Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Martin Hua

Erstellung eines Konzeptes zur Portierung von Basis-funktionen der Interprozesskommunikation von QNX auf ein Linux Betriebssystem

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Martin Hua

Erstellung eines Konzeptes zur Portierung von Basisfunktionen der Interprozesskommunikation von QNX auf ein Linux Betriebssystem

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Wolfgang Fohl
Zweitgutachter : Prof. Dipl.-Inf. Tonja Kaltenhäuser

Abgegeben am 11. März 2013

Martin Hua

Thema der Bachelorarbeit

Erstellung eines Konzeptes zur Portierung von Basisfunktionen der Interprozesskommunikation von QNX auf ein Linux Betriebssystem

Stichworte

IPC, shared memory, Nachrichtenaustausch, Client, Server, Betriebssysteme, QNX, Linux, POSIX, Bibliotheken, Frameworks, IPC-Mechanismen, Evaluation, zeroMQ, SIMPL

Kurzzusammenfassung

Im Rahmen dieser Arbeit soll ein Konzept erstellt werden, wie die QNX-Basisfunktionen unter dem Betriebssystem Linux umgesetzt werden können. Dabei werden die QNX-Basisfunktionen analysiert, die in einer firmenspezifischen IPC-Bibliothek enthalten sind. Aufbauend auf dieser Analyse und die daraus resultierenden Anforderungen soll eine neue IPC-Bibliothek unter Linux erstellt werden, die diese Anforderungen umsetzen. Für die Umsetzung der neuen IPC-Bibliothek werden nach IPC-Techniken, externen Frameworks oder Bibliotheken gesucht, die dann bei der Evaluation analysiert und bewertet werden. Die neue IPC-Bibliothek wird jeweils mit der aus der Evaluation resultierenden Bibliothek SIMPL und zeroMQ umgesetzt.

Martin Hua

Title of the paper

Development of a concept for porting basic functions of interprocess communication of QNX on a Linux operating system

Keywords

IPC, shared memory, communication, client, server, operating systems, QNX, Linux, POSIX, libraries, frameworks, IPC-mechanisms, evaluation, zeroMQ, SIMPL

Abstract

In this thesis, a concept is being developed, how the QNX basic functions can be implemented on the Linux operating system. The QNX basic functions which are contained in a company-specific IPC library will be analysed. Based on this analysis and the resulting requirements, a new IPC library is being created on Linux that fulfils the requirements. In the context of the implementation of the new IPC library, IPC techniques, external frameworks or libraries are searched for, which are analysed and assessed in the following evaluation. Afterwards, the new IPC library will be implemented with the resulting libraries SIMPL and zeroMQ of the evaluation.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

1 Einleitung

Diese Arbeit befasst sich mit der Frage, ob es möglich ist, die Basisfunktionen unter dem Betriebssystem QNX auf das Linux Betriebssystem abzubilden. Hierzu soll ein Konzept erarbeitet werden, wie die QNX-Basisfunktionen mit minimalem Aufwand unter dem Betriebssystem Linux umzusetzen sind.
Diese Arbeit wurde für ein mittelständisches Unternehmen geschrieben, das Geräte zur Messung bestimmter elementarer Stoffe herstellt. Auf Wunsch des Unternehmens wird sein Name in der Arbeit nicht erwähnt.

1.1 Motivation

Für die Geräte, die das mittelständische Unternehmen herstellt, verwenden sie separate BoxPCs, die in den Geräten eingesetzt werden. Auf diesen BoxPCs läuft das Betriebssystem QNX. Es werden viele Schnittstellen, wie beispielsweise serielle Schnittstellen, Audiotreiber oder zwei LAN Ports genutzt, die je nach Chipsatz bei den BoxPCs von QNX unterstützt werden oder nicht. Daher muss immer darauf geachtet werden, welche Chipsätze in den BoxPCs eingebaut sind und ob sie von dem Betriebssystem QNX unterstützt wird. Die Suche nach dem passenden BoxPC für die Geräte ist enorm schwierig und zeitintensiv. Durch den Umstieg vom QNX Betriebssystem auf Linux soll die Suche nach passenden BoxPCs leichter gestaltet werden.
Zuerst wurde QNX als Betriebssystem gewählt, weil die firmeneigenen Applikationen in Echtzeit arbeiten sollten. Inzwischen ist es nicht mehr bedeutend, ob die firmeneigenen Applikationen in Echtzeit arbeiten müssen und somit der Umstieg auf Linux in Frage kommt. Durch den Umstieg auf das Betriebssystem Linux würden die Lizenzkosten entfallen, die für jedes ausgelieferte Gerät bei dem QNX Betriebssystem bezahlt werden müssen.

1.2 Zielsetzung

Ziel dieser Arbeit ist es, ein Konzept zu erarbeiten, wie die QNX-Basisfunktionen unter dem Betriebssystem Linux abzubilden sind. Dabei sollen die QNX-Basisfunktionen, die in der firmenspezifischen IPC-Bibliothek verwendet wurden, unter dem Betriebssystem Linux abgebildet werden.

1.3 Gliederung der Arbeit

Die Arbeit ist folgenderweise aufgebaut:

Im zweiten Kapitel wird die firmenspezifische IPC-Bibliothek unter dem QNX Betriebssystem analysiert. Dabei werden die in der IPC-Bibliothek verwendeten QNX- und POSIX-Funktionen erläutert, die für das Verständnis dieser Bibliothek wichtig sind. Die Kommunikationsstruktur zwischen den firmeneigenen Programmen, die als Client und Server fungieren, wird beschrieben. Im Anschluss des zweiten Kapitels werden die Funktionen der firmenspezifischen IPC-Bibliothek beschrieben.
Im dritten Kapitel wird ein Konzept erstellt, wie die QNX-Funktionen der firmenspezifischen IPC-Bibliothek unter dem Linux Betriebssystem umgesetzt werden. Aufbauend auf dem vorigen Kapitel werden zwei Anwendungsszenarien beschrieben, wie ein Client und ein Server sich verhalten, wenn sie miteinander kommunizieren. Daraus ergeben sich Anforderungen bezüglich der Kommunikation zwischen Client und Server, die unter dem Linux Betriebssystem umgesetzt werden sollen. Dafür wird eine neue IPC-Bibliothek unter dem Linux Betriebssystem erstellt, die diese Anforderungen umsetzt. Für die Umsetzung dieser Anforderungen benötigt die neue IPC-Bibliothek geeignete IPC-Techniken, externe Frameworks oder Bibliotheken, die evaluiert werden müssen. Nach der Evaluation werden zwei Testszenarien beschrieben, wie das An- und Abmelden eines Clients bei einem Server und die Benachrichtigung der beim Server registrierten Clients erfolgen. Im Anschluss dieses Kapitels wird die neue IPC-Bibliothek jeweils mit der aus der Evaluation resultierenden Bibliothek *SIMPL* und *zeroMQ* umgesetzt. Nach der Umsetzung werden beide externen Bibliotheken verglichen.
Die Ergebnisse im dritten Kapitel werden im vierten Kapitel zusammengefasst.
Im fünften und letzten Kapitel wird das Erarbeitete zusammengefasst und mit einem Ausblick über dieses Thema abgeschlossen.

1.4 Abgrenzung

Diese Arbeit konzentriert sich auf die lokale Kommunikation innerhalb eines Rechnersystems. Die Kommunikation zwischen Rechnersystemen über das Netzwerk wird momentan nicht verstärkt genutzt und wird somit in dieser Arbeit nicht weiter betrachtet. Die Kommunikation über das Netzwerk wird in Zukunft noch ein wichtiges Thema werden, dann wenn sie verstärkt zum Einsatz kommt.
Neben der firmenspezifischen IPC-Bibliothek existieren noch weitere, firmenspezifische Bibliotheken und Programme, die die Firmensoftware bilden. Wie diese Bibliotheken und Programme arbeiten und welche QNX-Funktionen sie verwenden, wird in dieser Arbeit nicht weiter betrachtet.
Diese Arbeit untersucht ausschließlich nur die QNX-Basisfunktionen, die in der firmenspezifischen IPC-Bibliothek verwendet wurden. Andere QNX-spezifische Funktionen, die QNX anbietet, bleiben außen vor und werden in dieser Arbeit nicht weiter betrachtet.

2 Analyse der firmenspezifischen IPC-Bibliothek unter QNX

Die IPC-Bibliothek ist eine firmenspezifische Bibliothek, die den firmeninternen Programmen ermöglicht, unter QNX miteinander zu kommunizieren. Sie besteht aus einer Anzahl von APIs (*application programming interface*) und den dazugehörigen Dateien, die jeweils individuell für die einzelnen, firmeninternen Programme bezüglich der Kommunikation angepasst wurden. Der Kern der IPC-Bibliothek ist das API **lib_ipc.h**, auf dem die anderen APIs basieren. Das API lib_ipc.h wird in diesem Kapitel auf seine Funktionalitäten analysiert. Diese Funktionalitäten sollen dann unter dem Betriebssystem Linux abgebildet werden. Die APIs, die auf lib_ipc.h basieren, werden in dieser Arbeit nicht weiter betrachtet.

Im Kapitel 2.1 und 2.2 werden die QNX- und POSIX-Funktionen beschrieben, die für das Verständnis der Kommunikation zwischen Programmen und der Behandlung von shared memory Bereichen wichtig sind. Im Kapitel 2.3 wird die Kommunikationsstruktur zwischen den Programmen in der firmeneigenen Software erläutert. Im Kapitel 2.4 werden die Funktionen des APIs lib_ipc.h erläutert, die die QNX- und POSIX-Funktionen enthalten.

2.1 Beschreibung der in der IPC-Bibliothek verwendeten QNX-Funktionen

In diesem Kapitel werden die QNX-Funktionen vorgestellt, die für die Kommunikation unter dem Betriebssystem QNX relevant sind. Ein Teil dieser QNX-Funktionen werden in den Funktionen des APIs lib_ipc.h verwendet. Der andere Teil wird von einem Client-Thread oder einem Server-Thread direkt aufgerufen.

Die Kommunikation zwischen Threads innerhalb eines Prozesses oder zwischen Threads anderer Prozesse erfolgt über den Austausch von Nachrichten.
Angenommen, es existiert ein Client-Thread und ein Server-Thread, die innerhalb eines Prozesses kommunizieren möchten. Damit der Server-Thread Nachrichten vom Client-Thread empfangen kann, muss er einen sogenannten Kommunikationskanal (Channel) erzeugen. Dafür ruft er die QNX-Funktion `ChannelCreate()` auf:

Listing 1: QNX-Funktion `ChannelCreate()`

```
int ChannelCreate (unsigned flags);
```

Dem formalen Parameter `flags` können bestimmte Flags übergeben werden, die die Eigenschaften des erzeugten Channels bestimmen. Als Rückgabewert liefert diese QNX-Funktion im Erfolgsfall die Channel ID eines neu erzeugten Channels zurück.

Durch diese ID kann ein erzeugter Channel eindeutig identifiziert werden. Im Fehlerfall wird der Wert -1 zurückgeliefert.
Wenn der Server-Thread `ChannelCreate()` aufruft, dann gehört der erzeugte Channel dem Prozess, in dem der Server-Thread läuft [1] [2].

Damit der Client-Thread eine Nachricht an den Server-Thread senden kann, muss er eine Verbindung zum Channel des Server-Threads herstellen. Dafür ruft er die QNX-Funktion `ConnectAttach()` auf:

Listing 2: QNX-Funktion `ConnectAttach()`

```
int ConnectAttach (uint32_t nd, pid_t pid, int chid, unsigned index, int flags);
```

„Diese Funktion stellt die Verbindung zu dem Channel chid des Prozesses pid, der auf dem Node/Rechner nd läuft, her.“ [1].
Dem ersten formalen Parameter `nd` wird der „node descriptor“ des Knotens beziehungsweise des Rechners übergeben. Die Konstante `ND_LOCAL_NODE` wird nur an diesen Parameter übergeben, wenn es sich um einen lokalen Rechner handelt.
Dem formalen Parameter `pid` wird die Prozess ID des Prozesses, der den Channel besitzt, übergeben. Dem formalen Parameter `chid` wird der Rückgabewert von der Funktion `ChannelCreate()` übergeben. Wenn eine Verbindung zu einem Channel hergestellt wird, soll laut nachfolgendem Zitat die Konstante `_NTO_SIDE_CHANNEL` an den formalen Parameter `index` übergeben werden: “Treating a connection as a file descriptor can lead to unexpected behavior. Therefore, you should OR _NTO_SIDE_CHANNEL into *index* when you create a connection. If you do this, the connection ID is returned from a different space than file descriptors; the ID is greater than any valid file descriptor.“ [2].
Durch die Übergabe dieser Konstante ist sichergestellt, dass die Funktion `ConnectAttach()` die Connection ID und nicht den Filedeskriptor als Rückgabewert zurückliefert. Über den formalen Parameter `flags` kann bestimmt werden, ob die Verbindung geschlossen wird, wenn eine `exec*()` Funktion aufgerufen wird.
Die Funktion `ConnectAttach()` liefert im Erfolgsfall die Connection ID zurück.
Im Fehlerfall liefert sie den Wert -1 zurück [2].
Durch die Connection ID kann eine Verbindung eindeutig identifiziert werden [1].

Angenommen, ein Client-Thread und ein Server-Thread laufen nicht innerhalb eines Prozesses, sondern in unterschiedlichen Prozessen und möchten miteinander kommunizieren. Mit den oben genannten QNX-Funktionen ist es auch möglich, eine Kommunikation zwischen dem Client-Thread und dem Server-Thread, die auf unterschiedlichen Prozessen laufen, aufzubauen. Es ist jedoch angenehmer, für die oben genannte Situation die QNX-Funktionen `name_attach()` und `name_open()` zu verwenden.
Mit der Funktion `name_attach()` registriert der Server-Thread seinen Namen im Namensraum und erzeugt einen Channel (Siehe Listing 3). Der Client-Thread ruft die Funktion `name_open()` auf, um mit dem Namen des Server-Threads eine Verbindung zu ihm zu öffnen (Siehe Listing 4) [2].

Listing 3: QNX-Funktion `name_attach()`

```
name_attach_t* name_attach (dispatch_t* dpp, const char* path, unsigned flags);
```

Als ersten Übergabeparameter kann eine sogenannte *„dispatch“* Struktur übergeben werden, wenn diese explizit mit der QNX-Funktion `dispatch_create()` erzeugt wurde [3].
Falls nicht, wird dem formalen Parameter `dpp` der Wert NULL übergeben. Durch die Übergabe von NULL ruft `name_attach()` intern die QNX-Funktionen `dispatch_create()` und `resmgr_attach()` auf, die einen Channel erzeugen. Diese beiden QNX-Funktionen werden in dieser Arbeit nicht weiter betrachtet.
Als zweiten Übergabeparameter wird der Name des Server-Threads an den formalen Parameter `path` übergeben. Dieser Name wird im Namensraum unter `/dev/name/[local|global]/`*`path`* angelegt. Beim letzten formalen Parameter `flags` kann bestimmt werden, ob dieser Name im Verzeichnis `local` oder `global` abgelegt wird.
Die Funktion `name_attach()` liefert im Erfolgsfall einen Zeiger zurück, der auf die Struktur *name_attach_t* verweist. Im Fehlerfall liefert sie den Wert NULL zurück.
Mit diesem Zeiger kann der Server-Thread auf die Channel ID zugreifen [2].

Listing 4: QNX-Funktion `name_open()`

```
int name_open (const char* name, int flags);
```

Als ersten Übergabeparameter wird der Name des Servers-Threads übergeben, zu dem eine Verbindung geöffnet werden soll. Über den formalen Parameter `flags` kann bestimmt werden, ob der Name des Server-Threads im lokalen oder im globalen Verzeichnis gesucht wird. Die Funktion `name_open()` liefert im Erfolgsfall die Connection ID zurück. Im Fehlerfall liefert sie den Wert -1 zurück [2].

Die nachfolgende Abbildung veranschaulicht noch einmal das Zusammenspiel zwischen dem Funktionspaar `name_attach()` und `name_open()`:

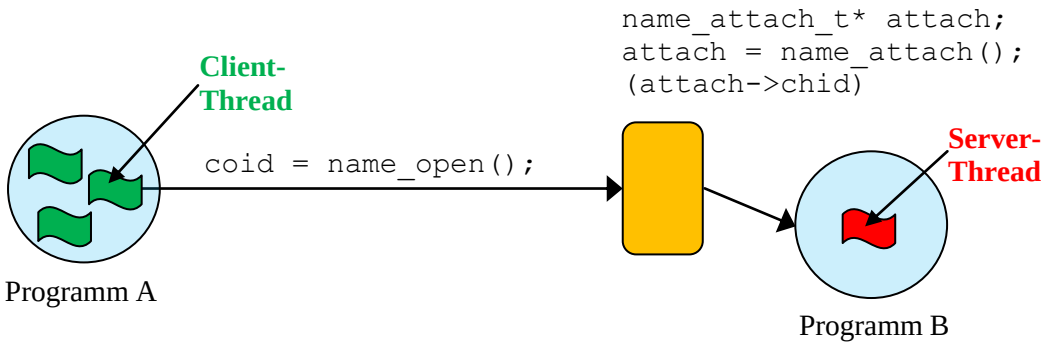

Abb. 1: Entstehung einer Verbindung zwischen Client-Thread und Server-Thread mit `name_attach()` und `name_open()`

Die Interaktion zwischen dem Funktionspaar `ChannelCreate()` und `ConnectAttach()` entspricht der in der Abbildung 1 dargestellten Interaktion zwischen `name_attach()` und `name_open()`.
Beide Verfahren wurden hier vorgestellt, weil beide in der Firmensoftware verwendet werden.

Steht die Kommunikation zwischen dem Client-Thread und dem Server-Thread, gibt es zwei Möglichkeiten, wie sie miteinander kommunizieren können:

- Synchrone Kommunikation über *„Message Passing“*
- Asynchrone Kommunikation über *„Pulses“*

Im Folgenden werden diese beiden Kommunikationsarten näher erläutert.

Synchrone Kommunikation über *Message Passing*

Unter dem QNX Betriebssystem wird die synchrone Kommunikation über das ***Message Passing*** realisiert. Das Message Passing wird durch die nachfolgenden QNX-Funktionen gebildet:

- `MsgSend()`
- `MsgReceive()`
- `MsgReply()`

Die nachfolgenden Abbildungen 2 und 3 zeigen, in welchem Zustand ein Client-Thread und ein Server-Thread sich befinden, wenn sie die oben genannten QNX-Funktionen aufrufen:

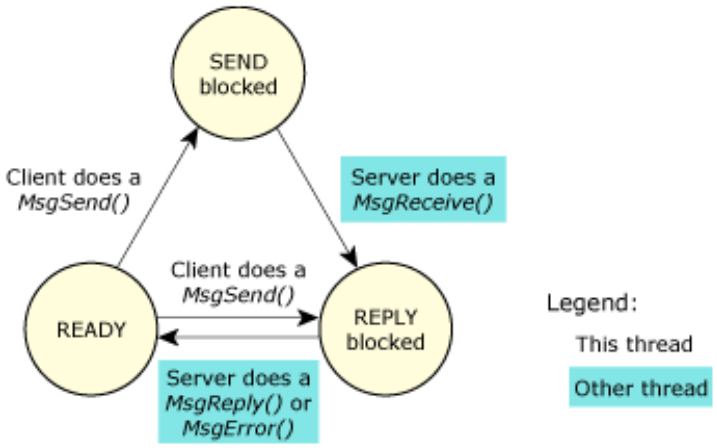

Abb. 2: Der Client-Thread durchläuft diese Zustände, wenn er eine Nachricht an den Server-Thread sendet

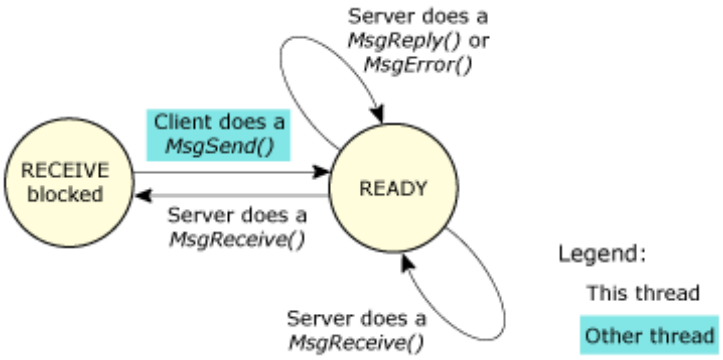

Abb. 3: Der Server-Thread wechselt zwischen diesen beiden Zuständen, wenn er auf Nachrichten wartet und diese empfängt

Zu Beginn der synchronen Kommunikation befindet sich der Client-Thread und der Server-Thread im Zustand *READY*.
Mit der QNX-Funktion `MsgSend()` sendet der Client-Thread eine Nachricht an den Channel des Server-Threads (Siehe Listing 5). Die Ausführung des Client-Threads wird durch den Aufruf von `MsgSend()` blockiert. Der Client-Thread wechselt vom Zustand *READY* in den Zustand *SEND blocked*. Dieser Zustand bedeutet, dass der Server-Thread die Nachricht des Client-Threads noch nicht aus seinem Channel gelesen hat. Somit hat der Server-Thread die Nachricht des Client-Threads noch nicht bearbeitet [1].

Listing 5: QNX-Funktion `MsgSend()`

```
int MsgSend (int coid, const void* smsg, int sbytes, void* rmsg, int rbytes);
```

Als ersten Übergabeparameter wird die Connection ID übergeben, die die QNX-Funktion `name_open()` oder `ConnectAttach()` als Rückgabewert zurückliefert. Die Nachricht, die der Client-Thread dem Server-Thread übermitteln möchte, wird an den formalen Parameter `smsg` übergeben. Die Größe dieser Nachricht wird an den formalen Parameter `sbytes` in Bytes angegeben. Die Antwort des Server-Threads wird im Puffer `rmsg` gespeichert. Der formale Parameter `rbytes` gibt die Größe dieses Puffers in Bytes an. Der Rückgabewert von `MsgSend()` ist vom Datentyp *int*. Diese QNX-Funktion liefert im Erfolgsfall einen Status zurück, den sie von der QNX-Funktion `MsgReply()` erhält. Im Fehlerfall liefert sie den Wert -1 zurück [1] [2].

Der Server-Thread ruft die QNX-Funktion `MsgReceive()` auf, um Nachrichten vom Client-Thread zu empfangen (Siehe Listing 6). Durch den Aufruf von `MsgReceive()` wird die Ausführung des Server-Threads blockiert. Der Server-Thread wechselt vom Zustand *READY* in den Zustand *RECEIVE blocked*. Die Ausführung des Server-Threads wird solange blockiert, bis eine Nachricht vom Client-Thread bei ihm eintrifft. Der Server-Thread wechselt dann vom Zustand *RECEIVE*

blocked wieder in den Zustand *READY*. Er liest die empfangene Nachricht vom Client-Thread aus seinem Channel und bearbeitet diese.
Da der Server-Thread die Nachricht des Client-Threads aus seinem Channel gelesen hat, wechselt der Client-Thread vom Zustand *SEND blocked* in den Zustand *REPLY blocked*. Dieser Zustand bedeutet, dass der Server-Thread die Nachricht vom Client-Thread gelesen hat, aber noch nicht dazu gekommen ist, diese Nachricht zu beantworten [1].

Listing 6: QNX-Funktion `MsgReceive()`

```
int MsgReceive (int chid, void* msg, int bytes, struct _msg_info* info);
```

Als ersten Übergabeparameter erhält diese QNX-Funktion die Channel ID, die die QNX-Funktion `name_attach()` oder `ChannelCreate()` zurückliefert. Eine empfangene Nachricht wird im Puffer `msg` mit der Größe `bytes` gespeichert. Beim letzten formalen Parameter kann ein Zeiger auf die *_msg_info* Struktur übergeben werden, in der zusätzliche Informationen über diese Nachricht gespeichert wird. Falls keine zusätzlichen Informationen über die Nachricht gespeichert werden soll, wird der Wert NULL übergeben. Die Funktion `MsgReceive()` liefert im Erfolgsfall eine sogenannte Receive ID zurück. Sie ist vom Datentyp *int* und kann entweder einen positiven Wert größer null oder den Wert null annehmen. Wenn die Receive ID gleich null ist, bedeutet es, dass der Server eine sogenannte *„Pulse“* empfangen hat. Dieser Begriff wird später im Abschnitt der asynchronen Kommunikation näher erläutert. Wenn die Receive ID größer null ist, bedeutet es, dass der Server eine Nachricht empfangen hat, die er beantworten muss. Im Fehlerfall liefert die Funktion `MsgReceive()` den Wert -1 zurück [1] [2].

Sobald der Server-Thread die Nachricht des Client-Threads fertig bearbeitet hat, sendet er mit der QNX-Funktion `MsgReply()` seine Antwort auf die Nachricht des Client-Threads (Siehe Listing 7).

Listing 7: QNX-Funktion `MsgReply()`

```
int MsgReply (int rcvid, int status, const void* msg, int size);
```

Dem ersten formalen Parameter wird die Receive ID übergeben, die die QNX-Funktion `MsgReceive()` zurückliefert. Über die Receive ID `rcvid` wird die jeweilige Antwort der zugehörigen Nachricht zugeordnet [1]. Der formale Parameter `status` ist der Rückgabewert von dem `MsgSend()`, die die Nachricht gesendet hat, die nun mit `MsgReply()` beantwortet wird. Die Antwort, die der Server-Thread senden möchte, wird dem formalen Parameter `msg` mit der Größe `size` in Bytes übergeben. Der Rückgabewert von `MsgReply()` ist vom Datentyp *int*. Diese QNX-Funktion liefert im Fehlerfall den Wert -1 zurück, ansonsten wurde sie fehlerfrei ausgeführt [1] [2].

Sobald der Client-Thread eine Antwort auf seine gesendete Nachricht erhalten hat, wechselt er vom Zustand *REPLY blocked* in den Zustand *READY*. Die Blockierung

seiner Ausführung, die durch den Aufruf von `MsgSend()` ausgelöst wurde, wird aufgehoben und der Client-Thread kann weiter arbeiten.

Die nachfolgende Abbildung verdeutlicht noch einmal die Interaktion zwischen den QNX-Funktionen `MsgSend()`, `MsgReceive()` und `MsgReply()`:

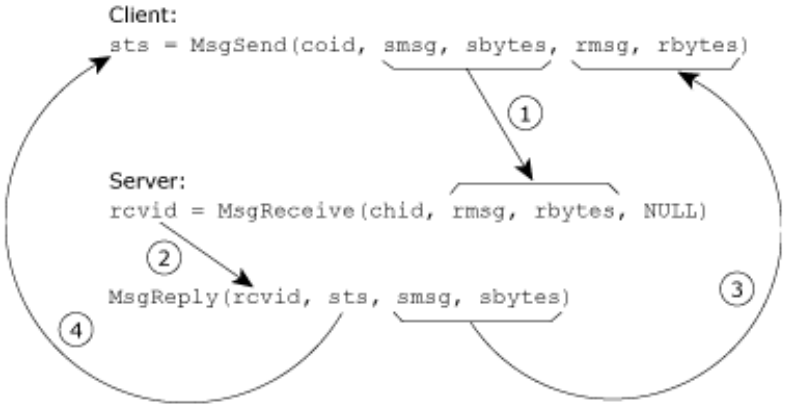

Abb. 4: Interaktion zwischen `MsgSend()`, `MsgReceive()` und `MsgReply()`

Bei der synchronen Kommunikation besteht die Gefahr, dass ein Deadlock auftreten kann, wenn Threads zweier Prozesse sich gegenseitig eine Nachricht zusenden (Siehe Abbildung 5).

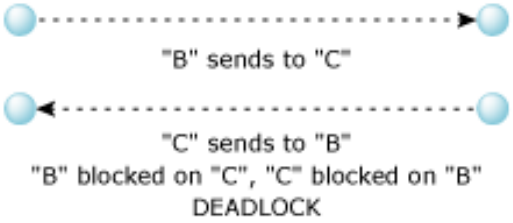

Abb. 5: Auftreten eines Deadlocks bei synchroner Kommunikation

Wie die Abbildung 5 zeigt, sendet ein Thread von Prozess B mit der QNX-Funktion `MsgSend()` synchron eine Nachricht an einen Thread von Prozess C. Zeitgleich sendet auch der Thread von Prozess C mit der QNX-Funktion `MsgSend()` eine synchrone Nachricht an den Thread von Prozess B. Die Ausführung beider Threads der Prozesse B und C sind blockiert (Zustand *SEND blocked*). Die Threads der beiden Prozesse warten nun auf die Antwort des Anderen. Dies hat zur Folge, dass die beiden Threads keine Möglichkeit haben, dem Anderen auf ihre Nachricht zu antworten. Somit ist bei dieser Kommunikation ein Deadlock aufgetreten [1] [2] [4].
Um einen Deadlock bei der synchronen Kommunikation zu vermeiden, müssen die Threads der Prozesse nach der sogenannten „*send-hierarchy*“ kommunizieren (Siehe Abbildung 6).

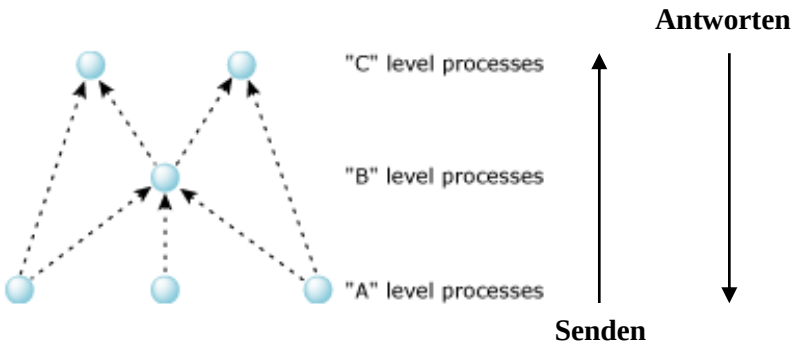

Abb. 6: Die „*send-hierarchy*“

Die Send-Hierarchie besagt, dass das synchrone Senden von Nachrichten nur in eine Richtung verlaufen darf. Die Antwort auf diese Nachricht verläuft in eine andere Richtung.
Wie die Abbildung 6 zeigt, können zum Beispiel Threads der Prozesse, die sich auf der Ebene „A“ befinden, synchron Nachrichten an die Threads der Prozesse, die sich auf der Ebene „B“ und „C“ befinden, schicken. Die Threads der Prozesse auf der Ebene „B“ und „C“ können auf die Nachrichten der Threads der Prozesse auf der Ebene „A“ antworten.
Niemals dürfen Threads der Prozesse auf der Ebene „C“ synchron Nachrichten an die Threads der Prozesse auf der Ebene „A“ und „B“ senden. Auch ein Thread eines Prozesses auf der Ebene „C“ darf nicht synchron Nachrichten an andere Threads auf derselben Ebene senden.
Die Prozesse der Ebene „A“ betrachten die Prozesse auf der Ebene „B“ und „C“ als ihre Server. Die Prozesse auf der Ebene „B“ betrachten die Prozesse auf der Ebene „A“ als ihre Clients und die Prozesse auf der Ebene „C“ als ihre Server [1] [2].
Zusammenfassend kann man feststellen, dass Clients synchron Nachrichten an die Server senden können. Die Server sollten allerdings keine synchronen Nachrichten an die Clients senden, um einen Deadlock zu vermeiden.
Mit der Send-Hierarchie wird das Problem mit dem Deadlock behoben. Jedoch besteht immer noch das Problem, dass Threads zweier Prozesse sich nicht gegenseitig Nachrichten zusenden können, ohne dass ein Deadlock auftritt.
Dieses Problem wird mit der QNX-Funktion `MsgDeliverEvent()` behoben, die im Abschnitt der asynchronen Kommunikation beschrieben wird.

Asynchrone Kommunikation über *Pulses*

Unter dem QNX Betriebssystem wird die asynchrone Kommunikation durch das Versenden von sogenannten „*Pulses*“ realisiert. Ein *Pulse* ist eine sehr kleine Nachricht, die 40 Bits an Nutzdaten aufnehmen kann. Diese 40 Bits sind in 8-Bit Code und 32-Bit Daten aufgeteilt. Die 8-Bit Code wird benötigt, um den Typ des Pulses eindeutig zu identifizieren. Ein Pulse kann auch als eine Pulse-Nachricht bezeichnet werden.

Die Struktur eines Pulses sieht wie folgt aus (Siehe Listing 8):

Listing 8: Struktur einer Pulse-Nachricht

```
struct _pulse {
        _Uint16t        type;
        _Uint16t        subtype;
        _Int8t          code;
        _Uint8t         zero[3];
        union sigval    value;
        _Int32t         scoid;
};
```

Listing 9: Struktur vom Variablen `value` aus der Struktur `_pulse`

```
union sigval {
   int      sival_int;
   void     *sival_ptr;
};
```

Wenn Programmierer eigene Pulse-Nachrichten definieren möchten, dürfen sie für die Variable `code` nur Werte zwischen `_PULSE_CODE_MINAVAIL` (0) und `_PULSE_CODE_MAXAVAIL` (127) vergeben. Die negativen Werte, die bis -128 gehen, sind vom Kernel reserviert [2] [6].

Eine Pulse-Nachricht wird über die nachfolgenden QNX-Funktionen gesendet (Siehe Listing 10):

Listing 10: QNX-Funktionen zum Versenden von Pulse-Nachrichten

```
int MsgSendPulse (int coid, int priority, int code, int value);
int MsgDeliverEvent (int rcvid, const struct sigevent* event);
```

Im Folgenden werden diese Funktionen näher erläutert.

MsgSendPulse()

Dem formalen Parameter `coid` wird die Connection ID des Channels übergeben, an den eine Pulse-Nachricht gesendet wird [1]. Mit dem formalen Parameter `priority` wird die Priorität der Pulse-Nachricht angegeben. Dem formalen Parameter `code` wird der 8-Bit Pulse-Code übergeben. Dem formalen Parameter `value` werden die 32-Bit Daten übergeben, die gesendet werden sollen. Der Rückgabewert der Funktion `MsgSendPulse()` ist vom Datentyp *int*. Konnte diese Funktion nicht fehlerfrei ausgeführt werden, liefert sie den Wert -1 zurück [1] [2].

MsgDeliverEvent()

Angenommen, ein Client möchte von einem Server benachrichtigt werden, wenn neue Daten beim Server vorhanden sind.

Der Client sendet dem Server synchron eine Nachricht mit der Bitte, ihn zu benachrichtigen, wenn neue Daten verfügbar sind. Da der Client nicht blockiert werden möchte, bestätigt der Server umgehend dem Client, dass er seine Nachricht erhalten hat. Der Client und der Server arbeiten parallel weiter. Beim Server sind nun neue Daten vorhanden, er möchte den Client darüber benachrichtigen. Wenn der Server synchron eine Nachricht an den Client sendet, um ihn zu informieren, wird die Send-Hierarchie verletzt. Es besteht die Gefahr, dass ein Deadlock auftritt, wenn der Client zur selben Zeit auch eine Nachricht an den Server sendet.
Die Frage ist nun, wie der Server dem Client eine Nachricht senden kann, ohne dabei die Send-Hierarchie zu verletzen [2] [6]. Die Antwort liegt in der Verwendung der QNX-Funktion `MsgDeliverEvent()`.
Die nachfolgende Abbildung zeigt, wie diese QNX-Funktion sinnvoll einzusetzen ist:

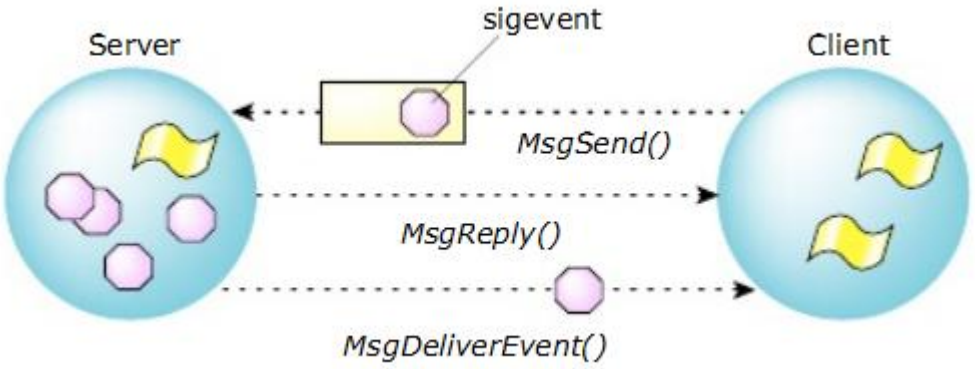

Abb. 7: Kommunikationsablauf zwischen Client und Server mit `MsgDeliverEvent()`

Zu Beginn definiert der Client seine eigene *sigevent* Struktur. In dieser Struktur legt der Client fest, wie er auf die Benachrichtigung des Servers über ein Ereignis reagieren wird. Es gibt verschiedene Möglichkeiten, eine *sigevent* Struktur zu definieren beziehungsweise diese zu initialisieren. Durch die Initialisierung wird bestimmt, wie die Benachrichtigung erfolgen soll. Für die oben beschriebene Situation initialisiert der Client seine *sigevent* Struktur mit dem QNX-Makro `SIGEV_PULSE_INIT()`, um über eine Pulse-Nachricht benachrichtigt zu werden (Siehe Listing 11).

Listing 11: Initialisierung einer *sigevent* Struktur mit `SIGEV_PULSE_INIT()`

```
SIGEV_PULSE_INIT(&event, coid, priority, code, value)
```

Der Übergabeparameter `event` ist die *sigevent* Struktur, die initialisiert wird. Die Definition der Parameter `coid`, `priority`, `code` und `value` entspricht die der Parameter von der Funktion `MsgSendPulse()` [2].

Wie die Abbildung 7 zeigt, sendet der Client mit `MsgSend()` seine initialisierte *sigevent* Struktur in einer Nachricht synchron an den Server. Der Server soll diese *sigevent* Struktur nutzen, um den Client zu benachrichtigen, wenn neue Daten beim

Server vorhanden sind.
Der Server empfängt mit `MsgReceive()` die Nachricht vom Client, die unter anderem die *sigevent* Struktur enthält. Die Receive ID *rcvid*, die er als Rückgabewert von `MsgReceive()` erhalten hat, und die *sigevent* Struktur speichert er ab. Der Server bestätigt umgehend dem Client mit `MsgReply()`, dass er seine Nachricht empfangen hat. Der Client läuft dann parallel zum Server weiter. Sobald neue Daten beim Server vorhanden sind, ruft er die QNX-Funktion `MsgDeliverEvent()` auf, um den Client darüber zu benachrichtigen. Dabei übergibt der Server die gespeicherte *rcvid* und die *sigevent* Struktur als Übergabeparameter an die Funktion `MsgDeliverEvent()`.
Durch den Aufruf von `MsgDeliverEvent()` löst der Server das Event aus, das in der *sigevent* Struktur vom Client beschrieben wurde – der Client empfängt als Benachrichtigung eine Pulse-Nachricht. Der Client wertet die Pulse-Nachricht bezüglich des Pulse-Codes und der Pulse-Daten aus. Durch die Auswertung der Pulse-Nachricht weiß der Client, wie er reagieren soll, beziehungsweise welche Aktionen er entsprechend ausführen muss.
In diesem Fall weiß der Client, dass neue Daten beim Server verfügbar sind. Er sendet synchron eine Nachricht an den Server, mit der Bitte ihm die neuen Daten zu liefern. Der Server antwortet dann dem Client mit den neuen Daten [2] [6].

Die Send-Hierarchie wird bei diesem Ansatz nicht verletzt [1], da die Ausführung des Servers nicht durch die QNX-Funktion `MsgDeliverEvent()` blockiert wird. Wenn der Client dem Server synchron eine Nachricht sendet, während der Server die QNX-Funktion `MsgDeliverEvent()` aufruft, kann er nach diesem Aufruf die Nachricht vom Client empfangen und diese beantworten. Daher ist ein Auftreten eines Deadlocks hier nicht möglich.

Um eine Verbindung zu einem Server, die mit der QNX-Funktion `name_open()` geöffnet wurde, zu schließen, wird die QNX-Funktion `name_close()` aufgerufen (Siehe Listing 12) [2].

Listing 12: QNX-Funktion `name_close()`

```
int name_close (int coid);
```

Dem formalen Parameter `coid` wird die Connection ID übergeben, die als Rückgabewert von `name_open()` zurückgeliefert wurde.
Die Funktion `name_close()` liefert bei fehlerfreier Ausführung den Wert null zurück, ansonsten den Wert -1 [2].

Die QNX-Funktion `name_close()` wurde in einer Funktion des APIs lib_ipc.h verwendet. Auf Wunsch des Unternehmens soll diese QNX-Funktion nicht unter Linux abgebildet werden, da die Verbindungen zu den Servern bestehen bleiben soll.

2.2 Beschreibung der in der IPC-Bibliothek verwendeten POSIX-Funktionen

In diesem Kapitel werden die POSIX-Funktionen vorgestellt, die zur Behandlung von shared memory Bereichen benötigt werden. Im Folgenden werden diese POSIX-Funktionen beschrieben:

- `shm_open()`
- `ftruncate()`
- `mmap()`
- `memset()`
- `munmap()`
- `close()`
- `shm_unlink()`

Die POSIX-Funktion `shm_open()` hat zwei Funktionalitäten (Siehe Listing 13):

- Existiert noch kein shared memory Objekt im QNX Betriebssystem, erzeugt die Funktion `shm_open()` ein neues shared memory Objekt.

- Existiert bereits ein shared memory Objekt im QNX Betriebssystem, öffnet die Funktion `shm_open()` dieses Objekt.

Listing 13: POSIX-Funktion zum Öffnen/Erzeugen eines shared memory Objekts

```
int shm_open (const char* name, int oflag, mode_t mode);
```

Dem ersten formalen Parameter `name` wird der Name des shared memory Objekts übergeben, das geöffnet oder erzeugt werden soll. Über den formalen Parameter `oflag` kann bestimmt werden, ob ein neues shared memory Objekt erzeugt oder ein bereits existierendes shared memory Objekt geöffnet werden soll. Dafür werden folgende Flags verwendet:

Tabelle 1: Flags für die POSIX-Funktion `shm_open()`

Flag	Bedeutung
O_RDONLY	Öffnen des Bereiches nur zum Lesen
O_RDWR	Öffnen des Bereiches zum Lesen und Schreiben
O_CREAT	Erzeugung eines neuen Bereiches

Über den formalen Parameter `mode` können die Zugriffsrechte für das shared memory Objekt bestimmt werden. Dieser Parameter kommt erst zur Geltung, wenn das Flag `O_CREAT` im `oflag` gesetzt ist. Die Funktion `shm_open()` liefert im Erfolgsfall einen Filedeskriptor vom Datentyp *int* zurück, der mit dem shared memory Objekt verknüpft ist. Im Fehlerfall liefert sie den Wert -1 zurück [2] [8].

Da die Größe eines neu erzeugten shared memory Objekts null ist, muss die Größe für dieses Objekt noch angepasst werden [9]. Dafür wird die POSIX-Funktion `ftruncate()` aufgerufen:

Listing 14: POSIX-Funktion zur Anpassung der Größe des shared memory Objekts

```
int ftruncate (int fildes, off_t length);
```

Dem formalen Parameter `fildes` wird der Filedeskriptor übergeben, der mit dem neu erzeugten shared memory Objekt verknüpft ist. Über den formalen Parameter `length` wird die Größe des Objektes in Bytes angegeben. Wurde die Funktion `ftruncate()` fehlerfrei ausgeführt, liefert sie den Wert null zurück. Im Fehlerfall liefert sie den Wert -1 zurück [2].

Die POSIX-Funktion `mmap()` bildet einen bestimmten Bereich des shared memory Objekts in den Adressraum eines Prozesses ab (Siehe Listing 15) [2].
Die nachfolgende Abbildung veranschaulicht, wie ein Bereich dieses Objekts im Adressraum eines Prozesses abgebildet wird [9]:

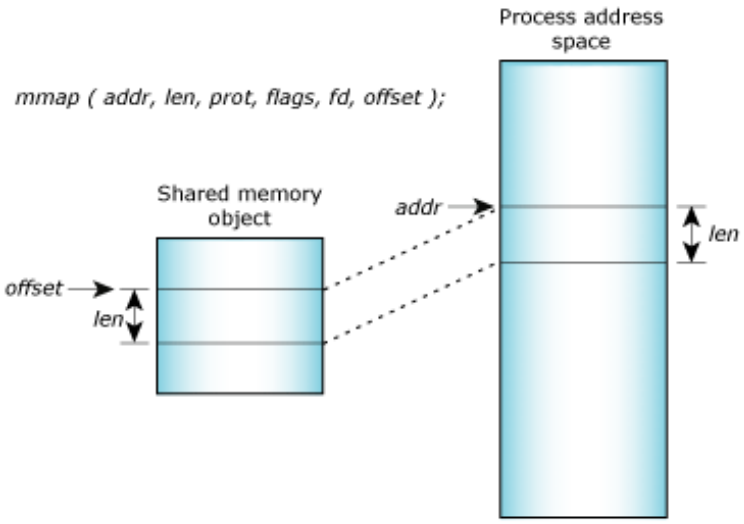

Abb. 8: Abbildung eines Bereiches des shared memory Objektes in dem Adressraum eines Prozesses

Listing 15: POSIX-Funktion zur Abbildung eines Bereiches des shared memory Objekts in dem Adressraum eines Prozesses

```
void* mmap (void* addr, size_t len, int prot, int flags, int fildes, off_t off);
```

Wie die Abbildung 8 zeigt, kann der Programmierer über den formalen Parameter `addr` bestimmen, wo er das shared memory Objekt im Adressraum eines Prozesses abbilden möchte. Üblicherweise wird für diesen Parameter der Wert NULL übergeben. Durch diese Übergabe wählt dann der Kernel die Adresse aus, wo er mit der Abbildung dieses Objektes im Adressraum beginnen will. Über den formalen Para-

meter `off` wird der Anfang des Bereiches des shared memory Objektes bestimmt, der mit der Größe `len` im Adressraum des Prozesses abgebildet werden soll. Die Größe wird in Bytes angegeben. Über den formalen Parameter `prot` kann bestimmt werden, wie der Zugriff auf den abgebildeten shared memory Bereich erfolgen soll. Der formale Parameter `flags` bestimmt die Handhabung des abgebildeten shared memory Bereichs. Zum Beispiel kann über diesen Parameter bestimmt werden, ob dieser Bereich mit anderen Prozessen geteilt wird oder nicht. Dem formalen Parameter `fildes` wird der Filedeskriptor übergeben, der mit dem shared memory Objekt verknüpft ist.
Die Funktion `mmap()` liefert im Erfolgsfall die Adresse vom abgebildeten shared memory Bereich zurück, über die auf diesen Bereich lesend oder schreibend zugegriffen werden kann.
Im Fehlerfall liefert sie den Wert der Konstante `MAP_FAILED` zurück [2] [9] [10].

Mit der POSIX-Funktion `memset()` kann der abgebildete shared memory Bereich mit einem bestimmten Wert initialisiert werden (Siehe Listing 16) [2].

Listing 16: POSIX-Funktion zur Initialisierung eines abgebildeten shared memory Bereichs

```
void* memset (void* dst, int c, size_t length);
```

Dabei wird dem formalen Parameter `dst` der Zeiger übergeben, der die Adresse vom abgebildeten shared memory Bereich enthält. An dieser Adresse soll die Initialisierung mit dem Wert, den der formale Parameter `c` enthält, beginnen. Über den formalen Parameter `length` kann bestimmt werden, wie viel Bytes des abgebildeten shared memory Bereiches mit diesem Wert ab `dst` initialisiert werden soll [2].

Wird der shared memory Bereich im Adressraum des Prozesses nicht mehr genutzt, kann dieser mit der POSIX-Funktion `munmap()` gelöscht werden (Siehe Listing 17) [2].

Listing 17: POSIX-Funktion zum Löschen eines abgebildeten shared memory Bereiches

```
int munmap (void* addr, size_t len);
```

Dem formalen Parameter `addr` wird die Adresse übergeben, wo der abgebildete shared memory Bereich beginnt. Dem formalen Parameter `len` wird die Größe dieses Bereiches in Bytes übergeben. Wenn diese Funktion fehlerfrei ausgeführt wurde, liefert sie den Wert null zurück. Im Fehlerfall liefert sie den Wert -1 zurück [2].

Wird der Filedeskriptor, der mit dem shared memory Objekt verknüpft ist, nicht mehr benötigt, kann dieser mit der POSIX-Funktion `close()` geschlossen werden (Siehe Listing 18) [2].

Listing 18: POSIX-Funktion zum Schließen eines Filedeskriptors

```
int close (int filedes);
```

Dem formalen Parameter `filedes` wird der Filedeskriptor übergeben, der geschlossen werden soll. Die Funktion `close()` liefert im Erfolgsfall den Wert null zurück, ansonsten im Fehlerfall den Wert -1 [2].

Soll das shared memory Objekt gelöscht werden, wird die POSIX-Funktion `shm_unlink()` aufgerufen [2]:

Listing 19: POSIX-Funktion zum Löschen eines shared memory Objekts

```
int shm_unlink (const char* name);
```

Dem formalen Parameter `name` wird der Name des shared memory Objektes übergeben, das gelöscht werden soll. Wenn die Funktion `shm_unlink()` fehlerfrei ausgeführt wurde, liefert sie den Wert null zurück. Im Fehlerfall liefert sie den Wert -1 zurück [2].

2.3 Client / Server Kommunikationsmodell

Die Programme aus der firmeneigenen Software übernehmen sowohl die Rolle eines Clients als auch die Rolle eines Servers. Damit ein Programm beide Rollen ausführen kann, wird neben dem Hauptthread des Programms noch ein eigenständiger Thread innerhalb des Programms benötigt. Dieser Thread wird als sogenannter Client-Thread erzeugt, der parallel zum Hauptthread des Programms gestartet wird. Der Hauptthread des Programms fungiert als sogenannter Server-Thread.
Der Server-Thread macht sich mit seinem Namen im System bekannt und kann von anderen Programmen angesprochen werden. Er legt einen oder mehrere shared memory Bereiche an, über die Daten ausgetauscht werden können (Siehe Abbildung 9). Die Namen dieser Speicherbereiche, die bei der Erzeugung vergeben werden, sind in dem API lib_ipc.h definiert.

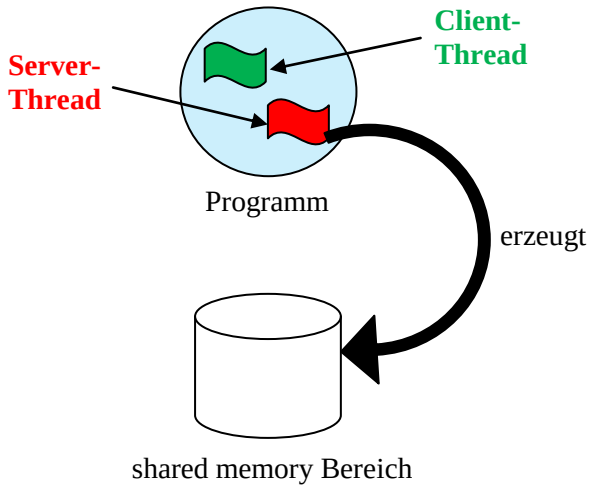

Abb. 9: Server-Thread erzeugt einen shared memory Bereich

Der Server-Thread verwaltet eine Liste von Client-Threads, die sich bei ihm anmelden, um über bestimmte Ereignisse informiert zu werden. Ereignisse können zum Beispiel sein, wenn neue Daten im shared memory Bereich existieren oder wenn bestehende Daten im shared memory Bereich aktualisiert werden.
Möchte ein Client-Thread sich beim Server-Thread anmelden, sendet er synchron seine Daten in einer Nachricht an den Server-Thread. Diese Daten werden von ihm benötigt, um ihn über Ereignisse informieren zu können. Außerdem erwähnt der Client-Thread in dieser Nachricht, über welches Ereignis er informiert werden möchte. Der Client-Thread wartet auf eine Bestätigung vom Server-Thread, dass seine Nachricht bei ihm angekommen ist. Der Server-Thread nimmt die Anmeldedaten des Client-Threads in seine Liste auf und bestätigt ihm mit einer Antwort, dass er seine Anmeldedaten erhalten hat. Der Client-Thread empfängt die Bestätigung vom Server-Thread und kann weiterarbeiten.
Die nachfolgende Abbildung veranschaulicht den Ablauf, wie ein Client-Thread sich beim Server-Thread anmeldet:

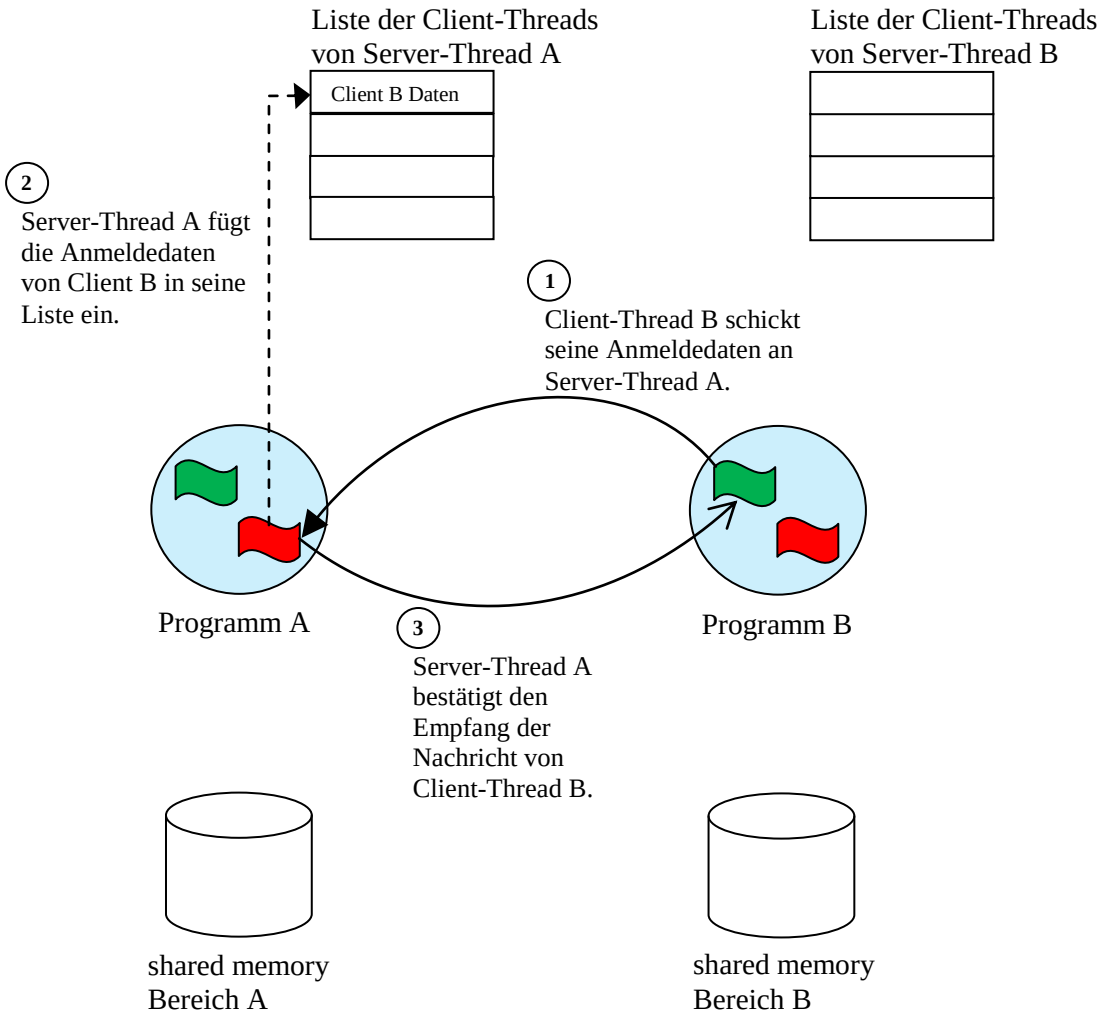

Abb. 10: Client-Thread B meldet sich beim Server-Thread A an

Sobald neue Daten im shared memory Bereich des Server-Threads vorhanden sind, werden nur die angemeldeten Client-Threads aus der Liste des Server-Threads informiert, die über die Aktualisierung dieser Daten benachrichtigt werden wollen. Die anderen angemeldeten Client-Threads, die sich nicht für die Aktualisierung dieser Daten interessieren, werden nicht informiert.
Der Client-Thread, der parallel zum Server-Thread läuft, benachrichtigt die angemeldeten Client-Threads über dieses Ereignis. Dieser Client-Thread nutzt die Daten der angemeldeten Client-Threads aus der Liste, um sie asynchron über dieses Ereignis zu benachrichtigen. Die angemeldeten Client-Threads erhalten diese Benachrichtigung und lesen die neuen Daten aus dem shared memory Bereich des Server-Threads aus. Sie verarbeiten diese Daten und schreiben die neu verarbeiteten Daten in ihrem shared memory Bereich. Sie informieren ihrerseits dann die Client-Threads, die auf die Aktualisierung dieser Daten warten.
Die nachfolgende Abbildung zeigt, wie ein Client-Thread über ein Ereignis informiert wird:

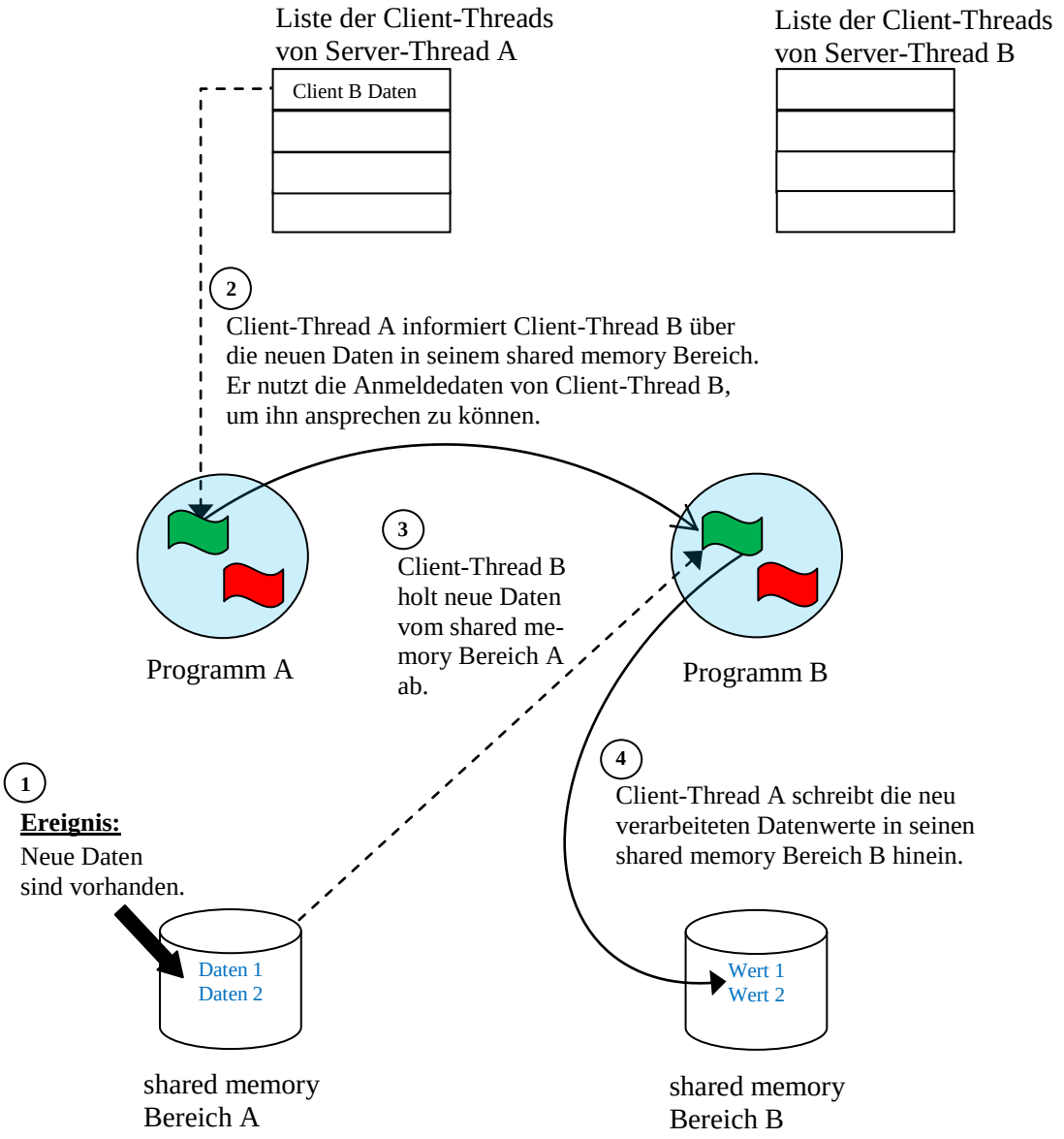

Abb. 11: Client-Thread B wird über das Vorhandensein neuer Daten benachrichtigt

Möchte ein Client-Thread nicht mehr über bestimmte Ereignisse informiert werden, meldet er sich beim Server-Thread ab. Dabei sendet er nochmals synchron seine Daten in einer Nachricht an den Server-Thread, damit der Server-Thread anhand dieser Daten den richtigen Client-Thread aus seiner Liste entfernt. Der Client-Thread wartet auf eine Bestätigung vom Server-Thread, dass seine Nachricht bei ihm angekommen ist. Sobald der Server-Thread den Client-Thread aus seiner Liste entfernt hat, bestätigt er ihm mit einer Antwort, dass er seine Nachricht erhalten hat. Der abgemeldete Client-Thread wird zukünftig nicht mehr über bestimmte Ereignisse informiert.
Die nachfolgende Abbildung veranschaulicht den Ablauf, wie ein Client-Thread sich bei einem Server-Thread abmeldet:

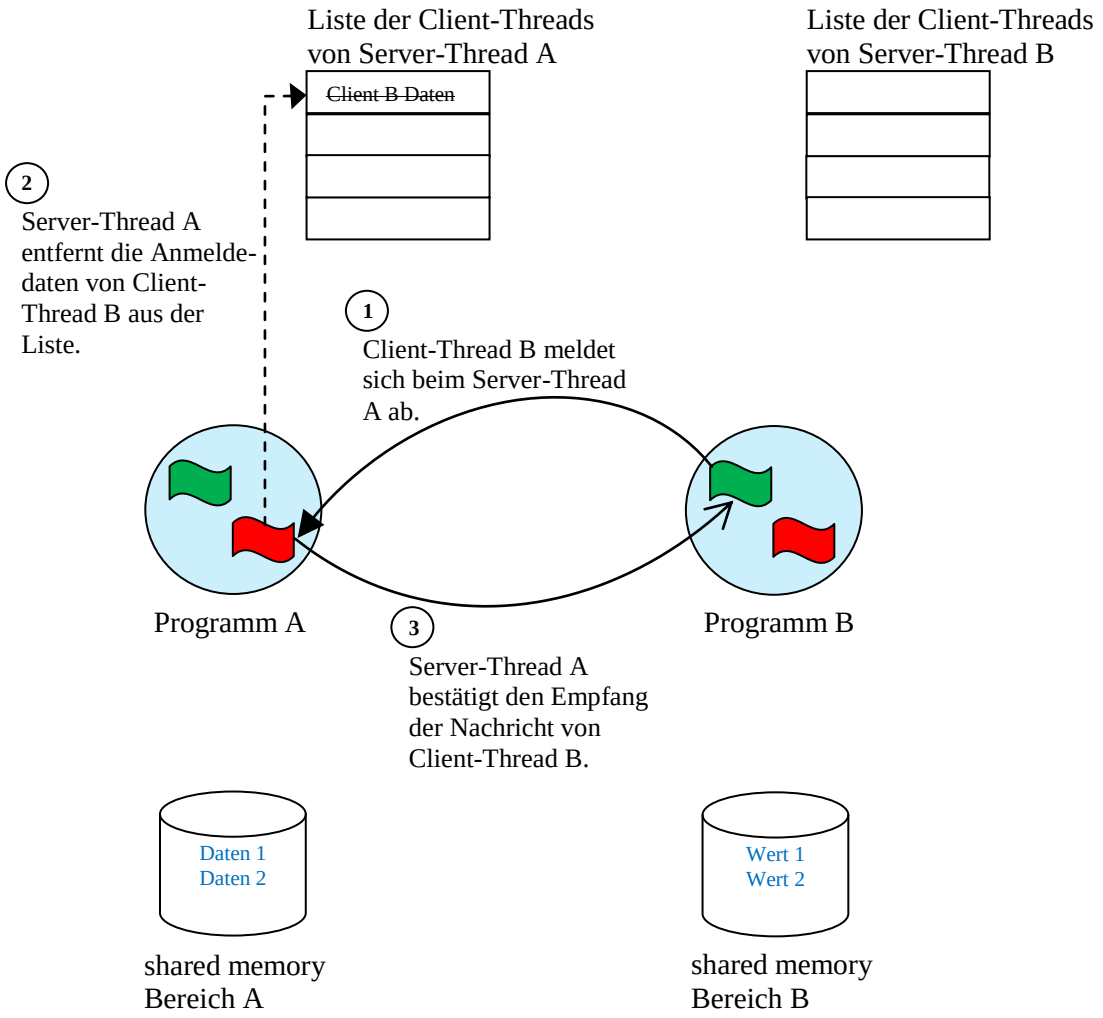

Abb. 12: Client-Thread B meldet sich beim Server-Thread A ab

Der Verlauf dieser Kommunikationsstruktur zwischen dem Client-Thread und dem Server-Thread ähnelt dem, der im Kapitel 2.1 unter dem Abschnitt *„MsgDeliverEvent()“* beschrieben wurde.

Nach dem Konzept von QNX benachrichtigt der Server die angemeldeten Clients aus seiner Liste, wenn ein Ereignis bei ihm aufgetreten ist. In der Kommunikationsstruktur, die in der Firmensoftware stattfindet, wurde das Konzept von QNX bezüglich der Benachrichtigung der angemeldeten Clients ein wenig modifiziert. Hier benachrichtigt der Client, der parallel zum Server in einem Programm läuft, die angemeldeten Clients aus der Liste des Servers, wenn beim Server ein Ereignis auftritt.

2.4 Beschreibung der Funktionen der IPC-Bibliothek

In der Datei **lib_ipc.c** existiert ein eindimensionales Array `ipc_task[]`, das Elemente vom Typ *struct _TASK_IPC_TYPE_T* enthält (Siehe Listing 20). Dieses Array ist ein wichtiger Bestandteil für die Kommunikation zwischen Programmen sowie für die Behandlung von shared memory Bereichen.

Listing 20: lib_ipc.c Auszug aus dem Array `ipc_task[]`

```
static struct _TASK_IPC_TYPE_T ipc_task[] = {
   {"????????",          -1, 0, {{-1, NULL},          {-1, NULL}, {-1, NULL}, …}},
   {IO_TASK,             -1, 0, {{-1, IO_SHM},        {-1, NULL}, {-1, NULL}, …}},
                                …
   {ABLA_NAME,       -1, 0, {{-1, ABLA_SHM},      {-1, NULL}, {-1, NULL}, …}},
   {DBASE_NAME,      -1, 0, {{-1, DBASE_SHM},     {-1, NULL}, {-1, NULL}, …}},
                                …
   {TIME_NAME,       -1, 0, {{-1, NULL},          {-1, NULL}, {-1, NULL}, …}},
   {USER_NAME,       -1, 0, {{-1, NULL},          {-1, NULL}, {-1, NULL}, …}},
   {SCALE_NAME,      -1, 0, {{-1, SCALE_SHM},     {-1, NULL}, {-1, NULL}, …}},
                                …
   {MONCHK_NAME,-1, 0, {{-1, NULL},          {-1, NULL}, {-1, NULL}, …}}
};
```

Die Struktur *struct _TASK_IPC_TYPE_T* besteht aus einem *char*-Array, zwei *int*-Variablen und einem Array vom Typ *struct _TASK_IPC_SHM_TYPE_T* (Siehe Listing 21).

Listing 21: lib_ipc.h Typ eines Elements des Arrays `ipc_task[]`

```
typedef struct _TASK_IPC_TYPE_T
{
        char name[IPC_NAME_LENGTH + 1];
        int task_fd;
        int in_use;
        struct _TASK_IPC_SHM_TYPE_T shm_array[MAX_SHM];
}
TASK_IPC;
```

Die Struktur *struct _TASK_IPC_SHM_TYPE_T* ist aus einer *int*-Variablen und einem *char*-Zeiger zusammengesetzt (Siehe Listing 22).

Listing 22: lib_ipc.h Typ eines Elements des Arrays `shm_array[]`

```
typedef struct _TASK_IPC_SHM_TYPE_T
{
        int shm_fd;
        char *shm_name;
}
SHM_TYPE;
```

Das *char*-Array `name[]` aus der Struktur *_TASK_IPC_TYPE_T* enthält den Namen eines jeweiligen Programms aus der firmeneigenen Software. Die Variable `task_fd` ist mit dem Wert -1 und die Variable `in_use` mit dem Wert null initialisiert. Die Variable `shm_fd` und der *char*-Zeiger `shm_name` aus dem Array `shm_array[]` sind mit dem Wert -1 und NULL initialisiert.

Die nachfolgenden Funktionen sind Funktionen aus dem API lib_ipc.h. Funktionen, die speziell für bestimmte Programme implementiert wurden, werden hier nicht weiter betrachtet.

Listing 23: Funktion zum Öffnen einer Verbindung zu einem Server

```
int wait_for_task (unsigned int target_id, unsigned int wait);
```

Die Funktion `wait_for_task()` öffnet eine Verbindung zu einem Server, der im Array `ipc_task[]` eingetragen ist. Diese Funktion wird von Clients aufgerufen.
Sie enthält zwei formale Parameter `target_id` und `wait`, die vom Datentyp *unsigned int* sind. Der Rückgabewert von `wait_for_task()` ist vom Datentyp *int*.
Über den formalen Parameter `target_id` wird derjenige Server aus `ipc_task[]` angesprochen, zu dem ein Client eine Verbindung mit der QNX-Funktion `name_open()` öffnen möchte. Ist die Verbindung zu diesem Server bereits geöffnet, wird die Funktion `wait_for_task()` erfolgreich verlassen.
Der formale Parameter `wait` enthält die maximale Wartezeit in Sekunden. Der Client versucht in dieser Zeit eine Verbindung zum Server herzustellen. Schlägt der Verbindungsaufbau zu ihm fehl, wartet der Client eine Sekunde ab und versucht erneut, eine Verbindung zu ihm zu öffnen. Wenn die Wartezeit `wait` überschritten ist und der Client keine Verbindung zum Server öffnen konnte, liefert die Funktion `wait_for_task()` als Rückgabewert einen Fehler zurück. Konnte er innerhalb der Wartezeit `wait` eine Verbindung zum Server öffnen, wird der Rückgabewert von der QNX-Funktion `name_open()` in die Variable `task_fd`, die sich an der Position `target_id` im Array `ipc_task[]` befindet, gespeichert.

Listing 24: Funktionen zum In- und Dekrementieren des Zählers `in_use`

```
void ipc_inc_connection (unsigned int target_id);
void ipc_dec_connection (unsigned int target_id);
```

Die Funktion `ipc_inc_connection()` zählt die Variable `in_use` eines Servers

aus dem Array `ipc_task[]` hoch. Diese Variable wird inkrementiert, wenn ein Client einen shared memory Bereich des Servers öffnet oder wenn er sich bei dem Server anmeldet.

Die Funktion `ipc_dec_connection()` verringert die Variable `in_use` eines Servers um eins. Sie wird dekrementiert, wenn ein Client einen shared memory Bereich des Servers schließt oder wenn er sich bei dem Server abmeldet.

Die beiden Funktionen erwarten als Übergabeparameter einen Wert vom Datentyp *unsigned int*, der die Position von `in_use` im Array `ipc_task[]` angibt. Beide Funktionen liefern keinen Wert zurück.

Die Variable `in_use` gibt an, ob Clients eine Verbindung zu einem Server noch aktiv nutzen. Erreicht `in_use` den Wert null, wird die Verbindung zum Server mit der QNX-Funktion `name_close()` geschlossen. Die Variable `task_fd` des Servers, die sich an der Position `target_id` im Array `ipc_task[]` befindet, wird auf -1 gesetzt.

Listing 25: Funktionen zur Behandlung von shared memory Bereichen

```
int ipc_get_shm_fd (unsigned int target_id, int *shm_fd, unsigned int shm_num);
void* ipc_shm_open (unsigned int target_id, long size, int *err, unsigned int shm_num);
int ipc_shm_close (void *addr, long size, unsigned int target_id, unsigned int shm_num);
void* ipc_shm_create (const char *name, long size, int *err);
```

ipc_get_shm_fd()

Über die Funktion `ipc_get_shm_fd()` wird der Filedeskriptor `shm_fd` aus dem Array `shm_array[]` geholt.

Der formale Parameter `target_id` gibt die Position vom `shm_array[]` im Array `ipc_task[]` an und ist vom Datentyp *unsigned int*. Der formale Parameter `shm_num` gibt die Position vom Filedeskriptor `shm_fd` im Array `shm_array[]` an. Der Datentyp dieses Parameters ist *unsigned int*. Der formale Parameter `shm_fd` ist ein Zeiger vom Datentyp *int*. Der Filedeskriptor `shm_fd`, der sich an der Position `shm_num` im Array `shm_array[]` befindet, wird in die Variable kopiert, auf die der Zeiger `shm_fd` verweist.

Die Funktion `ipc_get_shm_fd()` liefert einen Status zurück, ob die Funktion fehlerfrei ausgeführt wurde, und ist vom Datentyp *int*. Im Fehlerfall liefert sie als Rückgabewert einen Fehler zurück.

ipc_shm_open()

Die Funktion `ipc_shm_open()` öffnet einen von einem Server bereits erzeugten shared memory Bereich. Sie wird von Clients aufgerufen.

Der formale Parameter `target_id` ist der Index des Arrays `ipc_task[]`. Dieser gibt an, wo sich das Array `shm_array[]` im Array `ipc_task[]` befindet. Er wird auch als Übergabeparameter an die später aufrufenden Funktionen `wait_for_task()` und `ipc_inc_connection()` übergeben. Dieser Parameter ist vom Datentyp *unsigned int*. Der formale Parameter `size` enthält eine Größe in Bytes, die an die später aufrufende POSIX-Funktion `mmap()` übergeben wird. Dieser

Parameter ist vom Datentyp *long*. Der formale Parameter `err` ist ein Zeiger vom Datentyp *int*. Der Status, ob die Funktion `ipc_shm_open()` fehlerfrei oder fehlerhaft ausgeführt wurde, wird in die Variable kopiert, auf die `err` zeigt. Der formale Parameter `shm_num` gibt die Position des Filedeskriptors `shm_fd` und des *char*-Zeigers `shm_name` im Array `shm_array[]` an. Dieser Parameter ist vom Datentyp *unsigned int*. Der Rückgabewert von `ipc_shm_open()` ist ein *void*-Zeiger.
Zu Beginn der Funktion `ipc_shm_open()` wird die Funktion `wait_for_task()` mit dem Übergabeparameter `target_id` und dem Wert null aufgerufen. Sie überprüft, ob der Server, der sich an der Position `target_id` im Array `ipc_task[]` befindet, bereits läuft.
Läuft er bereits, öffnet – wie im Kapitel 2.2 beschrieben – die POSIX-Funktion `shm_open()` über den Namen `shm_name` das shared memory Objekt des Servers. Dieses Objekt wird mit der POSIX-Funktion `mmap()` in den Adressraum vom Client abgebildet. Dabei werden unter anderem der Rückgabewert von `shm_open()` und der formale Parameter `size` an diese POSIX-Funktion übergeben.
Der Rückgabewert von `shm_open()` wird im Erfolgsfall dem Filedeskriptor `shm_fd` zugewiesen, der sich an der Position `shm_num` im Array `shm_array[]` befindet. Wurde die POSIX-Funktion `mmap()` erfolgreich ausgeführt, wird die Variable `in_use` des Servers mit der Funktion `ipc_inc_connection()` inkrementiert.
Die Funktion `ipc_shm_open()` liefert im Erfolgsfall den Rückgabewert von `mmap()` zurück, ansonsten den Wert NULL. Mit dem Rückgabewert von `ipc_shm_open()` beziehungsweise von `mmap()` kann dann auf den abgebildeten shared memory Bereich zugegriffen werden.

ipc_shm_close()

Die Funktion `ipc_shm_close()` schließt einen Filedeskriptor, der mit dem shared memory Objekt eines Servers verknüpft ist.
Der *void*-Zeiger `addr` enthält den aus der POSIX-Funktion `mmap()` resultierenden *void*-Zeiger. Der formale Parameter `size` gibt die Größe in Bytes an und wird als Übergabeparameter an die später aufrufende POSIX-Funktion `munmap()` übergeben. Dieser Parameter ist vom Datentyp *long*. Der formale Parameter `target_id` ist vom Datentyp *unsigned int*. Dieser Parameter wird als Übergabeparameter an die später aufrufenden Funktionen `ipc_get_shm_fd()` und `ipc_dec_connection()` übergeben. Der formale Parameter `shm_num` ist vom Datentyp *unsigned int* und wird später als Übergabeparameter an `ipc_get_shm_fd()` übergeben. Der Rückgabewert dieser Funktion ist vom Datentyp *int*.
Bevor ein Filedeskriptor geschlossen wird, wird die POSIX-Funktion `munmap()` mit den Parametern `addr` und `size` aufgerufen. Sie soll den abgebildeten shared memory Bereich aus dem Adressraum löschen.
Über die Funktion `ipc_get_shm_fd()` wird ein Filedeskriptor geholt, der mit der POSIX-Funktion `close()` geschlossen werden soll. Mit den übergegebenen Parametern `target_id` und `shm_num` wird bestimmt, welcher Filedeskriptor zum Schließen geholt wird.

Wurden `munmap()` und `close()` fehlerfrei ausgeführt, wird die Variable `in_use` des Servers mit der Funktion `ipc_dec_connection()` dekrementiert. Der formale Parameter `target_id` gibt an, wo sich die Variable `in_use` im Array `ipc_task[]` befindet. Der Filedeskriptor `shm_fd` an der Position `shm_num` des Arrays `shm_array[]` wird auf -1 gesetzt und `ipc_shm_close()` wird erfolgreich verlassen. Im Fehlerfall liefert diese Funktion einen Fehler zurück.

ipc_shm_create()

Die Funktion `ipc_shm_create()` erzeugt ein neues shared memory Objekt, das in den Adressraum des Aufrufers abgebildet wird. Der Aufrufer ist in diesem Fall der Server.
Sie erhält als ersten Übergabeparameter den Namen `name`, nach dem das neue shared memory Objekt benannt wird. Existiert bereits ein shared memory Objekt unter diesem Namen, wird es mit der POSIX-Funktion `shm_unlink()` aus dem Betriebssystem QNX gelöscht. Mit dem formalen Parameter `size` wird sowohl die Größe des shared memory Objekts als auch die Größe des abgebildeten shared memory Bereichs bestimmt. Dieser Parameter ist vom Datentyp *long*. `err` ist ein Zeiger vom Datentyp *int*. Tritt während der Ausführung von `ipc_shm_create()` ein Fehler auf, wird in die Variable, auf die `err` zeigt, dieser Fehler kopiert. Der Rückgabewert der Funktion `ipc_shm_create()` ist ein *void*-Zeiger.
Wie im Kapitel 2.2 beschrieben, erzeugt die POSIX-Funktion `shm_open()` ein neues shared memory Objekt, dessen Größe die POSIX-Funktion `ftruncate()` mit dem Parameter `size` festlegt. Dieses Objekt wird mit der gleichen Größe `size` von der POSIX-Funktion `mmap()` in den Adressraum des Servers abgebildet. Somit wurde der gesamte Bereich des shared memory Objekts in den Adressraum des Servers abgebildet. Die POSIX-Funktion `memset()` initialisiert dann den abgebildeten shared memory Bereich mit dem Wert null.
Die Funktion `ipc_shm_create()` liefert im Erfolgsfall den Rückgabewert von `mmap()` zurück. Im Fehlerfall liefert sie den Wert NULL zurück.

Listing 26: Funktionen zum An- und Abmelden eines Clients bei einem Server

```
int ipc_target_atta_value (int target_id, int pulse_nr, int value, int *chid, int *coid);
int ipc_target_deta_value (int target_id, int pulse_nr, int value, int *coid);
```

ipc_target_atta_value()

Mit der Funktion `ipc_target_atta_value()` meldet sich ein Client bei einem Server an.
Der formale Parameter `target_id` ist ein Index des Arrays `ipc_task[]` und ist vom Datentyp *int*. Ein Client bestimmt über diesen Index, bei welchem Server aus dem Array `ipc_task[]` er sich anmelden möchte. Außerdem wird dieser Parameter an die später aufrufenden Funktionen `wait_for_task()` und `ipc_inc_connection()` übergeben. Dem formalen Parameter `pulse_nr` wird ein Pulse-Code übergeben, der als Konstante in dem API lib_ipc.h definiert ist. Dem formalen Parameter `value` werden Daten übergeben, die 32-Bit groß sind. Dem Zei-

ger `chid` wird die Adresse einer Variablen übergeben, die die Channel ID des Clients enthält. Enthält die Variable, auf die `chid` zeigt, den Wert -1, wird die QNX-Funktion `ChannelCreate()` aufgerufen. Der Variablen, auf die `chid` zeigt, wird der Rückgabewert von `ChannelCreate()` zugewiesen. Dem Zeiger `coid` wird die Adresse einer Variablen übergeben, in der die Connection ID des Clients steht. Enthält die Variable, auf die `coid` zeigt, den Wert -1, wird die QNX-Funktion `ConnectAttach()` aufgerufen. Dieser Variablen wird der Rückgabewert von `ConnectAttach()` zugewiesen.
Die Funktion `ipc_target_atta_value()` liefert einen Status zurück, ob die Funktion fehlerfrei oder fehlerhaft ausgeführt wurde. Der Rückgabewert dieser Funktion ist vom Datentyp *int*.
Diese Funktion enthält eine lokale Variable `atta_msg`, die vom Typ *struct _CLIENT_PULSE_TYPE_T* ist. Die Struktur besteht aus der *sigevent* Struktur `ev`, der *long*-Variablen `delay`, der Variablen `ch_nr` vom Datentyp *unsigned char* und einer weiteren Struktur, die aus den Variablen `target` und `cmd` zusammengesetzt ist.
Die Variablen `delay` und `ch_nr` sind nicht relevant für das Verständnis der Funktion `ipc_target_atta_value()`. Daher werden sie nicht weiter betrachtet.
Den Elementen dieser Struktur werden Daten vom Client zugewiesen, die an den Server geschickt werden. Diese Daten sind die sogenannten „Anmeldedaten“, die im Kapitel 2.3 erwähnt wurden.
Der Wert vom formalen Parameter `target_id` wird der Variablen `target` zugewiesen. Das Kommando `ATTA_CMD`, das als Konstante im lib_ipc.h definiert ist, wird der Variablen `cmd` zugewiesen. Dieses Kommando signalisiert einem Server, dass ein Client sich bei ihm anmelden möchte.
Die *sigevent* Struktur `ev` aus der Struktur *_CLIENT_PULSE_TYPE_T* wird – wie in der Definition der QNX-Funktion `MsgDeliverEvent()` im Kapitel 2.1 beschrieben – mit dem QNX-Makro `SIGEV_PULSE_INIT()` initialisiert. Dabei wird die *sigevent* Struktur `ev`, die Connection ID `*coid`, die Priorität des Pulses, der Pulse-Code `pulse_nr` und die Daten `value` an diesem Makro übergeben. Damit hat der Client festgelegt, dass er über eine Pulse-Nachricht benachrichtigt werden möchte, wenn ein Ereignis auftritt.
Bevor diese Daten an den Server geschickt werden, muss mit der Funktion `wait_for_task()` überprüft werden, ob der Server bereits läuft und eine Verbindung zu ihm geöffnet ist. Dabei wird der formale Parameter `target_id` und der Wert null an diese Funktion übergeben. Steht die Verbindung zum Server, wird die initialisierte *sigevent* Struktur gemeinsam mit den Anmeldedaten als Nachricht mit der QNX-Funktion `MsgSend()` synchron an den Server gesendet.
Sobald der Empfang der Nachricht vom Server mit der QNX-Funktion `MsgReply()` bestätigt wurde, wird die Variable `in_use` des Servers mit der Funktion `ipc_inc_connection()` inkrementiert und die Funktion `ipc_target_atta_value()` verlassen. In der Funktion `pulse_new()` wird dann erläutert, was der Server mit den empfangenen Daten macht.

ipc_target_deta_value()

Mit der Funktion `ipc_target_deta_value()` meldet sich ein Client bei einem Server ab.
Der formale Parameter `target_id` ist ein Index des Arrays `ipc_task[]` und ist vom Datentyp *int*. Der Client meldet sich bei dem Server ab, der sich an der Position `target_id` im Array `ipc_task[]` befindet. Dieser Parameter wird außerdem an die später aufrufenden Funktionen `wait_for_task()` und `ipc_dec_connection()` übergeben. Die formalen Parameter `pulse_nr` und `value` sind vom Datentyp *int*, der formale Parameter `coid` ist ein Zeiger vom Datentyp *int*. Die Definition dieser Parameter entspricht die der Parameter der Funktion `ipc_target_atta_value()`.
Die Funktion `ipc_target_deta_value()` liefert einen Status zurück, ob die Funktion fehlerfrei oder fehlerhaft ausgeführt wurde. Der Rückgabewert dieser Funktion ist vom Datentyp *int*.
Wie in der Beschreibung von `ipc_target_atta_value()` enthält die Funktion `ipc_target_deta_value()` auch eine lokale Variable `atta_msg`, die vom Typ *struct _CLIENT_PULSE_TYPE_T* ist. Die Daten des Clients werden für das Abmelden in diese Struktur aufgenommen. Der Wert vom formalen Parameter `target_id` wird der Variablen `target` zugewiesen. Anstelle von `ATTA_CMD` wird die Konstante `DETA_CMD` der Variablen `cmd` zugewiesen. Dieses Kommando signalisiert einem Server, dass ein Client sich bei ihm abmelden möchte. Die *sigevent* Struktur `ev` aus der Struktur *_CLIENT_PULSE_TYPE_T* wird mit dem QNX-Makro `SIGEV_PULSE_INIT()` initialisiert. Dabei wird die *sigevent* Struktur `ev`, die Connection ID `*coid`, die Priorität des Pulses, der Pulse-Code `pulse_nr` und die Daten `value` an diesen Makro übergeben.
Bevor diese Daten gemeinsam mit der *sigevent* Struktur als Nachricht versendet wird, wird die Funktion `wait_for_task()` mit dem Übergabeparameter `target_id` und dem Wert null aufgerufen. Diese Funktion überprüft, ob die Verbindung zum Server noch geöffnet ist. Besteht die Verbindung noch, wird die Nachricht synchron mit der QNX-Funktion `MsgSend()` an den Server gesendet. Sobald der Empfang der Nachricht vom Server mit der QNX-Funktion `MsgReply()` bestätigt wird, wird die Variable `in_use` des Servers mit der Funktion `ipc_dec_connection()` dekrementiert und die Funktion `ipc_target_deta_value()` verlassen. In der Funktion `pulse_del()` wird dann erläutert, wozu der Server die Daten des Clients benötigt.

Listing 27: Funktion zum Senden einer asynchronen Nachricht

```
int ipc_pulse_send (int target_id, int pulse_code, int value);
```

Die Funktion `ipc_pulse_send()` ist eine Wrapper-Funktion, die die QNX-Funktion `MsgSendPulse()` aufruft.
Über den formalen Parameter `target_id` wird bestimmt, welches Programm aus dem Array `ipc_task[]` die Pulse-Nachricht empfangen soll. Dem formalen Parameter `pulse_code` wird ein Pulse-Code übergeben, der in dem API lib_ipc.h defi-

niert ist. Dem formalen Parameter `value` können Daten übergeben werden, die 32-Bit groß sind.
Bevor die QNX-Funktion `MsgSendPulse()` aufgerufen wird, wird mit der Funktion `wait_for_task()` überprüft, ob das Programm bereits läuft und zu ihm eine Verbindung hergestellt ist. Steht die Verbindung, kann mit `MsgSendPulse()` eine Pulse-Nachricht zum Channel des Programms gesendet werden.
Der Rückgabewert der Funktion `ipc_pulse_send()` ist vom Datentyp *int*. Diese Funktion liefert einen Status zurück, ob sie fehlerfrei oder fehlerhaft ausgeführt wurde.

Listing 28: Funktionen zur Verwaltung der Client-Liste eines Servers

```
int pulse_new(struct _SERVER_PULSE_TYPE_T *msg,
                struct _SERVER_PULSE_TYPE_T *pulse_arr, int max_pulse);
int pulse_del (struct _SERVER_PULSE_TYPE_T *msg,
                struct _SERVER_PULSE_TYPE_T *pulse_arr, int max_pulse);
```

pulse_new()

Mit der Funktion `pulse_new()` nimmt ein Server die Anmeldedaten eines Clients, die er mit `MsgReceive()` empfangen hat, in sein eindimensionales Array `pulse_arr` auf. Dieses Array ist die im Kapitel 2.3 erwähnte „Liste“, in der der Server die Anmeldedaten der Clients verwaltet.
Die Elemente dieses Arrays sind vom Typ *struct _SERVER_PULSE_TYPE_T*. Diese Struktur besteht aus einer *int*-Variablen, einer *sigevent* Struktur, einer *long*-Variablen und einer Variablen vom Datentyp *unsigned char* (Siehe Listing 29).

Listing 29: lib_ipc.h Struktur *struct _SERVER_PULSE_TYPE_T*

```
typedef struct _SERVER_PULSE_TYPE_T
{
        int rcvid;
        struct sigevent ev;
        long delay;                     // used for IO_TASK
        unsigned char ch_nr;            // used for IO_TASK
}
SERVER_PULSE;
```

In der Variablen `rcvid` wird der Rückgabewert der QNX-Funktion `MsgReceive()` gespeichert. Die *sigevent* Struktur `ev` speichert die *sigevent* Struktur, die ein Server von einem Client empfangen hat. Die Variablen `delay` und `ch_nr` sind nicht relevant für das Verständnis der Funktion `pulse_new()`. Daher werden sie nicht weiter betrachtet.
Die Funktion `pulse_new()` enthält drei formale Parameter:
Die formalen Parameter `msg` und `pulse_arr` sind Zeiger vom Typ *struct _SERVER_PULSE_TYPE_T*. Der Inhalt an der Adresse, auf die der Zeiger `msg` verweist, enthält die empfangene *sigevent* Struktur des Clients und die Receive ID, die von der QNX-Funktion `MsgReceive()` zurückgeliefert wurde. Der Zeiger

pulse_arr zeigt auf den Beginn des Arrays pulse_arr, das beim Server deklariert ist. Der formale Parameter max_pulse gibt die Größe des Arrays pulse_arr an und ist vom Datentyp *int*. Der Rückgabewert der Funktion pulse_new() ist vom Datentyp *int*.
Die empfangene *sigevent* Struktur des Clients und die Receive ID, auf die msg zeigt, werden in das Array pulse_arr aufgenommen. Existieren diese Daten bereits im Array pulse_arr oder können keine neuen Daten eines Clients aufgenommen werden, liefert pulse_new() einen Fehler zurück.

pulse_del()

Die Funktion pulse_del() entfernt die Receive ID und die *sigevent* Struktur eines abgemeldeten Clients aus dem Array pulse_arr. Diese Funktion erwartet die gleichen Parameter wie die Funktion pulse_new(). Der Rückgabewert von pulse_del() ist vom Datentyp *int*.
Der Zeiger msg verweist auf die Daten des abzumeldenden Clients, die der Server von ihm erhalten hat. Diese Daten werden mit den Einträgen des Arrays pulse_arr verglichen, um den Eintrag dieses Clients zu finden. Stimmt ein Eintrag des Arrays pulse_arr mit den Daten des abzumeldenden Clients überein, werden die Variable rcvid und die Elemente der *sigevent* Struktur ev dieses Eintrages auf null zurückgesetzt. Somit wurden die Anmeldedaten dieses Clients gelöscht.
Konnte keine Übereinstimmung gefunden werden, liefert die Funktion pulse_del() einen Fehler zurück.

Listing 30: Funktion zur Benachrichtigung der Clients aus dem Array pulse_arr

```
int pulse_deliver (struct _SERVER_PULSE_TYPE_T *pulse_arr, int max_pulse,
                   int pulse_nr, int value);
```

Wenn ein Ereignis auftritt, werden die registrierten Clients aus dem Array pulse_arr mit der Funktion pulse_deliver() umgehend benachrichtigt.
Der Zeiger pulse_arr zeigt auf den Beginn des Arrays pulse_arr, das beim Server deklariert ist. Er ist vom Typ *struct _SERVER_PULSE_TYPE_T*. Der formale Parameter max_pulse gibt die Größe des Arrays pulse_arr an und ist vom Datentyp *int*. Dem formalen Parameter pulse_nr wird ein Pulse-Code übergeben, der in dem API lib_ipc.h definiert ist. Der formale Parameter value enthält Daten, die dem Client übermittelt wird.
Die Benachrichtigung dieser Clients erfolgt über die QNX-Funktion MsgDeliverEvent(). Wie in der Definition dieser QNX-Funktion aus Kapitel 2.1 beschrieben wurde, werden die gespeicherten Receive IDs und *sigevent* Strukturen der registrierten Clients aus dem Array pulse_arr nacheinander als Übergabeparameter an MsgDeliverEvent() übergeben.
Es werden nur die registrierten Clients benachrichtigt, deren Pulse-Code dem formalen Parameter pulse_nr entspricht.

Durch den Aufruf der QNX-Funktion `MsgDeliverEvent()` wird eine Pulse-Nachricht an den jeweiligen, registrierten Client gesendet, die über das Auftreten eines Ereignisses informiert.

Für die Entwicklung der firmenspezifischen IPC-Bibliothek wurde kein Design-Pattern verwendet. Jedoch ähnelt die Kommunikationsstruktur, die in der IPC-Bibliothek umgesetzt wurde, dem *Observer Pattern*.
Die Autoren des Buches *Head First Design Patterns* haben das Observer Pattern folgenderweise definiert:
"**The Observer Pattern** defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically." [11].
Objekte, die sich für die Zustandsänderung eines anderen Objektes interessieren, melden sich bei diesem Objekt an. Dieses Objekt wird als *Subject* bezeichnet.
Das Subject nimmt die angemeldeten Objekte in seine Liste auf. Die beim Subject registrierten Objekte werden als *Observers* bezeichnet. Sobald der Zustand des Subjects sich ändert, wird jeder registrierte Observer aus der Liste des Subjects benachrichtigt. Eine Zustandsänderung des Subjects kann dadurch ausgelöst werden, wenn zum Beispiel der Subject neue Daten erhält. Jeder Observer weiß bei Erhalt der Benachrichtigung des Subjects, wie er darauf reagieren soll.
Ein Objekt, das als Observer fungiert, kann sich beim Subject abmelden, wenn er nicht mehr benachrichtigt werden möchte. Das Subject wird dieses Objekt aus seiner Liste entfernen. Bei einer erneuten Zustandsänderung des Subjects wird dieses abgemeldete Objekt nicht mehr vom Subject informiert [11].
Der Ablauf des Observer Patterns ähnelt sehr dem der Kommunikationsstruktur, die im Kapitel 2.3 beschrieben wurde.

Jeder Observer benötigt eine Prozedur in seinem API, auf die das Subject zugreift, wenn er sie benachrichtigen möchte. Jeder Observer implementiert individuell seine eigene Prozedur und bestimmt damit, welche Aktionen er ausführen wird, wenn er vom Subject benachrichtigt wird. Wenn der Zustand des Subjects sich ändert, ruft es iterativ jede Prozedur des Observers aus seiner Liste auf, um sie über diese Zustandsänderung zu informieren. Die Aktionen, die individuell in der Prozedur jedes einzelnen Observers implementiert wurden, werden ausgeführt.
Diese Art der Benachrichtigung unterscheidet sich von der Benachrichtigung, die in der IPC-Bibliothek implementiert wurde. Wie im Kapitel 2.3 beschrieben wurde, findet die Benachrichtigung der registrierten Clients über das asynchrone Senden einer Nachricht statt.
Dieser Unterschied und weitere Unterschiede zwischen dem Observer Pattern und der IPC-Bibliothek sind im Folgenden in der Tabelle 2 aufgelistet.

Tabelle 2: Unterschiede zwischen der IPC-Bibliothek und dem Observer Pattern

IPC-Bibliothek	Observer Pattern
Wenn ein Ereignis auftritt, werden die registrierten Client-Threads über das asynchrone Senden einer Nachricht informiert.	Wenn ein Ereignis auftritt, ruft das Subject iterativ die Prozedur jedes Observers aus seiner Liste auf, um sie zu informieren.
Jeder angemeldete Client möchte nicht über jedes Ereignis informiert werden, das beim Server auftritt. Jeder angemeldete Client möchte nur über die Ereignisse benachrichtigt werden, für die sie sich interessieren.	Alle Observers werden vom Subject über das gleiche Ereignis informiert.
Die Clients können sich neue Daten aus dem shared memory Bereich holen.	Das Subject übergibt neue Daten als Übergabeparameter an die Prozedur, mit deren Aufruf er die Observers benachrichtigt. Somit erhalten die Observers über die formalen Parameter dieser Prozedur die neuen Daten.

Das Observer Pattern ist eine gute Alternative zur Lösung der Situation, wie bestimmte Clients über ein Ereignis informiert werden möchten, das beim Server auftritt.
Das Observer Pattern wird als Lösungsmuster immer für wiederkehrende Probleme verwendet. Durch den Einsatz dieses Design Patterns werden Begriffe vereinheitlicht und verbessert somit die Verständlichkeit. Die Verwendung des Observer Patterns führt zu einer besseren Entwicklung, Wartung und Dokumentation einer Software [12].

Obwohl das Observer Pattern diese Vorteile mit sich bringt, kann es die Umsetzung der IPC-Bibliothek aufgrund der in der Tabelle 2 genannten Unterschiede nicht vollständig ersetzen. Das Observer Pattern kann zusätzlich als Hilfsmittel für die Kommunikation zwischen Client und Server verwendet werden.

3 Konzept zur Umsetzung der firmenspezifischen IPC-Bibliothek unter Linux

In diesem Kapitel wird ein Konzept erarbeitet, wie die QNX-Funktionen der firmenspezifischen IPC-Bibliothek unter dem Linux Betriebssystem umgesetzt werden.

3.1 Anforderungsanalyse

Unter dem Betriebssystem Linux wird eine neue IPC-Bibliothek benötigt, die die im Kapitel 2 analysierten Funktionalitäten des APIs lib_ipc.h aus der firmenspezifischen IPC-Bibliothek nachbilden wird.
Der Kern der Umsetzung der neuen IPC-Bibliothek wird die Kommunikation zwischen Client und Server sein. Der Grund ist, dass die QNX-Funktionen, mit denen diese Kommunikation umgesetzt wurde, nicht mit dem Betriebssystem Linux kompatibel sind.
Die im Kapitel 2 beschriebene Kommunikation zwischen einem Client und einem Server soll auch unter Linux über den Austausch von Nachrichten stattfinden. Der Client und der Server sollen unter Linux die Möglichkeit haben, sowohl synchron als auch asynchron miteinander zu kommunizieren. Welche Funktionalitäten bezüglich der Kommunikation benötigt werden, werden in den nachfolgenden Anwendungsszenarien beschrieben.
Die Funktionen des APIs lib_ipc.h, die für die Behandlung der shared memory Bereiche aufgerufen werden, müssen in der neuen IPC-Bibliothek unter Linux nicht geändert werden. Der Grund ist, dass diese Funktionen die im Kapitel 2.2 beschriebenen POSIX-Funktionen verwenden und diese POSIX-Funktionen kompatibel mit dem Linux Betriebssystem sind [8]. Daher können diese Funktionen in die neue IPC-Bibliothek übernommen werden.

3.1.1 Anwendungsszenario eines Servers unter QNX

Zu Beginn erzeugt der Server durch die Verwendung der Funktion `ipc_shm_create()` einen neuen shared memory Bereich, in den Daten hineingeschrieben werden. Anschließend ruft er die QNX-Funktion `name_attach()` mit seinem Namen als Übergabeparameter auf, um einen Channel zu erzeugen. Außerdem registriert der Server mit dem Aufruf dieser QNX-Funktion seinen Namen im Namensraum.
Mit dem Aufruf von der QNX-Funktion `MsgReceive()` wartet der Server auf eingehende Nachrichten von Clients, die sich bei ihm an- oder abmelden möchten.
Seine Aufgabe ist die Verwaltung einer Liste von Daten, die die Clients ihm synchron zusenden, wenn sie sich bei ihm an- oder abmelden. Wenn er eine Nachricht synchron von einem Client empfängt, der sich bei ihm anmelden möchte, dann

nimmt der Server mit der Funktion `pulse_new()` die Daten dieses Clients in seine Liste auf. Mit der QNX-Funktion `MsgReply()` bestätigt er dem Client, dass er seine Nachricht empfangen hat.
Empfängt der Server synchron eine Nachricht von einem Client, der sich bei ihm abmelden möchte, entfernt der Server mit der Funktion `pulse_del()` die Daten dieses Clients aus seiner Liste. Mit der QNX-Funktion `MsgReply()` bestätigt er dem Client, dass er seine Nachricht empfangen hat.
Nach jeder beantworteten Nachricht der Clients ruft der Server die QNX-Funktion `MsgReceive()` wieder auf, um auf eingehende Nachrichten von Clients zu warten.

3.1.2 Anwendungsszenario eines Clients unter QNX

Der Client stellt zu Beginn mit dem Aufruf der Funktion `wait_for_task()` eine Verbindung zu dem Server her, mit dem er kommunizieren möchte. Es wird vorausgesetzt, dass der Server bereits läuft, so dass der Client eine Verbindung zu ihm öffnen kann. Ist diese Voraussetzung erfüllt, öffnet der Client mit der Funktion `ipc_shm_open()` den shared memory Bereich dieses Servers.
Die QNX-Funktionen `ChannelCreate()` und `ConnectAttach()` wurden vom Client, anders als im Kapitel 2.1 beschrieben, verwendet.
Der Client ruft die QNX-Funktion `ChannelCreate()` auf, um Nachrichten empfangen zu können. Mit dieser QNX-Funktion macht der Client sich nicht im System bekannt, da er anonym bleiben soll. Der Client ruft die QNX-Funktion `ConnectAttach()` auf, um eine Verbindung zu seinem eigenen Channel herzustellen. Seine Connection ID wird der Server erhalten, um ihn ansprechen zu können.
Um sich beim Server anzumelden, verwendet der Client die Funktion `ipc_target_atta_value()`. Wie im Kapitel 2.4 beschrieben, übergibt der Client als Übergabeparameter unter anderem seine Connection ID und seinen Pulse-Code an diese Funktion, die synchron in einer Nachricht an den Server gesendet wird. Durch den Pulse-Code bestimmt der Client, über welches Ereignis er benachrichtigt werden möchte.
Sobald der Client eine Antwort vom Server erhält, ruft er die QNX-Funktion `MsgReceive()` auf, um auf eingehende Pulse-Nachrichten zu warten. Sind neue Daten im shared memory Bereich des Servers vorhanden, ruft der Client, der parallel zum Server in einem Programm läuft, die Funktion `pulse_deliver()` auf, um den Client, der sich beim Server angemeldet hat, zu benachrichtigen. Dabei wird der angemeldete Client eine Pulse-Nachricht empfangen. Er liest die neuen Daten aus dem shared memory Bereich des Servers aus und verarbeitet diese. Die neu verarbeiteten Datenwerte schreibt der Client in seinen shared memory Bereich und ruft seinerseits die Funktion `pulse_deliver()` auf, um die Clients zu benachrichtigen, die auf die Aktualisierung dieser Daten warten.
Anschließend ruft der Client die QNX-Funktion `MsgReceive()` wieder auf, um auf eingehende Pulse-Nachrichten zu warten.
Wenn der Client sich beim Server abmelden möchte, verwendet er die Funktion `ipc_target_deta_value()`. Dabei übergibt der Client die gleichen Übergabepa-

rameter, die er bei der Anmeldung an die Funktion `ipc_target_atta_value()` übergeben hat. Diese Parameter werden synchron in einer Nachricht an den Server gesendet. Der Client erhält dann eine Antwort vom Server, wenn seine Nachricht bei ihm angekommen ist.

3.2 Erstellung eines APIs für eine neue IPC-Bibliothek unter Linux

Wie bereits im Kapitel 3.1 erwähnt, wird unter dem Betriebssystem Linux eine neue IPC-Bibliothek erstellt. Diese soll die Kommunikation über den Nachrichtenaustausch zwischen Client und Server sowie die Behandlung von shared memory Bereichen unterstützen. Für die neue IPC-Bibliothek werden Funktionsprototypen in einem API definiert, die die Funktionalitäten des APIs lib_ipc.h später umsetzen werden. Für die formalen Parameter und für den Rückgabewert der Funktionsprototypen, bezüglich der Kommunikation, werden noch keine Datentypen bestimmt. Diese können erst festgelegt werden, wenn später ein geeignetes Mittel unter Linux gefunden wurde, um die Kommunikation umzusetzen. Für die formalen Parameter und für den Rückgabewert der Funktionsprototypen, bezüglich der Behandlung von shared memory Bereichen, können die formalen Parameter der Funktionen `ipc_shm_open()`, `ipc_shm_close()` und `ipc_shm_create()` übernommen werden. Im Folgenden werden diese Funktionsprototypen kurz beschrieben, welche Funktionalitäten von lib_ipc.h sie umsetzen werden. Um zwischen den Funktionen der neuen IPC-Bibliothek und Funktionen zum Beispiel von externen Bibliotheken später unterscheiden zu können, wurde auf Wunsch des Unternehmens ein Präfix bei jedem Funktionsnamen eingeführt. Dieses Präfix trägt den Namen **mi_**.

Listing 31: Funktionsprototypen, die für einen Server definiert sind

```
void mi_create_server();
void mi_new_member();
void mi_del_member();
```

mi_create_server()

Mit der Funktion `mi_create_server()` erzeugt ein Server einen Kommunikationsendpunkt, über den der Server Nachrichten empfangen kann. Ein Kommunikationsendpunkt ist zum Beispiel unter dem Betriebssystem QNX ein Channel. Der Server soll mit der Funktion `mi_create_server()` die Möglichkeit haben, sich im System bekannt zu machen, damit Clients eine Verbindung zu ihm herstellen können.
Die Funktion `mi_create_server()` wird die QNX-Funktion `name_attach()` abbilden.

mi_new_member()

Mit der Funktion `mi_new_member()` nimmt ein Server die Anmeldedaten eines Clients in seine Liste auf, der sich bei ihm anmeldet. Diese Anmeldedaten empfängt der Server in einer Nachricht, die der Client synchron gesendet hat. Auf diese Nachricht wird der Server mit der Funktion `mi_msg_sync_reply()` antworten, die später noch erläutert wird.
Die Funktion `mi_new_member()` wird die Funktionalität der Funktion `pulse_new()` umsetzen.

mi_del_member()

Mit der Funktion `mi_del_member()` entfernt ein Server die Daten eines Clients aus seiner Liste, der sich bei ihm abmeldet.
Der Sender empfängt synchron die Daten in einer Nachricht vom Client, der sich bei ihm abmelden möchte. Die empfangenen Daten stimmen mit den Daten überein, die der Client dem Server bei der Anmeldung gesendet hat. Der Server vergleicht diese Daten mit den anderen Daten der Clients aus seiner Liste, damit er die Daten des richtigen Clients entfernen kann.
Auf die Nachricht des Clients wird der Server mit der Funktion `mi_msg_sync_reply()` antworten, die später noch erläutert wird.
Die Funktion `mi_del_member()` wird die Funktionalität der Funktion `pulse_del()` umsetzen.

Listing 32: Funktionsprototypen, die für einen Client definiert sind

```
void mi_wait_for_server();
void mi_create_msgbox();
void mi_connect_server();
void mi_disconnect_server();
void mi_notify_member();
```

mi_wait_for_server()

Mit der Funktion `mi_wait_for_server()` stellt ein Client eine Verbindung zu einem Server her, mit dem der Client kommunizieren möchte.
Der Client versucht für eine gewisse Zeit den Server im System zu lokalisieren. Ist der Server bereits im System bekannt, öffnet der Client eine Verbindung zu diesem Server. Kann der Client innerhalb der Zeit den Server im System nicht lokalisieren, bricht er den Vorgang ab.
Die Funktion `mi_wait_for_server()` wird die Funktionalität der Funktion `wait_for_task()` umsetzen.

mi_create_msgbox()

Mit der Funktion `mi_create_msgbox()` erzeugt ein Client einen Kommunikationsendpunkt, über den der Client Nachrichten empfangen kann.

Die Funktion `mi_create_msgbox()` wird die QNX-Funktion `ChannelCreate()` abbilden.

mi_connect_server()

Mit der Funktion `mi_connect_server()` meldet sich ein Client bei einem Server an. Dabei sendet der Client seine Anmeldedaten in einer Nachricht synchron an den Server. Wenn seine Nachricht beim Server angekommen ist, erhält er eine Bestätigung vom Server.
Die Funktion `mi_connect_server()` wird die Funktionalität der Funktion `ipc_target_atta_value()` umsetzen.

mi_disconnect_server()

Mit der Funktion `mi_disconnect_server()` meldet sich ein Client bei einem Server ab. Dabei sendet der Client seine Anmeldedaten in einer Nachricht synchron an den Server, die er auch bei der Anmeldung an den Server geschickt hat. Wenn seine Nachricht beim Server angekommen ist, erhält er eine Bestätigung vom Server.
Die Funktion `mi_disconnect_server()` wird die Funktionalität der Funktion `ipc_target_deta_value()` umsetzen.

mi_notify_member()

Wenn bei einem Server ein Ereignis auftritt, benachrichtigt der Client, der parallel zum Server in einem Programm läuft, mit der Funktion `mi_notify_member()` die angemeldeten Clients aus der Liste des Servers. Dabei werden nur die angemeldeten Clients, die sich für dieses Ereignis interessieren, benachrichtigt. Die anderen angemeldeten Clients werden darüber nicht informiert.
Die Funktion `mi_notify_member()` wird die Funktionalität der Funktion `pulse_deliver()` umsetzen.

Listing 33: Funktionsprototypen für die Kommunikation zwischen Client und Server

```
void mi_msg_send_sync();
void mi_msg_send_async();
void mi_msg_receive();
void mi_msg_sync_reply();
```

mi_msg_send_sync()

Die Funktion `mi_msg_send_sync()` sendet synchron eine Nachricht an einen Kommunikationsendpunkt eines Empfängers. Diese Funktion blockiert solange die Ausführung des Senders, bis die Antwort des Empfängers beim Sender eingetroffen ist.
Die Funktion `mi_msg_send_sync()` wird die QNX-Funktion `MsgSend()` abbilden.

mi_msg_send_async()

Die Funktion `mi_msg_send_async()` sendet asynchron eine Nachricht an einen Kommunikationsendpunkt eines Empfängers. Die Ausführung des Senders wird nicht blockiert.
Die Funktion `mi_msg_send_async()` wird die QNX-Funktion `MsgSendPulse()` abbilden.

mi_msg_receive()

Die Funktion `mi_msg_receive()` empfängt eine Nachricht von einem Sender. Durch den Aufruf dieser Funktion wird der Empfänger solange blockiert, bis eine Nachricht beim Kommunikationsendpunkt eingetroffen ist.
Die Funktion `mi_msg_receive()` wird die QNX-Funktion `MsgReceive()` abbilden.

mi_msg_sync_reply()

Mit der Funktion `mi_msg_sync_reply()` antwortet der Empfänger auf die Nachricht des Senders, die er von ihm empfangen hat.
Die Funktion `mi_msg_sync_reply()` wird die QNX-Funktion `MsgReply()` abbilden.

Listing 34: Funktionsprototypen für die Behandlung von shared memory Bereichen

```
void* mi_open_shm (unsigned int target_id, long size, int *err, unsigned int shm_num);
int mi_close_shm (void *addr, long size, unsigned int target_id, unsigned int shm_num);
void* mi_create_shm (const char *name, long size, int *err);
```

Für die Umsetzung der oben genannten Funktionen können die Implementierungen der Funktionen `ipc_shm_open()`, `ipc_shm_close()` und `ipc_shm_create()` aus dem API lib_ipc.h der IPC-Bibliothek übernommen werden. Die Beschreibung der Funktionen `ipc_shm_open()`, `ipc_shm_close()` und `ipc_shm_create()` befindet sich im Kapitel 2.4 „Beschreibung der Funktionen der IPC-Bibliothek“.

3.3 Erörterung von IPC-Techniken unter Linux sowie von externen Frameworks und Bibliotheken

Für die Umsetzung der im Kapitel 3.2 beschriebenen Funktionen werden IPC-Techniken sowie externe Frameworks und Bibliotheken ermittelt, die diese Umsetzung unter dem Betriebssystem Linux ermöglichen. Das primäre Ziel bei der Suche ist, geeignete IPC-Techniken, Frameworks oder Bibliotheken zu finden, die die Kommunikation zwischen Client und Server mit wenig Aufwand unter dem Betriebssystem Linux abbilden können. Für die Behandlung von shared memory Bereichen ist es nicht notwendig, nach IPC-Techniken, externen Frameworks oder exter-

nen Bibliotheken zu suchen. Der Grund ist, dass die Behandlung von shared memory Bereichen – wie im Kapitel 3.2 erwähnt – mit POSIX-Funktionen umgesetzt wurde.
Bevor die Suche nach geeigneten IPC-Techniken, Frameworks und Bibliotheken beginnt, müssen Kriterien für die Suche festgelegt werden. Diese sollen die IPC-Techniken, externe Frameworks oder externe Bibliotheken filtern, die für die Umsetzung der Kommunikation geeignet sind.
Im folgenden Kapitel werden Kriterien aufgestellt und die ermittelten IPC-Techniken, Frameworks und Bibliotheken nach diesen Kriterien evaluiert.

3.4 Evaluation

3.4.1 Festlegung der Kriterien und Vorgehensweise bei der Bewertung

Bei der Festlegung, welche Kriterien die IPC-Techniken, externe Frameworks und Bibliotheken als geeignete Kandidaten erfüllen sollen, werden zwei Arten von Kriterien unterschieden:
Mandatory criteria und *flexible criteria*.

Mandatory criteria

Das *mandatory* Kriterium ist ein K.O.-Kriterium. Wenn ein Kandidat dieses Kriterium nicht erfüllt, dann ist er ungeeignet und wird nicht weiter analysiert [13].

Flexible criteria

Das *flexible* Kriterium wird nicht so streng wie bei dem *mandatory* Kriterium gesehen. Wenn dieses Kriterium von einem Kandidat nicht erfüllt wird, ist es noch kein Grund, ihn als „ungeeignet“ auszuschließen. Der Kandidat sollte dieses Kriterium erfüllen, muss es aber nicht.
Ein *flexible* Kriterium ist verhandelbar. Wenn es nicht erfüllt wurde, kann zu einem späteren Zeitpunkt noch verhandelt werden, ob dieses Kriterium erfüllt werden muss oder nicht [13].

Die Kriterien, die für die Evaluation der IPC-Techniken, der externen Frameworks und Bibliotheken aufgestellt werden, können nicht alle erfüllt werden. Wenn alle Kriterien erfüllt werden müssten, würde die Suche nach potentiellen Kandidaten sehr schwierig werden. Daher werden die aufgestellten Kriterien in *mandatory* und in *flexible* Kriterien unterteilt, um geeignete Kandidaten einfacher finden zu können. Es ist zunächst wichtig, dass die Kandidaten die wichtigsten Kriterien, die *mandatory* Kriterien, erfüllen müssen.

In der Tabelle 4 sind die Kriterien aufgestellt, die die Kandidaten bezüglich der Umsetzung der Kommunikation zwischen Client und Server erfüllen sollen. Dabei werden die Kriterien Gewichtungen zugeordnet, die in der nachfolgenden Tabelle 3 aufgelistet sind.

Tabelle 3: Gewichtungen, die den Kriterien zugeordnet werden

Gewichtung *w*	**Wichtigkeit**
0	nicht wichtig
1	wichtig
2	zwingend

Alle Kriterien, die mit der Gewichtung zwei zugeordnet wurden, sind *mandatory* Kriterien. Alle Kriterien, die mit der Gewichtung eins oder null zugeordnet wurden, sind *flexible* Kriterien.

Bei der Aufstellung der Kriterien wurde der Artikel „A process and criteria for the evaluation of software frameworks in the domain of computer assisted surgery“ von Stefan Bohn, Werner Korb und Oliver Burgert als Anregung für die Definition der Kriterien verwendet.
Die Zuordnung der Gewichtung zu den Kriterien, die nachfolgend in der Tabelle 4 aufgelistet sind, wurde mit dem Unternehmen abgesprochen und festgelegt.

Tabelle 4: Bestimmung der Kriterien, die eine IPC-Technik, ein externes Framework oder eine externe Bibliothek erfüllen sollte

Kategorie	Kriterien	Gewichtung *w*
Kommunikation für Nachrichten- und Datenaustausch	Die Erzeugung der Kommunikationsendpunkte wird unterstützt.	2
	Die Öffnung einer Verbindung zum Server wird unterstützt.	2
	Die synchrone Kommunikation wird unterstützt.	2
	Die Kommunikation innerhalb eines Rechners ist möglich.	2
	Die Kommunikation über das Netzwerk ist möglich.	2
	Eine 1-zu-n Kommunikation wird unterstützt.	2
	Die asynchrone Kommunikation wird unterstützt.	1
	Die Verwaltung von Client-Listen wird explizit angeboten.	1
	Die Benachrichtigung einer bestimmten Anzahl von Clients (Multicast) wird unterstützt.	1
	Wenn ein bestimmter Prozess nicht mehr existiert, dann werden die Kommunikationspartner des Prozesses darüber informiert.	1
Dokumentation	Die Dokumentation ist vorhanden.	2
	Die Dokumentation ist auf Deutsch oder Englisch geschrieben.	2
	Die Dokumentation ist vollständig.	1
	Die Dokumentation ist verständlich aufgebaut.	1
	Der Quellcode und API sind jeweils vollständig mit Kommentaren versehen.	1
	Die Dokumentation bezüglich der Installation ist verständlich beschrieben.	1
Programmierung	Der Quellcode ist verfügbar.	2
	Die Installation läuft einwandfrei.	2
	Die Zeit für die Installation ist angemessen.	1
	Das Software-Produkt ist leicht zu verstehen und einfach zu nutzen.	1
	Die Einarbeitung in die Grundlagen des Software-Produkts ist zeitlich angemessen.	1

Kategorie	Kriterien	Gewichtung *w*
Support	Es existiert ein Internet Community Forum, in der Fragen zwischen Nutzern und Entwicklern ausgetauscht werden können.	1
	Support wird entweder über Telefon, E-Mail oder Internetseite angeboten.	1
	Die Internetseite ist übersichtlich und gut strukturiert.	1
Preis und Lizenz	Entwickelte Software, die das Software-Produkt nutzen, können kommerziell vertrieben werden.	2
	Die Bezahlung für eine Lizenz ist einmalig.	1

Für die Bewertung der in Frage kommenden Kandidaten wird eine Matrix aufgestellt, die im Folgenden dargestellt ist:

Tabelle 5: Beispiel für den Aufbau einer Matrix für die Evaluation der IPC-Techniken, externe Frameworks und Bibliotheken

	Kandidat 1	**Kandidat 2**
Kriterium 1	nicht erfüllt	erfüllt
Kriterium 2	erfüllt	erfüllt
Kriterium 3	erfüllt	erfüllt

In den Spalten werden die möglichen Kandidaten x_i (i = 1,...,n) und in den Zeilen die Kriterien c_j (j = 1,...,m) eingetragen. Wenn ein Kandidat ein *mandatory* Kriterium nicht erfüllt, wird die Zelle seines Eintrages grau eingefärbt. Die Einfärbung soll hervorheben, dass ein *mandatory* Kriterium nicht erfüllt wurde und somit die Analyse dieses Kandidaten nicht weiter fortgesetzt wird.

Um bewerten zu können, welcher Kandidat für die Umsetzung der Kommunikation am geeignetsten ist, wird die nachfolgende Formel verwendet:

$$C(x_i) = \sum_{j=1}^{m} (s_{ij} w_j)$$

Die Variable w_j ist die Gewichtung der Kriterien. Mit der Variable s_{ij} wird bewertet, ob ein Kandidat ein Kriterium vollständig, teilweise oder nicht erfüllt hat (siehe Tabelle 6).

Tabelle 6: Punkte, die aussagen, ob ein Kriterium vollständig, teilweise oder nicht erfüllt wurde

Punkteverteilung s	**Beschreibung**
0	Das Kriterium wurde nicht erfüllt.
1	Das Kriterium wurde teilweise erfüllt.
2	Das Kriterium wurde vollständig erfüllt.

Jede Gewichtung w_j der Kriterien wird mit einem Wert aus der Punkteverteilung s multipliziert. Das Ergebnis der Bewertung eines Kandidaten x_i ist die Summe $C(x_i)$, die aus den Ergebnissen der Multiplikation von w_j und s_{ij} gebildet wird [13].

3.4.2 Analyse und Bewertung von IPC-Techniken für die Umsetzung der Kommunikation unter Linux

Für die Umsetzung der Kommunikation unter dem Betriebssystem Linux werden folgende IPC-Techniken analysiert:

- (Namenlose) Pipes
- Benannte Pipes
- Message Queues (System-V-IPC)
- POSIX Message Queues
- Unix-Domain-Sockets

Bevor diese IPC-Techniken analysiert und ausgewertet werden, werden im Folgenden diese IPC-Techniken kurz vorgestellt.

(Namenlose) Pipes

Eine (namenlose) Pipe ist ein unidirektionaler Kommunikationskanal, über die zwei verwandte Prozesse miteinander kommunizieren können (Eltern-Kind-Prozess). Der Begriff „unidirektional“ bedeutet, dass ein Prozess Daten nur in diese Pipe schreiben kann. Wenn er Daten auslesen möchte, wird eine zweite, (namenlose) Pipe benötigt.
Wenn die (namenlose) Pipe voll ist, wird die Ausführung des schreibenden Prozesses solange blockiert, bis aus der vollen, (namenlosen) Pipe mindestens ein Byte gelesen wurde. Erst dann wird die Ausführung des schreibenden Prozesses nicht mehr blockiert und er kann wieder Daten in diese Pipe schreiben.
Wenn die (namenlose) Pipe leer ist, wird die Ausführung des lesenden Prozesses solange blockiert, bis der schreibende Prozess Daten in diese Pipe schreibt. Die Blockierung der Ausführung des lesenden Prozesses wird aufgehoben und er kann die neuen Daten aus der (namenlosen) Pipe lesen [8].

Benannte Pipes

Eine benannte Pipe ist ähnlich wie eine (namenlose) Pipe aufgebaut. Der Unterschied zwischen einer benannten Pipe und einer (namenlosen) Pipe ist, dass über eine benannte Pipe fremde Prozesse miteinander kommunizieren können. Diese Prozesse müssen nicht miteinander verwandt sein [8] [14].

Message Queues (System-V-IPC)

Prozesse können sich über die Message Queue Nachrichten austauschen. Ein Prozess kann eine Nachricht an eine Message Queue schicken, aus der ein anderer Prozess diese Nachricht abholt. Die Message Queue kann als eine Art Mailbox interpretiert werden.

Die Kommunikation über die Message Queue verläuft bidirektional. Das heißt, dass ein Prozess über die Message Queue Nachrichten schicken und aus dieser Message Queue Nachrichten lesen kann.
Wenn die Message Queue voll ist, wird der sendende Prozess solange blockiert, bis wieder Platz in der Message Queue vorhanden ist, um neue Nachrichten aufnehmen zu können. Die Blockierung der Ausführung des sendenden Prozesses wird aufgehoben und er kann seine Nachricht in der Message Queue ablegen. Diese Nachricht wurde vom sendenden Prozess mit einem „Typ“ versehen, der vom Datentyp *long* sein muss. Dieser „Typ“ ermöglicht einem empfangenen Prozess, gezielt Nachrichten eines bestimmten Typs zu empfangen.
Der empfangene Prozess kann auf jede oder auf bestimmte Nachrichten vom sendenden Prozess warten. Dabei wird seine Ausführung solange blockiert, bis die Nachricht des sendenden Prozesses in der Message Queue abgelegt wurde. Wenn der empfangene Prozess auf eine bestimmte Nachricht wartet, wertet er den „Typ“ beim Lesen der Nachricht aus. Stimmt dieser „Typ“ mit dem Typ seiner erwarteten Nachricht überein, holt er diese Nachricht aus der Message Queue und verarbeitet sie. Liegt nicht seine erwartete Nachricht in der Message Queue vor, bleibt die Nachricht in der Message Queue und seine Ausführung wird blockiert, bis eine neue Nachricht eintrifft [8] [16].

POSIX Message Queues

Wie die Message Queue (System-V-IPC) ermöglicht auch die POSIX Message Queue die Kommunikation zwischen Prozessen über den Nachrichtenaustausch.
Ein Prozess schickt eine Nachricht an eine Message Queue, aus der ein anderer Prozess diese Nachricht liest.
Genau wie bei der Message Queue (System-V-IPC) wird ein sendender Prozess blockiert, wenn die Message Queue voll ist. Die Blockierung seiner Ausführung wird erst aufgehoben, wenn wieder Platz in der Message Queue vorhanden ist.
Wenn die Message Queue leer ist, wird die Ausführung eines empfangenen Prozesses blockiert. Die Blockierung seiner Ausführung wird erst aufgehoben, wenn die Message Queue wieder Nachrichten enthält.
Im Unterschied zur Message Queue (System-V-IPC) ist ein empfangener Prozess bei der POSIX Message Queue immer dann blockiert, wenn die Message Queue leer ist. Bei der Message Queue (System-V-IPC) kann auch ein empfangener Prozess blockiert werden, wenn seine erwartete Nachricht nicht in der Message Queue vorliegt. Dabei muss die Message Queue nicht leer sein, wenn der empfangene Prozess blockiert wird [10] [15].

Unix-Domain-Sockets

Die Unix-Domain-Sockets ermöglichen die Kommunikation zwischen Prozessen auf demselben Rechner. Jeder Prozess muss sich einen Socket erzeugen, an den er seine Daten schicken und aus dem er Daten lesen kann. Die Adressen von Unix-Domain-Sockets sind Dateipfade, die den Ort der Dateien im Dateisystem angeben. Diese

Dateien werden für die Kommunikation zwischen Socket-Endpunkten der Prozesse verwendet. Sie werden im Dateisystem erst angelegt, wenn ein Socket an eine Datei gebunden wird.
Die Unix-Domain-Sockets sind verbindungsorientiert. Das heißt, dass eine Verbindung zwischen zwei Prozessen existieren muss, bevor sie miteinander kommunizieren können. Wenn zwei Prozesse miteinander kommunizieren, kann kein dritter Prozess an dieser Kommunikation teilnehmen, weil zwischen den beiden Prozessen eine private Verbindung besteht. Möchte ein Prozess gleichzeitig mit mehreren Prozessen kommunizieren, muss er für jeden Kommunikationskanal einen eigenen Filedeskriptor erzeugen.
Ein empfangener Prozess wartet auf eingehende Daten und wird solange blockiert, bis der sendende Prozess ihm Daten zuschickt. Der empfangene Prozess liest diese Daten aus seinem Socket und verarbeitet sie.
Der sendende Prozess arbeitet nach dem Senden der Daten weiter, er wird nicht blockiert. Seine Ausführung wird nur blockiert, wenn der Speicher voll ist und keine neuen Daten aufgenommen werden können [8] [16].

Diese IPC-Techniken und die Kriterien aus der Tabelle 4, Kapitel 3.4.1, werden in einer Matrix dargestellt. Diese IPC-Techniken werden analysiert und nach den Kriterien bewertet, die in den Kategorien *Kommunikation für Nachrichten- und Datenaustausch, Dokumentation, Programmierung, Support* sowie *Preis und Lizenz* eingeordnet sind. Die *mandatory* Kriterien aus jeder Kategorie sind blau eingefärbt.
Im Folgenden wird in der Matrix bewertet, ob die IPC-Techniken die Kriterien der Kategorie *Kommunikation für Nachrichten- und Datenaustausch* erfüllen können (Tabelle 7).

Tabelle 7: Bewertung der IPC-Techniken nach den Kriterien der Kategorie *Kommunikation für Nachrichten- und Datenaustausch*

	(Namenlose) Pipes	Benannte Pipes	Message Queues (System-V-IPC)	POSIX Message Queues	Unix-Domain-Sockets
Die Erzeugung der Kommunikationsendpunkte wird unterstützt.	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*
Die Öffnung einer Verbindung zum Server wird unterstützt.	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*
Die synchrone Kommunikation wird unterstützt.	Nein	Nein	*Ein K.O. Kriterium wurde nicht erfüllt.*	Nein	*Ein K.O. Kriterium wurde nicht erfüllt.*
Die Kommunikation innerhalb eines Rechners ist möglich.	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*
Die Kommunikation über das Netzwerk ist möglich.	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*	Nein	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*
Eine 1-zu-n Kommunikation wird unterstützt.	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*	Es wird nur eine 1-zu-1 Kommunikation unterstützt.
Die asynchrone Kommunikation wird unterstützt.	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*
Die Verwaltung von Client-Listen wird explizit angeboten.	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*
Die Benachrichtigung einer bestimmten Anzahl von Clients (Multicast) wird unterstützt.	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*
Wenn ein bestimmter Prozess nicht mehr existiert, dann werden die Kommunikationspartner des Prozesses darüber informiert.	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Ein K.O. Kriterium wurde nicht erfüllt.*
Summe:	**0**	**0**	**0**	**0**	**0**

Das Ergebnis aus der Analyse der in der Tabelle 7 genannten IPC-Techniken zeigt, dass keine von ihnen für die Umsetzung der Kommunikation geeignet ist.
Die (namenlose) und benannte Pipes sowie die POSIX Message Queues erfüllen das *mandatory* Kriterium *„Die synchrone Kommunikation wird unterstützt.“* nicht.
Die Art der Blockierung eines sendenden oder eines empfangenen Prozesses, die diese IPC-Techniken anbieten, ist nicht die Art, die für die Umsetzung der Kommunikation benötigt wird. Wie unter dem Abschnitt „*Synchroner Kommunikation über Message Passing*“ im Kapitel 2.1 beschrieben, soll der sendende Prozess blockiert werden, wenn er eine Nachricht an den empfangenen Prozess sendet. Die Ausführung des empfangenen Prozesses ist bereits blockiert, da er auf eingehende Nachrichten wartet. Wenn der empfangene Prozess die Nachricht vom sendenden Prozess erhält, wird seine Blockierung aufgehoben. Er verarbeitet diese Nachricht und antwortet dann dem sendenden Prozess. Sobald der sendende Prozess die Antwort erhält, wird seine Blockierung aufgehoben und er kann weiter arbeiten. Bei den eben erwähnten IPC-Techniken findet die Blockierung eines sendenden und eines empfangenen Prozesses dann statt, wenn die (namenlose) Pipe, die benannte Pipe oder die Message Queue voll oder leer ist.
Die Message Queues (System-V-IPC) erfüllen das *mandatory* Kriterium *„Die Kommunikation über das Netzwerk ist möglich.“* nicht. Die Unix-Domain-Sockets erfüllen das *mandatory* Kriterium *„Eine 1-zu-n Kommunikation wird unterstützt.“* nicht. Wie zu Beginn unter dem Abschnitt *„Unix-Domain-Sockets“* beschrieben, muss ein Prozess für jeden Kommunikationskanal einen Filedeskriptor einrichten, wenn er gleichzeitig mit mehreren Prozessen kommunizieren möchte. Das heißt, dass dieser Prozess nur jeweils mit einem anderen Prozess über einen Kommunikationskanal kommunizieren kann (Eins-zu-Eins Kommunikation).

Da jede IPC-Technik ein *mandatory* Kriterium nicht erfüllt hat, wird die Analyse nicht weiter fortgesetzt.
Durch die Evaluation der IPC-Techniken stellt sich heraus, dass diese IPC-Techniken nicht geeignet sind, um die Kommunikation zwischen Client und Server mit minimalem Aufwand umzusetzen. Daher wird im folgenden Kapitel nach externen Frameworks und Bibliotheken gesucht, die diese Kommunikation umsetzen können.

3.4.3 Analyse und Bewertung von Frameworks und Bibliotheken für die Umsetzung der Kommunikation unter Linux

Es gibt zahlreiche, externe Frameworks und Bibliotheken, die auf dem Markt kommerziell und nicht kommerziell vorhanden sind. Aufgrund der geringen Zeit, die für diese Bachelor Arbeit angesetzt wurde, können nicht alle externen Frameworks und Bibliotheken nach den im Kapitel 3.4.1 festgelegten Kriterien bewertet werden.
Daher wird eine Vorauswahl der externen Frameworks und Bibliotheken durchgeführt, die die externe Frameworks und Bibliotheken filtern, bei denen es sich lohnt, sie weiter zu analysieren.
Für die Vorauswahl werden grobe Kriterien definiert, die im Folgenden aufgelistet sind:

- Der Preis eines externen Frameworks oder einer externen Bibliothek soll angemessen sein.
- Das externe Framework oder die externe Bibliothek soll unter dem Betriebssystem Linux lauffähig sein.
- Das externe Framework oder die externe Bibliothek soll die Programmiersprache C oder C++ unterstützen.
- Das externe Framework oder die externe Bibliothek soll die Kommunikation über den Nachrichtenaustausch unterstützen.
- Die letzte Aktualisierung des externen Frameworks oder der externen Bibliothek soll nicht älter als fünf Jahre sein.
- Das externe Framework oder die externe Bibliothek soll fehlerfrei sein.

Diese groben, festgelegten Kriterien sind *mandatory* Kriterien. Erfüllt ein externes Framework eines dieser Kriterien nicht, wird dieses Framework von der Analyse ausgeschlossen. Dies gilt ebenso für die externen Bibliotheken.

In der nachfolgenden Matrix wird ein Ausschnitt aus der Matrix dargestellt, in der externe Frameworks und Bibliotheken nicht die Vorauswahl geschafft haben.
Die gesamte Matrix, in der externe Frameworks und Bibliotheken bei der Vorauswahl ausgeschieden sind, ist im Anhang unter dem Punkt A aufgeführt.

Tabelle 8: Externe Frameworks und Bibliotheken, die die Vorauswahl nicht geschafft haben

	BSPlib	GAMMA	MSMQ	RabbitMQ
Preis / Lizenz	*Keine weitere Betrachtung dieses Programms.*	***GNU GPL***	*Keine weitere Betrachtung dieses Programms.*	***Mozilla Public License***
Betriebssystem	*Keine weitere Betrachtung dieses Programms.*	**Linux 2.6.24**	Windows (2000)	Solaris, BSD, **Linux**, MacOSX, TRU64, Windows, VxWorks
Programmiersprache	**C, C++**, Fortran	**C**	*Keine weitere Betrachtung dieses Programms.*	---
Kommunikation über den Nachrichtenaustausch	*Keine weitere Betrachtung dieses Programms.*	Ja	Asynchrone Kommunikation	*Keine weitere Betrachtung dieses Programms.*
Version / Letzte Aktualisierung	1.4 / 30.09.1998	unbekannt / 15.04.2009	*Keine weitere Betrachtung dieses Programms.*	2.8.7 / 27.09.2012
Der Kandidat soll fehlerfrei sein	*Keine weitere Betrachtung dieses Programms.*	Nein	Nein	*Keine weitere Betrachtung dieses Programms.*
Information zum Typ:	Bibliothek	Bibliothek	Framework	message-oriented middleware

Die externe Frameworks und Bibliotheken, die bei der Vorauswahl durchgekommen sind, sind in der nachfolgenden Matrix aufgelistet.

Tabelle 9: Externe Frameworks und Bibliotheken, die die Vorauswahl geschafft haben

	SIMPL	**zeroMQ**	**D-Bus**	**Open MPI**
Preis / Lizenz	kostenfrei ***(GNU Library or Lesser General Public License (LGPL))***	kostenfrei ***(GNU Lesser General Public License (LGPL))***	kostenfrei ***(GNU General Public License or Academic Free License 2.1)***	kostenfrei ***(New BSD License)***
Betriebssystem	**Linux**	**Linux**, Windows, OS X	Unix, **Linux**	**Linux**, Windows, OS X, Solaris
Programmiersprache	**C**, Python, Tcl, (Java)	**C, C++** und viele andere Programmiersprachen	**C, C++** und viele andere Programmiersprachen	**C, C++**, Fortran
Kommunikation über den Nachrichtenaustausch	Ja	Ja	Ja	Ja
Version / Letzte Aktualisierung	3.3.8 / 30.10.2012	3.2 / 15.10.2012	1.6.8 / 28.09.2012	1.6.1 / 22.08.2012
Der Kandidat soll fehlerfrei sein	Ja	Ja	Ja	Ja
Information zum Typ:	Bibliothek	Bibliothek	Framework	API

	boost	Cogent API	SRR Kernel Module	libqmsg
Preis / Lizenz	kostenfrei ***(Boost Software License)***	kommerziell ***(Cogent Software License)***	1x SRR Kernel Module Commercial Site License = 1086 Euro exkl. MwSt. Dies ist eine einmalige Developer Lizenz. ***(Cogent Software License)***	***Mozilla Public License 1.1 (MPL 1.1)*** ***Copyright © 2005 Siemens AG***
Betriebssystem	UNIX, GNU/**Linux**, Mac OS, Windows	**Linux**, QNX	32-Bit Linux OS der Version 2.4.18 oder später. Frühere Versionen 2.4.x gehen auch.	**Linux**
Programmiersprache	**C++**	**C**	**C**	**C**
Kommunikation über den Nachrichtenaustausch	Ja	Ja	Ja	Ja
Version / Letzte Aktualisierung	1.51.0 / 20.08.2012	6.4.11 / unbekannt (1995 - 2012)	1.4.44 / unbekannt (1995 - 2012)	0.4 / 17.07.2009
Der Kandidat soll fehlerfrei sein	Ja	Ja	Ja	Ja
Information zum Typ:	Bibliothek	Bibliothek	Bibliothek	Framework

Die in der Tabelle 9 aufgelisteten Frameworks und Bibliotheken werden nun nach den Kriterien, die in der Tabelle 4 im Kapitel 3.4.1 festgelegt wurden, bewertet.
Die Bewertung dieser Frameworks und Bibliotheken wird im Folgenden in den Kategorien *Kommunikation für Nachrichten- und Datenaustausch, Dokumentation, Programmierung, Support* sowie *Preis und Lizenz* erfolgen. Die Kriterien, die blau eingefärbt sind, sind *mandatory* Kriterien.

Tabelle 10: Bewertung der externen Frameworks und Bibliotheken in der Kategorie *Kommunikation für Nachrichten- und Datenaustausch*

	SIMPL	**zeroMQ**
Die Erzeugung der Kommunikationsendpunkte wird unterstützt.	Ja	Ja
Die Öffnung einer Verbindung zum Server wird unterstützt.	Ja	Ja
Die synchrone Kommunikation wird unterstützt.	Ja	Teilweise
Die Kommunikation innerhalb eines Rechners ist möglich.	Ja	Ja
Die Kommunikation über das Netzwerk ist möglich.	Ja	Ja
Eine 1-zu-n Kommunikation wird unterstützt.	Ja	Ja
Die asynchrone Kommunikation wird unterstützt.	Ja	Ja
Die Verwaltung von Client-Listen wird explizit angeboten.	Nein	Nein
Die Benachrichtigung einer bestimmten Anzahl von Clients (Multicast) wird unterstützt.	Mit Hilfe der Implementierung zur Verwaltung der Client-Liste aus der ipc-Bibliothek.	Mit Hilfe der Implementierung zur Verwaltung der Client-Liste aus der ipc-Bibliothek.
Wenn ein bestimmter Prozess nicht mehr existiert, dann werden die Kommunikationspartner des Prozesses darüber informiert.	Ja	Teilweise. Es existiert nur eine Beschreibung, wie man es umsetzen könnte.
Summe:	**30**	**27**

	D-Bus	Open MPI	libqmsg
Die Erzeugung der Kommunikations-endpunkte wird unterstützt.	Adressierung jedes beliebigen Endpunktes auf dem Bus.	Ja	Ja
Die Öffnung einer Verbindung zum Server wird unterstützt.	Es wird eine Verbindung zum Bus hergestellt.	Verbindungen werden automatisch hergestellt.	Ja
Die synchrone Kommunikation wird unterstützt.	(scheinbar) synchron. Der Sender wird durch Angabe einer Zeit solange blockiert, bis die Zeit abgelaufen ist. Ob der Empfänger auch blockierend ist, konnte nicht ermittelt werden.	Ja	Ja
Die Kommunikation innerhalb eines Rechners ist möglich.	Ja	Ja	Ja
Die Kommunikation über das Netzwerk ist möglich.	Ja	Ja	Ja
Eine 1-zu-n Kommunikation wird unterstützt.	Ja	Ja	Ja
Die asynchrone Kommunikation wird unterstützt.	Ja	Ja	Ja
Die Verwaltung von Client-Listen wird explizit angeboten.	Nein	Nein	Nein
Die Benachrichtigung einer bestimmten Anzahl von Clients (Multicast) wird unterstützt.	Ja	(Ja, wenn Prozesse in Gruppen zusammen-gefasst werden.)	Mit Hilfe der Implemen-tierung zur Verwaltung der Client-Liste aus der ipc-Bibliothek.
Wenn ein bestimmter Prozess nicht mehr existiert, dann werden die Kommunikationspartner des Prozesses darüber informiert.	Ja	Keine Informationen dazu gefunden.	Keine Informationen dazu gefunden.
Summe:	**28**	**27**	**28**

Zu der externen Bibliothek *boost* sollte erwähnt werden, dass sie aus einer Vielzahl von Unterbibliotheken besteht, die verschiedene Aufgaben erfüllen. Für die Umsetzung der Kommunikation zwischen Client und Server werden die Unterbibliotheken *MPI* und *Interprocess* der Bibliothek *boost* näher analysiert.

	boost (Boost.MPI)	**boost (Boost.Interprocess)**
Die Erzeugung der Kommunikationsendpunkte wird unterstützt.	Ja	*Ein K.O. Kriterium wurde nicht erfüllt.*
Die Öffnung einer Verbindung zum Server wird unterstützt.	Verbindungen werden automatisch hergestellt.	*Ein K.O. Kriterium wurde nicht erfüllt.*
Die synchrone Kommunikation wird unterstützt.	Ja	Nein, message queue unterstützt keine synchrone Kommunikation.
Die Kommunikation innerhalb eines Rechners ist möglich.	Ja	*Ein K.O. Kriterium wurde nicht erfüllt.*
Die Kommunikation über das Netzwerk ist möglich.	Ja	*Ein K.O. Kriterium wurde nicht erfüllt.*
Eine 1-zu-n Kommunikation wird unterstützt.	Ja	*Ein K.O. Kriterium wurde nicht erfüllt.*
Die asynchrone Kommunikation wird unterstützt.	Ja	*Ein K.O. Kriterium wurde nicht erfüllt.*
Die Verwaltung von Client-Listen wird explizit angeboten.	Nein	*Ein K.O. Kriterium wurde nicht erfüllt.*
Die Benachrichtigung einer bestimmten Anzahl von Clients (Multicast) wird unterstützt.	(Ja, wenn Prozesse in Gruppen zusammengefasst werden.)	*Ein K.O. Kriterium wurde nicht erfüllt.*
Wenn ein bestimmter Prozess nicht mehr existiert, dann werden die Kommunikationspartner des Prozesses darüber informiert.	Keine Informationen dazu gefunden.	*Ein K.O. Kriterium wurde nicht erfüllt.*
Summe:	27	**0**

	SRR Kernel Module	Cogent API
Die Erzeugung der Kommunikationsend-punkte wird unterstützt.	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Keine weitere Betrachtung dieses Programms (Grund: siehe rechts).*
Die Öffnung einer Verbindung zum Server wird unterstützt.	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Keine weitere Betrachtung dieses Programms (Grund: siehe rechts).*
Die synchrone Kommunikation wird unterstützt.	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Keine weitere Betrachtung dieses Programms (Grund: siehe rechts).*
Die Kommunikation innerhalb eines Rechners ist möglich.	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Keine weitere Betrachtung dieses Programms (Grund: siehe rechts).*
Die Kommunikation über das Netzwerk ist möglich.	Nein	*Keine weitere Betrachtung dieses Programms (Grund: siehe rechts).*
Eine 1-zu-n Kommunikation wird unterstützt.	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Keine weitere Betrachtung dieses Programms (Grund: siehe rechts).*
Die asynchrone Kommunikation wird unterstützt.	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Keine weitere Betrachtung dieses Programms (Grund: siehe rechts).*
Die Verwaltung von Client-Listen wird explizit angeboten.	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Keine weitere Betrachtung dieses Programms (Grund: siehe rechts).*
Die Benachrichtigung einer bestimmten Anzahl von Clients (Multicast) wird unterstützt.	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Keine weitere Betrachtung dieses Programms (Grund: siehe rechts).*
Wenn ein bestimmter Prozess nicht mehr existiert, dann werden die Kommunikationspartner des Prozesses darüber informiert.	*Ein K.O. Kriterium wurde nicht erfüllt.*	*Keine weitere Betrachtung dieses Programms (Grund: siehe rechts).*
Summe:	**0**	**0**

Der Grund für das Ausscheiden von Cogent API ist, dass diese Bibliothek nur in Verbindung mit anderen Cogent Produkten, zum Beispiel mit Cascade Datahub, genutzt werden kann.

Die Frameworks oder die Bibliotheken, die die *mandatory* Kriterien in der Kategorie *Kommunikation für Nachrichten- und Datenaustausch* nicht erfüllt haben, werden im Folgenden nicht weiter analysiert.

Tabelle 11: Bewertung der externen Frameworks und Bibliotheken in der Kategorie *Dokumentation*

	SIMPL	zeroMQ
Die Dokumentation ist vorhanden.	Ja	Ja
Die Dokumentation ist auf Deutsch oder Englisch geschrieben.	Ja, auf Englisch.	Ja, auf Englisch.
Die Dokumentation ist vollständig.	Ja	Ja
Die Dokumentation ist verständlich aufgebaut.	Ja	Ja
Der Quellcode und API sind jeweils vollständig mit Kommentaren versehen.	Ja	Teilweise
Die Dokumentation bezüglich der Installation ist verständlich beschrieben.	Ja	Ja
Summe:	**16**	**15**

	D-Bus	Open MPI	boost (Boost.MPI)	libqmsg
Die Dokumentation ist vorhanden.	Ja	Ja	Dokumentation, Beispiele und Tutorials sind vorhanden.	Nein
Die Dokumentation ist auf Deutsch oder Englisch geschrieben.	Ja, auf Englisch.	Ja, auf Englisch.	Ja, auf Englisch.	*Ein K.O. Kriterium wurde nicht erfüllt.*
Die Dokumentation ist vollständig.	Ja	Ja	Ja	*Ein K.O. Kriterium wurde nicht erfüllt.*
Die Dokumentation ist verständlich aufgebaut.	Ja	Ja	Ja	*Ein K.O. Kriterium wurde nicht erfüllt.*
Der Quellcode und API sind jeweils vollständig mit Kommentaren versehen.	Ja	Ja	Ja	*Ein K.O. Kriterium wurde nicht erfüllt.*
Die Dokumentation bezüglich der Installation ist verständlich beschrieben.	Teilweise	Ist akzeptabel.	Nein	*Ein K.O. Kriterium wurde nicht erfüllt.*
Summe:	**15**	**15**	**14**	**0**

Tabelle 12: Bewertung der externen Frameworks und Bibliotheken in der Kategorie *Programmierung*

	SIMPL	zeroMQ	D-Bus	Open MPI	boost (Boost.MPI)
Der Quellcode ist verfügbar.	Ja	Ja	Ja	Ja	Ja
Die Installation läuft einwandfrei.	Ja	Ja	Ja. Man muss Expat zusätzlich installieren, wenn diese während der Installation von D-Bus nicht gefunden wurde.	Ja	Ja
Die Zeit für die Installation ist angemessen.	Ja	Ja	Teilweise	Ja	Ja
Das Software-Produkt ist leicht zu verstehen und einfach zu nutzen.	Ja	Ja	Ist akzeptabel.	Die Theorie ist komplex.	Die Theorie ist komplex.
Die Einarbeitung in die Grundlagen des Software-Produkts ist zeitlich angemessen.	Ja	Ja	Von der Theorie her hat es lange gedauert, da die Dokumentation umfangreich ist.	Die Theorie ist komplex. Für die weitere Einarbeitung wird mehr Zeit benötigt.	Die Theorie ist komplex. Für die weitere Einarbeitung wird mehr Zeit benötigt.
Summe:	**14**	**14**	**8**	**10**	**10**

Tabelle 13: Bewertung der externen Frameworks und Bibliotheken in der Kategorie *Support*

	SIMPL	**zeroMQ**
Es existiert ein Internet Community Forum, in der Fragen zwischen Nutzern und Entwicklern ausgetauscht werden können.	Internet Community Forum existiert nicht. Es existiert nur eine Mailing List, die auf den ersten Blick nicht aktiv genutzt wird.	Eine Mailing List existiert, die aktiv genutzt wird.
Support wird entweder über Telefon, E-Mail oder Internetseite angeboten.	Teilweise. Die Internetseite und die Mailing List existieren. Zur Unterstützung gibt es ein Buch, das gut strukturiert und sehr verständlich ist.	Kommerzieller Support verfügbar.
Die Internetseite ist übersichtlich und gut strukturiert.	Ist akzeptabel.	sehr übersichtlich, sehr gut strukturiert.
Summe:	3	6

	D-Bus	**Open MPI**	**boost (Boost.MPI)**
Es existiert ein Internet Community Forum, in der Fragen zwischen Nutzern und Entwicklern ausgetauscht werden können.	Internet Community Forum existiert nicht.	Es existiert eine Mailing List.	Internet Community Forum existiert.
Support wird entweder über Telefon, E-Mail oder Internetseite angeboten.	Es existiert nur eine Mailing List.	Nein	• Mailing Lists existieren. • Kommerzieller Support von Experten. • Tutorials existieren.
Die Internetseite ist übersichtlich und gut strukturiert.	Übersichtlichkeit und Strukturierung der Internetseite sind akzeptabel.	Übersichtlichkeit und Strukturierung der Internetseite sind akzeptabel.	Übersichtlichkeit und Strukturierung der Internetseite sind akzeptabel.
Summe:	2	3	5

Tabelle 14: Bewertung der externen Frameworks und Bibliotheken in der Kategorie *Preis und Lizenz*

		SIMPL	zeroMQ	D-Bus	Open MPI	boost (Boost.MPI)
Entwickelte Software, die das Software-Produkt nutzen, können kommerziell vertrieben werden.		Ja	Ja	Ja	Ja	Ja
	Die Bezahlung für eine Lizenz ist einmalig.	SIMPL ist nicht kommerziell.	zeroMQ ist nicht kommerziell.	D-Bus ist nicht kommerziell.	Open MPI ist nicht kommerziell.	boost ist nicht kommerziell.
Summe:		**6**	**6**	**6**	**6**	**6**

Die Ergebnisse der Evaluation der externen Frameworks und Bibliotheken werden in der nachfolgenden Tabelle aufgelistet:

Tabelle 15: Ergebnis der Evaluation der externen Frameworks und Bibliotheken, die zumindest die *mandatory* Kriterien erfüllt haben

Framework/Bibliothek	**Ergebnis**
SIMPL	69
zeroMQ	68
boost (Boost.MPI)	62
Open MPI	61
D-Bus	59

Tabelle 16: Ergebnis der Evaluation der externen Frameworks und Bibliotheken, die die *mandatory* Kriterien **nicht** erfüllt haben

Framework/Bibliothek	**Ergebnis**
libqmsg	28
SRR Kernel Module	0
boost (Boost.Interprocess)	0
Congent API	0

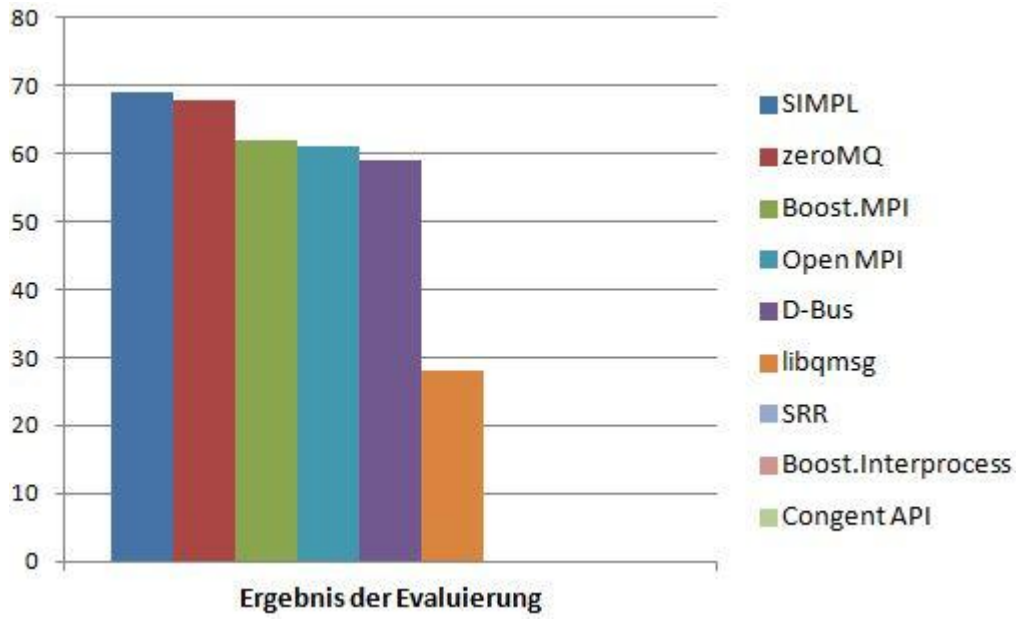

Abb. 13: Ergebnis der Evaluation der externen Frameworks und Bibliotheken als Säulendiagramm dargestellt

Wie in der Abbildung 13 zu sehen ist, haben die beiden externen Bibliotheken *SIMPL* und *zeroMQ* bei der Evaluation am besten abgeschnitten. Danach folgt auf Platz drei die Unterbibliothek *MPI* der externen Bibliothek *boost*. Knapp dahinter belegt *Open MPI* den vierten und das Framework *D-Bus* den fünften Platz.

Das Ergebnis dieser Evaluation zeigt, dass diese fünf Kandidaten, die in der Tabelle 15 aufgeführt sind, für die Umsetzung der Kommunikation zwischen Client und Server geeignet sind. Der Grund ist, dass sie die *mandatory* Kriterien jeder einzelnen Kategorie erfüllt haben.
Die Kandidaten, die während der Evaluation ein *mandatory* Kriterium nicht erfüllt haben, sind in der Tabelle 16 aufgelistet. Diese Kandidaten sind für die Umsetzung der Kommunikation zwischen Client und Server nicht geeignet, da jeder von ihnen ein *mandatory* Kriterium nicht erfüllt hat.

Da die externen Bibliotheken *SIMPL* und *zeroMQ* bei der Evaluation am besten abgeschnitten haben, werden diese Bibliotheken verwendet, um die Kommunikation zwischen Client und Server unter dem Betriebssystem Linux umzusetzen.
Wie die neue IPC-Bibliothek mit diesen externen Bibliotheken unter Linux umgesetzt wird, wird im Kapitel 3.6 beschrieben. Im nächsten Kapitel werden nun Testfälle beschrieben, gegen die die neue IPC-Bibliothek getestet wird.

3.5 Testszenarien

In diesem Kapitel werden Testszenarien vorgestellt, mit denen die neue IPC-Bibliothek unter dem Betriebssystem Linux getestet wird. Die Testbedingungen, die in den Testszenarien vorkommen werden, muss die neue IPC-Bibliothek erfüllen, um den Test zu bestehen.

3.5.1 Testszenario für das An- und Abmelden des Clients beim Server

Bei diesem Testszenario sollen ein oder mehrere Clients sich bei einem Server an- und abmelden können. Ebenfalls soll die synchrone Kommunikation getestet werden.
Bevor ein Client sich bei einem Server anmelden kann, muss eine Verbindung zwischen ihnen hergestellt sein. Wenn der Client keine Verbindung zum Server öffnen kann, kann er den Vorgang abbrechen. Üblicherweise wartet der Client für eine gewisse Zeit auf den Server, sofern der Server noch nicht gestartet wurde. Konnte der Server innerhalb dieser Zeit gestartet werden, stellt der Client eine Verbindung zu ihm her. Wenn die Zeit überschritten und der Server immer noch nicht gestartet wurde, soll der Vorgang des Verbindungsaufbaus abgebrochen werden.
Sobald die Verbindung zwischen dem Client und dem Server besteht, soll der Client sich beim Server anmelden. Dabei soll er seine Anmeldedaten in einer Nachricht synchron an den Server senden. In dieser Nachricht gibt der Client auch an, über

welches Ereignis er benachrichtigt werden möchte. Seine Ausführung wird solange blockiert, bis er eine Antwort vom Server erhält.
Der Server soll auf eingehende Nachrichten von Clients warten, die sich bei ihm an- oder abmelden möchten. Seine Aufgabe ist es, die Anmeldedaten der Clients in einer Liste zu verwalten.
Empfängt der Server synchron die in der Nachricht enthaltenen Anmeldedaten vom Client, soll er diese Daten in seine Liste aufnehmen. Der Server soll dann dem Client bestätigen, dass er die Nachricht des Clients empfangen hat. Sobald der Client die Antwort des Servers erhält, wird die Blockierung seiner Ausführung aufgehoben. Danach soll der Client weiterarbeiten können.
Der Vorgang des Abmeldens eines Clients bei einem Server verläuft genauso wie beim Anmelden. Wenn ein Client sich beim Server abmelden möchte, muss er nochmals seine Anmeldedaten in einer Nachricht synchron an den Server senden. Seine Ausführung wird solange blockiert, bis er eine Antwort vom Server erhält. Der Server empfängt synchron die in der Nachricht enthaltenen Anmeldedaten vom Client. Er soll diese Anmeldedaten mit den anderen Daten der angemeldeten Clients aus seiner Liste vergleichen, um den richtigen Client entfernen zu können. Der Server darf nicht irrtümlich den falschen noch bei ihm angemeldeten Client aus seiner Liste entfernen. Sobald der Server die Anmeldedaten des abgemeldeten Clients aus seiner Liste entfernt hat, soll er ihm den Empfang seiner Nachricht bestätigen. Sobald der Client die Antwort des Servers erhält, wird die Blockierung seiner Ausführung aufgehoben. Danach soll der Client weiterarbeiten können.
Der abgemeldete Client darf keine Benachrichtigungen über Ereignisse, die beim Server auftreten, erhalten.
Wenn alle registrieren Clients sich beim Server abmelden, muss der Server weiterlaufen. Er muss stets bereit sein, um neue Anmeldungen von Clients entgegenzunehmen und diese in seine Liste aufzunehmen.

3.5.2 Testszenario für die Benachrichtigung der beim Server registrieren Clients

Bei diesem Testszenario sollen die registrierten Clients eine Nachricht erhalten, wenn beim Server ein Ereignis auftritt. Ebenfalls soll die asynchrone Kommunikation getestet werden.
Wie im Kapitel 2.3 beschrieben, benachrichtigt der Client-Thread, der parallel zum Server-Thread läuft, die angemeldeten Clients über das Auftreten eines Ereignisses. Sobald ein Ereignis auftritt, soll der Client-Thread asynchron eine Nachricht an die angemeldeten Clients senden, die sich für das Auftreten dieses Ereignisses interessieren. Die übrigen, angemeldeten Clients sollen darüber nicht informiert werden. In dieser Nachricht informiert der Client-Thread die angemeldeten Clients, welches Ereignis aufgetreten ist. Sobald die angemeldeten Clients die Nachricht empfangen, sollen sie diese auswerten. Durch die Auswertung der empfangenen Nachricht wissen sie, welches Ereignis aufgetreten ist und wie sie entsprechend darauf reagieren sollen, beziehungsweise welche Aktionen sie ausführen müssen.

3.6 Umsetzung der neuen IPC-Bibliothek mit der aus der Evaluation resultierenden Bibliothek SIMPL und zeroMQ

Für die Umsetzung der neuen IPC-Bibliothek unter dem Betriebssystem Linux werden jeweils die externen Bibliotheken *SIMPL* und *zeroMQ* verwendet. Diese externen Bibliotheken wurden für diese Umsetzung auserwählt, da sie bei der Evaluation, die im Kapitel 3.4 beschrieben ist, am besten abgeschnitten haben.
Das eindimensionale Array `ipc_task[]`, das im Kapitel 2.4 erwähnt wurde, wird in die neue IPC-Bibliothek übernommen, unabhängig davon, ob sie mit *SIMPL* oder *zeroMQ* umgesetzt wird. Die Konstanten und Strukturen, die in dem API lib_ipc.h definiert sind, werden größtenteils in die neue IPC-Bibliothek übernommen, sofern sie nicht QNX-spezifisch sind.
In den nachfolgenden Kapiteln wird die Umsetzung der neuen IPC-Bibliothek mit den beiden externen Bibliotheken *SIMPL* und *zeroMQ* beschrieben. Im Anschluss dieses Kapitels werden diese beiden externen Bibliotheken miteinander verglichen.

3.6.1 Umsetzung der neuen IPC-Bibliothek mit SIMPL

In diesem Kapitel werden die Funktionen der neuen IPC-Bibliothek vorgestellt, die mit Hilfe der externen Bibliothek *SIMPL* umgesetzt wurden.
Die Dokumentation der in den neuen Funktionen verwendeten SIMPL-Funktionen ist im Anhang B zu finden.

Listing 35: Die Funktion der neuen IPC-Bibliothek `mi_create_server()` wurde mit der externen Bibliothek *SIMPL* umgesetzt

```
int mi_create_server (const char* name);
```

Die Funktion `mi_create_server()` ist eine Wrapper-Funktion, die die SIMPL-Funktion `name_attach()` aufruft. Sie ermöglicht einem Server, Nachrichten empfangen zu können.
Dem formalen Parameter `name` wird der Name des Servers übergeben. Intern wird dieser Parameter als erster Übergabeparameter an die SIMPL-Funktion `name_attach()` übergeben. Als zweiter Übergabeparameter wird der Wert NULL an diese SIMPL-Funktion übergeben. Der Grund dieser Übergabe ist, dass keine Funktion benötigt wird, die vor dem Beenden des Servers noch ausgeführt werden soll.
Der Rückgabewert der Funktion `mi_create_server()` ist vom Datentyp *int*. Wenn diese Funktion erfolgreich ausgeführt wurde, liefert sie den Wert null zurück. Im Fehlerfall liefert sie den Wert -1 zurück.

Listing 37: Die Funktion der neuen IPC-Bibliothek `mi_create_msgbox()` wurde mit der externen Bibliothek *SIMPL* umgesetzt

```
int mi_create_msgbox (const char* name);
```

Die Umsetzung der Funktion `mi_create_msgbox()` ist identisch mit der Umsetzung der Funktion `mi_create_server()`. Der Unterschied hierbei ist nur, dass anstelle des Servernamens der Name des Clients an den formalen Parameter `name` übergeben wird. Die Funktion `mi_create_msgbox()` ermöglicht einem Client, Nachrichten empfangen zu können.

Listing 36: Die Funktion der neuen IPC-Bibliothek `mi_wait_for_server()` wurde mit der externen Bibliothek *SIMPL* umgesetzt

```
int mi_wait_for_server (unsigned int target_id, unsigned int wait);
```

Die Funktion `mi_wait_for_server()` ermöglicht einem Client, eine Verbindung zu einem Server herzustellen, mit dem er kommunizieren möchte. Diese Funktion ist ein Abbild der Funktion `wait_for_task()` des APIs lib_ipc.h. Die Implementierung dieser Funktion wird in die neue Funktion `mi_wait_for_server()` übernommen. Anstelle der QNX-Funktion `name_open()` wird hier die SIMPL-Funktion `name_locate()` aufgerufen. Dabei wird der Name des Servers an `name_locate()` übergeben, der sich an der Position `target_id` im Array `ipc_task[]` befindet.
Die Beschreibung des formalen Parameters `wait` entspricht die des formalen Parameters `wait` der Funktion `wait_for_task()` aus dem Kapitel 2.4. Der Client soll innerhalb der Zeit `wait` eine Verbindung zum Server herstellen.
Der Rückgabewert der Funktion `mi_wait_for_server()` ist vom Datentyp *int*.
Wenn diese Funktion nicht fehlerfrei ausgeführt wurde, liefert sie einen Fehler zurück. Ansonsten liefert sie den Wert null zurück.

In den Funktionen, die im Folgenden beschrieben werden, werden die Strukturen *struct _CLIENT_PULSE_TYPE_T* und *struct _SERVER_PULSE_TYPE_T* verwendet, die aus dem Kapitel 2.4 bekannt sind. Für die nachfolgenden Funktionen werden diese Strukturen modifiziert. Die *sigevent* Struktur `ev` wird sowohl aus der Struktur *struct _CLIENT_PULSE_TYPE_T* als auch aus der Struktur *struct _SERVER_PULSE_TYPE_T* entfernt. Der Grund ist, dass das QNX-Makro `SIGEV_PULSE_INIT()` nicht mit dem Linux Betriebssystem kompatibel ist und somit die *sigevent* Struktur nicht mit diesem Makro initialisiert werden kann.
Es werden den beiden Strukturen drei neue Elemente hinzugefügt: `name`, `notification_nr` und `value`. Das Element `name` ist ein *char*-Array und die Elemente `notification_nr` und `value` sind vom Datentyp *int*.
Die nachfolgende Funktion, die nun beschrieben wird, ist `mi_connect_server()`.

Listing 38: Die Funktion der neuen IPC-Bibliothek `mi_connect_server()` wurde durch die Verwendung der externen Bibliothek *SIMPL* neu umgesetzt

```
int mi_connect_server (int target_id, int notification_nr, int value, const char* name);
```

Die Funktion `mi_connect_server()` ermöglicht einem Client, sich bei einem Server anzumelden.
Die Implementierung der Funktion `ipc_target_atta_value()` des APIs lib_ipc.h wird größtenteils in die neue Funktion `mi_connect_server()` übernommen.
Über den formalen Parameter `target_id` bestimmt der Client, bei welchem Server aus dem Array `ipc_task[]` er sich anmelden möchte. Dieser Parameter wird auch für die später aufrufende Funktion `mi_wait_for_server()` benötigt. Über den formalen Parameter `notification_nr` kann der Client bestimmen, über welches Ereignis er benachrichtigt werden möchte. Dem formalen Parameter `value` übergibt der Client Daten, die er dem Server übermitteln möchte. Dem formalen Parameter `name` übergibt der Client seinen Namen, über den der Client lokalisiert und angesprochen werden kann.
Der Rückgabewert der Funktion `mi_connect_server()` ist vom Datentyp *int*. Die Funktion liefert einen Status zurück, ob diese Funktion fehlerfrei oder fehlerhaft ausgeführt wurde.
Die Anmeldedaten des Clients werden in die modifizierte Struktur *struct _CLIENT_PULSE_TYPE_T* aufgenommen, die dann an den Server geschickt werden. Der Wert des formalen Parameters `target_id` und die Konstante `ATTA_CMD` werden der Variablen `target` und `cmd`, die Bestandteile der modifizierten Struktur *struct _CLIENT_PULSE_TYPE_T* sind, zugewiesen. Die Werte der formalen Parameter `notification_nr`, `value` und `name` werden den Variablen `notification_nr`, `value` und `name` dieser Struktur zugewiesen.
Bevor diese Daten an den Server geschickt werden, wird anstelle der Funktion `wait_for_task()` die neue Funktion `mi_wait_for_server()` mit dem Übergabeparameter `target_id` und dem Wert null aufgerufen. Sie überprüft, ob die Verbindung zum Server hergestellt wurde. Steht die Verbindung zum Server, werden die Daten des Clients mit der Funktion `mi_msg_send_sync()` synchron an den Server gesendet. Sobald der Empfang der Nachricht vom Server mit der Funktion `mi_msg_sync_reply()` bestätigt wird, wird die Funktion `mi_connect_server()` erfolgreich verlassen. Die Umsetzung dieser beiden Funktionen wird später im weiteren Verlauf beschrieben.

Listing 39: Die Funktion der neuen IPC-Bibliothek `mi_disconnect_server()` wurde durch die Verwendung der externen Bibliothek *SIMPL* neu umgesetzt

```
int mi_disconnect_server (int target_id, int notification_nr,
                          int value, const char* name);
```

Mit der Funktion `mi_disconnect_server()` meldet sich ein Client bei einem

Server ab.
Die Implementierung der Funktion `ipc_target_deta_value()` des APIs lib_ipc.h wird größtenteils in die neue Funktion `mi_disconnect_server()` übernommen.
Der Client meldet sich bei dem Server ab, der sich an der Position `target_id` im Array `ipc_task[]` befindet. Dieser Parameter wird auch für die später aufrufende Funktion `mi_wait_for_server()` benötigt. Die Beschreibung der formalen Parameter `notification_nr`, `value` und `name` entspricht die der formalen Parameter von der Funktion `mi_connect_server()`.
Der Rückgabewert der Funktion `mi_disconnect_server()` ist vom Datentyp *int*. Diese Funktion liefert einen Status zurück, ob sie fehlerfrei oder fehlerhaft ausgeführt wurde.
Die Daten des Clients werden für die Abmeldung beim Server in die modifizierte Struktur *struct _CLIENT_PULSE_TYPE_T* aufgenommen. Der Wert vom formalen Parameter `target_id` und die Konstante `DETA_CMD` werden den Variablen `target` und `cmd` der modifizierten Struktur zugewiesen. Die Werte der formalen Parameter `notification_nr`, `value` und `name` werden den Variablen `notification_nr`, `value` und `name` der modifizierten Struktur zugewiesen.
Bevor die Daten des Clients an den Server geschickt werden, wird die Funktion `mi_wait_for_server()` mit dem Übergabeparameter `target_id` und dem Wert null aufgerufen. Diese Funktion überprüft, ob die Verbindung zum Server noch existiert. Steht die Verbindung zum Server noch, werden diese Daten mit der Funktion `mi_msg_send_sync()` synchron an den Server gesendet. Sobald der Empfang der Nachricht vom Server mit der Funktion `mi_msg_sync_reply()` bestätigt wird, wird die Funktion `mi_disconnect_server()` erfolgreich verlassen. Die Umsetzung dieser beiden Funktionen wird im Folgenden nun beschrieben.

Listing 40: Die Funktion der neuen IPC-Bibliothek `mi_msg_send_sync()` wurde mit der externen Bibliothek *SIMPL* umgesetzt

```
int mi_msg_send_sync (int id, void* out_msg, unsigned int out_msg_size,
                      void* in_msg, unsigned int in_msg_size);
```

Die Funktion `mi_msg_send_sync()` ist eine Wrapper-Funktion, die die SIMPL-Funktion `Send()` aufruft.
Die Beschreibung der formalen Parameter `id`, `out_msg`, `out_msg_size`, `in_msg` und `in_msg_size` entspricht die der formalen Parameter `id`, `out`, `outSize`, `in` und `inSize` der SIMPL-Funktion `Send()`.
Im Erfolgsfall liefert die Funktion `mi_msg_send_sync()` den Wert null zurück, ansonsten den Wert -1.

Listing 41: Die Funktion der neuen IPC-Bibliothek `mi_msg_sync_reply()` wurde mit der externen Bibliothek *SIMPL* umgesetzt

```
int mi_msg_sync_reply (char* sender, void* out_msg, unsigned int out_msg_size);
```

Die Funktion `mi_msg_sync_reply()` ist eine Wrapper-Funktion, die die SIMPL-Funktion `Reply()` aufruft.
Die Beschreibung der formalen Parameter `sender`, `out_msg` und `out_msg_size` entspricht die der formalen Parameter `ptr`, `outArea` und `size` der SIMPL-Funktion `Reply()`.
Im Erfolgsfall liefert die Funktion `mi_msg_sync_reply()` den Wert null zurück, ansonsten den Wert -1.

Listing 42: Die Funktion der neuen IPC-Bibliothek `mi_msg_send_async()` wurde mit der externen Bibliothek *SIMPL* umgesetzt

```
int mi_msg_send_async (int id, int code, int value);
```

Die Funktion `mi_msg_send_async()` ist eine Wrapper-Funktion, die die SIMPL-Funktion `Trigger()` aufruft.
Dem formalen Parameter `id` wird der Rückgabewert der SIMPL-Funktion `name_locate()` übergeben. Dem formalen Parameter `code` wird ein *int*-Wert und dem formalen Parameter `value` werden Daten übergeben, die an den Empfänger gesendet werden. Die Daten müssen vom Datentyp *int* sein. Beim Empfang des Codes `code` weiß der Empfänger, wie er entsprechend darauf reagieren muss.
Wenn zum Beispiel beim Server ein Ereignis aufgetreten ist, benachrichtigt der Client-Thread, der parallel zum Server läuft, mit der Funktion `mi_msg_send_async()` die angemeldeten Clients über dieses Ereignis. Dabei übergibt der Client-Thread dem formalen Parameter `code` ein *int*-Wert, der aussagt, welches Ereignis aufgetreten ist. Wenn die angemeldeten Clients den Code erhalten, können sie anhand dieses Codes eindeutig identifizieren, welches Ereignis aufgetreten ist. Durch die Auswertung des Codes weiß der Client, wie er reagieren soll, beziehungsweise welche Aktionen er entsprechend ausführen muss.
Der nachfolgende Programmausschnitt zeigt, wie es möglich ist, zwei *int*-Werte der formalen Parameter `code` und `value` gemeinsam über die SIMPL-Funktion `Trigger()` zu senden:

Listing 43: Programmausschnitt aus der neuen Funktion `mi_msg_send_async()`

```
int info = 0;
int errvalue = 0;

info = code;
info = info << 16;
info = info | value;

errvalue = Trigger(id, info);
```

Der Datentyp *int* ist vier Bytes groß. Die ersten zwei Bytes werden für den Wert des formalen Parameters `code` reserviert, die letzten zwei Bytes für den Wert des formalen Parameters `value`.
Wie der Programmausschnitt (siehe Listing 43) zeigt, wird der Wert des formalen Parameters `code` der Variablen `info` zugewiesen, die vom Datentyp *int* ist. Da der Wert des formalen Parameters `code` die letzten zwei Bytes der Variablen `info` belegt, muss er um 16 Bit nach links verschoben werden. Nach der Verschiebung werden die ersten zwei Bytes der Variablen `info` vom Wert des formalen Parameters `code` belegt. Um die letzten zwei Bytes der Variablen `info` mit dem Wert des formalen Parameters `value` zu belegen, wird eine *ODER*-Verknüpfung verwendet.
Der Inhalt der Variablen `info` wird als *int*-Wert eines Proxys an den Empfänger asynchron gesendet.
Wenn die Funktion `mi_msg_send_async()` erfolgreich ausgeführt wurde, liefert sie den Wert null zurück. Im Fehlerfall liefert diese Funktion den Wert -1 zurück.

Listing 44: Die Funktion der neuen IPC-Bibliothek `mi_msg_receive()` wurde mit der externen Bibliothek *SIMPL* umgesetzt

```
int mi_msg_receive (char** sender, void* in_msg, unsigned int in_msg_size,
                    int* code, int* value);
```

Die Funktion `mi_msg_receive()` ist eine Wrapper-Funktion, die die SIMPL-Funktion `Receive()` aufruft.
Die Beschreibung der formalen Parameter `sender`, `in_msg` und `in_msg_size` entspricht die der formalen Parameter `ptr`, `inArea` und `maxBytes` von der SIMPL-Funktion `Receive()`. Die übrigen, formalen Parameter `code` und `value` sind *int*-Zeiger.
Der nachfolgende Programmausschnitt zeigt, welche Anweisungen ausgeführt werden müssen, wenn ein Empfänger einen Proxy empfängt:

Listing 45: Programmausschnitt aus der neuen Funktion `mi_msg_receive()`

```
int rcvid = 0;
int trigger_nr = 0;
int errvalue = 0;

rcvid = Receive(sender, in_msg, in_msg_size);

if (rcvid == -1)
{
    errvalue = rcvid;
    return errvalue;
}
else if (rcvid <= -2)
{
    trigger_nr = returnProxy(rcvid);

    *code = trigger_nr >> 16;
    *value = trigger_nr & 0x0000FFFF;

    return rcvid;
}
```

Wie der Programmausschnitt (siehe Listing 45) zeigt, liefert die SIMPL-Funktion `returnProxy()` als Rückgabewert ein *int*-Wert des Proxys zurück. Dieser *int*-Wert enthält den Code und die Daten, die der Variablen `trigger_nr` zugewiesen wird. Um den Code auslesen zu können, muss der *int*-Wert des Proxys um 16 Bit nach rechts verschoben werden. Dieser wird in die Variable kopiert, auf die der Zeiger `code` verweist. Um die Daten auslesen zu können, wird eine *UND*-Verknüpfung verwendet, die den *int*-Wert des Proxys mit dem Hex-Wert 0x0000FFFF verknüpft. Durch die *UND*-Verknüpfung können die Daten aus dem *int*-Wert des Proxys herausgefiltert und in die Variable kopiert werden, auf die der Zeiger `value` verweist.
Wenn die Funktion `mi_msg_receive()` erfolgreich ausgeführt wurde, liefert sie den Rückgabewert der SIMPL-Funktion `Receive()` zurück. Im Fehlerfall liefert die Funktion `mi_msg_receive()` den Wert -1 zurück.

Listing 46: Die Funktion der neuen IPC-Bibliothek `mi_new_member()` wurde durch die Verwendung der externen Bibliothek *SIMPL* neu umgesetzt

```
int mi_new_member (struct _SERVER_PULSE_TYPE_T *msg,
                   struct _SERVER_PULSE_TYPE_T *member_data_arr,
                   int max_member);
```

Mit der Funktion `mi_new_member()` nimmt ein Server die Anmeldedaten eines Clients in sein Array `member_data_arr` auf, der sich bei ihm anmeldet.
Die Implementierung der Funktion `pulse_new()` des APIs lib_ipc.h wird größtenteils in die neue Funktion `mi_new_member()` übernommen.
Der Zeiger `msg` ist vom Typ *struct _SERVER_PULSE_TYPE_T*, der unter anderem für die Funktion `mi_new_member()` modifiziert wurde. Die Beschreibung der formalen Parameter `member_data_arr` und `max_member` entspricht die der formalen Parameter `pulse_arr` und `max_pulse` der Funktion `pulse_new()` aus dem Kapitel 2.4.
Der Rückgabewert der Funktion `mi_new_member()` ist vom Datentyp *int*.
Der Inhalt an der Adresse, auf die der Zeiger `msg` verweist, enthält den Namen des Clients, den Wert der Variablen `notification_nr`, die Daten und die Receive ID, die von der Funktion `mi_msg_receive()` zurückgeliefert wurde. Dieser Inhalt wird in das Array `member_data_arr` aufgenommen. Wenn die im Inhalt enthaltenen Daten bereits im Array `member_data_arr` vorhanden sind oder keine neuen Daten eines Clients aufgenommen werden können, liefert `mi_new_member()` einen Fehler zurück. Ansonsten liefert diese Funktion den Wert null zurück, wenn sie fehlerfrei ausgeführt wurde.

Listing 47: Die Funktion der neuen IPC-Bibliothek `mi_del_member()` wurde durch die Verwendung der externen Bibliothek *SIMPL* neu umgesetzt

```
int mi_del_member (struct _SERVER_PULSE_TYPE_T *msg,
                   struct _SERVER_PULSE_TYPE_T *member_data_arr,
                   int max_member);
```

Mit der Funktion `mi_del_member()` entfernt ein Server die Daten eines Clients aus seinem Array `member_data_arr`, der sich bei ihm abmeldet.
Die Implementierung der Funktion `pulse_del()` des APIs lib_ipc.h wird größtenteils in die neue Funktion `mi_del_member()` übernommen.
Die Funktion `mi_del_member()` erwartet die gleichen Parameter wie die Funktion `mi_new_member()`. Der Rückgabewert von `mi_del_member()` ist vom Datentyp *int*.
Die Daten des abzumeldenden Clients, auf die der Zeiger `msg` verweist, werden mit den Einträgen des Arrays `member_data_arr` verglichen, um den Eintrag dieses Clients zu finden. Ist der Eintrag des abzumeldenden Clients im Array `member_data_arr` gefunden, wird sein Eintrag aus dem Array entfernt. Dabei wird sein Name gelöscht und die Variablen `rcvid`, `notification_nr` und `value` auf null zurückgesetzt.
Konnte keine Übereinstimmung gefunden werden, liefert die Funktion `mi_del_member()` einen Fehler zurück. Ansonsten liefert sie bei fehlerfreier Ausführung den Wert null zurück.

Listing 48: Die Funktion der neuen IPC-Bibliothek `mi_notify_member()` wurde durch die Verwendung der externen Bibliothek *SIMPL* neu umgesetzt

```
int mi_notify_member (struct _SERVER_PULSE_TYPE_T *member_data_arr,
                      int max_member, int notification_nr, int value);
```

Wie im Kapitel 2.4 unter dem Abschnitt pulse_deliver() beschrieben wurde, erfolgt die Benachrichtigung der angemeldeten Clients unter dem Betriebssystem QNX durch den Aufruf der QNX-Funktion `MsgDeliverEvent()`. Unter dem Betriebssystem Linux kann diese QNX-Funktion nicht mit minimalem Aufwand umgesetzt werden.
Entscheidend bei der Umsetzung der neuen Funktion `mi_notify_member()` ist nicht die Abbildung von `MsgDeliverEvent()`, sondern die Art der Benachrichtigung, wie die angemeldeten Clients durch diese QNX-Funktion benachrichtigt werden. Die registrierten Clients werden über die QNX-Funktion `MsgDeliverEvent()` asynchron benachrichtigt, dieses Szenario soll in der neuen Funktion `mi_notify_member()` umgesetzt werden.
Über den formalen Parameter `notification_nr` wird angegeben, welches Ereignis beim Server aufgetreten ist. Die Beschreibung der formalen Parameter `member_data_arr`, `max_member` und `value` entspricht die der formalen Parameter `pulse_arr`, `max_pulse` und `value` der Funktion `pulse_deliver()`.

Die Benachrichtigung der angemeldeten Clients erfolgt über die Funktion `mi_msg_send_async()`. Bevor diese Funktion aufgerufen wird, wird die SIMPL-Funktion `name_locate()` aufgerufen, um die angemeldeten Clients zu lokalisieren. Dabei werden die Namen der registrierten Clients aus dem Array `member_data_arr` nacheinander als Übergabeparameter an `name_locate()` übergeben. Es werden nur die registrierten Clients benachrichtigt, deren im Array `member_data_arr` gespeicherten Eintrag `notification_nr` dem formalen Parameter `notification_nr` entspricht.
Konnte ein registrierter Client zum Beispiel aufgrund seines Absturzes nicht lokalisiert werden, liefert die SIMPL-Funktion `name_locate()` den Wert -1 zurück. Die Daten des nicht lokalisierten Clients werden aus dem Array `member_data_arr` gelöscht. Ansonsten werden der Rückgabewert von `name_locate()` sowie die formalen Parameter `notification_nr` und `value` als Übergabeparameter an `mi_msg_send_async()` übergeben. Diese Funktion sendet asynchron den Wert des formalen Parameters `notification_nr` und die Daten `value` an die jeweiligen registrierten Clients, um sie über das auftretende Ereignis zu informieren.
Der Rückgabewert der Funktion `mi_notify_member()` ist vom Datentyp *int*. Wenn diese Funktion erfolgreich ausgeführt wurde, liefert sie den Wert null zurück. Ansonsten liefert sie im Fehlerfall den Wert -1 zurück.

3.6.2 Umsetzung der neuen IPC-Bibliothek mit zeroMQ

In diesem Kapitel werden die Funktionen der neuen IPC-Bibliothek beschrieben, die mit der externen Bibliothek *zeroMQ* umgesetzt wurden.
Die Dokumentation der in den neuen Funktionen verwendeten zeroMQ-Funktionen ist im Anhang C zu finden.

Listing 49: Die Funktion der neuen IPC-Bibliothek `mi_create_server()` wurde mit der externen Bibliothek *zeroMQ* umgesetzt

```
void* mi_create_server (const char* endpoint_path_reg,
                        const char* endpoint_path_inform, void** push, int* err);
```

Mit der Funktion `mi_create_server()` erzeugt ein Server zwei Sockets:
Ein Socket wird für den Empfang der Anmeldedaten eines Clients benötigt, der andere Socket wird dafür verwendet, angemeldete Clients asynchron über Ereignisse zu informieren.
Zu Beginn dieser Funktion wird die zeroMQ-Funktion `zmq_ctx_new()` aufgerufen, um einen zeroMQ *context* zu erzeugen.
Das An- und Abmelden eines Clients beim Server soll über die synchrone Kommunikation stattfinden. Daher wird beim Aufruf der zeroMQ-Funktion `zmq_socket()` der Socket Typ `ZMQ_REP` verwendet. Dieser Socket Typ und der *opaque handle*, der auf den neu erzeugten zeroMQ *context* verweist, werden als Übergabeparameter an

`zmq_socket()` übergeben. Der neu erzeugte Socket wird mit der zeroMQ-Funktion `zmq_bind()` an den Endpunkt `endpoint_path_reg` gebunden.
Die Benachrichtigung der registrierten Clients über bestimmte Ereignisse soll über die asynchrone Kommunikation stattfinden. Daher wird beim zweiten Aufruf der zeroMQ-Funktion `zmq_socket()` der Socket Typ `ZMQ_PUSH` verwendet. Genau wie beim ersten Aufruf wird der *opaque handle*, der auf den zweiten Socket verweist, und der Socket Typ `ZMQ_PUSH` an `zmq_socket()` übergeben. Der zweite Socket wird mit der zeroMQ-Funktion `zmq_bind()` an den Endpunkt `endpoint_path_inform` gebunden.
Der *opaque handle*, der auf den zweiten Socket verweist, wird in den Inhalt an der Adresse kopiert, auf die der formale Parameter `push` zeigt.
Der formale Parameter `err` ist ein Zeiger vom Datentyp *int*. Der Status, ob die Funktion `mi_create_server()` fehlerfrei oder fehlerhaft ausgeführt wurde, wird in die Variable kopiert, auf die `err` zeigt.
Der Rückgabewert der Funktion `mi_create_server()` ist ein *void*-Zeiger. Wenn diese Funktion fehlerfrei ausgeführt wurde, liefert sie den *opaque handle*, der auf den ersten Socket verweist, zurück. Im Fehlerfall liefert sie den Wert NULL zurück.

Listing 50: Die Funktion der neuen IPC-Bibliothek `mi_create_msgbox()` wurde mit der externen Bibliothek *zeroMQ* umgesetzt

```
void* mi_create_msgbox (const char* endpoint_path_reg,
                        const char* endpoint_path_inform, void** pull, int* err);
```

Mit der Funktion `mi_create_msgbox()` erzeugt ein Client zwei Sockets.
Über den einen Socket sendet der Client synchron seine Daten in einer Nachricht an den Server, bei dem er sich an- oder abmelden möchte. Über den anderen Socket wird der Client benachrichtigt, wenn bestimmte Ereignisse beim Server aufgetreten sind.
Die Struktur der Implementierung der Funktion `mi_create_msgbox()` ist ähnlich der von `mi_create_server()` aufgebaut. Anstelle des Socket Typs `ZMQ_REP` wird `ZMQ_REQ` als zweiter Übergabeparameter an die zeroMQ-Funktion `zmq_socket()` übergeben. Mit dem Aufruf der zeroMQ-Funktion `zmq_connect()` wird eine Verbindung zwischen dem neu erzeugten Socket des Clients und dem Endpunkt `endpoint_path_reg` hergestellt.
Das gleiche Verfahren verläuft ebenfalls beim Erzeugen des zweiten Sockets. Anstelle des Socket Typs `ZMQ_PUSH` wird `ZMQ_PULL` als zweiter Übergabeparameter an die Funktion `zmq_socket()` übergeben. Mit dem Aufruf der zeroMQ-Funktion `zmq_connect()` wird eine Verbindung zwischen dem zweiten Socket des Clients und dem Endpunkt `endpoint_path_inform` hergestellt.
Der *opaque handle*, der auf den zweiten Socket verweist, wird in den Inhalt an der Adresse kopiert, auf die der formale Parameter `pull` zeigt. Die Beschreibung des formalen Parameters `err` entspricht die des formalen Parameters `err` der Funktion `mi_create_server()`.

Der Rückgabewert der Funktion `mi_create_msgbox()` ist ein *void*-Zeiger. Im Erfolgsfall liefert diese Funktion den *opaque handle*, der auf den ersten Socket verweist, zurück. Im Fehlerfall liefert sie den Wert NULL zurück.

Für die Umsetzung der Kommunikation zwischen Client und Server mit der externen Bibliothek *zeroMQ* wird die Funktion `mi_wait_for_server()` nicht benötigt und daher nicht umgesetzt.
Der Grund ist folgender: *zeroMQ* ermöglicht, dass ein Client, der die zeroMQ-Funktion `zmq_connect()` aufruft, vor einem Server gestartet werden kann, der die zeroMQ-Funktion `zmq_bind()` verwendet. Wenn der Client gegebenenfalls Nachrichten in seinen Socket geschrieben hat, bevor der Server aktiv wird, werden die Nachrichten in die *message queue* abgelegt. Sobald der Server gestartet wurde und er die zeroMQ-Funktion `zmq_bind()` aufruft, beginnt *zeroMQ* intern, die Nachrichten an den Server zu übermitteln [18].
Daher muss der Client nicht auf den Server warten, bis dieser aktiv wird.

Die nachfolgenden Funktionen verwenden die Strukturen *struct _CLIENT_PULSE_TYPE_T* und *struct _SERVER_PULSE_TYPE_T*, die aus dem Kapitel 2.4 bekannt sind. Für die Umsetzung der nachfolgenden Funktionen müssen diese Strukturen modifiziert werden. Die *sigevent* Struktur `ev` wird sowohl aus der Struktur *struct _CLIENT_PULSE_TYPE_T* als auch aus der Struktur *struct _SERVER_PULSE_TYPE_T* entfernt. Der Grund ist, dass das QNX-Makro `SIGEV_PULSE_INIT()` nicht unter dem Linux Betriebssystem lauffähig ist und somit die *sigevent* Struktur nicht mit diesem Makro initialisiert werden kann.
Für die beiden Strukturen werden vier neue Elemente hinzugefügt: `req_socket`, `pull_socket`, `notification_nr` und `value`. Die Elemente `req_socket` und `pull_socket` sind *void*-Zeiger, die Elemente `notification_nr` und `value` sind vom Datentyp *int*.
Die nachfolgende Funktion, die nun vorgestellt wird, ist `mi_connect_server()`.

Listing 51: Die Funktion der neuen IPC-Bibliothek `mi_connect_server()` wurde durch die Verwendung der externen Bibliothek *zeroMQ* neu umgesetzt

```
int mi_connect_server (unsigned int target_id, int notification_nr, int value,
                       void* requester, void* puller);
```

Mit der Funktion `mi_connect_server()` kann sich ein Client bei einem Server anmelden.
Der formale Parameter `target_id` identifiziert im Array `ipc_task[]` den Server, mit dem der Client kommuniziert. Über den formalen Parameter `notification_nr` kann der Client bestimmen, über welches Ereignis er benachrichtigt werden möchte. Dem formalen Parameter `value` übergibt der Client Daten, die er dem Server übermitteln möchte. Den formalen Parametern `requester` und `puller` übergibt der Client die *opaque handles*, die auf seine beiden Sockets verweisen.

Der Rückgabewert der Funktion `mi_connect_server()` ist vom Datentyp *int*. Diese Funktion liefert einen Status zurück, ob sie fehlerfrei oder fehlerhaft ausgeführt wurde.
Die formalen Parameter werden in die modifizierte Struktur *struct _CLIENT_PULSE_TYPE_T* aufgenommen. Der Variablen `cmd`, die ein Bestandteil dieser modifizierten Struktur ist, wird die Konstante `ATTA_CMD` zugewiesen. Die in dieser Struktur enthaltenen Anmeldedaten werden synchron mit der Funktion `mi_msg_send_sync()` an den Server gesendet. Dabei wird der formale Parameter `requester` als erster Übergabeparameter an `mi_msg_send_sync()` übergeben.
Sobald der Empfang der Nachricht vom Server mit der Funktion `mi_msg_sync_reply()` bestätigt wird, wird die Funktion `mi_connect_server()` erfolgreich verlassen.
Die Umsetzung dieser beiden Funktionen wird später im weiteren Verlauf beschrieben.

Listing 52: Die Funktion der neuen IPC-Bibliothek `mi_disconnect_server()` wurde durch die Verwendung der externen Bibliothek *zeroMQ* neu umgesetzt

```
int mi_disconnect_server (unsigned int target_id, int notification_nr, int value,
                          void* requester, void* puller, const char* endpoint_path);
```

Mit der Funktion `mi_disconnect_server()` meldet sich ein Client bei einem Server ab.
Die Beschreibung der formalen Parameter `target_id`, `notification_nr`, `value`, `requester` und `puller` entspricht die der formalen Parameter der Funktion `mi_connect_server()`. Dem formalen Parameter `endpoint_path` wird der Dateipfad übergeben, der als Übergabeparameter an den formalen Parameter `endpoint_path_inform` der Funktion `mi_create_msgbox()` übergeben wurde.
Der Rückgabewert der Funktion `mi_disconnect_server()` ist vom Datentyp *int*. Diese Funktion liefert einen Status zurück, ob sie fehlerfrei oder fehlerhaft ausgeführt wurde.
Die formalen Parameter werden bis auf `endpoint_path` in die modifizierte Struktur *struct _CLIENT_PULSE_TYPE_T* aufgenommen. Der Variablen `cmd`, die ein Bestandteil dieser modifizierten Struktur ist, wird die Konstante `DETA_CMD` zugewiesen. Diese Daten, die nun in der Struktur *struct _CLIENT_PULSE_TYPE_T* enthalten sind, werden mit der Funktion `mi_msg_send_sync()` synchron an den Server gesendet. Dabei wird der formale Parameter `requester` als erster Übergabeparameter an `mi_msg_send_sync()` übergeben. Sobald der Empfang der Nachricht vom Server mit der Funktion `mi_msg_sync_reply()` bestätigt wird, wird die zeroMQ-Funktion `zmq_disconnect()` aufgerufen. Dabei wird der formale Parameter `puller` und `endpoint_path` an diese zeroMQ-Funktion übergeben. Die zeroMQ-Funktion `zmq_disconnect()` trennt die Verbindung zwischen dem Socket `puller` und dem Endpunkt `endpoint_path`, damit der abgemeldete Client keine weiteren Benachrichtigungen bezüglich auftretender Ereignisse beim Server empfängt.

Die Umsetzung der Funktionen `mi_msg_send_sync()` und `mi_msg_sync_reply()` werden nun im Folgenden beschrieben.

Listing 53: Die Funktion der neuen IPC-Bibliothek `mi_msg_send_sync()` wurde mit der externen Bibliothek *zeroMQ* umgesetzt

```
int mi_msg_send_sync (void* socket, void* out_msg, unsigned int out_msg_size,
                      void* in_msg, unsigned int in_msg_size);
```

Die Funktion `mi_msg_send_sync()` ermöglicht einem Prozess, eine Nachricht synchron an einen anderen Prozess zu senden.
Dem formalen Parameter `socket` wird der *opaque handle* übergeben, der auf den erzeugten Socket des sendenden Prozesses verweist. Die Nachricht, die der sendende Prozess schicken möchte, wird an den formalen Parameter `out_msg` mit der Größe `out_msg_size` übergeben. Die Antwort des empfangenen Prozesses wird im Puffer `in_msg` mit der Größe `in_msg_size` gespeichert.
Der Rückgabewert der Funktion `mi_msg_send_sync()` ist vom Datentyp *int*. Diese Funktion liefert einen Fehler zurück, wenn sie nicht fehlerfrei ausgeführt wurde. Im Erfolgsfall liefert sie den Wert null zurück.
Der nachfolgende Programmausschnitt zeigt, wie das synchrone Senden einer Nachricht mit den zeroMQ-Funktionen `zmq_send()` und `zmq_recv()` umgesetzt wurde:

Listing 54: Programmausschnitt aus der neuen Funktion `mi_msg_send_sync()`

```
int ret = 0;
...
ret = zmq_send (socket, out_msg, out_msg_size, 0);
...
ret = zmq_recv (socket, in_msg, in_msg_size, 0);
...
```

Die zeroMQ-Funktion `zmq_send()` sendet die Nachricht des Prozesses und blockiert die Ausführung dieses Prozesses nicht. Die Blockierung findet erst bei der zeroMQ-Funktion `zmq_recv()` statt. Über diese zeroMQ-Funktion wartet der sendende Prozess auf die Antwort des empfangenen Prozesses. Sobald der sendende Prozess eine Antwort vom empfangenen Prozess erhält, wird die Blockierung der Ausführung des sendenden Prozesses aufgehoben.
Die Funktion `mi_msg_send_sync()` wird erfolgreich verlassen.

Listing 55: Die Funktion der neuen IPC-Bibliothek `mi_msg_sync_reply()` wurde mit der externen Bibliothek *zeroMQ* umgesetzt

```
int mi_msg_sync_reply (void* socket, void* out_msg, unsigned int out_msg_size);
```

Mit der Funktion `mi_msg_sync_reply()` beantwortet ein Prozess eine Nachricht, die er von einem anderen Prozess empfangen hat.

Die Beschreibung der formalen Parameter `socket`, `out_msg` und `out_msg_size` entspricht die der formalen Parameter `socket`, `buf` und `len` der zeroMQ-Funktion `zmq_send()`.
Über diese zeroMQ-Funktion sendet der Prozess eine Antwort an den anderen Prozess, von dem er die Nachricht empfangen hat.
Der Rückgabewert der Funktion `mi_msg_sync_reply()` ist vom Datentyp *int*. Diese Funktion liefert einen Fehler zurück, wenn sie nicht fehlerfrei ausgeführt wurde. Ansonsten liefert sie im Erfolgsfall den Wert null zurück.

Listing 56: Die Funktion der neuen IPC-Bibliothek `mi_msg_send_async()` wurde mit der externen Bibliothek *zeroMQ* umgesetzt

```
int mi_msg_send_async (void* socket, void* out_msg, unsigned int out_msg_size);
```

Die Funktion `mi_msg_send_async()` ist eine Wrapper-Funktion, die die zeroMQ-Funktion `zmq_send()` aufruft.
Die Beschreibung der formalen Parameter `socket`, `out_msg` und `out_msg_size` entspricht die der formalen Parameter `socket`, `buf` und `len` der zeroMQ-Funktion `zmq_send()`.
Der Rückgabewert der Funktion `mi_msg_send_async()` ist vom Datentyp *int*. Diese Funktion liefert einen Fehler zurück, wenn sie nicht fehlerfrei ausgeführt wurde. Im Erfolgsfall liefert sie den Wert null zurück.

Listing 57: Die Funktion der neuen IPC-Bibliothek `mi_msg_receive()` wurde mit der externen Bibliothek *zeroMQ* umgesetzt

```
int mi_msg_receive (void* socket, void* in_msg, unsigned int in_msg_size);
```

Die Funktion `mi_msg_receive()` ist eine Wrapper-Funktion, die die zeroMQ-Funktion `zmq_recv()` aufruft.
Die Beschreibung der formalen Parameter `socket`, `in_msg` und `in_msg_size` entspricht die der formalen Parameter `socket`, `buf` und `len` der zeroMQ-Funktion `zmq_recv()`.
Der Rückgabewert der Funktion `mi_msg_receive()` ist vom Datentyp *int*. Diese Funktion liefert den Wert null zurück, wenn sie fehlerfrei ausgeführt wurde. Ansonsten liefert sie einen Fehler zurück.

Listing 58: Die Funktion der neuen IPC-Bibliothek `mi_new_member()` wurde durch die Verwendung der externen Bibliothek *zeroMQ* neu umgesetzt

```
int mi_new_member (struct _SERVER_PULSE_TYPE_T *msg,
                   struct _SERVER_PULSE_TYPE_T *member_data_arr,
                   int max_member);
```

Mit der Funktion `mi_new_member()` nimmt ein Server die Anmeldedaten eines Clients in sein Array `member_data_arr` auf, der sich bei ihm anmeldet.

Die Implementierung dieser Funktion ist identisch mit der Implementierung der Funktion `mi_new_member()`, die im Kapitel 3.6.1 beschrieben wurde. Der einzige Unterschied ist nur, dass das Array `member_data_arr` in dieser Funktion andere Daten aufnimmt. Diese Daten befinden sich im Inhalt an der Adresse, auf die der Zeiger `msg` verweist. Dieser Inhalt enthält die *opaque handles* des Clients, die auf seine beiden Sockets verweisen, den Wert der Variablen `notification_nr` und die Daten, die der Server vom Client erhalten hat.

Listing 59: Die Funktion der neuen IPC-Bibliothek `mi_del_member()` wurde durch die Verwendung der externen Bibliothek *zeroMQ* neu umgesetzt

```
int mi_del_member (struct _SERVER_PULSE_TYPE_T *msg,
                   struct _SERVER_PULSE_TYPE_T *member_data_arr,
                   int max_member);
```

Mit der Funktion `mi_del_member()` entfernt ein Server die Daten eines Clients aus seinem Array `member_data_arr`, der sich bei ihm abmeldet.
Die Implementierung dieser Funktion ist identisch mit der Implementierung der Funktion `mi_del_member()` aus dem Kapitel 3.6.1.
Der einzige Unterschied ist nur, dass anstelle des Namens des Clients und der Receive ID die *opaque handles*, die auf den Sockets des Clients verweisen, aus dem Array `member_data_arr` entfernt werden. Dabei werden im Eintrag des abzumeldenden Clients die *void*-Zeiger `req_socket` und `pull_socket` auf den Wert NULL zurückgesetzt.

Listing 60: Die Funktion der neuen IPC-Bibliothek `mi_notify_member()` wurde durch die Verwendung der externen Bibliothek *zeroMQ* neu umgesetzt

```
int mi_notify_member (void* pusher,
                      struct _SERVER_PULSE_TYPE_T *member_data_arr,
                      int max_member, int notification_nr, int value);
```

Wie in dem Abschnitt mi_notify_member() im Kapitel 3.6.1 beschrieben, soll die QNX-Funktion `MsgDeliverEvent()` nicht in der neuen Funktion `mi_notify_member()` umgesetzt werden. Die Benachrichtigung der registrierten Clients über auftretende Ereignisse soll asynchron erfolgen. Dieses Szenario soll in der neuen Funktion umgesetzt werden.
Dem formalen Parameter `pusher` wird der *opaque handle* übergeben, der auf den Socket des Servers verweist. Über den formalen Parameter `notification_nr` wird angegeben, welches Ereignis beim Server aufgetreten ist. Die Beschreibung der formalen Parameter `member_data_arr`, `max_member` und `value` entspricht die der formalen Parameter `pulse_arr`, `max_pulse` und `value` der Funktion `pulse_deliver()` aus dem Kapitel 2.4.
Es werden nur die registrierten Clients benachrichtigt, deren im Array `member_data_arr` gespeicherten Eintrag `notification_nr` dem formalen Parameter `notification_nr` entspricht. Diese registrierten Clients werden nach-

einander über das auftretende Ereignis benachrichtigt. Dabei werden ihre Daten, die der Server in sein Array `member_data_arr` aufgenommen hat, und die neuen Daten `value` mit der Funktion `mi_msg_send_async()` an sie gesendet. Diese Funktion erwartet als Übergabeparameter den formalen Parameter `pusher` und die zu übermittelten Daten.
Der Rückgabewert der Funktion `mi_notify_member()` ist vom Datentyp *int*. Wenn diese Funktion erfolgreich ausgeführt wurde, liefert sie den Wert null zurück. Ansonsten liefert sie einen Fehler zurück.

3.6.3 Vergleich zwischen SIMPL und zeroMQ

Die im Kapitel 3.6.1 und 3.6.2 umgesetzten Funktionen der neuen IPC-Bibliothek mit den externen Bibliotheken *SIMPL* und *zeroMQ* erfüllen die Testbedingungen, die im Kapitel 3.5 definiert wurden.
Die neuen Funktionen mit der externen Bibliothek *SIMPL* waren einfach und nicht zeitintensiv umzusetzen. Der Grund ist die Ähnlichkeit einiger SIMPL-Funktionen mit den QNX-Funktionen, die im Kapitel 2.1 beschrieben wurden. Durch die erworbenen Kenntnisse über diese QNX-Funktionen hat man eine bessere Vorstellung, wie die entsprechend ähnlichen SIMPL-Funktionen funktionieren. Durch diese Ähnlichkeit und die verständliche Beschreibung der SIMPL-Funktionen in der Dokumentation konnten die Funktionen der neuen IPC-Bibliothek mit wenig Aufwand implementiert werden. Das An- und Abmelden mehrerer Clients bei einem Server über die synchrone Kommunikation konnte ohne Probleme mit den SIMPL-Funktionen `Send()`, `Receive()` und `Reply()` implementiert werden.
Ein Nachteil der externen Bibliothek *SIMPL* ist, dass bei der SIMPL-Funktion `Trigger()` nur ein *int*-Wert asynchron an den Empfänger gesendet werden kann. Strukturen, in den Elemente verschiedener Datentypen enthalten sein können, können nicht asynchron über diese SIMPL-Funktion gesendet werden. Ein weiterer Nachteil ist, dass einige Beispiele aus der Dokumentation zu den SIMPL-Funktionen fehlerbehaftet sind. Diese Fehler hatten jedoch keine Auswirkungen auf die Verständlichkeit der SIMPL-Funktionen.
Im Gegensatz zur externen Bibliothek *SIMPL* wurde für die Funktionen, die mit der externen Bibliothek *zeroMQ* umgesetzt wurden, viel Zeit benötigt. In ihrer Dokumentation existieren neben dem request-reply Pattern und dem Pipeline Pattern noch weitere, unter anderem auch erweiterte, *messaging patterns*, die die externe Bibliothek *zeroMQ* anbietet. Es wurde viel Zeit benötigt, um die passenden Patterns für die Umsetzung der Kommunikation zwischen Client und Server zu finden.
Nachdem die passenden Patterns gefunden wurden, konnten mit Hilfe der verständlichen Beschreibung der zeroMQ-Funktionen in der Dokumentation die synchrone und die asynchrone Kommunikation ohne Probleme implementiert werden. Daher konnten das An- und Abmelden mehrerer Clients bei einem Server und die Benachrichtigung registrierter Clients mit wenig Aufwand in den neuen Funktionen umgesetzt werden.

Bei der Benachrichtigung der registrierten Clients ermöglicht die zeroMQ-Funktion `zmq_send()` – im Gegensatz zur SIMPL-Funktion `Trigger()` – ihre Daten in einer Struktur asynchron an sie zu senden.

Der Vergleich dieser beiden externen Bibliotheken zeigt, dass sie ihre Vorteile, aber auch ihre Nachteile haben. Der entscheidende Vorteil bei *SIMPL* ist, dass einige ihrer Funktionen gewisse Ähnlichkeiten mit den entsprechenden QNX-Funktionen haben. Daher können die SIMPL-Funktionen größtenteils die QNX-Funktionen in den Funktionen ersetzen, die im Kapitel 2.4 beschrieben sind. Für die Verständlichkeit der externen Bibliothek *SIMPL* wurde im Gegensatz zu *zeroMQ* nicht viel Zeit benötigt. Daher ist der Aufwand die neuen Funktionen mit *SIMPL* zu implementieren, geringer als bei *zeroMQ*.

4 Resultate

Die Analysen und Umsetzungen im Kapitel 3 haben gezeigt, dass die Kommunikation zwischen Client und Server unter dem Linux Betriebssystem umsetzbar sind. Ebenfalls können shared memory Bereiche im Linux Betriebssystem für den Datenaustausch angelegt werden.
Im Kapitel 3.6 konnte gezeigt werden, dass die synchrone und die asynchrone Kommunikation jeweils mit Hilfe von *SIMPL* und *zeroMQ* umgesetzt werden können. Durch die Arbeit mit diesen externen Bibliotheken konnte festgestellt werden, dass sie auf systemabhängigen IPC-Mechanismen basieren. Die externe Bibliothek *SIMPL* basiert auf *shared memory* und *FIFOs*, die externe Bibliothek *zeroMQ* basiert auf *message queues*. Bei der Verwendung der externen Bibliothek *zeroMQ* bezüglich der lokalen Kommunikation ist es sogar erforderlich, dass ein Betriebssystem die Unix-Domain-Sockets unterstützen muss.
Es ist also möglich, durch Kombination von IPC-Mechanismen die synchrone und die asynchrone Kommunikation unter dem Linux Betriebssystem umzusetzen. Der Aufwand für die Umsetzung der Kommunikation mit den IPC-Mechanismen wird enorm hoch und zeitintensiv sein. Es besteht außerdem das Risiko, dass man bei Unachtsamkeit bei der Verwendung der IPC-Mechanismen einen Deadlock produzieren kann. Daher ist es empfehlenswert, die externen Bibliotheken weiterhin zu verwenden.
In Bezug auf die im Kapitel 2.1 beschriebenen QNX-Funktionen konnten ihre Verhalten bezüglich der Kommunikation größtenteils unter dem Betriebssystem Linux abgebildet werden. Das Ziel bei der Abbildung der QNX-Funktionen unter Linux war es, die nötigen Funktionsweisen und Verhalten dieser QNX-Funktionen unter diesem Betriebssystem entsprechend umzusetzen. Das Ziel war nicht, jede einzelne Funktionalität, die die QNX-Funktionen anbietet, exakt unter Linux abzubilden. Daher wurde auch bei der QNX-Funktion `MsgDeliverEvent()` nicht ihre Funktionalität unter Linux abgebildet, sondern ihr Verhalten bezüglich der Kommunikation. Die Art der Benachrichtigung, wie die registrierten Clients asynchron über auftretende Ereignisse beim Server informiert werden, wurde unter Linux umgesetzt.
Wie unter dem Betriebssystem QNX können auch unter dem Betriebssystem Linux Kommunikationsendpunkte erzeugt werden. Anstelle von Channels werden zum Beispiel bei *zeroMQ* Sockets als Kommunikationsendpunkte erzeugt.
Die synchrone Kommunikation unter dem Betriebssystem QNX konnte größtenteils unter Linux umgesetzt werden. Die Blockierung der Ausführung eines Senders unter QNX konnte nicht exakt unter Linux abgebildet werden. Während es unter QNX eine Unterscheidung der Blockierung zwischen *SEND blocked*, *RECEIVE blocked* und *REPLY blocked* gibt, wird unter Linux nur zwischen der Blockierung des Sendens und des Empfangens unterschieden.

Zusammenfassend kann man feststellen, dass eine Abbildung der Verhalten der QNX-Basisfunktionen bezüglich der Kommunikation unter dem Betriebssystem Linux größtenteils möglich ist. Eine exakte Abbildung der Funktionalitäten der QNX-Funktionen unter Linux ist jedoch sehr schwierig und komplex umzusetzen.

5 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde ein Konzept erstellt, wie die in der firmenspezifischen IPC-Bibliothek verwendeten QNX-Basisfunktionen unter dem Betriebssystem Linux umzusetzen sind.
Es war zu Beginn erforderlich, die firmenspezifische IPC-Bibliothek zu analysieren, besonders die QNX- und POSIX-Funktionen, die in dieser IPC-Bibliothek verwendet wurden. Es wurde untersucht, wie die firmeneigenen Programme sich verhalten, wenn sie über die firmenspezifische IPC-Bibliothek miteinander kommunizieren und Daten über shared memory Bereiche austauschen.
Auf dieser Analyse aufbauend wurden Anforderungen bezüglich der Erstellung einer neuen IPC-Bibliothek unter dem Betriebssystem Linux bestimmt. Für die neue IPC-Bibliothek wurden Funktionsprototypen in einem API definiert, die diese Anforderungen umsetzen. Die POSIX-Funktionen, die in der firmenspezifischen IPC-Bibliothek verwendet wurden, konnten in die neue IPC-Bibliothek übernommen werden, da sie unter Linux lauffähig sind.
Es wurde nach geeigneten IPC-Techniken, Frameworks und Bibliotheken gesucht, die den gestellten Anforderungen genügen. Anhand dieser und weiterer Anforderungen bezüglich der Dokumentation, Programmierung, Support sowie Preis und Lizenz wurde eine Evaluation durchgeführt. Sie bewertete, ob die in Frage kommenden IPC-Techniken, externe Frameworks und Bibliotheken die gestellten Anforderungen erfüllen können. Am Ende der Evaluation haben die externen Bibliotheken *SIMPL* und *zeroMQ* am besten abgeschnitten.
Es wurden Testszenarien erstellt, anhand derer die mit *SIMPL* und *zeroMQ* umgesetzten Funktionen der neuen IPC-Bibliothek später getestet wurden.
Nachdem diese Funktionen mit Hilfe der beiden externen Bibliotheken *SIMPL* und *zeroMQ* die erforderliche Kommunikation zwischen Programmen umgesetzt haben, wurden sie anhand der Testszenarien getestet. Sie haben die gestellten Testbedingungen erfüllt und somit die Testfälle erfolgreich bestanden.

Der nächste Schritt ist, eine IPC-Bibliothek zu erstellen, die sowohl unter QNX als auch unter Linux lauffähig sein sollte, da der Umstieg von QNX auf Linux sich über einen längeren Zeitraum hinziehen wird. Diese IPC-Bibliothek wird auf der firmenspezifischen IPC-Bibliothek und der unter Linux neu entwickelten IPC-Bibliothek basieren. Außerdem ist es wichtig, Schritt für Schritt die QNX-Funktionen wie `MsgReceive()` und `name_attach()`, die die firmeneigenen Programme explizit aufrufen, aus diesen Programmen auszulagern und in die firmenspezifische IPC-Bibliothek aufzunehmen. Diese Programme sollen zu einer geraumen Zeit vollständig von den QNX-Funktionen entkoppelt werden, so dass sie sowohl unter QNX als auch unter Linux lauffähig sein werden.

Anhang

A. Evaluationsmatrix der ausgeschiedenen Frameworks und Bibliotheken

	BSPlib	GAMMA	MSMQ	RabbitMQ
Preis / Lizenz	*Keine weitere Betrachtung dieses Programms.*	kostenfrei ***(GNU GPL)***	*Keine weitere Betrachtung dieses Programms.*	kostenfrei ***(Mozilla Public License)***
Betriebssystem	*Keine weitere Betrachtung dieses Programms.*	**Linux 2.6.24**	Windows (2000)	Solaris, BSD, **Linux**, MacOSX, TRU64, Windows, VxWorks
Programmiersprache	**C, C++**, Fortran	**C**	*Keine weitere Betrachtung dieses Programms.*	---
Kommunikation über den Nachrichtenaustausch	*Keine weitere Betrachtung dieses Programms.*	Ja	*Keine weitere Betrachtung dieses Programms.*	*Keine weitere Betrachtung dieses Programms.*
Version / Letzte Aktualisierung	1.4 / 30.09.1998	unbekannt / 15.04.2009	*Keine weitere Betrachtung dieses Programms.*	2.8.7 / 27.09.2012
Der Kandidat soll fehlerfrei sein	*Keine weitere Betrachtung dieses Programms.*	Nein	Nein	*Keine weitere Betrachtung dieses Programms.*
Information zum Typ:	Bibliothek	Bibliothek	Framework	message-oriented middleware

	BSPonMPI	**Mbus**	**libevent**	**POCO**
Preis / Lizenz	kostenfrei ***(GNU Lesser General Public License (LGPL))***	*Keine weitere Betrachtung dieses Programms.*	kostenfrei ***(BSD (Berkeley Software Distribution))***	kostenfrei ***(Boost Software License 1.0)***
Betriebssystem	*Keine weitere Betrachtung dieses Programms.*	*Keine weitere Betrachtung dieses Programms.*	*Keine weitere Betrachtung dieses Programms.*	Windows, **Linux**, Mac OS X, Solaris, HP-UX, AIX
Programmiersprache	**C**	**C, C++**	**C**	**C++**
Kommunikation über den Nachrichtenaustausch	*Keine weitere Betrachtung dieses Programms.*	*Keine weitere Betrachtung dieses Programms.*	Nicht die Art der Kommunikation, die gesucht wird.	Nicht die Art der Kommunikation, die gesucht wird.
Version / Letzte Aktualisierung	0.3 / 29.05.2010	1.15 / 07.11.2005	2.0.19 / 03.05.2012	1.4.4 / 19.09.2012
Der Kandidat soll fehlerfrei sein	Nein	*Keine weitere Betrachtung dieses Programms.*	*Keine weitere Betrachtung dieses Programms.*	*Keine weitere Betrachtung dieses Programms.*
Information zum Typ:	Bibliothek	light-weight message-oriented coordination protocol	network library	Bibliothek

	libnl	**YAMI4**	**LCM**	**MulticoreBSP**
Preis / Lizenz	kostenfrei ***(GNU Lesser General Public License)***	dual-licensed: • General Public License (GPL) • Boost Software License	kostenfrei ***(GNU Lesser General Public License)***	kostenfrei ***(GNU LGPL v3 license)***
Betriebssystem	**Linux**	*Keine weitere Betrachtung dieses Programms.*	*Keine weitere Betrachtung dieses Programms.*	**Linux**
Programmiersprache	**C**	**C++**, Ada, Java, .NET, Objective-C, Python	**C, C++**, C#, Java, Python	**C, C++**
Kommunikation über den Nachrichtenaustausch	Zwischen Kernel und user space Prozesse. Nicht die Art der Kommunikation, die gesucht wird.	Lokale Kommunikation auf einem Rechner wird nicht unterstützt.	Lokale Kommunikation auf einem Rechner wird nicht unterstützt.	Nicht die Art der Kommunikation, die gesucht wird.
Version / Letzte Aktualisierung	3.2.14 / 19.10.2012	1.6.0 / unbekannt (2007-2012)	0.9.0 / 06.05.2012	1.0.1 / 17.10.2012
Der Kandidat soll fehlerfrei sein	*Keine weitere Betrachtung dieses Programms.*	*Keine weitere Betrachtung dieses Programms.*	*Keine weitere Betrachtung dieses Programms.*	*Keine weitere Betrachtung dieses Programms.*
Information zum Typ:	Bibliothek	Bibliothek	Bibliothek	Bibliothek

B. Dokumentation der verwendeten SIMPL-Funktionen

Die externe Bibliothek *SIMPL* ermöglicht, dass Prozesse sowohl lokal als auch über das Netzwerk kommunizieren können. Die Kommunikation zwischen den Prozessen findet über den Austausch von Nachrichten statt. Diese Bibliothek bietet die Möglichkeit, dass die Prozesse synchron und auch asynchron miteinander kommunizieren können.
Intern findet der Nachrichtenaustausch zwischen den Prozessen über einen shared memory Bereich statt, wenn die Prozesse innerhalb eines Rechners miteinander kommunizieren. Der shared memory Bereich wird intern von dem Prozess erzeugt, der die SIMPL-Funktion `Send()` aufruft. Diese SIMPL-Funktion wird später im weiteren Verlauf näher erläutert. Nachdem ein shared memory Bereich vom sendenden Prozess erzeugt wurde, kann neben ihm auch ein empfangener Prozess auf den shared memory Bereich zugreifen.
Die Zugriffe auf den shared memory Bereich werden durch zwei FIFOs reguliert, die im Linux Betriebssystem angelegt werden. Über diesen FIFOs wird die shared memory ID zwischen den beiden Prozessen ausgetauscht, die den shared memory Bereich im System identifiziert. Über diese ID können der sendende und der empfangene Prozess auf den shared memory Bereich zugreifen. Durch die beiden FIFOs können die Zugriffe des sendenden und des empfangenen Prozesses synchronisiert werden, wer wann auf den shared memory Bereich zugreifen darf [17].

Listing 61: SIMPL-Funktion `name_attach()`

```
int name_attach (char* processName, void (*exitFunc)());
```

Die SIMPL-Funktion `name_attach()` muss von jedem Prozess aufgerufen werden, der über die externe Bibliothek *SIMPL* kommunizieren möchte. Durch den Aufruf dieser SIMPL-Funktion werden die FIFOs im Linux Betriebssystem angelegt.
Ein Prozess muss zuerst die SIMPL-Funktion `name_attach()` aufrufen, bevor er eine andere, beliebige SIMPL-Funktion aufruft.
Als ersten Übergabeparameter erwartet die SIMPL-Funktion `name_attach()` den Namen des Prozesses, der diese Funktion aufruft. Die FIFOs werden nach seinem Namen, kombiniert mit dem PID (process identification), benannt. Somit wird der Name des Prozesses im System bekannt gemacht. Dieser Name darf nur einmalig vergeben werden, solange die FIFOs mit diesem Namen im Linux Betriebssystem existieren. Der Name darf nicht länger als `MAX_PROGRAM_NAME_LEN` (31) sein, ansonsten wird der Name gekürzt, bis er genau 31 Zeichen enthält.
Als zweiten Übergabeparameter wird ein Funktionszeiger übergeben, der auf eine Funktion verweist. Diese Funktion soll ausgeführt werden, wenn der Prozess beendet wird.
Der Rückgabewert der SIMPL-Funktion `name_attach()` ist vom Datentyp *int*. Wenn sie erfolgreich ausgeführt wurde, liefert sie den Wert null zurück, ansonsten den Wert -1 [17].

Listing 62: SIMPL-Funktion `name_locate()`

```
int name_locate (char* protocolName:hostName:processName);
```

Mit der SIMPL-Funktion `name_locate()` stellt ein Prozess eine Verbindung zu einem anderen Prozess her, an den er eine Nachricht senden möchte. Für die lokale Kommunikation genügt es, wenn der sendende Prozess den Namen des empfangenen Prozesses als Übergabeparameter an `name_locate()` übergibt. Der sendende Prozess braucht nicht den Protokoll- und den Hostnamen des empfangenen Prozesses anzugeben.
Der Rückgabewert der SIMPL-Funktion `name_locate()` ist vom Datentyp *int*. Diese SIMPL-Funktion liefert die ID des empfangenen Prozesses als Rückgabewert zurück, wenn sie erfolgreich ausgeführt wurde. Ansonsten liefert sie im Fehlerfall den Wert -1 zurück [17].

Listing 63: SIMPL-Funktion `Send()`

```
int Send (int id, void* out, void* in, unsigned outSize, unsigned inSize);
```

Mit der SIMPL-Funktion `Send()` sendet ein Prozess synchron eine Nachricht an einen anderen Prozess, der diese Nachricht erhalten soll. Die Ausführung des sendenden Prozesses wird solange blockiert, bis er eine Antwort vom empfangenen Prozess erhält.
Der sendende Prozess übergibt dem formalen Parameter `id` die ID, die er als Rückgabewert von der SIMPL-Funktion `name_locate()` erhalten hat. Die Nachricht, die der sendende Prozess schicken möchte, wird an den formalen Parameter `out` mit der Größe `outSize` in Bytes übergeben. Die Antwort des empfangenen Prozesses wird im Puffer `in` gespeichert. Der formale Parameter `inSize` gibt die Größe dieses Puffers in Bytes an.
Der Rückgabewert der SIMPL-Funktion `Send()` ist vom Datentyp *int*. Wenn diese SIMPL-Funktion erfolgreich ausgeführt wurde, liefert sie als Rückgabewert die Größe der Antwort des empfangenen Prozesses zurück. Im Fehlerfall liefert die SIMPL-Funktion `Send()` den Wert -1 zurück [17].

Listing 64: SIMPL-Funktion `Receive()`

```
int Receive (char** ptr, void* inArea, unsigned maxBytes);
```

Mit der SIMPL-Funktion `Receive()` empfängt ein Prozess eine Nachricht, die ein anderer Prozess ihm gesendet hat.
Als ersten Übergabeparameter wird die Adresse einer Variablen vom Datentyp *char** an den formalen Parameter `ptr` übergeben. Dieser Parameter identifiziert den sendenden Prozess, der die Nachricht geschickt hat. Der Name des sendenden Prozesses wird in der Variablen abgelegt. Sie wird später als Übergabeparameter für die SIMPL-Funktion `Reply()` benötigt. Die empfangene Nachricht wird im Puffer `inArea` mit der Größe `maxBytes` gespeichert.
Der Rückgabewert der SIMPL-Funktion `Receive()` ist vom Datentyp *int*. Wenn der Rückgabewert größer gleich null ist, bedeutet es, dass der empfangene Prozess eine Nachricht empfangen hat, auf die er antworten muss. Der positive Rückgabewert ist

die Größe der eingehenden Nachricht. Wenn der Rückgabewert kleiner gleich -2 ist, bedeutet es, dass der empfangene Prozess einen sogenannten *„Proxy“* empfangen hat. Dieser Begriff wird später im Verlauf der Beschreibung der übrigen SIMPL-Funktionen näher erläutert. Die SIMPL-Funktion `Receive()` liefert im Fehlerfall den Wert -1 zurück [17].

Listing 65: SIMPL-Funktion `Reply()`

```
int Reply (char* ptr, void* outArea, unsigned size);
```

Mit der SIMPL-Funktion `Reply()` antwortet der empfangene Prozess auf die Nachricht des sendenden Prozesses, der diese Nachricht mit der SIMPL-Funktion `Send()` gesendet hat.
Als ersten Übergabeparameter wird der Name des sendenden Prozesses übergeben, der durch die SIMPL-Funktion `Receive()` identifiziert wurde. Die Antwort des empfangenen Prozesses wird im Puffer `outArea` mit der Größe `size` gespeichert.
Die SIMPL-Funktion `Reply()` liefert im Erfolgsfall den Wert null zurück, ansonsten den Wert -1 [17].

Listing 66: SIMPL-Funktion `Trigger()`

```
int Trigger (int id, int proxy);
```

Mit der SIMPL-Funktion `Trigger()` kann ein Prozess asynchron einen *Proxy* an einen anderen Prozess senden.
Ein Proxy ist ein *int*-Wert, der die Werte zwischen 0 und 7FFFFFFF annehmen darf.
Als ersten Übergabeparameter wird die ID übergeben, die der sendende Prozess durch den Aufruf der SIMPL-Funktion `name_locate()` als Rückgabewert erhalten hat. Dem formalen Parameter `proxy` wird der oben genannte *int*-Wert übergeben, der an den empfangenen Prozess gesendet werden soll.
Wenn die SIMPL-Funktion `Trigger()` erfolgreich ausgeführt wurde, liefert sie den Wert null zurück. Im Fehlerfall liefert diese SIMPL-Funktion den Wert -1 zurück [17].

Listing 67: SIMPL-Funktion `returnProxy()`

```
int returnProxy (int proxyNumber);
```

Der *int*-Wert des Proxys, den ein Prozess mit der SIMPL-Funktion `Receive()` empfängt, ist in dem Rückgabewert dieser SIMPL-Funktion enthalten.
Der Prozess erhält nicht direkt den *int*-Wert des Proxys als Rückgabewert von der SIMPL-Funktion `Receive()`, sondern er erhält erst einmal einen negativen Rückgabewert kleiner gleich -2. Wie bei der SIMPL-Funktion `Receive()` beschrieben, deutet der negative Rückgabewert auf den Empfang eines Proxys hin. Um den *int*-Wert des Proxys holen zu können, muss der Prozess die SIMPL-Funktion `returnProxy()` aufrufen. Dabei übergibt er den negativen Rückgabewert der SIMPL-Funktion `Receive()` als Übergabeparameter an `returnProxy()`.

Wenn die SIMPL-Funktion `returnProxy()` erfolgreich ausgeführt wurde, liefert sie den *int*-Wert des Proxys zurück [17].

C. Dokumentation der verwendeten zeroMQ-Funktionen

Die externe Bibliothek *zeroMQ* ermöglicht, dass Threads innerhalb eines Prozesses, Prozesse innerhalb eines Rechners und über das Netzwerk kommunizieren können. Ein zeroMQ Socket kann viele eingehende und ausgehende Verbindungen haben.
Die Kommunikation zwischen Threads und zwischen Prozessen findet über die zeroMQ Sockets statt. Über diese Sockets können Threads und Prozesse ihre Nachrichten senden und empfangen. Eine Nachricht wird nicht umgehend an den Socket ihres Empfängers gesendet, sondern sie wird beim Sender in einer *message queue* abgelegt. Diese Nachricht wird dann asynchron an den Empfänger gesendet [18].

Listing 68: zeroMQ-Funktion `zmq_ctx_new()`

```
void* zmq_ctx_new ();
```

Die zeroMQ-Funktion `zmq_ctx_new()` erzeugt einen neuen zeroMQ *context*.
Ein zeroMQ *context* ist eine Art Container, in dem alle Sockets enthalten sind, die ein Prozess erzeugt hat [18]. Daher erzeugt ein Prozess immer zu Beginn einen zeroMQ *context*, bevor er die Sockets erzeugt.
Der Rückgabewert der zeroMQ-Funktion `zmq_ctx_new()` ist ein *void*-Zeiger. Wenn diese Funktion erfolgreich ausgeführt wurde, liefert sie einen *opaque handle* auf den neu erzeugten zeroMQ *context* zurück. Ein *opaque handle* ist im Allgemeinen eine Referenz auf eine Instanz, die nicht direkt de-referenziert werden kann.
Im Fehlerfall liefert die zeroMQ-Funktion `zmq_ctx_new()` den Wert NULL zurück [19].

Listing 69: zeroMQ-Funktion `zmq_socket()`

```
void* zmq_socket (void* context, int type);
```

Mit der zeroMQ-Funktion `zmq_socket()` erzeugt ein Prozess einen Socket innerhalb des zeroMQ *contexts*.
Dem formalen Parameter `context` wird der Rückgabewert der zeroMQ-Funktion `zmq_ctx_new()` übergeben. Über den formalen Parameter `type` kann bestimmt werden, wie die Kommunikation über den Socket erfolgen soll. Die Socket Typen, die die externe Bibliothek *zeroMQ* anbietet, sind nach dem allgemeinen *messaging pattern* kategorisiert. Im Anschluss der Beschreibung dieser zeroMQ-Funktion werden die *messaging patterns* vorgestellt, die für die Umsetzung der Funktionen der neuen IPC-Bibliothek verwendet wurden.
Der Rückgabewert der zeroMQ-Funktion `zmq_socket()` ist ein *void*-Zeiger. Wenn diese Funktion erfolgreich ausgeführt wurde, liefert sie einen *opaque handle* auf den neu erzeugten Socket zurück. Im Fehlerfall liefert sie den Wert NULL zurück [19].

Request-reply Pattern

Das request-reply Pattern wird verwendet, wenn ein Client eine Nachricht synchron an einen oder mehrere Server senden möchte. Der Client erhält auf jede seiner gesendeten Nachrichten eine Antwort von den jeweiligen Servern.
Der Client übergibt beim Erzeugen eines neuen Sockets dem formalen Parameter `type` den Socket Typ `ZMQ_REQ`. Damit kann der Client über seinen neu erzeugten Socket Nachrichten an den Server senden und Antworten von ihm empfangen.
Der Server übergibt beim Erzeugen eines neuen Sockets dem formalen Parameter `type` den Socket Typ `ZMQ_REP`. Damit kann der Server über seinen neu erzeugten Socket Nachrichten vom Client empfangen und entsprechend Antworten an den Client senden [19].

Pipeline Pattern

Das Pipeline Pattern wird verwendet, wenn Daten unidirektional in einer Pipeline an *nodes* übermittelt werden sollen. Bezüglich der Umsetzung der Client-Server Kommunikation sind mit *nodes* die Clients gemeint.
Wenn ein Client einen neuen Socket erzeugt, übergibt er dem formalen Parameter `type` den Socket Typ `ZMQ_PULL`. Damit kann der Client über seinen neu erzeugten Socket Daten in einer Nachricht von einem oder mehreren Servern empfangen. Der Client kann nur über seinen Socket Nachrichten empfangen, er kann dem Server keine Nachrichten senden. Der Server übergibt beim Erzeugen eines neuen Sockets dem formalen Parameter `type` den Socket Typ `ZMQ_PUSH`. Damit kann der Server über seinen neu erzeugten Socket Daten in einer Nachricht an Clients senden. Über seinen Socket kann der Server nur Nachrichten versenden, er kann keine Nachrichten von den Clients empfangen [19].

Listing 70: zeroMQ-Funktion `zmq_bind()`

```
int zmq_bind (void* socket, const char* endpoint);
```

Mit der zeroMQ-Funktion `zmq_bind()` bindet ein Prozess, der als Server fungiert, seinen neu erzeugten Socket `socket` an einen Endpunkt `endpoint`. Er akzeptiert eingehende Verbindungen, die zu diesem Endpunkt hergestellt werden.
Der Endpunkt ist eine bekannte Netzwerkadresse oder ein Dateipfad, wenn es sich um eine lokale Kommunikation handelt. Für die lokale Kommunikation wird dem formalen Parameter `endpoint` ein String übergeben, der aus dem Präfix `ipc://` und dem Dateipfad zusammengesetzt wird. Ein Beispiel, wie dieser String aussehen könnte, wird im Folgenden dargestellt: `"ipc:///tmp/feeds/0"`.
Der Rückgabewert der zeroMQ-Funktion `zmq_bind()` ist vom Datentyp *int*. Im Erfolgsfall liefert diese zeroMQ-Funktion den Wert null zurück, ansonsten den Wert -1 [19].

Listing 71: zeroMQ-Funktion `zmq_connect()`

```
int zmq_connect (void* socket, const char* endpoint);
```

Mit der zeroMQ-Funktion `zmq_connect()` verbindet ein Prozess, der als Client

fungiert, seinen neu erzeugten Socket `socket` an einen Endpunkt `endpoint`.
Bezüglich der lokalen Kommunikation muss der Client den gleichen Dateipfad für den formalen Parameter `endpoint` angeben, den der Server auch angegeben hat.
Der Rückgabewert der zeroMQ-Funktion `zmq_connect()` ist vom Datentyp *int*. Im Erfolgsfall liefert diese zeroMQ-Funktion den Wert null zurück, ansonsten den Wert -1 [19].

Listing 72: zeroMQ-Funktion `zmq_disconnect()`

```
int zmq_disconnect (void* socket, const char* endpoint);
```

Mit der zeroMQ-Funktion `zmq_disconnect()` trennt ein Client die Verbindung, die zwischen seinem Socket `socket` und dem Endpunkt `endpoint` besteht.
Der Rückgabewert der zeroMQ-Funktion `zmq_disconnect()` ist vom Datentyp *int*. Diese zeroMQ-Funktion liefert im Erfolgsfall den Wert null zurück, ansonsten den Wert -1 [19].

Listing 73: zeroMQ-Funktion `zmq_send()`

```
int zmq_send (void* socket, void* buf, size_t len, int flags);
```

Mit der zeroMQ-Funktion `zmq_send()` sendet ein Prozess eine Nachricht an einen anderen Prozess, der diese Nachricht erhalten soll.
Dem formalen Parameter `socket` wird der *opaque handle* übergeben, den der sendende Prozess von der zeroMQ-Funktion `zmq_socket()` erhalten hat. Die Nachricht, die gesendet werden soll, wird an den formalen Parameter `buf` mit der Größe `len` übergeben. Dem formalen Parameter `flags` können bestimmte Flags übergeben werden, die die Eigenschaften des Sendens über den Socket `socket` bestimmen.
Die Nachricht, die an den Empfänger gesendet wird, wird in den Socket des sendenden Prozesses geschrieben, beziehungsweise wird sie in der *message queue* abgelegt. Diese Nachricht wird dann asynchron an den Socket des empfangenen Prozesses gesendet.
Der Rückgabewert der zeroMQ-Funktion `zmq_send()` ist vom Datentyp *int*. Im Erfolgsfall liefert diese zeroMQ-Funktion die Größe der gesendeten Nachricht in Bytes zurück, ansonsten den Wert -1 [19].

Listing 74: zeroMQ-Funktion `zmq_recv()`

```
int zmq_recv (void* socket, void* buf, size_t len, int flags);
```

Mit der zeroMQ-Funktion `zmq_recv()` empfängt ein Prozess über seinen Socket `socket` eine Nachricht, die ein anderer Prozess ihm gesendet hat.
Dem formalen Parameter `socket` wird der *opaque handle* übergeben, den der empfangene Prozess von der zeroMQ-Funktion `zmq_socket()` erhalten hat. Die empfangene Nachricht wird im Puffer `buf` mit der Größe `len` gespeichert. Über den formalen Parameter `flags` können die Eigenschaften des Empfangens über den Socket `socket` bestimmt werden.

Wenn keine Nachrichten im Socket des empfangenen Prozesses vorhanden sind, blockiert die zeroMQ-Funktion `zmq_recv()` die Ausführung des empfangenen Prozesses. Sie wird solange blockiert, bis wieder Nachrichten in seinem Socket vorhanden sind.
Der Rückgabewert der zeroMQ-Funktion `zmq_recv()` ist vom Datentyp *int*. Wenn diese zeroMQ-Funktion erfolgreich ausgeführt wurde, liefert sie als Rückgabewert die Größe der empfangenen Nachricht in Bytes zurück. Im Fehlerfall liefert sie den Wert -1 zurück [19].

Literaturverzeichnis

[1] Korf, F. (2011): Inter Process Communication (IPC). https://pub.informatik.haw-hamburg.de/home/pub/prof/korf/TI4_SY/VL_BTI4-SY_SS_2011_Teil_3_version_0.60.pdf (abgerufen am 16. Januar 2013).

[2] QNX Software Systems: Specific entries. http://www.qnx.com/developers/docs/6.4.1/neutrino/lib_ref/ (abgerufen am 16. Januar 2013).

[3] QNX Software Systems: Initialize the dispatch interface. http://www.qnx.com/developers/docs/6.4.1/neutrino/resmgr/skeleton.html#InitDispatchInterface (abgerufen am 20. Januar 2013).

[4] QNX Software Systems: Using message passing. http://www.qnx.com/developers/docs/6.4.1/neutrino/getting_started/s1_msg.html#Using (abgerufen am 17. Januar 2013).

[5] QNX Software Systems: Synchronous message passing. http://www.qnx.com/developers/docs/6.4.1/neutrino/sys_arch/ipc.html#Sync_messaging (abgerufen am 06. Februar 2013).

[6] QNX Software Systems: Pulses. http://www.qnx.com/developers/docs/6.4.1/neutrino/getting_started/s1_msg.html#Pulses (abgerufen am 22. Januar 2013).

[7] QNX Software Systems: Events. http://www.qnx.com/developers/docs/6.4.1/neutrino/sys_arch/ipc.html#Events (abgerufen am 21. Januar 2013).

[8] Wolf, J.: Linux-UNIX-Programmierung. Das umfassende Handbuch. 3., aktualisierte und erweiterte Auflage. Bonn: Galileo Press, 2009.

[9] QNX Software Systems: Shared memory. http://www.qnx.com/developers/docs/6.4.1/neutrino/sys_arch/ipc.html#Shared_memory (abgerufen am 23. Januar 2013).

[10] Kerrisk, M. (2010): The Linux man-pages project. http://man7.org/linux/man-pages/dir_all_alphabetic.html (abgerufen am 23. Januar 2013).

[11] Freeman, E./Freeman, E./Sierra, K./Bates, B.: Head First Design Patterns. Sebastopol, Kalifornien: O'Reilly Media, 2004.

[12] Spiess, M.: Software-Entwurfsmuster. Prinzip von Entwurfsmustern und einige elementare Beispiele. http://www.mathematik.uni-ulm.de/stochastik/lehre/ws03_04/seminarws03_04/spiess.pdf (abgerufen am 11. Februar 2013).

[13] Bohn, S./Korb, W./Burgert, O.: A process and criteria for the evaluation of software frameworks in the domain of computer assisted surgery. http://download.springer.com/static/pdf/293/art%253A10.1007%252Fs11517-008-0336-9.pdf?auth66=1352722288_d568f860234e8ad74ea8ccb3c4c7bf93&ext=.pdf (abgerufen am 12. November 2012).

[14] Glatz, E.: Betriebssysteme. Grundlagen, Konzepte, Systemprogrammierung. 2., aktualisierte und überarbeitete Auflage. Heidelberg: dpunkt.verlag, 2010.

[15] Kerrisk, M.: THE LINUX PROGRAMMING INTERFACE. A Linux and Unix System Programming Handbook. http://man7.org/tlpi/download/TLPI-52-POSIX_Message_Queues.pdf (abgerufen am 26. September 2012).

[16] Vogt, C.: Nebenläufige Programmierung. Ein Arbeitsbuch mit UNIX/Linux und Java. München: Carl Hanser Verlag, 2012.

[17] Collins, J./Findlay, R.: Programming the SIMPL Way. iCanProgram Inc., 2010.

[18] Hintjens, P. (2013): ØMQ - The Guide. http://zguide.zeromq.org/page:all (abgerufen am 01. Februar 2013).

[19] iMatix Corporation: ØMQ API. ØMQ/3.2.2 API Reference. http://api.zeromq.org/3-2:_start (abgerufen am 27. Februar 2013).

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, den 11. März 2013 _____

Martin Hua