FH Vorarlberg
University of Applied Sciences

Verwendung von ZeroMQ in verteilter Automatisierung

Masterarbeit
zur Erlangung des akademischen Grades
Master of Science in Engineering, MSc

Fachhochschule Vorarlberg
Master Mechatronik

Betreut von
Prof. (FH) DI Horatiu O. Pilsan

Vorgelegt von
David Pfefferkorn

Dornbirn, Januar 2022

Kurzreferat

IEC 61499 ist ein Standard für die verteilte Automatisierung. Damit lassen sich Systeme designen, die eine Vielzahl von Knoten vernetzen können. Dadurch lassen sich komplexe Aufgaben in der Industrie leichter bewältigen. Um dies zu ermöglichen, müssen alle Systembestandteile miteinander kommunizieren können. Ein solches Kommunikationsnetzwerk kann dann schnell sehr komplex werden und damit schwer zu verwalten sein. Wenn es darum geht, an viele Teilnehmer Daten zu verteilen, hat sich das Publish-Subscribe Kommunikationsmuster bewährt.
ZeroMQ ist eine Bibliothek, mit der über Sockets unter anderem solche Publish-Subscribe Kommunikationsnetzwerke realisiert werden können. Sie baut auf dem ZeroMQ Message Transport Protocol (ZMTP) auf. Dabei ist die Verwendung unwesentlich komplexer als mit Berkeley Sockets.
4DIAC/Forte bietet ein Framework um Applikationen nach IEC 61499 zu erstellen.
In dieser Thesis wird 4DIAC/Forte und ZeroMQ vorgestellt. Es wird die Struktur einer Kopplungsschicht zwischen Forte und ZeroMQ präsentiert und implementiert. Anschließend werden Messungen von Latenz und Throughput durchgeführt, um die Performance zu evaluieren.
In Forte sind traditionelle Sockets für die Kommunikation bereits implementiert. Sie unterstützen das Client-Server Muster und Publish-Subscribe mittels UDP Multicast. Mit diesen Lösungen werden die Messungen ebenfalls durchgeführt um einen Vergleich mit ZeroMQ aufzustellen. Es werden auch vorhandene Daten für DDS herangezogen. Die Ergebnisse zeigen, dass sich ZeroMQ für die Kommunikation in Forte eignet und dabei hilft, die Komplexität zu reduzieren und die Handhabung zu vereinfachen.

Abstract

IEC 61499 is a standard for distributed automation. It can be used to design systems that can consist of a large number of nodes. This makes it easier to handle complex tasks in an industrial setting. To make this possible, all system components must be able to communicate with each other. Such a communication network can quickly become very complex and thus difficult to manage. When it comes to distributing data to many participants, the publish-subscribe communication pattern has proven its worth.
ZeroMQ is a library that can be used to implement such publish-subscribe communication networks via sockets. It is based on the ZeroMQ Message Transport Protocol (ZMTP). The usage is not much more complex than with Berkeley Sockets.
4DIAC/Forte provides a framework to create applications according to IEC 61499.
This thesis introduces 4DIAC/Forte and ZeroMQ. The structure of a coupling layer between Forte and ZeroMQ is presented and implemented. Then, measurements of latency and throughput are performed to evaluate performance.
In Forte, traditional sockets are already implemented for communication. They support the client-server pattern and publish-subscribe using UDP multicast. With these solutions the measurements will also be performed to set up a comparison with ZeroMQ. Existing data for DDS is also used.
The results show that ZeroMQ is suitable for communication in Forte and helps to reduce complexity and simplify handling.

Inhaltsverzeichnis

Abbildungsverzeichnis

Alle Abbildungen ohne Quellenverweis sind selbst erstellt.

Tabellenverzeichnis

1 Einleitung

1.1 Motivation

Die Masterarbeit befasst sich mit der Kommunikation in einem verteilten Regelungs- und Automatisierungssystems. Im Englischen distributed control system (DCS) genannt, handelt es sich dabei um automatisierte Regelsysteme, bei denen die Regelelemente (Regler, Mikrocontroller, etc.) räumlich über eine Plant (z.B.: industrielle Produktion, Kraftwerk, Stromversorgung, etc.) verteilt sind.

Es unterscheidet sich von zentralisierten Regelungssystemen, bei denen eine zentrale Steuerungseinheit alle Regelaufgaben übernimmt. In DCS wird jedes Prozesselement (z.B.: Maschine oder Gruppe von Maschinen) von einem eigenen Regler gesteuert. Diese Regler sind nah an der Regelstrecke und übernehmen dort die Datenerhebung (Sensoren) und Regelung (Aktoren). Da sie in der Lage sein müssen, untereinander kommunizieren zu können werden sie über ein Netzwerk miteinander verbunden. Ein großer Vorteil solcher Systeme ist die Ausfallssicherheit. Wenn ein Controller ausfällt, funktioniert der Rest des Systems weiter.

IEC 61499 [13] ist ein offener Standard für verteilte Regelung und Automatisierung. Es erlaubt die Implementierung einer oder mehrerer Applikationen, die aus Funktionsblöcken zusammengebaut werden. Diese Blöcke werden für das gesamte System erstellt und anschließend an die Elemente (Geräte) des Systems verteilt. Diese Vorgehensweise erlaubt es, die gesamte Applikation als Gesamtes zu verwalten.

Diese Funktionsblöcke tauschen Nachrichten untereinander aus. Wenn dafür End-zu-End Verbindungen eingerichtet werden, kann so ein Netzwerk schnell sehr komplex werden. Um die Komplexität solcher verteilten Systeme zu verringern, können andere Kommunikationsmodelle wie Publish-Subscribe verwendet werden. Dabei ist einem Teilnehmer nicht bekannt, wer noch an der Kommunikation im Netz teilnimmt. Der Sender schickt eine nach Thema kategorisierte Nachricht ins Netz, und die Subscriber dieses Themas empfangen diese. Der Rest ignoriert es. Durch dieses Vorgehen kann das System ohne viel Aufwand beliebig erweitert werden.

Im Rahmen dieser Masterarbeit soll mit ZeroMQ [21] ein solches Netzwerk aufgebaut und getestet werden. Dabei handelt es sich um eine Open Source Library, die für den Nachrichtenaustausch in verteilten Systemen entwickelt wurde. Sie arbeitet ohne Message Broker und unterstützt unterschiedliche Messaging Patterns wie Publish-

Subscribe und Request-Reply. Die Library wird dort verwendet, wo Übertragungsgeschwindigkeiten eine große Rolle spielen und viele Teilnehmer direkt miteinander kommunizieren (Börse, Handelsplattformen, Echtzeitmesssysteme, Smart Grids) [15] [18].

1.2 Zielsetzung

Es soll untersucht werden, ob sich ZeroMQ als Kommunikationsbasis für IEC 61499 eignet. Dafür soll ein Konzept für die Kopplung erarbeitet und eine Kopplungsschicht implementiert werden. Der Nachrichtenaustausch erfolgt nach dem Publish-Subscribe Modell, das von ZeroMQ unterstützt wird.

Um das Verhalten und die Performance zu überprüfen, wird ein kleines System aufgesetzt. Die einzelnen Systemknoten werden mit RaspberryPis realisert. Auf diesen wird mit dem OpenSource Framework 4DIAC [4] IEC 61499 umgesetzt. Der Fokus liegt auf der Kommunikation. Es wird keine Regelung oder dergleichen mit dem System durchgeführt.

Das grundlegende Verhalten wird mit zwei Netzwerkteilnehmern an einem Ggigabit Ethernet Switch untersucht. Dabei werden leere Nachrichten ausgetauscht, die in ihrer Größe variieren. Verschiedene Netzwerkmetriken werden gemessen, um die Performance zu bewerten. Um ein Netz unter Last zu simulieren, wird der Versuch auf vier Netzteilnehmer ausgeweitet. Damit soll die Stabilität und Erweiterbarkeit evaluiert werden.

Die Ergebnisse der Performance Messung sollen mit den Ergebnissen für das Data Distribution System (DDS) aus [19] verglichen werden.

Die Forschungsfrage lässt sich folgendermaßen formulieren:

> "Wie muss die Kopplungsschicht gestaltet werden, damit ein verteiltes System (IEC 61499) auf Basis von ZeroMQ implementiert werden kann? Wie gut eignet sich ZeroMQ als Kommunikationsbasis für IEC 61499 im Vergleich zu DDS (Data Distribution System), wenn die Messwerte für Througput, Latenz und Round Trip Time zur Evaluierung herangezogen werden?"

1.3 Aufbau der Arbeit

Kapitel 2 beschreibt den technischen Hintergrund der relevanten Themen. Kapitel 3 zeigt das Konzept für die Kopplung und erklärt, wie die Kopplungsschicht implementiert wurde. Kapitel 4 beschreibt die Versuche. Kapitel 5 präsentiert die Ergebnisse und zieht einen Vergleich mit DDS. Kapitel 6 beinhaltet das abschließende Fazit.

2 Technischer Hintergrund

In diesem Kapitel werden die benötigten technischen Grundlagen beschrieben. Den Anfang macht der Standard IEC 61499. Das Framework 4DIAC zur Implementierung von IEC 61499 wird erläutert. Den Schluss macht die Bibliothek ZeroMQ, mit der die Middleware umgesetzt werden soll.

2.1 IEC 61499

2.1.1 Allgemein

Eine große Neuerung in der Automatisierung war die Einführung von Speicherprogrammierbarer Steuerungen (SPS, programmable logic controller, PLC). Die Programmierung dieser Einheiten wurde mit IEC 61131 standardisiert. Er beinhaltet die folgenden fünf Programmiersprachen [11]:

- Instruction List
- Ladder Diagram
- Function Block Diagram (FBD)
- Sequential Function Chart
- Structured Text

Dieser Standard wurde aber nicht für verteilte Systeme entwickelt. In [12] werden Funktionsbausteine definiert, die der Kommunikation zwischen SPS dienen. Dieses Designmodell zeigt aber eine Reihe von Problemen beim Entwickeln von verteilten Systemen auf. Zum einen lassen sich Applikationen nach IEC 61131-3 nicht auf unterschiedliche Ressourcen verteilen. Ein weiterer Nachteil ist, dass die Reihenfolge der Ausführung von Funktionsbausteinen nicht klar festgelegt ist und vom Entwickler nicht direkt beeinflusst werden kann.

IEC 61499 ist eine Erweiterung von IEC 61131 [24]. Die Unterschiede zwischen den Beiden sind das eingeführte Systemmodell, die geänderte Schnittestelle des Funktionsbausteins und die Execution Control Chart (ECC) als Grundbaustein des Softwaredesigns.

Der Standard erlaubt die komponentenbasierte Modellierung und Entwicklung von Automatisierungsanwendungen. Er stellt ein standardisiertes Modell und Designmethoden zum Beschreiben von Funktionsbausteinen in einem implementationsunabhängigen Format bereit. Damit soll erreicht werden, dass die Software folgende Eigenschaften hat.

- **Portabel**: Die Applikationen müssen zwischen Software Tools unterschiedlicher Hersteller austauschbar sein.
- **Konfigurierbar**: Das Verhalten der Applikation wird durch ein Netzwerk von Funktionsbausteinen konfiguriert.
- **Interoperabel**: Verschiedene Hardware kann zu einem System verbunden werden.

2.1.2 Funktionsbaustein

Funktionsbausteine (FB) sind der Grundbaustein jeder Applikation. Anwendungen werden durch FB-Netzwerke definiert. Die Bausteine ähneln derer in IEC 61131 [4] [24]. Ein großer Unterschied ist die Datenkapselung. Anders als im Vorgänger existieren keine globalen Variablen.

Des Weiteren wird zwischen Daten und Event Schnittstellen unterschieden. Das erlaubt die Umsetzung einer ereignisorientierten Designmethode. Die Abfolge, in der Funktionsbausteine ausgeführt werden, wird durch Events klar festgelegt. Abbildung 2.1 zeigt die Struktur eines Standardfunktionsbausteines.

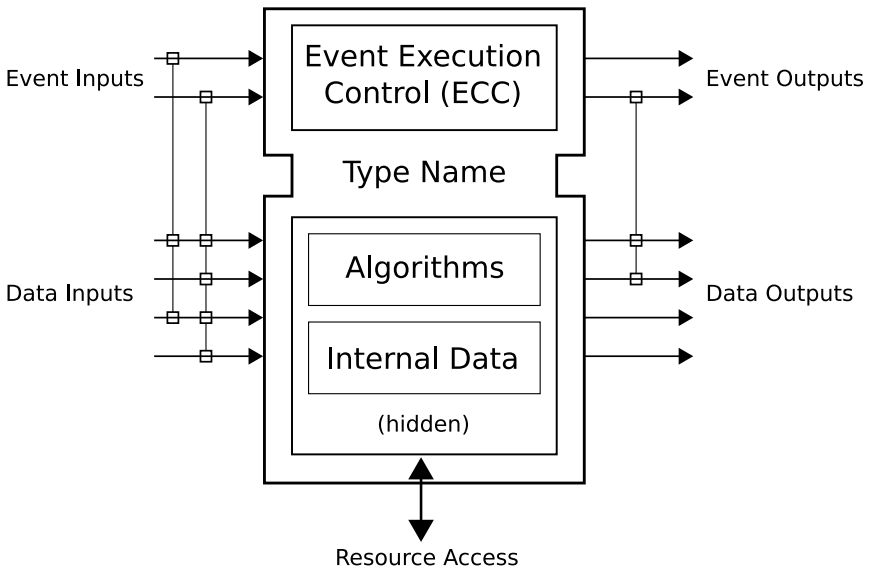

Abb. 2.1: Struktur Funktionsbaustein

Der FB besitzt Ein- und Ausgänge für Events und Daten. Diese können über eine sogenannte WITH Beziehung miteinander verbunden werden (vertikale Linien), damit das Event erst bei Anliegen der Daten abgehandelt wird.
Das Verhalten des Bausteines wird über Execution Control Charts (ECC) definiert, die einen Zustandsautomaten beschreiben.

Abbildung 2.2 zeigt ein einfaches Beispiel. Abhängig vom einkommenden Event löst dieser einen Algorithmus aus. Diese Algorithmen können in den von IEC 61131 spezifizierten Programmiersprachen oder in anderen, höheren Sprachen (Java, C++) realisiert sein. Nach der Ausführung wird ein Event am Ausgang erzeugt und die Datenausgänge aktualisiert.

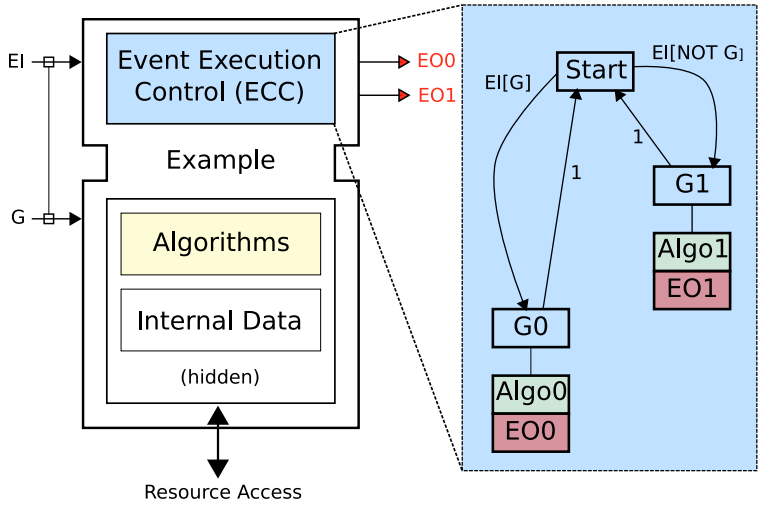

Abb. 2.2: Einfaches Beispiel für ECC

Es gibt drei Typen von FBs.

- **Standard Funktionsbaustein**: Abgebildet in Abbildung 2.1. Diese können nur auf einer Ressource laufen.
- **Zusammengesetzter Funktionsbaustein**: Besteht aus einem Netzwerk von Standard FBs, das das Verhalten des FB bestimmt.
- **Service Interface Funktionsbaustein**: SIFBs bieten eine Schnittstelle zu Ressourcen wie Hardware und Kommunikationsmedien. Der Standard legt keine Implementierung für SIFBs fest. Über Teil 4 von IEC 61499 (Regeln für normgerechte Profile/Compliance Rules) [10] werden die Verhaltensregeln und die Schnittstellen definiert, damit der Baustein oder die Applikation standardkonform sind. Die Service Interface Funktionsbausteine werden im Rahmen dieser Masterarbeit verwendet, um die Kommunikation zwischen den FBs mittels ZeroMQ zu implementieren. Daher wird auf diese in Unterabschnitt 2.1.4 genauer eingegangen.

2.1.3 Verteilte Applikation

Das Design eines verteilten Systems erfolgt mit IEC 61499 anwendungszentriert. Dabei wird eine Applikation für das Gesamtsystem erstellt, indem die Funktionsbausteine miteinander zu einem Netzwerk verbunden werden. Zu beachten ist, dass Event- und Datenleitungen nicht kompatibel sind.

Das Systemmodell dient der zentralen Verwaltung der Anwendungen, Ressourcen und Kommunikationsmöglichkeiten. Ein Beispiel ist in Abbildung 2.3 dargestellt.
Es besteht aus einem Anwendungs- und Gerätemodell. Im Anwendungsmodell finden sich die verschiedenen Anwendungen, die mittels FB Netzwerken realisiert sind. Über das Gerätemodell können Hardware, Ressourcen und Kommunikationsschnittstellen verwaltet werden. Die verschiedenen Anwendungen sind auf die verfügbaren Ressourcen verteilt. Die Funktionsbausteine eines Netzwerkes können sich dabei auf unterschiedlicher Hardware befinden.

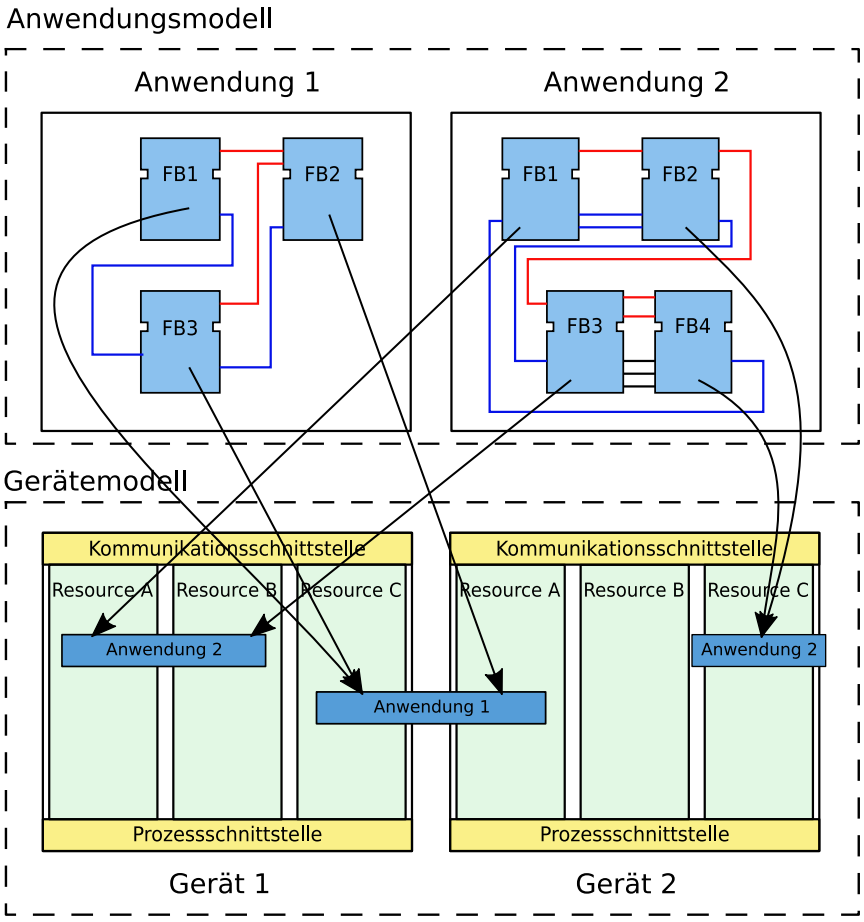

Abb. 2.3: Beispiel für ein Systemmodell

Im Beispiel befindet sich FB1 und FB3 der Anwendung 1 auf Gerät 1. FB 2 ist auf Gerät 2. Damit diese Daten und Events austauschen können, werden Kommunikations-SIFBs benötigt. Der Datenaustausch erfolgt dann über die Kommunikationsschnittstelle. Siehe Abbildung 2.4 für ein Beispiel.

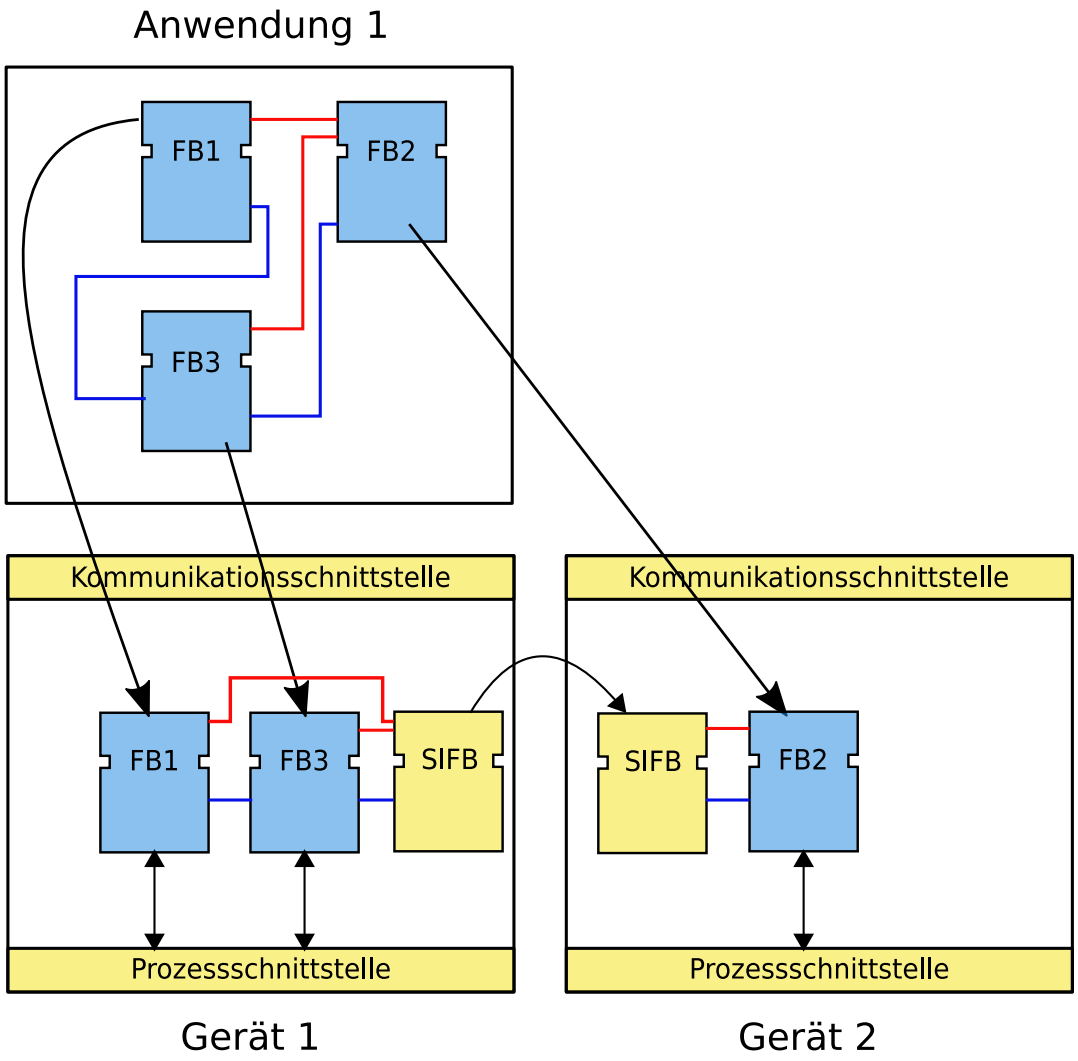

Abb. 2.4: Beispiel Kommunikations-SIFB

2.1.4 Service Interface Funktionsbaustein

Mit Service Interface Funktionsbausteine lassen sich Funktionalitäten implementieren, die nicht im Rahmen von IEC 61499 modellierbar sind. Bei der Programmiersprache für die Implementierung hat man quasi freie Wahl.
Üblicherweise stellt ein SIFB eine Schnittstelle nach Außen dar. Der Zugriff auf Hardware und die Kommunikationsschnittstelle wird so ermöglicht.

Im Standard wird die Umsetzung eines solchen Bausteines nicht vorgegeben. Es werden nur die Schnittstellen definiert, über die ein solcher Baustein verfügen sollte. Abbildung 2.5 zeigt das Template für einen Service Interface Funktionsbaustein.

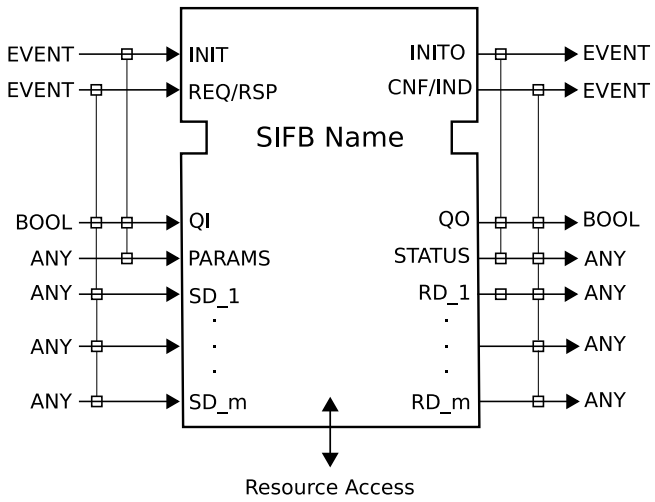

Abb. 2.5: SIFB Template

- **Eventeingänge**
  - INIT: Startet den Service, den der Baustein bereitstellt. Überlicherweise ist das Event mit Initialisieriungsparametern gekoppelt.
  - REQ (Request): Initiiert die Anfrage von Daten an eine externe Ressource.
  - RSP (Response): Initiiert die Rückmeldung auf eine Anfrage an eine externe Ressource.
- **Eventausgänge**
  - INITO: Wird als Ergebnis des INIT Eingangsevents ausgelöst. Teilt mit, dass die Initialisierung abgeschlossen ist. Sagt nichts über den Erfolg des Prozesses aus.
  - CNF (Confirmation): Wird ausgelöst, wenn die Übermittlung einer Anfrage an eine externe Ressource beendet ist.

  - IND (Indication): Wird ausgelöst, wenn eine Rückmeldung einer externen Ressource erhalten wurde. Bei Netzwerkkommunikation zum Beispiel wird dieses Event gesetzt, wenn die Daten empfangen wurden und diese an den Datenausgängen liegen.

- **Dateneingänge**
  - QI [BOOL]: Kann als Parameter für einen Eventeingang verwendet werden, um zusätzliche Informationen mitzugeben. Zum Beispiel kann mit TRUE bei INIT definiert sein, dass der Service gestartet werden soll und mit FALSE beendet.
  - PARAMS [BELIEBIG]: Dabei handelt es sich um Parameter, die den Dienst des Funktionsblockes definiert. Zum Beispiel kann das die Adresse eines anderen Gerätes sein, das über ein Netzwerk erreicht werden soll.
  - SD_1, ... SD_m [BELIEBIG]: Die Eingänge, über die Daten erhalten werden. Ein Beispiel sind Positionen, die ein Aktuator einnehmen soll.

- **Datenausgänge**
  - QO [BOOL]: Zeigt an, ob der Service, der zu einem Event gehört, erfolgreich ausgeführt wurde.
  - STATUS [BELIEBIG]: Stellt genauere Informationen zum Status des letzten ausgeführten Events bereit.
  - RD_1, ... RD_n [BELIEBIG]: Die Ausgänge, über die der Funktionsblock Daten bereitstellen kann.

Im folgenden Abschnitt werden drei Typen von SIFBs vorgestellt, wie sie häufig vorkommen.

Generischer Service Interface Funktionsblock

Abbildung 2.5 zeigt einen generischer SIFB. Diese dienen als Template für die verschiedenen Services.
Sie werden in zwei Typen aufgeteilt. Entweder wird der SIFB durch ein Event aufgefordert, externe Daten (z.B.: von der Hardware) anzufordern. Hierfür ist der Eventeingang REQ und Eventausgang CNF zu verwenden.

Beim zweiten Typ wird das Event durch eine externe Ressource ausgelöst. Der Eventeingang RSP und Eventausgang IND sind für diesen Fall vorgesehen.

Kommunikationsfunktionsblock

IEC 61499 beinhaltet zwei verschiedene Modelle, die die Kommunikation zwischen Teilen der Applikation auf verschiedenen Ressourcen regeln soll.

Abbildung 2.6 zeigt das Modell für den unidirektionalen Datenaustausch. Ein Publish Block kann dabei an mehrere Subscribe Blöcke senden. Ein großer Vorteil ist die Entkopplung. Das Hinzufügen oder Entfernen von SUB-SIFBs beeinflusst den Rest des Netzwerkes nicht. Der Parametereingang PARAMS des Publish Blockes legt fest, welche Art von Nachrichten von SUB Blöcken verarbeitet werden können. Diese Parameter sind abhängig vom Protokoll oder der Library, die für die Implementierung der Kommunikationsschicht genutzt wird.

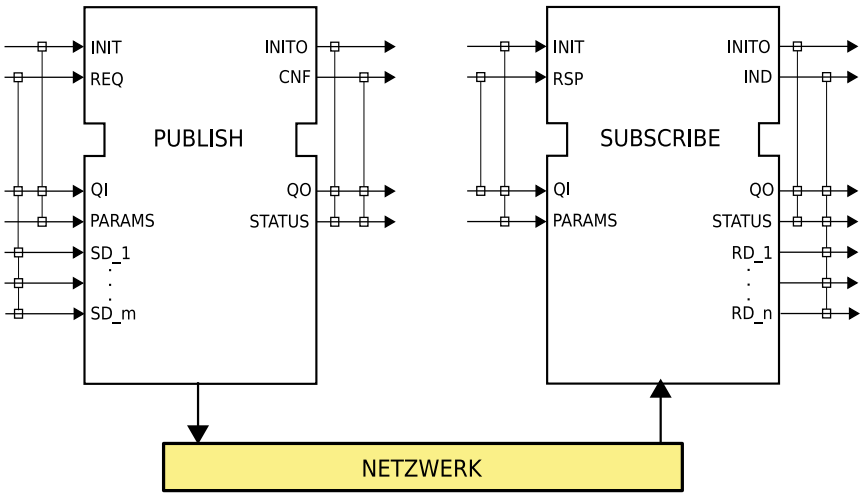

Abb. 2.6: SIFB für die Kommunikation

Abbildung 2.7 zeigt das Client Server Modell für die bidirektionale Kommunikation. Dabei kann zwischen zwei Ressourcen Daten ausgetauscht werden. Der Client muss sich beim Server registrieren.

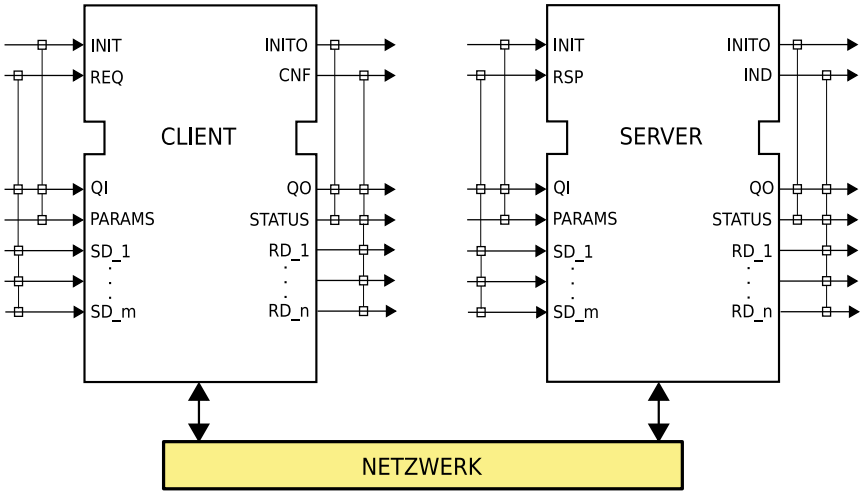

Abb. 2.7: Client Server Modell für Kommunikation

Management Funktionsblock

Über die in Abbildung 2.8 dargestellten Management Funktionsblöcke lässt sich die Applikation verwalten. Zum Beispiel wird so die Erzeugung und Ausführung von Funktionsblöcken in den Ressourcen gesteuert.

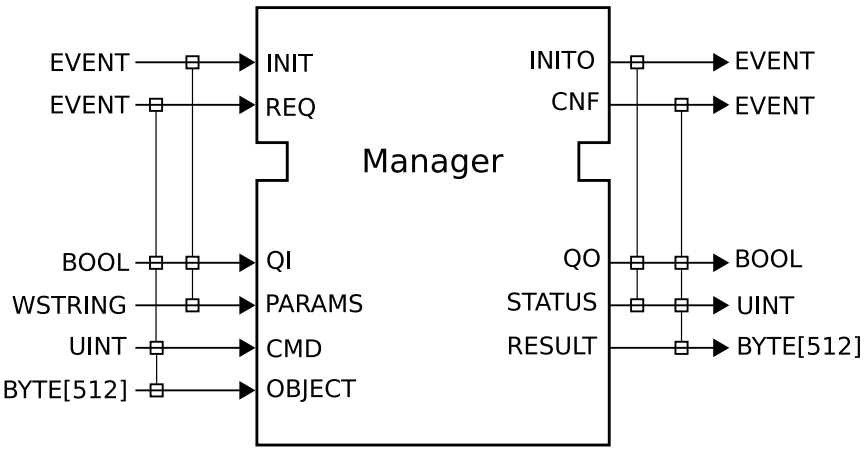

Abb. 2.8: Management Funktionsblock

2.1.5 Werkzeuge

Für die Entwicklung von Applikationen nach dem Standard IEC 61499 exisitieren unterschiedliche Tools. Zu den kommerziell erhältlichen Werkzeugen zählen ISaGRAF [14], NxtOne [16] und FBbuilder [20]. FBench [6] und 4DIAC [2] werden als OpenSource angeboten. Im Rahmen dieser Masterarbeit wird 4DIAC genutzt. Ausschlaggebend waren die umfassende Dokumentation, die aktive Community und die OpenSource Lizenz.

2.2 4DIAC

2.2.1 Einführung

4DIAC ist eine Open Source Entwicklungsumgebung für verteilte Automatisierungs- und Steuerungssysteme. Mit ihr lassen sich mittels Funktionsbausteinen verteilte Anwendungen erstellen. Dabei wird das Design und die Ausführung getrennt. In Abbildung 2.9 wird der Aufbau des Frameworks dargestellt.
Das blaue Rechteck in der oberen Ebene stellt die Werkzeuge dar, mit denen Funktionsbausteine erstellt und verbunden werden können.
Die Rechtecke der unteren Ebene repräsentieren die Hardware (z.B.: SPS, Raspberry Pi, etc.), auf denen die Applikation ausgeführt werden soll. Auf jedem dieser Geräte läuft eine Laufzeitumgebung (Runtime Environment- RTE), auf der das Programm ausgeführt wird. Entsprechend der beschriebenen Teilung lässt sich 4DIAC in zwei Komponenten trennen.

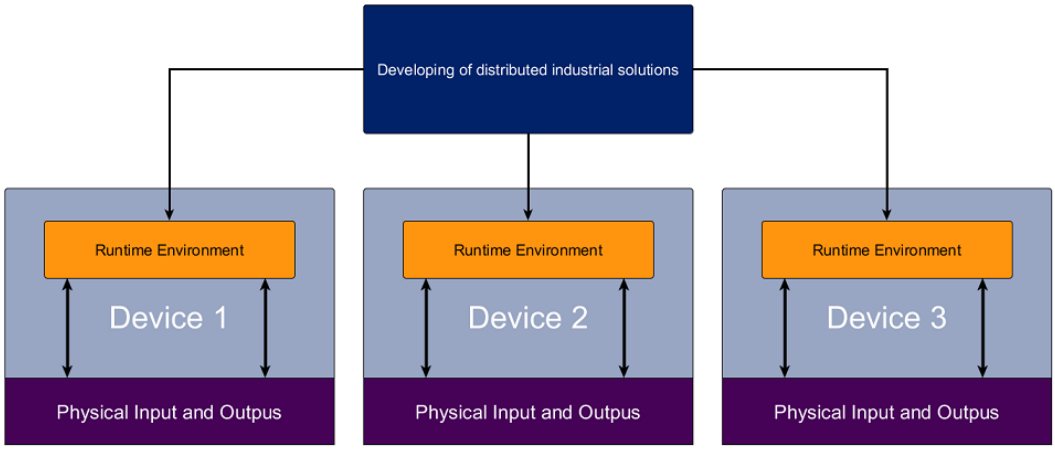

Abb. 2.9: 4DIAC Übersicht [4]

2.2.2 Aufbau

FORTE wird die Laufzeitumgebung genannt, die auf ein Betriebssystem aufgesetzt wird. Sie ist in C++ implementiert, damit sie auf möglichst vielen Plattformen läuft. Dazu zählen unter anderem Windows, Linux, FreeRTOS und vxWorks. Durch den geringen Speicherverbrauch werden auch kleine Embedded Geräte unterstützt.

Die IDE fasst die Tools zusammen, die für das Design notwending sind. Abbildung 2.10 zeigt die Benutzeroberfläche. Sie basiert auf dem Eclipse Framework.
Hier lassen sich die Funktionsbausteine und FB Netzwerke designen. Es bietet auch eine Oberlfäche zur Konfiguration der Hardware an. Ein fertiges Programm lässt sich direkt über die IDE auf ein Gerät spielen, auf der FORTE oder eine andere RTE aufgesetzt ist.

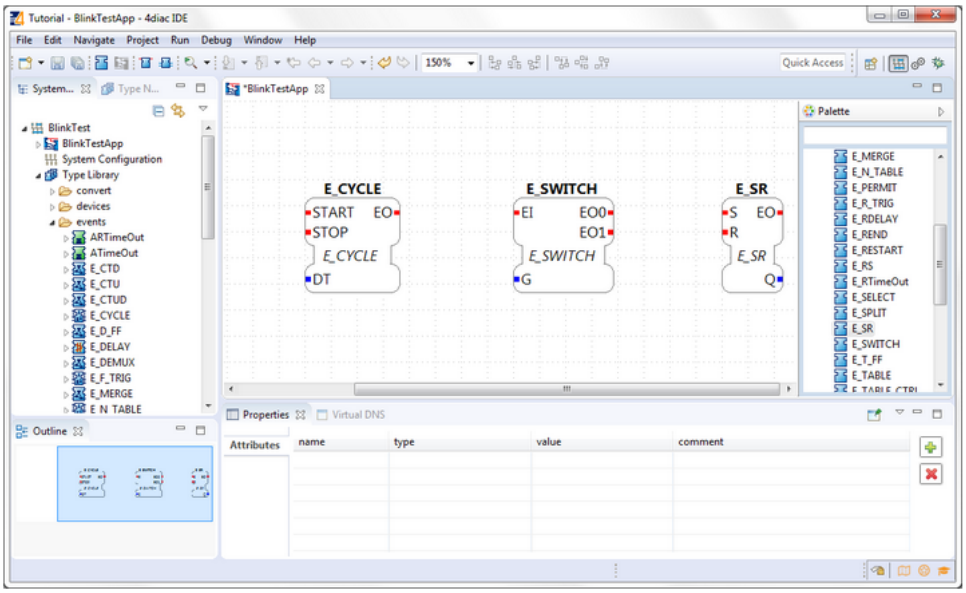

Abb. 2.10: Screenshot 4DIAC-IDE

Die Entkopplung der Laufzeit- und Entwicklungsumgebung bringt eine Besonderheit mit sich. Wenn in der IDE neue Funktionsbausteine modelliert werden, sind diese zuerst nicht auf FORTE lauffähig. Über die IDE lassen sich die FBs als FORTE Code (C++ Dateien) exportieren. Anschließend wird FORTE neu kompiliert. Der Codeexport funktioniert nur für Standard und zusammengesetzte Funktionsbausteine. Für SIFBs werden die Interfaces definiert. Die Funktion wird "manuell" in C++ implementiert.

4DIAC unterstützt das "IEC 61499 compliance profile for feasibility demonstrantions", definiert von HOLOBLOC Inc. [9].

2.3 ZeroMQ

2.3.1 Einleitung

ZeroMQ ist eine Bibliothek, die den asynchronen Austausch von Nachrichten über ein Netzwerk ermöglicht [21]. Sie wurde für verteilte und nebenläufige Systeme (concurrent systems) entwickelt [15].

Die Nachrichten werden in Form binärer Daten oder Strings über Sockets verschickt. Dabei können verschiedene Methoden für Unicast und Multicast verwendet werden. Anders als bei vielen Message Oriented Middleware Lösungen existiert für ZMQ kein Message Broker. Die Daten werden direkt zwischen den Applikationen ausgetauscht. ZeroMQ kann auch für den lokalen Datenaustausch zwischen verschiedenen Prozessen oder Threads verwendet werden.

Die Bibliothek ist plattformunabhängig und unterstützt die meisten gängigen Betriebssysteme wie MacOS, Windows und Linux. Durch eine Vielzahl von APIs werden verschiedene Programmiersprachen wie C, Java und Python unterstützt. ZeroMQ wird von iMatrix entwickelt und ist LGPLv3 Open Source.

2.3.2 Socket-API

Die ZeroMQ Socket API verbirgt einen Großteil der Komplexität, die bei traditioneller Socket Programmierung auftritt. Sie übernimmt das Routing, Framing, Auf- und Abbau der Verbindung und das Wiederherstellen nach Verbindungsabbruch. Die API kann wie bei traditionellen Berkeley Sockets (BSD) verwendet werden:

- Der Socket muss zuerst erzeugt und später wieder zerstört werden. (`zmg_socket()`, `zmg_close()`)
- Der Socket kann über das setzen von Optionen konfiguriert werden. (`zmg_setsockopt()`, `zmg_getsockopt()`)
- Um den Socket im Netzwerk zu integrieren, muss er an einen Knoten gebunden und verbunden werden (`zmg_bind()`, `zmg_connect()`). Es existiert kein `accept`() Befehl. Sobald ein Socket gebunden wird, werden alle eingehenden Verbindungen akzeptiert.
- Anschließend können die Daten über Nachrichten versendet und erhalten werden. (`zmg_send()`, `zmg_recv()`)

Um einen Socket zu erstellen, muss zuerst ein ZeroMQ Context erzeugt werden. Dabei handelt es sich um einen Container der alle ZeroMQ Sockets innerhalb eines Prozesses beinhaltet. Das entspricht einer ZeroMQ Instanz.

Um anschließend eine Verbindung zu erzeugen, wird ein Knoten gebunden (`zmq_bind()`) und ein anderer verbunden (`zmq_connect()`). Anders als bei Berkeley Sockets ist nicht vorgegeben, welche Seite `zmq_bind()` oder `zmq_connect()` aufruft. Bei den Knoten kann es sich um Netzwerkknoten, Prozesse oder Threads handeln.
Verbindungen in ZeroMQ Anwendungen sind dynamisch. ZeroMQ Sockets stellen automatisch eine unterbrochene Verbindung wieder her. Knoten können zu jeder Zeit kommen und gehen. Die Reihenfolge, in der Komponenten gestartet werden ist beliebig. Anders als mit Berkeley Sockets ist es zum Beispiel möglich, ein `zmq_connect()` vor dem `zmq_bind()` aufzurufen.

2.3.3 Nachrichten senden und empfangen

Die Kommunikation zwischen zwei Endpunkten wird über das ZMTP (ZeroMQ Message Transport Protocol) geregelt [23]. Die Daten sind auf der Leitung binär kodiert. Die Serialisierung muss implementiert werden. Das verwendete Format ist dabei frei wählbar, zum Beispiel kann JSON, XML oder MessagePack verwendet werden.
ZeroMQ ist Nachrichten-orientiert. Wenn eine 300kb Nachricht gesendet wird, kommt auch eine 300kb Nachricht an. Dafür muss kein Framing oder Buffering implementiert werden.

Um diskrete Messages senden und empfangen zu können, verwendet ZMTP ein einfaches Framing. Ein Frame besteht aus einem Body und einem Header. Der Header beinhaltet die Datenmenge. Abbildung 2.11 zeigt einen Frame anhand eines Strings.

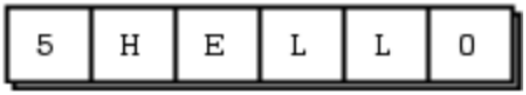

Abb. 2.11: ZeroMQ Nachricht auf der Leitung [8]

ZeroMQ stellt zwei verschiedene APIs für den Nachrichtenaustausch bereit.

Die erste und einfacher zu verwendende Möglichkeit wurde bereits in Unterabschnitt 2.3.2 vorgestellt. `zmq_recv()` eignet sich nicht für das Empfangen von Nachrichten die in der Größe varrieren. Die maximale Länge der Nachricht wird durch die festgelegte Buffergröße beschränkt.
Die zweite API bietet mehr Möglichkeiten, ist in der Handhabung aber komplexer. Im Speicher werden die Nachrichten als `zmq_msg_t` Strukturen abgeleget. Abhängig von der verwendeten Programmiersprachen sind auch Klassen möglich.
Die Daten werden als `zmq_msg_t` Objekte versendet und empfangen. Die Methoden der API sind unten aufgelistet.

ZeroMQ unterstützt mit `zmq_msg_t` Strukturen auch Multipart Messages. Dabei werden mehrere Frames zu einer Nachricht zusammengefasst, die empfangen und gesendet wird.

- Die Nachricht initialisieren. (`zmg_msg_init()`, `zmg_msg_init_size()`, `zmg_msg_init_data()`)
- Nachricht empfangen und senden. (`zmg_msg_send()`, `zmg_msg_recv()`)
- Nachricht freigeben. (`zmg_msg_close()`)
- Auf die Daten in der Struktur/Nachricht zugreifen. (`zmg_msg_data()`, `zmg_msg_size()`, `zmg_msg_more()`)
- Die Eigenschaften der Nachricht setzen und auslesen. (`zmg_msg_set()`, `zmg_msg_get()`)
- Mit der Nachricht arbeiten. (`zmg_msg_copy()`, `zmg_msg_move()`)

2.3.4 Asynchrone Kommunikation

Die ZeroMQ Bibliothek unterstützt standardmäßig vier Transport Protocols, welche über den Prefix eines einfachen Connection Strings ausgewählt werden:

1. **TCP** (*tcp://hostname:port*): Dient der Kommunikation in einem Netzwerk. Das ZeroMQ TCP wird disconnected TCP genannt. Der Endpunkt muss nicht vor dem Verbindungsaufbau existieren. Clients und Server können zu jeder Zeit die Verbindung aufbauen und trennen.
2. **INPROC** (*inproc://name*): Dient der Kommunikation zwischen Threads innerhalb des selben Prozesses (inter-thread) Ist deutlich schneller als *tcp* und *ipc*.
3. **IPC** (*ipc://tmp/filename*): Dient der Kommunikation zwischen Prozessen auf dem selben Host (inter-process). Ist wie ZeroMQ TCP verbindungslos.
4. **(E)PGM** (*(e)pgm://interface:address:port*): Erlaubt Multicast Transport über ein Netzwerk. Es werden zwei Varianten unterstützt. Bei PGM wird direkt mit IP Datagrammen gearbeitet. Bei EPGM werden die PGM Datagramme in UDP Datagramme eingebettet.

Die Socket-API ist unabhängig vom gewählten Protokoll und ist immer gleich zu verwenden. Die Applikation muss damit bei einem Wechsel der Kommunikation nicht angepasst werden.

Es ist möglich, dass ein Socket mehrere ein- und ausgehende Verbindungen akzeptiert. Dafür ist eine beliebige Kombination von Adressen und Protokollen erlaubt.
Der Code in 2.1 zeigt ein Beispiel. `socket` akzeptiert alle Netzwerkverbindungen auf Port 5000 und Verbindungen auf die Adresse 127.0.0.1 mit Port 1234. Über inproc kommen Daten vom Thread `worker_thread`.

```
zmq_bind (socket, "tcp://127.0.0.1:1234");
zmq_bind (socket, "tcp://*:5000");
zmq_bind (socket, "inproc://worker_thread");
```

Listing 2.1: Socket mit mehreren Verbindungen

Die Kommunikation in einer ZeroMQ Applikation läuft asynchron ab. Wenn ein Context erzeugt wird startet im Hintergrund ein I/O Thread automatisch, der das Empfangen und Senden übernimmt. Die Applikation wird nicht blockiert. Für leistungsintensive Anwendungen können zusätzliche Threads gestartet werden.

Der Thread verwaltet Warteschlangen für ein- und ausgehende Nachrichten. Zum Beispiel stellt ein Aufruf von `zmg_msg_send()` die Nachricht in die Warteschlange, statt sie direkt zu senden.

Über High Water Mark (HWM) wird die Anzahl Nachrichten in der Queue festgelegt. Bei Überschreiten dieser Marke wird die Anwendung blockiert oder Nachrichten verworfen.

2.3.5 Messaging Patterns

In einem verteilten System werden Teile eines Systems oder einer Applikation miteinander verbunden, damit eine Kommunikation stattfinden kann. Messaging Patterns beschreiben die Topologie und den Kommunikationsfluss.

ZeroMQ beinhaltet verschiedene Sockettypen. Durch Kombination passender Typen lassen sich Messaging Patterns umsetzen. Nicht alle Typen sind kompatibel.
In der Core-Library finden sich vier solcher Patterns [22], die in den folgenden Abschnitten näher erläutert werden.

Request-Reply

Das Request-Reply Pattern kann als einfaches Client-Server Modell betrachtet werden. Ein Client sendet Anfragen und der Server antwortet.
Die Kommunikation erfolgt synchron. Der Server und Client blockieren jeweils ihren Prozess, bis eine Anfrage bzw. Antwort empfangen wird. Die Knoten müssen explizit verbunden werden. Dadurch ergibt sich eine starke Kopplung. Fällt ein Teilnehmer aus, wird das gesamte System beeinflusst. Dadurch gelten hohe Anforderungen an die Robustheit (Fault Tolerance).

In ZeroMQ wird das Request-Reply Pattern über REQ und REP Socketpaare realisiert, siehe Abbildung 2.12. Dabei kann ein REQ_Socket zu mehreren REP_Sockets verbinden. In diesem Fall werden die Anfragen auf beide Server gleichmäßig verteilt (round-robined).
Mit ROUTER und DEALER Sockets lässt sich ein asynchrones Request-Reply Pattern umsetzen [22].

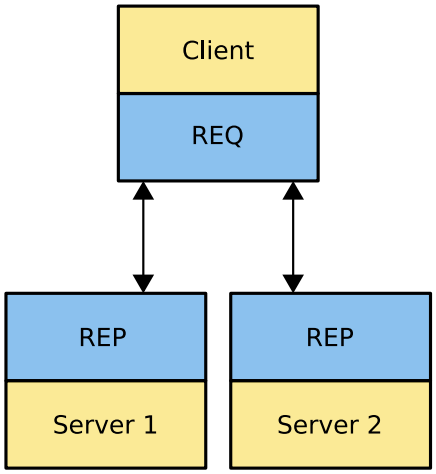

Abb. 2.12: Request-Reply Pattern

Pub-Sub

Mit dem Pub-Sub Pattern lässt ich ein Publish-Subscribe System umsetzen, siehe Abbildung 2.13. Die Knoten kommen über Topics zusammen, die sie interessieren.

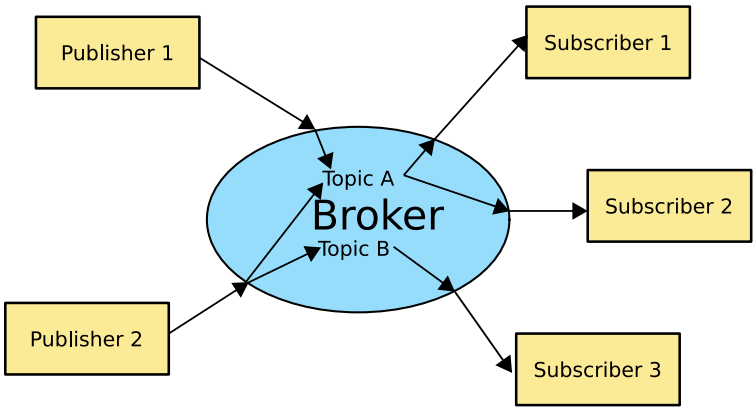

Abb. 2.13: Allgemeines Publish-Subscribe Paradigma

Publisher erzeugen Daten und Events. Diese veröffentlichen sie an Topics. Subscriber abonnieren diese. Jede Nachricht, die an ein Topic veröffentlicht wurde, wird sofort von allen Subscribern dieses Topics empfangen.
Die Kommunikation kann direkt oder über einen zentralen Knoten, den sogenannten Broker, erfolgen. An diesen wird veröffentlicht und abonniert. Er übernimmt damit das Filtern der Publisher Daten und das Weiterleiten an interessierte Subscriber. Die Knoten verbinden nur an den Broker und brauchen damit keine Kentniss voneinander.

Die Verbindung zwischen Publisher und Subscriber kann auch direkt aufgebaut werden. Publisher veröffentlichen die Daten direkt an alle Subscriber, die abonniert haben. Jeder Knoten muss Kenntnis über alle anderen Teilnehmer haben, um die Verbindung aufbauen zu können. Dieses Modell ist aufwendiger als das Broker Modell, wo dieser den Verbindungsaufbau übernimmt. Dadurch sind komplexe Mechanismen nötig, damit sich die Knoten gegenseitig finden.

Das Publish-Subscribe Modell erlaubt die asynchrone Übertragung von Nachrichten an verschiedene Teile eines Systems. Die Knoten sind lose gekoppelt. Teilnehmer können jederzeit gehen/kommen oder ausfallen, ohne den Rest des Systems zu beeinflussen.

Über PUB und SUB Socketpaare wird das Pub-Sub Pattern in ZeroMQ realisiert (Abbildung 2.14). Auf Publisherseite wird ein PUB_Socket gebunden. Auf Subscriberseite wird mit einem SUB_Socket zum Publisher verbunden.

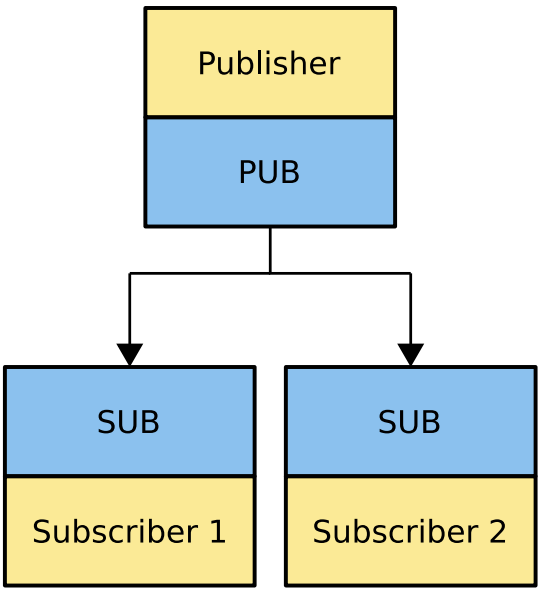

Abb. 2.14: Publish-Subscribe Pattern

Topics werden als Subscriptions bezeichnet. Der Subscriber setzt eine oder mehrere Subscription, um nur ausgewählte Nachrichten zu erhalten. Subscriptions werden als Byte Array ausgedrückt. Der erste Frame einer eingehende Nachricht wird bitweise mit diesem Array verglichen. Nur bei Übereinstimmung wird gesendet oder empfangen. Abhängig vom gewählten Transport Protokoll werden die Nachrichten auf Publisher- (tcp, ipc) oder Subscriberseite ((e)pgm) gefiltert. Für tcp und ipc werden Subscriptions vom Subscriber an den Publisher weitergeleitet.

Jeder Knoten besitzt eine eigene Queue. Diese existieren solange die Verbindung nicht von der Gegenseite getrennt wird. Neue Nachrichten werden verworfen, wenn die Warteschlange voll ist.
Die Kommunikation verläuft in nur eine Richtung. Nachrichten können nur vom PUB zum SUB Socket gesendet werden. Aufruf von `zmq_send()` mit einem SUB_Socket führt zum Beispiel zu einem Fehler.

Ein Subscriber kann ebenfalls mehrere Publisher verbinden. Dafür wird jedes mal ein `zmq_connect()` Aufruf benötigt. Alternativ kann `zmq_bind()` auf Empfängerseite aufgerufen werden, damit mehrere Sender darauf verbinden können. Im letzteren Fall verhält er sich wie ein PUB_Socket, der empfangen kann.

Anders als zum Beispiel DDS [19] [7] gibt es bei ZeroMQ keinen Message Broker. Es müssen PUB und SUB_Sockets explizit verbunden oder komplexe Verfahren zur gegenseitigen Entdeckung verwendet werden. Das macht die Implementierung eines großen Systems umständlich. In [8] wird vom Dynamic Discovery Problem gesprochen. Durch Einführen eines Proxy wird der anonyme Datenaustausch ermöglicht.

Abbildung 2.15 zeigt einen Proxy. Die Subscriber und Publisher haben jeweils einen fixen Knoten, zu dem sie sich verbinden können.

Für die Umsetzung des Proxy werden XPUB/XSUB Sockets verwendet. Das ist nötig, da ein PUB_Socket die Subscriptions nicht an einen SUB_Socket übermitteln kann. XPUB/XSUB können jeweils Nachrichten empfangen und senden. Sie haben für jede Verbindung zwei Warteschlangen, jeweils für Ein- und Ausgang. Somit kann der Proxy Subscriptions von der Subscriberseite auf die Publisherseite weiterleiten.

Der XPUB sendet eine spezielle Subscription Message an den XSUB. Dieser leitet an die verbundenen PUB_Sockets die Subscriptions weiter. Am XSUB kommen jetzt nur noch die von der Subscriberseite abonnierten Nachrichten an.

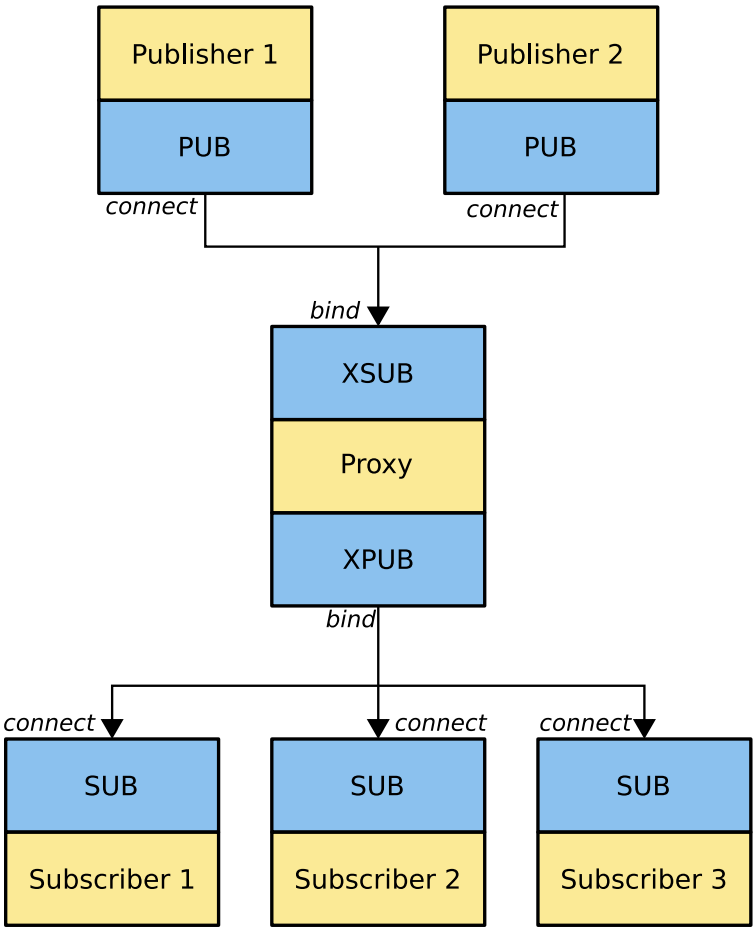

Abb. 2.15: Publish-Subscribe Pattern mit Proxy

Pipeline

Das Pipeline Patter wird mit PUSH_Sockets und PULL_Sockets realisiert. Damit lassen sich Nachrichten an mehrere Endpunkte versenden. Die einzelnen Knoten sind entlang einer Pipeline angeordnet. Ein PUSH_Socket verteilt seine Nachrichten gleichmäßig an verbundene PULL_Sockets. Das wird´ zum Beispiel für parallele Datenverarbeitung verwendet. Abbildung 2.16 zeigt ein Beispiel.

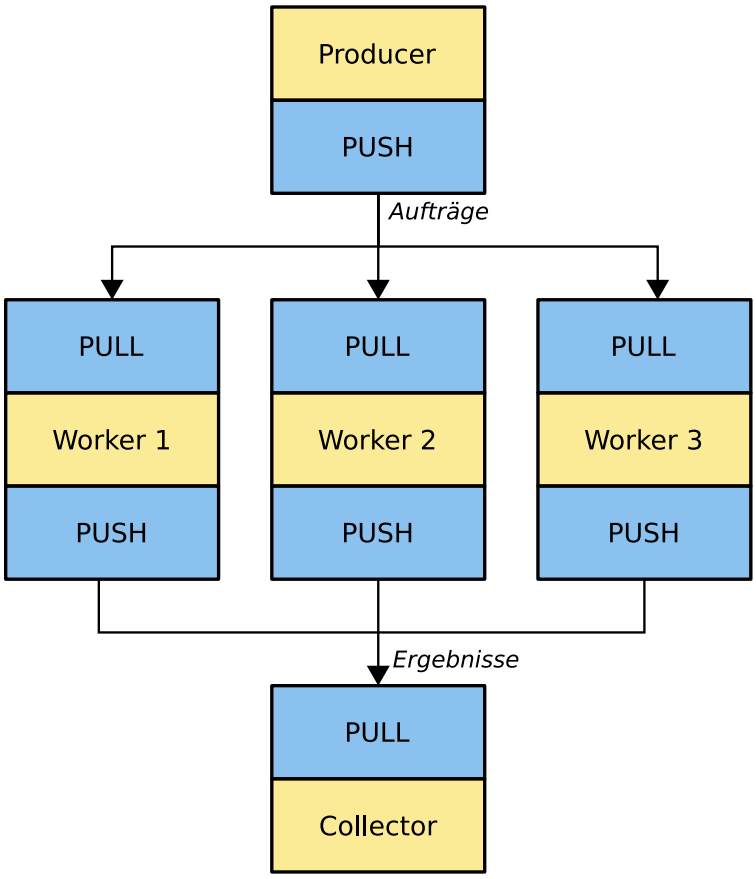

Abb. 2.16: Parallele Datenverarbeitung mittels Pipline (Push-Pull) Pattern

Exclusive Pair

Dieses Pattern wird mit PAIR_Sockets implementiert. Zwei Knoten werden exklusiv zueinander verbunden. Die Kommunikation ist bidirektional. Es ist immer nur eine Verbindung möglich.
Diese Sockets sind nicht für ein Netzwerk gedacht, sondern für die Kommunikation zwischen zwei Threads mittels `inproc` Transport Protocol, siehe Unterabschnitt 2.3.4.

3 Kopplungsschicht zwischen 4DIAC und ZeroMQ

Um zu zeigen, dass ZeroMQ für die Kommunikation in einer IEC61499 Anwendung geeignet ist, soll mittels der Bibliothek eine Kopplungsschicht in das 4DIAC/Forte Framework implementiert werden.
Im folgenden Kapitel wird das Interface des Service Interface Funktionsblockes definiert. Weiters wird das Einbinden von ZeroMQ in Forte beschrieben.

3.1 Kommunikationsarchitektur in Eclipse 4DIAC/Forte

In 4DIAC/Forte wird das Netzwerkinterface als Netzwerkstack implementiert. Jeder Kommunikationsfunktionsblock in einer 4DIAC Applikation nutzt einen eigenen Stack, um über das Netzwerk zu kommunizieren [3]. Ein solcher Stack wird aus mehreren Schichten aufgebaut, ähnlich dem OSI-Referenzmodell. Nur der Kommunikations-SIFB interagiert mit der IEC61499 Anwendung.

Abbildung 3.1 zeigt eine Übersicht über das Modell des Interfaces in Forte. Es werden dabei drei Rollen unterschieden. Die oberste Netzwerkschicht greift direkt auf die Datenein- und Ausgänge des Funktionsblockes zu. Die mittlere Netzwerkschicht übernimmt optionale Aufgaben wie Encoding und Komprimierung. Die unterste Schicht greift direkt auf das Netzwerk zu. Dabei soll der Datenaustausch asynchron erfolgen, damit der Funktionsblock nicht blockiert. Die Anzahl der Schichten ist dabei nicht begrenzt oder vorgegeben.

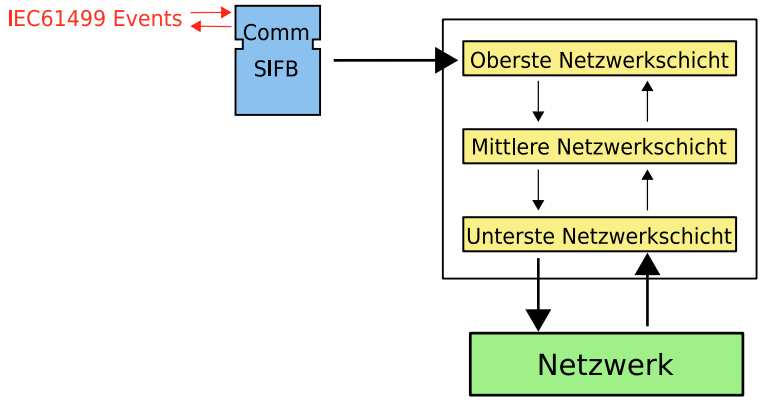

Abb. 3.1: Übersicht Netzwerkschichten

Der SIFB wird in Forte mit der Klasse CCommFB umgesetzt. Jeder der Netzwerkschichten implementiert die Standardmethoden, die von der Klasse CComLayer vorgegeben werden. Diese sind in der folgenden Aufzählung aufgelistet.

- **openConnection**: Wird vom Funktionsblock aufgerufen, um eine Verbindung zu öffnen.
- **closeConnection**: Wird vom Funktionsblock aufgerufen, um eine Verbindung zu schließen.
- **sendData**: Über diese Methode werden die Daten von einer Schicht zur nächsten weitergeleitet, bis sie von der untersten an das Netzwerk geht.
- **recvData**: Wird aufgerufen, wenn eine Nachricht über das Netzwerk ankommt.
- **processInterrupt**: Wird aufgerufen, wenn eine der Schichten Daten verarbeiten muss.

3.2 ZeroMQ Netzwerkschicht

Das in Abbildung 3.1 dargestellte Modell wird mit ZeroMQ Version 4.3.5 umgesetzt. Die Implementierung erfolgt über C++. Dafür wird die Bibliothek libzmq über den cppzmq Header [1] eingebunden.

Abbildung 3.2 zeigt den Netzwerkstack, wie er mit ZeroMQ umgesetzt wird. Es sind zwei Netzwerkschichten implementiert, MSGZMQComLayer und ZMQComLayer.

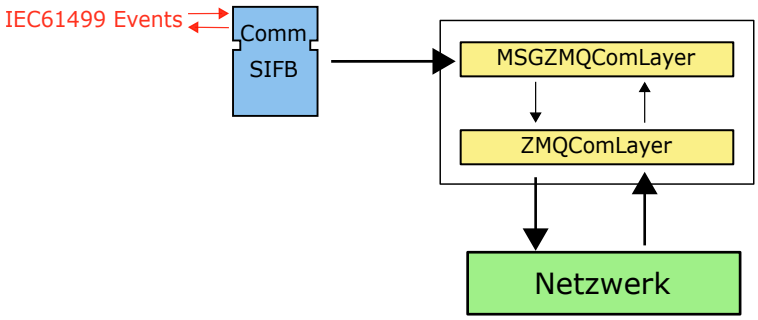

Abb. 3.2: Übersicht ZeroMQ in FORTE

Im Klassendiagramm Abbildung 3.3 werden diese zwei Schichten mittels der Klassen CMSGZMQComLayer und CZMQComLayer definiert.

Der Datenaustausch zwischen diesen zwei Schichten erfolgt über *sendData()* und *recvData()*. Außerdem wird ein externer EventHandler CZMQHandler benötigt, um Nachrichten asynchron erhalten zu können.

Im Folgenden werden die drei wichtigsten Elemente genauer erläutert.

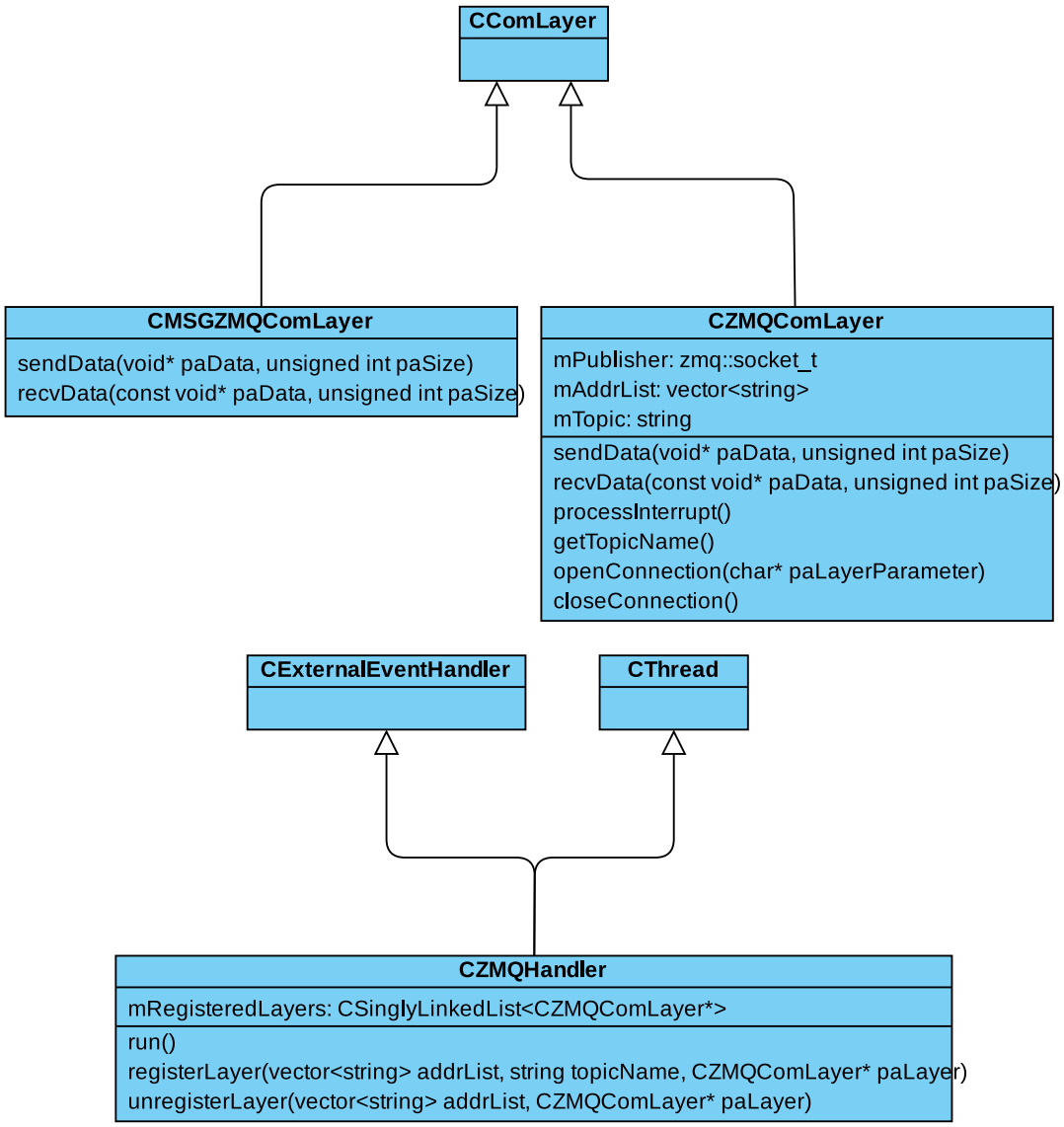

Abb. 3.3: Klassenstruktur ZeroMQ in Forte

3.2.1 CMSGZMQComLayer

Die Klasse CMSGZMQComLayer implementiert den obersten Layer. Hier findet der Datenaustausch mit den Datenein- und ausgängen des Funktionsblockes statt, siehe Unterabschnitt 2.1.2. In Forte sind die Datentypen dieser Kanäle als C++ Klassen implementiert. Daher werden diese zuerst in C++ Datentypen konvertiert.

Um die Daten an die unterste Schicht ZMQComLayer weiterzuleiten, wird für jeden Kanal eine eigene ZeroMQ Message erzeugt, die dann zu einer Multipart Message zusammengesetzt werden, wie in Unterabschnitt 2.3.3 erläutert. Der umgekehrte Ablauf geschieht beim Empfangen der Daten.

3.2.2 CZMQComLayer

Beim Initialisieren des SIFB werden über *openConnection()* die Adresse(n) und das Topic übergeben. Für den Publisher gibt es noch die Option, folgende Parameter zu setzen:

- **HWM:** Die High Water Mark definiert ein Limit für die maximale Anzahl noch zu sendender Nachrichten, die sich in den Warteschlangen des Sockets befinden dürfen. Wird dieser Wert erreicht, werden Nachrichten verworfen. Bei Problemen mit der Verbindung wird so ein Überlaufen des Speichers verhindert. Für Subscriber gibt es das HWM für die Eingangswarteschlange.
- **DSCP:** Über den differentiated services code point wird der TOS (Type of Service) des Sockets gesetzt. Tabelle **??** zeigt mögliche Optionen für den DSCP Wert.
- **Ratelimit:** Das Ratelimit kann nur bei EPGM gesetzt werden. Es begrenzt die Datenrate bei Multicast. Standardmäßig wird es auf 810 Mbit/s gesetzt.

Service Class	DSCP Wert
Standard	0
Low-priority data	0x20
Low-latency data	0x48, 0x50, 0x58
High-throughput data	0x28, 0x30, 0x38

Tabelle 3.1: DSCP Werte

Handelt es sich um einen Publisher SIFB, wird im ZMQComLayer der ZeroMQ Publisher Socket erzeugt und gebunden bzw. verbunden.

Vor dem Senden von Daten wird das Topic als Message an den Anfang der Multipart Message angefügt, die vom MSGZMQComLayer weitergeleitet wurde. Das Ergebnis ist in Abbildung 3.4 abgebildet.

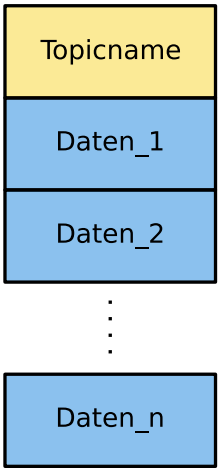

Abb. 3.4: ZeroMQ Multi-Message Format mit Topicframe

Dieser Vorgang ist notwendig, da die Subscriptions über Prefix Matching realisiert sind. Da das Matching nicht über Framegrenzen hinausgeht, können Daten so nicht fälschlicherweise als Topic erkannt werden.
Der cppzmq Header [1] stellt eine *send(zmq_ socket_ t)* Methode bereit, die das Senden der Multipart Message übernimmt und die Daten über das Netzwerk schickt.

In Abbildung 3.5 ist der Sendevorgang als Sequenzdiagram dargestellt.

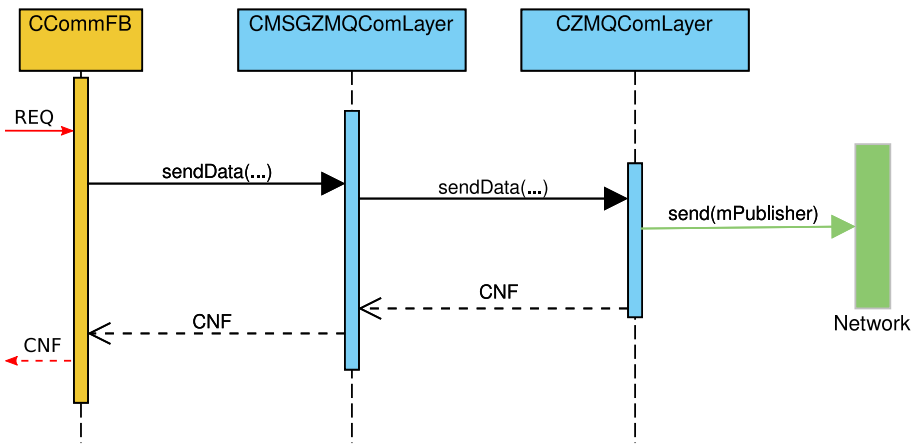

Abb. 3.5: Sequenzdiagramm sendData(...)

Handelt es sich bei dem SIFB um einen Subscriber, wird der ZMQComLayer beim CZMQHandler mit *registerLayer()* registriert, siehe Unterabschnitt 3.2.3. Dabei wird die Adresse an diesen übergeben, um eine Verbindung aufzubauen. Es können mehrere Publisher Adressen sein, zu dem der Subscriber Socket verbindet.

3.2.3 CZMQHandler

Der CZMQHandler kümmert sich um das Empfangen der Nachrichten. Jeder Subscriber Funktionsblock in der 4DIAC Applikation verwaltet einen eigenen Netzwerkstack. Die Adressen und Subscription sind im ZMQComLayer gespeichert. Jeder Subscriber registriert bei Initialisierung seinen ZMQComLayer beim Handler. Diese werden in der Liste *mRegisteredLayers* gespeichert. Wenn das erste Mal *registerLayer* aufgefrufen wird, wird *run()* in einem eigenen Thread aufgerufen. Durch aufrufen eines eigenen Threads wird die Ausführung der Funktionsblöcke nicht blockiert. Anschließend wird der Subscribe Socket erzeugt, verbunden und die Subscription gesetzt.

In *run()* wird über die Funktion *inPoller.wait_ all(inEvents, timeout)* in einer Endlos-Schleife auf einkommende Nachrichten gewartet. Dafür werden die Sockets dem Poller hinzugefügt. Über *timeout* wird festgelegt, wie lange die Funktion warten soll, bevor sie zurückkehrt. Es ist auf null gesetzt.

Wenn Daten an einem oder mehreren Sockets ankommen, werden entsprechende Socketreferenzen dem Vektor *inEvents* hinzugefügt. Der Poller empfängt dabei die Nachrichten nicht. Dafür wird für alle Einträge im Vektor die *recv(socket_ ref, 1)* Methode aufgerufen. Die 1 bedeutet, dass die Methode nicht auf Daten warten soll, um den Thread nicht zu blockieren.
Wenn eine Multipart Message über einen ZeroMQ Socket empfangen wird, ist garantiert dass alle Teilnachrichten angekommen sind. ZeroMQ arbeitet mit best-effort, das heißt es werden alle Teile empfangen oder gar keine.

Die empfangene Nachricht wird über *recvData()* an den ZMQComLayer weitergeleitet. Mittels *interruptCommFB* wird der SIFB unterbrochen. Anschließend wird im Handler über *startEventChain(mFb)* eine neue Ereigniskette gestartet.
Eine Ereigniskette ist eine Abfolge von Funktionsblöcken, die über Eventein- und Ausgänge miteinander verbunden sind und so ihre entsprechenden Aufgaben erledigen. Die Eventchain wird im *EventChainEventExecutionThread* ausgeführt.
Der Übergabeparameter mFb steht für den ersten FB der Kette, in diesem Fall der Subscriber SIFB. Dieser ruft über *processInterrupt() recvData()* im MSGZMQComLayer auf, um die empfangenen Daten an seine Datenausgänge zu legen. Über das *IND* Event wird das nächste Glied in der Kette informiert. Abbildung 3.6 zeigt das dazugehörige Sequenzdiagramm.

Zu beachten ist, dass über *startEventChain(mFb)* die Eventchain nicht sofort ausgeführt wird, sondern zuerst einer Warteschlange für externe Events hinzugefügt wird. Diese ist in der Größe begrenzt. Der *EventChainEventExecutionThread* fügt die externen Events einer internen Queue hinzu, wenn sie ausgeführt werden sollen. Bei voller externer Queue werden neu ankommende Events verworfen.

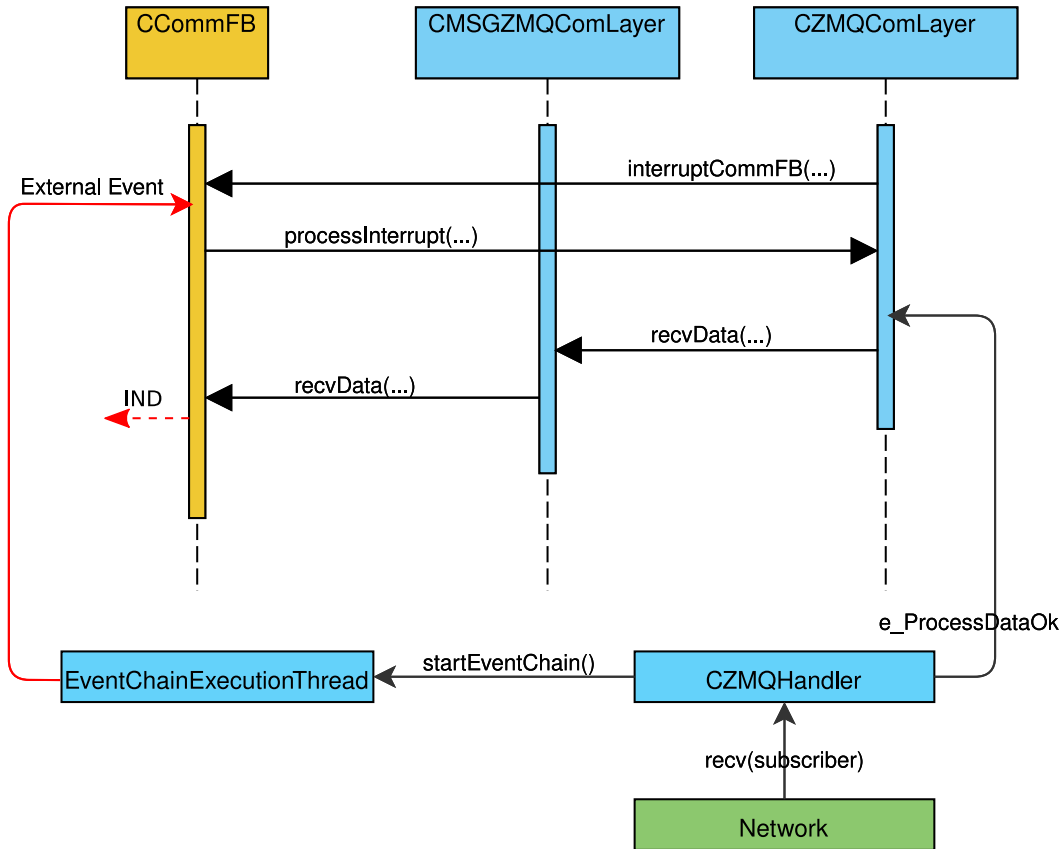

Abb. 3.6: Sequenzdiagramm recvData(...)

3.3 Kommunikations-SIFB

Für die Kommunikation mittels ZeroMQ wird ein SIFB benötigt. In 4DIAC werden Service Interface Funktionsblöcke nach dem "IEC 61499 compliance profile for feasibility demonstrantions" von HOLOBLOC Inc. [9] umgesetzt.

3.3.1 Interface

Abbildung 3.7 zeigt den Publish SIFB, wie er in der 4DIAC IDE verwendet wird. Das Interface für eine Publish-Subscribe Kommunikation wurde in Unterabschnitt 2.1.4 definiert.

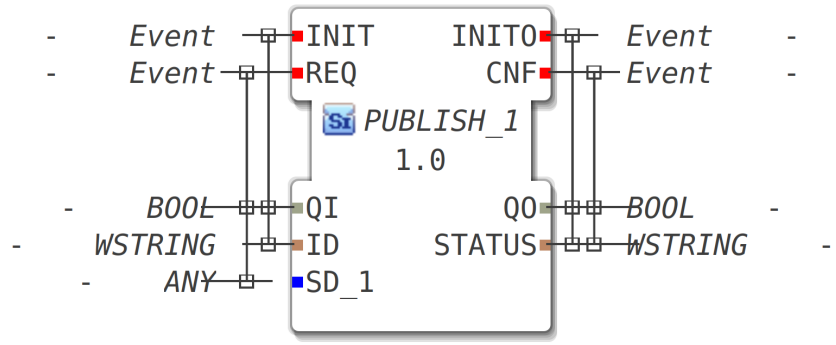

Abb. 3.7: Publish SIFB

Über den Eventeingang INIT wird der Funktionsblock initialisiert. Dafür muss QI auf true gesetzt sein, mit false wird der FB deaktiviert.
Die Parameter für die Initialisierung werden als String über den Dateneingang ID bereitgestellt.

Das ANY bei SD_n bedeutet, dass der Datentyp des Dateneinganges vom Ausgang des angeschlossenen Funktionsblockes abhängt. Es sind Publish-Funktionsblöcke mit bis zu zehn Dateneingängen implementiert.

INITO wird gesetzt, wenn die Initialisierung beendet ist. Wenn eine Nachricht abgeschickt wurde, wird CNF ausgelöst. STATUS und QO geben Auskunft darüber, ob der Service erfolgreich ausgeführt wurde.

Abbildung 3.8 zeigt einen Subscribe SIFB mit sechs Datenausgängen. IND zeigt an, dass eine Nachricht angekommen ist.

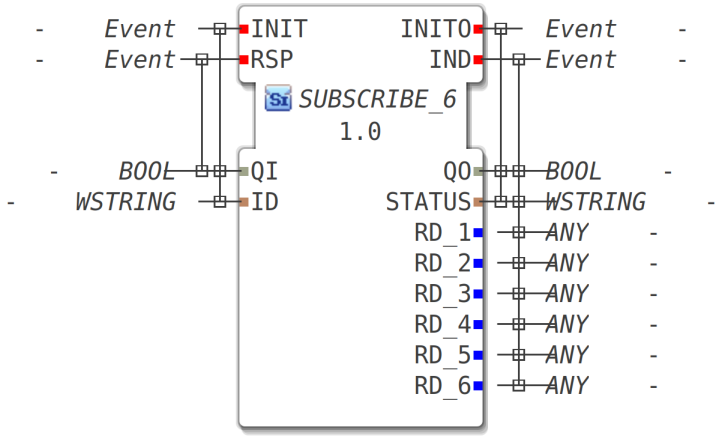

Abb. 3.8: Subscribe SIFB

3.3.2 ID Parameterstring

Über den ID String werden die Parameter an ZeroMQ übergeben. Abbildung 3.9 zeigt das Template für den Publish FB.

```
"Serialisierung[].zmq[Adresse | topic | hwm, DSCP, Ratelimit]"
```

Abb. 3.9: Parameter String für Publish ID

Über Serialisierung[] kann festgelegt werden, in welcher Form die Daten übermittelt werden. Das ZeroMQ Interface unterstützt msgzmq und fbdk.

fbdk ist die Standard Encoding Variante von 4DIAC/Forte und ist im compliance profile [9] definiert. Die serialisierten Daten werden als Buffer an den ZMQComLayer übergeben und dort in eine ZeroMQ Message kopiert. Die Multipart Message besteht nur aus dem Topic- und einem Datenframe.

Im zmq[] Teil wird mittels '|' zwischen den Parametern getrennt. Adresse und Topic müssen angegeben werden. Der dritte Abschnitt für die Konfiguration ist optional.

Der ID-String einer Subscribe FB, dargestellt in Abbildung 3.10, hat keinen Konfigurationsabschnitt. Es können mehrere Adressen von Publishern angegeben werden, um ein Netzwerk aufzubauen.

```
"Serialisierung[].zmq[Adresse1, Adresse2, ... Adresse_n | topic]"
```

Abb. 3.10: Parameter String für Subscribe ID

Über den Adressenprefix wird der Modus festgelegt, den ZeroMQ verwendet. Es werden drei Moden unterstützt:

- **Direkte Kommunikation** [*tcp://IP:Port*]: Subscriber können zu mehreren Publishern direkt verbinden. Dafür müssen mehrere Adressen übergeben werden.
- **Proxy** [*ptcp://IP:Port*]: Es wird ein Proxy gestartet, wie er in Abbildung 2.15 definiert wird. Publisher und Subscriber verbinden zum Proxy. Dieser Modus ist flexibler, da nur die Adresse des Proxies bekannt sein muss. Der Publisher sendet die gleiche Nachricht mehrmals für jeden Subscriber, der eine Subscription hat.
- **EPGM** [*epgm://interface;multicast-address:port*]: Multicast. Ein Router oder Switch leitet die Nachrichten weiter. Ein Vorteil gegenüber des Proxy ist die Anzahl Nachrichten, die gesendet werden müssen. Der Publisher sendet nur eine Message, die vom Router oder Switch kopiert und an die Subscriber weitergeleitet wird. Dafür wird OpenPGM [17] genutzt.

4 Testaufbau und Durchführung

Die in Kapitel 3 vorgestellte Kopplungsschicht soll getestet und die Performance evaluiert werden. Die dafür verwendeten Messwerte und Testaufbauten werden erläutert.

Es sollen Latenz, Throughput und Jitter gemessen werden, wenn für die Kommunikation ZeroMQ verwendet wird. 4DIAC/Forte bietet standardmäßig bereits Client-Server und Pub-Sub Funktionsblöcke. Diese werden ebenfalls in den Tests verwendet, um einen Vergleich für die Evaluierung zu haben.

4.1 Verwendete Hard- und Software

Für die Messungen werden Tests auf vier Raspberry Pi 4 Model B ausgeführt. Der kleine Rechner verfügt über 4GB Arbeitsspeicher, einen 64bit Quadcore mit 1.5Ghz und einen Gigabit Ethernet Anschluss.
Als Betriebssystem ist Raspbian GNU/Linux 10 (buster) mit dem 4.19.75 Kernel installiert, welches kein 64bit unterstützt. Die Geräte werden über einen Gigabit Ethernet Switch miteinander verbunden. Die Bandbreite wurde mit iperf3 gemessen und beträgt 950 Mbit/s.

Die Tests werden in 4DIAC IDE Version 2.0.1 designed. Forte wird als Quellcode heruntergeladen und mittels Make-File Projekt in Eclipse IDE für C/C++ Developers Version 4.22.0 [5] kompiliert.

4.2 Messwerte und Methoden

Die Messungen werden mit zwei und vier RaspberryPi durchgeführt, die über einen Switch miteinander verbunden sind. Abbildung 4.1 zeigt den Aufbau mit zwei Knoten im Gerätemanager von 4DIAC. Der Aufbau mit vier Knoten ist in Abbildung 4.2 dargestellt.

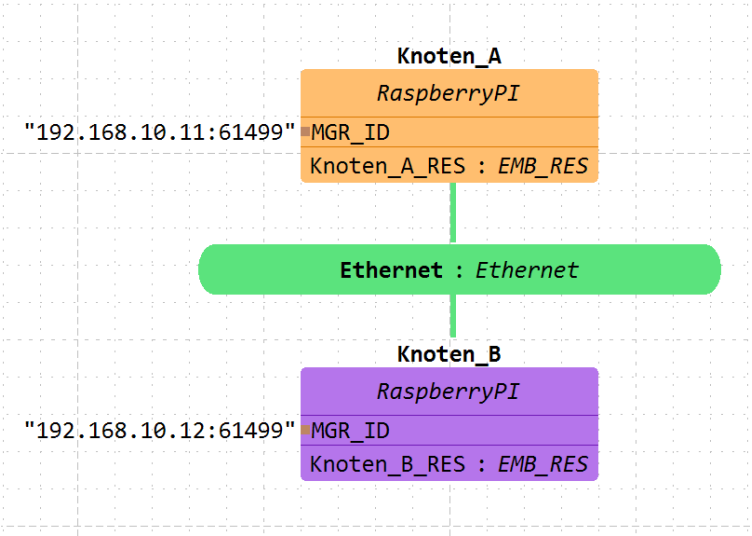

Abb. 4.1: Aufbau mit zwei Knoten

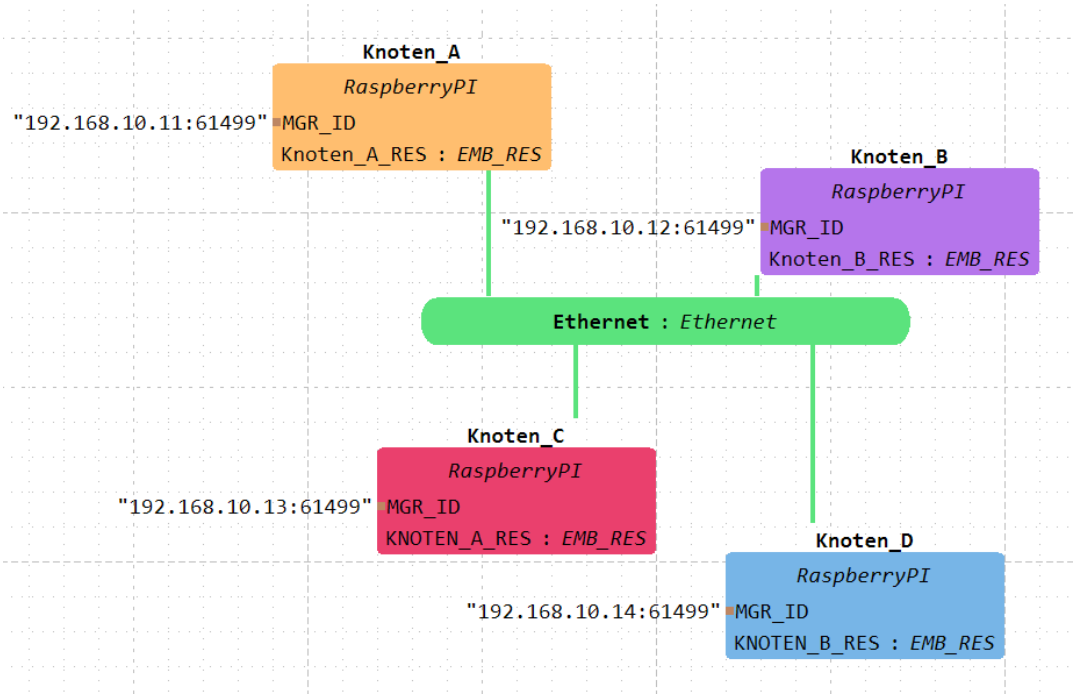

Abb. 4.2: Aufbau mit vier Knoten

Für alle Messungen wird die gleiche Nachrichtenstruktur verwendet. In C++ wird die Nachricht als struct definiert Listing 4.1.

```
struct Message
{
        TForteUInt32 index;  //32 bit unsigned
        std::string msg;
}
```

Listing 4.1: Testnachricht

In Forte sind String die einfachste Möglichkeit, die Größe der Nachricht zu skalieren. Dafür muss nur die Länge verändert werden.

Um einen solchen Struct über Datenverbindungen transportieren zu können, muss dieser in der 4DIAC IDE zuerst definiert werden. Anschließend wird eine dazugehörige C++ Klasse exportiert. Nun findet sich in der IDE der Datentyp TESTMSG für die Datenein- und Ausgänge der Funktionsblöcke.

4.2.1 Latenz und Jitter

Die Latenz und Jitter sind wichtige Messwerte, um die Stabilität und Effizienz einer Kommunikation über ein Netzwerk zu bewerten. Die Latenz bezeichnet die Zeit, die ein Datenpaket benötigt, um von einem Punkt zu einem anderen zu gelangen. Schwankungen in diesen Laufzeiten werden als Jitter bezeichnet. Großen Einfluss auf die Latenz hat das Übertragungsmedium, die Größe des Pakets und andere Computer- und Speicherverzögerungen.

Um die Messung der Latenz zu vereinfachen, wird die Roundtriptime (RTT) gemessen. Die RTT ist die Zeit die ein Datenpaket von seinem Ursprungpunkt zum Zielpunkt (dies entspricht der Latenz) und wieder zurück braucht. Die Latenz ergibt sich in Folge als die halbe RTT. Für die direkte Latenzmessung müssten die Systemclocks auf den Netzwerkknoten synchronisiert werden.

Abbildung 4.4 zeigt die 4DIAC Applikation, mit der die RTT zwischen zwei Knoten gemessen wird. Die orangenen FBs werden auf Knoten A ausgeführt, die lilanen auf Knoten B. Die Publish und Subscribe FBs verwenden den fbdk Layer (Unterabschnitt 3.3.2), um den Vergleich mit den Referenzmessungen zu vereinfachen. Um die Nachricht von Knoten A zu senden, wird das REQ Event an PUBLISH_Knoten_A gesetzt. Die Nachricht wird auf Knoten B einfach weitergeleitet. Der Funktionsblock Msg2Msg kopiert den Struct Datentyp mit den Daten von seinem Dateneingang an seinen Datenausgang. Dieser ist notwendig, da zwei ANY Datenverbindungen nicht miteinander verbunden werden können, siehe Unterabschnitt 3.3.1.

Mit average_time soll bestimmt werden wie sich das Weiterleiten auf die Latenz auswirkt. Wird die Testnachricht wieder an Knoten A empfangen, wird das IND Event ausgelöst.
Um die RTT zu bestimmen wird die Zeit gemessen, die zwischen dem Setzen des REQ Events des und dem Auslösen des IND Events des SUBSCRIBE_Knoten_A vergeht.

Der in Abbildung 4.3 dargestellte RTT_Test generiert die Testnachricht mit der spezifizierten Größe.

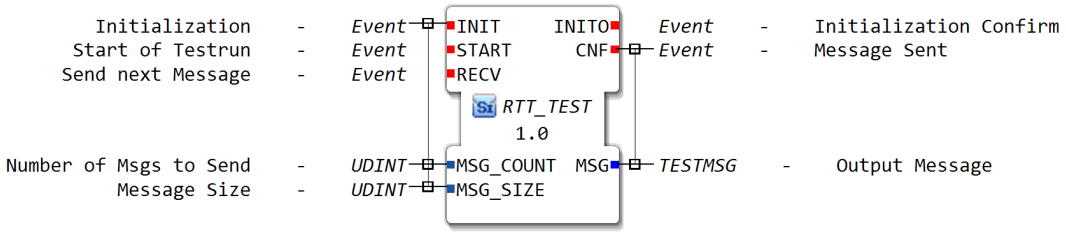

Abb. 4.3: Funktionsblock für RTT Messung

Mit CNF wird der Sendevorgang ausgelöst. Wenn mit IND am Subscriber FB angezeigt wird, dass die letzte Nachricht wieder zurückgekommen ist, wird RECV gesetzt. Daraufhin wird die nächste Nachricht gesendet. Über MSG_COUNT wird festgelegt, wie viele Nachrichten insgesamt versendet werden sollen.

Die Zeit wird in RTT_Test gemessen. Eine Stopuhr wird gestartet wenn CNF gesetzt wird und wieder bei einem RECV Event gestoppt. Die vergangene Zeit ist die RTT. Dafür werden die in ZeroMQ implementierten Funktionen *zmq_stopwatch_start()* und *zmq_stopwatch_stop()* verwendet. *zmq_stopwatch_start()* startet die Stoppuhr und gibt einen void* zum Handler zurück. *zmq_stopwatch_stop(void* watch)* stoppt die Stoppuhr und gibt die Anzahl der Mikrosekunden zurück, die seit dem Starten vergangen sind.

Dieser Test wird für verschiedene Nachrichtengrößen zwischen 8 byte und 16384 byte ausgeführt. Während jedes Testlaufes werden 1.000.000 Nachrichten verschickt. Es wird die mittlere Latenz aus dem Mittelwert der Messwerte bestimmt. Der größte Wert wird als worst-case Latenz bezeichnet und entspricht dem größten Wert der 1.000.000 gemessenen Zeiten. Der Jitter wird als Differenz zweier aufeinanderfolgender Messwerte berechnet. Auch hier wird der Mittelwert und wost-case Wert ermittelt.

Dieser Test wird mit vier Knoten nochmals wiederholt. Die in Abbildung 4.4 dargestellte Applikation wird auf einem Knoten C und D nochmals ausgeführt. Damit soll die Auswirkung einer erhöhten Netzauslastung auf die RTT zwischen Knoten A und B ermittelt werden. Die Messung wird wieder auf Knoten A durchgeführt.

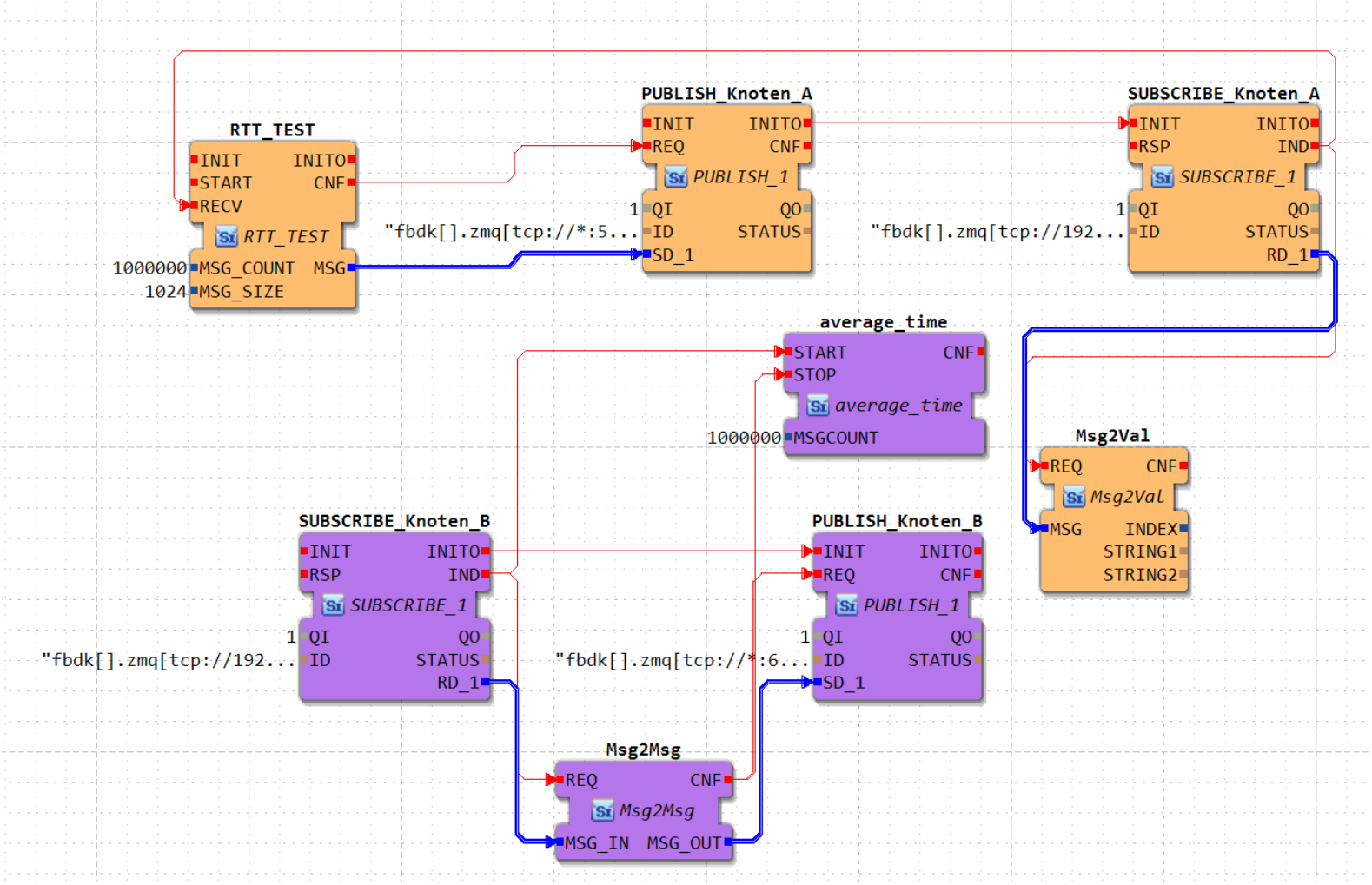

Abb. 4.4: Messapplikation für RTT mit zwei Knoten

Für die Referenzmessung wird grundsätzlich die gleiche Messmethode verwendet. Die Kommunikation findet hingegen über die Client-Server Funktionsblöcke statt, die in Forte standardmäßig implementiert sind. Die Daten werden über TCP Sockets gesendet. Abbildung 4.5 zeigt den Messaufbau. Die Nachricht, die an RD_1 am Server ankommt wird an SD_1 weitergeleitet. Über RSP geht diese zurück an CLIENT_Knoten_A. Dieser gibt mit CNF bekannt, dass die Nachricht zurückgekommen ist.

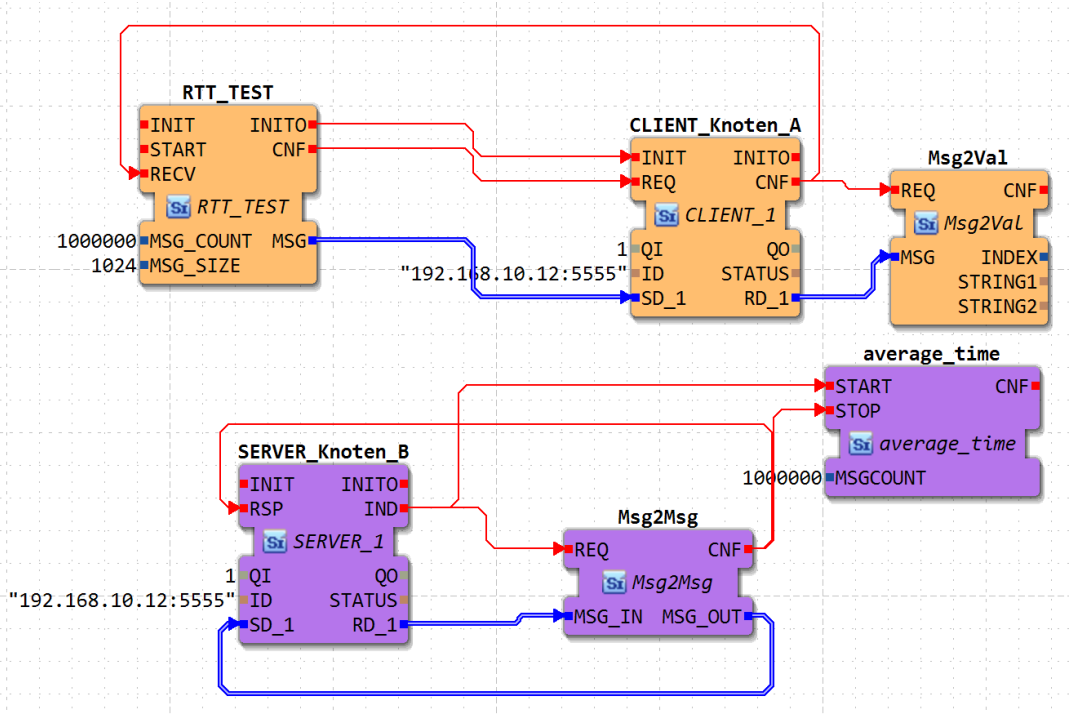

Abb. 4.5: Messapplikation für RTT mit zwei Knoten - Client Server Kommunikation

4.2.2 Throughput

Der Datendurchsatz, im Folgenden auch Throughput bezeichnet, ist ein Maß dafür, wie viele Daten während eines gewissen Zeitraums erfolgreich über ein Medium übertragen werden können.

In Unterabschnitt 3.2.3 wurde erläutert, dass für jede empfangene Nachricht am Subscriber Funktionsblock ein externes Event ausgelöst wird. Diese werden in einer Warteschlange gereiht und anschließend bearbeitet. Die Größe dieser Warteschlange ist aber begrenzt. Auch wenn die Nachrichten erfolgreich über das Netzwerk transportiert worden sind, ist nicht gewährleistet, dass sie auch an die Applikation gelangen. Daher wird im Rahmen dieser Arbeit der Throughput als die Anzahl Nachrichten pro Sekunde definiert, die der Subscriber erfolgreich an die Applikation weiterleiten kann.

Abbildung 4.8 zeigt die Messapplikation für zwei Knoten.
Der Funktionsblock THR_TEST_SEND (Abbildung 4.6) auf Knoten A generiert mit den initialisierten Parametern die Testnachricht.

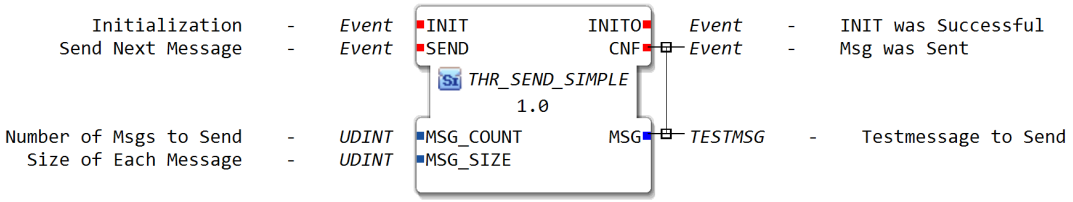

Abb. 4.6: Sende-Funktionsblöcke für Throughput Messung

Mit CNF wird die Nachricht an den PUBLISH_Knoten_A weitergegeben und REQ gesetzt, um diese zu senden. CNF zeigt einen erfolgreichen Sendevorgang an. Der Publisher soll die Nachrichten so schnell wie möglich senden. Daher löst ein CNF sofort über SEND den nächsten Sendevorgang aus.

SUBSCRIBE_Knoten_B auf Knoten B wird die Nachrichten solange bearbeiten und weiterleiten, wie seine Warteschlange Platz hat. Tritt der Fall ein, dass die Warteschlange von Knoten B voll ist, werden Events und damit ankommende Nachrichten verworfen. Für jede erfolgreich weitergeleitete Testnachricht wird IND gesetzt, was wiederum RECV am THR_TEST_RECV (Abbildung 4.7) auslöst. Die RECV Events werden gezählt.

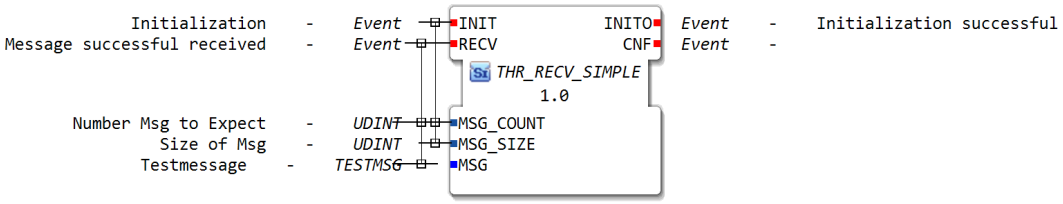

Abb. 4.7: Empfangs-Funktionsblock für Throughput Messung

Mit dem ersten RECV wird mittels *zmq_stopwatch_start()* die Stoppuhr gestartet. Wenn 1.000.000 erfolgreiche Empfangsvorgänge gezählt wurden, wird mit *zmq_stopwatch_stop()* die Stoppuhr angehalten. Teilt man die 1.000.000 durch die gemessene Zeit erhält man den Throughput. Für die Megabit pro Sekunde werden die 1.000.000 mit der Nachrichtengröße multipliziert.

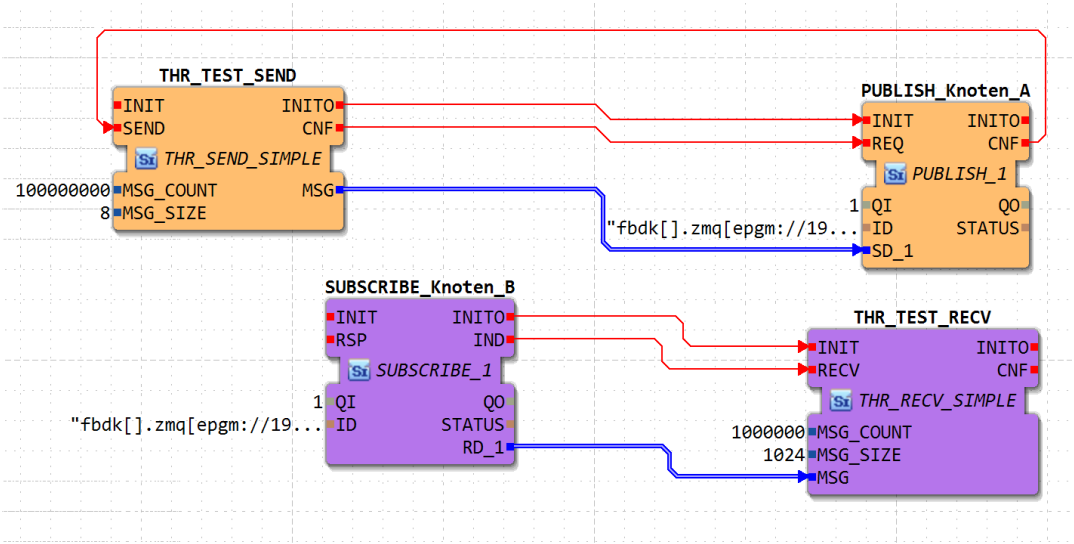

Abb. 4.8: Messapplikation für Throughput mit 2 Knoten

In einem weiteren Test werden vier Knoten verwendet Es wird die gleiche Applikation wie in Abbildung 4.8 verwendet. Der Teil auf Knoten B (lila) wird auf Knoten C und D ebenfalls ausgeführt. Alle drei Knoten empfangen die Testnachrichten. Die Performance bei der Verteilung von Daten an mehrere Subscriber soll so gemessen werden.

Für die Referenzmessung werden die exakt gleichen Applikationen verwendet. Über die ID lässt sich steuern, welche Implementierung verwendet wird.
Die Standard Pub/Sub FBs in Forte verwenden Multicast über UDP Sockets. Die ID entspricht der Multicast Adresse mit einem Port. Damit Subscriber die Daten empfangen können, treten sie der gleichen Multicast Gruppe bei.

5 Testaufbau und Durchführung

In diesem Kapitel werden die Ergebnisse der in Kapitel 4 erläuterten Tests aufgezeigt. Für die RTT Messungen wurden jeweils 1.000.000 Durchläufe durchgeführt. Für den Throughput werden 1.000.000 erfolgreich zugestellte Testnachrichten erwartet. Diese Messungen wurden jeweils fünf mal wiederholt und anschließend der Mittelwert gebildet.

5.1 Ergebnisse für RTT/Latenz Messungen

5.1.1 Messergebnisse - Zwei Knoten

Die folgenden Ergebnisse beziehen sich auf den Messaufbau aus Abbildung 4.4.

In Abbildung 5.1 sind die Ergebnisse der Messungen zu RTT und Latenzen für verschiedene Größen der Testnachricht zusammengefasst. Dabei wurde die Messapplikation auf zwei Knoten ausgeführt. Für ZeroMQ wurden die Pub/Sub FBs direkt verbunden, siehe Unterabschnitt 3.3.2 unter "direkte Kommunikation".
Forte verfügt über einen statischen Receivebuffer für TCP Verbindungen. Dieser wurde für die Messungen auf 4 MB erhöht.
Die Werte für DDS und die dazugehörigen TCP Messungen sind aus [19] entnommen.

Nachricht (byte)		8	16	32	64	128	256
ZMQ (µs)	RTT	212	208	212	218	221,8	231
	Latenz	106	104	106	109	110,9	115,5
TCP (µs)	RTT	154	150	148,9	147,9	148,8	177,5
	Latenz	77	75	74,45	73,95	74,4	88,75
DDS (µs)	RTT	294	302	302	313	315	318
	Latenz	147	151	151	156,5	157,5	159
TCP (µs)	RTT	162	162	163	162	164	169
	Latenz	81	81	81,5	81	82	84,5
Nachricht (byte)		512	1024	2048	4096	8192	16384
ZMQ (µs)	RTT	243,5	268,9	340	385,6	560	763
	Latenz	121,75	134,45	170	192,8	280	381,5
TCP (µs)	RTT	238,5	279	327	x	x	x
	Latenz	119,25	139,5	163,5	x	x	x
DDS (µs)	RTT	322	365	411	767	1475	2930
	Latenz	161	182,5	205,5	383,5	737,5	1465
TCP (µs)	RTT	170	196	236	290	362	490
	Latenz	85	98	118	145	181	245

Abb. 5.1: Übersicht der Messergebnisse zu RTT und Latenzen mit zwei Knoten

In [19] wurde keine IEC61499 Implementierung verwendet. Stattdessen wurde eine eigene Testumgebung in C++ realisiert, die aus der Klasse iec2dds besteht.

Für Datenpakete kleiner 256 byte sind die TCP Messungen vergleichbar. Für die Forte Implementierung braucht eine Nachricht mit 128 byte 77 µs, bis sie am Datenausgang des Subscriber FBs anliegt. Für iec2dds liegt der Messwert bei 81 µs.
Ab 512 byte steigen die Werte für TCP in Forte stark an. Eine Erklärung dürfte der Overhead durch die Serialisierung im fbdk Layer sein. Ab 4096 byte waren keine Messungen mehr mit Client Server FBs möglich, da Forte ständig abstürtzte.

Im Bereich bis 256 byte ist der Vergleich zwischen DDS und ZeroMQ durchaus machbar. Für DDS dauert eine Übertragung von 128 byte 157,5 µs. Mit ZeroMQ dauert diese 110,9 µs und ist damit ungefähr um ein Drittel schneller. Dies liegt daran, dass ZeroMQ direkt auf Berkeley Sockets aufbaut und damit weniger Overhead aufweist als DDS. Mit größeren Nachrichten steigt der Unterschied stark an. Ab 2048 byte verdoppelt sich die Latenz für DDS.

In [19] wird vermutet, dass der Anstieg ab 1024 byte mit der Größe des Ethernet Frame zusammenhängt. Ist die Nachricht größer, muss die Nachricht in mehreren Teilen verschickt werden, was die Latenz erhöht.
Es wurde noch der Vergleich zwischen dem fbdk Layer und dem msgzmq durchgeführt. Die Nachrichtengröße beträgt 1024 byte mit zwei Knoten. Für fbdk ist das Throughput 23.605 Nachrichten pro Sekunde. Mit msgzmq werden 36.245 Nachrichten pro Sekunde erreicht. Das ist darauf zurückzuführen, dass die Daten nur in eine ZeroMQ Message kopiert werden müssen. Anders bei fbdk, wo die Daten noch serialisiert werden.

5.1.2 Messergebnisse - Vier Knoten

Die folgenden Ergebnisse beziehen sich auf den Versuch aus Abbildung 4.4. Es wird die Latenz und Jitter für eine 1024 byte Testnachricht gemessen. Im Versuch mit vier Knoten gibt es zwei Szenarien.

Für Szenario 1 verbindet jeder Subscriber sich mit jedem Publisher. Damit hat jeder Subscriber drei Verbindungen.

Für Szenario 2 wird die Verbindung über einen Proxy hergestellt, wie er in Abbildung 2.14 dargestellt ist. ZeroMQ bietet dafür die Funktion *zmq_proxy(frontend, backend)*. Das Frontend ist ein XSUB Socket, zudem sich die Publisher verbinden. Das Backend ist ein XPUB Socket, der als Verbindungsstelle für die Subscriber dient. Der Proxy wird auf Knoten A gestartet.

In [19] werden zwischen den zwei zusätzlichen Knoten 1024 byte große Nachrichten gesendet, was eine zusätzliche Netzauslastung von 96,90 Mbps erzeugt. Für Forte/-

ZeroMQ wird zwischen Knoten C und D eine 4096 byte große Nachricht hin und hergeschickt, um eine zusätzliche Last von 94 Mbps zu generieren.

Abbildung 5.2 zeigt die Ergebnisse für Szenario 1. Wenn die Publisher und Subscriber direkt miteinander verbunden sind, hat die zusätzliche Last keine Auswirkung, solange die Bandbreite der Verbindung nicht ausgeschöpft wird. Das gleiche gilt für DDS.

Nachricht (1024 byte)	RTT (µs)		Jitter (µs)	
	Mittelwert	Worst-case	Mittelwert	Worst-case
ZeroMQ (ohne Last)	268,9	11113	18,26	5429
ZeroMQ (mit Last)	271,38	15002	19	7344,5
DDS (ohne Last)	353	11719	58,047	11722
DDS (mit Last)	348	11641	59,863	11290

Abb. 5.2: Übersicht der Messergebnisse zu RTT und Latenzen mit vier Knoten (Szenario 1)

In Szenario 2 (Abbildung 5.3) sorgt der zusätzliche Proxy für eine Verdoppelung der Latenz. Damit steigt auch der Jitter an. Die Worst-case Messwerte unterscheiden sich nur gering. Zusätzliche Last wirkt sich ebenfalls schlecht auf die Performance aus. Die RTT steigt um 100 µs.
Für DDS sinkt der Mittelwert für Jitter und RTT sogar mit Belastung.

Nachricht (1024 byte)	RTT (µs)		Jitter (µs)	
	Mittelwert	Worst-case	Mittelwert	Worst-case
ZeroMQ (ohne Last)	511	11595	30,8	5549
ZeroMQ (mit Last)	618,4	30574	27,2	15008

Abb. 5.3: Übersicht der Messergebnisse zu RTT und Latenzen mit vier Knoten (Szenario 2)

5.2 Ergebnisse für Throughput

5.2.1 Messergebnisse - Zwei Knoten

Die folgenden Ergebnisse beziehen sich auf den Messaufbau aus Abbildung 4.8.

Der Receivebuffer für UDP wird in Forte 4 MB erhöht. Abbildung 5.4 zeigt den Vergleich von ZeroMQ mit direkter Kommunikation und UDP Multicast. Der Aufbau besteht aus zwei Knoten. Auf einem Knoten befindet sich der Publisher auf dem zweiten der Subscriber. Für den Vergleich wird ZeroMQ wie UDP mit fbdk Serialisierung verwendet.

Für Nachrichten ab 1024 byte muss das HWM im Forte Code angepasst werden, siehe Unterabschnitt 3.2.2. Ab dieser Grenze braucht der Subscriber FB deutlich länger, um die Nachricht an den Datenausgang zu legen. Kommen zu viele Nachrichten an, wird nicht nur die externe Event Warteschlange voll, sondern auch die interne. Dadurch bleibt der FB hängen und arbeitet nicht mehr richtig. Die HWM wird auf 50 gesetzt. Die externe Warteschlange kann 25 externe Events zwischenspeichern.

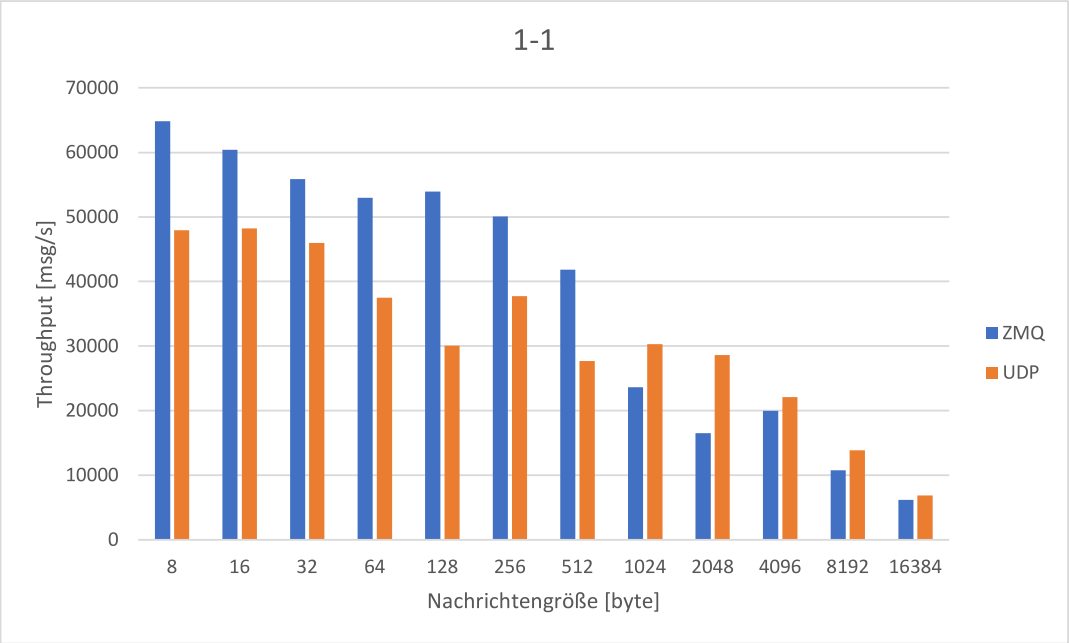

Abb. 5.4: Throughput mit zwei Knoten

Die dargestellten Messwerte entsprechen der Anzahl an Nachrichten, die der Subscriber Funktionsblock pro Sekunde verarbeiten kann.

ZeroMQ ist bei kleinen Datenpaketen bis 512 byte schneller als UDP. Für 128 byte werden mit UDP 30035 Nachrichten in der Sekunde erreicht. Mit ZeroMQ sind es mit

53946 Nachrichten pro Sekunde beinahe doppelt so viele.
Das liegt am intelligenten Message Batching, dass ZeroMQ verwendet. Dabei werden die Nachrichten nicht einzeln verschickt, sondern zuerst in der Warteschlange gereiht. Sobald die Netzwerkschnittstelle bereit ist, werden alle Nachrichten sofort losgeschickt. Es benötigt weniger Zeit, eine große Nachricht über den Stack zu versenden, als viele kleine Nachrichten.

5.2.2 Messergebnisse - Vier Knoten

Abbildung 5.5 zeigt die Ergebnisse mit vier Knoten. Dabei wird von einem Publisher an drei Subscriber Daten verteilt. ZeroMQ wird mit direkter Verbindung, epgm und einem Proxy getestet. Für epgm wird OpenPGM 5.2 [17] eingesetzt. Die HWM aller Subscriber wird wieder auf 50 gesetzt. Für epgm wird die Datenrate auf 900 Mbps erhöht. Der Proxy wird auf Knoten A gestartet, der als Publisher agiert.

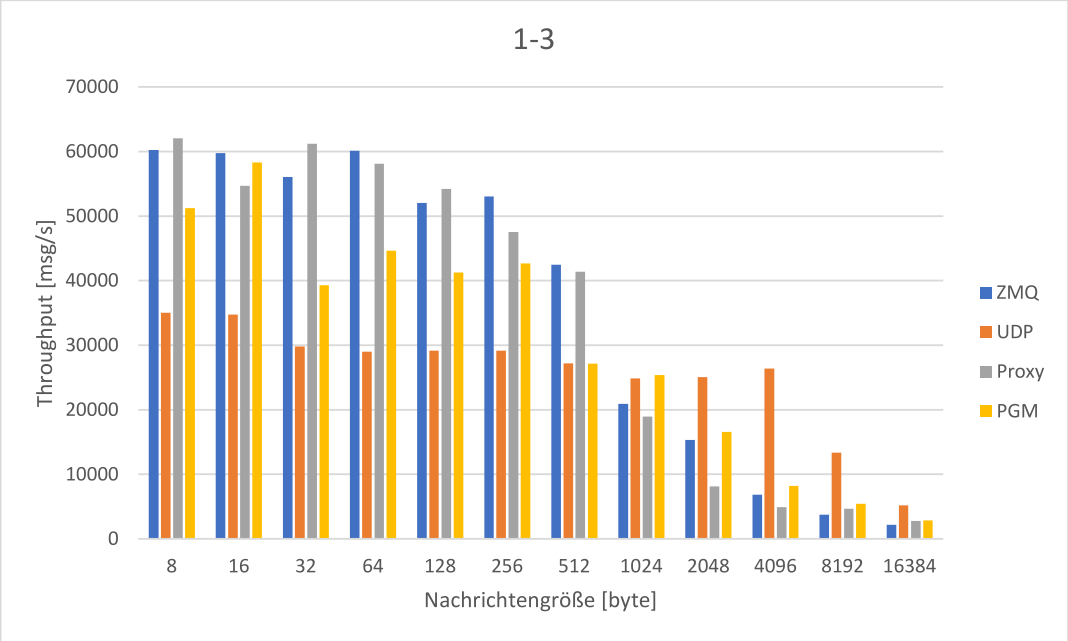

Abb. 5.5: Throughput mit vier Knoten. Ein Publisher und drei Subscriber.

Bei kleinen Nachrichtengrößen sind alle ZeroMQ Varianten durch das Message Batching im Vorteil gegenüber UDP. Für Nachrichten über 1024 byte weist UDP einen höheren Throughput auf. Sehr deutlich wird das für 4096 byte. UDP ist mit 26415 Nachrichten pro Sekunde dreimal so schnell wie PGM, das mit 8199 Nachrichten nur ein Drittel in der gleichen Zeit schafft.

Mit Multicast steht jedem Subscriber die volle Bandbreite von 900 Mbps zur Verfügung. Die Nachricht wird nur einmal versendet und vom Switch weitergeleitet.

Für ZeroMQ mit direkter Verbindung und mit Proxy wird für jeden Subscriber eine eigene Nachricht gesendet. Ab 1024 byte wird damit bereits die volle Bandbreite ausgenutzt. Jeder Subscriber verfügt somit nur über ein Drittel der Bandbreite. Ab diesem Punkt ist nicht mehr der Subscriber FB der begrenzende Faktor, sondern die zur Verfügung stehende Bandbreite.

Für epgm sollte das Gleiche gelten wie für UDP Multicast. In den Versuchen wurden aber maximal 300 Mbps erreicht. Es kam auch öfter zu einem Absturz des pgm-receiver Socket. Daraufhin wurde versucht, die Datenrate anzupassen und die Sende- und Empfangsbuffer zu erhöhen. Dies hat das Ergebnis jedoch nicht verändert.
Für Nachrichten kleiner 512 byte hat diese obere Grenze keine weitere Auswirkung. Ab 1024 byte werden die 300 Mbps bereits voll ausgenutzt.

6 Fazit

In dieser Thesis wurde untersucht, ob sich ZeroMQ erfolgreich in 4DIAC/Forte anwenden lässt. Dafür wurde eine Kopplungsschicht implementiert. Mit dieser wurden Messungen zu Latenz, Jitter und Throughput durchgeführt. Zum Vergleich sind die bereits implementierten Lösungen für Client/Server und Pub/Sub herangezogen worden. Es wurde ebenfalls versucht, einen Vergleich mit vorhandenen Daten zu DDS herzustellen.
Mit ZeroMQ lassen sich hohe Durchsatzraten mit geringer Latenz erzielen. Besonders bei kleinen Nachrichtengrößen hat die Bibliothek große Vorteile, wie zum Beispiel bei Anwendungen mit hohen Datenraten und kleinen Paketen. Ein Nachteil ist der hohe Implementierungsaufwand. ZeroMQ ist keine fertige Middleware Lösung. Es sind ebenfalls kaum Quality of Service Features vorhanden.

In 4DIA/Forte läuft ZeroMQ sehr stabil. Wird ein Proxy verwendet, ist das Aufbauen einer verteilten Applikation einfach und kann leicht erweitert werden. Mit Subscriptions lässt sich die Applikation ebenfalls einfacher handhaben als mit Multicast Gruppen.

OpenPGM für die pgm Implementierung ist aufgrund der vorliegenden Ergebnisse nicht zu empfehlen. Das Projekt wird seit mehreren Jahren nicht mehr weiterentwickelt. Das zeigt sich an der schlechten Leistung bei Datenmengen größer 1024 byte. Ebenfalls läuft Forte damit nicht stabil und es sind häufige Abstürze zu erwarten.

Ein Problem mit der derzeitigen Kopplungsschicht ist, dass die Daten im Handler empfangen werden müssen. PUB/SUB Sockets sind nicht thread-safe. Sie müssen dort eingesetzt werden, wo sie erzeugt wurden. Dadurch verliert der ZMQHandler zwischen 20 µs und 30 µs gegenüber dem Sockethandler. Die Daten müssen zuerst empfangen, dann kopiert und im ZMQComLayer nochmals kopiert werden. Der Sockethandler gibt den Socketdescriptor einfach an den nächsten Layer weiter, der dann die Daten empfängt.
Die Sockettypen RADIO/DISH sind in der Beta Phase und sind thread-safe. Dadurch wäre es möglich, eine Socketreferenz an den ZMQComLayer weiterzugeben, statt die Daten zu kopieren.
Mittlerweile wurde auch UDP Multicast implementiert, dass sich als Ersatz für pgm und Proxy anbietet.

ZeroMQ eignet sich hervorragend, wenn die Middleware speziell an die Anwendung angepasst werden soll und es vertretbar ist, fehlende Features selber hinzuzufügen.

Literatur

[1] *Cppzmq - C++ Binding for Libzmq*. The ZeroMQ project. URL: https://github.com/zeromq/cppzmq (besucht am 04. 01. 2022).

[2] *Eclipse 4diac - The Open Source Environment for Distributed Industrial Automation and Control Systems*. URL: https://www.eclipse.org/4diac/index.php (besucht am 10. 06. 2021).

[3] *Eclipse 4diac Communication Architecture*. URL: https://www.eclipse.org/4diac/en_help.php?helppage=html/development/forte_communicationArchitecture.html (besucht am 03. 01. 2022).

[4] *Eclipse 4diac Documentation*. URL: https://www.eclipse.org/4diac/en_help.php (besucht am 24. 10. 2020).

[5] *Eclipse CDT™ (C/C++ Development Tooling) | Projects.Eclipse.Org*. URL: https://projects.eclipse.org/projects/tools.cdt (besucht am 23. 01. 2022).

[6] *FBench Project : Home*. URL: http://oooneida-fbench.sourceforge.net/ (besucht am 10. 06. 2021).

[7] Daniel Happ u. a. „Meeting IoT Platform Requirements with Open Pub/Sub Solutions“. In: *Annals of Telecommunications* 72.1-2 (1-2), S. 41–52. ISSN: 0003-4347, 1958-9395. DOI: 10.1007/s12243-016-0537-4. URL: http://link.springer.com/10.1007/s12243-016-0537-4 (besucht am 11. 03. 2021).

[8] Pieter Hintjens. *ZeroMQ - The Guide*. URL: https://zguide.zeromq.org/ (besucht am 24. 10. 2020).

[9] *IEC 61499 Compliance Profile for Feasibility Demonstrations*. URL: https://holobloc.com/doc/ita/ (besucht am 17. 06. 2021).

[10] *IEC 61499-4:2013 - IEC-Normen - VDE VERLAG*. URL: https://www.vde-verlag.de/iec-normen/219630/iec-61499-4-2013.html (besucht am 15. 05. 2021).

[11] International Electrotechnical Commission. *IEC 61131: Programmable Controllers - Part 3: Programming Languages*. Geneva: IEC.

[12] International Electrotechnical Commission. *IEC 61131: Programmable Controllers - Part 5: Communications*. Geneva: IEC.

[13] International Electrotechnical Commission. *International Standard IEC 61499-1*. second edition. Geneva: IEC.

[14] *ISaGRAF Technology*. Rockwell Automation. URL: https://www.rockwellautomation.com/en-us/support/documentation/technical-data/isagraf_20190326-0743.html (besucht am 10. 06. 2021).

[15] *Market Analysis - Zeromq*. URL: http://wiki.zeromq.org/whitepapers:market-analysis (besucht am 31. 03. 2021).

[16] NXTcontol. *Wir sind Pioniere*. nxtControl. URL: https://www.nxtcontrol.com/ (besucht am 10. 06. 2021).

[17] *OpenPGM*. URL: https://code.google.com/archive/p/openpgm/ (besucht am 05. 01. 2022).

[18] Bo Petersen u. a. „Smart Grid Communication Middleware Comparison - Distributed Control Comparison for the Internet of Things:“ In: *Proceedings of the 6th International Conference on Smart Cities and Green ICT Systems*. 6th International Conference on Smart Cities and Green ICT Systems. Porto, Portugal: SCITEPRESS - Science and Technology Publications, S. 219–226. DOI: 10.5220/0006303302190226.

[19] Jinsong Yang u. a. „Data Distribution Service for Industrial Automation“. In: *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*. 2012 IEEE 17th Conference on Emerging Technologies & Factory Automation (ETFA 2012). Krakow, Poland: IEEE, S. 1–8. DOI: 10.1109/ETFA.2012.6489544.

[20] *Yueyi Automation - FBuilder*. URL: http://www.yueyiautomation.com/ (besucht am 10. 06. 2021).

[21] *ZeroMQ*. URL: https://zeromq.org/ (besucht am 30. 03. 2021).

[22] *Zmq_socket - ZeroMQ Api Manual*. URL: http://api.zeromq.org/master:zmq-socket (besucht am 26. 10. 2021).

[23] *ZMTP Specification*. URL: http://rfc.zeromq.org/spec/37/ (besucht am 26. 10. 2021).

[24] Alois Zoitl und Thomas Strasser. *Distributed Control Applications*. 2. Aufl. CRC Press. ISBN: 978-1-315-21504-4. DOI: 10.1201/b19391. URL: https://www.taylorfrancis.com/books/9781482259063 (besucht am 13. 04. 2021).

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Stellen sind als solche kenntlich gemacht. Die Arbeit wurde bisher weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Dornbirn, am 23. Januar 2022 David Pfefferkorn