranoris, C. S. und K. C. Thramboulidis: *An IEC-compliant Engineering Tool for Distributed Control Applications.* In: *11th Mediterranean Conference on Control and Automation (MED '03)*, Juni 2003.

[TT04] Thramboulidis, K. C. und C. S. Tranoris: *Developing a CASE Tool for Distributed Control Applications.* The International Journal of Advanced Manufacturing Technology, 24(1–2), Juli 2004.

[TvS03] Tanenbaum, A. und M. van Steen: *Verteilte Systeme: Grundlagen und Paradigmen.* Paerson Studium, 2003.

[Val] VALESKY, T.: *The Free CORBA page*. http://adams.patriot.net/~tvalesky/freecorba.html, zugegriffen am 19.06.2007.

[WB05] WÖRN, H. und U. BRINKSCHULTE: *Echtzeitsysteme. Grundlagen, Funktionsweisen, Anwendungen*. Springer, Berlin, 2005.

[Wei04] WEIBEL, H.: *Uhren mit IEEE 1588 synchronisieren - Auf eine Mikrosekunde genau im lokalen Netzwerk*. Bulletin SEV/VSE (electrosuisse), 17:35–39, 2004. http://ines.zhwin.ch/uploads/media/Fachartikel_IEEE1588_02.pdf, zugegriffen am 07.07.2007.

[Wika] WIKIPEDIA, DIE FREIE ENZYKLOPÄDIE: *Carrier Sense Multiple Access/Collision Detection*. http://de.wikipedia.org/wiki/CSMA/CD, zugegriffen am 12.08.2007.

[Wikb] WIKIPEDIA, DIE FREIE ENZYKLOPÄDIE: *DiffServ*. http://de.wikipedia.org/wiki/DiffServ, zugegriffen am 30.07.2007.

[Wikc] WIKIPEDIA, DIE FREIE ENZYKLOPÄDIE: *Konsistenz (Datenbank)*. http://de.wikipedia.org/wiki/Konsistenz_%28Datenbank%29, zugegriffen am 30.07.2007.

[Wikd] WIKIPEDIA, DIE FREIE ENZYKLOPÄDIE: *Multiprotocol Label Switching*. http://de.wikipedia.org/wiki/MPLS, zugegriffen am 30.07.2007.

[Wike] WIKIPEDIA, DIE FREIE ENZYKLOPÄDIE: *OLE for Process Control*. http://de.wikipedia.org/wiki/OLE_for_Process_Control, zugegriffen am 30.07.2007.

[Wikf] WIKIPEDIA, DIE FREIE ENZYKLOPÄDIE: *OSI-Modell*. http://de.wikipedia.org/wiki/OSI-Modell, zugegriffen am 21.08.2007.

[Wikg] WIKIPEDIA, DIE FREIE ENZYKLOPÄDIE: *PROFINET*. http://de.wikipedia.org/wiki/Profinet, zugegriffen am 30.07.2007.

[Wikh] WIKIPEDIA, DIE FREIE ENZYKLOPÄDIE: *Resource Reservation Protocol*. http://de.wikipedia.org/wiki/Resource_Reservation_Protocol, zugegriffen am 31.07.2007.

[Wiki] WIKIPEDIA, DIE FREIE ENZYKLOPÄDIE: *Verfügbarkeit*. http://de.wikipedia.org/wiki/Verf%C3%BCgbarkeit, zugegriffen am 30.07.2007.

[Wikj] Wikipedia, The Free Encyclopedia: *Client-server*. `http://en.wikipedia.org/wiki/Client_server`, zugegriffen am 21.06.2007.

[WSG+03] Wang, N., D. C. Schmidt, A. Gokhale, C. D. Gill, B. Natarajan, C. Rodrigues, J. P. Loyall und R. E. Schantz: *Total Quality of Service Provisioning in Middleware and Applications*. The Journal of Microprocessors and Microsystems, 27(2):45–54, März 2003. `http://www.cs.wustl.edu/~nanbor/papers/elsevier.pdf`, zugegriffen am 01.06.2007.

[ZBS97] Zinky, J. A., D. E. Bakken und R. E. Schantz: *Architectural Support for Quality of Service for CORBA Objects*. Theory and Practice of Object Systems, 3(1):55–73, April 1997.

[Zen] Zen. `http://zen.ece.uci.edu/zen/`, zugegriffen am 19.06.2007.