Kosten der Abschirmung von Code und Daten

Alexander Züpke[1], Kai Beckmann[2], Andreas Zoor[2] und Reinhold Kröger[2]

[1] Forschungsgruppe Neue Betriebssystemkonzepte
[2] Labor für Verteilte Systeme
Fachbereich Design Informatik Medien
Hochschule RheinMain, 65195 Wiesbaden
vorname.nachname@hs-rm.de

Zusammenfassung. Aktuellen Systemplattformen für *Internet der Dinge*-Geräte mangelt es oft an Isolationskonzepten. Verglichen mit traditionell isolierten Eingebetteten Systemen vergrößert die Anbindung an das Internet die Angriffsfläche, so dass durch Fehler die Funktionalität des gesamten Gerätes beeinträchtigt werden kann.
Diese Arbeit vergleicht verschiedene Isolationskonzepte zur Auftrennung von Betriebssystemkomponenten für eine Publish/Subscribe-Middleware nach dem OMG-Standard *Data Distribution Service*, auf einem partitionierenden Mikrokern-System. Die Konzepte werden auf einem STM32F4-Mikrocontroller evaluiert, welcher geeignete Speicherschutzmechanismen bietet. Die Ergebnisse zeigen moderate Kosten durch erhöhten Speicherbedarf und zusätzliche Kontextwechsel.

1 Motivation

Heutzutage bieten preisgünstige Mikrocontroller genug Schnittstellen und Leistung, um vormals isolierten Eingebetteten Systemen eine Anbindung an das Internet zu bieten. Diese *Internet der Dinge* (engl. *Internet of Things*, IoT) genannten Systeme beinhalten dabei Sensoren und/oder Aktoren in neuartigen vernetzten Anwendungen. Allerdings fehlen der ersten Generation dieser Geräte oft Isolationskonzepte zwischen den auf dem System eingesetzten Softwarekomponenten, so dass ein potentieller Fehler im Anwendungscode, im Betriebssystem oder in den Protokoll-Stacks das Gesamtsystem kompromittieren könnte. Für sicherheitskritische Anwendungen sind diese Systeme daher nur bedingt geeignet.

In letzter Zeit sind allerdings preisgünstige und sparsame 32-bit Mikrocontroller erhältlich, die Speicherschutzmechanismen (engl. *Memory Protection Unit*, MPU) bieten, womit auch bei IoT-Anwendungen eine Isolierung möglich wird. Leider bieten aktuelle Betriebssystemplattformen für preisgünstige IoT-Systeme, wie *FreeRTOS*, *Contiki* oder *RiotOS*, oft keine Unterstützung für solche Speicherschutzmechanismen. Auf der anderen Seite des Spektrums erwarten Betriebssysteme wie *VxWorks*, *Integrity* oder *PikeOS*, welche explizit für sicherheitskritische Anwendungen entworfen wurden, Hardware-Unterstützung für eine virtuelle Speicherverwaltung (engl. *Memory Management Unit*, MMU). Letztere ist allerdings nur auf deutlich höherpreisigen Prozessoren verfügbar.

Zusätzliche Sicherheit gibt es nicht umsonst: der Einsatz von Isolationskonzepten bedingt oft erhöhten Speicherverbrauch durch die Mehrfachhaltung benötigter Daten oder durch die Granularität der Schutzmechanismen. Ebenso kann die System-Performance negativ beeinflusst werden, zum Beispiel durch häufiges Kopieren von Daten zwischen isolierten Komponenten oder durch die Umschaltung der zugreifbaren Speicherbereiche bei Kontextwechseln.

Diese Arbeit analysiert diese Kosten anhand des Fallbeispiels einer Portierung einer datenzentrierten Middleware auf eine Mikrokern-Plattform für sicherheitskritische Anwendungen. Das Ziel dabei ist es, dass Anwendungen mit Hilfe der Middleware über das Netzwerk kommunizieren können, Fehler sich allerdings nicht ausbreiten können.

Die an der HSRM entwickelte, *sDDS* [1] genannte Middleware ist eine Implementierung des *Data Distribution Service* (DDS) Standards der *Object Management Group* (OMG) [2] für Sensornetzwerke und besonders für kleine Systeme mit beschränkten Systemressourcen geeignet. Als Betriebssystem wird *AUTOBEST* [3] benutzt, welches für automotive Anwendungsfälle mit hohen Sicherheitsanforderungen entwickelt wurde. AUTOBEST bietet die Möglichkeit, unterschiedliche Software-Komponenten in sogenannten Partitionen zu isolieren, um so Auswirkungen möglicher Fehler auf die betroffene Partition zu beschränken. Für die Netzwerkanbindung von sDDS wird der *Lightweight TCP/IP-Stack* (lwIP) [4] unter AUTOBEST verwendet. Als Hardware-Plattform wurde ein STM32F4-Mikrocontroller von Texas Instruments gewählt, eine typische Plattform für industrielle Echtzeit-Anwendungen. Neben einem Cortex-M4-basierten ARM-Kern mit 168 MHz Takt bietet dieser Prozessor 112 KiB SRAM, 1 MiB Flash und 100 Mbit/s Ethernet.

Im Weiteren beschreibt Kapitel 2 die verwendeten Software-Komponenten im Detail. Kapitel 3 erläutert verschiedene Trennungskonzepte, welche in Kapitel 4 evaluiert werden. In Kapitel 5 werden die Ergebnisse zusammengefasst, diskutiert und mit anderen Arbeiten verglichen.

2 Software-Architektur

Die Software-Architektur basiert auf den zwei unanbhängig voneinander am Fachbereich entwickelten Komponenten: AUTOBEST als Betriebssystem und sDDS als Middleware. Des Weiteren werden lwIP als TCP/IP-Stack, ein Ethernet-Treiber und passende Anwendungen, die per sDDS mit anderen Knoten kommunizieren, verwendet. Alle Komponenten zeichnen sich durch einen hohen Grad an Anpassbarkeit hinsichtlich ihres Ressourcenverbrauchs aus.

AUTOBEST AUTOBEST [3] wurde in Kooperation mit der Firma Easycore GmbH als Software-Plattform für den gleichzeitigen, rückwirkungsfreien Betrieb unterschiedlich sicherheitskritischer Softwarekomponenten in automobilen Steuergeräten mit beschränkten Hardware-Ressourcen entworfen und implementiert.

AUTOBEST basiert auf einem Mikrokern, welcher unterschiedliche Applikationen in sogenannten Partitionen isoliert voneinander ausführen kann. Ei-

ne Partition stellt einen Container sowohl im zeitlichen (Scheduling) als auch im räumlichen Sinne (getrennte Adressräume) dar, der eine zugehörige Menge von Ausführungskontexten (Tasks) und ihre notwendigen Betriebsmittel vom Rest des Systems isoliert. Alle Kommunikations- und Synchronisationmechanismen über Partitionsgrenzen hinweg müssen statisch konfiguriert werden. Dies erlaubt eine feingranulare Zugriffskontrolle möglicher Interferenzen zur Kompilationszeit. Dieses Partitionierungskonzept wird auch in anderen Anwendungsbereichen wie der Avionik erfolgreich eingesetzt. AUTOBEST ist dabei nicht nur auf automotive Anwendungsfälle beschränkt: Ein abstraktes Programmiermodell ermöglicht es, dass domänenspezifische Bibliotheken in den Partitionen die für *AUTOSAR* (Automotive) [5] und *ARINC 653* (Avionik) [6] notwendigen Betriebssystemschnittstellen ihren jeweiligen Anwendungen bereitstellen können.

Der Datenaustausch über Partitionsgrenzen hinweg wird durch gemeinsame Speicherbereiche (engl. *Shared-Memory-Segments*, SHM) zwischen den Partitionen realisiert. Der Kern macht hierbei keine Vorgaben zur Struktur, dies ist den beteiligten Partitionen überlassen. Der Kern bietet lediglich an, Variablen im Speicher mit Warteschlangen zu verknüpfen. Dies ermöglicht die Konstruktion blockierender Synchronisationsmuster nach dem Erzeuger-Verbraucher-Prinzip. Weiterhin können mit Hilfe von asynchronen Events andere Partitionen über Änderungen im Speicher benachrichtigt werden. Ebenso bietet der Kern einen synchronen *Remote Procedure Call* (RPC) an. Dies erlaubt die Ausführung von Funktionen in einer anderen Partitionen.

sDDS Der *Data Distribution Service*-Standard (DDS) der OMG [7] spezifiziert eine datenzentrierte Middleware-Schnittstelle, die dem Publish-Subscribe-Paradigma entspricht und plattform- und herstellerunabhängig spezifiziert ist. Der Standard sieht eine Reihe von Dienstgütemerkmalen (engl. *Quality-of-Service*, QoS) vor, mit denen Echtzeiteigenschaften, Redundanz, Persistenz, Zuverlässigkeit und Ressourcenbeschränkungen konfiguriert werden können. Die Architektur von DDS basiert auf dem Konzept eines gedachten globalen Datenraums, in dem gleichrangige Teilnehmer über ein Netzwerk Daten bereitstellen oder abonnieren (Peer-to-Peer). Die auszutauschenden Daten werden strukturiert als anwendungsspezifische Datentypen spezifiziert. Ein sogenanntes *Topic* verbindet einen Namen mit einem solchen Datentyp und festgelegten QoS-Eigenschaften. Als Schnittstelle der Anwendung zum Datenraum dienen *DataWriter*- und *DataReader*-Klassen.

Seinen Ursprung hat DDS im militärischen Bereich; inzwischen wird die Entwicklung und Verbreitung des Standards vor allem aus industriellen Anwendungsfällen getrieben. Insbesondere für das Internet der Dinge und im Bereich Industrie 4.0 verspricht DDS in Zukunft relevant zu werden. Obwohl der DDS-Standard in Hinblick auf verteilte eingebettete Anwendungen entworfen wurde, sind vollständige Implementierungen auf den ressourcenbeschränkten Plattformen typischer drahtloser Sensornetze nicht lauffähig.

Für den in dieser Arbeit beschriebenen Ansatz wird daher sDDS (*sensor-network DDS*) verwendet [1]. Anwendungen für Sensornetze oder das IoT sind,

bezogen auf die ausführenden Knoten, funktional häufig statisch und benötigen nur eine Untermenge der DDS-Middleware-Funktionalität. Bei sDDS wird daher ein modellgetriebener Software-Entwicklungsprozess verwendet. Dabei werden die Anforderungen einer Anwendung auf einer Knoteninstanz an die spezifische Funktionalität einer DDS-Middleware in einem Systemmodell erfasst und individuell angepassten Middleware-Code für jeden Zielknoten generiert. Dies ermöglicht es, DDS auf sehr heterogenen Plattformen einzusetzen, vom 8-Bit Microkontroller bis hin zu einer PC-Umgebung, und vereinfacht damit sowohl die horizontale als auch die vertikale Integration.

Bisher wird in sDDS eine konfigurierbare Untermenge des DDS-Standards umgesetzt. Neben dem Austausch von Daten mit Unterstützung von Callbacks und Polling werden einige QoS-Eigenschaften unterstützt. Die Kommunikationsbeziehungen der Anwendungen können statisch, dynamisch und auch als eine Kombination aus beidem spezifiziert und generiert werden. Im dynamischen Fall wird ein *Discovery*-Mechanismus auf Basis von fest eingebauten Topics realisiert.

sDDS ist in *C* mit Hinblick auf Plattformunabhängigkeit implementiert. Die bereitgestellte API ist DDS-standardkonform. Notwendige Betriebssystem- und Plattform-abhängige Funktionalität ist über Schnittstellen abstrahiert. sDDS benötigt Heap-Speicher zum Initialisierungszeitpunkt, die Möglichkeit, Aufgaben in Form von Callbacks zyklisch oder nach dynamisch anzupassenden Zeitspannen auszuführen und wechselseitigen Ausschluss. Als unterlagertes Netzwerkprotokoll setzt DDS minimal einen unzuverlässigen Datagram-Dienst mit Routing- und Broadcast- bzw. Multicast-Funktionen, wie beispielsweise bei UDP/IP, voraus. Der Discovery-Mechanismus baut auf Multicast-Gruppen auf. Durch das Einsatzgebiet IoT liegt der Fokus auf IPv6-basierten Transporttechnologien.

lwIP Da AUTOBEST keinen eigenen TCP/IP-Stack mitbringt, wird in dieser Arbeit auf das Open-Source-Projekt *Lightweight TCP/IP-Stack* (lwIP) [4] zurückgegriffen. LwIP ist ein stark konfigurierbarer TCP/IP-Stack mit BSD-kompatiblen Socket-Schnittstellen, für sDDS wird hingegen nur die UDPv6-Funktionalität über Ethernet sowie die IPv6-Adresskonfiguration benötigt.

LwIP bietet zwei Schnittstellen an. Die Schnittstelle zur Netzwerkseite hin sendet oder empfängt Ethernet-Frames. Die Schnittstelle zur Applikation (hier sDDS) erlaubt den Datenaustausch über ein Socket-ähnliches API. Beide Seiten benutzen entkoppelte Pufferstufen mit einem Erzeuger-Verbraucher-Muster. Die Protokollumsetzung findet intern transparent für die Anwendung statt.

LwIP bringt eine eigene Speicherverwaltung für Datenpakete und Ethernet-Frames, sogenannte *Pbufs*, mit. Ein Pbuf kann beliebige Nutzdaten enthalten. LwIP reserviert genügend Platz am Anfang und am Ende eines Pbufs, um bei ausgehenden Frames die IP- und Ethernet-Header hinzufügen zu können. Ebenso werden beim Empfang die Header-Informationen entfernt. Frames mit mehr als 536 B Nutzdaten können in mehreren Pbufs verkettet werden.

Anwendungen Im präsentierten Szenario kommuniziert eine Anwendung auf dem Embedded-Target unter AUTOBEST mit einer zweiten auf einem Linux-

System. Es gibt dafür zwei Typen von Anwendungen und zugehörigen Topics. Die Middleware, die anwendungsabhängigen Initialisierungen und ein Anwendungsstub wurden dazu aus dem Systemmodell generiert.

Die erste Anwendung publiziert zyklisch Daten für das Topic *Ipc* mit dem synchronen DDS-Methodenaufruf `DDS_IpcDataWriter_write()`, während die zweite Anwendung Daten von diesem Topic empfängt. Um die Latenz gering zu halten, registriert die Anwendung bei der DDS-Middleware einen Callback, *Listener* genannt, der aufgerufen wird, wenn neue Daten verfügbar sind. Die Daten selber werden mit der Topic-spezifischen Methode `DDS_IpcDataReader_take_next_sample()` ausgelesen. Diese entnimmt die Daten aus der Empfangswarteschlange des Topics und kann nicht blockieren. Die Größe der Topics beträgt jeweils 2 Byte, um den Overhead für Kopieroperationen während der Messungen gering zu halten.

3 Trennungskonzepte

Die im vorherigen Kapitel genannten Softwarekomponenten bilden in einem monolithischen System ein vertikales Schichtenmodell, die nun auf horizontale Partitionen aufgeteilt werden müssen. Hierzu bieten sich die verschiedenen Über-gänge zwischen den Komponenten an, um einen Schnitt anzusetzen. Das Ziel dabei ist es, mehrere voneinander isolierte Anwendungen über sDDS und das Internet kommunizieren zu lassen, ohne dass der Kommunikations-Overhead (Speicherverbrauch und Performance) zwischen den Partitionen zu groß wird. Ebenso ist eine mehrfache Auftrennung denkbar. Im Folgenden werden die verschiedenen möglichen Szenarien für eine Trennung diskutiert.

Isolation der Netzwerkkomponenten Die einfachste Variante ist es, die Anwendung, sDDS, lwIP und den Ethernet-Treiber in einer Partition zu gruppieren. Dies erlaubt es, die Teile mit Netzwerk-Anbindung vom Rest des Systems zu isolieren.

Dieses Szenario wird als Baseline für die in dieser Arbeit durchgeführten Messungen verwendet, da hier alle Komponentenübergänge mit reinen Funktionsaufrufen realisiert werden. Alle Daten können ohne Kopieraufwand über Zeiger übergeben werden. Diese Variante ist zudem vergleichbar mit Implementierungen auf anderen Embedded- und IoT-Betriebssystemen.

Trennung zwischen Ethernet-Treiber und lwIP Im Schichtenmodell von unten beginnend wird als erstes der Schnitt zwischen dem Ethernet-Treiber und lwIP betrachtet. Diese Schnittstelle sieht nach dem Aufsetzen von Ethernet-Adressen hauptsächlich den Austausch von Ethernet-Frames vor. Sowohl ein- als auch ausgehende Frames können in Bursts auftreten. Daher bieten sich hier zur robusten Entkopplung der Komponenten zwei Ringpuffer von Ethernet-Frames in einem Shared-Memory-Segment an. Dies vermeidet unnötige Kontextwechsel,

und beide Komponenten können sich asynchron über neue Frames benachrichtigen. Ebenso erlaubt die Speicherverwaltung von lwIP eine statische Konfiguration der Speicherbereiche für Pbufs. Zusätzlich können DMA-Transfers direkt auf den Frames im Shared-Memory-Segment durchgeführt werden. Wenn die Netzwerk-Hardware DMA von fragmentierten Frames unterstützt, können im Ringpuffer kleinere Speicherbereiche verwendet werden.

Zusammengefasst bietet diese Schnittstelle gute Möglichkeiten zur Entkopplung und einiges Optimierungspotential, wenn direktes DMA möglich ist. Auf der anderen Seite kann der Speicherbedarf größer sein.

Trennung zwischen lwIP und sDDS Eine weitere Trennung ist zwischen lwIP und sDDS möglich. sDDS benutzt für die Netzwerkanbindung eine interne Schnittstelle von lwIP, welche dem BSD-Socket-Interface nachempfunden ist. Allerdings werden die in die Netzwerkdarstellung überführten Daten nicht kopiert sondern direkt in Pbufs gehalten.

Die Netzwerkanbindung von sDDS öffnet eine Multicast- und eine Unicast-Server-Verbindung für UDP, über die der Datenaustausch und das Discovery ausgeführt werden. Die Aufgaben von lwIP beschränken sich in diesem Fall auf das Durchreichen der Daten sowie die Adressauflösung. Ebenso wie bei der Trennung zwischen Ethernet-Treiber und lwIP können die Datagramme hier in Ringpuffern zwischen den Partitionen gehalten werden; die Speicherverwaltung von lwIP würde dies erlauben, solange nur eine Anwendung den IP-Stack benutzt.

Ansonsten bietet sich an, dass lwIP asynchron über empfangene Datagramme benachrichtigt, während sDDS synchron sendet. Diese Asymmetrie ist notwendig, da sDDS dem lwIP-Stack vertrauen kann. Im Umkehrschluss gilt dies für den lwIP-Stack allerdings nicht, da sonst ein Fehler im sDDS die Funktionalität des IP-Stacks beeinflussen könnte. Im Vergleich zur Trennung zwischen Ethernet-Treiber und lwIP ist der Aufwand für eine Trennung an dieser Stelle höher.

Trennung zwischen sDDS und Anwendung Als dritte Option wird die Trennung zwischen Anwendungen und sDDS mit IP-Stack und Ethernet-Treiber betrachtet. Wie zuvor beschrieben, verwendet die erste Anwendung zum Publizieren einen blockierenden Funktionsaufruf. Die zweite registriert einen Callback, der darüber informiert, wenn neue Daten bereitstehen, und entnimmt diese anschließend synchron und nicht blockierend der Empfangswarteschlange des Topics. Je nach Puffertiefe der Warteschlange bietet es sich an, eingehende Daten eines Topics in einem Shared-Memory-Segment zu halten, so dass die Anwendung die Daten ohne Kontextwechsel lesen kann. Allerdings benötigt die Anwendung Schreibzugriff auf die Warteschlange, um ein Sample als gelesen markieren zu können. Da das Publizieren von Daten blockieren kann, muss ein synchroner Kommunikationsmechanismus verwendet werden.

Eine Trennung zwischen Anwendung und sDDS erfordert demnach einen Nachbau der Funktionsaufrufe des DDS-APIs mittels synchroner Kommunikation und einen asynchronen Rückkanal zur Applikation zur Benachrichtigung über

aktualisierte Daten. Die Menge der übertragenen Daten hängt von der Größe der Topics ab und ist in der Regel klein.

Gewählter Lösungsansatz Der Vergleich der Trennungskonzepte zeigt, dass jedes Schreiben von Daten aus der Anwendung zu mindestens einem Ethernet-Frame führt, solange sDDS nicht mehrere Topics in einem Frame gruppiert. Eingehende Frames hingegen enthalten nicht unbedingt Daten, die bei der Anwendung ankommen, da sie auch für interne Verwaltungsaufgaben bestimmt sein können. Ebenso werden die ausgetauschten Datenmengen (bzw. die vorzuhaltenden Speicherbereiche) von der Anwendung zum Netzwerktreiber durch den Overhead größer. Allerdings ist die Anbindung der Komponenten zu unteren Schichten deutlich entkoppelter, so dass hier häufiger asynchrone Kommunikationsmechanismen verwendet werden können.

Im Hinblick auf den meist knappen RAM-Speicher trennt der gewählte Lösungsansatz daher zwischen sDDS und Anwendungen. Dies bedingt als Preis die Implementierung eines komplexeren APIs. Die gewählte Implementierung kombiniert Ethernet-Treiber, lwIP und sDDS in einer Partition. Der Ethernet-Treiber nutzt dabei die Speicherverwaltung von lwIP für die Ethernet-Frames. Der Ethernet-Treiber besteht aus einem ISR-Task, der empfangene Pakete über eine Message-Queue an die UDP/IP-Task von lwIP weiterreicht. sDDS besteht aus einer Task für eingehende Daten, welche die Anwendungspartitionen über asynchrone Events signalisiert, sowie einer weiteren Task für jede Anwendungs-Partition, welche synchrone RPC-Calls der Anwendungen abarbeitet.

Die Anwendungs-Partitionen beinhaltet je zwei Tasks: Ein Callback-Manager-Task wartet auf Events vom sDDS und führt registrierte Callbacks in seinem Kontext aus. Generierter Anwendungscode läuft im Client-Task. Für den Datenaustausch zwischen den Partitionen wird pro Anwendung ein Shared-Memory-Segment verwendet, welches Topics, Benachrichtigungen und RPC-Argumente enthält. Die Struktur des gemeinsamen Speichers und notwendige Stubs auf Seiten der Applikation und sDDS werden von einem Code-Generator erzeugt. Die Trennung ist für sDDS und die Anwendungen vollständig in generiertem Code gekapselt. Über die Code-Generierung sind auch die Rollen der Anwendungen bekannt, so dass die entsprechenden DataReader und DataWriter erstellt werden können. Diese werden für die Kommunikation mit anderen Knoten genutzt.

4 Evaluation

Zur Evaluation der Kosten der gewählten Trennung werden der Speicherverbrauch sowie die Zeiten für das Senden und Empfangen eines Topics mit der Variante ohne Trennung verglichen. Für die Messung des Performce-Overheads wurden dazu in den Sende- und Empfangspfaden alle Komponentenübergänge mit Trace-Punkten versehen, die einen Pin des Mikrocontrollers „toggeln“. Die Pegeländerungen an den Pins werden dann mit einem Logikanalysator aufgezeichnet. Der dadurch entstehende zusätzliche Aufwand durch die Ausführungszeiten von ein- und derselben Codesequenz beträgt weitgehend deterministisch

0,920 µs, da der Mikrokontroller keine Caches hat. Die einzigen Störquellen sind Interrupts vom Timer und vom Ethernet.

Die Messpunkte kennzeichnen die einzelnen Phasen eines Sendevorgangs von der Anwendung über sDDS und lwIP bis in den Ethernet-Treiber samt der notwendigen Datenaufbereitung. Der Empfangsvorgang verläuft analog vom Eintreffen eines Ethernet-Frames bis zur Sichtbarkeit der Daten in der Applikation. Um den Overhead nachzuvollziehen, wurden bei der Lösung mit Trennung weitere Messpunkte an den Event- und RPC-Aufrufen hinzugefügt.

Die Messwerte in µs für den Empfangsvorgang sind in Tabelle 1 aufgeführt. In einem Messexperiment mit wiederholten Durchläufen und insgesamt 16 500 Messpunkten wurden die Messwerte mit Minimal- und Maximalwerten (MIN, MAX), Durchschnitt (AVG) und Standardabweichung (STD) zusammengefasst.

Tabelle 1. Messwerte Empfangsvorgang in µs

Phase	ohne Trennung				mit Trennung			
	MIN	AVG	MAX	STD	MIN	AVG	MAX	STD
IRQ im Kern	6,040	6,762	12,120	0,074	6,010	6,707	12,140	0,097
Ethernet-ISR	8,240	8,276	18,380	0,428	8,240	8,291	18,380	0,573
lwIP-Stack	29,310	36,457	41,890	0,178	31,220	36,501	43,670	0,195
sDDS UDP-Modul	17,490	17,505	23,380	0,125	17,410	17,426	20,320	0,073
sDDS `DataSink_processFrame`	8,080	9,001	25,710	1,194	7,750	8,680	25,580	1,204
Aufruf *Data Available* Callback	5,630	5,639	10,940	0,081	5,460	5,471	8,490	0,054
sDDS sendet Event an Callback-Manager					1,140	1,147	3,070	0,023
Aktivierung Callback-Mngr					20,380	20,415	27,990	0,367
`DataReader_take_next_sample`; bei Trennung: RPC an sDDS					1,160	1,167	4,490	0,044
Eingang RPC im sDDS					8,240	8,252	13,750	0,107
sDDS sendet RPC Antwort					3,390	3,396	6,960	0,048
Daten lesbar in Anwendung	3,340	3,352	8,820	0,075	1,160	1,167	4,490	0,044

Der Empfangsvorgang läuft in den ersten Phasen bis zur Komponententrennung ähnlich ab. Danach unterscheiden sich die Ausführungszeiten: Während die Anwendung ohne Trennung im Callback `DataReader_take_next_sample()` direkt aufrufen kann, sendet die Variante mit Trennung ein Event an den Callback-Manager in der Anwendung. Die Aktivierung des Callback-Managers benötigt durchschnittlich 20,415 µs, da die Tasks der Anwendung eine niedrigere Priorität als die der sDDS-Partition haben und in dieser Zeit der sDDS-Stack wieder empfangsbereit wird. Der Callback-Manager ruft den registrierten Callback auf, welcher die Daten per `DataReader_take_next_`

`sample()` abruft. Dies wird durch in einen synchronen RPC ins sDDS umgesetzt, welcher zwei weitere Kontextwechsel erfordert.

Tabelle 2 zeigt den Sendevorgang, der sich in den ersten Schritten unterscheidet: Während die Anwendung ohne Trennung `DataWriter_write()` direkt aufruft, wird bei der Trennung ein RPC an sDDS geschickt. Danach zeigen beide Ansätze ähnliche Ausführungszeiten. Ein Unterschied zeigt sich am Ende der Sendephase bis zur Rückkehr in die Anwendung. Interne Arbeiten der lwIP-State-Machine führen zu Verzögerungen, die durch die unterschiedliche Priorisierung der Tasks sichtbar werden.

Tabelle 2. Messwerte Sendevorgang in µs

Phase	ohne Trennung				mit Trennung			
	MIN	AVG	MAX	STD	MIN	AVG	MAX	STD
Aufruf `DataWriter_write`; bei Trennung: RPC an sDDS					6,630	8,272	8,300	0,168
Eingang RPC im sDDS					1,180	1,390	18,070	1,093
Ausführung `DataWriter_write`	6,010	13,909	14,030	0,864	8,910	13,957	14,000	0,392
Generierung SNPS Paket	4,090	5,482	18,060	0,736	4,130	5,409	6,230	0,103
sDDS UDP-Modul	7,130	7,145	7,390	0,020	7,130	7,142	7,390	0,020
lwIP-Stack	23,140	23,175	23,570	0,031	23,210	23,240	23,610	0,030
Ethernet-Treiber	25,550	25,567	25,580	0,005	16,270	16,284	16,300	0,005
Zurück in der Anwendung					13,170	13,178	13,190	0,004

Der Bedarf an RAM-Speicher für das Gesamtprojekt mit Kernel steigt bei der Lösung mit Trennung im Vergleich zu der ohne um 8,6 % von 60 592 Byte auf 65 792 Byte an. Hauptkostenpunkt sind hier die Stacks für zusätzliche Tasks auf Anwendungs- und sDDS-Seite. Ebenso steigt die Größe des Programmcodes um 7,9 % von 83 944 Byte auf 90 608 Byte an. Dies ist durch das Hinzukommen der Stubs, weiterer Tasks und zwei Partitionen zu erklären.

5 Diskussion und vergleichbare Arbeiten

Die durchschnittliche Gesamtausführungszeit des Empfangsvorgangs beträgt ohne Trennung 87,0 µs und mit Trennung 127,0 µs. Die Zeiten beim Sendevorgang liegen dagegen näher beieinander: 75,3 µs für die Lösung ohne Trennung, und 88,9 µs mit Trennung. Diese Werte beinhalten allerdings noch den Overhead für die Messpunkte. Bereinigt um die Zeit für das Setzen eines Messpunktes ergeben sich für den Empfangsvorgang 80 µs bzw. 115 µs, was einen Unterschied von 44 % entspricht. Die Zeiten beim Sendevorgang reduzieren sich auf 70 µs bzw. 81 µs, der Unterschied beträgt hier 16 %. Treibender Faktor hierbei sind die zusätzlichen Kontextwechsel zwischen den eingefügten Tasks, die auch den Speicherverbrauch mit ihren Stacks erhöhen.

Ähnliche Ergebnisse wurden auch bei Arbeiten zur Dekomposition von monolitischen Systemen auf Mikrokernen beobachtet. *SawMill* [8] implementiert ein Dateisystem als Server auf dem Mikrokern *L4*. Dieser Ansatz ist mit der Trennung auf Applikationsseite vergleichbar. Dagegen sind Trennungskonzepte für Hypervisor eher auf unteren Schichten zu finden: *XEN* [9] virtualisiert Platten- und Netzwerkzugriffe auf der Ebene von Blöcken und Ethernet-Frames. Dies bietet sich für Systeme im Server-Umfeld an, da hier vollständige Betriebssysteme virtualisiert werden.

Einen vergleichbaren Middleware-Ansatz wie DDS bietet *AUTOSAR* mit dem *Runtime Environment, RTE* [5]. Die RTE-Schicht besteht aus generiertem Code und ermöglicht den Datenaustausch zwischen Anwendungen auf verschiedenen Steuergeräten unabhängig von der verwendeten Übertragungstechnik (CAN, Ethernet, usw.). Ebenfalls können hier Anwendungen vom Rest des Systems, der sogenannten *Basis Software (BSW)*, getrennt werden. Allerdings ist AUTOSAR bisher auf Anwendungen aus dem Automobilbereich beschränkt.

Zusammengefasst zeigen die Ergebnisse, dass Trennungskonzepte (und damit mehr funktionale Sicherheit) ihren Preis haben. Den größten Faktor machen die Performance-Einbußen durch zusätzliche Kontextwechsel aus. In diesem Beitrag wurde eine Lösung gewählt, die die Trennung an der Schnittstelle zwischen Anwendung und Middleware vorsieht, um den Speicherbedarf für Netzwerk-Frames möglichst klein zu halten. Allerdings treibt dieser Ansatz den Speicherverbrauch für Stacks zusätzlicher Tasks in die Höhe. Somit ist eine Trennung immer ein Ausgleich zwischen RAM-Verbrauch (Puffer, Stacks), ROM-Verbrauch (geteilter Code, generierte Stubs) und Overhead durch zusätzliche Kontextwechsel.

In zukünftigen Arbeiten bietet es sich an, die Trennung auch an den anderen diskutierten Stellen zu implementieren und zu vermessen. Ebenso kann der bisherige Ansatz vom Speicherverbrauch her weiter optimiert werden.

Literaturverzeichnis

1. K. Beckmann, O. Dedi: *sDDS: A portable data distribution service implementation for WSN and IoT platforms*, WISES, 2015
2. Object Management Group: *Data Distribution Service*, Version 1.4, April 2015, `http://www.omg.org/spec/DDS/1.4/` (abgerufen am 22.06.2016)
3. A. Züpke, M. Bommert, D. Lohmann: *AUTOBEST: A United AUTOSAR-OS and ARINC 653 Kernel*, RTAS, 2015
4. Lightweight TCP/IP Stack, `http://savannah.nongnu.org/projects/lwip/` (abgerufen am 22.06.2016)
5. AUTOSAR AUTomotive Open System ARchitecture
6. AEEC: ARINC Specification 653: Avionics Application Software Standard Interface, 2010
7. Object Management Group, `http://www.omg.org/` (abgerufen am 22.06.2016)
8. A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J. E. Tidswell, L. Deller, L. Reuther: *The SawMill Multiserver Approach*, 9th ACM SIGOPS European Workshop, 2000
9. B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, R. Neugebauer: *Xen and the Art of Virtualization*, SOSP, 2003

http://www.springer.com/978-3-662-53442-7

Internet der Dinge
Echtzeit 2016
Halang, W.A.; Unger, H. (Hrsg.)
2016, VIII, 142 S. 26 Abb., Softcover
ISBN: 978-3-662-53442-7