Entwurf und Implementierung eines Systems zur gesicherten Übertragung von Daten für die automatisierte Autorisierung von Kommunikationspartnern in Constrained Environments

Tobias Hartwich

Matrikelnummer: 2291287

DIPLOMARBEIT

eingereicht am Diplomstudiengang

Informatik

an der Universität Bremen

im Juni 2015

Vorgelegt bei: Dr.-Ing. Olaf Bergmann
Prof. Dr. Ute Bormann

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Bremen, am 11. Juni 2015

Tobias Hartwich

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

ACL	Access Control List
AES	Advanced Encryption Standard
AI	Autorisierungsinformationen
AIF	Authorization Information Format
C	Client
CA	Certification Authority
CAM	Client Authorization Manager
CBC-Mode	Cipher-Block-Chaining-Mode
CBOR	Concise Binary Object Representation
CCM-Mode	Counter with CBC-MAC-Mode
CO_2	Kohlenstoffdioxid
CoAP	Constrained Application Protocol
COP	Client Overseeing Principal
CoRE	Constrained RESTful Environments
CSR	Certificate-Signing-Request
DAC	Discretionary Access Control
DCAF	Delegated CoAP Authentication and Authorization Framework
DTLS	Datagram Transport Layer Security
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
JSON	JavaScript Object Notation
M2M	Maschine-zu-Maschine-Kommunikation
MAC	Message Authentication Code
MTU	Maximum Transmission Unit
PolA	Principle Of Least Authority
PSK	Pre-Shared Key
R	Ressource
ROP	Resource Overseeing Principal
RTT	Round-Trip-Time
SAI	SAM Authorization Information
SAM	Server Authorization Manager
S	Server
SLIP	Serial Line Internet Protocol
TLS	Transport Layer Security
URI	Uniform Resource Identifier
WPAN	Wireless Personal Area Network
WSN	Wireless Sensor Network

Kapitel 1

Einleitung

Das Internet hat sich in den vergangenen 50 Jahren von einem kleinen Forschungsverbund zu einem globalen Netz mit mehrere Milliarden Teilnehmern entwickelt und gilt Vielen als größte Errungenschaft seit der industriellen Revolution vor 200 Jahren [Thi03]. Fortschritte in der Fertigungstechnik in den letzten Jahrzehnten ermöglichen die Produktion immer kleinerer und leistungsfähigerer Computer. Während das Internet bis vor einigen Jahren noch nahezu ausschließlich aus dem Verbund vollwertiger Rechner bestand, geht heute bereits mehr als 30 % des Traffics von mobilen vernetzten Geräten, wie Smartphones, aus.[1]

Als nächster Entwicklungsschritt des Internets gilt die Vernetzung von Gegenständen der realen Welt und eine Integration dieser in das Internet [Eva11]. Dieser Schritt und die für die Realisierung nötigen technischen Entwicklungen werden unter dem Begriff *Internet of Things (IoT)* zusammengefasst. Durch den technischen Fortschritt können Gegenstände der realen Welt mit immer kleineren Computern ausgestattet werden. Diese kommunizieren untereinander und interagieren mit ihrer Umwelt, um Informationen über diese über das Internet bereitzustellen. Damit tragen sie zu einer Verschmelzung von digitaler und realer Welt bei.

Für das Internet of Things wird bis 2020 ein explosionsartiger Anstieg auf 50 Milliarden Geräte und eine Durchdringung sämtlicher Lebensbereiche erwartet [Eva11]. Die Integration von Gegenständen in das Internet führt zu einem veränderten Interaktionsmuster. Während das Internet die Vernetzung von Menschen und eine Interaktion zwischen diesen zum Ziel hatte, führt die Integration von Milliarden intelligenter Gegenstände zu einem Anstieg der direkten Kommunikation zwischen Computern ohne direkte Beteiligung eines Menschen. Diese Interaktionsform wird als *Maschine-zu-Maschine-Kommunikation (M2M)* bezeichnet und gilt als treibende Kraft des Internet of Things.

Um die Geräte des Internet of Things in reale Gegenstände zu integrieren, müssen diese andere Anforderungen erfüllen als herkömmliche Computer. Sie müssen klein genug sein, um transparent in Gegenstände integriert zu werden. Es handelt sich häufig um Geräte, die nur wenige spezielle Aufgaben erfüllen müssen. Diese müssen sie allerdings autonom, ohne feste Stromversorgung und über einen langen Zeitraum, durchführen.

[1] http://de.statista.com/infografik/1092/anteil-mobiler-geraete-am-internet-traffic (abgerufen am 08. Juni 2015)

Aus diesem Grund werden für das Internet of Things Geräte benötigt, die einen niedrigen Energiebedarf aufweisen, der auch über einen längeren Zeitraum durch Batterien erfüllt werden kann. Daher können als Geräte des Internet of Things keine leistungsstarken Geräte, wie Smartphones, verwendet werden, die nur wenige Tage mit einer Batterieladung auskommen.

Stattdessen werden im Internet of Things in der Regel sogenannte *Constrained Devices* verwendet. Sie sind hinsichtlich ihrer Rechenleistung und ihres Speichers im Vergleich zu herkömmlichen Computern deutlich eingeschränkter. Darüber hinaus müssen die Geräte mit den sich spontan verändernden Netzstrukturen des Internet of Things zurechtkommen.

Constrained Devices können aufgrund dieser konzeptionellen Unterschiede nicht ohne Weiteres die Protokolle des Internets verwenden und in dieses integriert werden. Beispielsweise können die Protokolle TCP und HTTP, die im Internet häufig für das Abfragen von Ressourcen verwendet werden, eingeschränkte Geräte bereits überfordern. Daher sind Anpassungen bestehender Protokolle an die Anforderungen eingeschränkter Geräte vorzunehmen oder neue Protokolle zu entwickeln, die speziell auf eingeschränkte Geräte zugeschnitten sind.

Die in Gegenstände der realen Welt integrierten Geräte enthalten in der Regel Sensoren und Aktoren, mit denen sie ihre Umwelt wahrnehmen und Einfluss auf diese nehmen können. Die anfallenden Sensordaten enthalten Zustände der realen Welt und damit Informationen, die für den Besitzer des Gerätes unter Umständen schützenswert sind. Daher ist die Sicherstellung der Vertraulichkeit, Integrität und Authentizität der erhobenen Sensordaten in vielen Anwendungsfällen des Internet of Things eine zentrale Anforderung. Aktoren können sogar aktiv in die reale Welt eingreifen, indem sie beispielsweise andere Geräte schalten oder Systeme basierend auf Sensordaten regulieren. Ein Zugriff durch unautorisierte Personen auf Aktoren kann somit direkte Folgen in der realen Welt haben. Daher gilt auch an dieser Stelle eine Absicherung der Kommunikation und des Zugriffs als zentrale Anforderung für sichere Systeme des Internet of Things.

Der Schutz von Vertraulichkeit, Integrität und Authentizität der während der Kommunikation übertragenen Daten kann durch Verschlüsselung erreicht werden. Das im Internet für die Verschlüsselung von HTTP-Verbindungen verwendete TLS ist allerdings ebenfalls nicht auf die Bedürfnisse eingeschränkter Geräte zugeschnitten. Insbesondere die häufig eingesetzte asymmetrische Verschlüsselung und die notwendige Auswertung von Zertifikaten würde eingeschränkte Geräte überfordern.

Neben der Verschlüsselung der Kommunikation ist für einen effektiven Zugriffsschutz außerdem eine Authentifizierung und Autorisierung der Kommunikationspartner notwendig. Im Internet können diese Aufgaben meist die nicht eingeschränkten Kommunikationspartner selbst durchführen. Die eingeschränkten Geräte des Internet of Things können dies im Allgemeinen nicht. Im Internet of Things werden daher häufig zentralisierte Systeme verwendet, in denen eine zentrale, nicht eingeschränkte Einheit sowohl für die Authentifizierung und Autorisierung als auch für die Durchsetzung der Zugriffskontrolle zur Zugriffszeit zuständig ist. In solchen Systemen stellt die zentrale Einheit häufig einen Single Point of Failure dar, da durch den Ausfall oder die Nichterreichbarkeit der zentralen Einheit ein Zugriff nicht mehr möglich ist.

Einen anderen Ansatz zur Authentifizierung und Autorisierung von Kommunikationspartnern und zur sicheren Kommunikation in eingeschränkten Netzen verfolgt das *Delegated CoAP Authentication and Authorization Framework (DCAF)* [GBB15a]. DCAF verwendet eine dezentrale Architektur, in der die Authentifizierung und Autorisierung von der Durchsetzung der Zugriffskontrolle getrennt auf unterschiedlichen Geräten abläuft. Mit DCAF können eingeschränkte Geräte Authentifizierungs- und Autorisierungsaufgaben, die sie selbst überfordern würden, an weniger eingeschränkte Autorisierungsmanager delegieren. Die anschließend notwendige Zugriffskontrolle kann zur Zugriffszeit auf dem eingeschränkten Gerät durchgeführt werden. Durch diese dezentrale Architektur fällt in DCAF der, in zentralisierten Systemen vorhandene, Single Point of Failure weg, da die Zugriffskontrolle autonom vom eingeschränkten Gerät durchgeführt werden kann.

Darüber hinaus ermöglicht DCAF eine Ende-zu-Ende verschlüsselte Kommunikation zwischen eingeschränkten Geräten. DCAF stellt hierfür einen auf Tickets basierenden Mechanismus bereit, der die eingeschränkten Geräte davon befreit für jeden ihrer Kommunikationspartner Schlüssel zur sicheren Kommunikation aufzubewahren.

1.1 Ziele der Arbeit

In dieser Arbeit wird, basierend auf der Architektur und dem Protokollablauf von DCAF, ein System zur sicheren, dezentralen Kommunikation in eingeschränkten Umgebungen anhand eines konkreten Nutzungsszenarios umgesetzt und evaluiert. Der Schwerpunkt liegt hierbei auf dem Entwurf und der Umsetzung des in DCAF benötigten *Server Authorization Managers (SAM)*. SAM ist in DCAF für die Durchführung der Authentifizierung und Autorisierung zuständig und muss zu diesem Zweck Tickets ausstellen können. In DCAF werden zwar die Aufgaben und Funktionen von SAM definiert, allerdings wird nicht festgelegt, wie SAM zu entwerfen ist, um diese Aufgaben zu erfüllen.

Darüber hinaus wird in dieser Arbeit der DCAF-Protokollablauf implementiert und evaluiert. Insbesondere wird der Frage nachgegangen, ob DCAF für die sichere Kommunikation und die dezentrale Authentifizierung und Autorisierung in eingeschränkten Netzen geeignet ist. Zur Umsetzung des Protokollablaufs und zur Evaluation anhand eines konkreten Nutzungsszenarios sind neben SAM noch weitere Akteure zu entwerfen und umzusetzen.

Eingeschränkte Geräte des Internet of Things werden während ihres Lebenszyklus' in der Regel in verschiedenen Sicherheitsdomänen verwendet. DCAF setzt voraus, dass zwischen den eingeschränkten Geräten und ihren Autorisierungsmanagern bereits Sicherheitsbeziehungen bestehen. Daher wird in dieser Arbeit zusätzlich eine Lösung für den Übergang eines eingeschränkten Gerätes zwischen zwei Sicherheitsdomänen umgesetzt und ausgewertet. Hierfür wird insbesondere der für den Aufbau der Sicherheitsbeziehung notwendige Datenaustausch entworfen und implementiert.

1.2 Herangehensweise

Kapitel 2 gibt zunächst einen Überblick über das Thema Internet of Things und die in diesem verwendeten Netzstrukturen und Übertragungsmechanismen. Anschließend werden verschiedene Modelle zur Autorisierung und Zugriffskontrolle vorgestellt. Im Anschluss werden die in DCAF verwendeten Kommunikationsprotokolle DTLS und CoAP beschrieben. Abgeschlossen wird das Grundlagenkapitel durch die Beschreibung der von DCAF vorgesehenen Akteure und des Protokollablaufs.

In Kapitel 3 wird zunächst das verwendete Nutzungsszenario aus dem Logistikbereich vorgestellt. Außerdem werden die Verwendung und der Ablauf von DCAF anhand des Szenarios erklärt. Anschließend werden die umzusetzenden Anforderungen an das gesamte System und an die einzelnen Akteure der DCAF-Architektur beschrieben.

In Kapitel 4 wird beschrieben wie die, in Kapitel 3 definierten, Anforderungen für das konkrete Nutzungsszenario entworfen und umgesetzt werden, um eine sichere Kommunikation nach dem in DCAF definierten Protokollablauf zu ermöglichen. Außerdem werden Entwurfsentscheidungen für das gesamte System und die einzelnen Akteure getroffen. Da der Schwerpunkt bei den Akteuren in dieser Arbeit auf dem Server Authorization Manager liegen soll, werden Entwurfsentscheidungen für diesen ausführlich beschrieben. Insbesondere wird das System zur Verwaltung und Auswertung der zur Autorisierung notwendigen Zugriffsregeln erklärt.

In Kapitel 5 werden anschließend technische Details zur Implementierung der, in Kapitel 4 entworfenen, Akteure beschrieben. Außerdem wird die konkrete technische Umsetzung des Nutzungsszenarios mit den eingeschränkten und weniger eingeschränkten Akteuren und die auf diesen verwendeten Softwarebibliotheken vorgestellt.

Der Protokollablauf von DCAF und die konkrete Umsetzung des Nutzungsszenarios werden in Kapitel 6 ausgewertet. Hierbei wird zunächst der Frage nachgegangen, ob die von DCAF vorgeschlagene Architektur und der Protkollablauf für eingeschränkte Geräte geeignet sind oder, ob diese die eingeschränkten Geräte hinsichtlich ihres Speichers oder ihrer Prozessorleistung überfordern. Im Anschluss wird zunächst die Inbetriebnahme des Servers und anschließend die einzelnen Schritte des DCAF-Protokollablaufs anhand des konkreten Nutzungsszenarios ausgewertet.

Kapitel 2

Grundlagen

2.1 Internet of Things

Das Internet of Things (IoT) ist bereits seit einigen Jahren Gegenstand der Forschung. Zum ersten Mal wurde der Begriff 1999 von Kevin Ashton im Zusammenhang mit der RFID-gestützten Überwachung von Lieferketten in der Logistik verwendet [Ash09]. Der Begriff vereint eine ganze Reihe von, auch bereits zuvor bekannten, Konzepten. Hierzu zählen unter anderem die Konzepte des Pervasive Computing, der Sensornetze, der Ambient Intelligence und der ubiquitären Computer [Fle09]. Bereits 1991 wurde in [Wei91] von allgegenwärtigen Computern geschrieben, die sämtliche Lebensbereiche durchdringen werden und zu intelligenten Umgebungen führen.

Das Internet of Things steht für eine Vision, in der die physische und die digitale Welt miteinander verschmelzen. Physische Gegenstände (Things) werden mit Computern ausgestattet und zu sogenannten *Smart Things*. Sie können mit unterschiedlichen Sensoren, Aktoren und Kommunikationsmitteln ausgestattet werden und in unterschiedlichen Netzen miteinander, mit ihrer Umgebung und ihren Besitzern kommunizieren und interagieren [Fle09]. Ermöglicht wird diese neue Art der Interaktion durch Fortschritte in der Hardware- und Fertigungstechnik. Diese ermöglichen eine Verringerung der Größe, der Kosten und des Energieverbrauchs von Computern und eine Verbesserung der drahtlosen Kommunikationsmöglichkeiten [PM04].

Das Internet of Things kann als Erweiterung des bisherigen Internets angesehen werden, da auf existierende Technologien des Internets aufgebaut wird. Dennoch gibt es einige wesentliche Unterschiede zu herkömmlichen Netzstrukturen. Im Jahr 2014 gab es weltweit etwa drei Milliarden mit dem Internet verbundene Geräte.[1] Obwohl auch im Internet in den letzten Jahrzehnten ein rasantes Wachstum der Nutzerzahlen zu beobachten war, wird dieser Wachstum und die gesamte Anzahl an Geräten im Internet of Things um ein Vielfaches höher sein. Das Unternehmen *Cisco Systems* geht davon aus, dass in fünf Jahren etwa 50 Milliarden Geräte im Internet of Things aktiv sein werden, wobei jeder Bewohner des Planeten durchschnittlich 6,85 mit dem IoT verbundene Geräte besitzt [Eva11]. Bei der Entwicklung der heute verwendeten Netzstrukturen des Internets wurde zwar bereits soweit vorausgeplant, dass die Strukturen dem bisheri-

[1] Internet World Stats: http://www.internetworldstats.com/stats.htm (abgerufen am 08. Juni 2015)

gen Nutzeranstieg gewachsen waren, dem explosionsartigen Anstieg an Nutzern und Geräten durch das Internet of Things werden diese allerdings nicht vollständig gerecht [Fle09]. Daher sind andere, auf die Bedürfnisse des Internet of Things zugeschnittene Strukturen, wie Kommunikationsprotokolle oder Adressierungsverfahren, nötig.

Die Milliarden hinzukommenden Geräte des Internet of Things unterscheiden sich insbesondere hinsichtlich ihrer Leistungsfähigkeit von den bisherigen Netzteilnehmern. Bisher werden im Netz in der Regel Geräte wie herkömmliche Desktop-Computer oder Smartphones verwendet, die vergleichsweise leistungsstark sind und die vorhandenen Strukturen nutzen können. Die Geräte des Internet of Things sollen mit ihrer Umgebung und Dingen verschmelzen. Daher wird es sich bei diesen in der Regel um kleinere, sehr viel eingeschränktere Geräte handeln, die jeweils nur wenige Aufgaben ausführen können und häufig sogar über längere Zeit im Batteriebetrieb verwendet werden. Aufgrund ihrer begrenzten Systemressourcen ist eine Eingliederung in die bisherigen Strukturen mit den heute üblichen Protokollen nicht problemlos möglich [Fle09].

Das Internet wurde als nutzerzentriertes Netz für die Kommunikation zwischen Menschen entwickelt. Im eng vernetzten Internet of Things wird hingegen verstärkt eine Kommunikation zwischen Computern, ohne menschliches Zutun, stattfinden. Durch diese sogenannte Maschine-zu-Maschine-Kommunikation kann eine sehr viel höhere Automatisierung von Abläufen und eine Verbindung zwischen Dingen der physischen Welt erreicht werden [Jay+13].

Durch das Zusammenwachsen von physischer und digitaler Welt ergeben sich viele mögliche Anwendungsbereiche in sämtlichen Lebensbereichen. Beispielsweise finden sich zahllose Anwendungsfälle in der Industrie und der Logistik zur Überwachung von Produktionsabläufen oder Lieferketten. Auch in der Medizin und im Gesundheits- und Sportbereich sind vielfältige Anwendungen denkbar. Darüber hinaus kann auch der Unterhaltungs- und Lifestylebereich, insbesondere in Form von Smart Homes, vom Internet of Things profitieren [Kop11].

Da die Geräte des Internet of Things durch Sensoren und Aktoren mit der physischen Welt interagieren, sind besondere Anforderungen an Sicherheit und Datenschutz zu stellen. Gesammelte Sensordaten können, sollten sie Dritten zugänglich sein, private und schützenswerte Informationen preisgeben und eine Profilbildung ermöglichen. Sicherheitsprobleme auf Aktoren können noch gravierender sein, da sie aktiv in ihre Umgebung eingreifen und diese unter Umständen manipulieren können [May09]. Darüber hinaus werden Systeme des Internet of Things, durch die Durchdringung aller Lebensbereiche und die Verwendung von sämtlichen Benutzerschichten, häufig von Nutzern verwendet, die generell wenig Erfahrungen mit der Konfiguration von sicherheitsrelevanten Systemen haben [RNL11].

2.2 Constrained-Node Networks

Wie bereits in Abschnitt 2.1 beschrieben, bestehen die Geräte des Internet of Things häufig aus leistungsschwacher Hardware. Für diese Art von Geräten werden in [RFC7228] die Begriffe *Constrained Device* und *Constrained Node* eingeführt und definiert. Die Geräte können hinsichtlich ihrer Rechenleistung, ihres Speichers, ihrer Übertragungskapazität und der zur Verfügung stehenden Energie beschränkt sein.

Daher können sie in der Regel nicht alle herkömmlichen Protokolle des Internets verwenden.

In [RFC7228] werden die Geräte in drei Klassen nach ihrer Eingeschränktheit hinsichtlich ihres verfügbaren Speichers eingeteilt. Tabelle 2.1 zeigt diese Einteilung.

Klasse	*Arbeitsspeicher*	*Programmspeicher*
Class 0 (C0)	unter 10 KiB	unter 100 KiB
Class 1 (C1)	etwa 10 KiB	etwa 100 KiB
Class 2 (C2)	etwa 50 KiB	etwa 250 KiB

Tabelle 2.1: Klassifizierung eingeschränkter Geräte nach [RFC7228]

Geräte der Klasse C0 haben weniger als 10 KiB Arbeitsspeicher und weniger als 100 KiB Programmspeicher, weshalb sie als sehr eingeschränkt gelten. Sie können nicht direkt und sicher mit Geräten des Internets kommunizieren, sondern müssen hierfür die Hilfe weniger eingeschränkter Proxies, Gateways oder Server in Anspruch nehmen. Sie können außerdem nicht wie traditionelle Geräte gesichert oder verwaltet werden. Geräte die in Klasse C1 fallen, gelten mit etwa 10 KiB Arbeitsspeicher und etwa 100 KiB Programmspeicher, ebenfalls noch als ziemlich eingeschränkt. Sie sind für die Verwendung von gängigen Internetprotokollen, wie HTTP oder TLS, nicht geeignet. Sie können allerdings Protokolle verwenden, die speziell für eingeschränkte Geräte entwickelt wurden. Geräte der Klasse C1 haben allerdings nicht ausreichend Speicher, um als universelle Computer eingesetzt zu werden. Sie können daher ausschließlich anwendungsspezifisch genutzt werden. In Klasse C2 haben die Geräte bereits etwa 50 KiB Arbeitsspeicher und etwa 250 KiB Programmspeicher. Sie gelten daher als weniger eingeschränkt und können in der Regel die meisten Funktionen und Protokolle des Internets verwenden. Daher werden für diese Geräteklasse keine Protokolle, die speziell auf die Bedürffnisse eingeschränkter Geräte zugeschnitten sind, benötigt [RFC7228].

Eingeschränkte Geräte können untereinander kommunizieren und sogenannte *Constrained-Node Networks* bilden. Auch diese Netze selbst können, wie die darin verwendeten Geräte, beispielsweise hinsichtlich der zu erwartenden Übertragungskapazität eingeschränkt sein [RFC7228]. Constrained-Node Networks werden häufig auch als *Wireless Sensor Networks (WSN)* oder *Low power and Lossy Networks* bezeichnet.

2.3 Netzwerkstack

Im Internet werden auf der Bitübertragungs- und Sicherungsschicht des OSI-Modells in der Regel Ethernet (IEEE 802.3)[2] oder Wireless LAN (IEEE 802.11)[3] und dazu leistungsfähige Hardware verwendet, die Bandbreiten von vielen Mbit/s oder gar Gbit/s ermöglicht. Ethernet und Wireless LANs sind für eingeschränkte Geräte, aufgrund ihrer geringen Leistungsaufnahme und geringen maximalen Datenübertragungsraten, nicht geeignet. Daher wird auf diesen häufig eine Kommunikation über die, in IEEE

[2] http://www.ieee802.org/3/ (abgerufen am 08. Juni 2015)
[3] http://www.ieee802.org/11/ (abgerufen am 08. Juni 2015)

802.15.4[4] festgelegten, Protokolle verwendet. IEEE 802.15.4 definiert Protokolle für die Funkkommunikation von Geräten in *Wireless Personal Area Networks (WPAN).* Bei der in IEEE 802.15.4 festgelegten Übertragung auf der Bitübertragungs- und der Sicherungsschicht, wurde auf die Bedürfnisse von eingeschränkten Geräten Rücksicht genommen und beispielsweise auf Einfachheit und einen geringen Energiebedarf geachtet. Bei der Funkkommunikation kann im Frequenzband von 2400 MHz bis 2483,5 MHz maximal eine Datenrate von 250 kbit/s erreicht werden. Die Maximum Transmission Unit (MTU) beträgt in IEEE 802.15.4 lediglich 127 Byte, sodass größere Datenpakete unter Umständen auf den darüber liegenden Schichten fragmentiert werden müssen.

IEEE 802.15.4 beschreibt ausschließlich die unteren beiden Schichten des OSI-Modells und nimmt keine Definition der darüber liegenden Vermittlungsschicht vor. Daher müssen für die Adressierung und das Routing von Datenpaketen auf IEEE 802.15.4 aufbauende Protokolle verwendet werden. Das Standardprotokoll des Internets hierfür ist das *Internet Protocol (IP)* mit einer Adressierung von Datenpaketen im IPv4- oder IPv6-Format. Da der verfügbare IPv4-Adressraum bereits im derzeitigen Internet knapp wird, wird für das Internet of Things in der Regel vollständig auf IPv6 gesetzt. Bei der Verwendung von IPv6 ist eine MTU von mindestens 1280 Byte vorgeschrieben [RFC2460]. Daher können IPv6-Pakete nicht ohne Weiteres in IEEE-802.15.4-Frames mit einer MTU von 127 Byte übetragen werden. Um dennoch eine IPv6-Kommunikation in IEEE-802.15.4-Netzen zu ermöglichen, kann eine weitere Zwischenschicht, wie *6LoWPAN*, verwendet werden [RFC4944]. 6LoWPAN ist ein Protokoll für die IPv6-basierte Funkkommunikation in *Low power Wireless Personal Area Networks.* Der *6LoWPAN Adaption Layer* ermöglicht eine Umsetzung von IPv6-Paketen der Vermittlungsschicht auf IEEE-802.15.4-Frames der Sicherungsschicht und somit eine Verwendung von IPv6 in IEEE-802.15.4-Netzen. Hierfür werden Mechanismen zur Fragmentierung von Paketen und zur Kompression der Paketheader verwendet [RFC6282]. Außerdem werden auf der 6LoWPAN-Ebene verschiedene Routingverfahren angeboten, wobei beispielsweise das RPL-Routing speziell an die Bedürfnisse der sich häufig verändernden Netzstrukturen in Sensornetzen angepasst wurde und ein Routing über IP ermöglicht.

Die Fragmentierung von Paketen in 6LoWPAN kann in Constrained-Node Networks allerdings selbst wieder zu Problemen für die eingeschränkten Geräte führen. In Constrained-Node Networks treten Paketverluste, aufgrund der heterogenen Netzstrukturen und den eingeschränkten Übertragungskapazitäten, häufiger auf. Bei verloren gegangenen 6LoWPAN-Fragmenten muss allerdings das gesamte Paket, das unter Umständen viele Frames benötigt, erneut übertragen werden. Das Zusammensetzen der einzelnen Frames auf dem Zielgerät bedeutet für eingeschränkte Geräte einen hohen Aufwand und somit einen erhöhten Energiebedarf. Insbesondere in Netzen, in denen Pakete über mehrere Hosts weitergeleitet werden müssen, müssen in jedem einzelnen Schritt (Hop) die Fragmente zusammengesetzt und für das Weiterleiten erneut fragmentiert werden [RFC6606]. Darüber hinaus kann die Fragmentierung in 6LoWPAN zu Sicherheitsproblemen führen. In [Hum+13] werden Angriffszenarien auf die Fragmentierung aufgezeigt, die dazu genutzt werden können, aktiv das korrekte Zusammensetzen von Fragmenten mit geringen Kosten zu verhindern.

[4] http://www.ieee802.org/15/pub/TG4s.html (abgerufen am 08. Juni 2015)

Oberhalb der 6LoWPAN- und der IPv6-Schicht können auf der Transportschicht in Constrained-Node Networks prinzipiell die gleichen Protokolle wie im Internet, insbesondere TCP und UDP, verwendet werden. Allerdings ist der Einsatz des verbindungsorientierten TCP auf eingeschränkten Geräten nicht uneingeschränkt zu empfehlen. Da TCP verbindungsorientiert arbeitet, muss zunächst die Verbindung über einen 3-Wege-Handshake aufgebaut werden. Bei lange bestehenden Verbindungen oder beim Übertragen von großen Datenmengen fällt der Overhead des TCP-Handshakes nicht so sehr ins Gewicht. Im Internet of Things, in dem häufig kleine Datenpakete mit beispielsweise Sensordaten gesendet werden, ist der TCP-Handshake ineffizient [AIM10]. Um Überlastungen und Staus zu vermeiden, verwendet TCP eine Ende-zu-Ende-Staukontrolle, bei der zu Beginn mit einer geringen Senderate gestartet (Slow Start) und diese im Folgenden nach bestimmten Algorithmen erhöht wird. Die häufig kleinen Datenpakete im Internet of Things würden in der Regel über diese Slow-Start-Phase gar nicht hinauskommen und ausschließlich mit der langsamen Senderate übertragen werden [Wan+05]. Darüber hinaus erfordert TCP, dass auf Quell- und Zielseite der Verbindung Zustände und Daten vorgehalten werden müssen, um eine erneute Übertragung von verloren gegangenen Paketen zu ermöglichen. Die zu speichernden Zustände können, insbesondere bei mehreren parallelen Verbindungen, eingeschränkte Geräte überfordern, da diese in der Regel nur wenige KiB Arbeitsspeicher zur Verfügung haben [AIM10]. Daher ist das verbindungslos arbeitende UDP als Transportprotokoll für den Einsatz in Constrained-Node Networks eher geeignet als TCP.

2.4 Datagram Transport Layer Security (DTLS)

Die Datenübertragung auf den, in Abschnitt 2.3 genannten, Schichten, findet zunächst einmal ungesichert und unverschlüsselt statt. Es findet keine Sicherstellung der Vertraulichkeit, Integrität und Authentizität der Daten statt. Insbesondere bei der Funkkommunikation ist Wert auf den Schutz der übertragenen Daten zu legen, da bei der Kommunikation über Funk die Luft als Übertragungsmedium verwendet wird und diese prinzipiell immer ein Broadcast-Medium darstellt, sodass potenziell alle Geräte innerhalb der Funkreichweite die Daten empfangen können.

Die genannten Schutzziele Vertraulichkeit, Integrität und Authentizität können auf unterschiedlichen Schichten des OSI-Modells erreicht werden. Es kann beispielsweise auf IEEE-802.15.4-Ebene eine Verschlüsselung der Daten stattfinden. Auf der Sicherungsschicht kann eine Verschlüsselung allerdings immer nur zwischen zwei Hops und nicht Ende-zu-Ende erreicht werden, da auf dieser Schicht keine Adressinformationen, wie IPv6-Adresse, des Paketempfängers bekannt sind, sondern lediglich die MAC-Adresse des nächsten Hops.

Eine Ende-zu-Ende-Verschlüsselung kann auf der Vermittlungs- oder Transportschicht erreicht werden. Der De-facto-Standard für sichere Verbindungen im Internet unter Verwendung des Transportprotokolls TCP ist *Transport Layer Security (TLS)* [RFC5246]. TLS verschlüsselt Daten, die oberhalb der Transportschicht anfallen.

Aufgrund der, in Abschnitt 2.3 genannten, Probleme beim Einsatz von TCP in Anwendungen des Internet of Things, empfiehlt sich, wie beschrieben, der Einsatz von UDP. Allerdings funktioniert die Transportverschlüsselung TLS nicht ohne Weiteres

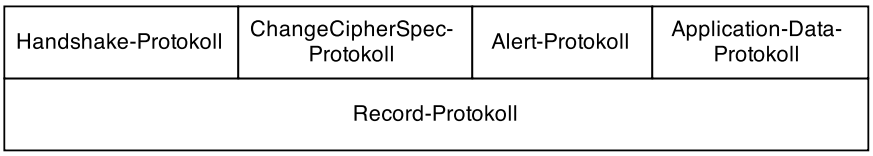

Abbildung 2.1: TLS/DTLS Schichtenarchitektur

über das verbindungslose UDP, da bei UDP Paketverluste auftreten oder Pakete in unterschiedlicher Reihenfolge eintreffen können. TLS sieht keinen Mechanismus vor, um mit verlorenen oder unsortiert eintreffenden Paketen umzugehen. Um TLS über UDP nutzen zu können, wurde *Datagram Transport Layer Security (DTLS)* entwickelt. DTLS ist im Kern identisch zu TLS und wird nur durch Änderungen an diesem beschrieben. DTLS erfüllt allerdings die gleichen Sicherheitsanforderungen wie TLS [RFC6347].

TLS und DTLS basieren intern auf zwei Schichten von Protokollen, wobei die oberen Schichten auf die untere Schicht aufbauen. Der Zusammenhang der Schichten ist in Abbildung 2.1 dargestellt.

Die untere Schicht wird durch das Record-Protokoll realisiert und ermöglicht eine Absicherung der Verbindung durch eine Ende-zu-Ende-Verschlüsselung. Die zu übertragenden Daten werden in TLS und DTLS in verschlüsselte Records verpackt. Das Record-Protokoll ist auf der einen Seite dafür verantwortlich, Nachrichten im Klartext von den höheren Schichten entgegen zu nehmen, sie zu verschlüsseln, zur Sicherung ihrer Integrität mit einem Message Authentication Code (MAC) auszustatten und das Ergebnis zu übertragen. Auf der anderen Seite muss das Record-Protokoll die verschlüsselt empfangenen Daten entschlüsseln, ihre Authentizität und Integrität verifizieren und sie zur weiteren Verarbeitung im Klartext an die höheren Schichten weiterleiten [RFC6347].

In DTLS können Pakete durch die Übertragung über das unzuverlässige UDP in falscher Reihenfolge ankommen. Daher muss auf DTLS-Ebene eine Sortierung von ankommenden Records ermöglicht werden. Zu diesem Zweck wird das Record-Protokoll von TLS in DTLS um eine explizite Sequenznummer und eine Epoch-Nummer erweitert. Jeder Record muss darüber hinaus in DTLS in ein einzelnes UDP-Paket passen. Daher sollte, um eine IP-Fragmentierung zu vermeiden, die DTLS-Implementierung die MTU erkennen und Records senden die kleiner als diese sind [RFC6347, S.9].

Die oberen Schichten in der, in Abbildung 2.1 dargestellten, DTLS-Architektur bauen auf dem darunterliegenden Record-Protokoll auf. Das Handshake-Protokoll dient zum Aufbauen sicherer Verbindungen und dem Aushandeln der dafür nötigen Parameter. In TLS kommt der Handshake mit vier Schritten (Flights) aus. In DTLS wurden zwei zusätzliche Schritte hinzugefügt. Im Folgenden werden die einzelnen Schritte des DTLS-Handshakes beschrieben und es wird auf die Unterschiede zum Handshake in TLS eingegangen. Die im Handshake gesendeten Nachrichten zeigt Abbildung 2.2.

In TLS und DTLS werden den Kommunikationspartnern immer fest die Rollen des Clients und des Servers zugewiesen. Der Handshake wird im ersten Schritt vom Client initiiert, indem dieser eine *ClientHello*-Nachricht an den Server sendet. Diese enthält Informationen zur zu verwendenden DTLS-Version, eine Liste an Algorithmen und

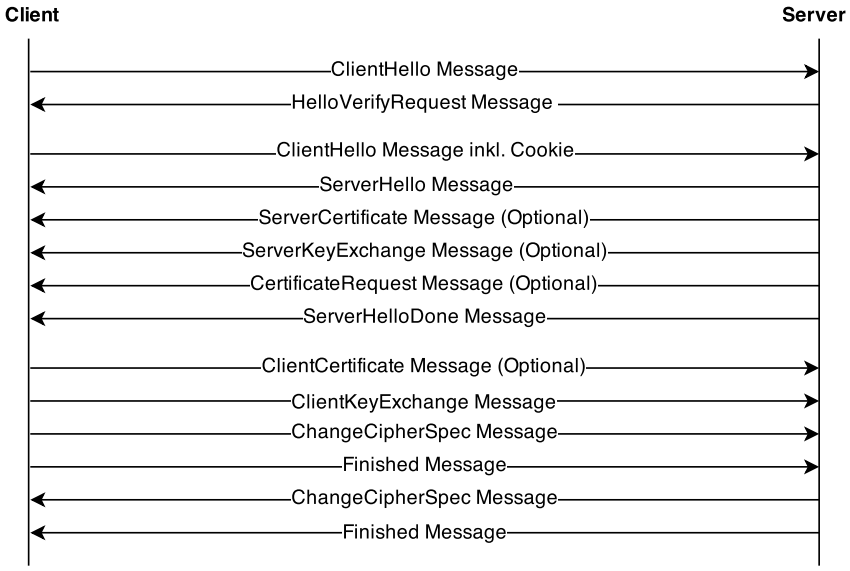

Abbildung 2.2: Ablauf des DTLS-Handshakes nach [RFC6347, S.20]

Kompressionsverfahren, die vom Client unterstützt werden, und eine zufällige Nonce. Diese wird genutzt, um *Replay-Attacken* zu erschweren, bei denen Dritte mitgeschnittene Nachrichten erneut abspielen.

Im TLS-Handshake würde der Server nun bereits mit einer *ServerHello*-Nachricht antworten. In DTLS wurde der Handshake hingegen an dieser Stelle um zwei Schritte erweitert, um *Denial-of-Service-Angriffe (DoS-Angriffe)* zu erschweren. Da während und nach der *ServerHello*-Nachricht auf dem Server bereits Zustände gehalten und aufwändige Operationen durchgeführt werden, könnte ein Angreifer einen Server mit dem Senden von vielen *ClientHello*-Nachrichten überlasten. Daher sendet der Server in DTLS an dieser Stelle zunächst eine *HelloVerifyRequest*-Nachricht an den Client. Diese enthält ein zustandsloses Cookie, welches für den Client zuvor unbekannte Daten enthält. Der Client muss das Cookie anschließend in einer weiteren *ClientHello*-Nachricht zum Server zurücksenden, um diesem zu zeigen, dass er Antworten des Servers unter der verwendeten Adresse empfangen kann.

Der Server verifiziert das erhaltene Cookie und, falls es mit dem zuvor erzeugtem Cookie übereinstimmt, antwortet der Server im vierten Schritt mit einer *ServerHello*-Nachricht. Die Nachricht enthält analog zur *ClientHello*-Nachricht eine Liste mit vom Server unterstützten Algorithmen, wobei diese diesmal eine Untermenge der vom Client unterstützten Algorithmen darstellen. Falls Client und Server keine Teilmenge an unterstützten Algorithmen besitzen, schlägt der Handshake an dieser Stelle fehl.

Während des vierten Schritts können vom Server im Folgenden noch weitere Nachrichten gesendet werden. Mit der *ServerCertificate*-Nachricht und der *ServerKeyExchange*-Nachricht kann der Server dem Client sein Zertifikat senden, um sich auszuweisen und Parameter zur Berechnung des Pre-Master-Secrets auszutauschen. Außerdem kann er mit einer *CertificateRequest*-Nachricht ein Zertifikat oder andere Informationen zum Ausweisen vom Client anfordern. Als letzte Nachricht dieses Schrittes sendet der Server eine *ServerHelloDone*-Nachricht, die dem Client signalisiert, dass dieser Schritt

abgeschlossen ist und keine weiteren Nachrichten vom Server mehr in diesem Schritt zu erwarten sind.

Falls der Client vom Server eine *CertificateRequest*-Nachricht erhalten hat, sendet er ihm im fünften Schritt des Handshakes als Erstes eine *ClientCertificate*-Nachricht, um sich auszuweisen. Anschließend muss auch der Client eine *ClientKeyExchange*-Nachricht zum Server senden, mit der wiederum das Pre-Master-Secret, je nach ausgewählter Cipher Suite, ausgehandelt werden kann.

An dieser Stelle haben sowohl der Client als auch der Server ausreichend Informationen, um jeweils das für die Verbindung zu nutzende Master-Secret zu berechnen. Mit der *ChangeCipherSpec*-Nachricht signalisiert der Client, dass er fortan die ausgehandelten Sicherheitsparameter verwendet. Diese Nachricht gehört intern eigentlich nicht zum Handshake-Protokoll, sondern zu einem eigenen Change-Cipher-Spec-Protokoll. Sie wird allerdings, wie beschrieben, während des Handshakes versendet. Zum Abschluss des fünften Schrittes sendet der Client eine *Finished*-Nachricht. Da diese Nachricht nach der *ChangeCipherSpec*-Nachricht gesendet wird, ist dies die erste Nachricht, die bereits verschlüsselt übertragen wird. Alle weiteren Nachrichten des Clients werden ebenfalls verschlüsselt.

Im sechsten Schritt des Handshakes sendet der Server seinerseits eine *ChangeCipherSpec*-Nachricht und anschließend ebenfalls die erste verschlüsselte *Finished*-Nachricht. Nach diesem letzten Schritt gilt der Handshake als erfolgreich abgeschlossen.

Nachrichten im DTLS-Handshake können unter Umständen, insbesondere wenn Zertifikate ausgetauscht werden, zu groß werden, um als DTLS-Record in einem UDP-Paket gesendet zu werden. Daher sieht DTLS einen Mechanismus vor, um Handshake-Nachrichten zu fragmentieren. Da Nachrichten über das unzuverlässige UDP verloren gehen können, stellt DTLS außerdem während des Handshakes einen einfachen Mechanismus bereit, um verloren gegangene Handshake-Nachrichten erneut zu übertragen (DTLS-Retransmission) [RFC6347, S.19-23].

Neben dem Handshake-Protokoll gibt es auf der oberen Schicht der DTLS-Architektur noch drei weitere Protokolle, die auf das Record-Protokoll der unteren Schicht aufbauen. Hierzu zählt das bereits erwähnte Change-Cipher-Spec-Protokoll, welches aus lediglich einer einzigen 1 Byte großen Nachricht besteht, die dem Kommunikationspartner mitteilt, dass im Folgenden auf die zuvor ausgehandelte Cipher Suite gewechselt wird. Das Alert-Protokoll wird verwendet, um Benachrichtigungen, Statusänderungen und Meldungen im Fehlerfall an den Kommunikationspartner zu senden. Schließlich gibt es noch das Application-Data-Protokoll, welches eine Schnittstelle zur Anwendungsebene bereitstellt und für den verschlüsselten Versand von Nachrichten zuständig ist. Der verschlüsselte Nachrichtenaustausch über dieses Protokoll kann erst nach erfolgreichem Handshake genutzt werden.

Während des Handshakes findet zwischen Client und Server ein Schlüsselaustausch statt, der zur Generierung des Pre-Master-Secrets verwendet wird. Das Pre-Master-Secret kann nur von den beiden Kommunikationspartnern berechnet werden und ist nur diesen bekannt. In [RFC5246] werden für den TLS-Handshake der RSA- und der Diffie-Hellman-Schlüsselaustausch zugelassen. Durch die Erweiterung in [RFC4492] können auch Verfahren, die auf elliptischen Kurven basieren, für den Schlüsselaustausch verwendet werden. Diese sind, aufgrund ihrer kleineren Schlüsselgröße und der

damit verbundenen Reduzierung der zu übertragenden Daten, auch für den Einsatz in Constrained Environments geeignet [RFC4492, S.2].

[RFC5246] erlaubt für TLS die Verwendung von Strom- und Blockchiffren zur Verschlüsselung, wobei allerdings `RC4` der einzige erlaubte Stromchiffre ist. Da dieser allerdings einige bekannte Schwächen aufweist, wird die Verwendung untersagt [RFC7465]. Daher sind derzeit in TLS nur Blockchiffren möglich. In DTLS kommen Stromchiffren grundsätzlich nicht in Frage, da in diesen kein wahlfreier Zugriff möglich ist und daher Zustände und Daten über mehrere Records hinweg aufbewahrt werden müssen [RFC6347, S.12]. An Blockchiffren unterstützt TLS `AES` und `3DES` im Cipher-Block-Chaining-Mode (CBC-Mode) und durch die Erweiterung in [RFC6655] auch im CCM-Mode (Counter with CBC-MAC-Mode). Diese Chiffren stehen auch in DTLS zur Verfügung. Zur Berechnung von *Message Authentication Codes (MACs)* unterstützen TLS und DTLS die Verfahren `MD5` (`HMAC_MD5`), `SHA` (`HMAC_SHA1`) und `SHA256` (`HMAC_SHA256`) [RFC5246, S.74]

Durch die Erweiterungen in [RFC4279] und [RFC5487] können TLS und DTLS statt mit Zertifikaten und asynchroner Verschlüsselung mit Pre-Shared-Keys (PSK) und symmetrischer Verschlüsselung verwendet werden. Durch die Verwendung von symmetrischer Verschlüsselung können aufwändige Operationen zur Verifikation des Zertifikats und der Zertifikatskette vermieden werden. Darüber hinaus sind Pre-Shared-Keys in eingeschränkten Umgebungen aus Verwaltungssicht unkomplizierter als Zertifikate, insbesondere da eingeschränkte Geräte häufig bereits ab Werk mit Sicherheitsparametern bestückt werden und hierbei das Bestücken mit Pre-Shared-Keys weniger aufwändig ist als das Bestücken mit Zertifikaten [Kot+12]. Bei der Verwendung von Pre-Shared-Keys wird im Handshake in der *ClientKeyExchange*-Nachricht eine PSK-Identity mitgesendet, die es dem Kommunikationspartner ermöglicht einen passenden Schlüssel auszuwählen.

2.5 Constrained Application Protocol (CoAP)

In Constrained-Node Networks empfiehlt sich nicht nur auf den bisher betrachteten Schichten des OSI-Modells der Einsatz von speziell für eingeschränkte Geräte angepassten Protokollen, sondern auch auf der Anwendungsebene. Im Internet ist auf der Anwendungsschicht in den meisten Fällen das *Hypertext Transfer Protocol (HTTP)* das Protokoll der Wahl. Allerdings wurde dieses nicht auf die Bedürfnisse von eingeschränkten Geräten zugeschnitten und ist somit nicht ideal für den Einsatz in Constrained-Node Networks geeignet. Zum einen ist HTTP auf ein zuverlässiges Transportprotokoll, wie TCP, angewiesen, welches, wie in Abschnitt 2.3 beschrieben, seinerseits Probleme beim Einsatz in Constrained-Node Networks verursachen kann. Zum anderen besitzen Nachrichten in HTTP häufig einen hohen Overhead, da die Header in HTTP-Nachrichten in einem textuellen Format vorliegen.

Ein Kommunikationsprotokoll der Anwendungsschicht, welches speziell für den Einsatz in Constrained-Node Networks zugeschnitten ist, ist das *Constrained Application Protocol (CoAP)* [RFC7252]. CoAP verwendet inhaltlich viele Konzepte von HTTP wieder und wurde darüber hinaus so entwickelt, dass eine unkomplizierte Übersetzung von CoAP-Nachrichten in HTTP-Nachrichten möglich ist und sich somit CoAP-Geräte

in das Internet integrieren lassen. Als Transportprotokoll kann CoAP unter anderem das für eingeschränkte Geräte geeignete UDP verwenden.

Ein Konzept, das CoAP von HTTP übernommen hat, ist das Interaktionsmodell. In CoAP wird, wie in HTTP, ein Client-Server-Modell verwendet. CoAP und HTTP stellen universelle Protokolle für Dienste nach der REST-Architektur (Representational State Transfer) dar. Beide Protokolle arbeiten zustandslos und können Ressourcen bereitstellen, die jeweils über eindeutige Web-Adressen, sogenannte *Uniform Resource Identifier (URI)* [RFC3986], erreichbar sind. CoAP unterstützt mit `GET`, `POST`, `PUT`, und `DELETE` eine Untermenge der von HTTP unterstützten Zugriffsmethoden. [RFC7252] schreibt für CoAP analog zu HTTP folgendes URI-Schema vor:

```
CoAP-URI = "coap(s):" "//" host [ ":" port ] path-abempty [ "?" query]
```

Sichere CoAP-Verbindungen werden über DTLS ermöglicht und verwenden das URI-Prefix `coaps://`.

CoAP verwendet, im Gegensatz zu HTTP, ein binäres Nachrichtenformat, welches nur einen geringen Overhead benötigt und somit eine Anforderung von eingeschränkten Geräten erfüllt. In Constrained-Node Networks können große Datenpakete zur Fragmentierung führen. Außerdem könnten große Nachrichten, wie sie bei HTTP üblich sind, die eingeschränkten Geräte selbst überfordern. In Abschnitt 2.1 wurde beschrieben, dass im Internet of Things die M2M-Kommunikation eine große Rolle spielen wird. CoAP unterstützt diese Art der autonomen Kommunikation zwischen Geräten, indem es die Standardisierung von Schnittstellen mit der REST-Architektur unterstützt und darüber hinaus die Verwendung von Verzeichnisdiensten ermöglicht. Diese werden in Constrained-Node Networks benötigt, um den beteiligten Akteuren Informationen über die im Netz verfügbaren Ressourcen bereitzustellen. In CoAP können entweder separate Akteure als Verzeichnisdienst verwendet werden oder die Endgeräte stellen diesen selbst im dafür vorgesehenen *Constrained RESTful Environments (CoRE) Link Format* [RFC6690] unter dem URI `/.well-known` bereit [RFC5785].

Die Hauptaufgabe eines Transferprotokolls der Anwendungsschicht wie CoAP ist das übertragen von Nachrichten in einem festgelegten Format. CoAP sieht vier Nachrichtentypen vor: *Confirmable-*, *Non-Confirmable-*, *Acknowledgement-* und *Reset*-Nachrichten. Da CoAP-Nachrichten in der Regel über unzuverlässige Transportprotokolle wie UDP übertragen werden, stellt CoAP mit den Confirmable-Nachrichten auf Anwendungsebene eine Möglichkeit der zuverlässigen Übertragung zur Verfügung. Der Eingang einer Confirmable-Nachricht muss zur Sicherstellung der Zuverlässigkeit mit einer Acknowledgement-Nachricht beantwortet werden. Wird in einer bestimmten Zeit keine Acknowledgement-Nachricht gesendet, wird die Confirmable-Nachricht erneut übertragen (Retransmission). Alternativ zur Acknowledgement-Nachricht kann als Antwort auch eine Reset-Nachricht gesendet werden, die anzeigt, dass die Nachricht zwar empfangen, aber nicht verarbeitet werden konnte.

In Fällen, in denen eine zuverlässige Übertragung nicht zu den Anforderungen gehört, können Non-Confirmable-Nachrichten verwendet werden. Beispielsweise kann bei Sensorknoten in Constrained-Node Networks, die in regelmäßigen Abständen einen Sensorwert verschicken, der Verlust einer dieser Nachrichten als vertretbar angesehen werden. Anfragen nach REST-Ressourcen können, je nach Anwendungsfall, sowohl als Confirmable- als auch als Non-Confirmable-Nachricht verschickt werden. Die Antwort

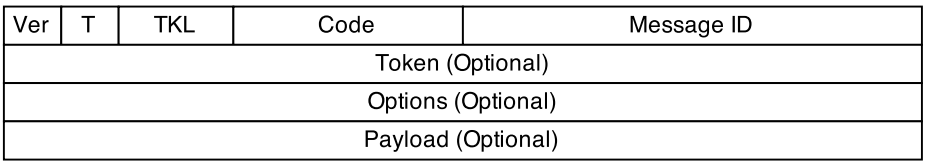

Abbildung 2.3: Aufbau einer CoAP-Nachricht nach [RFC7252, S.15]

kann wiederum ebenfalls in einer Confirmable- oder Non-Confirmable-Nachricht enthalten sein. Alternativ hierzu können Antworten, falls sie in ausreichender Zeit zur Verfügung stehen, auch Huckepack in einer Acknowledgement-Nachricht gesendet werden [RFC7252, S.32]. Hierdurch kann eine separate Nachricht eingespart werden.

CoAP-Nachrichten bestehen, wie in Abbildung 2.3 dargestellt, aus einem festen vier Byte großen binären Header, gefolgt von einem Token, binären Optionen und optional einem Payload. Im Header wird zunächst die CoAP-Versionsnummer (Ver) und der verwendete Nachrichtentyp (T) angegeben. Anschließend folgt die Länge des Tokens (TKL). Token werden in CoAP verwendet, um Anfragen und Antworten einander zuordnen zu können. Sie werden vom Client generiert und müssen vom Server in der Antwort unverändert mitgesendet werden. Auf die Token-Länge folgt im Header die Angabe der in der Anfrage verwendeten Zugriffsmethode (`GET`, `POST`, `PUT` oder `DELETE`) beziehungsweise in der Antwort den Statuscode, der angibt, ob die Anfrage erfolgreich war (Code). Abgeschlossen wird der Header durch eine Message-ID. Sie wird benötigt, um Confirmable- oder Non-Confirmable-Nachrichten ihren jeweiligen Acknowledgement- oder Reset-Nachrichten zuordnen zu können. Außerdem können mit Hilfe der Message-ID doppelte Nachrichten erkannt werden.

Weitere Informationen können mit Hilfe von CoAP-Optionen zur Nachricht hinzugefügt werden. Hierbei orientiert sich CoAP wiederum an den HTTP-Headern. Beispielsweise kann über Optionen die abzufragende Ressource als URI angegeben werden. Außerdem können unter anderem der verwendete Content-Type und Informationen, die zum Caching verwendet werden, angegeben werden.

2.6 Authentifizierung

Zu den Sicherheitsgrundfunktionen zur Abwehr möglicher Angriffe und zur Sicherstellung von Vertraulichkeit und Authentizität zählen die Authentifizierung und Autorisierung[Eck13]. Bei der Authentifizierung überprüft eine Einheit, ob ein Subjekt tatsächlich dasjenige ist, das es vorgibt zu sein. Subjekte können Personen oder auch Maschinen sein. Das Subjekt muss nachweisen, dass es bestimmte Eigenschaften besitzt. Bei diesen Eigenschaften kann es sich beispielsweise um eine Kombination von Benutzername und Passwort handeln, mit der sich das Subjekt authentisiert und als dieser Benutzer ausgibt. Der Kommunikationspartner kann das Subjekt anhand der Kombination eindeutig authentifizieren. Die Kombination aus Benutzername und Passwort ist allerdings nur eine Möglichkeit für das Subjekt seine Identität nachzuweisen. Es kann beispielsweise auch der Aufbau einer verschlüsselten Verbindung zur Authentifizierung verwendet werden. Das Subjekt kann sich in diesem Fall authentisieren,

indem es durch den erfolgreichen Aufbau der verschlüsselten Verbindung nachweisen kann, dass es im Besitz des notwendigen Schlüssels ist.

2.7 Autorisierung

Neben der Authentifizierung gehört die Autorisierung ebenfalls zu den Sicherheitsgrundfunktionen [Eck13]. Die Autorisierung bezeichnet den Prozess der Bestimmung, ob ein Subjekt die notwendigen Rechte besitzt, um auf ein Objekt (beispielsweise eine Ressource) zuzugreifen oder eine andere Aktion auf diesem auszuführen. Die Autorisierung findet meist nach einer erfolgreichen Authentifizierung statt und soll die Vertraulichkeit und Integrität der Objekte sicherstellen [Ker01].

Der Begriff der Autorisierung ist von dem Begriff der Zugriffskontrolle (engl. Access Control) abzugrenzen. Während die Autorisierung den Prozess zur Feststellung, ob bestimmte Rechte vorliegen, bezeichnet, beschreibt die Zugriffskontrolle die eigentliche Durchsetzung der Autorisierung [5]. Für die Autorisierung werden Zugriffsrechte oder -regeln benötigt, die die Subjekte und Objekte der Autorisierung verbinden und festlegen, wie die Subjekte die Objekte verwenden können. Das Ziel der Zugriffskontrolle ist die Verhinderung von unautorisierten Zugriffen auf Objekte [GW11]. Hierfür muss zur Zugriffszeit die Autorisierung überprüft werden. Die Kontrolle sollte sämtliche Zugriffe umfassen und nicht umgangen werden können [Eck13].

Die, in der Literatur zu findenden, herkömmlichen Modelle zur Autorisierung und Zugriffskontrolle beziehen sich häufig auf den Zugriffsschutz bei Betriebssystemen für das Dateisystem oder den Arbeitsspeicher. Die meisten dieser Modelle lassen sich allerdings ebenfalls für den Zugriffsschutz auf Anwendungs- oder Netzwerkebene verwenden, weshalb sie im Folgenden kurz erläutert werden.

Die beiden am meisten erforschten Klassen für Zugriffsrechte sind *Discretionary Access Control (DAC)* und *Mandatory Access Control (MAC)*. Bei DAC handelt es sich um ein benutzerbestimmbares Modell zur Rechteverwaltung [Eck13]. Es basiert auf einer Menge an Objekten, die geschützt werden sollen, und einer Menge von Subjekten, die die zu schützenden Objekte nutzen wollen. Durch Zugriffsregeln können Subjekte und Objekte miteinander verknüpft werden. Bei DAC hat jedes Objekt ein Subjekt als Eigentümer, der für die Rechteverwaltung dieses Objekts verantwortlich ist und die Rechte des Objekts individuell, benutzerbestimmt, vergeben kann [LOP04]. DAC ermöglicht eine feingranulare Einstellung von Zugriffsrechten. Die Speicherung dieser erfolgt häufig in sogenannten *Access Control Matrices*. Bei diesen handelt es sich häufig um dreidimensionale Matrizen, bei denen die Subjekte in den Zeilen und die Objekte in den Spalten angeordnet sind. In der dritten Ebene werden diese durch die Angabe der erlaubten oder verbotenen Operationen miteinander verknüpft. Eine Spalte für ein bestimmtes Objekt dieser Matrix wird als *Access Control List (ACL)* bezeichnet [Aus01]; [LOP04].

Durch die feingranulare Einschränkung von Zugriffsrechten kann mit dem DAC-Modell

[5] Authentication vs Authorisation vs Access Control: http://priocept.com/2011/08/30/authentication-vs-authorisation-vs-access-control/ (abgerufen am 08. Juni 2015)

das *Principle Of Least Authority (PolA)*[6] umgesetzt werden, welches fordert, dass einem Subjekt nur die für seine Aufgaben nötigen und darüber hinaus keine Rechte eingeräumt werden. Verwendung findet DAC überlicherweise in Betriebssystemen für die Zugriffskontrolle auf Dateisystem- oder Speicherebene [Aus01].

Bei Mandatory Access Control (MAC) werden im Gegensatz zum DAC-Modell nicht die Besitzverhältnisse an Objekten, sondern der Fluss der Informationen modelliert [LOP04]. Bei MAC handelt es sich um ein systembestimmtes Modell, bei dem a priori Klassifizierungen festgelegt werden und diese anschließend von autorisierten Personen den Subjekten und Objekten zugewiesen werden können [Eck13]. Da das MAC-Modell häufig im Regierungs- und Militärbereich verwendet wird, stellen die Klassifizierungen häufig Sicherheitsstufen dar [Eck13].

Die genannten Modelle DAC und MAC verursachen in Situationen mit vielen Subjekten und Objekten einen hohen Verwaltungsaufwand durch die zu verwaltenden Zuordnungen und die direkte Verknüpfung von Subjekten und Objekten mit Zugriffsrechten. Insbesondere in dynamischen, sich häufig verändernden Umgebungen skalieren diese Modelle daher nicht besonders gut [Aus01]; [LOP04]. Um den Verwaltungsaufwand dieser Modelle in sich häufig verändernden Umgebung zu verringern, können Abstraktionsschichten eingeführt werden, in denen Eigenschaften und Attribute von Subjekten stellvertretend für diese stehen können. Beispielsweise kann eine rollenbasierte Zugriffskontrolle (Role Based Access Control, RBAC) verwendet werden [RP12]. In RBAC werden Rollen definiert, die organisatorischen Funktionen beziehungsweise die von den Subjekten durchzuführenden Aufgaben beschreiben [Eck13]. Statt festzulegen, welches Subjekt eine Aktion ausführen darf, wird festgelegt, welche Aktionen ausgeführt werden müssen [LOP04]. Subjekte können anschließend beliebig vielen Rollen zugewiesen werden. Die Zugriffsrechte oder -regeln werden in RBAC nicht den Subjekten direkt, sondern den Rollen zugewiesen. So können Subjekten in Szenarien, in denen sich Zuständigkeiten häufig ändern, jederzeit neuen Rollen zugeordnet werden, ohne dafür Zugriffsrechte neu vergeben zu müssen [RP12].

Alternativ zu den genannten Modellen können auch Autorisierungsmodelle verwendet werden, die auf Beschreibungssprachen für die Festlegung von Zugriffsregeln beruhen. Diese können ebenfalls mit dynamischen Veränderungen umgehen und bieten in der Regel feingranulare Einstellmöglichkeiten für Zugriffsrechte, wie die Unterstützung von logischen Verknüpfungen oder lokalen Bedingungen zur Auswertungszeit.

Die vorgestellten Modelle DAC, MAC, RBAC und auch die sprachbasierten Modelle werden in der Regel in zentralisierten Systemen mit homogenen Gerätelandschaften und homogenen Kommunikationsmitteln eingesetzt. In diesen ist eine zentrale Einheit für die Verwaltung von Autorisierungsinformationen und deren Durchsetzung verantwortlich [Her+13]. In diesen Umgebungen haben sich die beschriebenen Autorisierungsmodelle bewährt, allerdings führt ihr Einsatz in heterogenen und verteilten Umgebungen, wie den Constrained-Node Networks des Internet of Things, zu Problemen. Die eingeschränkten Geräte verfügen nicht über ausreichend Rechenleistung und Speicher, um Daten wie Subjekte und Zugriffsregeln in Netzen mit potenziell sehr vielen Nutzern zu verwalten [LOP04].

[6] Interpreting Power: The Principle of Least Authority: http://szabo.best.vwh.net/interpretingpower.html

In zentralisierten Systemen muss häufig jede Kommunikation über die zentrale Einheit ablaufen. Eine Ende-zu-Ende-Sicherheit zwischen dem Anfragenden und dem Endgerät ist nicht möglich. Darüber hinaus stellt die zentrale Einheit häufig einen Single Point of Failure dar, da bei ihrem Ausfall keine Zugriffe mehr stattfinden können. Außerdem kann eine Kompromittierung der zentralen Einheit zum Verlust der Vertraulichkeit des gesamten Systems führen.

Das Internet of Things führt zu einem veränderten Interaktionsmuster, für das die vorgestellten Modelle nicht entwickelt wurden. In traditionellen Netzen waren Autorisierungsinformationen in der Regel langlebig und planbar. In den sich häufig dynamisch und spontan verändernden Netzen des Internet of Things müssen allerdings kurzfristig spontane Änderungen möglich sein. Die vorgestellten Modelle sind hierfür zu unflexibel [LOP04]; [RP12]. In dezentralen Netzen ohne zentrale Einheit, können viele dieser Probleme vermieden werden. Da allerdings, wie bereits erwähnt, die Geräte in Constrained Networks zu schwach sind, um selbstständig Zugriffskontrolllisten oder Rollenzuweisungen zu verwalten, bietet sich der Einsatz alternativer Modelle an [Her+13].

2.7.1 Zugriffsausweise

Eine mögliche Alternative für verteilte Systeme kann in der Verwendung von Zugriffsausweisen (engl. Capabilities) liegen. Während die bisher vorgestellten Modelle eine objektbezogene Sicht einnehmen, bei der Objekte im Mittelpunkt der Zugriffsregeln stehen, verfolgt das auf Zugriffsausweisen basierende Modell eine subjektbezogene Sicht [CO11]. Bei dieser wird die Verantwortung zum Vorzeigen einer gültigen Autorisierung dem Subjekt der Autorisierung aufgetragen [Eck13].

Zugriffsausweise sind kommunizierbare Datenstrukturen, die Informationen enthalten, auf welche Objekte das Subjekt auf welche Art zugreifen darf. Anders als bei den zuvor betrachteten Modellen, führt in diesem Modell nicht eine zentrale Einheit auf Grundlage der ihr vorliegenden Informationen, wie beispielsweise ACLs, eine Zugriffskontrolle durch. Stattdessen legt das Subjekt diese Informationen zur Zugriffszeit in Form eines Zugriffsausweises vor. Das Objekt kann die Zugriffskontrolle anschließend ausschließlich basierend auf den Informationen im Zugriffsausweis durchsetzen [Eck13]. Hierdurch kann eine Entkoppelung der Autorisierung von der Zugriffskontrolle erfolgen, sodass die Zugriffskontrolle dezentral durchgeführt werden kann. Diese Trennung ist insbesondere für verteilte Systeme und Constrained-Node Networks nützlich, da für die Auswertung der Zugriffsausweise und die Durchsetzung der enthaltenen Informationen nicht so viel Rechenleistung und Speicher notwendig ist, wie für das Verwalten von Zugriffsregeln und der Durchführung der Autorisierung [Eck13].

Zugriffsausweise werden in der Literatur in verschiedenen Zusammenhängen auch als Capabilities, Token oder Tickets bezeichnet. Die Analogie von Tickets ermöglicht eine anschauliche Beschreibung für die Verwendung von Zugriffsausweisen. Tickets werden häufig zur Einlasskontrolle bei Veranstaltungen wie Konzerten, verwendet. Bei diesen entspricht der Konzertbesucher dem Subjekt der Autorisierung, welches das Ticket (auf Papier oder elektronisch) zum Konzert mitbringt und am Eingang vorzeigt. Das Personal am Eingang setzt die Zugriffskontrolle allein basierend auf den Informationen im Ticket durch.

Bereits 1966 wurden Zugriffsausweise in [DV66] im Zusammenhang mit dem Speicherschutz in Betriebssystemen diskutiert. Das Konzept wird seitdem allerdings auch in anderen Zusammenhängen betrachtet. Ein System, welches Tickets als Mittel zur Authentifizierung und Zugriffskontrolle verwendet ist Kerberos[7], bei dem ebenfalls die Authentifizierung und Autorisierung dezentral und getrennt von der Durchsetzung der Zugriffskontrolle stattfindet.

Da Zugriffsausweise prinzipiell beliebige Informationen enthalten können, kann eine feingranulare Einstellung und Einschränkung von Zugriffsrechten erfolgen und das Principle Of Least Authority eingehalten werden [RP12]. Darüber hinaus sind Zugriffsausweise deutlich flexibler als die anderen Modelle, da jederzeit dezentral neue Zugriffsausweise ausgestellt werden können [Eck13].

Um die Integrität und Authentizität der im Zugriffsausweis enthaltenen Informationen sicherzustellen, müssen diese gegen Manipulation geschützt werden. Manipulationen können verhindert werden, indem die Informationen verschlüsselt und somit kryptographisch geschützt werden. Die Integrität kann allerdings auch ohne Verschlüsselung sichergestellt werden, indem beispielsweise ein Message Authentication Code (MAC) verwendet wird, der eine Manipulation zwar nicht verhindert, diese aber erkennbar macht. Auch in der Analogie der Konzerttickets werden Maßnahmen ergriffen, um Fälschungen und Manipulationen der Tickets zu verhindern. Häufig werden optische und seit einigen Jahren auch digitale Sicherheitsmerkmale, wie RFID-Chips, auf Eintrittstickets verwendet, die das Fälschen erschweren sollen.

Eine Schwierigkeit bei der Verwendung von Zugriffsausweisen besteht in der Änderung von Zugriffsrechten, da hierzu bereits ausgestellte Zugriffsausweise auf irgendeine Weise ungültig gemacht werden müssen. Das einfache Zurückrufen dieser ist nicht praktikabel, da Zugriffsausweise im Falle von digitalen Systemen in der Regel aus Datenstrukturen bestehen, die beliebig kopiert werden können [Eck13]. Daher kommt statt dem Zurückrufen eher ein Widerrufen der Zugriffsausweise in Frage. Zum Widerrufen gibt es im wesentlichen zwei Möglichkeiten. Zum einen können Zugriffsausweise ein Ablaufdatum oder einen Gültigkeitszeitraum enthalten und somit ein implizites Widerrufen ermöglichen. Diese Methode ist allerdings eher weniger für Systeme, in denen Zugriffsregeln dynamisch und spontan geändert werden, geeignet, da die Zugriffsausweise trotz Änderung der Zugriffsregeln noch bis zum Ablauf ihrer Gültigkeit verwendet werden können [Eck13]. Dieser Ansatz wird beispielsweise von dem System Kerberos genutzt. Die Zeit, in der ein Zugriffsausweis, welcher durch die Änderung einer Zugriffsregel eigentlich ungültig ist, noch verwendet werden kann, wird als *Revocation Window* bezeichnet [MJ00]. In Kerberos beträgt die Gültigkeit von Tickets und damit auch das maximale Revocation Window in der Standardeinstellung 8 Stunden [KS07, S.20].

Alternativ können Zugriffsausweise explizit widerrufen werden. Hierfür verwaltet die Einheit, die Zugriffsausweise ausstellt, eine Liste widerrufener Zugriffsausweise (Revocation List) und macht diese den Objekten der Autorisierung, welche die Zugriffskontrolle durchsetzen müssen, bekannt [Eck13]. Allerdings kann dieser Ansatz wiederum zu Problemen in dynamischen unzuverlässigen Netzen führen, in denen die Teilnehmer nicht jederzeit miteinander kommunizieren können. Die Objekte können die Widerrufung eines Zugriffsausweises nicht durchsetzen, wenn sie die Liste der widerrufenen Zugriffsausweise nicht erhalten können. Auch hierbei kann ein Revocation Window

[7] Kerberos: The Network Authentication Protocol: http://web.mit.edu/kerberos/

entstehen, welches maximal so lang ist wie der Zeitraum in dem eine Kommunikation und damit die Zustellung der Widerrufung nicht möglich ist [Eck13].

2.8 Delegated CoAP Authentication and Authorization Framework (DCAF)

Ein System, welches Zugriffsausweise zur Authentifizierung und Autorisierung verwendet und darüber hinaus explizit auf die Bedürfnisse eingeschränkter Geräte des Internet of Things zugeschnitten ist, ist das *Delegated CoAP Authentication and Authorization Framework (DCAF)* [GBB15a]. DCAF wird in der *ACE*-Arbeitsgruppe[8] unter dem Dach der *Internet Engineering Task Force (IETF)*[9] mit dem Ziel entwickelt eine dezentrale Authentifizierung und Autorisierung in Constrained-Node Networks zu ermöglichen.

DCAF wird derzeit als Internet-Draft entwickelt und gilt somit als nicht fertiggestelltes Dokument (*Work in progress*), bei dem zwischen den Versionen noch grundlegende Änderungen möglich sind. Daher bezieht sich die folgende Beschreibung und die Verwendung von DCAF in dieser Arbeit immer explizit auf eine bestimmte Version. In dieser Arbeit werden die Bezeichnungen und der Protokollablauf der zu diesem Zeitpunkt aktuellen DCAF-Version `draft-gerdes-ace-dcaf-authorize-02` vom 09.März 2015 aus [GBB15a] verwendet.

Die Hauptziele von DCAF sind zum einen das Aufbauen einer sicheren DTLS-Verbindung zwischen eingeschränkten Geräten und zum anderen die sichere Übertragung von Authentifizierungs- und Autorisierungsinformationen in Form von Zugriffsausweisen [GBB15a, S.3]. Diese werden in DCAF Tickets oder Access Token genannt. Sichere Kommunikation wird in DCAF über DTLS erreicht. Hierbei wird ausschließlich symmetrische Verschlüsselung mit Pre-Shared-Keys verwendet. Das Problem der sicheren Kommunikation zwischen eingeschränkten Geräten in Constrained-Node Networks ist in Abbildung 2.4 dargestellt.

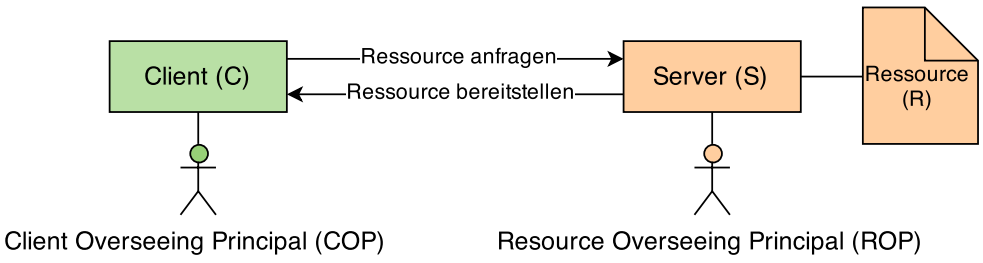

Abbildung 2.4: DCAF Problembeschreibung nach [GBB15b].

Der *Client (C)* auf der linken Seite der Abbildung ist im Besitz des *Client Overseeing Principals (COP)* und gehört zu dessen Sicherheitsdomäne. Der *Server (S)* der anderen Seite gehört zur Sicherheitsdomäne des *Resource Overseeing Principals (ROP)*. Der Server verwaltet *Ressourcen (R)*, bei denen es sich in Sensornetzen beispielsweise um

[8] https://datatracker.ietf.org/wg/ace/ (abgerufen am 08. Juni 2015)
[9] https://www.ietf.org/ (abgerufen am 08. Juni 2015)

anfallende Sensordaten handeln kann. Sowohl beim Client als auch beim Server kann es sich in DCAF um eingeschränkte Geräte handeln. In dieser Arbeit wird ausschließlich der Fall betrachtet, in dem beide Akteure hinsichtlich ihrer Leistungsfähigkeit beschränkt sind.

Ziel des Clients ist es, über eine sichere Ende-zu-Ende verschlüsselte Verbindung auf eine Ressource des Servers zuzugreifen oder eine andere Aktion auf dieser auszuführen. Da der Client und der Server sich zuvor nicht kennen und, wie beschrieben, zu unterschiedlichen Sicherheitsdomänen gehören, haben sie a priori keine Sicherheitsbeziehung zueinander und damit zunächst keine Möglichkeit autonom eine sichere DTLS Verbindung zueinander aufzubauen. Durch die Eingeschränktheit der Akteure können diese die für die sichere Verbindung nötigen Parameter nicht selbst aushandeln. Darüber hinaus kann der Server aufgrund seiner eingeschränkten Hardware die für die Authentifizierung und Autorisierung verschiedener Clients nötigen Informationen, wie etwa Schlüssel, nicht selbst verwalten [GBB15a].

In DCAF wird daher in den Sicherheitsdomänen jeweils eine weitere Schicht in Form eines weniger eingeschränkten Autorisierungsmanagers hinzugefügt, an den die eingeschränkten Geräte die Authentifizierung und Autorisierung, die sie selbst überfordern würden, delegieren können. In der Sicherheitsdomäne des COPs handelt es sich, wie in Abbildung 2.5 dargestellt, um den *Client Authorization Manager (CAM)* und in der Sicherheitsdomäne des ROPs um den *Server Authorization Manager (SAM).*

Bei den weniger eingeschränkten CAM und SAM kann es sich konkret beispielsweise um vollwertige Rechner oder Server handeln. Aber auch die Verwendung von nicht ganz so leistungsfähiger Hardware, wie etwa Smartphones, ist für Autorisierungsmanager möglich [SS15].

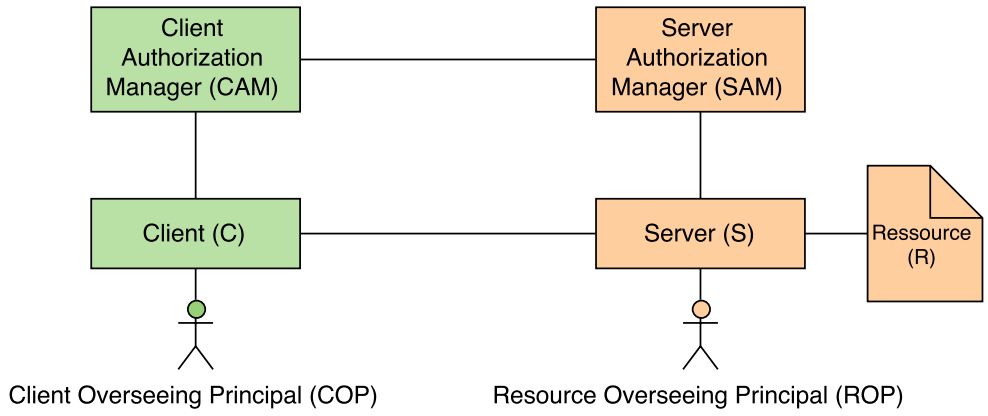

Abbildung 2.5: DCAF-Architektur mit den eingeschränkten Akteuren und den Autorisierungsmanagern nach [GBB15b].

Um eine sichere Verbindung zum Server aufbauen zu können, benötigt der Client in DCAF zunächst ein Ticket. Das Ausstellen des Tickets wird vom Server an seinen Autorisierungsmanager delegiert, wobei dieser stellvertretend für den Server durch das Ausstellen des Tickets eine Authentifizierung und Autorisierung durchführt. Da der Client aufgrund seiner Eingeschränktheit nicht in jedem Fall mit dem Autorisierungsmanager des Servers direkt kommunizieren kann, delegiert der Client das Anfordern

eines Ticketes wiederum an seinen Autorisierungsmanager. Die weniger eingeschränkten Autorisierungsmanager können miteinander beispielsweise über das Internet kommunizieren.

DCAF sieht vor, dass die Autorisierungsmanager CAM und SAM jeweils Zugriffsregeln verwalten, um eine Autorisierung durchführen zu können [GBB15a, S.5]. Diese müssen wiederum von Client Overseeing Principal (COP) beziehungsweise Resource Overseeing Principal (ROP) konfiguriert werden. CAM verwendet die Zugriffsregeln, um den Client zu autorisieren und zu überprüfen, ob der Client die in der delegierten Ticketanfrage genannten Ressourcen abfragen darf. Auf SAM legen die Zugriffsregeln die Rechte für den Zugriff auf die Ressourcen des Server fest und werden beim Ausstellen von Tickets ausgewertet.

Das ausgestellte Ticket kann anschließend vom Client für den Aufbau einer sicheren Verbindung zum Server verwendet werden und dem Server hierbei bekannt gemacht werden. Anhand der im Ticket enthaltenen Informationen kann der Server zum einen ebenfalls die sichere Verbindung zum Client aufbauen und zum anderen die Zugriffskontrolle für den Zugriff auf die Ressourcen durchsetzen. Eine Autorisierung muss vom Server nicht durchgeführt werden, weil diese bereits durch das Ausstellung des Tickets von SAM durchgeführt wurde.

Durch den beschriebenen Ablauf ermöglicht DCAF eine dezentrale Architektur, bei der die Autorisierung und die Zugriffskontrolle getrennt voneinander auf unterschiedlichen Akteuren und zu unterschiedlichen Zeitpunkten stattfinden können. Sobald der Client im Besitz eines Tickets ist, kann dieser mit dem eingeschränkten Server, ohne weitere Hilfe von CAM, sicher kommunizieren. Der Server muss zur Zugriffszeit ebenfalls nicht mit SAM kommunizieren, sondern kann die Zugriffskontrolle autonom, allein basierend auf den im Ticket enthaltenen Informationen, durchsetzen. Daher eignet sich DCAF insbesondere für den Einsatz in Constrained-Node Networks, in denen sich Netzstrukturen häufig verändern und eingeschränkt Geräte nicht zu jederzeit mit Geräten des Internets kommunizieren können.

DCAF fordert, dass zwischen einigen Akteuren bereits zu Beginn des Ablaufs Sicherheits- und Vertrauensbeziehungen bestehen, legt allerdings nicht fest, wie der Austausch der dafür nötigen Parameter abläuft [GBB15a, S.25]. Sowohl zwischen dem Client und CAM als auch zwischen dem Server und SAM müssen Sicherheitsbeziehungen in Form von ausgetauschten Schlüsseln bestehen. Diese werden im Folgenden mit $K(CAM, C)$ für den zwischen dem Client und CAM ausgetauschten Schlüssel und mit $K(SAM, S)$ für den zwischen dem Server und SAM verwendeten Schlüssel bezeichnet. In beiden Sicherheitsdomänen werden die Schlüssel für den Aufbau der sicheren Verbindung zwischen eingeschränktem Gerät und Autorisierungsmanager verwendet. Der Schlüssel $K(SAM, S)$ wird darüber hinaus vom Server und von SAM verwendet, um die Authentizität und Integrität des Tickets sicherzustellen. Da auch zwischen den Autorisierungsmanagern, die sich in unterschiedlichen Sicherheitsdomänen befinden, eine Kommunikation stattfinden soll, muss auch zwischen diesen Akteuren eine Sicherheitsbeziehung bestehen, um eine gegenseitige Authentifizierung und den Aufbau einer sicheren Verbindung zwischen ihnen zu ermöglichen.

Da die Autorisierungsmanager sich in unterschiedlichen Domänen befinden und der Client nicht direkt, sondern ausschließlich über seinen Autorisierungsmanager, mit SAM kommuniziert, hat SAM keine Möglichkeit einer Authentifizierung und Autorisierung

des Clients, sondern kann diese nur auf Grundlage des CAMs durchführen. SAM muss daher insoweit CAM vertrauen, als dass dieser ausgestellte Tickets nur an autorisierte Clients weiterleitet.

Auf der Ebene der Kommunikationsprotokolle legt DCAF fest, dass zwischen den Akteuren auf der Transportschicht DTLS mit Pre-Shared-Keys und symmetrischer Verschlüsselung und als Protokoll der Anwendungsschicht CoAP verwendet wird. Zwischen den weniger eingeschränkten Autorisierungsmanagern kann darüber hinaus auch die Kombination aus HTTP und TLS mit gegenseitiger Authentifizierung über Zertifikate verwendet werden.

DCAF wurde als Framework entwickelt, dass von Anwendungen genutzt werden kann, die eine dezentrale Authentifizierung und Autorisierung in Constrained-Node Networks benötigen. DCAF legt hierbei die Architektur und die zu verwendenden Kommunikationsprotokolle fest. Darüber hinaus beschreibt DCAF die zur sicheren Übertragung des Tickets nötigen Nachrichten und den zum Schlüsselaustausch zwischen den eingeschränkten Akteuren verwendeten Mechanismus. Allerdings nimmt DCAF keine Definition der auf den Autorisierungsmanagern zu verwendenden Modelle zur Authentifizierung und Autorisierung vor. Diese müssen daher auf Anwendungsebene für das jeweilige Einsatzszenario ausgewählt oder entworfen werden.

2.8.1 Protokollablauf

Der, in Abschnitt 2.8 bereits kurz umrissene, Protokollablauf von DCAF soll im Folgenden vertiefend beschrieben werden.

Als Datenformat für die Nachrichten und die ausgestellten Tickets wird in DCAF die speziell auf die Bedürfnisse eingeschränkter Geräte zugeschnittene *Concise Binary Object Representation (CBOR)* [RFC7049] verwendet. Diese stellt Datenstrukturen ähnlich der häufig im Internet für den Datenaustausch zwischen Browser und Webservern genutzten *JavaScript Object Notation (JSON)* [RFC7159] bereit. CBOR ist im Gegensatz zu JSON ein binäres Datenformat, welches eine effiziente und kompakte Kodierung von Daten ermöglicht.

Den Ablauf zur delegierten Authentifizierung und Autorisierung und die dafür benötigten Nachrichten zeigt Abbildung 2.6. Der Client startet den Protokollablauf, indem er im ersten Schritt eine *Unauthorized Resource Request Message* an den Server sendet. Da die Nachricht keine Autorisierungsinformationen enthält, wird sie vom Server im zweiten Schritt zunächst abgelehnt. Allerdings sendet dieser hierbei in einer *SAM Information Message* Informationen über den für ihn verantwortlichen SAM mit. Der Client kann diese Informationen nutzen, um sich von SAM ein Ticket ausstellen zu lassen. Da der Client nicht selbst mit SAM kommunizieren kann, sendet er im dritten Schritt einen *Access Request* an den für ihn verantwortlichen CAM. Dieser authentifiziert und autorisiert den Client anhand der gespeicherten Zugriffsregeln und sendet die in der Anfrage genannten Informationen im vierten Schritt als *Ticket Request Message* an SAM. Dieser führt wiederum eine Authentifizierung des CAMs durch und führt, anhand der in der Ticketanfrage enthaltenen Informationen und den auf SAM gespeicherten Zugriffsregeln, eine Autorisierung durch. Im Erfolgsfall erstellt SAM ein Ticket und sendet dieses im fünften Schritt in einer *Ticket Grant Message* zurück an CAM. Dieser leitet das erhaltene Ticket wiederum im sechsten Schritt in einer *Ticket Transfer*

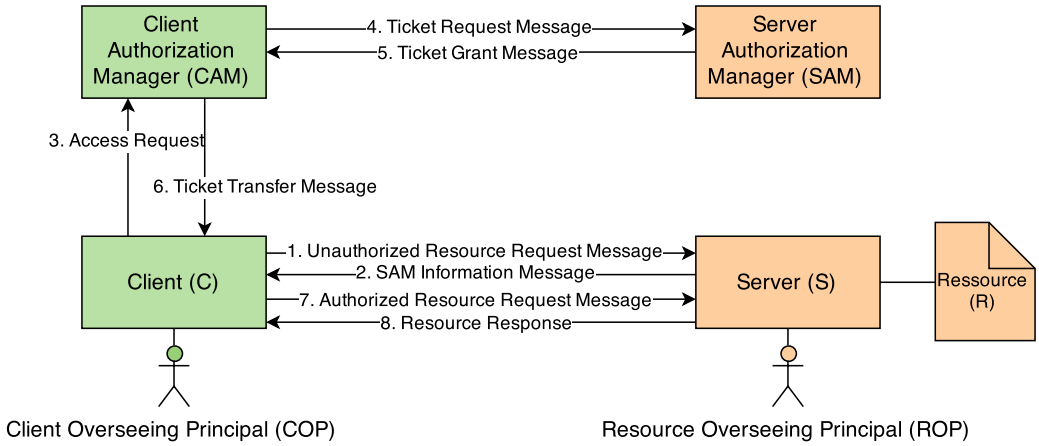

Abbildung 2.6: DCAF-Protokollablauf

Message an den Client weiter.

Der Client hat mit dem Ticket nun alle Informationen, die zum Aufbauen einer sicheren Verbindung zum Server nötig sind. Daher kann der Client die im ersten Schritt durchgeführte Anfrage nun als *Authorized Resource Request Message* wiederholen. Während des Verbindungsaufbaus macht der Client dem Server das Ticket bekannt, sodass dieser auf der Grundlage der darin enthaltenen Informationen eine Zugriffskontrolle durchführen kann. Falls diese positiv ausfällt, sendet der Server den Inhalt der angefragten Ressource im achten und letzten Schritt als Antwort an den Client.

2.8.2 Tickets

Die von SAM ausgestellten Tickets enthalten sowohl Autorisierungsinformationen, die vom Server während der Zugriffskontrolle durchgesetzt werden müssen, als auch Informationen, die von den eingeschränkten Akteuren Client und Server verwendet werden können, um sich gegenseitig zu authentifizieren und eine sichere Verbindung aufbauen zu können.

Tickets bestehen in DCAF aus zwei Teilen: Dem *Ticket-Face* und den *Client Information (CI)*. Das Face ist für den Server gedacht und enthält Autorisierungsinformationen, mit denen der Server die Zugriffskontrolle durchführen kann. Außerdem enthält das Ticket-Face Informationen, aus denen der Server den für die sichere Verbindung zum Client nötigen DTLS-PSK ableiten und die Authentizität und Integrität der Informationen im Ticker-Face sicherstellen kann. Hierfür werden in DCAF zwei Mechanismen vorgeschlagen, von denen einer bei der jeweiligen Umsetzung ausgewählt werden kann. Diese Mechanismen zum Schlüsselaustausch werden in Abschnitt 4.5 ausführlich beschrieben. Darüber hinaus enthält das Ticket-Face einen aktuellen Zeitstempel und optional eine Gültigkeitsdauer.

Die im Ticket enthaltenen Client Informationen sind, wie der Name es vermuten lässt, für den Client gedacht. Diese enthalten insbesondere einen Verifier, der vom Client als DTLS-PSK für die Verbindung zum Server genutzt wird. Da der Verifier in jedem Fall unverschlüsselt im Ticket enthalten ist, darf dieser nur über gesicherte Verbindungen

übertragen werden. Insbesondere darf der Verifier vom Client nicht mit an den Server gesendet werden. Durch den erfolgreichen Aufbau der DTLS-Verbindung können Client und Server sich gegenseitig authentifizieren, da sie sich gegenseitig nachweisen können, dass sie im Besitz des für die Verbindung nötigen Schlüssels sind.

In Unterabschnitt 2.7.1 wurde beschrieben, dass bei der Verwendung von Zugriffsausweisen zur Autorisierung die Zurücknahme von Rechten, insbesondere in Netzen, in denen die Akteure nicht jederzeit miteinander kommunizieren können, zu Problemen führen kann. In DCAF kann eine implizite oder explizite Widerrufung von Tickets verwendet werden. Das implizite Widerrufen wird durch die im Ticket-Face optional enthaltene Gültigkeitsdauer ermöglicht. Alternativ schlägt DCAF die Verwendung einer *Revocation Message* vor, die für ein explizites Widerrufen von Tickets genutzt werden kann, indem sie von SAM an den betroffenen Server geschickt wird. Im DCAF-Protokollablauf ist dies, neben dem in DCAF nicht festgelegten initialen Schlüsselaustausch, die einzige Stelle, an der eine direkte Kommunikation zwischen dem Server und seinem Autorisierungsmanager stattfindet. Je nach Einsatzszenario kann einer der Mechanismen oder eine Kombination beider zum Widerrufen von Tickets verwendet werden.

2.8.3 Vergleich zu bestehenden Systemen

Wie bereits in Unterabschnitt 2.7.1 beschrieben, existieren einige zu DCAF vergleichbare Systeme zur dezentralen Authentifizierung und Autorisierung mit Hilfe von Zugriffsausweisen. Im Folgenden wird ein kurzer Überblick über den Forschungsstand zu diesem Thema gegeben.

In Unterabschnitt 2.7.1 wurde bereits der verteilte Authentifizierungsdienst Kerberos als ein solches System genannt. Kerberos wird bereits seit 1983[10] entwickelt und wird in der Regel als Netzwerkauthentifizierungsprotokoll eingesetzt und erlaubt eine Authentifizierung zwischen einem Client und einem Service. Hierzu werden ebenfalls Tickets verwendet, die von einem zentralen *Ticket Granting Server (TGS)* ausgestellt werden. Kerberos ist sehr flexibel, wodurch es für einen breiten Anwendungsbereich einsetzbar ist. Allerdings wurde es nicht explizit für eingeschränkte Geräte, wie die des Internet of Things und die darin verwendeten kurzlebigen und spontanen Netzstrukturen, entwickelt. In [Har14a] wird die Eignung von Kerberos für Anwendungen des Internet of Things evaluiert.

Während Kerberos in der Regel den Netzzugang schützt, können Systeme zur delegierten Authentifizierung und Autorisierung auch auf Anwendungsebene verwendet werden. Ein solches System ist beispielsweise das *OAuth 2.0 Authorization Framework*, welches einem Dritten stellvertretend für einen Client den Zugriff auf HTTP-Dienste ermöglicht [RFC6749]. Die Architekturen und Begrifflichkeiten von DCAF und OAuth ähneln sich sehr stark, da viele Mechanismen in DCAF von OAuth inspiriert sind. OAuth wurde allerdings für den Einsatz über HTTP entwickelt und eignet sich daher nicht ohne Änderungen für den Einsatz in Constrained-Node Networks, in denen über CoAP kommuniziert werden soll. In [Tsc15] werden Änderungen und Erweiterungen diskutiert, um den Protokollablauf von OAuth über CoAP statt über HTTP abzuwickeln und damit eine Integration in Anwendungen des Internet of Things zu

[10] Kerberos: http://www.kerberos.org/docs/ (abgerufen am 08.06.2015)

ermöglichen.

Es existieren allerdings neben DCAF auch bereits einige Systeme, die explizit für die Verwendung in Constrained-Node Networks entwickelt wurden. Das System *Capability Based Access Control (CapBAC)* verwendet ebenfalls Zugriffsausweise, welche als Capabilities bezeichnet werden [GPR13]. Diese können in CapBAC nicht nur für den Zugriff auf Ressourcen verwendet werden, sondern durch ihre Besitzer auch an Dritte im Ganzen oder als Teilmenge weiter delegiert werden, sodass Capability-Ketten entstehen können. Darüber hinaus werden die Capabilities in CapBAC prinzipiell nicht, wie in DCAF, automatisiert durch einen Akteur ausgestellt, sondern ausschließlich durch Delegation schon vorhandener Capabilites, wobei es am Ende der Kette eine einmalig manuell ausgestellte Root-Capability geben muss. Als Datenstruktur für die Capabilities werden signierte XML-Datenstrukturen verwendet, die auf den Autorisierungssprachen SAML und XACML beruhen. Diese bieten zwar eine hohe Flexibilität für die Rechtevergabe, können aber durch ihre Beschreibung in XML zu großen Capabilities mit nicht unerheblichem Oberhead führen. Ein weiterer Unterschied zu DCAF besteht in der Auswertung der Zugriffsausweise. In DCAF können die Autorisierung und die Zugriffskontrolle zu unterschiedlichen Zeitpunkten stattfinden. Zur Zugriffszeit ist lediglich die Zugriffskontrolle nötig. In CapBAC wird zur Zugriffszeit die Capability zur Autorisierung an einen zentralen *Policy Decision Point (PDP)* weitergeleitet. Erst im Anschluss kann die Zugriffskontrolle ohne weitere Hilfe durchgesetzt werden. Hierdurch ist in CapBAC kein dezentraler Zugriff möglich, bei dem das eingeschränkte Gerät, welches Ressourcen verwaltet, nicht mit dem PDP kommunizieren kann [GPR13]. Darüber hinaus ist CapBAC nicht für Kommunikation über Sicherheitsdomänen hinweg ausgelegt [GBB14].

Um die Notwendigkeit eines zentralen PDP in CapBAC zu überwinden, wurde *Distributed Capability-based Access Control (DCapBaC)* entworfen [Her+13]. In DCapBAC fällt der zentrale PDP weg und jedes Endgerät wird selbst zum PDP, der die Autorisierung anhand der Capability selbstständig durchführt. Außerdem wird mit JSON ein Datenformat für die Capabilities verwendet, welches nicht ganz so viel Overhead hat wie das in CapBAC verwendete XML. In DCapBAC wird, anders als in DCAF, nicht die Verwendung einer Transportverschlüsselung, wie DTLS, vorgeschrieben. Stattdessen wird die Capability selbst durch kryptographische Methoden geschützt.

Kapitel 3

Anforderungen

Die in Kapitel 2 eingeführten Mechanismen und Protokolle sollen als Grundlage des zu entwickelnden Systems verwendet werden. Aufbauend auf dem, in Abbildung 3.1 dargestellten, Protokollstapel soll ein System für eine sichere Kommunikation und zur Übertragung von Autorisierungsinformationen in Constrained-Node Networks für ein konkretes Szenario entwickelt werden.

Im Folgenden wird zunächst das umzusetzende Nutzungsszenario *Bananen für München* vorgestellt. Anschließend werden die allgemeinen Anforderungen an das gesamte System und im Anschluss die Anforderungen an die einzelnen Akteure definiert. Bei diesen liegt der Schwerpunkt auf dem Server Authorization Manager (SAM).

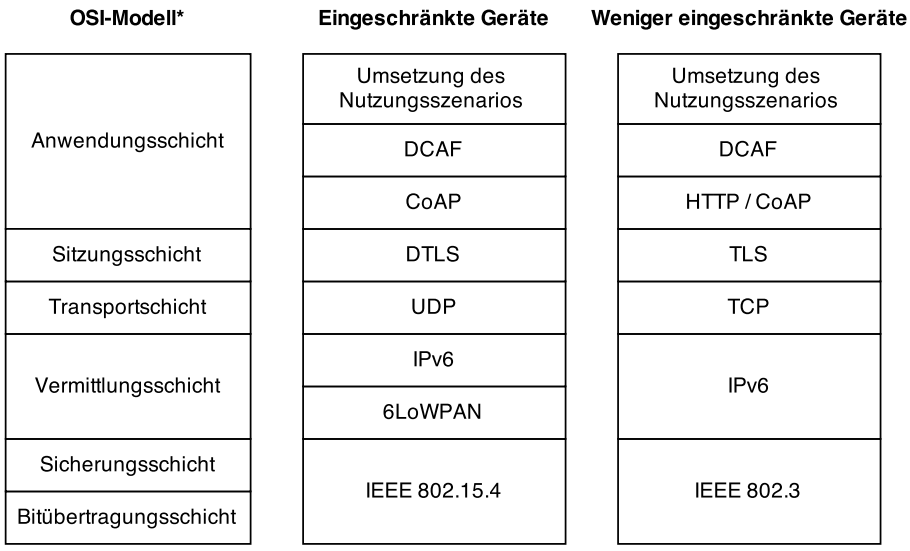

Abbildung 3.1: Verwendeter Netzwerk- und Protokollstack der eingeschränkten und weniger eingeschränkten Geräte. (*Die Präsentationsschicht des OSI-Modells ist nicht enthalten, da diese im Szenario keine Anwendung findet)

3.1 Nutzungsszenario: Bananen für München

Das gewählte Nutzungsszenario ist an den Anwendungsfall *Bananen für München* aus [Sei+15] angelehnt. In diesem Anwendungsfall baut ein Unternehmen (im Folgenden Unternehmen `A`) in Costa Rica Bananen für den deutschen Markt an und lässt diese von einem Transportunternehmen (im Folgenden Unternehmen `B`) in einem ISO-Container[1] nach Rotterdam verschiffen. Anschließend werden die Bananen mittels LKW des Unternehmens `A` aus dem Hafen in ein Reifelager transportiert und von dort, nach entsprechender Reifezeit, an die einzelnen Läden des Unternehmens `A` verteilt.

Bananen sind Beerenfrüchte und sind, wie die meisten Obstsorten, lebende Organe, die auch nach der Trennung von der Mutterpflanze Respirationsprozessen (Atmung) unterliegen [KS]. Dabei produzieren Bananen unter anderem Kohlenstoffdioxid (CO_2), Wasserdampf und Wärme, die den Reifeprozess beschleunigen können oder zur Bildung von Gärung und Fäulnis führen können. Daher ist für den Transport von Bananen eine aufwändige Ladungspflege notwendig [KS]. Zur Bestimmung der notwendigen Ladungspflege werden Transportgüter in Lagerklima-Konditionen klassifiziert. Bananen gehören zu der Klasse mit den anspruchsvollsten Konditionen LK VII und erfordern daher während des gesamten Transports bestimmte Temperatur-, Feuchte- und Lüftungs-Konditionen [KS]. Die Ladungspflege muss darauf ausgelegt sein die Respirationsprozesse so zu steuern, dass am Zielort der gewünschte Reifegrad erreicht ist. Bananen werden daher zumeist in Kühlcontainern mit steuerbarer Frischluftzufuhr und Klimaanlage transportiert [KS].

Der gängige Ablauf sieht vor, dass die einzuhaltenden Richtwerte für Temperatur, Luftfeuchtigkeit und Gaskonzentration schriftlich vor Ladebeginn vom Versender (`A`) an das Transportunternehmen (`B`) übermittelt werden und während der gesamten Transportkette einzuhalten sind [Jed+13]. Dafür werden die übermittelten Daten vor Transportbeginn fest in die Klima- und Lüftungsautomatik des Containers einprogrammiert und während des Transports mit den tatsächlich gemessenen Werten verglichen. Bei Abweichungen vom Sollwert reguliert der Kühlcontainer fortlaufend die Temperatur und die Luftzufuhr. Eine Änderung dieser Richtwerte während des Transports ist in der Regeln nicht möglich [KS].

Bananen werden in perforierten Wellpappschachteln in zwei Reihen und zwei Lagen übereinander verpackt. In einer Schachtel befinden sich etwa 70 bis 125 Bananen [KS]. Durch Variationen bei der Verpackung können innerhalb der Schachteln unterschiedlich große Luftkammern entstehen, die sich während des Transports durch die Respirationsprozesse der Bananen mit warmer CO_2-reicher Luft füllen und zu sogenannten Hot Spots werden [Jed+13]. Bananen in der Nähe dieser Hot Spots reifen schneller als die übrigen Bananen und produzieren hierdurch ebenfalls mehr Wärme und CO_2. Das beschriebene, gängige Verfahren mit Kühlcontainern, die an zentraler Stelle im Container Sensorwerte messen, bietet keine Möglichkeiten, solche Hot Spots zu erkennen und korrigierend einzugreifen, indem beispielsweise die Luftzufuhr erhöht wird.

Um auch auf die individuellen Bedingungen innerhalb der einzelnen Schachteln Rücksicht nehmen zu können, können alternativ zum oben genannten Vorgehen, Sensorknoten von Unternehmen `A` direkt neben der Ware innerhalb der Schachteln platziert wer-

[1] ISO 668:2013: http://www.iso.org/iso/catalogue_detail.htm?csnumber=59673 (abgerufen am 08. Juni 2015)

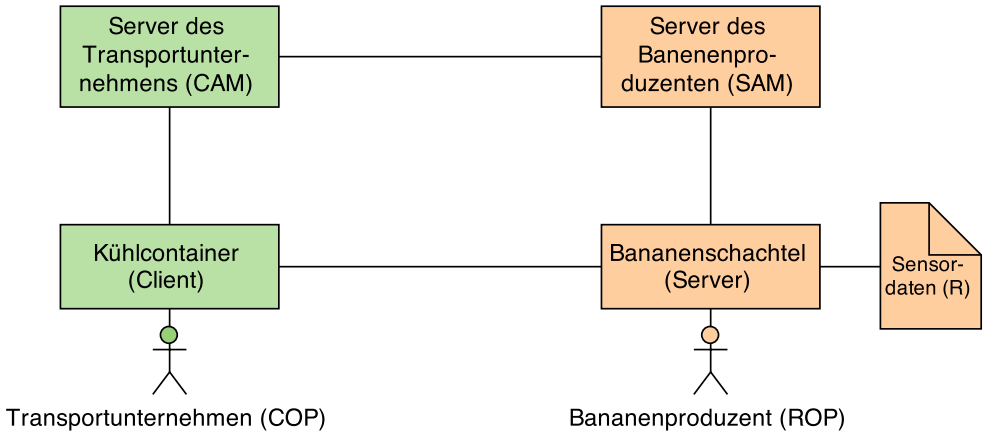

Abbildung 3.2: Nutzungsszenario in der DCAF-Architektur.

den, die fortlaufend die Luftqualität und den Reifegrad messen. Die einzelnen Knoten können zusammen ein Sensornetz bilden und mit den, von Unternehmen `B` eingesetzten, intelligenten Kühlcontainern kommunizieren, um so eine effektivere Regulierung zu erreichen.

Um eine solche Kommunikation zu ermöglichen und abzusichern, wird das, in Abschnitt 2.8 beschriebene, Delegated CoAP Authentication and Authorization Framework (DCAF) verwendet. Das hier skizzierte Nutzungsszenario kann in der Architektur von DCAF wie in Abbildung 3.2 dargestellt aussehen.

Der Container und CAM auf der linken Seite gehören zu Unternehmen `B` und damit in dessen Sicherheitsdomäne und die Bananenschachteln mit den Sensoren und SAM auf der rechten Seite gehören zur Sicherheitsdomäne des Unternehmens A. Der Kühlcontainer beinhaltet ein eingeschränktes Gerät, welches in der Rolle des DCAF-Clients mit den eingeschränkten Sensorknoten in den Bananenschachteln über Sicherheitsdomänen hinweg sicher kommunizieren möchte und die abgefragten Daten nutzen will, um das Klima im Container zu regulieren. Die Sensorknoten in den Bananenschachteln nehmen die Rolle des DCAF-Servers ein, der Ressourcen, in diesem Fall Sensordaten wie Temperatur und Luftfeuchtigkeit, bereitstellt.

Diese Rollenverteilung wird nicht in DCAF festgelegt, sondern wurde für das Szenario gewählt. Alternativ könnten der Kühlcontainer und die Bananenschachtel auch die Rollen tauschen, sodass der Kühlcontainer die Rolle des DCAF-Servers und die Bananenschachtel die des DCAF-Clients einnehmen würde. In diesem Fall könnte die Bananenschachtel die erhobenen Sensordaten selbstständig in regelmäßigen Abständen an den Kühlcontainer senden. DCAF sieht außerdem vor, dass Akteure zusammengefasst werden können. Daher könnten, alternativ zur beschriebenen Architektur, der Autorisierungsmanager und der Client gemeinsam in einem Gerät im Kühlcontainer untergebracht werden.

Da es sich sowohl beim Client als auch beim Server um eingeschränkte Geräte im Sinne der, in Abschnitt 2.2 vorgestellten, Contrained Devices handelt, können diese nicht selbstständig aufwändige Authentifizierungs- und Autorisierungsverfahren durchführen. Da in diesem Szenario aber Daten über Sicherheitsdomänen hinweg zwischen zwei

Unternehmen ausgetauscht werden sollen, ist eine Zugriffskontrolle und die Durchsetzung dieser essenziell wichtig. Hierzu steht in beiden Sicherheitsdomänen jeweils ein weniger eingeschränkter Autorisierungsmanager zur Verfügung, an den Clients und Server Aufgaben im Sinne der DCAF-Spezifikation delegieren können, die sie selbst überfordern würden. Als Autorisierungsmanager können die Unternehmen beispielsweise vollwertige leistungsstarke Computer verwenden, die über das Internet miteinander kommunizieren. Alternativ können aber auch, wie in Abschnitt 2.8 beschrieben, weniger leistungsfähige Geräte, wie etwa Smartphones von Mitarbeitern des jeweiligen Unternehmens, verwendet werden. Für das Szenario in dieser Arbeit wird angenommen, dass es sich bei den Autorisierungsmanagern um Server handelt, die dauerhaft mit dem Internet verbunden sind und sich in den jeweiligen Unternehmen befinden. Alternativ wäre es auch möglich, lokale Autorisierungsmanager zu verwenden, die sich während des Transports mit auf dem Schiff befinden.

Eine Kommunikation der, in Abbildung 3.2 dargestellten, Akteure, kann vereinfacht wie folgt ablaufen: Bei der Auftragserteilung für den Transport tauschen die beteiligten Unternehmen Informationen aus, mit denen sie sich gegenseitig authentifizieren können. Es kann sich hierbei um geheime Schlüssel oder auch Zertifikate handeln. Mitarbeiter der Unternehmen legen anschließend auf ihrem jeweiligen Autorisierungsmanager Zugriffsregeln für das jeweils andere Unternehmen fest. Insbesondere wird auf SAM von `A` festgelegt, auf welche Server und auf welche Ressourcen CAM von `B` — und damit die Clients des Unternehmens `B` — zugreifen dürfen. Kurz vor oder zu Beginn der Verladung erhält der DCAF-Client im Kühlcontainer Informationen über die sich in der Ladung befindlichen Sensoren und Ressourcen und fordert über seinen Autorisierungsmanager von SAM Tickets für die Kommunikation mit den DCAF-Servern in den Bananenschachteln an. Mit den Tickets kann der Kühlcontainer nun sichere Verbindungen mit den Sensorknoten in den Bananenschachteln aufbauen und fortlaufend während des Transports Sensordaten in Form von Ressourcen abfragen und diese für die Regulierung des Klimas verwenden.

Der beschriebene Ablauf kann sowohl für ersten Transportweg über das Meer als auch für den Weitertransport durch LKW an Land verwendet werden. Eine ausführliche Beschreibung und Auswertung des beschriebenen Ablaufs findet sich in Kapitel 6.

3.2 Allgemeine Anforderungen an das System

Das zu entwerfende und umzusetzende System soll die in Abschnitt 2.8 vorgestellte Architektur und die in DCAF vorgesehenen Akteure, Kommunikationsprotokolle und Datenformate nutzen. Die im Folgenden definierten Anforderungen an ein System zur dezentralen Authentifizierung und Autorisierung basieren zum Teil auf den in [Sei+15] und [Ger15a] beschriebenen Problemstellungen, werden hier aber dem Szenario entsprechend konkretisiert.

A1.1 Bei der Kommunikation im skizzierten Nutzungsszenario soll zwischen allen Beteiligten die Vertraulichkeit, Integrität und Authentizität der übertragenen Daten sichergestellt werden. Dies gilt insbesondere auch für die Verbindung zwischen Kühlcontainer und Bananenschachteln. Sowohl der Bananenproduzent als auch das Transportunternehmen wollen die Integrität der gemessenen Sensordaten si-

cherstellen können.

A1.2 Die Knoten in den Bananenschachteln sollen nicht ausschließlich zum Erfassen und Bereitstellen von Sensordaten verwendet werden. Unternehmen `A` will auf diesen zusätzlich Lieferinformationen speichern. `A` will für Sensor- und Lieferinformationen unterschiedliche Zugriffsregeln festlegen können. Es soll daher möglich sein, den Zugriff nur auf bestimmte Ressourcen eines DCAF-Servers zu erlauben.

A1.3 Die eingeschränkten Geräte Client und Server haben während des Transports keinen dauerhaften Zugang zum Internet und insbesondere nicht zu ihren Autorisierungsmanagern. Im Nutzungsszenario ist beispielsweise während des Transports auf See keine Verbindung zu den Autorisierungsmanagern möglich. Der Zugriff vom Client auf den Server muss daher dezentral ohne Intervention eines Mitarbeiters der Unternehmen oder des Autorisierungsmanagers möglich sein.

A1.4 Bei der Kommunikation in Sensornetzen ist darauf zu achten, dass übertragene Pakete möglichst kleine Datenmengen in einem effizienten Format enthalten, um die, in Abschnitt 2.3 beschriebenen, negativen Folgen der Fragmentierung von Datenpaketen zu vermeiden.

A1.5 Die Kontrolle über den Zugriff auf eingeschränkte Server soll jederzeit beim Besitzer des Servers, ROP, liegen. Zugriffsregeln sollen von diesem spontan geändert werden können. Hierfür ist ein Widerrufen der im DCAF-Ablauf ausgestellten Tickets notwendig.

3.3 Server Authorization Manager (SAM)

Im Folgenden sollen die Anforderungen an den zu entwerfenden und zu implementierenden SAM, der zur Sicherheitsdomäne des Unternehmens `A` gehört, definiert werden. Bei SAM handelt es sich um einen Autorisierungsmanager, dessen Hauptaufgaben in der Auswertung von Zugriffsregeln und dem Ausstellen von Tickets liegen. Wie in Abschnitt 1.1 beschrieben soll ein Schwerpunkt dieser Arbeit auf diesem Gerät liegen, daher ist eine ausführliche Analyse der Anforderungen nötig. Eine für alle Teile von SAM gültige Anforderung ist die Wichtigkeit der Nutzungsfreundlichkeit (engl. Usability). Durchschnittliche Nutzer sind in der Regel nicht vertraut mit der Konfiguration und Verwendung sicherheitsrelevanter Systeme. Daher müssen alle Konfigurationen und Schritte auf SAM von jemanden mit wenig Wissen über Kommunikationssicherheit durchgeführt werden können.

Verwaltung

Um Autorisierungsentscheidungen treffen zu können, muss SAM die Übersicht über die im Szenario verwendeten Akteure behalten und hierfür folgende Anforderungen erfüllen:

A2.1 SAM muss Subjekte und Objekte der durchzuführenden Autorisierung verwalten. Darüber hinaus muss SAM diese durch Zugriffsregeln miteinander verknüpfen und ausgestellte Tickets verwalten.

A2.2 Neben der Verwaltung der oben genannten Daten, muss SAM auch eine möglichst nutzungsfreundliche Möglichkeit zur Konfiguration im laufenden Betrieb für den

ROP bieten.

Kommunikation

Als Autorisierungsmanager muss SAM sowohl mit Autorisierungsmanagern der Clients als auch mit Servern, für dessen Verwaltung SAM zuständig ist, kommunizieren. Folgende Anforderungen werden an die Kommunikation gestellt:

A2.3 Zur sicheren Kommunikation müssen sowohl zwischen SAM und den Servern als auch zwischen SAM und den Autorisierungsmanagern der Clients Sicherheitsbeziehungen bestehen.

A2.4 SAM muss Ticketanfragen von CAMs empfangen und in angemessener Zeit verarbeiten und beantworten können.

A2.5 Außerdem soll SAM mit eingeschränkten Servern kommunizieren können, deren Zugriffe von SAM verwaltet werden. Über diesen Kommunikationsweg soll SAM widerrufene Tickets oder neues Schlüsselmaterial mit den Servern austauschen können.

Zugriffsregeln

Um eine Autorisierung durchführen zu können, muss SAM Zugriffsregeln verwalten und auswerten, die den Zugriff auf Ressourcen auf von SAM verwalteten Servern kontrollieren. Die Zugriffsregeln sollen nicht nur den Zugriff im Ganzen kontrollieren, sondern es ermöglichen feingranular zu bestimmen auf welche Ressource mit welcher Aktion zugegriffen werden darf. Folgende Anforderungen sind von SAM im Allgemeinen und für das konkrete Szenario zu erfüllen:

A2.6 Das System soll in der Standardeinstellungen sicher sein, das heißt ohne die Festlegung von Zugriffsregeln und ohne Autorisierung und Ausstellen eines Tickets soll kein Zugriff möglich sein. Bei den festzulegenden Regeln soll es sich um Positiv-Regeln handeln, die festlegen, unter welchen Bedingungen ein Zugriff erfolgen darf. Auf Ressourcen, die nicht von Zugriffsregeln erfasst werden, soll kein Zugriff möglich sein. Allerdings soll ein Zugriff mit korrekten Autorisierungsregeln im Ticket möglichst leicht zu erreichen sein.

A2.7 Es soll sowohl in den Zugriffsregeln als auch in den Tickets ein impliziter und auch ein expliziter Zugriff autorisiert werden können. In manchen Fällen sind eventuell keine feingranularen Zugriffsregeln notwendig, sondern es genügt den Zugriff auf ein Gerät im Ganzen zu erlauben oder zu unterbinden. Außerdem können bei einer impliziten Autorisierung auf dem Server standardmäßig feste Regeln vorgesehen werden. Neben dieser impliziten Autorisierung sollen aber auch explizite, feingranulare Zugriffsregeln festgelegt werden können.

A2.8 Durch die Zugriffsregeln sollen unterschiedliche Zugriffsrechte für eine Ressource für unterschiedliche Subjekte festgelegt werden können. Außerdem will der Gerätebesitzer verschiedene Zugriffsrechte für unterschiedliche Ressourcen auf einem Gerät festlegen. Bezogen auf das Nutzungsszenario könnte dies beispielsweise bedeuten, dass das Transportunternehmen `B` nur auf die Ressourcen der Sensordaten, aber nicht auf Lieferinformationen, zugreifen darf.

A2.9 Auf SAM sollen mehrere Ressourcen und Server in einer Autorisierungsregel zusammengefasst werden können, um nicht für jede Ressource und jeden Server einzelne Regeln festlegen zu müssen. Im Nutzungsszenario können beispielsweise alle Server in den Bananenschachteln mit ihren Ressourcen in einer Zugriffsregel zusammengefasst werden. Hierbei ist darauf zu achten, dass bei der Verwendung von Autorisierungsinformationen im Ticket keine Zugriffsregel über mehrere Server hinweg möglich ist, da Tickets in DCAF immer nur den Zugriff auf einen Server autorisieren.

A2.10 Der Gerätebesitzer will sowohl in den Zugriffsregeln auf SAM als auch in den späteren Tickets temporären Zugriff gewähren, sodass ein Zugriff nur eingeschränkt, bis zu einem bestimmten Zeitpunkt beziehungsweise eine bestimmte Dauer, möglich ist [Ger15a].

Tickets

Zum eigentlichen Zugriff auf Ressourcen sind in DCAF Tickets notwendig. Diese verhalten sich wie Zugriffsausweise in anderen Systemen, die alle Informationen enthalten, um auf die angegebenen Ressourcen zuzugreifen. Daher ist bei der Verwaltung und Übertragung dieser die Vertraulichkeit der im Ticket enthaltenen Informationen zu gewährleisten. Darüber hinaus werden für Tickets im konkreten Szenario folgende Anforderungen gestellt:

A2.11 Nach erfolgreicher Autorisierung anhand der oben definierten Zugriffsregeln muss SAM dem CAM ein Ticket ausstellen, welches den Zugriff auf die angeforderte Ressource erlaubt beziehungsweise dem Server erlaubt eine Zugriffskontrolle anhand der angeforderten Ressource und der im Ticket enthaltenen Informationen durchzuführen und durchzusetzen.

A2.12 Die ausgestellten Tickets müssen fälschungssicher und kryptographisch geschützt sein, sodass sie nur von berechtigten Akteuren verwendet werden und nicht manipuliert werden können.

A2.13 Es dürfen von SAM keine Tickets ausgestellt werden, die gegen die von ROP festgelegten Zugriffsregeln verstoßen. Bei Änderungen an den Regeln müssen auch bereits ausgestellte Tickets überprüft und gegebenenfalls widerrufen werden.

A2.14 SAM muss die Zugriffsregeln, die der Server für die Autorisierung verwenden soll, in das Ticket schreiben. Hierbei muss darauf geachten werden, dass die Informationen im Ticket aufzubereiten beziehungsweise zu reduzieren sind, sodass sie auch von einem eingeschränkten Server verstanden und durchgesetzt werden können.

A2.15 Von SAM ausgestellte Tickets sollen sowohl von SAM selbst als auch direkt durch den ROP widerrufen werden können, um einmal ausgestellten Tickets die Gültigkeit zu entziehen (explizites Widerrufen). Darüber hinaus sollen Tickets ein Ablaufdatum tragen können, um den Zugriff nur für eine bestimmte Zeit zu gewähren (implizites Widerrufen).

A2.16 Die Autorisierungsinformationen im Ticket sollen einen bedingten Zugriff auf Ressourcen ermöglichen. Der Gerätebesitzer will, dass zur Zugriffszeit der Server lokale kontextbezogene Bedingungen überprüft werden und diese in die Zugriffskontrolle mit einfließen. Dadurch soll es etwa möglich sein, den Zugriff nur an

bestimmten Orten oder in bestimmten Zeitintervallen zuzulassen.

3.4 Server (S)

Der Server ist in der DCAF-Architektur für das Bereitstellen von Ressourcen zuständig. Er muss mit Geräten innerhalb seiner eigenen Sicherheitsdomäne und mit Geräten in anderen Sicherheitsdomänen kommunizieren. Im Nutzungsszenario nehmen die einzelnen Sensorknoten in den Bananenschachteln die Rolle des Servers ein. Es handelt es sich hierbei um eingeschränkte Geräte, die über Sensoren die Umgebungsbedingungen in den Bananenschachteln ermitteln und diese Messungen als Ressourcen für die Clients zur Verfügung stellen. Beim Zugriff auf diese Ressourcen muss der Server eine Zugriffskontrolle anhand des vom Client mitgesendeten Tickets durchführen.

Neben der Kommunikation mit den Clients müssen die Server auch mit ihren Autorisierungsmanagern in Kontakt treten können, um Schlüsselmaterial oder Informationen über widerrufene Tickets auszutauschen. Eine Kommunikation mit dem verantwortlichen SAM ist allerdings nicht jederzeit möglich, da der Server als eingeschränktes Gerät nicht über einen ständigen Netzzugang verfügt. Entsprechend dem Nutzungsszenario soll der umzusetzende Server folgende Anforderungen erfüllen:

A3.1 Der Server soll Sensordaten erfassen und diese über geeignete Schnittstellen als Ressourcen zum Zugriff für Clients bereitstellen.

A3.2 Zum Schutz der Vertraulichkeit, Integrität und Authentizität muss der Server sichere Verbindungen mit Clients aushandeln. Hierfür muss der Server über ausreichend Speicher und Rechenleistung verfügen.

A3.3 Vom Client gesendete Anfragen müssen vom Server validiert und verarbeitet werden können.

A3.4 In ihrer Anfrage wird von den Clients ein Ticket mit Autorisierungsinformationen mitgesendet, die der Server während der Zugriffskontrolle überprüfen muss. Es muss geprüft werden, ob das Ticket von einem Autorisierungsmanager ausgestellt wurde, dem der Server vertraut und ob dieser berechtigt ist, für den Server Tickets auszustellen.

A3.5 Der Server muss die im Ticket enthaltenen Informationen nutzen, um eine sichere Verbindung mit dem Client aufzubauen und die entsprechenden Zugriffsregeln durchzusetzen. Es muss sichergestellt werden, dass kein unautorisierter Client Zugriff auf Ressourcen erhalten kann [Ger15a].

A3.6 Außerdem muss der Server überprüfen können, ob die Lebenszeit des vom Client verwendeten Tickets abgelaufen ist oder, ob es bereits widerrufen wurde und damit auch trotz gültiger Autorisierungsregeln nicht mehr vom Server akzeptiert werden darf.

A3.7 Die Durchsetzung der Autorisierung muss der Server autonom, ohne weitere Hilfe des ROPs oder seines Autorisierungsmanagers, ausschließlich basierend auf den im Ticket enthaltenen Informationen, durchführen.

A3.8 Nach erfolgreicher Zugriffskontrolle muss der Server die vom Client angefragte Operation auf die Ressource ausführen und eine Antwort sicher an den Client übertragen [Ger15a]. Hierbei ist die Vertraulichkeit und die Integrität der Ressource sicherzustellen.

A3.9 Neben der Kommunikation mit Clients muss der Server auch mit dem für ihn zuständigen SAM kommunizieren und sichere Verbindungen mit diesem aushandeln können. Über diese Verbindung soll der Server von SAM Informationen zu widerrufenen Tickets entgegen nehmen können. Außerdem soll der Austausch von Schlüsselmaterial über diese Verbindung möglich sein, um die Außer- oder Inbetriebnahme eines Servers zu unterstützen.

3.5 Client (C)

In der Architektur von DCAF ist der Client für den Zugriff auf Server beziehungsweise auf Ressourcen auf diesen zuständig. Für den Zugriff muss sich der Client zunächst ein Ticket über den für ihn zuständigen CAM besorgen. Im Nutzungsszenario entspricht das im Kühlcontainer vorhandene eingeschränkte Gerät dem DCAF-Client, der die Sensordaten in den Bananenschachteln abfragt und basierend auf diesen Daten das Klima innerhalb des Containers reguliert. Auch der Client kann als eingeschränktes Gerät nicht jederzeit mit seinem Autorisierungsmanager kommunizieren und muss Tickets daher unter Umständen bereits längere Zeit vor dem eigentlichen Zugriff anfragen und für den späteren Zugriff aufbewahren. Wie der Server muss auch der Client über ausreichend Rechenleistung und Speicher verfügen, um sichere Verbindungen aufzubauen und hierüber sicher kommunizieren zu können. Um im Sinne des Nutzungsszenarios verwendet zu werden, muss der Client darüber hinaus folgende Anforderungen erfüllen:

A4.1 Für den Zugriff auf Ressourcen auf einem Server wird zunächst ein Ticket benötigt. Dieses wird vom Autorisierungsmanager des Servers ausgestellt. Der Client muss ermitteln können, welcher SAM für eine Ressource und einen Server verantwortlich ist, um seine Ticket-Anfrage an diesen adressieren zu können.

A4.2 Der Client muss eine Sicherheitsbeziehung zu seinem Autorisierungsmanager haben und Anfragen (*Access Requests*) an diesen sicher übertragen können.

A4.3 Der Client muss die Antwort auf seine Anfrage von seinem Autorisierungsmanager empfangen, validieren und verarbeiten können. Der Client muss sich das in der Antwort enthaltene Ticket merken, um es für den Zugriff verwenden zu können.

A4.4 Die Informationen im Ticket müssen vom Client für die Kommunikation mit dem Server und zum Zugriff auf die Ressource verwendet werden. Der Client muss sicherstellen können, dass der Server berechtigt ist die angefragte Ressource bereitzustellen.

A4.5 Der Client muss eine Antwort des Servers entgegennehmen, verarbeiten und ihre Authentizität überprüfen können.

A4.6 Mit den abgefragten Ressourcen muss dem Client eine Regulierung des Klimas innerhalb des Kühlcontainers möglich sein.

3.6 Client Authorization Manager (CAM)

Eingeschränkte Clients können Aufgaben während des Autorisierungsprozesses, die sie selbst aufgrund ihrer eingeschränkt zur Verfügung stehenden Ressourcen nicht durch-

führen können, an ihren CAM delegieren. DCAF sieht vor, dass CAM Zugriffsregeln analog zu denen auf SAM verwaltet und auswertet, die vom COP festgelegt werden. Die Zugriffsregeln von CAM und SAM können sich hinsichtlich einer Ressource unterscheiden [GBB15b]. Beispielsweise könnte CAM dem Client nur eine Untermenge der im Ticket erlaubten Operationen erlauben. Da der Schwerpunkt in dieser Arbeit auf SAM liegt, soll für das Nutzungsszenario auf Clientseite keine feingranulare Zugriffskontrolle verwendet werden. Stattdessen soll CAM lediglich als Zwischenstation für die Kommunikation von Client und SAM eingesetzt werden, wobei er Anfragen des Clients an den zuständigen SAM weiterleitet und das Ticket aus der Antwort an den Client weiterleitet. Für das Nutzungsszenario soll der, oben beschriebene, minimale CAM folgende Anforderungen erfüllen:

A5.1 CAM muss Sicherheitsbeziehungen zu Clients und SAMs besitzen und sichere Verbindungen mit diesen aushandeln können.

A5.2 Die vom Client an CAM gesendeten *Access Requests* müssen empfangen und verarbeitet werden können. Basierend auf den Informationen in der Anfrage muss CAM eine gültige *Ticket Request Message* an den zuständigen SAM richten können.

A5.3 Die Antwort des SAMs muss von CAM verarbeitet und authentifiziert werden.

A5.4 Das in der Antwort enthaltene Ticket muss als Antwort auf den *Access Request* an den Client weitergeleitet werden.

A5.5 CAM muss sicherstellen, dass Tickets nur autorisierten Clients übermittelt werden und die Tickets für Dritte unzugänglich sind, da diese sonst zum Zugriff durch unautorisierte Akteure verwendet werden können.

3.7 Übergangsszenario

In der Regel durchlaufen eingeschränkte Geräte, wie etwa die Sensorknoten im Szenario, während ihres Lebenszyklus' mehrere Besitzerwechsel [Gar+13]. Ein solcher Wechsel findet beispielsweise nach der Herstellung des Gerätes während der ersten Inbetriebnahme (Commissioning-Phase [Gar+13]) statt. Aber auch danach kann sich die Zuständigkeit für ein eingeschränktes Gerät noch ändern. Beispielsweise wenn es innerhalb eines Unternehmes in einem anderen Kontext eingesetzt wird oder wenn es durch ein neues Gerät ersetzt wird und das alte Gerät noch verkauft werden kann und in einer anderen Sicherheitsdomäne weiter betrieben werden soll (Handover-Phase). Ist ein Gerät tatsächlich am Ende seines Lebenszyklus' angelangt und soll nicht weiter verwendet werden, so ist eine geordnete Außerbetriebnahme (Decommissioning-Phase) notwendig.

In dieser Arbeit soll anhand des Nutzungsszenario eine Lösung für die Handover-Phase eines eingeschränkten Gerätes implementiert und evaluiert werden. Diese Phase wurde gewählt, da während dieser sowohl eine Außerbetriebnahme als auch eine (Wieder)Inbetriebnahme stattfindet. Für das Szenario bedeutet dies, dass die Sensorknoten in den Bananenschachteln vor der Verwendung von ROP in seine Sicherheitsdomäne eingeführt werden müssen. Bei einem Handover haben sowohl der vorherige als auch der neue Besitzer Sicherheitsinteressen, die gewahrt werden sollen. Um dies zu ermöglichen und ROP beim Handover eines Gerätes zu unterstützen, sind vom Server und

von SAM folgende Anforderungen zu erfüllen:

A6.1 Der vorherige Besitzer will sicherstellen, dass durch die Weitergabe eines Sensorknotens keine Möglichkeit zur Kompromittierung seines übrigen Netzes möglich ist. Außerdem erwartet der vorherige Besitzer die Vertraulichkeit der auf dem Gerät eventuell noch vorhandenen Daten.

A6.2 Der neue Besitzer will sicherstellen, dass nach der Inbetriebnahme in seiner Sicherheitsdomäne kein Zugriff des vorherigen Besitzers mehr möglich ist. Tickets, die in der vorherigen Sicherheitsdomäne ausgestellt wurden, dürfen nicht mehr vom Server akzeptiert werden.

A6.3 SAM muss in seinem Konfigurationsinterface für den ROP Möglichkeiten vorsehen, einen neuen Server zu den von SAM verwalteten hinzuzufügen und diesen in seine Sicherheitsdomäne einzuführen.

3.8 Eingrenzungen und Annahmen

Zur Realisierung des Nutzungsszenarios sind einige Verfahren und Abläufe vor dem eigentlichen DCAF-Protokollablauf notwendig, die in dieser Arbeit nur eingeschränkt thematisiert werden und auch in DCAF als *out-of-scope* gekennzeichnet sind. Diese Einschränkungen werden im Folgenden benannt und es wird dargelegt in welcher Form sie umgesetzt werden.

Vertrauensbeziehung zwischen den Akteuren

In Abschnitt 2.8 wurde dargestellt, wie die Akteure sich untereinander direkt oder indirekt vertrauen. Akteure, zwischen denen eine direkte Vertrauensbeziehung besteht, müssen mit ihrem jeweiligen Partner Informationen austauschen, mit denen sie sich gegenseitig authentifizieren können. Auf der technischen Ebene handelt es sich bei diesen Informationen häufig um geheime Schlüssel oder Zertifikate. Ein solcher Austausch von Schlüsseln muss im Nutzungsszenario jeweils zwischen den eingeschränkten Geräten und ihren Autorisierungsmanagern stattfinden. Außerdem müssen die beiden Autorisierungsmanager untereinander Informationen zur gegenseitigen Authentifizierung austauschen. Während der Ablauf des Austausches dieser Informationen zwischen dem Server und SAM explizit implementiert und evaluiert werden sollen, soll der Austausch zwischen dem Client und CAM und zwischen den beiden Autorisierungsmanager in dieser Arbeit nicht näher betrachtet werden. Stattdessen wird im Weiteren angenommen, dass dieser Austausch bereits stattfand und die Akteure diese Informationen verwenden können.

Server und Client

Der Client im Kühlcontainer und die Server in den Bananenkisten verwenden im Szenario Sensoren und Aktoren, um die Umgebung zu messen und zu regulieren. Die Erfassung der Sensordaten und die Regulierung des Klimas ist nicht Teil dieser Arbeit. Stattdessen werden auf dem Server exemplarisch Sensorwerte für die Erfassung der

Temperatur in den Bananenschachteln simuliert, sodass dem Nutzungsszenario entsprechend einigermaßen passende Werte geliefert werden können. Im Regelfall befinden sich in einem Kühlcontainer Dutzende Bananenschachteln mit jeweils mindestens einem Sensorknoten. In dieser Arbeit wird exemplarisch ein solcher Knoten implementiert und sein Verhalten evaluiert.

DCAF legt die Besitzverhältnisse an einem eingeschränkten Server nicht explizit fest. So kann es möglich sein, dass ein Server Ressourcen unterschiedlicher ROPs verwaltet und bereitstellt. Dieser Anwendungsfall soll hier nicht betrachtet werden. Die Server sind im Nutzungsszenario zweifelsfrei dem Unternehmen `A` zugeordnet.

Verzeichnisdienst für Ressourcen

In Abschnitt 2.5 wurde erwähnt, dass Verzeichnisdienste Informationen über Server und von ihnen verwaltete Ressourcen in einem Netz bereitstellen können. Außerdem können in Verzeichnisdiensten Informationen über den zuständigen SAM eines Servers hinterlegt sein. Im Nutzungsszenario kann ein Verzeichnisdienst beispielsweise vom Kühlcontainer verwendet werden, um zu ermitteln, welche Ressourcen die Sensorknoten zur Verfügung stellen. Bei Verzeichnisdiensten kann es sich zum einen um dedizierte *Directory-Server* handeln, wobei diese für ein ganzes Netz oder ein Teilnetz verantwortlich sein können. Zum anderen können eingeschränkte Geräte, wie Sensorknoten, auch selbst Informationen über die von ihnen verwalteten Ressourcen bereitstellen. Hierfür kann das, in Abschnitt 2.5 vorgestellte, Verfahren innerhalb des CoAP-Protokolls verwendet werden. Für das Nutzungsszenario soll in dieser Arbeit kein Verzeichnisdienst und keine automatisierte Abfrage von Informationen über Ressourcen stattfinden. Es wird stattdessen davon ausgegangen, dass dieser Schritt bereits stattfand und der Client mit Informationen über die abzufragenden Ressourcen ausgestattet ist. Die Ermittlung des zuständigen Autorisierungsmanagers eines Servers soll, wie in Abschnitt 2.8 beschrieben, mit der *Unauthorized Resource Request Message* und der darauf folgenden *SAM Information Message* erreicht werden.

Zeitsynchronisation

Der Server und sein zuständiger Autorisierungsmanager benötigen für die Durchsetzung der im Ticket enthaltenen Autorisierungsinformationen eine gemeinsame Zeitbasis, da in den Autorisierungsinformationen eine Gültigkeitsdauer des Tickets enthalten sein kann. SAM schreibt die Gültigkeit auf Grundlage seiner lokalen Zeit in das Ticket. Daher muss der Server Informationen über die verwendete Zeitbasis besitzen. Eine solche gemeinsame Zeit kann beispielsweise über Synchronisationsverfahren stattfinden. Eine Auswahl geeigneter Verfahren zur Zeitsynchronisation in Sensornetzen findet sich in [EE03].

DCAF schlägt alternativ vor, dass ein vom Server vorgegebener Zeitstempel verwendet werden kann, welcher vom Server über die *SAM Information message* an den Client verschickt wird, welcher sie wiederum über seinen Autorisierungsmanager mit der Ticketanfrage an SAM sendet. Falls der Server keine Möglichkeit hat eine absolute Zeit zu ermitteln, kann er stattdessen die Anzahl der seit dem letzten Neustart vergangenen Prozessorzyklen als Zeitstempel verwenden. In dieser Arbeit wird dieser

Ansatz, mit vom Server generierten Zeitstempeln, verwendet.

Kapitel 4

Entwurf

Im Folgenden sollen Entwurfsentscheidungen bezüglich des zu implementierenden Systems getroffen werden. Ziel des Entwurfs ist ein System zur Errichtung eines sicheren Kanals zur Kommunikation zwischen eingeschränkten Geräten in einem Sensornetz gemäß den in DCAF spezifizierten Protokollen und Datenformaten und mit den in DCAF definierten Akteuren. Hierbei sollen die in Kapitel 3 festgelegten Anforderungen erfüllt werden. Viele allgemeine Anforderungen an das System werden bereits durch die korrekte Umsetzung des in DCAF festgelegten Ablaufs erfüllt.

Bei DCAF handelt es sich um ein Framework, in dem Entscheidungen über den Ablauf der Kommunikation und die zu verwendenden Protokolle und Datenformate bereits getroffen wurden, bei dem darüber hinaus aber keine Aussage über die konkrete Umsetzung der einzelnen Akteure getroffen wird. Diese müssen daher anwendungsspezifisch entworfen werden. Außerdem trifft DCAF keine Entscheidung über die auszutauschenden Autorisierungsinformationen. Die von DCAF offenen gelassenen Entscheidungen sollen in diesem Kapitel entsprechend dem Nutzungsszenario getroffen werden. Der von DCAF festgelegte Nachrichtenaustausch und die zu verwendenden Protokolle und Datenformate sollen nach Möglichkeit unverändert genutzt werden. Stellen, an denen von diesen Spezifikationen abgewichen wird oder Fälle, in denen diese erweitert oder konkretisiert werden, sind gekennzeichnet.

Die einzelnen Akteure sollen möglichst flexibel entworfen und implementiert werden, sodass sie zum einen teilweise auch in anderen Szenarien einsetzbar sind und zum anderen im gewählten Szenario beliebig durch einen gleichwertigen Akteur ersetzt werden können. Beispielsweise ist es möglich, einen Server auszutauschen, ohne Änderungen an Datenformaten oder Kommunikationsprotokollen vornehmen zu müssen, solange dieser sich an die DCAF-Spezifikation hält.

In der verteilten Architektur von DCAF sind im wesentlichen vier Akteure vorgesehen, welche im Folgenden nach dem Entwurf des Ablaufs der Kommunikation entworfen werden. Bei den Akteuren handelt es sich um die eingeschränkten Geräte Client und Server, denen wenig Speicher und Rechenleistung zur Verfügung steht und die daher nur wenige spezielle Aufgaben erfüllen können. Daher wird ausschließlich für die im Nutzungsszenario nötigen Aufgaben eine Lösung entworfen und umgesetzt.

Die beiden eingeschränkten Geräte können jeweils die Hilfe eines weniger eingeschränkten Autorisierungsmanagers in Anspruch nehmen, an den sie Aufgaben, die sie selbst

überfordern würden, delegieren können. Der Schwerpunkt liegt hierbei auf dem Entwurf und der Implementierung des SAMs.

4.1 Hardwareumgebung

Für die Nutzung im Szenario ist geeignete Hardware für die eingeschränkten und weniger eingeschränkten Geräte auszuwählen. Bei der Auswahl der Hardware für die eingeschränkten Geräte Client und Server ist darauf zu achten, dass diese über ausreichend Speicher, Rechenleistung und Übertragungskapazität verfügen, um die im DCAF-Ablauf verwendeten Protokolle CoAP und DTLS zu verwenden. Darüber hinaus müssen diese sich als Sensorknoten verwenden lassen. Die Geräte müssen entweder bereits über integrierte Sensoren verfügen oder Anschlussmöglichkeiten für externe Sensoren oder Aktoren bereitstellen, um die für das Szenario vorgesehenen Aufgaben bewältigen zu können. Darüber hinaus soll die zu verwendende Hardware den Einsatz von weit verbreiteter Standardsoftware, wie Betriebssystemen und Softwarebibliotheken, ermöglichen, um aufbauend auf diesen das Szenario und den DCAF-Protokollablauf implementieren zu können.

In Abschnitt 2.2 wurden die in [RFC7228] festgelegten Klassen für eingeschränkte Geräte eingeführt. DCAF ist nicht dafür ausgelegt auf sehr eingeschränkten Geräten der Klasse C0 verwendet zu werden. Die von DCAF verwendeten Protokolle, insbesondere die Verwendung von DTLS, würde die Geräteklasse überfordern. DCAF wurde für Geräte entwickelt, die mindestens die Anforderungen der Klasse C1 erfüllen [GBB14]. Geräte dieser Klasse verfügen über Arbeitsspeicher in einer Größenordnung von etwa 10 KiB und Platz für Programmcode in einer Größenordnung von etwa 100 KiB [RFC7228]. Die auszuwählende Hardware sollte dementsprechend mindestens die technischen Anforderungen der Klasse C1 erfüllen.

Eine Geräteklasse, welche diese Anforderungen erfüllt, ist die WiSMote-Plattform[1]. Hierbei handelt es sich um einen Sensorknoten, der speziell für den Einsatz in *Wireless Sensor Networks* entwickelt wurde. WiSMotes basieren auf einer MSP430 series 5 CPU von Texas Instruments, welche aus einem 16-bit RISC Mikrocontroller besteht und mit einer Taktfrequenz von 16 Mhz arbeitet. Der Mikrocontroller enthält 16 KiB Arbeitsspeicher zur Datenablage und 128 KiB bis 256 KiB Programmspeicher [Ant]. Damit liegen WiSMotes bezüglich des Arbeitsspeichers in einer Größenordnung mit den in Klasse C1 enthaltenen Geräten. Die Größe des Programmsspeichers entspricht eher den Geräten der Klasse C2. Daher ist dieser Mikrocontroller für den Einsatz von DCAF geeignet und wird für das Szenario, in einer Ausführung mit 128 KiB Arbeitsspeicher, verwendet.

Zur Kommunikation kann der auf dem WiSMote enthaltene *CC2520* RF-Transceiver von Texas Instruments verwendet werden, der im ISM-Band mit 2,4 GHz arbeitet, welches durch die Allgemeinzuteilung[2] in Deutschland frei verwendet werden kann. Der Chip bietet Funktionen nach IEEE 802.15.4 und kann eine maximale Übergagungsrate von 250 kbit/s erreichen, wobei diese stark von den, im Einsatzzweck vorherrschenden,

[1] WiSMote: http://www.aragosystems.com/en/wisnet-item/wisnet-wismote-item.html (abgerufen am 08. Juni 2015)

[2] Allgemeinzuteilung: https://www.mikrocontroller.net/articles/Allgemeinzuteilung (abgerufen am 08. Juni 2015)

Umgebungsbedingungen abhängt. WiSMotes können, abhängig von ihrer Konfiguration, bereits einen Temperatur-, Licht- und Beschleunigungssensor enthalten. Weitere Sensoren und Aktoren können via I^2C- oder SPI-Schnittstelle angeschlossen werden. Der Mikrocontroller verfügt außerdem, je nach Konfiguration, über zwei bis acht MiB nicht flüchtigen Flash-Speicher, der beispielsweise zur Aufzeichnung von anfallenden Sensordaten verwendet werden kann. WisMotes eignen sich, aufgrund ihres niedrigen Energiebedarfs und ihres weiten Eingangsspannungsbereich von 2,2 V bis 3,6 V, für den Batteriebetrieb. Alternativ ist eine Spannungsversorgung über USB möglich. Im Batteriebetrieb können verschiedene, von der CPU und dem RF-Transceiver unterstützte, Schlafzustände verwendet werden, in denen nur wenige Mikroampere benötigt werden. Hierdurch ist ein Betrieb über mehrere Monate mit einer Batterieladung möglich, wobei nur periodisch oder ereignisgesteuert der Schlafzustand zum Abarbeiten von Aufgaben verlassen wird. Die WiSMote-Plattform unterstützt unter anderem das Betriebssystem Contiki. Hierfür steht ein großer Softwarepool zur Verfügung.

Die Ebene der weniger eingeschränkten Autorisierungsmanager SAM und CAM stellt weniger strenge Anforderungen an die auszuwählende Hardware als die eingeschränkten Geräte. Bei diesen kann es sich, wie in Abschnitt 2.8 beschrieben, im Szenario entweder um vollwertige Computer oder beispielsweise Smartphones von Mitarbeitern der Transportfirma handeln. Diese müssen einen ständigen Netzzugang gewährleisten und über ausreichend nicht flüchtigen Speicher verfügen, um Zugriffsregeln und Tickets speichern zu können.

4.2 Contiki

Als Betriebssystem für die eingeschränkten Geräte im Szenario wird Contiki[3] verwendet, welches speziell an die Bedürfnisse von Sensornetzen angepasst wurde. Contiki wird seit 2003 als Open-Source-Projekt entwickelt und gilt als ausgereifte Software mit breiter Akzeptanz beim Einsatz auf Mikrocontrollern [DGV04]. Die Verwendung eines Betriebssystems kann auf eingeschränkter Hardware ähnliche Vorteile wie auf nicht eingeschränkten Geräten bieten. Das Betriebssystem stellt eine Schicht zwischen Anwendungssoftware und Hardware dar, die von der darunter liegenden Hardware abstrahiert, indem dem Anwendungsprogramm unabhängig von der Hardware einheitliche Schnittstellen zur Verfügung gestellt werden. Somit ist eine weitgehend hardwareunabhängige Entwicklung von Anwendungsprogrammen möglich, die potenziell auf allen Geräten, auf denen Contiki lauffähig ist, ausgeführt werden können. Contiki ist bereits auf einer großen Anzahl an Mikrocontrollern lauffähig und bietet eine einfache Möglichkeit zur Portierung auf weitere Hardware.[4] Contiki unterstützt unter anderem den in dieser Arbeit verwendeten WiSMote und die von ihm verwendete MSP430-CPU.

Ein weiterer Vorteil der Verwendung eines Betriebssystems wie Contiki sind die bereits mitgelieferten Treiber und Softwarebibliotheken, da diese die Anwendungsentwicklung unterstützen können. Beispielsweise ist in Contiki zur Kommunikation bereits der uIP TCP/IP Stack[5] integriert. Dieser erlaubt die Kommunikation über TCP und UDP mit Adressierung über IPv4 oder IPv6. Auch die Kommunikation über die darunter

[3] Contiki: http://contiki-os.org/ (abgerufen am 08. Juni 2015)
[4] Contiki Hardware: http://www.contiki-os.org/hardware.html (abgerufen am 08. Juni 2015)
[5] Contiki uIP: http://contiki.sourceforge.net/docs/2.6/a01793.html (abgerufen am 08. Juni 2015)

liegenden 6LoWPAN- und IEEE-802.15.4-Schichten sind in Contiki bereits integriert. Diese und der integrierte uIP-Stack werden in dieser Arbeit mit Adressierung über IPv6 zur Kommunikation über UDP-Verbindungen verwendet.

Wie die meisten Betriebssysteme unterstützt auch Contiki das Laden verschiedener Anwendungsprogramme zur Laufzeit. Contiki selbst und auch die Schnittstellen für die Anwendungsprogramme sind in der Programmiersprache C umgesetzt. Die Anwendungsprogramme, und damit die eingeschränkten Geräte des Szenarios, sind daher ebenfalls in C zu implementieren. Contiki benötigt selbst nur wenig Arbeitsspeicher und Programmspeicher. Ein kompiliertes Contiki-System benötigt exklusive Anwendungsprogrammen und inklusive voller IPv6 Unterstützung und RPL-Routing weniger als 10 KiB Arbeitsspeicher und 30 KiB Programmspeicher [Bac+13].

Contiki verwendet einen ereignisgesteuerten Kernel, bei dem Anwendungsprogramme nicht selbstständig ablaufen, sondern zustandsgesteuert beim Auftreten von Systemereignissen, wie etwa Timer- oder Hardwareinterrupts oder ankommenden Netzwerkpaketen, aufgerufen werden.[6] Als Prozessmodell werden von Contiki Protothreads mit kooperativem Multitasking unterstützt. Protothreads sind leichtgewichtige Prozesse, die nur wenige Zustandsinformationen verwalten müssen und daher auch für Geräte mit eingeschränktem Speicher geeignet sind. Weitere Informationen zum von Contiki verwendeten Prozessmodell finden sich in Abschnitt 5.5. Um Nebenläufigkeit von Anwendungsprogrammen zu ermöglichen, unterstützt Contiki optional preemptives Multithreading als Softwarebibliothek auf Anwendungsebene. Auch hierzu finden sich weitere Informationen in Abschnitt 5.5.

Um die Entwicklung von leichtgewichtigen Anwendungen mit einem geringen Energieverbrauch zu unterstützen, stellt Contiki Stromspartechniken bereit. Neben der Unterstützung der Schlafmodi des verwendeten Prozessors wird mit *ContikiMAC* ein Mechanismus bereit gestellt, um die Häufigkeit des Aufwachens aus dem Schlafmodus zu reduzieren. Mit ContikiMAC können Geräte im Schlafmodus verweilen und dennoch auf ankommende Nachrichten über Funk reagieren.

Neben Contiki existieren noch weitere Betriebssysteme, die explizit für eingeschränkte Geräte entwickelt wurden. TinyOS und RIOT OS sind zwei Beispiele, die unter anderem auch auf der hier verwendeten MSP430-CPU lauffähig sind. Die Betriebssysteme unterscheiden sich hinsichtliches ihres internen Aufbaus und Ablaufs, stellen aber einen ähnlichen Funktionsumfang zur Verfügung. So ist auch in TinyOS und RIOT OS eine Kommunikation über 6LoWPAN und einem IPv6-Stack möglich. Daher würden sich diese ebenfalls zur Verwendung im Szenario eignen. Ein Vergleich der drei Betriebssysteme Contiki, TinyOS und RIOT OS findet sich in [Bac+13].

4.2.1 Cooja

Neben dem eigentlichen Betriebssystem stellt Contiki noch einige Tools bereit, um die Entwicklung und Evaluation von Contiki-Anwendungen zu unterstützen. Dazu gehört unter anderem eine Simulationsumgebung für Sensornetze, die den Namen Cooja trägt. Statt die entwickelte Software direkt auf der Hardware zu testen, können die im Szena-

[6] Contiki Processes: https://github.com/contiki-os/contiki/wiki/Processes (abgerufen am 08. Juni 2015)

rio verwendeten eingeschränkten Geräte in Cooja simuliert werden. Cooja unterstützt die Simulation einer Reihe von Mikrocontrollern (in Cooja: Motes), unter anderem die MSP430-CPU beziehungsweise die WiSMote-Plattform. Es können in einer Simulation beliebig viele Geräte parallel verwendet werden und miteinander interagieren [Seh13].

Cooja kann die meisten Schnittstellen von Mikrocontrollern simulieren. Cooja unterstützt etwa die Kommunikation im Sensornetz, so dass auch das Zusammenwirken von Akteuren simuliert werden kann. Außerdem erlaubt Cooja eine Positionierung von Motes in einem zweidimensionalen Raum. Diese kann verwendet werden, um die Nichterreichbarkeit von Akteuren zu simulieren oder das Routing in einem Sensornetz zu beeinflussen. Dazu wird jedem Mote eine Funkreichweite zugewiesen, innerhalb der dieser Mote kommunizieren kann. Motes außerhalb dieser Reichweite können von diesem Mote nicht direkt erreicht werden, allerdings eventuell indirekt über einen Mote zwischen den beiden. Erstellte Simulationen können gespeichert und wieder geladen werden und hierbei optional die Contiki-Anwendungen für die enthaltenen Akteure neu übersetzen. Außerdem kann die Simulation jederzeit pausiert und fortgesetzt werden, wobei auch die Ausführungsgeschwindigkeit der Simulation angepasst werden kann [Seh13]. Dies kann genutzt werden, um beispielsweise Szenarien, in denen in großen Abständen Aufgaben erledigt werden, zu beschleunigen oder Szenarien, die in kurzer Zeit sehr viel Interaktion beinhalten, zu drosseln, um diesen so besser folgen zu können. Cooja verwendet intern zur Simulation von Mikrocontrollern mit MSP430-CPU den Simulator MSPSim[7], um die Contiki-Anwendungen auszuführen.

Um die Fehlersuche und die Evaluation eines in Cooja simulierten Szenarios zu unterstützen, bietet Cooja Funktionen zur Auswertung des Ressourcenverbrauchs von Motes und zum Bereitstellen von Informationen über die Motes. So werden beispielsweise die von dem Mote auf der Standardausgabe ausgegebenen Nachrichten mit Filtermöglichkeit in Cooja dargestellt. Während der Simulation können der Stack und die Prozessorregister überwacht werden. Außerdem enthält Cooja eine Zeitleiste, um im Sensornetz gesendete Pakete zu visualisieren. Diese Funktionen können zum Debuggen von Anwendungen und zum Testen des Zusammenspiels von Anwendungen und Geräten verwendet werden. Außerdem kann mit Cooja, mit wenigen Zeilen Code für jeden Mote, der Energiebedarf eines Motes geschätzt werden.

Cooja soll in dieser Arbeit zum Testen während der Entwicklung und zur Evaluation des Ressourcenverbrauchs der eingeschränkten Geräte im Szenario verwendet werden. Hierbei werden in Cooja der eingeschränkte Client und der eingeschränkte Server sowie ihre Kommunikation simuliert. Um den Geräte in der Simulation eine Kommunikation mit Geräten außerhalb von Cooja zu ermöglichen, ist ein Gateway-Gerät nötig, welches die beiden Netze miteinander verbindet. In Cooja kann hierzu ein sogenannter *Border-Router* verwendet werden. Dieser wird in Abschnitt 5.1 beschrieben.

4.3 Entwurf des Ablaufs nach DCAF

Bei DCAF handelt es sich, wie bereits in Abschnitt 2.8 beschrieben, um ein Framework zur sicheren Kommunikation zwischen eingeschränkten Geräten. DCAF trifft bereits Entscheidungen über die Architektur und den Nachrichtenaustausch beziehungswei-

[7] MSPSim: https://github.com/mspsim/mspsim (abgerufen am 08. Juni 2015)

se die zu verwendenden Kommunikationsprotokolle, lässt aber den internen Aufbau der einzelnen Akteure weitgehend undefiniert, da dieser stark abhängig vom Szenario ist, in dem er eingesetzt wird. Außerdem kann in DCAF bei einigen Entscheidungen eine Auswahl aus Alternativen getroffen werden. Beispielsweise erlaubt DCAF zwei unterschiedliche Mechanismen zur Schlüsselerzeugung für sichere Verbindungen. Im Folgenden sollen die von DCAF festgelegten Protokolle und Nachrichten erläutert und die für das Nutzungsszenario entsprechenden Entscheidungen getroffen werden. Stellen, an denen von den von DCAF vorgeschriebenen Entscheidungen abgewichen oder an denen etwas hinzugefügt wird, sind gekennzeichnet.

Für die Kommunikation zwischen den Akteuren sieht DCAF, in Fällen in denen mindestens ein eingeschränktes Gerät an der Kommunikation beteiligt ist, CoAP als Transportprotokoll der Anwendungsschicht vor. DCAF legt nicht fest, ob zwischen den Akteuren *Confirmable-* oder *Non-confirmable*-Nachrichten ausgetauscht werden. In dieser Arbeit werden zwischen allen Akteuren confirmable CoAP-Nachrichten ausgetauscht.

Zum Schutz von Vertraulichkeit, Integrität und Authentizität der CoAP-Nachrichten schreiben CoAP und DCAF die Verwendung von verschlüsselten DTLS-Verbindungen vor [GBB15a]; [RFC7252]. Hierbei soll ausschließlich symmetrische Verschlüsselung mit zuvor vereinbarten Schlüsseln (Pre-Shared Key, PSK) zur Verschlüsselung und Authentifizierung verwendet werden, da diese auch für eingeschränkte Geräte geeignet sind. [RFC7252] schreibt vor, dass für sichere CoAP-Verbindungen mindestens die Cipher Suite `TLS_PSK_WITH_AES_128_CCM_8` unterstützt werden muss. Da DCAF diese ebenfalls verpflichtend vorschreibt, wird diese Cipher Suite, in dieser Arbeit, für alle DTLS-Verbindungen verwendet.

Bei der Kommunikation zwischen den weniger eingeschränkten Autorisierungsmanagern sieht DCAF alternativ oder zusätzlich zu CoAP und DTLS die Verwendung von HTTP und TLS vor. In dieser Arbeit wird für die sichere Verbindung zwischen den Autorisierungsmanagern HTTP 1.1 und TLS 1.2 verwendet. Zur gegenseitigen Authentifizierung der Autorisierungsmanager werden selbst-signierte X.509-Zertifikate verwendet, welche zusätzlich durch eine ebenfalls selbst-signierte und erstellte Certification Authority (CA) signiert wurden. Die CA wird von CAM und SAM verwendet, um die Gültigkeit des vom Kommunikationspartner verwendeten Zertifikats zu überprüfen. Einzelheiten zur Umsetzung der gegenseitigen Authentifizierung finden sich bei den jeweiligen Akteuren in Abschnitt 5.3 und Abschnitt 5.4.

4.4 Nachrichtenformate

In den Grundlagen, in Abschnitt 2.8, wurde die Bedeutung der in einem System mit DCAF-Architektur gesendeten Nachrichten bereits erklärt und der Nachrichtenfluss dargelegt (siehe Abbildung 2.6). Im Folgenden werden daher die Form und Inhalte der Nachrichten festgelegt. Abweichungen und Erweiterungen zu den in DCAF festgelegten Inhalten werden gekennzeichnet und erläutert. Details zu den notwendigen Schritten zur Verarbeitung der Nachrichten finden sich bei den jeweiligen Akteuren in den Abschnitten 4.6, 4.8, 4.7 und 4.9.

Als Datenformat für die auszutauschenden Nachrichten sieht DCAF den Einsatz von CBOR vor, da dieses auf die Anforderungen von eingeschränkten Geräten zugeschnit-

ten ist, indem Nachrichten mit geringem Overhead ermöglicht werden [RFC7049]. Zwischen den Autorisierungsmanagern sind auch andere Datenformate, wie beispielsweise JSON, zum Datenaustausch möglich, da SAM und CAM weniger einegschräkt sind. In dieser Arbeit wird zwischen allen Akteuren CBOR als Datenformat in den auszutauschenden Nachrichten verwendet. Da Daten in CBOR in einem binären Format vorliegen, ist ein übersichtliches Aufschreiben zur Dokumentation nicht ohne Weiteres möglich. Daher sieht CBOR die *Diagnostic Notation* vor, in der CBOR-Daten in einer JSON ähnlichen Syntax aufgeschrieben werden können [RFC7049]. Um CBOR Datenstrukturen formal zu definieren, kann alternativ die *CBOR data definition language* verwendet werden [VBS15]. In dieser Arbeit wird die Diagnostic Notation oder die im Code benutzten Strukturen zur Beschreibung verwendet.

Unauthorized Resource Request Message

Um den für einen Server verantwortlichen Autorisierungsmanager zu ermitteln, kann der Client eine beliebige Ressource auf dem Server abfragen, ohne ein Ticket mitzusenden. Durch das Fehlen eines Tickets haben Client und Server keine Möglichkeit zur gegenseitigen Authentifizierung und zum Aufbauen einer sicheren Verbindung. Die *Unauthorized Resource Request Message* muss daher als unverschlüsselte CoAP-Nachricht vom Client an den Server gesendet werden. In den Beispielen im DCAF-Draft wird hierbei ein `PUT`-Request auf eine CoAP-Ressource ausgeführt. In dieser Anfrage werden Payload-Daten mitgesendet, in denen beispielsweise Sensordaten enthalten sind. Da die Nachricht unverschlüsselt übertragen wird, kann hierbei die Anforderung A1.1 nach Vertraulichkeit und Integrität der Sensordaten verletzt werden. Daher soll im Szenario für unautorisierte Anfragen beim Server ausschließlich `GET`-Request auf CoAP-Ressourcen, bei denen keine Payload-Daten mitgesendet werden, verwendet werden.

SAM Information Message

Der Server lehnt die unautorisierte Anfrage des Clients mit dem CoAP-Response-Code `4.01` (Unauthorized) ab und sendet in seiner Antwort eine *SAM Information Message* mit Informationen zum für den Server zuständigen Autorisierungsmanager. Diese liegen hierbei im CBOR-Format mit folgendem Inhalt vor:

```
struct sam_information_msg {
  char *SAM;
  long TS;
};
```

Bei der Zeichenkette SAM handelt es sich um einen URI, der auf eine Ressource auf einem SAM verweist, unter der Ticket-Anfragen von diesem Autorisierungsmanager angenommen werden. TS enthält einen Zeitstempel in einer vom Server gewählten Zeitbasis, der bis zu SAM mitgesendet wird und auf diesem für die Angabe einer Gültigkeitsdauer im Ticket auf Basis der Zeitbasis des Servers genutzt werden kann.

Access Request

Der Client nutzt die Informationen in der SAM Information Message, um mittels Access Request CAM anzuweisen ein Ticket vom zuständigen SAM abzufragen. Access Requests werden als POST-Request an den URI `/client-auth` gesendet. Dieser enthält einen Payload im CBOR-Format mit folgenden Informationen:

```
struct access_request {
  char *SAM;
  struct AIF SAI[];
  long TS;
};
```

SAM und der Timestamp werden aus der SAM Information Message übernommen. Außerdem muss in der Anfrage vom Client festgelegt werden, auf welche Ressource er auf dem Server mit welchen Aktionen (CoAP-Methoden) zugreifen möchte. Hierfür wird die Datenstruktur *SAM Authorization Information (SAI)* verwendet. Die Autorisierungsinformationen werden in DCAF im *Authorization Information Format (AIF)* [Bor15] angegeben. In der aktuellen Version des AIF werden die zu autorisierenden Ressourcen und Methoden als Liste (CBOR-Array) von AIF-Objekten angegeben. Wobei jedes AIF-Objekt ebenfalls als CBOR-Array mit zwei Elementen repräsentiert wird. Das erste Element besteht aus einem URI, welcher die Ressource beschreibt. Im zweiten Element sind die zu autorisierenden CoAP-Methoden als eine einzelne CBOR-Number kodiert. Details zu der Kodierung der CoAP-Methoden finden sich in Abschnitt 4.6 und [Bor15]. Außerdem finden sich in Abschnitt 4.6 ausführliche Informationen zu den auch im Ticket verwendeten Autorisierungsinformationen.

Ticket Request Message

Wie in Abschnitt 3.8 beschrieben, soll CAM, anders als in DCAF spezifiziert, keine eigenen Zugriffsregeln für Clients verwalten und durchsetzen. Stattdessen soll er ausschließlich die Daten aus dem, vom Client über CoAP und DTLS empfangenen, Access Request an den zuständigen SAM über HTTP und TLS unverändert, als Ticket Request Message, weiterleiten. Diese sendet CAM an die im Access Request genannte Ressource des SAMs.

Ticket Grant Message

Nachdem SAM die Ticket Request Message verarbeitet und entschieden hat, ob ein Zugriff auf die angegebenen Ressourcen gestattet werden soll, erstellt er im Erfolgsfall ein Ticket und sendet dieses als Antwort auf den HTTP-Request in einer *Ticket Grant Message* an CAM. Die Verarbeitung und Autorisierung auf SAM werden in Abschnitt 4.6 ausführlich betrachtet. Der Payload der Antwort besteht hierbei ausschließlich aus dem Ticket im CBOR-Format und beinhaltet folgende Felder:

```
struct ticket_grant_msg {
  struct face {
    struct aif SAI[];
    int SEQ_NR;
    long TS;
```

```
        long L;
        int G;
        unsigned char *PSK; // Optional
      }
      unsigned char *key;
      struct client_information {
        unsigned char *verifier;
      }
    }
```

Ein DCAF-Ticket besteht aus einem Ticket-Face, welches zur Verarbeitung auf dem Server vorgesehen ist, und aus Informationen für den Client (*Client Information*). Das Ticket-Face enthält *SAM Authorization Information (SAI)*, die in DCAF das gleiche Format, wie die vom Client angeforderten Ressourcen im Access Request haben.

Details zu den von SAM ins Ticket geschriebenen Autorisierungsinformationen finden sich in Abschnitt 4.6. Neben den Autorisierungsinformationen enthält das Ticket-Face in DCAF verpflichtend einen aktuellen Zeitstempel, der entweder von SAM auf Basis seiner lokalen Zeit generiert wird oder, falls ein vom Server generiertert Zeitstempel in der Ticket Request Message mitgesendet wurde, daraus verwendet wird. Außerdem muss im Face die verwendete Methode zur Schlüsselerzeugung (G) angegeben werden. Optional kann eine Lifetime (L) des Tickets im Face enthalten sein, welche die Gültigkeit des Tickets in Sekunden angibt. Diese Lifetime soll von SAM, wie in DCAF festgelegt, auch im HTTP-Header `Max-Age` angegeben werden, um das Zwischenspeichern von Tickets auf CAM zu ermöglichen.

In DCAF ist im Ticket kein Feld vorgesehen, um eine Identifizierung des Tickets zu ermöglichen. Für das verwendete Szenario wurde daher eine Sequenznummer zum Ticket-Face hinzugefügt, welche auch zum Widerrufen von Tickets verwendet wird. Details zur Erzeugung der Sequenznummer und zum Widerrufen von Tickets mit dieser finden sich in Abschnitt 4.6 und Abschnitt 4.7. Je nach gewählter Methode zur Berechnung des Pre-Shared Keys für die Kommunikation zwischen Client und Server, muss das Ticket-Face unter Umständen verschlüsselt werden. In diesem Fall wird im Face zusätzlich noch der PSK für die Verbindung vom Client zum Server angegeben. Außerdem muss im Ticket festgelegt werden, welcher Schlüssel für die Verschlüsselung verwendet wurde. Details zu möglichen und zur gewählten Methode zur Schlüsselerzeugung finden sich in Abschnitt 4.4

Die *Client Information* enthalten im vom SAM erstellten Ticket ausschließlich einen Verifier, der vom Client für die sichere Verbindung zum Server als DTLS-PSK verwendet wird. Der Verifier ist nur für den Client gedacht und darf von diesem nicht mit an den Server gesendet werden. Da der Verifier unverschlüsselt im Ticket enthalten ist, ist das Ticket über sichere Verbindungen zu übertragen.

Ticket Transfer Message

Da CAM im Szenario lediglich als Vermittler zwischen Client und SAM verwendet wird, fügt CAM dem von SAM erhaltenen Ticket, anders als im DCAF-Draft spezifiziert, keine eigenen Autorisierungsinformationen hinzu beziehungsweise nimmt keine Änderungen an den vorhandenen vor. Stattdessen leitet er das Ticket unverändert in einer *Ticket Transfer Message*, als Antwort auf den vom Client gesendeten Access

Requests, an den Client weiter.

Authorized Resource Request Message

Mit dem Ticket besitzt der Client nun einen Zugriffsausweis für die darin enthaltenen Ressourcen und darf die angegebenen Methoden auf diesen ausführen. Um nun eine Ressource abzufragen, muss der Client zunächst eine sichere DTLS-Verbindung zum Server herstellen. Die Verbindung zwischen Client und Server ist in DCAF die einzige, für die nicht bereits zuvor ein Schlüsselaustausch stattfand, sondern die Informationen im Ticket als Schlüssel beziehungsweise zum Erzeugen eines Schlüssels verwendet werden. Der Client benutzt den im Ticket enthaltenen Verifier als DTLS-PSK. Um dem Server die Berechnung des nötigen Schlüssels zu ermöglichen, muss der Client das Ticket-Face an den Server übertragen. Das Face muss den Server erreichen, bevor der PSK auf Serverseite, während des DTLS-Handshakes, zum ersten Mal verwendet werden kann.

Da in eingeschränkten Netzen die Übertragungskapazität begrenzt ist und durch den Versand jeder Nachricht Energie benötigt wird, sieht DCAF vor, dass für die Übertragung des Ticket-Faces keine zusätzliche Nachricht vom Client an den Server gesendet werden muss. Stattdessen wird das Face während des DTLS-Handshakes in der *ClientKeyExchange*-Nachricht im `psk_identity`-Feld gesendet. Die Verarbeitung des Ticket-Faces und die Zugriffskontrolle auf dem Server werden in Abschnitt 4.7 thematisiert. Nach einer erfolgreichen Zugriffskontrolle sendet der Server die angeforderte CoAP-Ressource an den Client.

Ticket Revocation Message

In Anforderung A1.5 wird für ROP eine Möglichkeit zum Widerrufen von Tickets gefordert. Der DCAF-Draft sieht hierfür zum einen ein implizites Widerrufen über die Gültigkeitsdauer eines Tickets und zum anderen das explizite Widerrufen über *Ticket Revocation Messages* vor. Allerdings nimmt DCAF keine genauere Definition der zu sendenden Nachricht und des zu verwendenden Mechanismus' vor. Daher wird diese Nachricht an dieser Stelle entworfen. In dieser Arbeit sollen beide Mechanismen zum Widerrufen von Tickets verwendet werden. Daher wird im Folgenden die Form und der Inhalt der Ticket Revocation Message erläutert.

Ticket Revocation Messages werden ausschließlich von SAM an die von ihm verwalteten Server gesendet. Da Tickets in DCAF immer explizit den Zugriff auf nur einen Server gestatten, muss zum Widerrufen eines Tickets auch nur dieser Server kontaktiert werden. Die Verbindung wird mit DTLS und dem zuvor zwischen SAM und dem Server ausgetauschten Schlüssel $K(SAM, S)$ abgesichert. SAM führt einen CoAP-POST-Request auf die Ressource `/revocations` mit einem Payload aus, der aus einer Liste von CBOR-Numbers besteht. Die einzelnen Elemente der Liste geben die zu widerrufenden Tickets anhand ihrer Sequenznummer an, die für jedes Ticket des Servers eindeutig ist. So können mit einer Ticket Revocation Message mehrere Tickets in einem Schritt widerrufen werden. Der Server merkt sich die widerrufenen Sequenznummern und antwortet mit dem Response-Code `2.00` und leerem Payload.

Der Server verwendet einen Sliding-Window-Mechanismus zur Verwaltung von widerrufenen Tickets. Details zu diesem finden sich in Abschnitt 4.6 und Abschnitt 4.7.

4.5 Schlüsselerzeugung

Der Client und der Server sind die beiden einzigen Kommunikationspartner, für die nicht bereits zuvor, im Vorlauf des Nutzungsszenarios, zwischen den beteiligten COP und ROP, Schlüssel zur Kommunikation ausgetauscht wurden. Der PSK für die DTLS-Verbindung wird dem Client und dem Server über das Ticket bekannt gemacht. DCAF sieht zwei Mechanismen zum Erzeugen des Schlüssels $K(C, S)$ vor, die im Folgenden erklärt werden und von denen einer für die Verwendung in dieser Arbeit ausgewählt wird [GBB15a, S.24-25].

DTLS PSK Transfer

Beim *DTLS PSK Transfer* wird der PSK von SAM erzeugt und über das Ticket sowohl an den Client als auch den Server gesendet. SAM legt den Schlüssel im für den Server gedachten Ticket-Face ab und verschlüsselt dieses mit dem Schlüssel $K(SAM, S)$, der bereits zuvor zwischen SAM und dem Server ausgetauscht wurde. Außerdem legt SAM den Schlüssel im Verifier ab, um dem Client eine Kopie von diesem zugänglich zu machen. Der Verifier kann nicht verschlüsselt werden, da SAM und der Client keinen gemeinsamen Schlüssel $K(SAM, C)$ besitzen. Daher fordert DCAF, dass Tickets nur über sichere Verbindungen zum Client gesendet werden. Beim Aufbauen der DTLS-Verbindung zwischen Client und Server verwendet der Client den Verifier als PSK und sendet in der ClientKeyExchange-Nachricht während des Handshakes das verschlüsselte Face zum Server. Dieser kann die erhaltenen Informationen mit seinem Schlüssel $K(SAM, S)$ entschlüsseln und den im Face enthaltenen PSK für die Verbindung verwenden.

Distributed Key Derivation

Bei der *Distributed Key Derivation* wird der PSK für die DTLS-Verbindung verteilt, jeweils von SAM und dem Server autonom, basierend auf dem Ticket-Face und dem Schlüssel $K(SAM, S)$, berechnet. DCAF legt den folgenden Ablauf fest: SAM erstellt ein Ticket und berechnet den PSK für den Verifier, indem er einen HMAC-Algorithmus auf das Ticket-Face zusammen mit dem, mit dem Server geteilten, Schlüssel $K(SAM, S)$ anwendet. Ein HMAC-Algorithmus erzeugt einen Message Authentication Code (MAC) basierend auf einer kryptographischen Hashfunktion und einem Schlüssel. In dieser Arbeit wird hierfür der in DCAF verpflichtend vorgegebene Algorithmus `HMAC_SHA256` verwendet. Der berechnete HMAC wird von SAM in den Verifier des Tickets geschrieben. Es findet keine Verschlüsselung des Ticket-Faces statt. Der Client benutzt den Verifier als PSK für die DTLS-Verbindung mit dem Server und sendet ebenfalls das, in diesem Fall unverschlüsselte, Ticket-Face in der ClientKeyExchange-Nachricht zum Server. Der Server kann nun den gleichen HMAC-Algorithmus auf das Ticket-Face und den mit SAM geteilten Schlüssel $K(SAM, S)$ anwenden. Hierdurch

erhält er, wenn keine Veränderung oder Manipulation des Ticket-Faces stattfand, einen zum vom SAM berechneten Verifier identischen DTLS-PSK.

Durch das unverschlüsselte Übertragen des Ticket-Faces in der ClientKeyExchange-Nachricht kann die Vertraulichkeit der darin enthaltenen Informationen verletzt werden, falls jemand die Nachricht mitschneiden kann. In diesem Fall kann ein Dritter beispielsweise Informationen darüber erlangen, für welche Ressourcen ein Client einen Zugriffsausweis besitzt. Allerdings kann der Server die Integrität und Authentizität des Ticket-Faces sicherstellen. Denn nur ein unmanipuliertes Ticket-Face führt in Verbindung mit dem Schlüssel $K(SAM, S)$ zu einem identischen PSK durch den verwendeten HMAC-Algorithmus. Bei einer Manipulation des Ticket-Faces und der damit verbundenen Erzeugung eines nicht identischen PSK schlägt der Aufbau der DTLS-Verbindung fehl. Zu keiner Zeit kann jedoch nach erfolgreichem DTLS-Handshake die Vertraulichkeit, Integrität und Authentizität der über die DTLS-Verbindung ausgetauschten Daten, wie etwa Ressourcen des Servers, durch das unverschlüsselte Übertragen des Ticket-Faces verletzt werden.

Bei der verteilten Schlüsselerzeugung kann der Client anhand des Tickets erkennen, welcher Zugriff ihm auf welche Ressourcen erlaubt wurde. Beim DTLS PSK Transfer besteht diese Möglichkeit für den Client nicht, da dieser das Ticket-Face nicht entschlüsseln kann. Bei der verteilten Schlüsselerzeugung wird zu keiner Zeit der vollständige PSK zwischen Client und Server ausgetauscht. Dies ist beim DTLS PSK Transfer der Fall, allerdings wird dieser hier verschlüsselt im Ticket-Face repräsentiert. In beiden Methoden beruht die Sicherheit der Schlüsselerzeugung demnach auf der Geheimhaltung des Schlüssels $K(SAM, S)$ und der Sicherheit der verwendeten Algorithmen. In dieser Arbeit wird für das Nutzungsszenario die verteilte Schlüsselerzeugung verwendet, da bei dieser zu keiner Zeit ein Schlüssel über ungesicherte Verbindungen übertragen werden muss.

4.6 Server Authorization Manager (SAM)

SAM hat in DCAF, wie in Abschnitt 2.8 beschrieben, die Aufgabe, eine Authentifizierung und Autorisierung für den Zugriff auf einen Server durchzuführen, sodass der Server lediglich die Zugriffskontrolle durchsetzen muss. Dafür muss SAM, wie in Anforderung A2.1 definiert, intern eine Vorstellung der verwendeten Subjekte und Objekte der Autorisierung haben und es ermöglichen, diese in Zugriffsregeln miteinander zu verknüpfen, damit eine Autorisierung durchgeführt werden kann.

SAM muss außerdem Ticket Request Messages, die von CAMs über HTTP gesendet werden, verarbeiten können und diese mit den gespeicherten Zugriffsregeln abgleichen und im Erfolgsfall ein Ticket erstellen können. Auch die ausgestellten Tickets und Widerrufungen dieser müssen von SAM verwaltet werden.

Des Weiteren soll SAM eine Möglichkeit zur Konfiguration der genannten Subjekte, Objekte, Zugriffsregeln und Tickets für den ROP bieten (Anforderung A2.2). Hierfür soll SAM eine REST-API bereitstellen, die ROP über HTTP und ein Webinterface benutzen kann. Außerdem muss SAM über CoAP und DTLS mit den von ihm verwalteten Servern kommunizieren können, um diesen Informationen zu widerrufenen Tickets mitzuteilen oder um neues Schlüsselmaterial, dem in Abschnitt 3.7 definierten

Übergangsszenario entsprechend, auszutauschen.

Hierbei sollen die einzelnen Komponenten von SAM möglichst modular entworfen werden, sodass ein flexibler Austausch und eine Wiederverwendung einzelner Module möglich ist. Im Folgenden werden für die oben genannten Aufgaben Entscheidungen entsprechend dem Nutzungsszenario getroffen.

4.6.1 Verwaltung und Speicherung von Daten

Um Subjekte, Objekte, Zugriffsregeln und Tickets verwalten und verwenden zu können, müssen diese auf SAM in einem nicht flüchtigen Speicher gehalten werden. Um die Datenhaltung flexibel zu gestalten, soll zwischen Anwendungslogik und Datenhaltung eine Abstraktionsschicht eingeführt werden, die nach dem DAO-Entwurfsmuster[8] einen einfachen Austausch der Datenhaltungsmethode ermöglicht. Bei einem Austausch muss keine Anpassung der Anwendungslogik erfolgen, das heißt die Verwendung erfolgt transparent.

Für den produktiven Einsatz bietet sich der Einsatz einer Datenbank (SQL oder NOSQL) für die Persistenzschicht an. Da dies aber zu weiteren Abhängigkeiten der Software führt und im verwendeten Nutzungsszenario nur wenige Daten anfallen, werden diese in Textdateien im JSON-Format gespeichert werden. Dies wurde gewählt, da Daten vom ROP bei der Konfiguration über die, in Abschnitt 4.6.7 beschriebene, REST-API und das, ebenfalls in Abschnitt 4.6.7 beschriebene, Webinterface bereits im JSON-Format vorliegen und an SAM gesendet werden und diese somit zur Speicherung nicht mehr konvertiert werden müssen. Um die Anzahl nötiger Dateizugriffe zu verringern, werden die Daten von der JSON-Persistenzschicht im Arbeitsspeicher zwischengespeichert, sodass nur bei schreibenden Operationen Dateizugriffe durchgeführt werden müssen. Details zur Umsetzung des Entwurfsmusters und der Datenhaltung finden sich in Kapitel 5.

4.6.2 Autorisierung und Zugriffsregeln

Subjekte

Zur Durchführung einer Autorisierung auf SAM werden Subjekte und Objekte benötigt. Da SAM in DCAF keine Informationen zur Authentifizierung der auf Seiten des COPs verwendeten Clients besitzt und ihm auch keine durch einen CAM mitgeteilt werden, kann er diese nicht als Subjekte der Autorisierung verwenden. Stattdessen werden CAMs als Subjekte verwendet. Im Nutzungsszenario entspricht dies dem nicht eingeschränkten Autorisierungsmanager auf Seiten des Transportunternehmens. In Abschnitt 3.8 wurde beschrieben, dass COP und ROP im Szenario bereits zuvor X.509-Zertifikate zur Authentifizierung und Verschlüsselung ausgetauscht haben. Daher können diese hier verwendet werden. SAM speichert folgende Informationen über jeden CAM als Subjekt der Autorisierung:

```
struct subject {
```

[8] The DAO Design Pattern: http://tutorials.jenkov.com/java-persistence/dao-design-pattern.html (abgerufen am 08. Juni 2015)

```
    char *cert_fingerprint;
    char *name;
};
```

Das Feld `cert_fingerprint` enthält einen SHA1-Fingerprint des X.509-Zertifikats des CAMs. Dieser wird zur eindeutigen Identifizierung der CAMs verwendet und wird von ROP während des Hinzufügens eines CAMs angegeben. Weitere Informationen zur Verwendung des Fingerprints sind in Unterabschnitt 4.6.4 und Einzelheiten zur Authentifizierung anhand des Zertifikats in Abschnitt 5.3 zu finden. Das Feld `name` enthält einen informellen textuellen Bezeichner für den CAM. Dieser wird von ROP beim Hinzufügen des CAMs angegeben und dient ausschließlich dem Zweck, ROP an Stelle des kryptischen Fingerprints zur Konfiguration einen lesbaren Namen präsentieren zu können. Insbesondere wird das Feld `name` nicht zur Authentifizierung des CAMs verwendet.

Weitere Informationen über die Subjekte sind für das Nutzungsszenario und das verwendete Autorisierungsmodell nicht notwendig. Bei der Verwendung anderer Autorisierungsmodelle ist unter Umständen die Speicherung weiterer Daten notwendig. So müssten beispielsweise für eine rollenbasierte Autorisierung die Rollenzugehörigkeiten der jeweiligen CAMs gespeichert werden.

Objekte

Als Objekte der Autorisierung muss eine Menge von Servern und die auf ihnen vorhandenen Ressourcen verwaltet werden. Die Server werden vom ROP im Sinne des in Unterabschnitt 4.6.6 festgelegten Ablaufs in seine Sicherheitsdömane eingeführt und SAM bekannt gemacht. SAM speichert folgende Eigenschaften zu jedem Server:

```
struct server {
  char *host;
  char *secret;
  uint32 next_seq_nr;
  uint32 rs_state_lowest_seq;
  LIST of char* conditions;
  LIST of resources;
};

struct resource {
  char *resource;
  int methods;
};
```

Das Feld `host` gibt den Hostname oder eine IPv6-Adresse des Servers an. Das Feld wird intern auf SAM zur Identifizierung des Servers verwendet und muss daher eindeutig sein. Das `secret` enthält den mit dem Server geteilten Schlüssel $K(SAM, S)$. Dieser wird für das Ausstellen von Tickets und für die direkte Kommunikation mit dem Server benötigt. Im feld `next_seq_nr` wird die Sequenznummer für das nächste, für diesen Server auszustellende, Ticket festgehalten, welche mit jedem neu ausgestellten Ticket inkrementiert wird. Außer zur Identifizierung soll die Sequenznummer auch zum Widerrufen von Tickets verwendet werden. Der Server verwendet hierzu einen Sliding-Window-Mechanismus, um widerrufene Tickets zu verwalten. Details zum verwendeten Sliding-Window auf dem Server finden sich in Abschnitt 4.7. Damit SAM

dem zuständigen ROP mitteilen kann, welche Tickets vom Server noch akzeptiert werden, muss er parallel zum Server den Sliding Window Algorithmus für jeden Server nachbilden können. Hierfür merkt er sich im Feld `rs_state_lowest_seq` die untere Grenze des vom Server verwendeten Fensters. Beim Widerrufen von Tickets wird diese gegebenenfalls angepasst. Details zur Umsetzung finden sich in Kapitel 5.

Zusätzlich verwaltet SAM für jeden Server im Feld `conditions` eine Liste mit, vom Server unterstützten, lokalen kontextbasierten Bedingungen, um Anforderung A2.16 zu erfüllen. Diese sollen ermöglichen, dass beispielsweise der Zugriff nur zu bestimmten Tageszeiten oder nur an bestimmten Orten erlaubt sein soll. In dieser Arbeit soll für das Nutzungsszenario lediglich die Einschränkung auf bestimmte Tageszeitintervalle möglich sein. In der Liste des Feldes `conditions` sollen Zeichenketten als Key verwendet werden, welche jeweils eine von diesem Server unterstützte Bedingung beschreiben und identifizieren. Für die Einschränkung nach Zeitintervallen soll die Zeichenkette `timeframe` verwendet werden. Die Bedingungen können in den Zugriffsregeln verwendet werden (siehe Abschnitt 4.6.2) und werden zur Durchsetzung auf dem Server mit in das Ticket geschrieben (siehe Unterabschnitt 4.6.3). Die Durchsetzung der Bedingungen wird in Abschnitt 4.7 beschrieben.

SAM speichert außerdem im Feld `resources` zu jedem Server eine Liste mit Ressourcen, die von diesem Server verwaltet werden. Zu jeder Ressource wird zum einen der URI als Zeichenkette und zum anderen die auf diese Ressource möglichen CoAP-Methoden gespeichert. Diese werden benötigt, um in Zugriffsregeln nicht mehr Methoden zu erlauben, als vom Server unterstützt werden. Für die Methoden wird das, in [Bor15] vorgeschlagene, AI-Format verwendet. Mit diesem kann eine Kombination von CoAP-Methoden in einer Ganzzahl ausgedrückt werden. Hierbei wird jede erlaubte Methode als Zweierpotenz ihres *CoAP Method Code* Minus 1 und mit den anderen erlaubten Methoden durch ein inklusives ODER verknüpft. Folgendes Beispiel verdeutlicht das Vorgehen: Angenommen die Ressource soll abgefragt (GET; Method-Code 1) und gelöscht (DELETE; Method-Code 4) werden können. Dies würde, wie anhand der folgenden Rechnung zu erkennen ist, zusammengefasst die Zahl 9 ergeben:

$$2^{(1-1)} \vee 2^{(4-1)} = 1 \vee 8 = 0b0001 \vee 0b1000 = 0b1001 = 9 \tag{4.1}$$

Durch dieses Vorgehen ist zum einen eine kompakte Darstellung einer Menge von Methoden möglich, welche sich daher insbesondere auch für die Verwendung auf eingeschränkten Geräten eignet. Zum anderen ist auch die Auswertung der Methoden bei der Zugriffskontrolle in dieser Darstellung besonders einfach, da hierfür nur eine UND-Verknüpfung von zwei Methoden-Mengen notwendig ist. Details hierzu finden sich in Unterabschnitt 4.6.4. Diese Art der Repräsentation von CoAP-Methoden soll im gesamten Szenario verwendet werden, so dass diese beispielsweise auch im Ticket in diesem Format vorliegen.

Wie in Abschnitt 3.8 beschrieben, würde SAM die Informationen über die, auf dem Server vorhandenen, Ressourcen in der Regel über einen Verzeichnisdienst abfragen. In dieser Arbeit wird davon ausgegangen, dass dieser Schritt bereits stattgefunden hat und SAM im Szenario bereits die Informationen über die in den Bananenschachteln enthaltenen Server und ihre Ressourcen besitzt.

Zugriffsregeln

Die von SAM verwalteten Subjekte und Objekte können in Zugriffsregeln miteinander verknpüft werden, um eine Autorisierung zu ermöglichen. DCAF selbst trifft keine Entscheidung über das auf SAM zu verwendende Autorisierungsmodell, sondern legt als Framework lediglich die für die Kommunikation notwendigen Protokolle, Nachrichteninhalte und den auf Zugriffsausweisen basierenden Mechanismus zum Durchsetzen von Autorisierungsinformationen fest. Das Autorisierungsmodell muss dementsprechend je nach Einsatzzweck ausgewählt werden.

Im Grunde ist hierbei der Einsatz sämtlicher in Abschnitt 2.7 vorgestellter Autorisierungsmodelle möglich. So kann beispielsweise das DAC-Modell mit Access Control Matrix verwendet werden, in der für jede Subjekt/Objekt-Kombination Zugriffsrechte festgehalten werden. Für eine große Anzahl an Subjekten oder Objekten können diese aber schnell lang und schlecht wartbar werden. Des Weiteren würde ein solches, auf Access Control Lists (ACL) basierendes, Modell keine feingranulare Autorisierung unter der Verwendung von lokalen Bedingungen und Ablaufdatum ermöglichen. Alternativ wäre auch der Einsatz eines rollenbasierten Ansatzes möglich, bei dem Zugriffsrechte nicht für Subjekte direkt, sondern für Rollen vergeben werden, in denen Subjekte Mitglied sein können. Da ACLs alleine zu unflexibel sind und eine rollenbasierte Autorisierung im Szenario mit nur wenigen Subjekten keine Vorteile bietet, wird dieses Modell nicht verwendet. Stattdessen wird ein regelbasierter Ansatz verwendet, bei dem in der Standardeinstellung kein Zugriff möglich ist und abweichend davon Positiv-Regeln angelegt werden können, die Ausnahmen festlegen. Eine Regel legt hierbei fest, auf welche Ressourcen ein Subjekt mit welchen CoAP-Methoden, unter welchen Bedingungen zugreifen darf. Positiv-Regeln werden auf SAM als Datenstruktur mit folgenden Eigenschaften verwaltet:

```
struct rule {
  char *id;
  char *subject;
  LIST of rule_resources;
  LIST of rule_condition;
  date expiration_date;
  int priority;
};

struct rule_resource {
  char *server;
  char *resource;
  int methods;
};

struct rule_condition {
  char *key;
  void *data;
};
```

Um eine Zugriffsregel im System zu identifizieren, trägt jede Regel einen, lokal auf SAM eindeutigen, Bezeichner im Feld `id`. Der Bezeichner kann von ROP beim Erstellen der Regel festgelegt werden. SAM akzeptiert diesen allerdings nur, wenn er bisher im System noch nicht verwendet wird. Das Feld `subject` gibt CAM anhand des SHA1-Fingerprints seines Zertifikats eindeutig an. Da eine Autorisierung von Gruppen von

Subjekten im Nutzungsszenario nicht vorgesehen ist, ist in einer Regel jeweils nur die Angabe von einem Subjekt möglich. Durch die Angabe des Subjekts in einer Regel wird die in Anforderung A2.8 geforderte Möglichkeit, verschiedene Zugriffsrechte für unterschiedliche Subjekte zu setzen, erfüllt.

Außerdem enthält jede Regel eine Liste an Ressourcen, für die der Zugriff für das Subjekt gewährt werden soll. Hierbei wird, zusätzlich zu den, bei den Ressourcen des Servers verwendeten, Angaben der URI und der CoAP-Methoden, noch die, zur Identifizierung verwendete, `id` des Servers gespeichert. Der URI und die erlaubten CoAP-Methoden liegen ebenfalls im, in Abschnitt 4.6.2 beschriebenen, AI-Format vor. Allerdings geben die CoAP-Methoden diesmal nicht an welche CoAP-Methoden der Server auf die Ressourcen unterstützt, sondern welche Aktionen das Subjekt auf die angegebene Ressource ausführen darf. In einer Regel dürfen nicht mehr CoAP-Methoden erlaubt werden, als der Server für die Ressource unterstützt. Hierzu vergleicht SAM beim Hinzufügen einer Regel die beiden Methoden-Mengen und lehnt das Anlegen der Regel ab, wenn diese zu viele Rechte fordert. (siehe Unterabschnitt 4.6.4). Durch die Speicherung als Liste von Ressourcen kann mit einer Regel der Zugriff auf verschiedene Ressourcen und Server erlaubt und somit Anforderung A2.9 erfüllt werden.

Zusätzlich kann im Feld `conditions` eine Liste mit lokalen kontextbezogenen Bedingungen angegeben werden. Diese werden bei der Autorisierung auf SAM nicht ausgewertet, sondern in einem reduzierten Format ins Ticket geschrieben, sodass sie bei der Zugriffskontrolle auf dem Server unter den lokalen Bedingungen des Servers ausgewertet werden können. Für jede Bedingung wird der in den Objekten der Autorisierung festgelegte Key zur Beschreibung der Bedingung gespeichert. Zusätzlich müssen noch die, für die Bedingung benötigten, Daten gespeichert werden. Bei der, im Nutzungsszenario verwendeten, Bedingung `timeframe` entspricht dies dem erlaubten Zeitintervall. Die Datentypen von verschiedenen Bedingungen können sich unterscheiden. Der verwendete Datentyp wird über den Key festgelegt. Für die Bedingung `timeframe` müssen zwei Uhrzeiten, zwischen denen der Zugriff erlaubt werden soll, festgehalten werden. Hierfür wird ein zweidimensionales Array mit zwei Elementen, in der Form `[[8,30],[18,15]]`, für ein Zeitintervall von 08:30 Uhr bis 18:15 Uhr, verwendet. Die Uhrzeit wird hierbei im 24-Stunden-Format angegeben. Da davon ausgegangen wird, dass SAM und der Server synchronisierte Uhren verwenden, kann auch der Server die Intervalle interpretieren.

Nach Anforderung A2.15 soll es möglich sein, nur temporären Zugriff auf Ressourcen zu gewähren. DCAF sieht hierfür die Angabe einer Lifetime im Ticket vor, die die Gültigkeit eines Tickets in Sekunden angibt. Auf SAM soll ein hybrider Ansatz für die Lifetime verwendet werden. Zum einen kann ROP global in der Konfiguration von SAM eine Lifetime angeben, die zunächst für alle ausgestellten Tickets gilt. Allerdings kann diese globale Einstellung durch Zugriffsregeln überschrieben werden. Wird beispielsweise ein Ticket aus einer Regel erstellt, die eine abweichende Gültigkeit vorsieht, so wird diese verwendet. Anderenfalls wird die globale Lifetime verwendet. Die Lifetime in der Zugriffsregel wird anders als im Ticket nicht in Sekunden, sondern relativ, in Form eines Datums, angegeben. Hierdurch kann der ROP exakt festlegen, bis zu welchem Datum aus dieser Regel ausgestellte Tickets genutzt werden dürfen. Das Datum wird beim Ausstellen des Tickets in die noch verbleibenden Sekunden bis zu diesem Zeitpunkt umgewandelt.

Abschließend können Regeln noch mit Prioritäten versehen werden. Diese werden durch Zahlen ausgedrückt: Je größer die Zahl, desto höher die Priorität für diese Regel. Bei der Auswertung von Zugriffsregeln kann der Fall eintreten, dass mehrere Regeln den Zugriff auf die angeforderten Ressourcen ermöglichen. Um in diesen Fällen eine Regel bevorzugen zu können, wird die Priorität verwendet. Falls nur die Auswahl zwischen Regeln mit gleicher Priorität möglich ist, wird die zuletzt angelegte Regel zur Autorisierung und Erstellung des Tickets verwendet.

In manchen Anwendungsfällen sind feingranulare Zugriffsregeln, wie sie oben definiert wurden, nicht nötig, sondern es reicht, wie in Anforderung A2.7 beschrieben, den Zugriff auf einen Server im Ganzen zu erlauben oder zu unterbinden (implizite binäre Autorisierung). Um dies zu ermöglichen, können Zugriffsregeln auch in einer vereinfachten Form von ROP angegeben werden. Hierbei fällt das Feld `conditions` in den Regeln weg und die Angabe der Ressourcen ändert sich. Statt, wie beschrieben, die Ressource als URI explizit anzugeben, wird diese im Falle einer impliziten Autorisierung mit der Zeichenkette '`*`' angegeben. Diese wird von SAM bei der Auswertung als Sonderfall betrachtet, sodass ein Ticket für den impliziten Zugriff erstellt werden kann. Da bei der impliziten Autorisierung auch die CoAP-Methoden keiner Beschränkung unterliegen sollen, werden diese so angegeben, dass alle Methoden erlaubt werden (als Zahl: 15).

Die für das Nutzungsszenario festgelegten Zugriffsregeln werden in Kapitel 6 erläutert.

4.6.3 Tickets

Die ausgestellten Tickets werden auf SAM ähnlich der Repräsentation in der, in Abschnitt 4.4, definierten Ticket Grant Message gespeichert. Folgendes Format wird auf SAM für die Tickets verwendet:

```
struct dcaf_ticket {
  char *id;
  struct dcaf_ticket_face face;
  unsigned char verifier[32];
  size_t verifier_size;
};

struct dcaf_ticket_face {
  struct AIF AI;
  int timestamp;
  int lifetime;
  int dtls_psk_gen_method;
  uint32_t sequence_number;
  LIST of conditions;
};
```

Da auf SAM die Distributed Key Derivation zur Schlüsselerzeugung verwendet werden soll, wird auf die Angabe eines Schlüssels im Ticket verzichtet. Zusätzlich zu den bereits in Abschnitt 4.4 vorgestellten Eigenschaften ist auf SAM im Ticket eine `id` hinzugekommen, mit der das Ticket lokal auf SAM identifiziert werden kann. Die Sequenznummer im Ticket-Face reicht hierfür nicht aus, da sie nur für jeden Server, nicht aber global auf SAM eindeutig ist. SAM führt anhand der Ticket-ID keine Authentifizierung durch, sondern verwendet sie ausschließlich, um Tickets bei der Kon-

figuration durch ROP identifizieren zu können. Im Ticket-Face sind außerdem die, in Abschnitt 4.6.2 definierten, lokalen Bedingungen zur Auswertung auf dem Server hinzugekommen. Detaillierte Informationen zu den einzelnen Eigenschaften im Ticket finden sich im folgenden Unterabschnitt 4.6.4.

4.6.4 Ausstellen von Tickets

SAM muss Ticket Request Messages von CAMs verarbeiten können (Anforderung A2.4). Hierfür stellt SAM über einen HTTP-Server eine Ressource unter der URI `/ep` bereit, auf die CAM einen HTTP POST-Request ausführen kann. Diese Anfrage wird über TLS 1.2 verschlüsselt und CAM sendet während des TLS-Handshakes sein, von der Certification Authority des ROPs signiertes, Zertifikat mit. Anhand des Zertifikats findet auf SAM die Authentifizierung des CAMs auf TLS-Ebene statt.

CAM sendet die in Abschnitt 4.4 festgelegten Daten als Payload im CBOR-Format in seiner Anfrage mit. SAM überprüft zunächst, ob diese im richtigen Format und vollständig sind. Anschließend berechnet SAM den SHA1-Fingerprint des von CAM verwendeten Zertifikats und verwendet diesen, um die zu diesem CAM gespeicherten Zugriffsregeln auszuwählen. Die Zugriffsregeln werden zur Auswertung in der Reihenfolge ihrer Priorität durchlaufen, wobei mit der höchsten Priorität begonnen wird. Für jede Zugriffsregel finden folgende Auswertungsschritte statt:

1. SAM prüft, ob die in der Ticket Request Message angefragten Ressourcen, auf den angefragten Servern, in der Zugriffsregel enthalten sind. Falls in der Ticket Request Message mehrere Ressourcen auf einmal autorisiert werden sollen, müssen diese Ressourcen alle in einer Zugriffsregel erfasst sein. Außerdem muss die Zugriffsregel dahingehend untersucht werden, ob sie einen impliziten Zugriff auf diesen Server erlaubt.
2. SAM überprüft, ob die in der Ticket Request Message geforderten CoAP-Methoden auf die Ressourcen von der Regel gedeckt sind.
3. Es muss überprüft werden, ob in der Zugriffsregel eine zur globalen Gültigkeitsdauer abweichende Lifetime angegeben wurde. Wenn der darin enthalene Zeitstempel in der Vergangenheit liegt, ist die Zugriffsregel ebenfalls abzulehnen, da ein ausgestelltes Ticket aus dieser Regel keine Gültigkeit besitzen würde. Hierdurch können Regeln auf SAM mit einem impliziten Ablaufdatum versehen werden.

Falls keine Regel die oben genannten Bedingungen erfüllt, ist die Ticket Request Message von SAM mit dem HTTP-Statuscode `401` (Unauthorized) abzuweisen. Erfüllt eine Zugriffsregel alle Bedingungen, so gilt die Anfrage als autorisiert und SAM kann ein Ticket ausstellen, dieses zur Verwaltung speichern und es als Antwort auf die Ticket Request Message an CAM senden.

Das Ticket wird in der in Unterabschnitt 4.6.3 festgelegten Struktur auf SAM angelegt. Zunächst wird hierbei das Ticket-Face konstruiert. In dieses werden die Autorisierungsinformationen (AI), wie in [Bor15] definiert, abgelegt. Da die Autorisierungsinformationen bereits in der Anfrage dieses Format hatten, können die Informationen aus dieser für das Ticket verwendet werden. SAM generiert einen Zeitstempel der aktuellen, mit dem Servern synchronisierten, Zeit und speichert diesen ebenfalls im Ticket-Face.

Falls die Zugriffsregel, die den Zugriff erlaubt beziehungsweise aus der das Ticket erstellt wird, ein Datum als Lifetime enthält, so berechnet SAM die bis zu diesem Datum noch verbleibenden Sekunden und schreibt diese als Lifetime ins Ticket-Face. Anderenfalls wird die global auf SAM konfigurierte Lifetime verwendet. Außerdem wird im Ticket-Face hinterlegt, welche Methode zur Schlüsselerzeugung verwendet wird. Da in dieser Arbeit ausschließlich die Distributed Key Derivation verwendet werden soll, hat dieses Feld immer den gleichen Wert 0, der für `DTLS_PSK_GEN_HMAC_SHA256` steht. Des Weiteren muss eine Sequenznummer in das Ticket-Face geschrieben werden, damit der Server Tickets unterscheiden und identifizieren kann. SAM merkt sich beim Ausstellen eines Tickets die für den Server zuletzt verwendete Sequenznummer und inkrementiert diese für jedes neue Ticket. Abschließend übernimmt SAM für das Ticket die lokalen Bedingungen, die unter Umständen in der Regel enthalten sind, zunächst unverändert.

Falls die Zugriffsregel eine implizite Autorisierung, ohne feingranulare Bedingungen, festlegt, werden keine Autorisierungsinformationen und keine lokalen Bedingungen ins Ticket geschrieben. Die entsprechenden Felder fehlen bei impliziter Autorisierung.

Das Ticket ist zur Berechnung des Verifiers, zum Versand des Tickets an CAM und zur Verwendung auf dem entsprechenden Server aufzubereiten. Hierzu muss das Ticket in das CBOR-Format umgewandelt werden. Die Daten werden im Ticket-Face weitestgehend unverändert in die entsprechenden CBOR-Datentypen umgewandelt. Lediglich die Keys der CBOR-Maps werden reduziert. Statt einer Zeichenkette zur Identifizierung werden kurze CBOR-Numbers als Key verwendet. Hierbei wird die von DCAF festgelegte Zuordnung verwendet und um Keys für lokale Bedingungen und Sequenznummern ergänzt. Sobald das Ticket im CBOR-Format vorliegt, kann aus dem Ticket-Face der Verifier mit dem festgelegten HMAC-Algorithmus und dem, von SAM und dem Server geteilten, Schlüssel $K(SAM, S)$ generiert werden. Das Face muss hierbei in CBOR-Repräsentation verwendet werden, da der Server das Face in diesem Format vom Client erhält und nur auf Grundlage dessen den für die sichere Verbindung nötigen Schlüssel generieren kann. Die Zufälligkeit und Eindeutigkeit des Verifiers wird durch die immer unterschiedliche Sequenznummer im Ticket-Face gewährleistet.

Anschließend kann das Ticket im CBOR-Format an CAM gesendet und vom Client für den Zugriff auf den Server verwendet werden. Die CBOR-Datenstruktur darf während des Transports bis zum Server nicht verändert werden. So darf beispielsweise auch die Reihenfolge der Attribute nicht verändert werden, da der Server nur aus exakt den von SAM generierten CBOR-Daten den korrekten Verifier generieren kann. Beispieltickets, die entsprechend dem Nutzungsszenario Zugriff erlauben, finden sich im CBOR- und JSON-Format in Kapitel 6.

4.6.5 Widerrufen von Tickets

Einmal ausgestellte Tickets sollen, wie in Anforderung A2.15 gefordert, sowohl implizit als auch explizit widerrufen werden können. Das implizite Widerrufen wird durch die im Ticket vorhandene Lifetime erreicht. Sobald die Lifetime des Tickets abgelaufen ist, verliert das Ticket automatisch seine Gültigkeit, da beim Zugriff auf dem Server erkannt wird, dass das Ticket abgelaufen ist und der Server die Anfrage ablehnt.

Das explizite Widerrufen wird durch Aktionen des ROPs über die REST-API beziehungsweise das Webinterface ausgelöst. Eine solche Aktion kann zum eine sein, dass

ROP explizit ein ausgestelltes Ticket zurückziehen will. Zum anderen ist es möglich, dass ROP eine Zugriffsregel auf SAM ändert und durch die Änderung ein bereits ausgestelltes Ticket, nach der veränderten Zugriffsregel nicht mehr ausgestellt werden dürfte. Dafür werden ROP nach dem Bearbeiten einer Zugriffsregel, die von der Änderung betroffenen Tickets aufgelistet und er hat die Möglichkeit auszuwählen, ob diese widerrufen werden oder bis zum Ende ihrer Lifetime weiterhin gültig bleiben.

Beim expliziten Widerrufen eines Tickets wird das Ticket aus den von SAM verwalteten Tickets über die Datenabstraktionsschicht entfernt und in einer von SAM verwalteten Revocation-List gespeichert. Einträge in dieser Liste enthalten, neben dem widerrufenen Ticket, folgende Daten:

```
struct dcaf_revocation {
  struct dcaf_ticket ticket;
  int delivery_time;
  int last_try;
  int tries;
};
```

SAM muss die widerrufenen Tickets den betroffenen Servern bekannt machen. In der dezentralen Architektur muss der Server zur Zugriffszeit autonom die Zugriffskontrolle durchsetzen und hat währenddessen keine Möglichkeit Rücksprache mit seinem Autorisierungsmanager zu halten. Daher muss der Server möglichst vor einem Zugriff über das Widerrufen eines Tickets informiert werden. SAM versucht dazu in regelmäßigen Abständen die entsprechenden Server zu kontaktieren und ihnen eine Ticket Revocation Message, wie in Abschnitt 4.4 spezifiziert, zukommen zu lassen. Die Verbindung wird über DTLS und den Schlüssel $K(SAM, S)$ gesichert. Gelingt dieser Verbindungsversuch nicht, wird eine bestimmte Zeit gewartet und anschließend ein erneuter Versuch gestartet. Die Zeit zwischen Versuchen verdoppelt sich jedes Mal, sodass ein sogenannter exponentieller Backoff Timer entsteht. Hierfür werden die Felder `last_try` und `tries` in der Revocation-List verwendet, welche einen Zeitstempel des letzten Verbindungsversuch und die Anzahl der bisher erfolgten Versuche angeben. Hierdurch ist die Berechnung der Zeit zwischen zwei Verbindungsversuchen in Abhängigkeit der Anzahl der Versuche möglich. Die maximale Wartezeit zwischen zwei Versuchen kann von ROP festgelegt werden. In der Standardeinstellung wird maximal 600 Sekunden beziehungsweise zehn Minuten zwischen zwei Versuchen gewartet und mit einer Wartezeit von zwei Sekunden gestartet. Dieses Vorgehen wird verwendet, um den Server nicht mit Anfragen zu überhäufen. Sobald die Ticket Revocation Message erfolgreich zugestellt werden konnte und der Server den Erhalt bestätigt hat, markiert SAM das Ticket in der Revocation-List als erfolgreich übertragen, indem er im Feld `delivery_time` einen Zeitstempel der Übermittlung des Widerrufs speichert. Außerdem berechnet SAM den auf dem Server verwendeten und in Unterabschnitt 4.7.1 beschriebenen Sliding-Window-Algorithmus für den Server parallel zu diesem, um dem ROP jederzeit über das Webinterface (siehe Abschnitt 4.6.7) kenntlich machen zu können, welche ausgestellten Tickets vom Server noch akzeptiert werden. Hierfür speichert SAM zu jedem Server (siehe Abschnitt 4.6.2) die untere Grenze des Sliding Windows des Servers im Feld `rs_state_lowest_seq`, wobei die Server alle Sliding Windows mit einer Größe von 32-Einträgen verwenden und dies SAM bekannt ist. Nach der erfolgreichen Zustellung einer Ticket Revocation Message vergleicht SAM die Sequenznummer vom widerrufenen Ticket mit dem Feld `rs_state_lowest_seq` und überprüft, analog

zum beschriebenen Vorgehen in Unterabschnitt 4.7.1, ob die Sequenznummer innerhalb des Fensters liegt oder ob dieses verschoben werden muss.

Im DCAF-Draft wird das explizite Widerrufen von Tickets und Revocation Messages zwar erwähnt, allerdings wird hier noch keine detaillierte Definition vorgenommen. Es wird aber gefordert, dass bei der Verwendung von Revocation Messages eine ständige Verbindung von SAM zum Server bestehen muss. Wenn der Server seinen Autorisierungsmanager nicht erreichen kann, muss er nach DCAF alle bestehenden Verbindungen zu Clients abbrechen und keine neuen Anfragen mehr annehmen, bis wieder eine Verbindung zum Autorisierungsmanager hergestellt werden kann. Dies würde allerdings den großen Vorteil der dezentralen Architektur, dass Server und Autorisierungsmanager zur Zugriffszeit nicht kommunizieren müssen, untergraben.

Auch im Nutzungsszenario wäre dieses Vorgehen problematisch, da hier während des Transports definitiv keine permanente Kommunikation zwischen Server und SAM möglich ist. Daher sollen in dieser Arbeit weniger strenge Anforderung an die Erreichbarkeit bei Verwendung von Revocation Messages gestellt werden. Eine permanente Erreichbarkeit soll nicht verlangt werden. Stattdessen soll in Kauf genommen werden, dass widerrufene Tickets unter Umständen noch bis zum Ende ihrer Lifetime verwendet werden können. Hierbei ist mit der Wahl einer möglichst kurzen, aber für den Anwendungszweck ausreichenden, Lifetime eine Abwägung zwischen Gültigkeit des Tickets und der Zeitspanne, die ein widerrufenes Ticket noch verwendet werden kann, zu treffen. Dieses Vorgehen findet sich auch bei anderen weit verbreiteten verteilten Systemen zur Authentisierung und Autorisierung. Beispielsweise verwendet Kerberos Tickets mit einer Gültigkeit von standardmäßig 8 Stunden und hat dementsprechend eine ebenso große maximale Zeitspanne, in der widerrufene Tickets noch verwendet werden können, auch wenn der Zugriff auf dem Kerberos-Server bereits unterbunden wurde [KS07, S.20]. Da im Szenario während des Transports auf dem Meer keine Kommunikation der eingeschränkten Geräte mit ihren Autorisierungsmanagern erfolgen kann, muss für das Szenario eine Lifetime gewählt werden, die für den gesamten Transport ausreichend ist. Der Transport eines Containers von Costa Rica nach Deutschland benötigt etwa zwei Wochen [Jed+13]. Die Lifetime müsste dementsprechend mindestens diesen Wert haben. Idealerweise sollte aber noch etwas Puffer übrig sein, falls sich der Transport verzögert.

4.6.6 Unterstützung des Übergangsszenarios

In Abschnitt 3.7 wurde beschrieben, dass die Zuständigkeit für eingeschränkte Geräte während ihres Lebenszyklus' mehrfach wechseln kann und sie hierdurch in verschiedenen Sicherheitsdomänen verwendet werden. In Bezug auf DCAF bedeutet dies, dass die Autorisierungsmanager der eingeschränkten Geräte beim Übergang in eine neue Sicherheitsdomäne wechseln. In dieser Arbeit wird der Übergang eines eingeschränkten Servers in eine neue Sicherheitsdomäne thematisiert und eine Lösung für die hierbei nötige Kommunikation umgesetzt werden. Bei dem Übergang eines Servers in eine neue Sicherheitsdomäne muss sowohl eine Außerbetriebnahme als auch eine Wiederinbetriebnahme stattfinden.

Bei der Außerbetriebnahme wird die Sicherheitsbeziehung zwischen dem Server und dem Autorisierungsmanager, in der vorherigen Sicherheitsdomäne, aufgelöst. Hierdurch

wird zum einen erreicht, dass der vorherige Besitzer nicht mehr auf den Server zugreifen kann und zum anderen, dass der ROP in der neuen Sicherheitsdomäne keine schützenswerten Daten, die auf dem Server vor der Außerbetriebnahme gespeichert waren, einsehen kann (Anforderung A6.1). Bei der Wiederinbetriebnahme wird der Server in die neue Sicherheitsdomäne eingeführt und Autorisierungsmanager und Server werden sich gegenseitig bekannt gemacht.

In DCAF können die Außerbetriebnahme und die Wiederinbetriebnahme in einem Schritt durchgeführt werden, indem dem Server ein neuer Schlüssel $K(SAM, S)$ zugewiesen wird. Hierdurch kann der Autorisierungsmanager der vorherigen Sicherheitsdomäne keine gültigen Tickets mehr für den Server ausstellen und bisher ausgestellte Tickets werden vom Server nicht mehr angenommen, da bei der Berechnung des DTLS-PSK bereits der neue Schlüssel $K(SAM, S)$ verwendet wird und somit kein zum Verifier des Clients identischer DTLS-PSK erzeugt werden kann. Durch die Zuweisung eines neuen Schlüssels $K(SAM, S)$ durch den Autorisierungsmanager der neuen Sicherheitsdomäne findet die Inbetriebnahme in dieser statt und die Sicherheitsbeziehung zwischen dem Autorisierungsmanager und dem Server wird hergestellt.

In [Ger15b] werden verschiedene Mechanismen vorgeschlagen, die einen solchen Austausch des, zwischen dem Autorisierungsmanager und einem eingeschränkten Gerät geteilten, Schlüssels in einer DCAF-Architektur ermöglichen. [Ger15b] schlägt vor, dass der Schlüssel auf dem eingeschränkten Gerät als CoAP-Ressource unter dem URI `/am/key` bereitgestellt wird. Auf diesem soll nur schreibend zugegriffen werden (POST-Request). Neben dem, mit dem Autorisierungsmanager geteilten, Schlüssel muss dem eingeschränkten Server auch der URI des neuen Autorisierungsmanagers, unter der dieser Ticketanfragen entgegen nimmt, mitgeteilt werden. Der Server benötigt diesen URI, um einem Client diese in der *SAM Information Message* bekannt zu machen. [Ger15b] sieht vor, dass der SAM-URI ebenfalls als CoAP-Ressource unter der URI `/am/uri` vom eingeschränkten Gerät bereit gestellt wird.

Für den sicheren Austausch des neuen Schlüssels, unter der Wahrung der Sicherheitsinteressen des vorherigen und neuen Besitzers, sieht [Ger15b] die Verwendung von Tickets vor. Der eingeschränkte Server kann Tickets bereits durch den DCAF-Protkollablauf verarbeiten und eine Zugriffskontrolle durchsetzen, daher benötigt er bei der Verwendung von Tickets zum Austauschen des Schlüssels keine weiteren Sicherheitsmechanismen. Der Autorisierungsmanager hat bei diesem Austausch die Rolle des DCAF-Clients inne, der das zum Zugriff auf die CoAP-Ressourcen `/am/key` und `/am/uri` nötige Ticket während des DTLS-Verbindungsaufbaus an den Server sendet. Da zum Zeitpunkt des Verbindungsaufbaus auf dem Server noch der alte Schlüssel $K(SAM, S)$ vorliegt, muss das Ticket auf Grundlage dieses Schlüssels erstellt werden.

In [Ger15b] werden drei Möglichkeiten vorgeschlagen, um dem Autorisierungsmanager der neuen Sicherheitsdomäne das benötigte Ticket bekannt zu machen. Die erste Möglichkeit sieht vor, dass der Autorisierungsmanager das Ticket direkt beim Autorisierungsmanager der vorherigen Sicherheitsdomäne anfordert. Hierfür müsste der Autorisierungsmanager der neuen Sicherheitsdomäne die Rolle des DCAF-CAM übernehmen und Ticketanfragen senden können. Allerdings muss hierfür eine Sicherheitsbeziehung, etwa in Form von Zertifikaten, zwischen den Autorisierungsmanagern bestehen.

Die zweite Möglichkeit sieht die Verwendung eines vorkonfigurierten Tickets vor. Der Autorisierungsmanager der alten Sicherheitsdomäne kann dieses Tickets bereits zur

Zeit der Inbetriebnahme des Servers ausstellen, es aufbewaren und dieses dem ROP zu geeigneter Zeit präsentieren. Der ROP der alten Sicherheitsdomäne kann das Ticket dem ROP der neuen Sicherheitsdomäne bei der Übergabe des eingeschränkten Gerätes übermitteln. Der ROP der neuen Sicherheitsdomäne muss dieses Ticket anschließend seinem Autorisierungsmanager bekannt machen.

Bei der dritten Möglichkeit wird der alte Schlüssel $K(SAM, S)$ selbst ausgetauscht und dem Autorisierungsmanager in der neuen Sicherheitsdomäne bekannt gemacht. Dieser kann anschließend selbst das zum Ändern des Schlüssels nötige Ticket ausstellen. Da der neue Autorisierungsmanager mit Hilfe des erhaltenen Schlüssels allerdings Tickets für beliebige Ressourcen des Servers ausstellen kann, kann unter Umständen die Vertraulichkeit von Daten, die sich bei der Übergabe noch auf dem Server befinden, verletzt werden.

In dieser Arbeit werden für die Außer- und Wiederinbetriebnahme des eingeschränkten Server vorkonfigurierte Tickets verwendet. Das Ausstellen des Tickets durch den Autorisierungsmanager der vorherigen Sicherheitsdomäne wird in dieser Arbeit nicht detailiert betrachtet, unterscheidet sich prinzipiell aber nicht von dem in Unterabschnitt 4.6.4 beschriebenen Vorgehen zum Ausstellen von Tickets. Während der Übergabe erhält der ROP der neuen Sicherheitsdomäne das Ticket und macht es dem SAM in seiner Sicherheitsdomäne über das zur Konfiguration verwendete Webinterface bekannt. Neben dem Ticket gibt ROP im Webinterface den vollständige CoAP-URI des Servers und den neuen Schlüssel $K(SAM, S)$ an, wobei dieser auch von SAM zufällig erzeugt werden kann.

Da die notwendigen Daten SAM über das Webinterface bekannt gemacht werden, liegen diese im JSON-Format vor. Auch das zwischen den ROPs ausgetauschte Ticket wird im JSON-Format im Webinterface angegeben. Für die Umsetzung und Auswertung der Übergabe eines Server in dieser Arbeit ist der Austausch in diesem Format ausreichend, allerdings ist es aus Sicht der Bedienbarkeit nicht optimal, wenn das Ticket in einem solchem textuellen Format ausgetauscht wird. Alternativ könnten die verantwortlichen ROPs das Ticket auch beispielsweise über QR-Codes[9] austauschen und ihrem Autorisierungsmanager bekannt machen. Das im JSON-Format vorliegende Ticket muss von SAM ins CBOR-Format umgewandelt werden, um es für die Verbindung zum Server und zur Außer- und Inbetriebnahme zu verwenden.

In dieser Arbeit wird auf dem Server, anders als in [Ger15b], nur eine CoAP-Ressource unter dem URI `/key` bereitsgestellt, die sowohl für den Austausch des Schlüssels als auch für den Austausch des SAM-URIs verwendet wird. Hierdurch kann ein CoAP-Request eingespart und die Außer- und Inbetriebnahme in einem atomaren Schritt durchgeführt werden. Der Schlüssel und der URI werden im CBOR-Format als Payload der CoAP-Anfrage zum Server gesendet. Als DTLS-PSK verwendet SAM den Verifier des Tickets und sendet das Ticket-Face im CBOR-Format während des Handshakes zum Server. Der Server verarbeitet die Anfrage und speichert den enthaltenen URI und den Schlüssel $K(SAM, S)$. Anschließend muss der Server alle noch geöffneten DTLS-Verbindungen zu Clients schließen, um die Verwendung des neuen Schlüssels zu erzwingen. Außerdem muss er zur Außerbetriebnahme alle noch gespeicherten Daten aus der alten Sicherheitsdomäne, wie beispielsweise aufgezeichnete Sensordaten,

[9] QR-Code: http://www.epo.org/learning-events/european-inventor/finalists/2014/hara.html (abgerufen am 08.06.2015)

löschen. Nachdem die Anfrage erfolgreich an den Server zugestellt werden konnte und der Austausch des Schlüssels stattgefunden hat, kann der Server zur Verwaltung auf SAM hinzugefügt werden und als Objekt der Autorisierung in Zugriffsregeln verwendet werden. Er gilt mit diesem Schritt als erfolgreich in die neue Sicherheitsdomäne eingeführt.

Die für das verwendete Szenario konkret auszutauschenden Daten und eine Auswertung der Außer- und Inbetriebnahme findet sich in Unterabschnitt 6.3.1.

4.6.7 Konfiguration

SAM gehört zur Sicherheitsdomäne des ROPs und wird durch diesen verwaltet und konfiguriert. Anforderung A2.2 sieht vor, dass eine Konfiguration im laufenden Betrieb durch den ROP durchgeführt werden kann. Hierbei müssen zum einen die zur Autorisierung nötigen Subjekte, Objekte und Zugriffsregeln hinzugefügt, verändert oder gelöscht werden können. Zum anderen müssen die von SAM ausgestellten Tickets verwaltet und widerrufen werden können. Außerdem soll ROP ermitteln können, ob ausgestellte Tickets noch vom jeweiligen Server akzeptiert werden und ob widerrufene Tickets bereits an den Server zugestellt werden konnten.

Die Konfiguration soll über das von SAM schon für die Kommunikation mit den CAMs verwendete HTTP-Interface möglich sein. Dazu soll SAM eine REST-API anbieten und für deren Verwendung eine statische Website ausliefern. Die REST-API und das Webinterface dürfen nur vom ROP verwendet werden. Daher ist eine Authentifizierung von ROP nötig. Die HTTP-Schnittstelle verwendet, wie in Abschnitt 4.3 beschrieben, eine Authentifizierung auf TLS-Ebene anhand des vom Kommunikationspartner verwendeten Zertifikats. Hierbei werden nur Zertifikate zugelassen, die von der Client Authority des ROPs erstellt beziehungsweise signiert wurden. Daher muss ROP für den HTTP-Zugriff zur Konfiguration ein Zertifikat verwenden, welches von seiner Client Authority signiert wurde. Da ROP die Konfiguration über ein Webinterface im Browser ausführt, muss sein Zertifikat im Browser hinterlegt werden, sodass dieses beim Zugriff vom Browser an SAM gesendet wird.

Darüber hinaus muss während des Zugriffs eine Autorisierung stattfinden, da auf die Konfigurations-API nur ROP, aber kein CAM der ebenfalls über TLS authentifiziert wurde, zugreifen darf. Hierfür wird der SHA1-Fingerprint des Zertifikats von ROP auf SAM in der globalen Konfigurationsdatei auf SAM gespeichert und beim Zugriff abgeglichen. Somit kann nur ROP die Konfigurations-API verwenden und nur CAMs können die API für Ticketanfragen verwenden.

REST-API zur Konfiguration

Die REST-API stellt die Schnittstelle zwischen ROP und der Datenhaltung auf SAM dar, über die während des Betriebs eine Konfiguration stattfinden kann. SAM stellt über den URI-Prefix `/cfg` HTTP-Ressourcen bereit, auf denen ausgewählte CRUD-Operationen (Create, Read, Update, Delete) über die HTTP-Methoden `PUT`, `GET`, `POST` und `DELETE` ausgeführt werden können. Als Ressourcen werden von SAM die Subjekte, Objekte und Zugriffsregeln der Autorisierung und die ausgestellten Tickets und Wi-

Ressource	GET	POST	PUT	DELETE
/subjects	+	+	+	+
/subjects/{camname}	+	−	+	+
/server	+	+	+	+
/server/{serverid}	+	−	+	+
/rules	+	+	+	+
/rules/{ruleid}	+	−	+	+
/tickets	+	−	−	−
/tickets/{ticketid}	+	−	−	+
/recovations	+	+	−	−
/commissioning	−	+	−	−

Tabelle 4.1: Ressourcen und HTTP-Methoden, die über die REST-API erreicht werden können. + =Operation wird unterstützt, − =Operation nicht unterstützt

derrufungen angeboten. Als Datenformat wird sowohl in den HTTP-Anfragen als auch in den Antworten JSON verwendet. Hierbei werden die in Abschnitt 4.4 definierten Datenstrukturen in ihren jeweiligen JSON-Datentypen verwendet. Um ungültige und fehlerhafte Daten auf SAM zu vermeiden, muss dieser die Anfragen validieren und auf ungültige Inhalte überprüfen.

Das REST-Paradigma[10] sieht vor, dass jede Ressource und Unterressource, also beispielsweise die Zugriffsregeln und jede einzelne Regel, über jeweils eindeutige URIs adressiert werden können. Die folgende Tabelle zeigt die von der REST-API angebotenen Ressourcen, Unterressourcen und die möglichen Operationen auf diese. Die HTTP-Operationen werden wie in [RFC7231] definiert verwendet: `GET` zum Abfragen, `POST` zum Hinzufügen, `PUT` zum Hinzufügen oder Ändern und `DELETE` zum Löschen von Ressourcen oder Unterressourcen. Die von SAM angebotenen Ressourcen und die unterstützten Methoden zeigt Tabelle 4.1.

Bei GET-Anfragen auf Ressourcen, wie beispielsweise auf `/rules`, wird eine Liste mit allen Zugriffsregeln und bei einer Anfrage an die Unterressource `/rules/1` die Regel mit der Id 1 zurückgegeben. Zum Erstellen von Ressourcen muss eine POST-Anfrage auf die jeweilige Ressource, beispielsweise `/rules`, ausgeführt werden, da die Ids zur Identifizierung der neu angelegten Ressource von SAM generiert und in der Antwort mitgesendet wird. Zum Verändern von ganzen Ressourcen oder einzelnen Unterressourcen können PUT-Anfragen und zum Löschen DELETE-Anfragen auf diese ausgeführt werden.

Auf Tickets und Ticket-Widerrufungen sind hierbei nur eingeschränkt Operationen möglich. Tickets können über die REST-API nur abgefragt aber nicht verändert werden. Die Tickets werden auf SAM bei der Verarbeitung von Ticket Request Messages generiert und können daher nicht manuell über die REST-API erstellt werden. Widerrufene Tickets können nur abgefragt und hinzugefügt werden, das Löschen von Widerrufungen ist nicht nötig. Synonym zur POST-Anfrage auf die Ressource `/revocations` kann eine DELETE-Anfrage auf ein einzelnes Ticket zum Widerrufen verwendet werden.

[10] REST: http://martinfowler.com/articles/richardsonMaturityModel.html (abgerufen am 08.06.2015)

Die Ressource `/commissioning` wird für die in Unterabschnitt 4.6.6 beschriebene Inbetriebnahme eines Servers, der zuvor in einer anderen Sicherheitsdomäne betrieben wurde, verwendet. Auf diese ist ausschließlich eine POST-Anfrage möglich, die die in 4.6.6 beschriebenen Daten enthält.

Bei allen schreibenden Operationen sind Seiteneffekte auf die übrigen Ressourcen unvermeidlich. Beispielweise müssen beim Ändern einer Zugriffsregel, Tickets widerrufen werden, die nach der neuen Regel nicht mehr gültig wären.

Eine Beispielanfrage über die REST-API findet sich in Kapitel 6.

Webinterface

Das Webinterface stellt eine leicht bedienbare Möglichkeit zur Verwendung der REST-API und zur Konfiguration des SAMs dar. Da typische Nutzer der Systeme in der Regel wenig Erfahrung mit der Konfiguration von sicherheitsrelevanten Systemen haben [Ger15a], wurde beim Entwurf des Webinterfaces auf eine möglichst gute Bedienbarkeit geachtet.

Das Webinterface wird als statische Website von SAM über das HTTP-Interface über die Root-Ressource (`/`) bereitgestellt und kann in einem Browser unter der IPv6-Adresse oder dem Hostname von SAM aufgerufen werden. SAM erlaubt die Verwendung des Webinterfaces nur, wenn vom Browser ein Client-Zertifikat mitgesendet wurde, welches zum einen von der Client Authority des ROPs signiert ist und zum anderen der Fingerprint des Zertifikats mit dem, in der Konfigurationsdatei von SAM hinterlegten Fingerprint, übereinstimmt. ROP muss daher seinen Browser so konfigurieren, dass ein Zertifikat mitgesendet wird, welches diese Kriterien erfüllt.

Im Webinterface können die, von SAM für die Autorisierung verwalteten, Ressourcen konfiguriert werden. Es können CAMs und Server als Subjekte und Objekte der Autorisierung eingerichtet werden. Durch Zugriffsregeln können diese anschließend zusammengeführt werden. Das Webinterface wurde so entworfen, dass es den Benutzer beim Anlegen von Zugriffsregeln unterstützt, indem die vorhandenen CAMs und Server und die auf den Servern vorhandenen Ressourcen im Webinterface angezeigt werden und ROP diese nur noch passend auswählen muss.

Außerdem werden im Webinterface Tickets und Widerrufungen aufgelistet. Die von SAM ausgestellten Tickets können mit einem Klick von ROP widerrufen werden. Im Webinterface wird angezeigt, ob eine Widerrufung schon an den betreffenden Server zugestellt werden konnte. Abbildung 4.1 zeigt einen Screenshot des Webinterfaces, in welchem gerade die vorhandenen Zugriffsregeln aufgelistet werden.

4.7 Server

Beim Entwurf des Servers muss darauf Rücksicht genommen werden, dass es sich bei diesem um ein eingeschränktes Gerät handelt, dass über begrenzte Systemressourcen verfügt. Im Szenario befindet sich ein Server in jeder Bananenschachtel in Form eines WiSMotes, der mit den integrierten Sensoren Messungen der Umgebung durchführt. Für die Evaluation anhand des Nutzungsszenarios wird auf dem Server exempla-

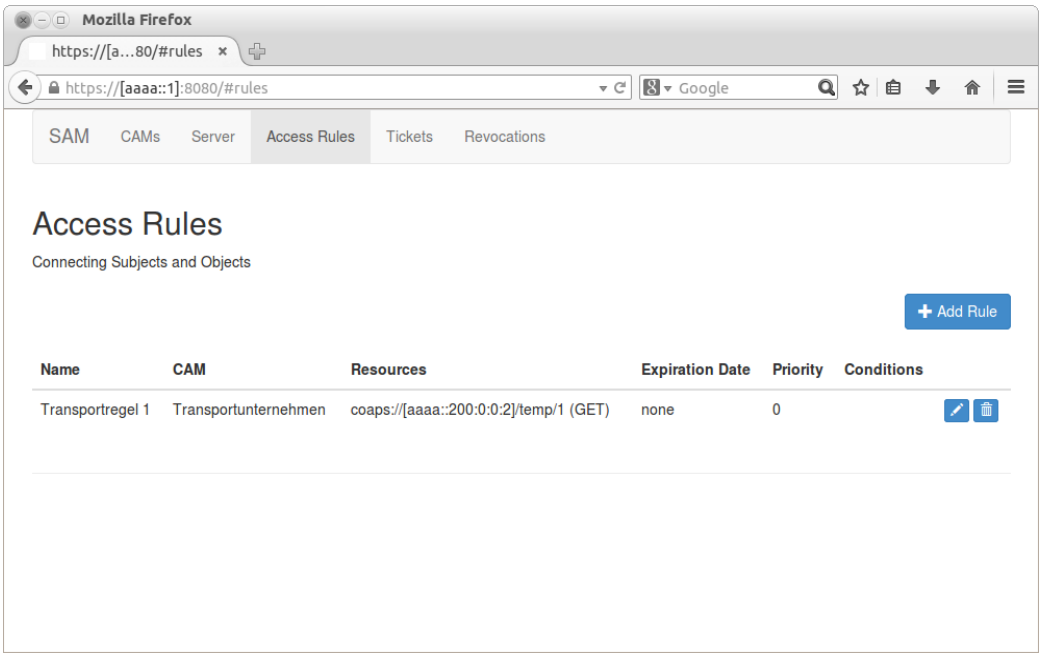

Abbildung 4.1: Screenshot des Webinterfaces zur Konfiguration von SAM.

risch ein Temperatursensor verwendet beziehungsweise simuliert, da die zu evaluierende Kommunikation auch bei mehreren Sensoren gleich abläuft. Der aktuelle Sensorwert wird vom Server als CoAP-Ressource unter dem URI `/temp/1` bereitgestellt. Da das Nutzungsszenario mit dem, in Unterabschnitt 4.2.1 vorgestellten, Programm Cooja simuliert werden soll, kann kein echter Temperatursensor verwendet werden, sondern dieser muss ebenfalls simuliert werden. Der Server simuliert in Abhängigkeit seiner Laufzeit eine für das Nutzungsszenario plausible Temperatur.

Der Server muss Verbindungsanfragen des Clients annehmen und mit Hilfe des, von diesem gesendeten, Ticket-Faces eine sichere DTLS-Verbindung zum Client aufbauen, um diesem das Abfragen von Ressourcen zu ermöglichen und die Vertraulichkeit, Integrität und Authentizität der Sensordaten sicherstellen zu können. Das Ticket-Face wird vom Client während des DTLS-Handshakes im `psk_identity` Feld der ClientKeyExchange-Nachricht zum Server gesendet. Der Server muss zum einen validieren, ob das Ticket von dem für ihn verantworlichen SAM ausgestellt wurde, und zum anderen muss der Server eine Authentifizierung des Clients durchführen. Beides wird durch das Berechnen des zu verwendenden DTLS-PSK aus dem Ticket-Face und dem mit SAM geteilten Schlüssel $K(SAM, S)$ erreicht. Der Server wendet hierfür den auch von SAM verwendeten und in Abschnitt 4.5 beschriebenen HMAC-Algorithmus auf das im CBOR-Format erhaltene Ticket-Face und den Schlüssel $K(SAM, S)$ an. Der generierte HMAC wird vom Server als PSK für die Verbindung zum Client verwendet. Der Client verwendet den von SAM generierten Verifier als PSK.

Der Server kann den richtigen, zum Verifier des Clients identischen, Schlüssel nur generieren, wenn hierfür die identischen Daten verwendet werden. Bei einer Manipulation des Ticket-Faces würde ein zum Verifier verschiedener Schlüssel vom Server generiert und der Aufbau der DTLS-Verbindung würde fehlschlagen. Hierdurch wird die Inte-

grität der im Ticket enthaltenen Informationen gewährleistet. Die Authentizität dieser Informationen wird ebenfalls sichergestellt, da der mit SAM geteilte, geheime Schlüssel $K(SAM, S)$ zur Berechnung des korrekten Schlüssels und Verifiers nötig ist. Es wird demnach sichergestellt, dass nur ein Akteur das Ticket ausgestellt haben kann, dem $K(SAM, S)$ bekannt ist. Durch den erfolgreichen Aufbau der DTLS-Verbindung findet eine Authentifizierung des Clients statt, da der Server nun sicherstellen kann, dass der Client im Besitz des korrekten Verifiers ist. Falls vom Client CoAP-Anfragen über eine ungesicherte Verbindung an den Server gesendet werden, werden diese mit der in Abschnitt 4.4 definierten SAM Information Message beantwortet, die der Server aus den gespeicherten Informationen über seinen Autorisierungsmanager generiert.

Neben der Authentifizierung muss der Server auch eine Zugriffskontrolle anhand der im Ticket-Face vorhandenen Autorisierungsinformationen, deren Integrität und Authentizität im vorherigen Schritt sichergestellt wurden, durchsetzen. Die Autorisierung muss autonom, allein basierend auf den Daten im Ticket-Face durchgeführt werden. Der Server prüft, ob die vom Client angefragte Ressource in der Liste der zugelassenen Ressourcen im Ticket-Face enthalten ist, und ob die CoAP-Methode, die auf die Ressource ausgeführt werden soll, von denen in den Autorisierungsinformationen gedeckt ist. Die CoAP-Methoden liegen im Ticket-Face als einzelne Zahl, wie in Abschnitt 4.6.2 beschrieben, vor. Zur Überprüfung muss ein logisches UND auf diese und die in der Anfrage verwendeten CoAP-Methode (beziehungsweise deren Zweierpotenz Minus 1. Siehe Abschnitt 4.6.2) ausgeführt werden. Entspricht das Ergebnis wiederum der CoAP-Methode aus der Anfrage, so ist die angeforderte Methode von den Autorisierungsinformationen im Ticket-Face gedeckt. Folgendes Beispiel verdeutlicht die Vorgehensweise. Der Client fragt eine Ressource ab (`GET`) und das Ticket-Face erlaubt `GET`- und `DELETE`-Operationen.:

$$\texttt{Autorisierungsinformationen: GET+POST} = 9 = 0b1001$$
$$\texttt{Angefragt: GET} = 1 = 0b0001$$
$$\texttt{Ergebnis: } 9 \wedge 2^{1-1} = 9 \wedge 1 = 0b1001 \wedge 0b0001 = 0b0001 = 1$$

Das Ergebnis entspricht der angefragten CoAP-Methode und somit decken die Autorisierungsinformationen im Ticket-Face diese ab.

Der Server überprüft zusätzlich, ob das Ticket bereits abgelaufen ist, indem er den Zeitstempel und die Lifetime im Ticket-Face mit seiner, mit SAM synchronisierten, Systemzeit vergleicht. Außerdem müssen eventuell im Ticket-Face vorhandene lokale Bedingungen abgeglichen werden. Dabei soll, wie in Abschnitt 4.6.2 definiert, nur die Bedingung `timeframe` umgesetzt werden, die den Zugriff auf bestimmte Tageszeiten beschränken kann. Der Server erkennt die Bedingung anhand des verwendeten CBOR-Map-Keys und überprüft, ob die aktuelle lokale Uhrzeit in dem, in der Bedingung angegebenen, Zeitraum liegt. Im letzten Schritt der Zugriffskontrolle wird vom Server noch kontrolliert, ob das verwendete Ticket bereits widerrufen wurde. Falls die Zugriffskontrolle an irgendeinem Schritt im Ablauf fehlschlägt, wird die CoAP-Anfrage vom Server abgelehnt und mit dem Statuscode `4.01` (Unauthorized) beantwortet. Wenn alle Überprüfungen positiv verliefen, gilt die Zugriffskontrolle als abgeschlossen und bestanden und der Server kann den Inhalt der Ressource im angeforderten Format ausliefern.

Neben der Kommunikation mit dem Client, muss der Server auch mit dem für ihn ver-

antwortlichen SAM sprechen können, um Ticket Revocation Messages und Handover-Nachrichten zum Schlüsselaustausch empfangen zu können. Die Kommunikation mit SAM läuft ebenfalls ausschließlich verschlüsselt über DTLS ab. Beide verwenden für die Verbindung ihren gemeinsamen Schlüssel $K(SAM, S)$ als DTLS-PSK. Der Server kann den DTLS-Verbindungsversuch von SAM und Clients anhand der `psk_identity` unterscheiden. Während Clients Ticket-Faces in der `psk_identity` senden, sendet SAM in dieser Arbeit die Zeichenkette '`sam`'.

Der Server bietet CoAP-Ressourcen an, an die die oben genannten Nachrichten in einer POST-Anfrage von SAM gesendet werden können. Darüber hinaus stellt der Server eine Ressource unter dem URI `/key` bereit, um den Schlüsselaustausch bei dem Übergang zwischen Sicherheitsdomänen zu ermöglichen. Empfängt der Server unter diesem URI neues Schlüsselmaterial, muss er alle noch bestehenden DTLS-Verbindungen schließen, um die Verwendung des neuen Schlüssels $K(SAM, S)$ zu erzwingen.

Ticket Revocation Messages können von SAM an die Ressource `/revocations` gesendet werden. In dieser werden, wie in Abschnitt 4.4 definiert, die Sequenznummern der zu widerrufenden Tickets in einem CBOR-Array angegeben. Um sich die Sequenznummern der widerrufenen Tickets zu merken, wird das im Folgenden erläuterte Sliding Window verwendet.

4.7.1 Sliding-Window

Der Server kann als eingeschränktes Gerät nur eine begrenzte Menge an Sequenznummern widerrufener Tickets im Speicher halten. Es ist ein Verfahren zu wählen bei dem sichergestellt wird, dass einmal erfolgreich widerrufene Tickets nicht wieder verwendet werden können. Daher kann ein primitives Verfahren, bei dem die widerrufenen Sequenznummern in einer Liste mit begrenzter Größe gespeichert werden, nicht verwendet werden, da hier die Situation auftreten kann, dass diese Liste voll ist. In diesem Fall müssten entweder die ältesten Widerrufungen entfernt werden, wodurch aber, falls die betroffenen Tickets noch nicht abgelaufen sind, wieder ein Zugriff mit den eigentlich widerrufenen Tickets möglich ist. Darüber hinaus könnte eine solche Liste auf dem eingeschränkten Server generell nicht viele Einträge aufnehmen, da der Server schnell sein Speicherlimit erreicht.

Stattdessen soll ein Sliding-Window-Mechanismus verwendet werden, der zum einen einen geringeren Verwaltungsoverhead besitzt und zum anderen sicherstellen kann, dass einmal widerrufene Tickets auch bei vollem Speicher nicht mehr zum Zugriff verwendet werden können. Sliding-Window-Protokolle werden häufig zur Datenflusskontrolle in Netzwerkprotokollen eingesetzt. Beispielsweise verwendet TCP ein Sliding-Window, um einen Datenstau zu vermeiden und einen Datenfluss zu gewährleisten auch wenn noch nicht alle Pakete bestätigt wurden [RFC1180].

Auf dem Server wird ein Fenster mit einer Größe von 32 Einträgen verwendet. Diese werden als 32-Bit Bitvektor dargestellt, wobei jedes Bit angibt, ob eine bestimmte Sequenznummer beziehungsweise ein bestimmtes Ticket widerrufen wurde (1) oder nicht widerrufen wurde (0). Zusätzlich muss sich der Server die Sequenznummer der unteren Grenze des Fensters in `lowest_seq` merken. Die 31 höheren Bits stellen jeweils die 31 nächsten Sequenznummern nach `lowest_seq` dar. Das Sliding-Window im Initialzustand ist in Abbildung 4.2 A) dargestellt. Das Fenster wird in zwei Situationen auf

dem Server verwendet. Zum einen wird das Fenster lesend bei der Zugriffskontrolle von Clientanfragen verwendet und zum anderen wird es beim Empfang von Ticket Revocation Messages beschrieben.

Die in einer Ticket Revocation Message gesendeten Sequenznummern müssen in das Fenster geschrieben werden. Hierbei können die folgenden drei Fälle auftreten:

1. Die zu widerrufende Sequenznummer befindet sich innerhalb des Fensters, das heißt die Sequenznummer liegt zwischen `lowest_seq` und `lowest_seq+31`: In diesem Fall muss das Bit an der Stelle im Fenster gesetzt werden, an der die Sequenznummer liegt. Die Position kann durch Subtraktion der Sequenznummern erreicht werden: $\texttt{seq\_nr} - \texttt{lowest\_seq} = \texttt{window\_index}$.
2. Die Sequenznummer liegt unter der kleinsten Sequenznummer `lowest_seq`: Hierbei kann die Sequenznummer einfach übersprungen werden. Sie muss nicht im Fenster gespeichert werden.
3. Die Sequenznummer liegt über dem Fenster, das heißt sie ist größer als `lowest_seq+31`: In diesem Fall muss das Fenster nach rechts verschoben werden, sodass die zu widerrufende Sequenznummer im letzten Element im verschobenen Fenster liegt. Das Fenster muss um so viele Stellen verschoben werden, wie die zu widerrufende Sequenznummer dieses überragt, also um $\texttt{seq\_nr} - (\texttt{lowest\_seq} + 31)$ Stellen. Hierbei müssen alle Bits im Fenster um diese Anzahl Stellen verschoben werden und die neue kleinste Sequenznummer in `lowest_seq` auf $\texttt{seq\_nr} - 31$ gesetzt werden.

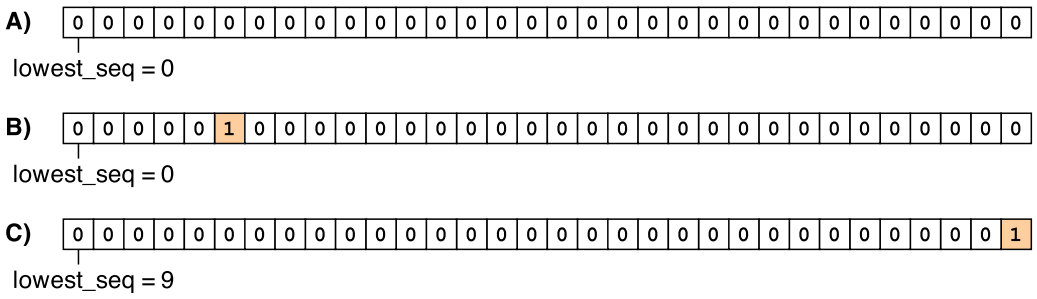

Abbildung 4.2: Sliding Window mit unterschiedlichen Zuständen

Wird beispielsweise das Ticket mit der Sequenznummer 5 widerrufen, so tritt mit dem initialen Fenster aus Abbildung 4.2 A) der erste Fall ein, da die Sequenznummer im Fenster liegt. Daher wird an der Stelle $\texttt{seq\_nr} - \texttt{lowest\_seq} = 5 - 0 = 5$ das Bit auf 1 gesetzt. Dies ist in Abbildung 4.2 B) dargestellt. Wird nun zusätzlich ein Ticket mit einer wesentlich höheren Sequenznummer, wie etwa 40, widerrufen, tritt der in Abbildung 4.2 C) dargestellte Fall ein. Die Sequenznummer liegt über dem Fenster, daher muss dieses verschoben werden, um Platz für die zu widerrufende Sequenznummer zu schaffen. Es muss eine Verschiebung um $\texttt{seq\_nr} - (\texttt{lowest\_seq} + 31) = 40 - (0 + 31) = 40 - 31 = 9$ Bits nach rechts erfolgen. Die Untergrenze `lowest_seq` wird auf $\texttt{seq\_nr} - 31 = 40 - 31 = 9$ gesetzt. Es ist zu erkennen, dass die zuvor widerrufene Sequenznummer 5 nun nicht mehr im Fenster, sondern unterhalb von diesem liegt. Die dadurch verlorene Information über das Widerrufen dieses Ticket führt allerdings, wie im Beispiel weiter unten zu erkennen ist, nicht dazu, dass dieses Ticket nun wieder

verwendet werden kann.

Bei der Zugriffskontrolle wird das Sliding Window gebraucht, um zu überprüfen, ob das verwendete Ticket bereits widerrufen wurde und somit abgelehnt werden muss. Auch hier können in Abhängigkeit der Sequenznummer wieder die drei Fälle von oben auftreten:

1. Die Sequenznummer des verwendeten Tickets befindet sich innerhalb des Windows, das heißt die Sequenznummer liegt zwischen `lowest_seq` und `lowest_seq+31`: In diesem Fall kann das Bit an der Stelle $\texttt{seq\_nr} - \texttt{lowest\_seq}$ überprüft werden. Falls es gesetzt ist, wurde das Ticket widerrufen, anderenfalls kann es akzeptiert werden.
2. Die Sequenznummer liegt unter der kleinsten Sequenznummer `lowest_seq`: In diesem Fall ist das Ticket in jedem Fall abzulehnen. Dadurch wird sichergestellt, dass einmal widerrufene Tickets in keinem Fall wieder verwendet werden können. Allerdings führt dies auch dazu, dass Tickets die unter der Grenze `lowest_seq` liegen, aber nicht widerrufen wurden, nicht mehr verwendet werden können.
3. Die Sequenznummer liegt über dem Fenster, das heißt sie ist größer als `lowest_seq+31`: In diesem Fall kann das Ticket ohne Abgleich mit dem Fenster zugelassen werden. Denn, wäre diese Sequenznummer bereits widerrufen worden, wäre das Fenster verschoben worden, sodass die Sequenznummer innerhalb des Windows liegen würde. Dies bedeutet, dass vom Server potenziell unbegrenzt beziehungsweise nur durch den, für die Sequenznummer verwendeten, Datentyp begrenzt viele Tickets zugelassen werden können, ohne dass einmal widerrufene Tickets noch verwendet werden können.

Wird beispielsweise im Zustand B) in Abbildung 4.2 ein Zugriff auf den Server mit der Sequenznummer 5 versucht, so tritt der erste Fall ein und der Server muss das Bit an der Stelle $\texttt{seq\_nr} - \texttt{lowest\_seq} = 5 - 0 = 5$ überprüfen. Da dieses gesetzt ist, muss der Zugriffsversuch abgelehnt werden. Wird dagegen im Zustand C) ein Ticket mit der Sequenznummer 5 verwendet, wird dieses abgelehnt, weil es unter der niedrigsten Sequenznummer im Fenster liegt. Ein Ticket mit der Sequenznummer 50 würde in allen, in Abbildung 4.2 dargestellten, Zuständen akzeptiert werden, da es immer über dem verwalteten Fenster liegt.

4.8 Client Authorization Manager (CAM)

Der Autorisierungsmanager des Clients, soll im Nutzungsszenario lediglich als Vermittler zwischen Client und Server Authorization Manager und als Zwischenspeicher für Tickets eingesetzt werden. Er soll, anders als in DCAF vorgesehen, keine eigenen Zugriffsregeln für die Clients verwalten und auswerten. Für CAM sind daher in dieser Arbeit nur wenige Entwurfsentscheidungen zu treffen.

Im verwendeten Szenario entspricht CAM einem weniger eingeschränkten Gerät in der Sicherheitsdomäne des Transportunternehmens. Es kann sich hierbei um einen fest installierten Server oder um das Smartphone eines Mitarbeiter handeln. Das Gerät muss über einen permanenten Zugang zum Internet verfügen, um mit Clients und SAMs kommunizieren zu können.

Um Access Requests von Clients annehmen zu können, muss CAM über CoAP kommunizieren können und eine Ressource bereitstellen, an die der Access Request gerichtet werden kann. Hier soll, wie in DCAF festgelegt, der URI `/client-auth` verwendet werden, an den ein POST-Request mit den in Abschnitt 4.4 definierten Inhalten gesendet werden kann. Zur Authentifizierung des Clients und zum Schutz der Vertraulichkeit der ausgetauschten Daten wird auch zwischen CAM und den Clients DTLS verwendet, wobei der mit dem Client zuvor ausgetauschte Schlüssel $K(CAM, C)$ als DTLS-PSK benutzt wird.

CAM muss an den im Access Request genannten SAM eine Ticket Request Message senden. CAM verwendet zunächst das Feld `SAM` aus dem Access Request (siehe Abschnitt 4.4), um zu überprüfen, ob der zu verwendende SAM dem CAM bekannt ist und ob zu diesem eine Sicherheitsbeziehung in Form ausgetauschter Zertifikate besteht. Ist dies nicht der Fall, wird die Anfrage abgelehnt und mit dem CoAP Statuscode `4.01` (Unauthorized) beantwortet. Andernfalls baut CAM mit Hilfe der Zertifikate eine sichere TLS-Verbindung zu dem im Access Request angegebenen SAM auf und sendet über diese eine Ticket Request Message als HTTP POST-Anfrage an die Ressource `/ep`. Diese enthält einen Payload aus den unveränderten Daten des Access Requests. Wenn die Authentifizierung und die Auswertung der Zugriffsregeln auf SAM positiv ausfallen und dieser ein Ticket als Antwort auf die HTTP-Anfrage sendet, leitet CAM das Ticket als Antwort auf den Access Request an den Client weiter.

4.9 Client

Der eingeschränkte Client entspricht im Nutzungsszenario dem Gerät im Kühlcontainer, der Sensordaten von den Bananenkisten abfragen will. Er gehört zur Sicherheitsdomäne des COPs beziehungsweise des Transportunternehmens im Szenario. Der Client soll im Szenario exemplarisch einen Sensorwert vom Server abfragen. Da die Kommunikation der Schwerpunkt dieser Arbeit ist, wird die Verwendung und Verarbeitung der abgefragten Sensorwerte nicht thematisiert. Zwischen dem Client und seinem Autorisierungsmanager besteht eine Sicherheitsbeziehung durch den bereits zuvor ausgetauschten Schlüssel $K(CAM, C)$. Dieser wird zur verschlüsselten Kommunikation mit CAM als DTLS-PSK verwendet.

Die Kommunikation unter Verwendung des DCAF-Protokollablaufs wird vom Client angestoßen, in dem dieser eine Unauthorized Resource Request Message an den abzufragenden Server sendet. Informationen über die abzufragende Ressource und den anzusprechenden Server wurden dem Client bereits zuvor, beispielsweise zusammen mit dem Schlüssel $K(CAM, C)$, vom COP oder von CAM mitgeteilt. Dieser Austausch wird in dieser Arbeit und auch in DCAF nicht thematisiert. Die unautorisierte Anfrage an den Server wird vom Client noch unverschlüsselt übertragen. Als Antwort erhält er vom Server eine SAM Information Message, in der unter anderem der für den Server zuständige Autorisierungsmanager und eine Ressource für Ticket-Anfragen auf diesem enthalten ist. Aus diesen Informationen und Informationen über die abzufragende Ressource, konstruiert der Client einen Access Request mit Daten im in Abschnitt 4.4 definierten Format und sendet diesen als CoAP-POST-Request an die Ressource `/client-auth` auf CAM. Die Kommunikation mit CAM läuft über sichere DTLS-Verbindungen unter Verwendung des Schlüssels $K(CAM, C)$ ab.

Nachdem die Anfrage vom Autorisierungsmanager des Clients an SAM weitergeleitet wurde, und dieser nach erfolgreicher Authentifizierung und Autorisierung ein Ticket zurücksendet, empfängt der Client dieses Ticket von CAM als Antwort auf seinen Access Request. Dieses Ticket kann der Client nun als Zugriffsausweis beim Abfragen des Sensorwertes vom Server verwenden. Hierfür startet der Client den DTLS-Verbindungsaufbau zum Server und sendet diesem im `psk_identity`-Feld der ClientKeyExchange-Nachricht während des Handshakes das im Ticket enthaltene Face zum Server. Der Client benutzt den im Ticket enthaltenen Verifier als DTLS-PSK für die Verbindung. Nachdem der Server die Autorisierungsinformationen im Ticket überprüft hat und den zu verwendenden PSK berechnet hat, kann der Verbindungsaufbau abgeschlossen werden. Anschließend kann der Client die im Ticket autorisierten Operationen auf die im Ticket enthaltenen Ressourcen auf dem Server ausführen. Der Client fragt im Nutzungsszenario die Ressource `temp/1` ab und erwartet als Ergebnis eine Temperaturangabe als CBOR-Number.

Kapitel 5

Implementierung

Die in Kapitel 4 getroffenen Entwurfsentscheidungen für das System und die einzelnen Akteure sind in Software umzusetzen. Für jeden Akteur des Szenarios ist jeweils eine eigenständige Anwendung zu entwickeln. Die Anwendungen werden einheitlich auf allen Akteuren in der Programmiersprache C implementiert, um die Wiederverwendung von Codeteilen in verschiedenen Akteuren zu ermöglichen. Es können hierbei nur Softwarebibliotheken verwendet werden, die im Sinne freier Software frei verwendet werden dürfen. Außerdem wird akteurübergreifend ein einheitlicher Coding-Style verwendet. Im Folgenden werden zunächst die Entwicklungs- und die Simulationsumgebung vorgestellt und im Anschluss Details zur Implementierung der entworfenen Akteure erläutert.

5.1 Umsetzung des Szenarios

Die Akteure des Nutzungsszenarios werden zur Entwicklung und für die Evaluation auf einem einzigen Desktop-Rechner gemeinsam betrieben und getestet. Hierbei werden allerdings alle beschriebenen Kommunikationsprotokolle verwendet. Der einzige Unterschied im Ablauf beim Testen aller Akteure auf einem Rechner ist der Wegfall beziehungsweise die Minimierung von Übertragungsverzögerungen und der Wegfall von Paketverlusten bei der Kommunikation der Akteure, da Datenpakete den Rechner nicht verlassen, sondern intern auf dem Rechner vom Betriebssystem zwischen den Akteuren weitergeleitet werden. Dies wird in der Evaluation berücksichtigt. Der Aufbau und die Informationsflüsse des Szenarios bei der Ausführung auf einem Rechner sind in Abbildung 5.1 dargestellt.

Im linken Teil der Abbildung befinden sich die eingeschränkten Geräte Client und Server als Motes innerhalb der Simulationsumgebung Cooja. Die beiden Geräte sind in der Simulation so angeordnet, dass sie direkt miteinander kommunzieren können. Außerhalb der Simulation befinden sich die weniger eingeschränkten Autorisierungsmanager CAM und SAM. Diese können ebenfalls direkt miteinander kommunizieren. Im Nutzungsszenario wird diese Verbindung zwischen den Autorisierungsmanagern des Transportunternehmen und des Bananenproduzenten über das Internet hergestellt. Im Testaufbau kommunizieren sie nicht über das Internet, da sie auf dem gleichen Rechner ausgeführt werden. Allerdings werden die gleichen Protokolle wie bei einer Verbindung

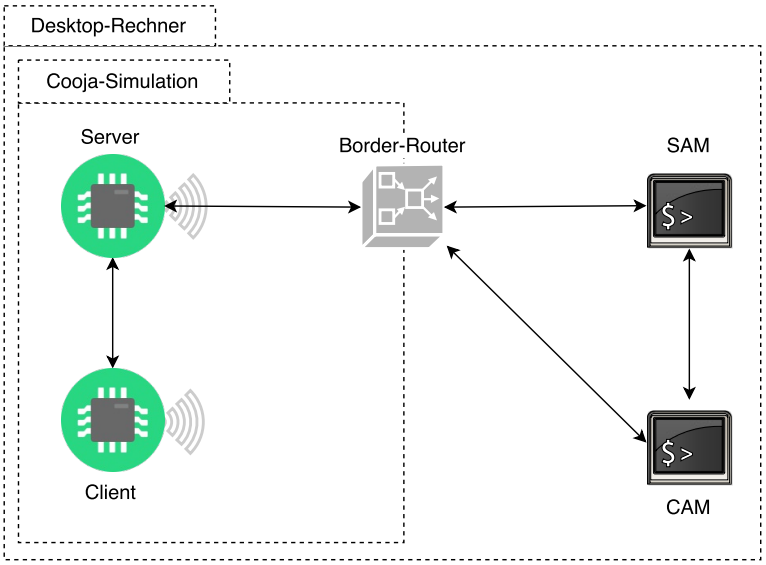

Abbildung 5.1: Aufbau des Szenarios auf einem Desktop-Rechner

über das Internet genutzt.

Um eine Verbindung zwischen den eingeschränkten Geräten in der Simulation und den weniger eingeschränkten Geräten außerhalb zu ermöglichen, sind zusätzliche Schritte notwendig. Auf der Sicherungsschicht des OSI-Modells werden innerhalb der Simulation IEEE 802.15.4 und 6LoWPAN zur Kommunikation verwendet, außerhalb der Simulation allerdings Ethernet. Daher wird eine Einheit benötigt, die eine Brücke zwischen diesen beiden Netztypen schlägt und die netzübergreifende Kommunikation ermöglicht. Beim realen Einsatz, ohne Einsatz der Simulationsumgebung, ist nicht unbedingt eine Netzwerkbrücke erforderlich. So könnten etwa auch die nicht eingeschränkten Geräte CAM und SAM selbst Hard- und Software enthalten, um neben der Kommunikation über das bereits verwendete Ethernet auch über 6LoWPAN- beziehungsweise mit RPL-Netzen zu kommunizieren.

Als Brücke zwischen den Netzen kann der, von Contiki bereits mitgelieferte[1], RPL-Border-Router verwendet werden. Dieser befindet sich entweder als eigenständiges eingeschränktes Gerät innerhalb der Cooja-Simulation oder könnte sich, parallel zur Anwendungssoftware, auch direkt auf den eingeschränkten Akteuren Server und Client befinden. In dieser Arbeit wurde ein eigenständiges Gerät als Netzwerkbrücke verwendet, um auf den Akteuren noch ausreichend Speicher für das Anwendungsszenario zur Verfügung zu haben. Der RPL-Border-Router aus Contiki horcht über einen *Serial Socket Server* auf ankommende Pakete von außerhalb der Simulation und leitet diese an die eingeschränkten Geräte weiter. Andersherum leitet er außerdem Pakete von den eingeschränkten Geräten in der Simulation nach außerhalb weiter. Der Border-Router wurde in der Simulation so angeordnet, dass nur der Server mit diesem direkt kommunizieren kann. Der Client kann mit diesem über den Server und das verwendete RPL-Routing kommunizieren. Diese Anordnung wurde gewählt, damit die Kommuni-

[1] RPL-Border-Router: https://github.com/contiki/contiki-mirror/tree/master/examples/ipv6/rpl-border-router (abgerufen am 08.06.2015)

Akteur	IPv6-Adresse	CoAP-Port	HTTP-Port
SAM	`aaaa::1`	–	`8080`
CAM	`aaaa::1`	`5684`	–
Border-Router	`aaaa::200:0:0:1`	–	–
Server	`aaaa::200:0:0:2`	`5684`	–
Client	`aaaa::200:0:0:3`	–	–

Tabelle 5.1: IP-Adressen und offene Ports der Akteure im Szenario

kation zwischen den eingeschränkten Geräten in jedem Fall direkt und nicht über den Border-Router abläuft.

Außerhalb der Simulation muss ein Gegenstück zum Border-Router existieren, welcher die Kommunikation mit diesem ermöglicht. Hierfür kann das von Contiki bereitgestellte Programm *Tunslip6*[2] verwendet werden. Dieses erstellt auf dem Rechner ein neues virtuelles Netzwerkinterface (`tun`) und verwendet das *Serial Line Internet Protocol (SLIP)* [RFC1055], um mit dem Serial Socket Server des Border-Routers zu kommunzieren und IP-Traffic auszutauschen. Dem neu eingerichteten Netzwerkinterface wird außerdem eine IPv6-Adresse und ein Prefix zugewiesen, welches von den eingeschränkten Geräten in der Simulation zum Erzeugen ihrer IPv6-Adressen verwendet wird.

In dem verwendeten Szenario soll das IPv6-Prefix `aaaa::/64` von allen Akteuren verwendet werden. Die eingeschränkten Geräte generieren ihre IPv6-Adresse aus dem Prefix und ihrer eindeutigen MAC-Adresse. In Cooja wird die MAC-Adresse wiederum durch die Reihenfolge des Hinzufügens der einzelnen Motes bestimmt. Die weniger eingeschränkten Geräte CAM und SAM verwenden beide als IPv6-Adresse die Adresse des tun-Interfaces. Tabelle 5.1 zeigt die IPv6-Adressen und die offenen Ports der Akteure im Szenario.

5.2 libcoap und tinydtls

Zur Kommunikation mit eingeschränkten Kommunikationspartnern werden von allen Teilnehmern die Bibliotheken *libcoap*[3] und *tinydtls*[4] verwendet. Beide Bibliotheken sind sowohl für den Einsatz auf eingeschränkten als auch auf weniger eingeschränkten Geräten geeignet und können in Client- und Serveranwendungen verwendet werden.

libcoap ist für die CoAP-Kommunikation zuständig und unterstützt alle gängigen CoAP-Features. Im Anwendungsprogramm kann libcoap transparent für das Versenden von Daten beliebiger Art verwendet werden. libcoap kümmert sich sowohl um die eigentliche Übertragung der Daten als auch um das Parsen und Verarbeiten von Antworten, sodass im Anwendungsprogramm lediglich die eigene Logik um die libcoap-Kommunikation herum entwickelt werden muss. Außerdem müssen bei der Verwendung als CoAP-Server lediglich die zur Verfügung gestellten Ressourcen implementiert werden. Den Zugriff auf die Ressourcen verwaltet libcoap und antwortet beim Zugriff auf

[2] Tunslip6: https://github.com/contiki/contiki-mirror/blob/master/tools/tunslip6.c (abgerufen am 08.06.2015

[3] libcoap: http://libcoap.sourceforge.net/ (abgerufen am 08.06.2015)

[4] tinydtls: http://tinydtls.sourceforge.net/ (abgerufen am 08.06.2015)

nicht vorhandene Ressourcen selbständig mit den entsprechenden CoAP-Statuscodes.

tinydtls kann von Anwendungsprogrammen für die sichere Übertragung von beliebigen Daten über UDP-Verbindungen genutzt werden. Hierbei ist wiederum eine transparente Nutzung von tinydtls auf der Transportschicht mit der, auf der Anwendungsschicht arbeitenden, Bibliothek libcoap möglich. Durch die Verwendung von DTLS können die Vertraulichkeit, Integrität und Authenzität der über die CoAP-Verbindung übertragenen Daten sichergestellt werden. Daher werden im Szenario alle CoAP-Verbindungen über DTLS abgesichert. tinydtls unterstützt unter anderem die von DCAF und CoAP verpflichtend vorgeschriebene Cipher Suite `TLS_PSK_WITH_AES_128_CCM_8` [RFC6655]. Diese basiert auf einer symmetrischen Blockchiffre mit Verschlüsselung nach dem *Advanced Encryption Standard (AES)* unter der Verwendung von Pre-Shared Keys. Der *Cipher Block Chaining-Message Authentication Code (CCM)* ermöglicht die Sicherstellung der Integrität und Authentizität der übertragenen Daten.

Die beiden Bibliotheken enthalten Beispiele, welche die einzelne Verwendung und das Zusammenwirken der Bibliotheken demonstrieren. Diese wurden in dieser Arbeit bei einigen Akteuren als Grundlage der sicheren CoAP-Kommunikation verwendet.

5.3 Server Authorization Manager (SAM)

Da SAM nicht zu den eingeschränkten Geräten zählt, ist auf diesem bei der Implementierung nicht so sehr auf Effizienz und Ressourcenverbrauch zu achten, wie bei den eingeschränkten Geräten Client und Server. SAM wurde für die `x86_64`-Architektur unter Linux entwickelt und ist auf dieser Plattform lauffähig. Das Kompilieren wurde mit einem `GCC Version 5.1.0` durchgeführt.

SAM hat, wie in Abschnitt 3.3 und Abschnitt 4.6 beschrieben, vielfältige Aufgaben, von denen einige parallel ablaufen können. So können beispielsweise Ticketanfragen unterschiedlicher CAMs über HTTP zur etwa gleichen Zeit ankommen oder Ticketwiderrufungen können sich mit Ticketanfragen überschneiden. Daher wurde für SAM eine Multithreading-Architektur entwickelt. Hierbei ist ein Thread für das Widerrufen von Tickets und die damit verbundene CoAP-Kommunikation mit den Servern zuständig. Ein weiterer Thread ist für jegliche HTTP-Kommunikation für Ticketanfragen und für die Konfiguration über die, in Unterabschnitt 4.6.7 beschriebene, REST-API zuständig. Die verwendete HTTP-Server-Bibliothek erstellt für jede HTTP-Verbindung wiederum einen eigenen Thread. Der HTTP-Thread und der Thread zur Kommunikation mit den Servern zum Widerrufen von Tickets arbeiten nach dem *Erzeuger-Verbraucher-Interaktionsmuster*[5] zusammen. Durch Aktionen des ROPs über die REST-API werden im HTTP-Thread Ticket-Widerrufungen erzeugt, die anschließend im Widerrufungs-Thread verwendet werden.

Abbildung 5.2 zeigt die Multithreading-Architektur von SAM und den Zusammenhang der einzelnen Komponenten. SAM kann von außerhalb Anfragen von ROP und von CAM über HTTP erhalten und selbst Anfragen über CoAP an Server stellen. Die unterschiedlichen Kommunikationspfade, die parallel abgelaufen werden können, sind in der Abbildung durch verschiedene Pfeile dargestellt. Der Austausch zwischen den Threads

[5] Erzeuger-Verbraucher-Problem: https://www.cs.mtu.edu/~shene/NSF-3/e-Book/SEMA/TM-example-buffer.html (abgerufen am 08.06.2015)

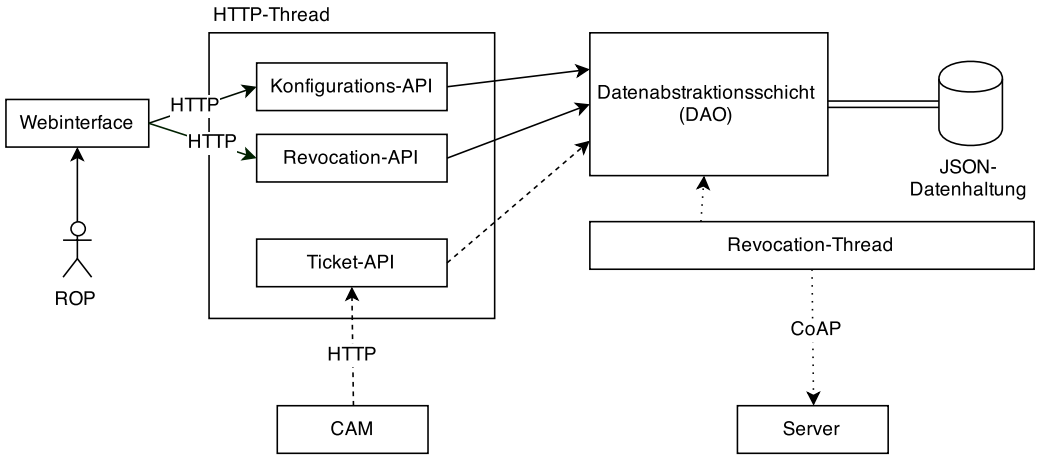

Abbildung 5.2: Multithreading-Architektur von SAM.

erfolgt über die Datenabstraktionsschicht, sodass die Anwendung selbst weitestgehend zustandslos arbeiten kann. Hierfür müssen Zugriffe auf die Datenabstraktionsschicht durch Locking-Verfahren synchronisiert und serialisiert werden. Details hierzu finden sich in Abschnitt 5.3.

Verwendete Bibliotheken

SAM benutzt eine Reihe von Softwarebibliotheken, um Netzwerkverbindungen aufzubauen oder, um zwischen Formaten zu konvertieren. Diese werden im Folgenden kurz vorgestellt.

libcoap und tinydtls

SAM muss ausschließlich mit den von ihm verwalteten Servern über CoAP kommunizieren können. Hierbei arbeitet SAM als CoAP-Client, der Informationen zu widerrufenen Tickets und neuem Schlüsselmaterial in POST-Anfragen zum Server sendet. In libcoap gibt es unter den Beispielen bereits einen CoAP-Client, der im Branch `dcaf` auch über tinydtls verschlüsselte Kommunikation ermöglicht. Dieser ist in der ursprünglichen Version als Kommandozeilenprogramm implementiert, sodass dieser zur Verwendung für die CoAP-Kommunikation auf SAM und dem Aufrufen aus dem Programmcode angepasst werden muss. Die wenigen Änderungen, die hierzu nötig waren, werden im Anhang in Anhang A aufgelistet.

HTTP und TLS

Auf SAM wird zur Konfiguration, für das Annehmen von Ticketanfragen und zum Ausliefern des Webinterfaces ein HTTP-Server benötigt. Hierfür wird die quelloffene (MIT-License) Bibliothek *Mongoose*[6] verwendet, die es ermöglicht, einen HTTP-

[6]Mongoose Web Server: https://github.com/cesanta/mongoose (abgerufen am 08.06.2015)

Server in die eigene Anwendung zu integrieren. Mongoose ist auf den gängigsten Betriebssystemen wie Linux, Windows oder Android lauffähig und unterstützt die wesentlichen Merkmale von HTTP/1.1, benötigt im kompilierten Zustand aber nur etwa 40 KiB Programmspeicher. Für nebenläufige Anfragen kann Mongoose entweder im Multithreading-Modus betrieben werden, bei dem für jede Anfrage ein eigener Thread verwendet wird, oder mit einem einzigen Thread, wobei hier asynchrone nichtblockierende Eingabe/Ausgabe (non-blocking I/O) verwendet wird, um das Blockieren bei einer Anfrage zu verhindern. Auf SAM wird Mongoose im Thread Multithreading-Modus betrieben. Außerdem kann Mongoose optional Unterstützung für IPv6 und TLS bereitstellen. Für die TLS-Unterstützung wird die weitverbreitete Bibliothek *OpenSSL*[7] verwendet.

Der HTTP-Server auf SAM horcht auf Port `8080`, welcher aber in der globalen Konfigurationsdatei geändert werden kann. Über diesen Port wird sowohl unter dem Root-URI (`/`) das Webinterface als auch über die URIs `/cfg` und `/ep` die REST-API zur Konfiguration und die API für Ticketanfragen bereitgestellt. Für das Webinterface werden statische HTML-, JavaScript- und CSS-Dateien ausgeliefert, die in jedem modernen Webbrowser dargestellt werden können. SAM verarbeitet nur HTTP-Anfragen, die über TLS verschlüsselt gestellt werden. Ticketanfragen von CAMs werden über TLS authentifiziert. Details hierzu finden sich in Abschnitt 5.3. Zugriffe des ROPs auf die REST-API und das Webinterface werden ebenfalls über TLS verschlüsselt und authentifiziert. Allerdings findet die Autorisierung des ROPs auf der Anwendungsschicht anhand des Fingerprints, des von ROP verwendeten Zertifikats, statt. Weitere Informationen zu dieser Authentifizierung finden sich in Abschnitt 5.3.

Um Mongoose für SAM verwenden zu können, mussten zwei kleine Änderungen an der Bibliothek vorgenommen werden. Zum einen musste ein Bug gefixt werden, der bei Clients, die über IPv6 Anfragen stellen, verhindert hat, dass deren IP-Adresse ausgelesen werden konnte. Darüber hinaus musste Mongoose dahingehend erweitert werden, dass der OpenSSL-Kontext an das Anwendungsprogramm weitergegeben wird. Dieser wird in SAM benötigt, um das Zertifikat und den Fingerprint des Zertifikats der aktuellen Anfrage ermitteln zu können. Eine Dokumentation der Änderungen findet sich im Anhang in Abschnitt A.

JSON und CBOR

SAM muss gespeicherte und in Nachrichten verschickte Daten intern in verschiedenen Datenformaten darstellen und sie zwischen diesen Datenformaten hin und her konvertieren können. Im Wesentlichen werden auf SAM drei Datenformate verwendet. Die Nachrichteninhalte bei der Kommunikation mit CAMs und mit Servern liegen im CBOR-Format vor, während die Inhalte bei Anfragen über die REST-API und bei der Datenhaltung auf SAM im JSON-Format kodiert sind. Im internen Ablauf verwendet SAM hingegegen C-Strukturen zur Datenrepräsentation, wie sie in Abschnitt 4.4 vorgestellt wurden. Daher müssen ankommende Nachrichteninhalte aus dem CBOR- beziehungsweise JSON-Format in die entsprechenden Strukturen und abgehende Nachrichteninhalte, umgekehrt, aus den Strukturen ins CBOR- beziehungsweise JSON-Format konvertiert werden.

[7] OpenSSL: https://www.openssl.org/ (abgerufen am 08.06.2015)

Zur Unterstützung der JSON-Konvertierung wird die Bibliothek Jansson[8] und für die CBOR-Konvertierung auf SAM die CBOR-Bibliothek des, in Abschnitt 4.2 erwähnten, RIOT-OS[9] verwendet. Diese wurde eigentlich zur Verwendung auf eingeschränkten Geräten entwickelt, funktioniert allerdings ebenfalls auf nicht eingeschränkten. Die CBOR-Bibliothek wird auch auf CAM verwendet.

Weitere Drittsoftware

Für kleinere Aufgaben verwendet SAM noch einige wenige weitere Bibliotheken. Da diese meistens nur einen speziellen Zweck erfüllen, werden sie im Folgenden lediglich kurz erwähnt, aber nicht weiter ausgeführt.

Als Bibliothek für abstrakte Datentypen, wie beispielsweise einfach verkettete Listen oder Warteschlangen, verwendet SAM die *QUEUE-Bibliothek* des FreeBSD-Projekts[10]. Um die, in textueller Darstellung vorliegenden, CoAP- oder HTTP-URIs zu parsen, wird die Bibliothek *uriparser*[11] verwendet. Außerdem wird zur Kodierung von Daten im Base64-Format die Base64-Bibliothek der Apache Software Foundation[12] verwendet. Die auf SAM verwendeten Datenstrukturen können unter Umständen, beispielsweise beim Verifier im Ticket, Binärdaten enthalten. Da die JSON-Spezifikation allerdings keinen Datentyp für die Repräsentation von Binärdaten vorsieht [RFC7159], werden auf SAM Binärdaten im JSON-Format als Base64-Zeichenketten dargestellt.

Authentifizierung von CAM

Die Authentifizierung von CAM, welcher Ticketanfragen an SAM stellen will, findet auf TLS-Ebene anhand von Zertifikaten statt. In der Sicherheitsdomäne des ROPs gibt es, wie in Abschnitt 4.3 beschrieben, eine Client Authority, die zum Signieren von Zertifikaten des CAMs verwendet wird. Die verwendete HTTP-Server-Bibliothek Mongoose unterstützt die Verwendung einer selbstsignierten CA zur Authentifizierung, sodass auf SAM lediglich das Zertfikat der CA und zu jedem CAM der Fingerprint des jeweiligen Zertifikat des CAMs gespeichert werden muss. Die einzelnen Zertifikate der CAMs müssen auf SAM nicht vorgehalten werden, da SAM diese während der Anfrage allein anhand der CA verifizieren kann. Der Fingerprint wird nicht zur Authentifizierung, sondern zur Autorisierung und zum Abgleich der zum CAM gehörenden Zugriffsregeln benötigt.

Datenhaltung

Wie in Unterabschnitt 4.6.1 beschrieben, soll vor der konkreten Datenhaltung eine Abstraktionsschicht nach dem DAO-Entwurfsmuster verwendet werden, die einen flexiblen

[8] Jansson: http://www.digip.org/jansson/ (abgerufen am 08.06.2015)
[9] RIOT-CBOR: https://github.com/RIOT-OS/RIOT/tree/master/sys/cbor (abgerufen am 08.06.2015)
[10] QUEUE: https://www.freebsd.org/cgi/man.cgi?query=queue (abgerufen am 08.06.2015)
[11] uriparser: http://uriparser.sourceforge.net/ (abgerufen am 08.06.2015)
[12] Base64: http://www.opensource.apple.com/source/QuickTimeStreamingServer/QuickTimeStreamingServer-452/CommonUtilitiesLib/base64.c (abgerufen am 08.06.2015)

Austausch der konkreten Datenhaltungsmethode ermöglicht. Zur Implementierung einer konkreten Methode zur Datenhaltung müssen im Wesentlichen Funktionen für die CRUD-Operationen (Create, Read, Update und Delete), für die auf SAM verwalteten Ressourcen, implementiert werden. Außerdem muss für jede Ressource, wie beispielsweise den Subjekten oder Objekten der Autorisierung, eine Funktion implementiert werden, die anhand der Angabe des eindeutigen Feldes der Ressource diese zurückgibt. Beispielsweise muss die Methode `dao_get_rule(Id)` implementiert werden, um aus den gespeicherten Zugriffsregeln die Regel mit der angegebenen `Id` zurückzugeben.

Die entworfene JSON-DAO implementiert die oben genannten Schnittstellen und speichert die Ressourcen zum einen in Dateien im JSON-Format im nichtflüchtigen Speicher. Zum anderen werden die von der JSON-Bibliothek Jansson verwendeten Datenstrukturen im Arbeitsspeicher gehalten, um einen schnellen Zugriff zu ermöglichen und die Dateizugriffe auf den nichtflüchtigen Speicher zu minimieren.

Da SAM eine Multithreading-Architektur verwendet, können Situationen entstehen, in denen parallel Zugriffe aus den verschiedenen Threads auf die Datenhaltung stattfinden. Um die Konsistenz der gespeicherten Daten zu erhalten, werden die Zugriffe synchronisiert. Hierfür werden Mutexe aus `PTHREAD`[13] verwendet.

Webinterface

Das Webinterface besteht, wie in Abschnitt 4.6.7 beschrieben, ausschließlich aus statischen Dateien, die von SAM unverändert an den Browser des ROPs gesendet werden. Um die Interaktion und die Kommunikation mit der HTTP-REST-API des SAMs zu ermöglichen, wird JavaScript verwendet. Hierzu wird auf einige JavaScript-Bibliotheken zurückgegriffen, die im Folgenden lediglich kurz erwähnt, aber nicht ausführlich beschrieben werden. Als Architektur-Framework und REST-Client wird die Bibliothek *Backbone.js*[14] verwendet. Diese nutzt eine Reihe weiterer Bibliotheken wie etwa *jQuery*[15] für HTML-Manipulationen und *Underscore.js*[16] für das Arbeiten mit JavaScript-Datentypen. Außerdem wird *Bootstrap*[17] für Icons und das Design einzelner HTML-Komponenten verwendet.

Das Framework Backbone.js sorgt dafür, dass die Daten im Webinterface automatisch mit denen auf SAM synchronisiert werden. Hierdurch können ausgestellte Tickets innerhalb weniger Sekunden automatisch im Webinterface angezeigt werden.

Wie in Unterabschnitt 4.6.7 beschrieben, soll für die Autorisierung des ROPs auf SAM der Fingerprint seines, für die Verbindung verwendeten, Zertifikats verwendet werden. Dieser wird mit einem, in der globalen Konfigurationsdatei auf SAM gespeicherten, Fingerprint verglichen. Durch diese Art der Autorisierung kann jede Anfrage individuell autorisiert werden, ohne dass auf SAM, wie in der Regel auf HTTP-Servern üblich, Zustände, wie Sessions, verwaltet werden müssen. Die Konfiguration des gültigen Fingerprints von ROP erfolgt über die statische Konfigurationsdatei `samcfg.json` von

[13] PTHREAD: http://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread_mutex_lock.html (abgerufen am 08.06.2015)
[14] Backbone.js http://backbonejs.org/ (abgerufen am 08.06.2015)
[15] jQuery: https://jquery.org/ (abgerufen am 08.06.2015)
[16] Underscore.js: http://underscorejs.org/ (abgerufen am 08.06.2015)
[17] Bootstrap: http://getbootstrap.com/ (abgerufen am 08.06.2015)

SAM. In dieser können neben dem Fingerprint auch das Netzwerkinterface und der Port angegeben werden, auf denen SAM Verbindungsanfragen entgegen nehmen soll.

Verhalten beim Start

SAM ist als Anwendung entwickelt, die über die Kommandozeile gestartet wird. Hierbei müssen keine Argumente beim Start übergeben werden, da alle Einstellungen über die, in Unterabschnitt 4.6.7 erklärte, globale Konfigurationsdatei geladen werden. Nach dem Start erstellt SAM zunächst einen Thread für das HTTP-Interface und wartet auf dem in der Konfigurationsdatei angegebenen Port passiv auf Verbindungsanfragen von CAMs oder ROP. Außerdem startet SAM einen weiteren Thread für die CoAP-Kommunikation mit Servern zum Widerrufen von Tickets. Falls in den gespeicherten widerrufenen Tickets noch welche enthalten sind, die noch nicht erfolgreich zum Server gesendet werden konnten, beginnt SAM unter Umständen direkt nach dem Start mit dem Versuch die Widerrufungen zuzustellen. Weitere Aktionen werden von SAM nach dem Start nicht automatisch, sondern nur ereignisgesteuert bei Anfragen durchgeführt.

5.4 Client Authorization Manager (CAM)

Der Autorisierungsmanager des Clients besteht ebenfalls aus einem weniger eingeschränkten Gerät und ist, wie SAM, als Kommandozeilenprogramm implementiert. CAM arbeitet als CoAP-Server, der Anfragen von Clients, die von ihm verwaltet werden, entgegennimmt und über HTTP weiter an die entsprechenden SAMs leitet. Hierfür verwendet CAM ebenfalls libcoap und tinydtls für die sichere Kommunikation mit den Clients. libcoap stellt im Branch `dcaf` eine Highlevel-API für die Verwendung von libcoap als CoAP-Server mit tinydtls Unterstützung bereit[18]. Dieses wird auf CAM weitestgehend unverändert vewendet.

Für das Weiterleiten der Ticketanfragen vom Client muss CAM ausschließlich als HTTP-Client arbeiten und Anfragen an SAM zustellen. Daher wird auf CAM keine vollständige HTTP-Server Bibliothek wie Mongoose, sondern die HTTP-Client Bibliothek *libcurl*[19] verwendet. Diese erlaubt, wie das Kommandozeilenwerkzeug *curl*, das Verschicken beliebiger HTTP-Anfragen und unterstützt darüber hinaus die Verwendung von OpenSSL für verschlüsselte Verbindungen. libcurl unterstützt sowohl das Senden von Client-Zertifikaten als auch die Authentifizierung des Kommunikationspartners anhand dessen Zertifikat. CAM hat sein Zertifikat bereits, wie in Abschnitt 3.8 beschrieben, vor dem Beginn des Protokollablauf erhalten und verwendet dieses als Client-Zertifikat. Außerdem verwendet er das ebenfalls in diesem Schritt erhaltene CA-Zertifikat, um SAM zu authentifizieren. Details zu dem Austausch der Zertifikate finden sich in Unterabschnitt 6.3.2.

Nach dem Start verhält CAM sich ausschließlich passiv und wird nur bei ankommenden Access Requests von Clients aktiv.

[18] libcoap Highlevel-API: http://sourceforge.net/p/libcoap/code/ci/dcaf/tree/app.h
[19] libcurl: http://curl.haxx.se/libcurl/

5.5 Eingeschränkte Akteure

Die eingeschränkten Geräte Client und Server sind als Contiki-Anwendung implementiert. Sie sind daher nur in der Simulationssoftware Cooja oder auf Hardware läuffähig, welche die Ausführung von Contiki-Anwendungen unterstützt. Beim Entwurf und der Implementierung des Clients und des Servers sind die beschränkt zur Verfügung stehenden Systemressourcen zu berücksichtigen. Es ist im Anwendungsprogramm darauf zu achten, dass möglichst wenig statische Variablen verwendet werden, die permanent im Arbeitsspeicher gehalten werden müssen. Es ist außerdem darauf zu achten, dass nach Möglichkeit wenig Speicher dynamisch alloziert wird und dass, wenn dies in Ausnahmenfällen nötig ist, dieser in jedem Fall nach der Verwendung wieder freizugeben ist.

Die eingeschränkten Geräte verwenden ebenfalls libcoap und tinydtls für die sichere CoAP-Kommunikation. Allerdings ist das in libcoap vorhandene und auf SAM verwendete High-Level-Interface nicht für die Verwendung auf eingeschränkten Geräten ausgelegt. Daher wurde die Integration von libcoap und tinydtls auf dem Server und dem Client in der Anwendungslogik explizit implementiert.

Contiki verwendet ein ereignisgesteuertes Prozessmodell, bei dem kein automatisches Scheduling zwischen Prozessen stattfindet, sondern die Prozesse auf Anwendungsebene die CPU freigeben, wenn sie gerade keine Aufgaben zu erfüllen haben (kooperatives Multitasking). Contiki-Anwendungen warten daher in der Regel auf Ereignisse wie Timer-Interrupts oder Interrupts durch ankommende Netzwerkpakete und werden beim Eintreten dieser benachrichtigt und führen erst dann die bei diesem Ereignis vorgesehenen Funktionen durch. Die beim Ereignis aufgerufenen Funktionen im Anwendungsprogramm müssen immer zunächst komplett abgearbeitet werden, bevor die Kontrolle über den Programmfluss wieder an Contiki abgegeben wird und auf weitere Ereignisse reagiert werden kann. Die Besonderheiten des Prozessmodells von Contiki sind bei der Implementierung des Servers und des Clients zu berücksichtigen. Sie müssen ebenfalls ihre Aufgaben beim Eintreten von Ereignissen durchführen können und anschließend die Kontrolle über den Programmfluss wieder abgeben. Es ist darauf zu achten, dass die Abarbeitung eines Ereignis' in einem angemessenen Zeitraum erfolgt, sodass nachfolgende Ereignisse nicht verloren gehen.

Um die entwickelte Contiki-Anwendung für die Verwendung auf WiSMotes oder in der Simulationssoftware zu übersetzen, wird der Quelltext des Contiki-Betriebssystems benötigt. Dieser wird im Makefile der eigenen Contiki-Anwendung bekannt gemacht. Außerdem können im Makefile einzelne Features von Contiki hinzugefügt oder deaktiviert werden. Dies kann genutzt werden, um ein möglichst kleines Binary zu erzeugen, und somit mehr Programmspeicher für das Anwendungsprogramm zur Verfügung zu haben. Beispielsweise wurde die TCP-Unterstützung zur Übersetzungszeit deaktiviert. Außerdem konnten Funktionen, die ein eingeschränktes Gerät als Border-Router benötigt, entfernt werden.

Um Nachrichten erstellen und Nachrichteninhalte verarbeiten zu können, benötigen die eingeschränkten Geräte ebenfalls eine CBOR-Bibliothek. Auf dieser wird die Bibliothek *cn-cbor*[20] verwendet.

[20] cn-cbor: https://github.com/cabo/cn-cbor (abgerufen am 08.06.2015)

Server

Der eingeschränkte Server initialisiert beim Start das CoAP-Interface und registriert die nötigen, in Abschnitt 4.7 beschriebenen, Ressourcen. Anschließend verhält er sich ausschließlich passiv und wartet auf ankommende Netzwerkpakete. Bei ankommenden Verbindungsfragen muss der Server bei der Auswahl des zu verwendenden DTLS-PSK entscheiden, ob der Verbindungsversuch von einem Client oder vom für den Server verantwortlichen SAM initialisiert wurde. Diese Auswahl trifft der Server auf Grundlage der `psk_identity`. Die Authentifizierung wird erst durch den erfolgreichen Aufbau der DTLS-Verbindung erreicht, bei der beispielsweise der SAM nachweisen kann, dass er den gleichen Schlüssel besitzt. Falls der Verbindungsversuch nicht von SAM, sondern von einem Client ausgeht, unterscheidet der Server bei der Auswahl beziehungsweise Berechnung des PSK noch, ob ein Ticket mitgesendet wurde oder nicht. Wenn ein Ticket mitgesendet wurde, wird dieses zur Berechnung verwendet. Anderenfalls wird überprüft, ob der Server bereits ein früheres Ticket für diesen Client gespeichert hat. Falls dies der Fall ist, wird dieses verwendet. Die Auswahl des gespeicherten Tickets findet hierbei anhand der vom Client verwendeten IPv6-Adresse statt.

Bei der Verwendung von tinydtls auf eingeschränkten Geräten kann es bei der Kommunikation mit weniger eingeschränkten Geräten zu Timing-Problemen auf dem eingeschränkten Gerät kommen, bei dem der DTLS-Handshake fehlschlägt und deshalb keine Verbindung aufgebaut werden kann. Dieses Problem wurde bei der, in dieser Arbeit verwendeten, tinydtls Version 0.70 (Commit `#4a739c2e90eef3c758642e707514614a133576dd`) bei der Verwendung des eingeschränkten Servers innerhalb der Simulationsumgebung Cooja beobachtet. Während des DTLS-Handshakes, der in Abschnitt 2.4 beschrieben wurde und in Kapitel 6 ausgewertet wird, werden an einigen Stellen von einem Kommunikationsteilnehmer mehrere Handshake-Nachrichten direkt nacheinander gesendet. Im Falle eines DTLS-Servers ist dies einerseits die Sequenz der Nachrichten *ServerHello* - *ServerKeyExchange* - *ServerHelleDone* und andererseits im letzten Schritt die Sequenz *ChangeCipherSpec* - *Finished*. Beim Client tritt diese Situation nur einmal während des Handshakes auf, und zwar während der Sequenz der Nachrichten *Client Key Exchange* - *ChangeCipherSpec* - *Finished*. Sendet nun beispielsweise der weniger eingeschränkte SAM als DTLS-Client diese Nachrichtensequenz an den eingeschränkten Server, so können hierbei Nachrichten verloren gehen. Bei dem von Contiki verwendeten kooperativen Prozessmodell müssen Anwendungen selbstständig die CPU abgeben, wenn sie ihre Aufgabe erledigt haben. Angenommen der eingeschränkte Server erhält die erste Nachricht der genannten Sequenz (Client Key Exchange) und befindet sich gerade in der Verarbeitung dieser. Falls nun bereits die nächste Nachricht (ChangeCipherSpec) vom DTLS-Client gesendet wird, so kann es passieren, dass die Nachricht auf dem Server verloren geht, da er sich noch in der Verarbeitung der vorherigen befindet. Gleiches kann bei der dritten Nachricht (Finished) passieren. Dies wird durch die eingesetzte Entwicklungsumgebung begünstigt, bei der sowohl die weniger eingeschränkten Geräte SAM und CAM auf einem Rechner als auch die eingeschränkten Geräte Server und Client auf dem gleichen Rechner innerhalb der Simulationsumgebung Cooja ausgeführt werden. Hierdurch entstehen bei dem Versand von Netzwerkpaketen quasi keine Übertragungsverzögerungen, da die Netzwerkpakete den Rechner nie verlassen. Es konnte im Rahmen dieser Arbeit nicht ermittelt werden, ob das Problem in einem Szenario, bei dem echte Hardware verwendet wird und bei

dem in der Kommunikation Latenzen vorhanden sind, ebenfalls auftritt. Um das Problem zu Umgehen beziehungsweise um trotz des Problems eine Verbindung aufbauen zu können, wurde auf der Seite der weniger eingeschränkten Geräte SAM und CAM eine künstliche Latenz bei der Übertragung der genannten Nachrichtensequenzen eingefügt. Diese beträgt jeweils 100 ms zwischen den Nachrichten. Bei der Evaluation werden diese künstlichen Latenzen berücksichtigt.

Client

Der eingeschränkte Client ist im Szenario dafür zuständig den Protokollablauf von DCAF durch das Absenden einer Unauthorized Resource Request Message und dem anschließenden Access Request, an den für ihn verantwortlichen CAM, zu starten. Hierfür wartet der Client nach der Initialisierung der Netzwerkverbindung und der Bibliotheken libcoap und tinydtls auf einen Timer-Interrupt, der 20 Sekunden nach dem Start ausgelöst wird, sodass die Geräte in den 20 Sekunden zuvor ausreichend Zeit haben, um sich untereinander zu finden und mit dem Border-Router Informationen zum RPL-Routing auszutauschen. Der Interrupt löst die erste Nachricht des Ablaufs aus. Alle weiteren Schritte werden durch den Protokollablauf und die erhaltenen Antworten bestimmt. Nach dem Erhalt des Tickets führt der Client in kurzen Intervallen von fünf Sekunden eine Anfrage auf die im Ticket autorisierten Ressourcen durch.

Kapitel 6

Evaluation

Die entworfenen und umgesetzten Akteure des Nutzungsszenarios werden in diesem Kapitel ausgewertet. Ziel der Evaluation ist einerseits die Auswertung der einzelnen Akteure und andererseits die Auswertung des DCAF-Protokollablaufs bei der Verwendung im konkreten Szenario. Die zentrale Frage ist hierbei, ob die dezentrale Architektur und der von DCAF vorgegebene Protokollablauf von den eingeschränkten Geräten ausgeführt werden können und ob die dezentrale Architektur für eingeschränkte Geräte allgemein und für das vorgesehene Szenario im konkreten geeignet ist.

Zunächst werden in Abschnitt 6.1 die Testumgebung und die sich aus dieser ergebenen Einflussfaktoren beschrieben, welche bei der Auswertung berücksichtigt werden müssen. Anschließend findet in Abschnitt 6.2 eine Auswertung des auf den eingeschränkten Geräten durch die Implementierung und die verwendeten Bibliotheken verbrauchten Speichers statt, bevor in Abschnitt 6.3 die eigentliche Evaluation der einzelnen Schritte des Protokollablauf detailliert anhand des Nutzungsszenarios durchgeführt wird. Hierbei wird zunächst der Vorlauf des Nutzungsszenarios ausgewertet, bei dem zwischen den beteiligten Unternehmen auf der Ebene des ROPs und COPs Daten ausgetauscht und den Akteuren bekannt gemacht werden. Außerdem wird die Inbetriebnahme eines bereits gebrauchten eingeschränkten Gerätes in der Sicherheitsdomäne des ROPs evaluiert. Anschließend werden die Schritte des Protokollablaufs detailliert abgearbeitet, wobei jede Kommunikationsform und jedes Kommunikationsprotokoll im Ablauf einmal exemplarisch ausgewertet wird. In Abschnitt 6.4 werden anschließend die Eignung des DCAF-Frameworks für das vorgesehene Nutzungsszenario evaluiert und mögliche Probleme und Angriffsszenarien aufgezeigt.

6.1 Testumgebung

Bei der Evaluation soll der in Abschnitt 5.1 beschriebene Aufbau der Akteure und des Szenarios verwendet werden. Für die eingeschränkten Akteure wird, wie in Abschnitt 4.1 beschrieben, die WiSMote-Plattform verwendet. Die enthaltene CPU arbeitet mit maximal 16 MHz und enthält 16 KiB Arbeitsspeicher und 128 KiB Programmspeicher. Als Host-Rechner, auf dem sowohl die Simulation als auch die weniger eingeschränkten Akteure ausgeführt werden, wird zur Evaluation ein Desktop-Computer mit einer `x86_64`-Architektur und einem aktuellen Linux Betriebssystem verwendet.

Die eingeschränkten Geräte verwenden die entwickelten Contiki-Anwendungen und die vorgestellten Bibliotheken — insbesondere tinydtls und libcoap. Bei der über DTLS gesicherten Übertragung wird zwischen allen Akteuren ausschließlich die Cipher Suite `TLS_PSK_WITH_AES_128_CCM_8` verwendet.

Durch die Verwendung eines einzelnen Rechners während der Evaluation und durch den Einsatz der Simulationsumgebung Cooja statt des Einsatzes echter eingeschränkter Hardware, existieren einige Einflussfaktoren, die es im realen Nutzungsszenario nicht gäbe und die daher nicht ausgewertet werden können. Da alle Akteure auf dem selben Rechner laufen, verlassen Netzwerkpakete den Rechner nie wirklich, sondern werden lediglich intern zwischen den Anwendungen weitergereicht. Hierdurch entstehen, anders als im realen Szenario bei der Kommunikation über Funk, kaum Übertragungsverzögerung. Dies kann einerseits zu den, in Abschnitt 5.5 beschriebenen, Problemen während des DTLS-Handshakes führen und andererseits kann durch das Fehlen von Latenzen keine realistische Angabe der Übertragungsdauer und einer Round-Trip-Time (RTT) erfolgen. Außerdem wird durch den Einsatz der Simulation eine weitere Ebene zum Szenario hinzugefügt, auf der Fehler enthalten sein können, welche das Ergebnis beeinflussen können.

Zur Evaluation der, während des Ablaufs des Nutzungsszenarios und der Kommunikation, benötigten Ressourcen auf den eingeschränkten Akteuren, sind geeignete Verfahren und Werkzeuge einzusetzen. Die Wahl der eingesetzten Werkzeuge kann zu einem gewissen Teil die Ergebnisse beeinflussen. So können beispielsweise verschiedene Werkzeuge Berechnungen intern unterschiedlich durchführen und zu abweichenden Ergebnissen kommen.

Um den Speicherbedarf der entwickelten Contiki-Anwendungen für die eingeschränkten Akteure zu schätzen, werden die Programme *msp430-size* und *msp430-objdump* aus der MSP430-Toolchain eingesetzt. Hiermit wird sowohl der statisch zugewiesene Arbeitsspeicher (RAM) als auch der benötigte Programmspeicher (ROM) ermittelt. Zum ermitteln des Speicherbedarf einzelner Bestandteile oder Bibliotheken wird die Anwendung jeweils einmal mit und einmal ohne dieses Feature übersetzt und anschließend mit den oben genannten Werkzeugen untersucht. Das Delta der Programmgrößen gibt dann den für dieses Feature benötigten Speicherbedarf an. Die Größe des von den eingeschränkten Akteuren verwendendeten Stacks während der einzelnen Schritte, wird mit dem Stack-Analyzer von Cooja ausgewertet.

Um die Verarbeitungszeit einzelner Schritte oder Codeteile zu messen, werden die von Contiki und der MSP430-CPU bereitgestellten Real-Time-Timer (Rtimer) verwendet. Der Timer hat eine Genauigkeit von 32786 Ticks pro Sekunde und damit eine Auflösung von $1s/32768hz = 30,5176\mu$s. Zum Messen wird das Contiki-Modul Energest verwendet, welches die Anzahl der seit der Timeraktivierung vergangenen Ticks zählt.

Die zwischen den Akteuren ausgetauschten Netzwerkpakete werden mit dem Trafficanalyseprogramm *Wireshark* und dem in Cooja vorhandenen *Radio Message Analyzer* aufgezeichnet und analysiert. Da die meisten Verbindungen verschlüsselt ablaufen, können mit diesen Werkzeugen nicht die Nutzdaten oberhalb der Transportschicht untersucht werden. Allerdings können sie zur Evaluation der darunter liegenden Schichten und des DTLS-Handshakes verwendet werden.

6.2 Speicherbedarf der Contiki-Anwendungen

In Abschnitt 5.5 wurde beschrieben, dass beim Entwurf und während der Umsetzung der Contiki-Anwendungen für die eingeschränkten Akteure auf den begrenzten Speicher der WiSMote-Plattform von 16 KiB Arbeits- und 128 KiB Programmspeicher zu achten ist. Ein großer Teil des Speichers wird bereits vom Betriebssystem selbst und den verwendeten Bibliotheken benötigt. Die eingeschränkten Akteure des Szenarios, Server und Client, haben auf Anwendungsebene zwar unterschiedliche Aufgaben, verwenden darunter aber den gleichen Protokollstack. Daher unterscheiden sie sich an einigen Stellen beim Speicherbedarf nicht. Abbildung 6.1 zeigt den von den eingeschränkten Akteuren benötigten Programmspeicher und Abbildung 6.2 den benötigten statischen Arbeitsspeicher.

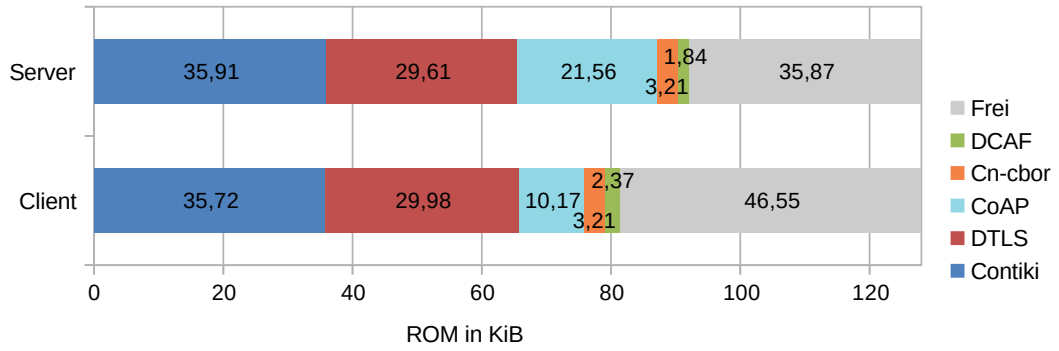

Abbildung 6.1: Aufteilung des Programmspeichers auf den eingeschränkten Akteuren

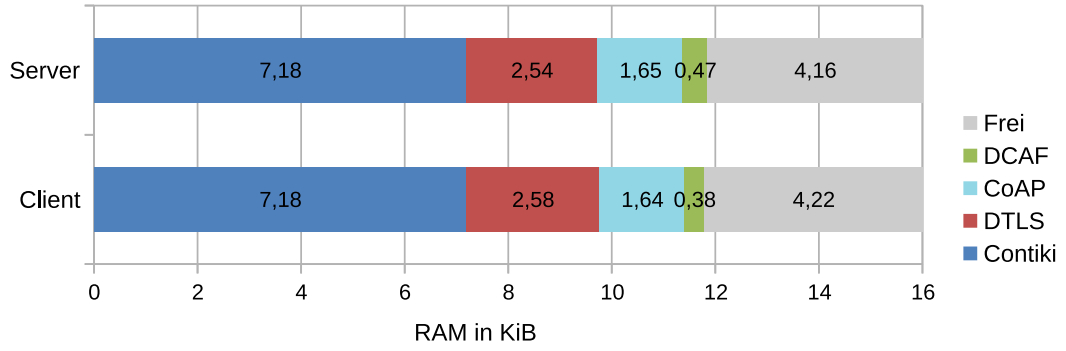

Abbildung 6.2: Aufteilung des statischen Arbeitsspeichers auf den eingeschränkten Akteuren

Contiki wurde so konfiguriert, dass nur die für das Szenario nötigen Feature integriert werden. Dazu gehören der uip-Stack mit UDP- und IPv6-Unterstützung und RPL-Routing, allerdings beispielsweise kein TCP und keine Unterstützung für das Dateisystem Coffee. Eine minimale Contiki-Anwendung mit diesen Features benötigt auf beiden eingeschränkten Akteuren knapp 36 KiB Programmspeicher und 7,18 KiB statischen Arbeitsspeicher. Damit werden ohne weitere Bibliotheken und Anwendungs-

software bereits knapp 28 % des Programmspeichers und fast 45 % des Arbeitsspeichers belegt.

Die eingeschränkten Akteure müssen miteinander und mit ihren Autorisierungsmanagern sicher über DTLS-Verbindungen kommunizieren. Die Unterstützung hierfür benötigt auf dem Client 29,98 KiB Programmspeicher und 2,58 KiB statischen Arbeitsspeicher. Auf dem Server werden 29,61 KiB Programmspeicher und 2,53 KiB statischer Arbeitsspeicher belegt. Enthalten ist hierbei sowohl der von der Bibliothek tinydtls benötigte Speicher als auch eine minimale Verwendung auf der Anwendungsebene, bei der insbesondere einige Funktionen für den Austausch von Datenpaketen zwischen dem uip-Stack und tinydtls implementiert werden müssen.

Für die Unterstützung der CoAP-Kommunikation auf der Anwendungsebene werden auf dem eingeschränkten Client 10,17 KiB Programmspeicher und 1,64 KiB statischer Arbeitsspeicher benötigt. Auf dem Server werden hingegen 21,56 KiB Programmspeicher und 1,65 KiB statischer Arbeitsspeicher benötigt. Auch hierbei wird der Großteil wieder durch die verwendete Bibliothek libcoap belegt und ein kleinerer Teil durch die Verwendung auf Anwendungsebene, auf der insbesondere die Verknüpfung zwischen tinydtls und libcoap implementiert werden muss. Beim benötigten Programmspeicher ist bei der CoAP-Unterstützung erstmals ein signifikanter Unterschied zu erkennen. Der Server benötigt mehr als doppelt so viel Programmspeicher wie der Client. Dies ist insbesondere darauf zurückzuführen, dass der Server CoAP-Resourcen verwalten und zur Verfügung stellen muss.

In DCAF wird für die übertragenen Nachrichten als Datenformat CBOR verwendet. Sowohl auf dem Client als auch auf dem Server müssen CBOR-Nachrichten sowohl gelesen als auch erstellt werden. Die auf den eingeschränkten Geräten verwendete Bibliothek cn-cbor benötigt auf beiden Akteuren lediglich 3,21 KiB Programmspeicher und keinen statischen Arbeitsspeicher. Da DCAF auf den genannten Bibliotheken aufbaut, benötigt die Implementierung des DCAF-Protokollablaufs selbst nicht viel Speicher. Der Client verwaltet Informationen wie Schlüssel und URI seines Autorisierungsmanagers. Außerdem enthält er Code, um Verbindungen zu CAM und dem Server aufzubauen und deren Antworten zu verarbeiten. Hierfür benötigt der Client 2,37 KiB Programmspeicher und 0,38 KiB statischen Arbeitsspeicher. Auch der Server verwaltet den URI seines Autorisierungsmanagers und den mit diesem geteilten Schlüssel. Darüber hinaus werden widerrufene Tickets im Sliding-Window gespeichert. Allerdings muss der Server, anders als der Client, selbst nicht aktiv Verbindungsanfragen stellen. Auf dem Server benötigt die Implementierung des DCAF-Protokollablaufs 5,05 KiB Programmspeicher und 0,47 KiB statischen Arbeitsspeicher.

Insgesamt werden von den zur Verfügung stehenden 128 KiB Programmspeicher, von den oben genannten Komponenten, vom Server 92,13 KiB und vom Client 81,45 KiB benötigt. Den eingeschränkten Geräten stehen demnach, bei der Verwendung der genannten Bibliotheken und der dezentralen Authentifizierung und Autorisierung mit DCAF, noch 35,87 KiB auf dem Server und 46,55 KiB auf dem Client für darauf aufbauende Anwendungen zur Verfügung. In einem realen Einsatz würde ein Teil des freien Speichers beispielsweise für die Kommunikation mit Sensoren oder Aktoren benötigt werden. Von den 16 KiB Arbeitsspeicher, die der WiSMote bereitstellt, sind auf dem Client bereits 11,78 KiB und auf dem Server 11,84 KiB durch die oben genannten Komponenten statisch belegt. Es stehen zur Laufzeit daher noch 4,22 KiB beziehungs-

weise 4,16 KiB Arbeitsspeicher für den Stack und die dynamische Speicherallokierung zur Verfügung.

6.3 Evaluation des Nutzungsszenarios

Im Folgenden werden der, in Abschnitt 3.1 bereits definierte, Ablauf des Nutzungsszenario detailliert durchgespielt und die einzelnen Schritte evaluiert. Die Evaluierung beginnt einige Schritte vor dem eigentlichen DCAF-Protokollablauf, da hier die für die Kommunikation notwendigen Daten zwischen den Unternehmen im Szenario ausgetauscht und den jeweiligen Akteuren mitgeteilt werden. Außerdem findet im Vorlauf auch die Inbetriebnahme des eingeschränkten Servers, gemäß des in Unterabschnitt 4.6.6 definierten Übergangsszenario, statt.

6.3.1 Außer- und Inbetriebnahme des Servers

Das Nutzungsszenario beginnt damit, dass die von Unternehmen `A` angebauten Bananen geerntet und für den Transport vorbereitet werden. Hierfür werden sie, wie in Abschnitt 3.1 beschrieben, in Bananenschachteln verpackt. Diese sind von Unternehmen `A` mit WiSMote-Knoten bestückt worden und enthalten die entworfene Software für den eingeschränkten Server. Die Bananenschachteln und damit auch die enthaltenen Geräte und Sensoren können wiederverwendet werden. Hierbei kann allerdings bei der vorherigen Verwendung ein anderer SAM für das Gerät zuständig sein als beim aktuellen Transport. Daher wird zunächst der eingeschränkte Server in die Sicherheitsdomäne des ROPs eingeführt und dem SAM in dieser Domäne bekannt gemacht.

Um diese Außer- und Inbetriebnahme des Servers zu erreichen wird, wie in Unterabschnitt 4.6.6 beschrieben, ein Ticket benötigt, welches von einem Autorisierungsmanager in der alten Sicherheitsdomäne ausgestellt wurde. Dieses enthält einen Verifier, der als DTLS-PSK für den Aufbau einer Verbindung zum neuen Server verwendet wird. Außerdem enthält das Ticket Autorisierungsinformationen die einen schreibenden Zugriff auf die Ressource `/key` gestattet. Im Szenario wurden die Bananenschachteln zuvor vom gleichen Unternehmen `A` verwendet, weshalb das benötigte Ticket bereits vorliegt. Um die Außer- und Inbetriebnahme des Servers zu erreichen verwendet ROP das Webinterface. In diesem gibt er das vom ROP der vorherigen Sicherheitsdomäne erhaltene Ticket, den fortan zu verwendenden neuen Schlüssel $K(SAM, S)$ und den URI des neuen Autorisierungsmanagers, an die Ticketanfragen gesendet werden können, an. Für das konkrete Szenario werden für die Inbetriebnahme des Servers in der Bananenschachtel folgende Daten von ROP über das Webinterface an SAM übertragen:

```
POST /commissioning HTTP/1.1
{
  "server_uri": "coaps://[aaaa::200:0:0:2]:5684/key",
  "new_key": "2NUH+rjrEUGxFywoYSpWBQ==",
  "new_sam": "https://[aaaa::1]:8080/ep",
  "ticket": {
    "face": {
      "sequence_number": 0,
      "AI": [
```

```
            {
              "server": "aaaa::200:0:0:2",
              "resource": "key",
              "methods": 2
            }
          ],
          "timestamp": 1000,
          "dtls_psk_gen_method": 0,
          "lifetime": 3600,
          "conditions": []
        },
        "verifier_size": 16,
        "verifier": "kz3AOzvM5PZviH68kfKaNQ=="
      }
    }
```


Da SAM weniger eingeschränkt ist, soll auf eine Evaluation der SSL-Verbindung an dieser Stelle verzichtet werden, da diese dem SAM keine Probleme bereitet.

SAM versucht unmittelbar nach dem Empfangen der Anfrage eine DTLS-Verbindung zu dem neuen Server herzustellen. Er verwendet hierfür den Verifier als DTLS-PSK und sendet das Ticket-Face im CBOR-Format während des DTLS-Handshakes an den Server. Der Aufbau der DTLS-Verbindung und die Auswertung der Autorisierungsinformationen im Ticket werden an dieser Stelle nicht, sondern erst bei der Auswertung des eigentlichen DCAF-Protokollablaufs in Abschnitt 6.3.4 evaluiert. Wenn der Verbindungsaufbau erfolgreich verläuft und der Server die Zugriffskontrolle auf Basis der Autorisierungsinformationen im Ticket-Face durchgesetzt hat, übernimmt er den erhaltenen Schlüssel $K(SAM, S)$ und den neuen URI des SAMs.

Nachdem SAM und der Server erfolgreich die fortan zu verwendenden Schlüssel ausgetauscht haben, gilt der Server als erfolgreich in die Sicherheitsdomäne des ROPs und in die Verwaltung des SAMs eingeführt. Im realen Szenario könnte SAM an dieser Stelle über einen Verzeichnisdienst die Ressourcen des neu hinzugefügten Servers abfragen und diese speichern. Da dies, wie in Abschnitt 3.8 beschrieben, außerhalb des Rahmens dieser Arbeit liegt, wird dieser Schritt als bereits durchgeführt angenommen. Dafür werden die vom Server verwalteten Ressourcen beim Hinzufügen des Servers über das Webinterface statisch konfiguriert. ROP wird im Webinterface signalisiert, dass die Außer- und Inbetriebnahme erfolgreich verlaufen ist und der Server fortan nach dem Hinzufügen zur Verwaltung von SAM als Objekt der Autorisierung verwendet werden kann.

COP muss während des Vorlaufs ebenfalls den Client in seine Sicherheitsdomäne einführen und diesen seinem Autorisierungsmanager bekannt machen. Dieser Schritt soll allerdings, wie in Abschnitt 3.8 beschrieben, in dieser Arbeit nicht betrachtet werden. Stattdessen wurden die zu verwendenden Schlüssel statisch im Client und in CAM hinterlegt.

6.3.2 Auftragserteilung

Nachdem die eingeschränkten Geräte in die jeweilige Sicherheitsdomäne eingeführt und die Bananen für den Transport verpackt und mit Sensoren ausgestattet wurden, erteilt der Bananenproduzent den Auftrag für den Transport an Unternehmen

B. Hierbei werden zwischen den Unternehmen, neben den für den Auftrag notwendigen Logistikinformationen, auch technische Informationen über die zu verwendenden Autorisierungsmanager und die vom Client abzufragende Ressource ausgetauscht.

Die Kommunikation zum Austausch der Informationen zwischen den Autorisierungsmanager wird, wie in Abschnitt 3.8 beschrieben, in DCAF und in dieser Arbeit nicht festgelegt und wird daher an diese Stelle auch nicht ausgewertet beziehungsweise durchgeführt. Stattdessen werden ausschließlich die Inhalte der auszutauschenden Informationen beschrieben und ihre Verwendung auf dem jeweiligen Akteur erläutert. Folgende Informationen müssen zwischen den Autorisierungsmanagern ausgetauscht werden, um den Ablauf des Nutzungsszenarios unter Verwendung von DCAF zu ermöglichen:

1 COP an ROP

- **1.1** Zertifikat-Request für CAM: `'---BEGIN CERTIFICATE REQUEST---...'`

2 ROP an COP

- **2.1** Mit CA signiertes Zertifikat: `'---BEGIN CERTIFICATE---...'`
- **2.2** ClientAuthority-Zertifikat von ROP: `'---BEGIN CERTIFICATE---...'`
- **2.3** Abzufragende Ressource: `coaps://[aaaa::200:0:0:2]:5684/temp/1`
- **2.4** Informationen zur Ladung, zum Beispiel Richttemperatur

Das Transportunternehmen muss dem Bananenproduzenten lediglich ein selbst erstelltes und bisher nur selbstsigniertes X.509-Zertifikat in Form eines *Certificate-Signing-Requests (CSR)* mitteilen. ROP muss den CSR mit seiner Client Authority signieren und das Ergebnis als Zertifikat COP mitteilen. Außerdem muss er COP das Zertifikat seiner CA übermitteln, um CAM eine Authentifizierung von SAM, anhand des von SAM verwendeten Zertifikats, zu ermöglichen. Ein Protokoll und eine Anleitung für die im Szenario nötigen Zertifikate findet sich im Anhang in Abschnitt C. ROP teilt COP außerdem die während des Transports abzufragende Ressource anhand eines URIs und eventuell weitere Informationen zur Ladung, wie die einzuhaltene Richttemperatur, mit.

CAM muss SAM bekannt gemacht werden, damit auf SAM Zugriffsregeln für diesen festgelegt werden können. Hierfür fügt ROP über das Webinterface einen neuen CAM als Autorisierungssubjekt hinzu und gibt einen Fingerprint des, mit der CA signierten, Zertifikats von CAM an. Außerdem wird ein informeller Name verwendet, um eine visuelle Zuordnung im Webinterface zu ermöglichen. Der vom Webinterface abgesetzte POST-Request zum SAM hat das folgende Format:

```
ROP -> SAM
POST /subjects HTTP/1.1
{
  "cert_fingerprint": "mJ6N/FQ55PDPfiOWqMzf3W2uWgk=",
  "name": "Transportunternehmen"
}
```

Nachdem der CAM auf SAM hinzugefügt wurde, kann ROP eine Zugriffsregel über das Webinterface hinzufügen, die zum Ausstellen von Tickets benötigt wird und einen Zugriff auf den eingeschränkten Server erlaubt. ROP fügt eine implizite Zugriffsregel hinzu, die dem CAM beziehungsweise den von ihm verwalteten Clients den Zugriff

auf sämtliche Ressourcen des Servers erlaubt. Der dafür notwendige HTTP-Request enthält die Zugriffsregel mit folgenden Informationen:

```
ROP -> SAM
PUT /rules HTTP/1.1
{
  "id": "Transportregel 1",
  "subject": "mJ6N/FQ55PDPfi0WqMzf3W2uWgk=",
  "resources": [
    {
      "server": "aaaa::200:0:0:2",
      "resource": "*",
      "methods": 15
    }
  ],
  "expiration_time": 0,
  "priority": 0,
  "conditions": []
}
```

SAM speichert die Zugriffsregel für die spätere Auswertung bei einer Ticketanfrage. Die Zugriffsregel liegt im JSON-Format vor und benötigt 186 Byte Speicher. Zum Erstellen eines Tickets aus dieser Regel muss sie daher später während der Ticketanfrage noch reduziert werden, um auch von eingeschränkten Geräten verarbeitet werden zu können. Für den weniger eingeschränkten SAM stellt die Speicherung in einem nicht-reduzierten Format allerdings kein Problem dar.

Die vier Akteure des Szenarios sind nun mit den für den DCAF-Protokollablauf notwendigen Informationen ausgestattet. Daher kann dieser im nächsten Schritt, während der Verladung der Bananenschachteln im Hafen in Costa Rica, gestartet werden.

6.3.3 Verladung

Bei der Verladung der Bananenschachteln in den Container sind die eingeschränkten Geräte Client und Server das erste Mal nah genug aneinander, um direkt miteinander kommunizieren zu können. Außerdem können die eingeschränkten Geräte während der Verladung im Hafen noch ihren jeweiligen Autorisierungsmanager erreichen.

Nachdem die Geräte sich und den Border-Router im Netz gefunden haben, wird der DCAF-Protokollablauf durch den Client durch das Senden einer unautorisierten Anfrage an den Server gestartet. Der Client verwendet hierbei die im vorherigen Schritt zwischen COP und ROP ausgetauschten Informationen, um den korrekten Server zu kontaktieren.

Unauthorized Resource Request Message

Der Client sendet eine Unauthorized Resource Request Message an den Server, um den für diesen Server verantwortlichen SAM zu ermitteln. Da der Client und der Server zu diesem Zeitpunkt noch keine Sicherheitsbeziehung zueinander haben, kann diese Nachricht als einzige im gesamten Ablauf nicht verschlüsselt übertragen werden. Der Client muss eine unverschlüsselte CoAP-Anfrage auf eine beliebige, auf dem Server vorhandene, Ressource ausführen. Da die Nachricht nicht verschlüsselt wird, ist, wie

in Abschnitt 4.4 beschrieben, darauf zu achten in der Anfrage keine schützenswerten Daten, wie etwa Sensordaten, mitzusenden. Daher fragt der Client mit einem GET-Request die ihm bekannte Ressource auf dem Server ab. Hierbei wird das folgende Netzwerkpaket gesendet, welches im Folgenden exemplarisch einmal ausgewertet wird:

```
Client -> Server
IEEE 802.15.4 + IPv6 + UDP
61CCA6CD AB020000 00000000 00030000 00000000  a...................
007AF700 00                                    .z...
CoAP-Message
11006304 001E0300 4EFD1633 0015AE3A 40015209  ..c.....N..3...:@.R.
B474656D 70013111 46                          .temp.1.F
```

Das gesendete Netzwerkpaket hat insgesamt eine Größe von 54 Byte. Das Paket besitzt auf der untersten Schicht einen IEEE-802.15.4-Header und darüber einen IPv6-Header mit Absender- und Zieladresse. Die beiden Header werden durch die *6LoWPAN-Header-Compression* zusammengefasst. Die Header können nur zusammengefasst werden wenn, wie im Szenario, die Geräte direkt, also über einen Hop, miteinander kommunizieren. Zusammen mit dem UDP-Header benötigen der gesamte Header 25 Byte. Die übrigen 29 Byte werden von der CoAP-Nachricht belegt. Die Gesamtgröße von 54 Byte ist klein genug, um in einen IEEE-802.15.4-Frame gesendet zu werden und eine Fragmentierung zu vermeiden. Die Größe der IEEE-802.15.4-, IPv6- und UDP-Header unterscheidet sich im Folgenden bei den verschiedenen Nachrichten nicht. Daher findet keine erneute Auswertung der unteren Schichten statt.

Während des Erstellens und dem Senden der Anfrage wächst der Stack auf dem Client auf maximal 500 Byte. Während des Empfangens und Verarbeitens der Antwort des Servers wächst der Bedarf auf dem Client kurzzeitig auf bis 600 Byte, liegt aber noch deutlich unter der maximalen Stackgröße. Um die im Folgenden beschriebene Antwort des Servers zu verarbeiten, benötigt der Client 0,61 ms.

SAM Information Message

Der Server verarbeitet die CoAP-Anfrage vom Client. Da sie unverschlüsselt übertragen wurde, muss der Server die Anfrage ablehnen und eine SAM Information Message aus den Informationen, die er über den für ihn verantwortlichen SAM während des in Unterabschnitt 6.3.2 beschriebenen Schritts bekommen hat, generieren. Diese wird im CBOR-Format in der Antwort gesendet. Neben den Informationen zum SAM muss der Server auch einen aktuellen Zeitstempel erstellen und diesen als Nonce mit in die Antwort schreiben. Hierfür verwendet der Server die seit dem letzten Start vergangenen Sekunden. Für die Verarbeitung der Anfrage und dem Generieren der CBOR-Nachricht benötigt der Server 0,3 ms. Die vom Server gesendete Antwort enthält im konkreten Fall folgende Informationen:

```
Client <- Server
{
  SAM: "https://[aaaa::1]:8080/ep",
  TS: 20
}
```

Die gesendete Antwort hat inklusive der SAM Information Message eine Größe von 79 Byte. Wiederum 25 Byte fallen für die verschiedenen Header unterhalb der Anwendungsschicht an, so dass 54 Byte von der CoAP-Nachricht belegt werden. Auch an dieser Stelle muss somit keine Fragmentierung der Nachricht auf der 6LoWPAN-Schicht stattfinden. Während der Verarbeitung der Anfrage und dem Generieren der SAM Information Message wächst der Stack auf dem Server auf maximal 750 Byte an. Das Senden von CoAP-Anfragen und das Verarbeiten und Beantworten dieser stellt demnach für die eingeschränkten Geräte Client und Server keine hohe Hürde dar.

Access Request

Mit den Daten aus der SAM Information Message kann der Client nun ein Ticket mit einem Access Request über seinen Autorisierungsmanager anfragen. Um mit dem Autorisierungsmanager sicher zu kommunizieren, baut der Client zunächst eine DTLS-Verbindung zu diesem auf. Hierfür verwendet er den mit dem Autorisierungsmanager geteilten Schlüssel $K(CAM, C)$ als Pre-Shared Key. Eine ausführliche Auswertung des DTLS-Handshakes findet sich in Abschnitt 6.3.4, bei der Analyse der Kommunikation zwischen den eingeschränkten Akteuren. Aufgrund der, in Abschnitt 5.5 beschriebenen, Probleme beim DTLS-Handshake zwischen weniger eingeschränkten und den eingeschränkten Geräten, kann an dieser Stelle keine Auswertung der Gesamtdauer des Handshakes erfolgen, da dieser auf der Seite des weniger eingeschränkten Gerätes künstlich verzögert werden musste.

Nachdem die DTLS-Verbindung zwischen dem Client und seinem Autorisierungsmanager erfolgreich aufgebaut wurde, erstellt der Client den Access Request und den dafür nötigen CBOR-Payload. Dieser enthält in diesem ersten Fall ausschließlich den in der SAM Information Message enthaltenen URI von SAM und den Timestamp. Der CBOR-Payload des Access Request enthält im konkreten Szenario 68 Byte Daten mit folgendem Inhalt:

```
Client -> CAM
POST /client-auth
{
  SAM: "https://[aaaa::1]:8080/ep",
  SAI: [["coaps://[aaaa::200:0:0:2]/temp/1",1]],
  TS: 20
}
```

Für das Erstellen und das Versenden des Access Requests benötigt der Client 20,75 ms. Hiervon werden 3,36 ms von tinydtls für das Verschlüsseln der Nachricht benötigt. Die CoAP-Nachricht ist mit dem Payload zu groß, um unfragmentiert zu werden. Daher findet eine Fragmentierung in zwei Pakete auf der 6LoWPAN-Schicht statt. Diese kann, wie in Abschnitt 2.3 beschrieben, negative Auswirkungen auf die Kommunikation im Sensornetz haben.

Ticket Request Message

Der weniger eingeschränkte CAM hat in jedem Fall ausreichend Speicher und Rechenleistung, um die festgelegten Protokolle zu verwenden. Daher wird auf eine detaillierte

Auswertung der auf CAM anfallenden Rechenzeit an dieser Stelle verzichtet. CAM entschlüsselt den vom Client empfangenen Access Request und leitet die enthaltenen CBOR-Daten inhaltlich unverändert an den in diesen Daten genannten SAM weiter. Die Anfrage an SAM enthält folgende Informationen:

```
CAM -> SAM
POST /ep HTTP/1.1
{
  SAI: [["coaps://[aaaa::200:0:0:2]/temp/1",1]],
  TS: 20
}
```

Die Kommunikation mit SAM läuft über HTTP und TLS verschlüsselt ab. Die Verbindung zwischen den Autorisierungsservern wird im Szenario über das Internet hergestellt. CAM verwendet das im Vorlauf ausgetauschte Zertifikat von SAM, um diesen zu authentifizieren.

Ticket Grant Message

SAM führt ebenfalls eine Authentifizierung des CAMs anhand des gespeicherten Zertifikats durch und verarbeitet anschließend die empfangene Ticket Request Message. SAM durchsucht die gespeicherten Zugriffsregeln und findet die zuvor erstellte implizite Regel. Die Autorisierung gilt mit dem Finden der passenden Regel als abgeschlossen und SAM kann ein Ticket ausstellen. Da die Zugriffsregel einen impliziten Zugriff auf den gesamten Server erlaubt, müssen keine expliziten Autorisierungsinformationen ins Ticket geschrieben werden. SAM speichert das Ticket intern im JSON-Format mit folgenden Informationen für das konkrete Szenario:

```
{
  "id": "dAuZVTJUm4qmxQ==",
  "face": {
    "AI": [
      {
        "server": "aaaa::200:0:0:2",
        "resource": "*",
        "methods": 15
      }
    ],
    "sequence_number": 0,
    "timestamp": 30,
    "conditions": [],
    "lifetime": 3600,
    "dtls_psk_gen_method": 0
  },
  "verifier": "cUbS3+ikTgOxJrNnWFY9DQ==",
  "verifier_size": 16
}
```

Die enthaltenen Felder werden wie in Unterabschnitt 4.6.4 beschrieben berechnet. Für den Versand und für die Berechnung des Verifiers muss das Ticket in das CBOR-Format konvertiert und hierbei möglichst reduziert werden, sodass es im späteren Schritt ohne Fragmentierung im Sensornetz verwendet werden kann. Im CBOR-Ticket fällt die nur intern auf SAM benötigte Ticket-Id weg. Auch die Autorisierungsinformationen wer-

den bei der impliziten Autorisierung im Ticket weggelassen. Außerdem werden statt Zeichenketten als Schlüssel der Map in CBOR Ganzzahlen verwendet. Das Ticket sieht im CBOR-Format wie folgt aus:

```
CAM <- SAM

Diagnostic Notation:
{
  Face: {
    TS: 30,
    LIFETIME: 86400,
    DTLS_PSK_GEN: 0,
    SEQ_NR: 0
  },
  Verifier: h'7146d2dfe8a44e03b126b36758563d0d'
}

Hex (32 Byte):
A2 08 A4 05 18 1E 06 19 0E 10 07 00 10 00 09 50
71 46 D2 DF E8 A4 4E 03 B1 26 B3 67 58 56 3D 0D
```

Hierdurch kann das JSON-Ticket von 246 Byte auf 32 Byte im CBOR-Format zusammengefasst werden. SAM sendet das Ticket im CBOR-Format in der Ticket Grant Message als Antwort auf die Ticket Request Message an CAM. ROP kann über das Webinterface sehen, dass ein Ticket ausgestellt wurde und hat hier auch die Möglichkeit dieses Ticket zu widerrufen.

Ticket Transfer Message

CAM leitet das von SAM ausgestellte Ticket wiederum unverändert an den Client als Antwort auf den zuvor gestellten Access Request weiter. CAM verwendet hierfür die noch bestehende DTLS-Verbindung zum Client. Die an den Client gesendeten Daten passen wiederum nicht in ein IEEE 802.15.4 Frame. Daher findet auch an dieser Stelle eine Fragmentierung in zwei Frames statt. Auf dem Client muss der erhaltene DTLS-Record zunächst entschlüsselt werden, um an das darin enthaltene Ticket zu gelangen. Für die Entschlüsselung des 32 Byte großen Tickets benötigt die Bibliothek tinydtls ca. 3,36 ms. Anschließend wird die DTLS-Verbindung zu CAM vom Client beendet. Für diesen Vorgang müssen noch einmal Nachrichten zwischen dem Client und CAM ausgetauscht werden.

Der Client besitzt nun alle nötigen Informationen zum dezentralen Zugriff auf den Server. Während der weiteren Kommunikation mit dem Server benötigen sowohl der Client als auch der Server keine Verbindung mehr zu ihren Autorisierungsmanagern. Daher ist die Vorbereitungsphase des ersten Transportschritts abgeschlossen und der Transport und die Kommunikation zwischen Container und Bananenschachtel kann beginnen.

6.3.4 Transport

Bevor der Client kontinuierlich die Sensorwerte des Servers abfragen kann, muss eine DTLS-Verbindung zwischen den beiden eingeschränkten Akteuren aufgebaut werden,

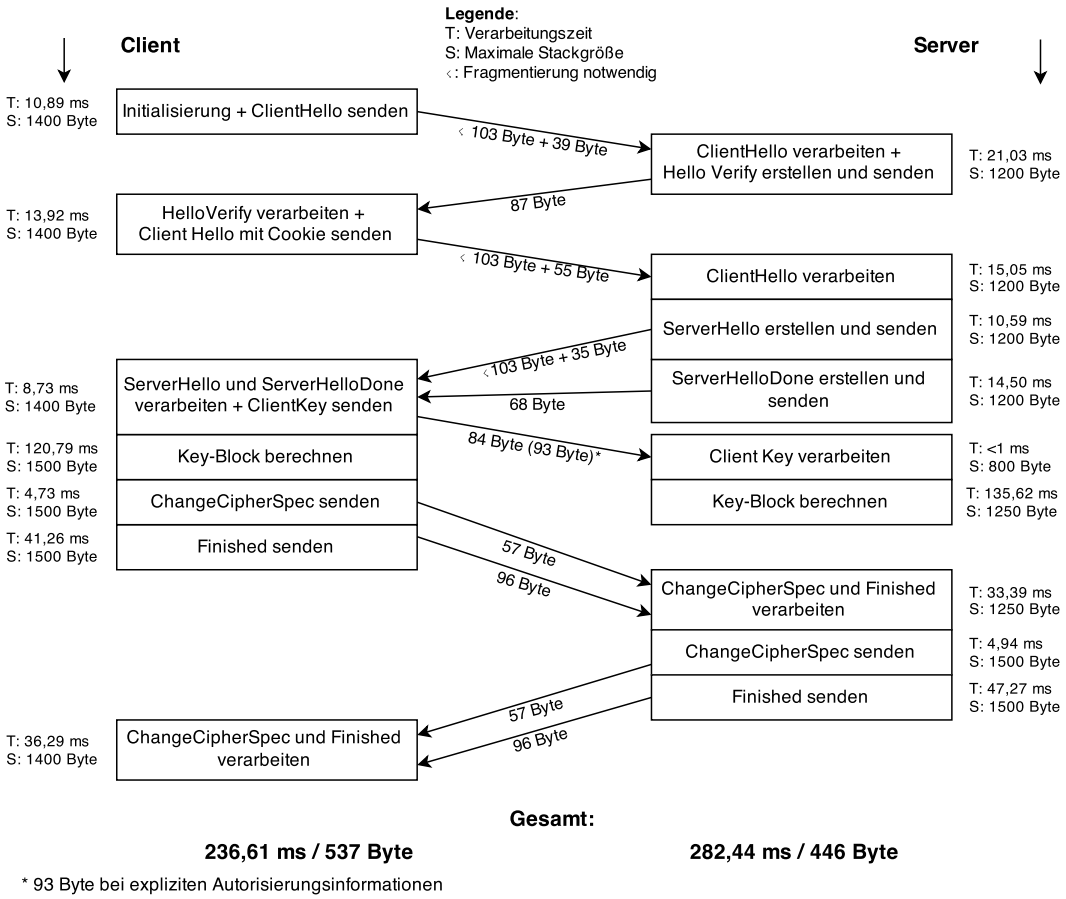

Abbildung 6.3: DTLS-Handshake zwischen Client und Server

wobei der Verbindungsaufbau vom Client initiiert wird. Der zeitliche Ablauf des DTLS-Handshakes und die Verarbeitungszeit der einzelnen Schritte ist in Abbildung 6.3 dargestellt. Auf der linken Seite werden die Verarbeitungsschritte des Clients und auf der rechten Seite die des Servers abgebildet. Zu jedem Schritt wird die Verarbeitungszeit und die maximale Stackgröße angegeben. Außerdem wird die Größe der ausgetauschten Fragmente dargestellt.

Im ersten Schritt sendet der Client eine ClientHello-Nachricht. Das Erstellen und Senden dieser dauert 10,89 ms. Die Nachricht muss in zwei Fragmente aufgeteilt werden. Diese sind 103 Byte und 39 Byte groß. Der Server verarbeitet die ClientHello-Nachricht und sendet eine HelloVerify-Nachricht an den Client. Diese enthält ein Cookie, um DoS-Angriffe zu erschweren. Da der Server das Cookie generieren muss, ist die Dauer dieses Schritts mit 21,03 ms vergleichsweise hoch. Die zu übertragenden Daten sind mit 87 Byte klein genug, um nicht fragmentiert zu werden. Der Server verarbeitet die HelloVerify-Nachricht und sendet erneut eine ClientHello-Nachricht. Da dieses Mal das Cookie mitgesendet werden muss, sind die zu übertragenden Fragmente mit 103 Byte und 55 Byte noch größer als beim ersten ClientHello. Der Client benötigt 13,92 ms zum Verarbeiten und Senden der Nachricht. Der Server benötigt anschließend 15,05 ms, um die ClientHello-Nachricht zu verarbeiten und 10,59 ms und 14,50 ms, um daraufhin die

ServerHello- und ServerHelloDone-Nachrichten zu senden. Die ServerHello-Nachricht muss ebenfalls in zwei Fragmente mit 103 Byte und 35 Byte aufgeteilt werden. Die ServerHelloDone-Nachricht passt mit 68 Byte in ein Frame. Der Client verarbeitet wiederum die Nachrichten vom Server und sendet die ClientKeyExchange-Nachricht. Dies dauert lediglich 8,73 ms.

Der Client sendet in der ClientKeyExchange-Nachricht, im `psk_identity`-Feld, das Ticket-Face im unveränderten CBOR-Format an den Server. Die Nachricht ist mit 84 Byte klein genug, um eine Fragmentierung zu vermeiden. Die Handshake-Nachricht wird gesendet, bevor das erste Mal eine Verschlüsselung stattfindet. Dadurch kann die Vertraulichkeit der Daten im Ticket-Face mit der verwendeten Schlüsselaustauschmethode (siehe Abschnitt 4.5) nicht sichergestellt werden. Die Authentizität und Integrität des Ticket-Faces ist allerdings sichergestellt, da eine Manipulation des Ticket-Faces durch den Client oder auf dem Übertragungsweg erkannt werden kann. Der Server berechnet den DTLS-PSK mit einer HMAC-Funktion. Bei einer Manipulation des Ticket-Faces würde ein zum Verifier des Clients verschiedener PSK berechnet und der DTLS-Verbindungsaufbau fehlschlagen.

Das Ticket-Face enthält im konkreten Fall 11 Byte Daten im CBOR-Format. [RFC4279] erlaubt eine maximale Größe der PSK-Identity von $2^{16} - 1 = 65535$ Byte. Allerdings legt dieser RFC auch fest, dass die PSK-Identity als UTF-8-Zeichenkette, in einem für Menschen lesbaren Format vorliegen muss [RFC4279, S.8]. In DCAF und damit auch im hier gezeigten Szenario wird das Ticket-Face allerdings im binären CBOR-Format übertragen. Die verwendete DTLS-Bibliothek tinydtls kommt allerdings auch mit Binärdaten in der PSK-Identity zurecht. Daher wurde dies hier wie in DCAF und abweichend zum [RFC4279] implementiert.

Der Server benötigt zur Verarbeitung der ClientKeyExchange-Nachricht weniger als eine Millisekunde. Anschließend erstellt der Server den für die Verbindung zu verwendenden Key-Block. Hierfür benötigt der Server einen DTLS-PSK, den er mit Hilfe des Ticket-Faces und dem mit SAM geteilten Schlüssel $K(SAM, S)$ mit der gleichen HMAC-Funktion erstellt, die zum Ausstellen des Tickets auf SAM verwendet wurde:

```
DTLS-PSK = HMAC_SHA256(Ticket-Face, K(SAM, S)) =
HMAC_SHA256( 0xa405181e06190e1007001000, 0xd8d507fab8eb1141b1172c28612a5605 ) =
0x7146d2dfe8a44e03b126b36758563d0d
```

Dabei erhält er bei einem unmanipuliertem Ticket-Face einen zum Verifier des Clients identischen Schlüssel. Die Berechnung des Key-Blocks dauert insgesamt 135,62 ms auf dem Server. Davon werden 14,80 ms für die Berechnung des DTLS-PSKs nach dem beschriebenen Verfahren benötigt.

Der Client berechnet, nachdem er die ClientKeyExchange-Nachricht gesendet hat, ebenfalls den zu verwendenden Key-Block. Da dem Client der DTLS-PSK in Form des Verifiers im Ticket bereits vorliegt, fällt die Berechnung von diesem weg. Daher benötigt der Client nur 120,79 ms für die Berechnung des Key-Blocks. Anschließend sendet der Client die ChangeCipherSpec-Nachricht, welche mit 57 Byte die kleinste Nachricht im Handshake ist und daher auch in nur 4,73 ms erstellt und gesendet wird. Nach dieser Nachricht werden alle weiteren Nachrichten verschlüsselt übertragen. Dazu gehört auch die im Anschluss gesendete Finished-Nachricht. Auch diese muss mit einer Größe von 96 Byte nicht fragmentiert werden. Da die Finished-Nachricht einen

Hash aller bisher gesendeten Handshake-Nachrichten enthält, dauert die Erstellung der Nachricht mit 41,26 ms vergleichsweise lang.

Der Server benötigt zum Verarbeiten der ChangeCipherSpec- und Finished-Nachricht 33,39 ms. Anschließend sendet er ebenfalls jeweils eine eigene ChangeCipherSpec- und Finished-Nachricht. Die Nachrichten haben die gleichen Paketgrößen wie zuvor beim Client und benötigen mit 4,94 ms und 47,27 ms ähnlich lang. Der Client benötigt ebenfalls mit 36,29 ms ähnlich lang für die Verarbeitung der Nachrichten. Der Handshake gilt mit diesem letzten Schritt als abgeschlossen und die DTLS-Verbindung wurde erfolgreich aufgebaut.

Während des gesamten Handshakes wächst der Stack auf den eingeschränkten Geräten zu keinem Zeitpunkt über 1500 Byte. Wie in Abschnitt 6.2 beschrieben, stehen auf den eingeschränkten Geräten während des Betriebs noch 4,16 KiB beziehungsweise 4,22 KiB zur Verfügung. Da nur in wenigen Fällen dynamische Speicherallokierung verwendet und in dieser schnell wieder freigegeben wird, kam es während des Handshakes nicht zu Engpässen beim Arbeitsspeicher.

Insgesamt wurden während des Handshakes in den Nachrichten 983 Byte ausgetauscht. Hierzu kommen allerdings noch für jeden gesendeten Frame ein ACK-Frame des Kommunikationspartners mit 5 Byte. Für die 13 gesendeten Frames fallen demnach 65 Byte zusätzlich an. Insgesamt werden somit 1048 Byte während des DTLS-Handshakes übertragen.

Auf dem Client benötigt der Handshake insgesamt eine Verarbeitungszeit von 236,61 ms und auf dem Server von 282,44 ms. Insgesamt dauert die Verarbeitung auf den eingeschränkten Akteuren während des Handshakes demnach 519,05 ms. In der verwendeten Simulationsumgebung entspricht dies auch in etwa der gesamt benötigten Zeit des Handshakes, da hier kaum zusätzlich Übertragungsverzögerungen bei der Kommunikation auftreten. Darüber hinaus gilt dies nur für den idealen Fall, in dem keine Handshake-Nachrichten verloren gehen und erneut übertragen werden oder in falscher Reihenfolge ankommen und sortiert werden müssen.

Durch den erfolgreichen Aufbau der DTLS-Verbindung authentifizieren sich der Client und der Server gegenseitig. Sobald die Verbindung steht, kann der Kühlcontainer anfangen kontinuierlich die Sensorwerte der Bananenschachtel auszulesen. Die DTLS-Verbindung kann im Szenario beliebig lange offen gehalten werden. Würde der Client allerdings Daten von vielen verschiedenen Servern abfragen müssen, könnte er, aufgrund seiner Eingeschränktheit, nicht zu jedem Server permanent eine DTLS-Verbindung geöffnet halten. In einem solchen Szenario müsste der Client regelmäßig DTLS-Verbindungen schließen und später erneut aufbauen. Hierdurch würde ein beachtlicher Overhead bei jeder Anfrage für den DTLS-Handshake anfallen. Bei dem Neuaufbau einer DTLS-Verbindung zum Server kann der Client allerdings auf das erneute Übertragen des Ticket-Faces verzichten, da der Server sich die zuletzt verwendeten Tickets merkt.

Authorized Resource Request Message

Im vorliegenden Szenario kann die DTLS-Verbindung permanent offen gehalten werden, sodass für das kontinuierliche Abfragen der Sensorwerte lediglich eine Verschlüs-

selung der Anfrage und kein neuer Handshake stattfinden muss. Die CoAP-Anfrage an den Server enthält die gleichen Informationen wie bei der, in Abschnitt 6.3.3 beschriebenen, Unauthorized Resource Request Message und ist ebenfalls 29 Byte groß. Allerdings ist das gesendete Netzwerkpaket mit insgesamt 83 Byte um 29 Byte größer als bei der unautorisierten Anfrage. Der Overhead fällt durch den DTLS-Header und das Padding bei der Verschlüsselung an. Für die Verschlüsselung der Anfrage benötigt der Client 1,22 ms. Währendessen wächst der Stack auf maximal 850 Byte. Insgesamt dauert das Erstellen und Senden der Anfrage 11,72 ms. Im Falle dieser GET-Anfrage mit relativ kurzem URI und ohne Payload ist das gesendete Paket mit 83 Byte klein genug, um eine Fragmentierung zu vermeiden. Dies kann allerdings bei einer Anfrage mit längerem URI oder mit einem Payload anders aussehen.

Auf der Seite des Servers wird die Nachricht empfangen und zunächst durch die DTLS-Schicht in 1,22 ms entschlüsselt. Anschließend wird die Nachricht an die CoAP-Bibliothek weitergeleitet, welche sie verarbeitet und eine festgelegte Callback-Funktion im Anwendungsprogramm aufruft. Dazu benötigt die Bibliothek libcoap 0,52 ms. Während der Entschlüsselung und der Verarbeitung wächst der Stack auf maximal 1200 Byte. Auf Anwendungsebene wird vom Server zunächst das zu dieser Verbindung gehörende Ticket-Face aus den gespeicherten herausgesucht und anschließend die darin enthaltenen Autorisierungsinformationen mit den Informationen in der CoAP-Anfrage verglichen. Die Daten im Ticket-Face liegen, wie in Unterabschnitt 4.6.4 beschrieben, in einem Format vor, dass vom Server in wenigen Schritten ausgewertet werden kann.

Zunächst prüft der Server, ob das Ticket bereits widerrufen wurde. Im konkreten Fall wird geprüft, ob die Sequenznummer 0 widerrufen wurde. Da bisher noch keine Tickets widerrufen wurden, ist die Sequenznummer 0 noch die untere Grenze des Sliding Windows. Daher muss der Server an dieser Stelle eine Bitverschiebungsoperation, eine logische UND-Verknüpfung und einen Ganzzahlenvergleich durchführen. Für diese Operationen benötigt der Server weniger als einen Tick des Realtime-Timers und somit weniger als 0,03 ms.

Anschließend prüft der Server, ob die Gültigkeitsdauer des Tickets bereits abgelaufen ist. Hierfür benötigt der Server die aktuelle Zeit beziehungsweise die Sekunden, die seit dem Start des Gerätes vergangen sind. Diese stellt Contiki zur Verfügung. Zum Bestimmen, ob das Ticket bereits abgelaufen ist, ist anschließend eine Addition und wiederum ein Ganzzahlenvergleich durchzuführen. Auch diese Operationen benötigen insgesamt auf dem Server weniger als 0,03 ms.

In diesem ersten Anwendungsfall wird eine implizite Autorisierung verwendet, das heißt das Ticket erlaubt den Zugriff auf alle Ressourcen des Servers und enthält keine expliziten Autorisierungsinformationen. Daher muss der Server diese an dieser Stelle auch nicht überprüfen. Eine Auswertung der Durchsetzung der Autorisierungsinformationen findet in Abschnitt 6.3.5 statt. Insgesamt benötigt die Auswertung der Autorisierungsregeln auf dem Server weniger als 0,43 ms, wobei 0,40 ms für das Parsen des Ticket-Faces durch die Bibliothek cn-cbor und insgesamt weniger als 0,03 ms für die Durchsetzung der Zugriffskontrolle benötigt werden. Sobald die Zugriffskontrolle auf dem Server abgeschlossen ist, wird die CoAP-Antwort mit den enthaltenen Sensordaten erstellt und anschließend über DTLS verschlüsselt an den Client gesendet. Die CoAP-Nachricht enthält im konkreten Fall lediglich die Temperatur als textuelle Repräsentation einer Ganzzahl. Der gesendete Frame hat eine Größe von 79 Byte und

muss daher nicht fragmentiert werden.

Der Client benötigt zur Verarbeitung und Entschlüsselung der empfangenen Antwort bis zum Zeitpunkt an dem der Sensorwert zur weiteren Nutzung zur Verfügung steht 1,68 ms. Der Client kann nun kontinuierlich Authorized Resource Request absenden, um aktuelle Sensorwerte zu erhalten. Die Weiterverarbeitung des Sensorwertes beziehungsweise die Regulierung des Klimas im Kühlcontainer wird an dieser Stelle nicht ausgewertet, da es nicht zum Protokollablauf von DCAF gehört.

Insgesamt benötigt der gesamte Authorized Resource Request vom Erstellen der Anfrage auf dem Client bis nach der Verarbeitung der Antwort vom Server 22,74 ms. Allerdings ist bei diesem Wert wiederum darauf hinzuweisen, dass dieser in der Simulation erreicht wurde, in der nur sehr geringe Übertragungsverzögerungen und keine Paketverluste auftraten. Während des Authorized Resource Requests fällt kein zusätzlicher Overhead durch DCAF bei der Übertragung an. Lediglich beim DTLS-Handshake fällt das zu übertragende Ticket-Face als Overhead an.

6.3.5 Übergabe und Weitertransport

Um auch das Widerrufen der Tickets und die Verwendung von expliziten Zugriffsregeln zu evaluieren, wird ausschnittsweise der Weitertransport der Bananenschachtel an Land in einem LKW betrachtet. Hierbei wird lediglich auf die Unterschiede zum bisher evaluierten Anwendungsfall eingegangen. Im Folgenden wird davon ausgegangen, dass der, in Unterabschnitt 6.3.2 beschriebene, Vorlauf und die Verladung für den Weitertransport bereits stattfanden. Bei dem Vorlauf wurde zunächst, wie im vorherigen Fall, eine implizite Zugriffsregel von ROP auf SAM festgelegt und ein Ticket aus dieser mit der Sequenznummer 1 ausgestellt.

Widerrufen eines Tickets

Um das Widerrufen eines Tickets auszuwerten, ändert ROP im Szenario seine Meinung und ändert die Zugriffsregel, sodass nur noch ein expliziter Zugriff auf festgelegte Ressourcen erlaubt werden soll. Die Evaluation der expliziten Autorisierung und der Durchsetzung der Autorisierungsregeln findet sich in Abschnitt 6.3.5. Beim Ändern der Zugriffsregel muss ROP entscheiden, ob Tickets, die bereits aus der vorherigen impliziten Regel erstellt wurden, widerrufen werden sollen. Im konkreten Fall betrifft dies das zuvor ausgestellte Ticket mit der Sequenznummer 1.

SAM versucht daraufhin fortlaufend den von der Ticket-Widerrufung betroffenen Server zu kontaktieren, um ihm eine Ticket Revocation Message zukommen zu lassen. Es wird im Folgenden davon ausgegangen, dass die Bananenschachtel sich noch im Zielhafen befindet und dort eine Verbindung zu ihrem Autorisierungsmanager aufbauen kann. Falls eine Verbindung nicht möglich wäre, könnte der Server die Widerrufung beim Zugriff nicht durchsetzen und würde das Ticket bis zum Ablauf der Gültigkeit weiter akzeptieren. Daher ist bei der Wahl der zu verwendenden Gültigkeit der Tickets eine Abwägung zwischen dem Komfort, der eine lange Gültigkeit bietet, und der Sicherheit, die kürzere Gültigkeiten bietet, zu treffen. Außerdem ist je nach Einsatzszenario zu entscheiden, inwieweit der Server und sein Autorisierungsmanager miteinander

kommunizieren können und inwieweit es akzeptabel ist, dass widerrufene Tickets unter Umständen noch bis zum Ablauf ihrer Gültigkeitsdauer verwendet werden können.

SAM verwendet DTLS mit dem Schlüssel $K(SAM, S)$ als PSK, um eine sichere Verbindung zum Server aufzubauen. Der Server kann den Verbindungsversuch anhand der von SAM gesendeten PSK-Identity von Clientanfragen unterscheiden. Eine erneute Auswertung des DTLS-Handshakes findet an dieser Stelle nicht statt. Stattdessen wird im Folgenden davon ausgegangen, dass die DTLS-Verbindung zwischen SAM und dem Server erfolgreich hergestellt wurde.

Die von SAM verschickte CoAP-Nachricht an den Server enthält im konkreten Fall lediglich die eine Sequenznummer des zu widerrufenden Tickets:

```
SAM -> Server
POST /revocations
[1]
```

Die Nachricht wird als Confirmable markiert, sodass SAM über die erfolgreiche Zustellung informiert wird oder anderenfalls auf CoAP-Ebene eine erneute Übertragung stattfindet. Das von SAM gesendete Fragment enthält im Sensornetz, inklusive des DTLS-Headers und den darunter liegenden Schichten, insgesamt 103 Byte und muss daher nicht fragmentiert werden.

Die Verarbeitungszeit auf dem Server auf der DTLS- und CoAP-Schicht unterscheidet sich nicht zu dem in Abschnitt 6.3.4 betrachteten Fall und wird daher an dieser Stelle nicht erneut ausgewertet. Bei der Verarbeitung auf Anwendungsebene legt der Server die empfangene und zu widerrufende Sequenznummer im Sliding Window ab. Hierfür muss im konkreten Fall bei der Sequenznummer 1 das Fenster nicht verschoben werden, sondern lediglich das zweite Bit des Fensters gesetzt werden. Diese Operation, inklusive der dafür notwendigen Vergleichsoperationen und das Parsen des CBOR-Payloads, benötigt auf dem Server lediglich 0,12 ms. Das Sliding-Window insgesamt benötigt jederzeit durch die, in Unterabschnitt 4.7.1 beschriebene, Speicherung lediglich zwei Mal 4 Byte (32 Bit), unabhängig von der Anzahl der widerrufenen Tickets.

Versucht ein Client nun eine DTLS-Verbindung aufzubauen und sendet das widerrufene Ticket mit der Sequenznummer 1 als PSK-Identity mit, so führt dies nicht sofort zum Fehlschlagen des Verbindungsaufbaus, denn, ob ein Ticket widerrufen wurde, wird erst auf Anwendungsebene, nachdem eine Authentifizierung über DTLS stattfand, geprüft. Der Aufwand zum Prüfen, ob ein Ticket widerrufen wurde, wurde bereits in Abschnitt 6.3.4 ausgewertet, weshalb dies an dieser Stelle nicht erneut geschehen soll. Da das verwendete Ticket auf dem Server als widerrufen markiert ist, lehnt dieser die CoAP-Anfrage ab und erlaubt keinen Zugriff auf die Ressource.

Durch das Verwehren des Zugriffs kann der Client feststellen, dass das Ticket nicht mehr gültig ist. In diesem Fall kann der Client versuchen ein neues Ticket ausgestellt zu bekommen. Dies kann entweder autonom vom Client geschehen oder durch COP manuell angestoßen werden. Im konkreten Fall fragt der Client selbständig ein neues Ticket an und erhält diesmal ein explizites Ticket.

Explizite Autorisierung

Wie in Abschnitt 6.3.5 geschrieben, ändert COP die erstellte Zugriffsregel in eine explizite Regel, die lediglich die Abfrage einer bestimmten Ressource auf dem Server erlaubt. Die über das Webinterface an SAM gesendete Anfrage sieht prinzipiell aus wie in Unterabschnitt 6.3.2, mit dem Unterschied, dass diesmal noch die expliziten Autorisierungsinformationen für die Ressource `/temp/1` im in Abschnitt 4.6.2 definierten Format enthalten sind. Die Anfrage hat für das konkrete Szenario folgenden Inhalt:

```
ROP -> SAM
POST /rules/Transportregel\%202 HTTP/1.1
{
  "id": "Transportregel 2",
  "subject": "mJ6N/FQ55PDPfi0WqMzf3W2uWgk=",
  "resources": [
    {
      "server": "aaaa::200:0:0:2",
      "methods": 1,
      "resource": "temp/1"
    }
  ],
  "priority": 0,
  "conditions": []
}
```

Wie in Abschnitt 6.3.5 beschrieben, versucht der Client, nachdem sein widerrufenes Ticket vom Server abgelehnt wurde, ein neues Ticket zu bekommen. Beim Ausstellen des neuen Tickets (Sequenznummer 2) wird dieses, wie in Unterabschnitt 4.6.4 beschrieben, in das CBOR-Format umgewandelt, wobei darauf geachtet wird, dass das Ticket möglichst klein bleibt. Die Autorisierungsinformationen werden in dem, in Abschnitt 4.4 beschriebenen, AI-Format in das Ticket geschrieben. Eventuelle lokale Bedingungen würden ebenfalls im Ticket enthalten sein, sind in diesem konkreten Fall allerdings nicht erforderlich. Das neu ausgestellte Ticket mit den expliziten Autorisierungsinformationen ist im CBOR-Format 45 Byte groß und enthält folgende Daten.

```
CAM <- SAM
{
  Face: {
    SAI: ['temp/1',1],
    TS: 24500,
    LIFETIME: 86400,
    DTLS_PSK_GEN: 0,
    SEQ_NR: 2
  },
  Verifier: h'e05c02385e7fe951d30ef1382cccc180'
}
```

Bei der Verwendung des neuen Tickets durch den Client muss der Server nun zusätzlich auf Anwendungsebene, nachdem die DTLS-Verbindung erfolgreich aufgebaut wurde, die Autorisierungsinformationen im Ticket-Face überprüfen. Die Informationen liegen als CBOR-Array vor, wobei jedes Element sowohl den autorisierten URI als auch die für den URI autorisierten CoAP-Methoden enthält. Der Server muss dieses Array durchlaufen und bei jedem Element zur Auswertung einen Zeichenkettenvergleich

durchführen, um den URI mit dem angefragten URI zu vergleichen und ein logisches UND und einen Integer-Vergleich zur Auswertung der CoAP-Methode durchführen. Im konkreten Fall hat das CBOR-Array nur ein Element, daher muss auch lediglich einmal die oben genannten Operationen ausgeführt werden. Der Server benötigt zur vollständigen Durchsetzung der Informationen im Ticket-Faces in diesem konkreten Fall 0,61 ms. Davon fallen 0.21 ms für die Auswertung der Autorisierungsinformationen und zusätzlich 0,40 ms durch die, in Abschnitt 6.3.4 beschriebene, restliche Auswertung, inklusive dem Parsen des Ticket-Faces im CBOR-Format, an.

6.4 Diskussion

Der in Abschnitt 6.3 durchgespielte Ablauf und die Auswertung des Nutzungsszenarios zeigen, dass die entworfenen und implementierten Akteure im Sinne des DCAF-Protokollablaufs miteinander kommunizieren können. Auf den eingeschränkten Akteuren wird für die Implementierung von DCAF auf der Anwendungsebene, mit 1,84 KiB auf dem Server und 2,37 KiB auf dem Client, vergleichsweise wenig Programmspeicher benötigt. Wobei zu beachten ist, dass für DCAF immer eine Möglichkeit zum Verarbeiten von CBOR vorgesehen werden muss, da DCAF dies als zentrales Datenformat der Nachrichten vorsieht. Zusammen mit der verwendeten CBOR-Bibliothek werden für DCAF auf dem Server 5,05 KiB und auf dem Client 5,58 KiB Programmspeicher benötigt.

Auch der benötigte statische Arbeitsspeicher ist, mit unter 0,5 KiB, gering. Im Vergleich zu den eingesetzten Bibliotheken libcoap und tinydtls fallen sie kaum ins Gewicht. Insgesamt bleibt auf den eingeschränkten Geräten, mit 35,87 KiB auf dem Server und 46,55 KiB auf dem Client, noch ausreichend Programmspeicher für auf DCAF aufbauende Anwendungen übrig. Vom Arbeitsspeicher stehen Anwendungen, die auf DCAF aufbauen, noch jeweils etwas über 4 KiB zur Verfügung, wobei hiervon mindestens 1,5 KiB für den im Szenario maximal benötigten Stackspeicher abgezogen werden müssen. Außerdem muss noch ausreichend Speicher für die dynamische Allokierung von Speicher, die beispielsweise von den Bibliotheken tinydtls und cn-cbor verwendet wird, zur Verfügung stehen.

Das Hauptziel von DCAF ist der Aufbau einer Ende-zu-Ende verschlüsselten Verbindung zwischen zwei eingeschränkten Geräten aus unterschiedlichen Sicherheitsdomänen und die Delegation von Authentifizierungs- und Autorisierungsaufgaben an weniger eingeschränkte Autorisierungsmanager. Dieses konnte im demonstrierten dezentralen Nutzungsszenario erfüllt werden. Client und Server können während des Transports autonom und sicher miteinander kommunizieren. Zur Zugriffskontrolle sind vom Server lediglich einfache Operationen aufzuführen, die auch von einem eingeschränkten Gerät erfüllbar sind. Außerdem konnten die in Kapitel 3 definierten Anforderungen an die Autorisierung nach feingranularer Einstellung der Zugriffsregeln und einem Ablaufdatum beziehungsweise der Möglichkeit zu Widerrufen umgesetzt und erfolgreich durchgeführt werden.

Das Internet of Things wird, wie in Abschnitt 2.1 beschrieben, aus potenziell vielen Millionen oder sogar Milliarden Teilnehmern bestehen. Auch innerhalb von einzelnen Domänen wird es noch viele Teilnehmer geben. Die Netze im IoT können sich spontan

verändern oder erst aufgebaut werden. Hierfür und für die große Zahl an Teilnehmern werden Lösungen benötigt, die gut skalieren. Dies gilt insbesondere auch für die zu verwendende Verschlüsselung.

In der Literatur wird häufig verallgemeinernd geschrieben, dass symmetrische Verschlüsselung bei mehreren Teilnehmern, die untereinander kommunizieren wollen, generell nicht gut skaliert [Kiz13]. Es wird davon ausgegangen, dass jeder Teilnehmer für jeden anderen Teilnehmer einen geheimen Schlüssel bereithalten muss (für n Teilnehmer $n * (n-1)$ Schlüssel insgesamt) und diese Verwaltung bei großer Teilnehmerzahl nicht skaliert. In DCAF wird zwar auch ausschließlich symmetrische Verschlüsselung verwendet, allerdings wird das Problem der Schlüsselverwaltung durch die dezentrale Architektur und die Delegierung der Autorisierung gelöst. Durch die verteilte Schlüsselerzeugung und das Ausstellen von Tickets muss kein Akteur in DCAF wie im oben geschilderten Fall Schlüssel zwischen allen Teilnehmern aufbewahren. SAM muss nur Schlüssel in Tickets verwalten, die explizit angefragt wurden. Darüber hinaus ist SAM nicht so sehr eingeschränkt, daher stellt die Verwaltung der Tickets kein Problem für diesen dar. Auf dem Server müssen, durch die verteilte Schlüsselerzeugung und die Delegierung der Autorisierungsaufgaben an SAM, überhaupt keine Schlüssel von Clients permanent gespeicherten werden, da diese erst bei Bedarf mit Hilfe des Tickets und dem Schlüssel $K(SAM, S)$ generiert werden. Der Server kann daher potenziell unbegrenzt vielen Clients Zugriff zu seinen Ressourcen gewähren, ohne dass hierdurch der Overhead durch zu speichernde Schlüssel wächst. Es kann daher festgehalten werden, dass die symmetrische Verschlüsselung zwischen Clients und Servern in DCAF insbesondere auf dem eingeschränkten Server gut skaliert.

6.4.1 DTLS

Der eingeschränkte Server kann zwar allgemein unbegrenzt vielen verschiedenen Clients Zugriff gewähren, ist allerdings hinsichtlich der gleichzeitig möglichen Zugriffe beschränkt. Insbesondere die Zustände, die für eine aufgebaute DTLS-Verbindung im Speicher gehalten werden müssen, führen dazu, dass der Server nur wenige Verbindungen gleichzeitig verwalten kann. Der Server im Szenario war im Test bereits mit mehr als zwei gleichzeitigen Verbindungen überlastet. Hierdurch ist bei vielen Teilnehmern ein häufiges Schließen und Neuaufbauen der DTLS-Verbindung notwendig. Da der DTLS-Hanshake allerdings, wie in Unterabschnitt 6.3.4 beschrieben, einen Großteil der insgesamt benötigten Rechenzeit und Übertragunskapazität einer Anfrage ausmacht, ist dieses Vorgehen nicht in jedem Szenario zu empfehlen. Um nicht jedes Mal den gesamten DTLS-Handshake erneut ausführen zu müssen, kann möglicherweise *DTLS-Session-Resumption* verwendet werden. [RFC5077] beschreibt eine Möglichkeit, um TLS-Verbindungen wieder aufzunehmen, ohne dass Zustände auf Serverseite gespeichert werden müssen. In der aktuellen Form ohne Session-Resumption skaliert der Server auf dieser Ebene bei vielen Clients nicht gut.

Der Aufwand, der für den DTLS-Verbindungsaufbau auf den eingeschränkten Geräten betrieben werden muss, ist im Verhältnis zum Aufwand für die eigentliche CoAP-Anfrage, wie in Unterabschnitt 6.3.4 beschrieben, sehr hoch. Auf der einen Seite lässt sich argumentieren, dass dies der Preis für eine sichere Kommunikation ist. Allerdings ist auf der anderen Seite ein Optimierungsbedarf an dieser Stelle ebenfalls nicht gänz-

lich von der Hand zu weisen. Eine Möglichkeit eingeschränkte Geräte beim DTLS-Handshake zu entlasten wird in [Hum+14] vorgeschlagen. Dort wird ebenfalls von einer dezentralen Architektur mit eingeschränkten Servern ausgegangen. Der Server kann in dem Vorschlag die Durchführung des DTLS-Handshakes an einen weniger eingeschränkten Akteur delegieren und diesen selbst nicht durchführen. Stattdessen verwendet der Server DTLS-Session-Resumption, um die vom weniger eingeschränkten Akteur aufgebaute Verbindung wiederaufzunehmen. Übertragen auf die DCAF-Architektur könnte dies bedeuten, dass der Server den DTLS-Verbindungsaufbau an seinen SAM delegiert. SAM würde dann stellvertretend für den Server den DTLS-Handshake mit dem Client durchführen und alle nötigen Informationen, die zur Wideraufnahme der Verbindung nötig sind, wie die generierten Schlüssel und Parameter der Verbindung, mit in das Ticket schreiben. Der Client würde diese an den Server weiterleiten und dieser könnte die DTLS-Verbindung wieder aufnehmen. Hierbei müsste allerdings beachtet werden, dass eine direkte Kommunikation von Client und SAM in DCAF nicht direkt vorgesehen ist und auch im Szenario nicht jederzeit möglich wäre. Darüber hinaus würden die vielen zusätzlichen Informationen im Ticket dieses sehr groß werden lassen, sodass während der Übertragung eventuell eine Fragmentierung auf den unteren Schichten stattfinden würde, die wiederum eigene Probleme mit sich bringen würde.

In Abschnitt 5.5 wurde beschrieben, dass bei der Kommunikation eines eingeschränkten Gerätes mit einem weniger eingeschränkten Gerät über DTLS ein Timing-Problem auftreten kann, wenn während des Handshakes vom weniger eingeschränkten Gerät in kurzer Folge mehrere Nachrichten nacheinander gesendet werden. Begünstigt wird dieses Problem durch die in der Simulation fehlende oder nur sehr kurze Übertragungsverzögerung. In dieser Arbeit fand kein Test auf echter Hardware statt, daher kann das Vorhandensein des Problems auf echter Hardware weder bestätigt noch ausgeschlossen werden. Eine Möglichkeit, um das Problem auf Protokollebene anzugehen, besteht darin die einzelnen Nachrichten eines Flights im Handshake in einem DTLS-Record zusammenzufassen, wie es in [RFC5246] vorgesehen ist. Allerdings vergrößert sich hierdurch wiederum die Größe des zu sendenden Netzwerkpakets, wodurch eventuell größere Puffer auf dem eingeschränkten Gerät notwendig sind und gegebenenfalls eine Fragmentierung auf der DTLS-Schicht notwendig ist.

Weitere Probleme und Besonderheiten, die beim Einsatz von DTLS in Sensornetzen zu beachten sind, finden sich in [Har14b] und [TF15]. Zwei von diesen Problemen sollen im Folgenden beschrieben werden, da diese auch im ausgewerteten Nutzungsszenario beobachtet werden konnten.

Da DTLS über das unzuverlässige UDP eingesetzt wird, können Nachrichten während der Übertragung verloren gehen. Während des Handshakes sieht DTLS hierfür einen Retransmission-Mechanismus vor, durch den verloren gegangene Handshake-Nachrichten erneut übertragen werden. Allerdings müssen hierbei, wenn während eines Flights mehrere Nachrichten gesendet werden, alle bisher während dieses Flights gesendeten Nachrichten erneut übertragen werden. Dies kann insbesondere in Netzen mit hohem Paketverlust, wie es in Sensornetzen häufig der Fall ist, zu einem erhöhten Nachrichtenaufkommen führen.

Eine weitere Eigenschaft von DTLS, welche sich negativ in Sensornetzen mit eingeschränkten Geräten auswirken kann ist, dass zum Abschluss des Handshakes in der

Finished-Nachricht sowohl vom Client als auch vom Server ein Hash aus allen zuvor gesendeten Handshake-Nachrichten gesendet werden muss. Hierfür wird ein Puffer benötigt, der alle Handshake-Nachrichten aufnehmen kann. Dies könnte auf eingeschränkten Geräten, denen weniger Arbeitsspeicher oder Programmspeicher als dem verwendeten WiSMote zur Verfügung steht, zu Engpässen führen.

6.4.2 Autorisierungsinformationen und Zugriffsregeln

Die bei expliziter Autorisierung in ausgestellten Tickets enthaltenen Autorisierungsinformationen im AI-Format erlauben eine feingranulare Zugriffskontrolle. Außerdem ist das Format flexibel, sodass potenziell beliebig viele Ressourcen und Methoden auf diesen in einem AIF-Objekt enthalten sein können. Allerdings wächst das AIF-Objekt und somit das gesamte Ticket bei vielen enthaltenen Ressource-Methoden-Paaren recht schnell. Große Tickets können, wie in Abschnitt 6.3.5 beschrieben, ein Problem darstellen, da Fragmentierung auf der 6LoWPAN-Schicht auftreten kann. Daher skalieren die Autorisierungsinformationen im verwendeten AIF nicht besonders gut. Im AIF-Objekt sind insbesondere die URIs, die als Zeichenkette repräsentiert werden, für die Größe verantwortlich, denn die CoAP-Methoden in der, in Abschnitt 4.6.2 vorgestellten, Darstellung belegen in CBOR-Darstellung für jede Ressource nur 1 Byte. Daher könnte zur Reduzierung der Größe eine alternative Darstellung der Ressource-URI im AIF-Objekt verwendet werden.

Beispielsweise könnte eine verlustfreie Kompression ähnlich zu der im *Lempel-Ziv-Welch-Kompressionsalgorithmus (LZW)*[1] verwendeten werden. Bei dieser werden Datenstücke durch zuvor festgelegte Ganzzahlen ersetzt, sodass diese anschließend an Stelle der längeren Datenstücke verwendet werden können. Bezogen auf DCAF könnte dies umgesetzt werden, indem SAM und der Server zuvor entweder untereinander oder mit Hilfe eines Verzeichnisdienstes eine Zuordnung von Ressourcen zu Ganzzahlen vornehmen und anschließend im AIF-Objekt des Tickets nur noch die Zahl verwendet werden müsste. Für das konkrete Szenario mit der abgefragten Ressource `temp/1"`, die beispielsweise durch die Zahl 0 substituiert werden könnte, würde eine Reduktion des AI-Formats bei nur einer enthaltenen Ressource von 8 auf 3 Byte erreicht werden.

Eine weitere Möglichkeit die Autorisierungsinformationen im Ticket möglichst kompakt zu halten könnte darin bestehen Gruppen oder Rollen für Ressourcen zu verwenden. Hierzu müssten SAM und der Server sich zuvor über die zu verwendenden Rollen-/Gruppenbezeichnungen und die Zuordnung von Ressourcen zu Rollen oder Gruppen verständigen. Im Ticket könnten statt der URIs von Ressourcen mehrere Ressourcen durch den Rollen-/Gruppenbezeichner ausgedrückt werden.

Die auf SAM definierbaren Zugriffsregeln unterscheiden sich inhaltlich lediglich an einer Stelle von den Informationen im Ticket. Auf SAM können in einer Regel Ressourcen über mehrere Server zusammengefasst werden, im Ticket hingegen immer nur für einen Server. Auf SAM kann prinzipiell, wie in Abschnitt 4.6.2 beschrieben, jedes beliebige Autorisierungsmodell verwendet werden. Es sind insbesondere auch ausdrucksstärkere Modelle als das hier verwendete Modell möglich. Die umgesetzten Zugriffsregeln sind in ihrer Struktur und ihrem Inhalt sehr statisch und können beispielsweise nicht beliebig

[1] LZW-Algorithmus: http://users.ece.utexas.edu/~ryerraballi/MSB/pdfs/M2L3.pdf (abgerufen am 08.06.2015)

durch logische Operatoren verknüpft werden. Dies könnte unter anderem durch die Verwendung von Beschreibungssprachen, wie sie in in Abschnitt 2.7 erwähnt wurden, erreicht werden.

6.4.3 Tickets

Im vorherigen Abschnitt wurde beschrieben, dass Zugriffsregeln auf SAM über mehrere Server hinweg erstellt werden können, dies im Ticket allerdings nicht möglich ist. In DCAF kann ein Ticket immer nur für einen expliziten Server ausgestellt werden, da die Erstellung des Verifiers beziehungsweise des Pre-Shared-Keys vom zwischen SAM und dem Server geteilten Schlüssel $K(SAM, S)$ abhängt. Da die verschiedenen Server aus Sicherheitsgründen jeweils unterschiedliche Schlüssel verwenden sollten, können die Autorisierungsinformationen im Ticket sich immer nur auf diesen Server beziehen und nicht geräteübergreifend Ressourcen autorisieren. Falls dies in einem Szenario allerdings gefordert wäre, könnte DCAF dahingehend erweitert werden, dass SAM in ein Ticket mehrere Verifier für unterschiedliche Server schreibt, sodass der Client das gleiche Ticket-Face für die Verbindung zu verschiedenen Servern verwenden kann.

In Abschnitt 4.6 wurde für Tickets die Sequenznummer eingeführt, um das Widerrufen der Tickets zu ermöglichen. Die Sequenznummer besteht aus einer 32-Bit Ganzzahl, die von SAM individuell für jeden Server verwaltet wird und mit jedem neuen Ticket inkrementiert wird. In einem Szenario mit sehr vielen Teilnehmern und sehr vielen ausgestellten Tickets können die Sequenznummern theoretisch ausgehen und überlaufen. Auch wenn hierfür bei einer 32-Bit Sequenznummer ca. 4,3 Milliarden Tickets ausgestellt werden können und dies auch in Szenarien des Internet of Things, in dem es potenziell viele Milliarden Teilnehmer geben kann, sicherlich nicht an der Tagesordnung ist, sollte dies berücksichtigt werden. Die einfachste Lösung hierfür wäre, statt einer 32-Bit eine 64-Bit Ganzzahl zu verwenden. Dadurch würden zwar für jedes denkbare Szenario ausreichend Sequenznummern zur Verfügung stehen, allerdings würde der Overhead im Ticket und in Ticket Revocation Messages steigen. Auch auf dem Server würde für das Sliding-Window mehr Speicher benötigt. Eine andere Möglichkeit wäre das geordnete und zwischen SAM und dem Server koordinierte Zurücksetzen der Sequenznummer auf Null. Um hierbei sicherzustellen, dass alte Sequenznummern nach dem Zurücksetzen nicht wieder verwendet werden können, könnte zu dieser Gelegenheit der zwischen SAM und dem Server geteilte Schlüssel $K(SAM, S)$ ausgetauscht werden. Hierdurch kann für alte Tickets auf dem Server kein gültiger PSK mehr erzeugt werden.

Der Zeitstempel im Ticket kann, wie in Abschnitt 4.4 beschrieben und auch hier im Szenario verwendet, dezentral vom Server generiert werden und über den Client bis zum SAM gesendet werden. Der Zeitstempel wird bei der Zugriffskontrolle vom Server verwendet, um zu überprüfen, ob die Gültigkeit des Tickets bereits abgelaufen ist. Daher kann eine Manipulation des Zeitstempels zu einer Beeinträchtigung der Sicherheit des Systems führen. Im verwendeten Ablauf gibt es gleich zwei Punkte, an denen eine Manipulation des Zeitstempels durch Akteure außerhalb der Sicherheitsdomäne des ROP beziehungsweise des Servers und SAMs möglich ist.

Zum einen erhält der Client den Zeitstempel als Antwort auf den Unauthorized Resource Request. Da kein Verfahren eingesetzt wird, um die Integrität und Authentizität

des Zeitstempels zu schützen, kann der Client diesen beliebig verändern und über seinen Autorisierungsmanager weiter an SAM senden. SAM hat keine Möglichkeit diese Manipulation zu erkennen. Würde der Client den Zeitstempel beispielsweise so manipulieren, dass er weiter in der Zukunft liegt, so kann er die Gültigkeit des Tickets beliebig verlängern. Als Gegenmaßnahme könnte der Server lediglich prüfen, ob der Zeitstempel in der Zukunft liegt und diesen dann ablehnen. Allerdings kann der Client den Zeitstempel auch so manipulieren, dass zwar eine Verlängerung des Tickets möglich ist, aber der Zeitstempel noch nicht in der Zukunft liegt. Daher ist diese Gegenmaßnahme nicht vollständig wirksam.

Die andere Stelle, an der eine Manipulation auch durch eine dritte Partei möglich ist, ist die SAM Information Message. Dieser Nachrichtenaustausch zwischen Client und Server wird nicht verschlüsselt, da die beiden noch keine Sicherheitsbeziehung in Form von Schlüsseln zueinander haben. Daher kann eine Manipulation durch einen dritten Akteur während der Funkübertragung nicht ausgeschlossen werden. Da dies eine Sicherheitslücke im Ablauf darstellt, ist eine Alternative zur verwendeten Methode zur Erzeugung und Übermittlung des Zeitstempels zu finden. Es könnte beispielsweise ein Message Authentication Code (MAC) zur SAM Information Message hinzugefügt werden, die die Integrität des Zeitstempels sicherstellen kann. Besser wäre es vermutlich den gesamten Ablauf so zu verändern, dass keine SAM Information Message nötig ist, sondern zum Erfragen des URIs von SAM ein Verzeichnisdienst verwendet wird und der Zeitstempel auf SAM auf Basis einer mit dem Server synchronisierten Uhr generiert wird.

Die in dieser Arbeit entworfene und eingesetzte Widerrufung von Tickets hat in der dezentralen DCAF-Architektur mit der nicht immer möglichen Kommunikation zwischen eingeschränkten Servern und seinem Autorisierungsmanager das Problem, dass Widerrufungen nicht immer zugestellt und damit durchgesetzt werden können. Das Ablaufdatum im Ticket kann vorhandene Problem zumindest zeitlich auf die noch gültige Zeit eingrenzen. Hierbei ist, wie häufig in Sicherheitsentscheidungen, zwischen Sicherheit, Funktionalität und Benutzerfreundlichkeit abzuwägen beziehungsweise es ist ein für das Szenario akzeptables Verhältnis dieser zu finden. In Szenarien, in denen das Widerrufen unmittelbar durchgesetzt werden muss, ist ein, wie in DCAF festgelegter, Ansatz besser geeignet, bei dem ein Server, sobald er die Verbindung zu seinem Autorisierungsmanager verliert, alle weiteren Anfragen ablehnen und bestehende Verbindungen abbrechen muss. Dies kann allerdings eines der Hauptziele von DCAF, dass der Server autonom und dezentral die Zugriffskontrolle durchsetzen kann, untergraben.

Wenn eine Ticket-Widerrufung im entworfenen System allerdings erfolgreich vom SAM an den Server zugestellt werden konnte, so kann der Server mit Hilfe des Sliding-Windows unbegrenzt lange sicherstellen, dass das widerrufene Ticket nicht mehr verwendet werden kann. Allerdings kann das verwendete Sliding-Window auch dazu führen, dass durch das Widerrufen eines Tickets andere Tickets, die eigentlich noch gültig sind, nicht mehr verwendet werden können. Dies passiert, wenn Tickets mit einer Sequenznummer, die über dem Fenster liegt, widerrufen werden, da das Fenster in diesem Fall verschoben werden muss und alle Tickets mit Sequenznummern, die nun unter dem Fenster liegen nicht mehr zugelassen werden. Da der Server hinsichtlich seines Speichers eingeschränkt ist, kann dieser in jedem Fall nur eine begrenzte Anzahl an widerrufenen Tickets speichern und ein Informationsverlust in irgendeiner Weise, wenn die Grenze erreicht ist, ist unvermeidlich. In vielen Szenarien ist es vermutlich das geringere Übel,

wenn noch gültige Tickets nicht mehr akzeptiert werden, als wenn widerrufene Tickets plötzlich wieder verwendet werden können, weil der Server sich alte Widerrufungen nicht mehr merken kann.

6.4.4 Angriffsebenen und -szenarien

Außer dem initialen Austausch der Unauthorized Resource Message und der SAM Information Message durch Client und Server wird jede Kommunikation zwischen den Akteuren im Szenario verschlüsselt. TLS und DTLS gewährleisten die Vertraulichkeit, Integrität und Authentizität der übertragenen Daten. Daher können zu keiner Zeit im Szenario sensible Daten, wie Tickets mit dem Verifier oder Sensordaten, durch passive Angriffe mitgeschnitten werden. Die zur verschlüsselten Kommunikation verwendeten Algorithmen, wie AES und SHA256, gelten bei korrekter Anwendung als sicher gegen die meisten heute möglichen und bekannten Angriffe. HMAC-Funktionen gelten als sicher solange der Schlüssel geheim bleibt, auch wenn der andere Teil, im konkreten Fall das Ticket-Face, einem Angreifer bekannt ist.

Im Folgenden werden ausgewählte Angriffszenarien beschrieben und ein Bezug zu DCAF und dem umgesetzten Szenario hergestellt.

Kompromittierung eines Akteurs

In keinem Fall lassen sich sicherheitskritische Fehler in der entwickelten Software und in den verwendeten Bibliotheken ausschließen. Da alle Akteure in C entwickelt wurden, sind Fehler wie Buffer Overflows möglich, durch die unter Umständen sensible Speicherbereiche ausgelesen werden oder gar ganze Akteure kompromittiert werden können. Dies hätte bei den verwendeten Akteuren unterschiedliche Auswirkungen auf die Sicherheit des Gesamtsystems.

Die weitreichendsten Folgen hätte die Kompromittierung von SAM, insbesondere da dieser zu jedem Server den verwendeten Schlüssel vorhält und somit alle Server als kompromittiert zu betrachten wären. Die gesamte Sicherheit des Systems in der Sicherheitsdomäne des ROPs wäre nicht mehr gegeben. Die Kompromittierung eines eingeschränkten Servers hätte in Bezug auf das Gesamtsystem weniger weitreichende Folgen, da einzelne Server keinen aktiven Einfluss auf andere Akteure im Szenario ausüben können. Dadurch wäre lediglich das eine Gerät betroffen und müsste ausgetauscht, neu aufgesetzt und mit neuen Schlüsseln versorgt werden. Eine Kompromittierung eines Akteurs in der Sicherheitsdomäne des COPs hätte ebenfalls keine dauerhaften Auswirkungen auf die Sicherheit des Gesamtsystems. Tickets, die für die kompromittierte Domäne ausgestellt wurden, können widerrufen werden und Zugriffsregeln auf SAM können geändert werden, sodass die kompromittierten Akteure der Zugriff verwehrt werden kann, wenn die Kompromittierung erkannt wird.

Denial of Service Angriffe

Akteure des Szenarios oder Dritte könnten versuchen, den Zugang zu einzelnen Diensten auf Akteuren zu blockieren, indem diese überlastet werden. Dies ist insbeson-

dere für die eingeschränkten Geräte relevant, da diese durch ihre Eingeschränktheit bei DoS-Angriffen recht schnell an ihre Belastungsgrenze stoßen. DoS-Angriffe können auf unterschiedlichen Schichten des Protokollstapels durchgeführt werden. Auf der Transportschicht sieht DTLS durch das, in Unterabschnitt 6.3.4 beschriebenen, Cookie während des Handshakes eine Gegenmaßnahme für DoS-Angriffe vor, die sicherstellt, dass der Absender verifiziert wird bevor viel Speicher und Rechenzeit während des Handshakes benötigt wird. Allerdings kann ein eingeschränktes Gerät auch mit dieser Gegenmaßnahme noch durch DTLS-Verbindungen überlastet werden, da er generell nur wenige parallele Verbindungen zulässt.

Ein Client mit bösartigen Absichten könnte beispielsweise parallel mehrere Verbindungen zu einem Server offen halten, sodass der Server keine weiteren Verbindungen mehr annehmen kann. Nicht nur wäre der Server somit für andere Clients nicht mehr erreichbar, sondern es könnten auch Ticket-Widerrufungen nicht mehr an den Server weitergeleitet werden. Der Client könnte dies nutzen, um die Zustellung einer Widerrufung zu verzögern oder zu verhindern und somit ein eigentlich widerrufenes Ticket weiterhin nutzen. Als Gegenmaßnahme hierzu könnte der eingeschränkte Server immer genug Speicher für eine weitere DTLS-Verbindung zu seinem Autorisierungsmanager bereit halten, sodass insgesamt ein Client weniger gleichzeitig zugreifen kann. Da DoS-Angriffe, wie beschrieben, allerdings auch auf anderen Ebenen möglich sind, kann es einem Angreifer dennoch gelingen die Zustellung einer Ticket-Widerrufung zu verzögern.

Kapitel 7

Fazit und Ausblick

In der vorliegenden Arbeit wurde der Frage nachgegangen, ob DCAF als Framework für die sichere Kommunikation zwischen eingeschränkten Geräten und zur dezentralen Authentifizierung und Autorisierung in eingeschränkten Umgebungen geeignet ist. Hierfür wurden in Kapitel 4 die in der DCAF-Architektur nötigen Akteure entworfen. Insbesondere wurde für den Server Authorization Manager ein System zum Verwalten und Auswerten von Zugriffsregeln entworfen, welches für die dezentrale Autorisierung geeignet ist. Darüber hinaus wurde der Protokollablauf anhand des konkreten Nutzungsszenarios *Bananen für München* beschrieben und ausgewertet.

Es konnte gezeigt werden, dass die Kommunikation nach dem DCAF-Protokollablauf und die Durchsetzung der Zugriffskontrolle von eingeschränkten Geräten der Klasse `C1` durchgeführt werden können. Außerdem konnte gezeigt werden, dass die symmetrische Ende-zu-Ende-Verschlüsselung zwischen eingeschränkten Geräten durch die Verwendung von Tickets auch bei vielen Teilnehmern skaliert, da die eingeschränkten Geräte von der Aufgabe der Schlüsselverwaltung befreit werden.

Für den Ablauf des DCAF-Protokolls und die durchzuführende Zugriffskontrolle werden auf den eingeschränkten Geräten nur wenige Kilobyte Code benötigt und nur einige hundert Byte Arbeitsspeicher belegt. Durch die Verwendung des Betriebssystems Contiki und der Bibliotheken werden insgesamt zwar 92,13 KiB auf dem eingeschränkten Server und 81,45 KiB auf dem Client von den insgesamt 128 KiB Programmspeicher benötigt. Allerdings stehen mit 35,87 KiB und 46,55 KiB noch ausreichend Platz für die wenigen speziellen Aufgaben von Geräten des Internet of Things zur Verfügung.

Allerdings konnten auch einige Probleme aufgedeckt werden, die insbesondere auf die DTLS-Kommunikation zurückzuführen sind. Der DTLS-Handshake dauert im Verhältnis zur eigentlichen Abfrage unverhältnismäßig lang. Da eingeschränkte Server nur wenige parallele DTLS-Verbindungen verwalten können, muss bei vielen parallelen Zugriffen von Clients der DTLS-Verbindungsaufbau für jede Anfrage erneut ausgeführt werden. Hier bietet sich eine weitere Evaluierung des Einsatzes von DTLS-Session-Resumption an.

Außerdem wurde bei der Kommunikation zwischen eingeschränktem und weniger eingeschränktem Gerät ein Timing-Problem während des DTLS-Handshakes festgestellt, durch den eingeschränkte Geräte nicht mit dem Verarbeiten von Handshake-Nachrichten hinterkommen und daher der Handshake stecken bleibt. Da

dieses Problem unter Umständen auf die in der Simulation fehlenden Übertragungsverzögerungen zurückzuführen ist, bietet es sich an, die implementierten Akteure und den Protokollablauf auf echter Hardware zu testen und auszuwerten.

Darüber hinaus konnte eine Fragmentierung von Paketen während des DTLS-Handshakes beobachtet werden, die in eingeschränkten Netzen zu Problemen führen kann. Hier bietet sich die Evaluierung des Einsatzes von Kompressionsverfahren für den DTLS-Header an.

Bei der Auswertung des DCAF-Protokollablaufs konnten mit der *SAM Information Message* eine Stelle ausgemacht werden, an der die Vertraulichkeit, Integrität und Authentizität der übertragenen Daten nicht sichergestellt werden kann, weil diese nicht verschlüsselt werden. Es konnte gezeigt werden, dass insbesondere bei der Verwendung von Zeitstempeln die vom Server generiert werden ein Sicherheitsproblem auftreten kann. Daher sollten statt der *SAM Information Message* in DCAF Verzeichnisdienste verwendet werden, um die benötigten Informationen zu erhalten.

Quellenverzeichnis

Literatur

[AIM10] L. Atzori, A. Iera und G. Morabito. „The Internet of Things: A survey“. In: *Computer Networks* 54.15 (2010), S. 2787–2805 (siehe S. 9).

[Ant] S. Antipolis. *WiSMote: IPv6 platform for Wireless Sensor Networks R&D*. URL: http://www.aragosystems.com/images/stories/WiSMote/Doc/wismote_en.pdf (abgerufen am 08.06.2015) (siehe S. 42).

[Ash09] K. Ashton. *That 'Internet of Things' Thing*. URL: http://www.rfidjournal.com/articles/view?4986 (abgerufen am 08.06.2015). 2009 (siehe S. 5).

[Aus01] R. Ausanka-Crues. *Methods for Access Control: Advances and Limitations*. URL: http://www.cs.hmc.edu/~mike/public_html/courses/security/s06/projects/ryan.pdf (abgerufen am 08.06.2015). 2001 (siehe S. 16, 17).

[Bac+13] E. Baccelli u. a. *An OS for the IoT – Goals, Challenges, and Solutions*. 2013 (siehe S. 44).

[Bor15] C. Bormann. *An Authorization Information Format (AIF) for ACE*. draft-bormann-core-ace-aif-02. Internet-Draft (Work in Progress). 9. März 2015 (siehe S. 48, 55, 59).

[CO11] S. Chin und S. Older. *Access Control, Security, and Trust: A Logical Approach*. 2011 (siehe S. 18).

[DGV04] A. Dunkels, B. Gronvall und T. Voigt. „Contiki - a lightweight and flexible operating system for tiny networked sensors“. In: *Local Computer Networks* (Nov. 2004), S. 455–462 (siehe S. 43).

[DV66] J. B. Dennis und E. C. Van Horn. „Programming Semantics for Multiprogrammed Computation“. In: *Communications of the ACM* 9.3 (1966), S. 143–155 (siehe S. 19).

[Eck13] C. Eckert. *IT-Sicherheit*. 8. Ausgabe. München: Oldenbourg Verlag, 2013 (siehe S. 15–20).

[EE03] J. Elson und D. Estrin. „Time Synchronization for Wireless Sensor Networks“. In: (2003) (siehe S. 38).

[Eva11] D. Evans. *The Internet of Things - How the Next Evolution of the Internet Is Changing Everything*. URL: https://www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf (abgerufen am 08.06.2015). Apr. 2011 (siehe S. 1, 5).

[Fle09] E. Fleisch. „What is the Internet of Things?“ In: (Nov. 2009) (siehe S. 5, 6).

[Gar+13] O. Garcia-Morchon u. a. *Security Considerations in the IP-based Internet of Things*. draft-garcia-core-security-06. Internet-Draft (Work in Progress). 11. Sep. 2013 (siehe S. 36).

[GBB14] S. Gerdes, O. Bergmann und C. Bormann. „Delegated Authenticated Authorization for Constrained Environments“. URL: http://netsec.cs.uoregon.edu/npsec2014/slides/dcaf_npsec2014.pdf (abgerufen am 08.06.2015). Universität Bremen, 21. Okt. 2014 (siehe S. 26, 42).

[GBB15a] S. Gerdes, O. Bergmann und C. Bormann. *Delegated CoAP Authentication and Authorization Framework (DCAF)*. draft-gerdes-ace-dcaf-authorize-02. Internet-Draft (Work in Progress). 9. März 2015 (siehe S. 3, 20–22, 46, 51).

[GBB15b] S. Gerdes, O. Bergmann und C. Bormann. „Delegated CoAP Authentication and Authorization Framework (DCAF)“. 24. März 2015 (siehe S. 20, 21, 36).

[Ger15a] S. Gerdes. *Actors in the ACE Architecture*. Internet Draft. Internet-Draft (Work in Progress). 9. März 2015 (siehe S. 30, 33, 34, 67).

[Ger15b] S. Gerdes. *Managing the Authorization to Authorize in the Lifecycle of a Constrained Device*. draft-gerdes-ace-a2a-00. Internet-Draft (Work in Progress). März 2015 (siehe S. 63, 64).

[GPR13] S. Gusmeroli, S. Piccione und D. Rotondi. „A capability-based security approach to manage access control in the Internet of Things“. In: *Mathematical and Computer Modelling* 58.5 (2013), S. 1189–1205 (siehe S. 26).

[GW11] Z. Guoping und G. Wentao. „The Research of Access Control Based on UCON in the Internet of Things“. In: *Journal of Software* 6.4 (2011), S. 724–731 (siehe S. 16).

[Har14a] T. Hardjono. „Kerberos for Internet-of-Things“. URL: http://www.tschofenig.priv.at/tutorials/Kerberos-Tutorial.pdf (abgerufen am 08.06.2015). Feb. 2014 (siehe S. 25).

[Har14b] T. Hartke. *Practical Issues with Datagram Transport Layer Security in Constrained Environments*. draft-hartke-dice-practical-issues-01. Internet-Draft (Work in Progress). 8. Apr. 2014 (siehe S. 108).

[Her+13] J. L. Hernandez-Ramos u. a. „Distributed Capability-based Access Control for the Internet of Things“. In: *Journal of Internet Services and Information Security* 3.3 (2013), S. 1–16 (siehe S. 17, 18, 26).

[Hum+13] R. Hummen u. a. „6LoWPAN Fragmentation Attacks and Mitigation Mechanisms“. In: (2013) (siehe S. 8).

[Hum+14] R. Hummen u. a. „Delegation-based Authentication and Authorization for the IP-based Internet of Things“. In: (2014) (siehe S. 108).

[Jay+13] G. Jayavardhana u. a. „Internet of Things (IoT): A vision, architectural elements, and future directions“. In: *Future Generation Computer Systems* 29.7 (2013), S. 1645–1660 (siehe S. 6).

[Jed+13] R. Jedermann u. a. „Sea transport of bananas in containers – Parameter identification for a temperature model“. In: *Journal of Food Engineering* 115.3 (2013), S. 330–338 (siehe S. 28, 62).

[Ker01] A. D. Keromytis. „Strongman: A Scalable Solution To Trust Management In Networks“. Diss. University of Pennsylvania, 2001 (siehe S. 16).

[Kiz13] J. M. Kizza. *Guide to Computer Network Security*. 2. Ausgabe. Springer-Verlag, 2013 (siehe S. 107).

[Kop11] H. Kopetz. *Real-Time Systems. Design Principles for Distributed Embedded Applications*. 2. Ausgabe. New York: Springer, 2011 (siehe S. 6).

[Kot+12] T. Kothmayr u. a. „A DTLS Based End-To-End Security Architecture for the Internet of Things with Two-Way Authentication“. In: (2012) (siehe S. 13).

[KS] B. Kupfer und U. Schieder (V.i.S.d.P). *Waren-Informationen: Bananen*. URL: http://www.tis-gdv.de/tis/ware/obst/banane/banane.htm (abgerufen am 08.06.2015) (siehe S. 28).

[KS07] A. D. Keromytis und J. M. Smith. „Requirements for Scalable Access Control and Security Management Architectures“. In: *ACM Transactions on Internet Technology* 7.2 (Mai 2007), S. 1–22 (siehe S. 19, 62).

[LOP04] J. Lopez, R. Oppliger und G. Pernul. „Authentication and authorization infrastructures (AAIs): a comparative survey“. In: *Computers & Security* 23.7 (2004), S. 578–590 (siehe S. 16–18).

[May09] C. P. Mayer. „Security and Privacy Challenges in the Internet of Things“. In: *Electronic Communications of the EASST* 17 (2009), S. 1–12 (siehe S. 6).

[MJ00] P. McDaniel und S. Jamin. *Windowed Certificate Revocation*. URL: http://www.patrickmcdaniel.org/pubs/info00.pdf (abgerufen am 08.06.2015). März 2000 (siehe S. 19).

[PM04] R. Payne und B. Macdonald. „Ambient technology — now you see it, now you don't“. In: *BT Technology Journal* 22.3 (2004), S. 119–129 (siehe S. 5).

[RFC1055] J. Romkey. *A Nonstandard for transmission of IP datagrams over serial lines: SLIP*. RFC 1055. Juni 1988 (siehe S. 77).

[RFC1180] T. Socolofsky und C. Kale. *A TCP/IP Tutorial*. RFC 1180. Jan. 1991 (siehe S. 70).

[RFC2460] S. Deering und R. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460. Dez. 1998 (siehe S. 8).

[RFC3986] T. Berners-Lee, R. Fielding und L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. RFC3986. Jan. 2005 (siehe S. 14).

[RFC4279] P. Eronen und H. Tschofenig. *Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)*. RFC4279. Dez. 2005 (siehe S. 13, 100).

[RFC4492] S. Blake-Wilson u. a. *Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)*. RFC 4492. Mai 2006 (siehe S. 12, 13).

[RFC4944] G. Montenegro u. a. *Transmission of IPv6 Packets over IEEE 802.15.4 Networks*. RFC 4944. Sep. 2007 (siehe S. 8).

[RFC5077] J. Salowey u. a. *Transport Layer Security (TLS) Session Resumption without Server-Side State*. RFC5077. Jan. 2008 (siehe S. 107).

[RFC5246] T. Dierks und E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC5246. Jan. 2008 (siehe S. 9, 12, 13, 108).

[RFC5487] M. Badra. *Pre-Shared Key Cipher Suites for TLS with SHA-256/384 and AES Galois Counter Mode*. RFC 5487. März 2009 (siehe S. 13).

[RFC5785] M. Nottingham und E. Hammer-Lahav. *Defining Well-Known Uniform Resource Identifiers (URIs)*. Techn. Ber. Apr. 2010 (siehe S. 14).

[RFC6282] J. Hui und P. Thubert. *Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks*. RFC 6282. Sep. 2011 (siehe S. 8).

[RFC6347] E. Rescorla und N. Modadugu. *Datagram Transport Layer Security Version 1.2*. Standards Track. Jan. 2012 (siehe S. 10–13).

[RFC6606] E. Kim u. a. *Problem Statement and Requirements for IPv6 over Low-Power Wireless Personal Area Network (6LoWPAN) Routing*. RFC 6606. Mai 2012 (siehe S. 8).

[RFC6655] D. McGrew und D. Bailey. *AES-CCM Cipher Suites for Transport Layer Security (TLS)*. RFC 6655. Juli 2012 (siehe S. 13, 78).

[RFC6690] Z. Shelby. *Constrained RESTful Environments (CoRE) Link Format*. Techn. Ber. Aug. 2012 (siehe S. 14).

[RFC6749] D. Hardt. *The OAuth 2.0 Authorization Framework*. RFC6749. Okt. 2012 (siehe S. 25).

[RFC7049] C. Bormann und P. Hoffman. *Concise Binary Object Representation (CBOR)*. RFC7049. Okt. 2013 (siehe S. 23, 47).

[RFC7159] Z. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC7159. März 2014 (siehe S. 23, 81).

[RFC7228] C. Bormann, M. Ersue und A. Keranen. *Terminology for Constrained-Node Networks*. RFC7228. Mai 2014 (siehe S. 6, 7, 42).

[RFC7231] R. Fielding und J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC7231. Juni 2014 (siehe S. 66).

[RFC7252] Z. Shelby, K. Hartke und C. Bormann. *The Constrained Application Protocol (CoAP)*. RFC7252. Juni 2014 (siehe S. 13–15, 46).

[RFC7465] A. Popov. *Prohibiting RC4 Cipher Suites*. RFC 7465. Feb. 2015 (siehe S. 13).

[RNL11] R. Roman, P. Najera und J. Lopez. „Securing the Internet of Things“. In: *Computer* 44.9 (2011), S. 51–58 (siehe S. 6).

[RP12] D. Rotondi und S. Piccione. „Managing Access Control for Things: a Capability Based Approach“. In: (2012), S. 263–268 (siehe S. 17–19).

[Seh13] A. Sehgal. „Using the Contiki Cooja Simulator“. In: (2013) (siehe S. 45).

[Sei+15] L. Seitz u. a. *ACE use cases*. draft-ietf-ace-usecases-03. Internet-Draft (Work in Progress). 9. März 2015 (siehe S. 28, 30).

[SS15] L. Seitz und G. Selander. *Problem Description for Authorization in Constrained Environments*. Internet Draft. Internet-Draft (Work in Progress). 9. März 2015 (siehe S. 21).

[TF15] H. Tschofenig und T. Fossati. *A TLS/DTLS Profile for the Internet of Things*. draft-ietf-dice-profile-12. Internet-Draft (Work in Progress). Mai 2015 (siehe S. 108).

[Thi03] W. Thierse. „Traditionswahrung und Modernisierung - Sozialdemokratie in der Entscheidung“. URL: http://library.fes.de/fulltext/historiker/01705toc.htm (abgerufen am 08.06.2015). 2003 (siehe S. 1).

[Tsc15] S. Tschofenig. *The OAuth 2.0 Internet of Things (IoT) Client Credentials Grant*. draft-tschofenig-ace-oauth-iot-01. Internet-Draft (Work in Progress). März 2015 (siehe S. 25).

[VBS15] C. Vigano, H. Birkholz und R. Sun. *CBOR data definition language: a notational convention to express CBOR data structures.* draft-greevenbosch-appsawg-cbor-cddl-05. 9. März 2015 (siehe S. 47).

[Wan+05] C. Wang u. a. „Issues of Transport Control Protocols for Wireless Sensor Networks“. In: (Mai 2005) (siehe S. 9).

[Wei91] M. Weiser. *The Computer for the 21st Century*. URL: http://www.ubiq.com/hypertext/weiser/SciAmDraft3.html (abgerufen am 08.06.2015). Sep. 1991 (siehe S. 5).

Anhang A

Änderungen an den verwendeten Bibliotheken

tinydtls (Contiki)

Da im entwickelten Code festgelegt wurde, dass ein Ticket-Face maximal 128 Byte haben darf, musste die Maximallänge der Client-Identity in tinydtls agehoben werden:

```
diff --git a/crypto.h b/crypto.h
index 7fc6b7c..c914fed 100644
--- a/crypto.h
+++ b/crypto.h
@@ -86,7 +86,7 @@ typedef struct {

 /* This is the maximal supported length of the psk client identity and psk
  * server identity hint */
-#define DTLS_PSK_MAX_CLIENT_IDENTITY_LEN   32
+#define DTLS_PSK_MAX_CLIENT_IDENTITY_LEN   128
```

Um im eingeschränkten Server DTLS-Verbindungen aktiv abzubrechen (zum Beispiel wenn ein Ticket widerrufen wurde), wurde die Funktion `dtls_destroy_peer` exportiert, sodass sie im Anwendungsprogramm genutzt werden kann:

```
diff --git a/dtls.c b/dtls.c
index 0831ddb..0750d2f 100644
--- a/dtls.c
+++ b/dtls.c
@@ -1494,7 +1494,7 @@ dtls_close(dtls_context_t *ctx, const session_t *remote) {
   return res;
 }

-static void dtls_destroy_peer(dtls_context_t *ctx, dtls_peer_t *peer, int unlink
-    )
+void dtls_destroy_peer(dtls_context_t *ctx, dtls_peer_t *peer, int unlink)
 {

--- a/dtls.h
+++ b/dtls.h
@@ -430,6 +430,7 @@ int dtls_handle_message(dtls_context_t *ctx, session_t *
    session,
```

```
 dtls_peer_t *dtls_get_peer(const dtls_context_t *context,
                const session_t *session);

+void dtls_destroy_peer(dtls_context_t *ctx, dtls_peer_t *peer, int unlink);
```

Der Aufruf der Contiki-Funktion `clock_init` musste aus tinydtls entfernt werden, da die Funktion bereits in libcoap und im Anwendungsprogramm aufgerufen wird und ein mehrfaches Aufrufen zu Problemen mit dem Clock-Modul von Contiki geführt hat.

```
diff --git a/dtls_time.c b/dtls_time.c
index 88c292a..32c69e5 100644
--- a/dtls_time.c
+++ b/dtls_time.c
@@ -37,7 +37,6 @@ clock_time_t dtls_clock_offset;

 void
 dtls_clock_init(void) {
-  clock_init();
   dtls_clock_offset = clock_time();
 }
```

tinydtls (x86)

Auch für die nicht eingeschränkten Geräte musste die maximale Größe der Client-Identity angepasst werden:

```
diff --git a/crypto.h b/crypto.h
index 7fc6b7c..c914fed 100644
--- a/crypto.h
+++ b/crypto.h
@@ -86,7 +86,7 @@ typedef struct {

 /* This is the maximal supported length of the psk client identity and psk
  * server identity hint */
-#define DTLS_PSK_MAX_CLIENT_IDENTITY_LEN   32
+#define DTLS_PSK_MAX_CLIENT_IDENTITY_LEN   128
```

Auf SAM wird tinydtls parallel zu OpenSSL verwendet. Da beide Bibliotheken Funktionen zur Berechnung von SHA-Hashes bereitstellen und diese in den Bibliotheken gleich benannt sind, wurden die SHA-Funktionen in tinydtls mit dem Prefix `DTLS_` versehen. Da der zugehörige Patch zu lang ist, um hier im Ganzen abgedruckt zu werden, sei auf die entsprechenden Commits im Repository verwiesen: https://gitlab.informatik.uni-bremen.de/tha/tinydtls/commit/b52df58fd72e2cb2d550cc8fa8a9bcf3d9a17caf und https://gitlab.informatik.uni-bremen.de/tha/tinydtls/commit/bc14b1a7a82e06c261067e1b82b6f1c04e80433c

Das in der Arbeit beschriebene Timing-Problem bei der Kommunikation zwischen eingeschränkten und weniger eingeschränkten Geräten, wurde durch das Einfügen künstlicher Verzögerungen zwischen den Handshake-Nachricht umgangen:

```
diff --git a/dtls.c b/dtls.c
index 0831ddb..5ec5c61 100644
--- a/dtls.c
+++ b/dtls.c
```

```
@@ -27,6 +27,8 @@
 #include "tinydtls.h"
 #include "dtls_config.h"
 #include "dtls_time.h"
+#include <time.h>
+#include <unistd.h>

 #include <stdio.h>
 #include <stdlib.h>
@@ -51,6 +53,10 @@
 #  include "sha2/sha2.h"
 #endif

+// Sleep 150 ms between some handshake messages to avoid errors
+// while handshaking with constrained node
+#define INTENTIONAL_DTLS_HANDSHAKE_DELAY usleep(150*1000);
+
 #define dtls_set_version(H,V) dtls_int_to_uint16((H)->version, (V))
 #define dtls_set_content_type(H,V) ((H)->content_type = (V) & 0xff)
 #define dtls_set_length(H,V)  ((H)->length = (V))
@@ -2112,6 +2118,8 @@ dtls_send_server_hello_msgs(dtls_context_t *ctx,
    dtls_peer_t *peer)
      * and check psk only. */
     CALL(ctx, get_psk_hint, &peer->session, &psk);

+    INTENTIONAL_DTLS_HANDSHAKE_DELAY
+
     if (psk) {
       res = dtls_send_server_key_exchange_psk(ctx, peer, psk);

@@ -2123,6 +2131,8 @@ dtls_send_server_hello_msgs(dtls_context_t *ctx,
    dtls_peer_t *peer)
   }
 #endif /* DTLS_PSK */

+   INTENTIONAL_DTLS_HANDSHAKE_DELAY
+
   res = dtls_send_server_hello_done(ctx, peer);

   if (res < 0) {
@@ -2136,6 +2146,8 @@ static inline int
 dtls_send_ccs(dtls_context_t *ctx, dtls_peer_t *peer) {
   uint8 buf[1] = {1};

+  INTENTIONAL_DTLS_HANDSHAKE_DELAY
+
   return dtls_send(ctx, peer, DTLS_CT_CHANGE_CIPHER_SPEC, buf, 1);
 }

@@ -2834,6 +2846,8 @@ check_server_hellodone(dtls_context_t *ctx,
     return res;
   }

+  INTENTIONAL_DTLS_HANDSHAKE_DELAY
+
   res = dtls_send_ccs(ctx, peer);
   if (res < 0) {
     dtls_debug("cannot send CCS message\n");
@@ -2844,7 +2858,13 @@ check_server_hellodone(dtls_context_t *ctx,
```

```
    dtls_security_params_switch(peer);

    /* Client Finished */
-   return dtls_send_finished(ctx, peer, PRF_LABEL(client), PRF_LABEL_SIZE(client)
     );
+   INTENTIONAL_DTLS_HANDSHAKE_DELAY
+
+   int r = dtls_send_finished(ctx, peer, PRF_LABEL(client), PRF_LABEL_SIZE(client
     ));
+
+   INTENTIONAL_DTLS_HANDSHAKE_DELAY
+
+   return r;
  }

  static int
@@ -3126,6 +3146,8 @@ handle_handshake_msg(dtls_context_t *ctx, dtls_peer_t *peer
     , session_t *session,

        dtls_security_params_switch(peer);

+       INTENTIONAL_DTLS_HANDSHAKE_DELAY
+
        err = dtls_send_finished(ctx, peer, PRF_LABEL(server), PRF_LABEL_SIZE(
    server));
        if (err < 0) {
          dtls_warn("sending server Finished failed\n");
```

libcoap (Contiki)

Um im Anwendungsprogramm Speicher für CoAP-Pakete anzufordern, wurde die von libcoap verwendete Funktion `coap_malloc_packet` exportiert:

```
diff --git a/coap_io.c b/coap_io.c
index 3206913..0ef2fad 100644
--- a/coap_io.c
+++ b/coap_io.c
@@ -316,7 +316,7 @@ coap_free_packet(coap_packet_t *packet) {
 }
 #endif /* WITH_POSIX */
 #ifdef WITH_CONTIKI
-static inline coap_packet_t *
+inline coap_packet_t *
 coap_malloc_packet(void) {
   return (coap_packet_t *)coap_malloc_type(COAP_PACKET, 0);
 }
```

Unter der verwendeten Contiki-Umgebung steht die Funktion `strerror` nicht zur Verfügung. Daher wurden deren Verwendung an zwei Stellen entfernt.

```
diff --git a/coap_io.c b/coap_io.c
index 0ef2fad..4b5c40c 100644
--- a/coap_io.c
+++ b/coap_io.c
@@ -377,7 +377,7 @@ coap_network_read(coap_endpoint_t *ep, coap_packet_t **packet
     ) {
```

```
    len = recvmsg(ep->handle.fd, &mhdr, 0);

    if (len < 0) {
-     coap_log(LOG_WARNING, "coap_network_read: %s\n", strerror(errno));
+     coap_log(LOG_WARNING, "coap_network_read:\n");
      coap_free_packet(*packet);
      *packet = NULL;
    } else {
diff --git a/net.c b/net.c
index fff1238..a50a0a1 100644
--- a/net.c
+++ b/net.c
@@ -559,7 +559,7 @@ coap_send_impl(coap_context_t *context,
    if (bytes_written >= 0) {
      coap_transaction_id(dst, pdu, &id);
    } else {
-     coap_log(LOG_CRIT, "coap_send_impl: %s\n", strerror(errno));
+     coap_log(LOG_CRIT, "coap_send_impl:\n");
    }

    return id;
```

Auch aus libcoap musste der Aufruf der Funktion `clock_init` entfernt werden. Die Funktion wird einmalig im Anwendungsprogramm aufgerufen.

```
diff --git a/coap_time.h b/coap_time.h
index c0c5321..7cb00ed 100644
--- a/coap_time.h
+++ b/coap_time.h
@@ -74,7 +74,6 @@ extern clock_time_t clock_offset;

 static inline void
 contiki_clock_init_impl(void) {
-  clock_init();
    clock_offset = clock_time();
 }
```

Der `msp430-gcc` war mit einer while-Schleife ohne geschweifte Klammern nicht einverstanden.

```
diff --git a/resource.c b/resource.c
index 1ca680c..298a0c8 100644
--- a/resource.c
+++ b/resource.c
@@ -509,8 +509,9 @@ coap_hash_request_uri(const coap_pdu_t *request, coap_key_t
     key) {
    coap_option_setb(filter, COAP_OPTION_URI_PATH);

    coap_option_iterator_init((coap_pdu_t *)request, &opt_iter, filter);
-  while ((option = coap_option_next(&opt_iter)))
+  while ((option = coap_option_next(&opt_iter))) {
      coap_hash(COAP_OPT_VALUE(option), COAP_OPT_LENGTH(option), key);
+  }
 }
```

Mongoose

Um auf SAM im Anwendungsprogramm den Fingerprint des vom Client verwendeten Zertifikats zu ermitteln, wurden eine Datenstruktur und zwei Funktionen so geändert, dass der OpenSSL-Kontext an den Request-Handler im Anwendungsprogramm weitergereicht wird:

```
diff --git a/sam/src/lib/mongoose/mongoose.c b/sam/src/lib/mongoose/mongoose.c
index aa25bcc..96749c5 100644
--- a/sam/src/lib/mongoose/mongoose.c
+++ b/sam/src/lib/mongoose/mongoose.c
 static int ns_use_cert(SSL_CTX *ctx, const char *pem_file)
@@ -2046,6 +2044,12 @@ static const char *status_code_to_str(int status_code)

 static int call_user(struct connection *conn, enum mg_event ev)
 {
+    conn->mg_conn.ssl = conn->ns_conn->ssl;
+    conn->mg_conn.ssl_ctx = conn->ns_conn->ssl_ctx;
     return conn != NULL && conn->server != NULL &&
            conn->server->event_handler != NULL
            ? conn->server->event_handler(&conn->mg_conn, ev)

@@ -5849,9 +5853,14 @@ static void iter2(struct ns_connection *nc, int ev, void *
    param)
     int n;
     (void)ev;

     // DBG(("%p [%s]", conn, msg));
     if (sscanf(msg, "\%p \%n", &func, &n) && func != NULL && conn != NULL) {
         conn->mg_conn.callback_param = (void *)(msg + n);
+        conn->mg_conn.ssl = nc->ssl;
+        conn->mg_conn.ssl_ctx = nc->ssl_ctx;
+
         func(&conn->mg_conn, MG_POLL);
     }
 }
diff --git a/sam/src/lib/mongoose/mongoose.h b/sam/src/lib/mongoose/mongoose.h
index 397fbd9..ee27c60 100644
--- a/sam/src/lib/mongoose/mongoose.h
+++ b/sam/src/lib/mongoose/mongoose.h
@@ -34,7 +36,8 @@ struct mg_connection {
   const char *query_string;    // URL part after '?', not including '?', or NULL
+  void *ssl;
+  void *ssl_ctx;
   char remote_ip[48];          // Max IPv6 string length is 45 characters
   char local_ip[48];           // Local IP address
   unsigned short remote_port;  // Client's port
```

Mongoose verwendete einen falschen regulären Ausdruck, um die IPv6-Adresse von Clients auszulesen. Dieser wurde korrigiert:

```
@@ -5052,24 +5388,25 @@ ns_sock_to_str(nc->sock, buf, sizeof(buf), is_rem ? 7 :
    3);
-  sscanf(buf, "%47[^:]:%hu",
-         is_rem ? c->remote_ip : c->local_ip,
+  sscanf(buf, "%47[^\]\[]:%hu", is_rem ? c->remote_ip : c->local_ip,
          is_rem ? &c->remote_port : &c->local_port);
 }
```

Anhang B

Messergebnisse

```
Speicherverbrauch auf den eingeschränkten Geräten
=================================================================

1. Client (dcaf-client.wismote) inkl.
   - Contiki
     - IPv6
     - RPL
   - tinydtls
   - libcoap
   - cn-cbor
   - DCAF

1.1 Insgesamt:
~/t/g/e/a/c\$ msp430-size dcaf-client.wismote
   text    data     bss     dec     hex filename
  83402     484   11580   95466   174ea dcaf-client.wismote

1.2 Ohne DCAF (und damit auch ohne cn-cbor) aber mit den
    anderen Komponenten
~/t/g/e/a/c\$ msp430-size dcaf-client.wismote
   text    data     bss     dec     hex filename
  77687     480   11196   89363   15d13 dcaf-client.wismote

1.3 Ohne DCAF+cn-cbor+libcoap aber mit tinydtls+contiki
~/t/g/e/a/c\$ msp430-size dcaf-client.wismote
   text    data     bss     dec     hex filename
  67274     330    9662   77266   12dd2 dcaf-client.wismote

1.4 Ohne DCAF+cn-cbor+libcoap+tinydtls (nur noch Contiki)
~/t/g/e/a/c\$ msp430-size dcaf-client.wismote
   text    data     bss     dec     hex filename
  36574     250    7098   43922    ab92 dcaf-client.wismote

 -----------------------------------------------------------------

2. Server (Gleiche Komponenten wie oben)

2.1 Insgesamt:
~/t/g/e/a/rs\$ msp430-size dcaf-server.wismote
   text    data     bss     dec     hex filename
  94343     504   11618  106465   19fe1 dcaf-server.wismote
```

```
2.2 Ohne DCAF (und damit auch ohne cn-cbor) aber mit den
    anderen Komponenten
~/t/g/e/a/rs\$ msp430-size dcaf-server.wismote
   text    data     bss     dec     hex filename
  89167     480   11156  100803   189c3 dcaf-server.wismote

2.3 Ohne DCAF+cn-cbor+libcoap aber mit tinydtls+contiki
~/t/g/e/a/rs\$ msp430-size dcaf-server.wismote
   text    data     bss     dec     hex filename
  67090     330    9618   77038   12cee dcaf-server.wismote

1.4 Ohne DCAF+cn-cbor+libcoap+tinydtls (nur noch Contiki)
~/t/g/e/a/rs\$ msp430-size dcaf-server.wismote
   text    data     bss     dec     hex filename
  36768     250    7098   44116    ac54 dcaf-server.wismote

 ---------------------------------------------------------------------

 3. Einzelne Objektdateien

~/t/g/e/a/r/obj_wismote\$ msp430-size hmac.o
   text    data     bss     dec     hex filename
    276       8     507     791     317 hmac.o

~/t/g/e/a/r/obj_wismote\$ msp430-size sha2.
   text    data     bss     dec     hex filename
   3127       0       0    3127     c37 sha2.o

~/t/g/e/a/r/obj_wismote\$ msp430-size crypto.o
   text    data     bss     dec     hex filename
   2168      16     631    2815     aff crypto.o

~/t/g/e/a/c/obj_wismote\$ msp430-size cn-*.o
   text    data     bss     dec     hex filename
   1108       0       0    1108     454 cn-cbor.o
    672       0       0     672     2a0 cn-create.o
   1209       0       0    1209     4b9 cn-encoder.o
    307       0       0     307     133 cn-get.o

 ---------------------------------------------------------------------

4. Umrechnung auf die einzelnen Komponenten

4.1. Programmspeicher
                           Client          Server
Contiki + ip etc.          36574           36768
dtls                       30700           30322
coap                       10413           22077
dcaf inkl revo + cn-cbor    5715            5176

 ---------------------------------------------------------------------

4.2 Statischer Rambedarf in Byte (data/bss)
                           Client          Server
Contiki + ip etc.          250 / 7098      250 / 7098
dtls                        80 / 2564       80 / 2520
coap                       150 / 1534      150 / 1538
dcaf inkl revoc. + cn-cbor   4 /  384       24 /  462 (cncbor hat 0)
```

Anhang C

Verwendung der entwickelten Software

Im Folgenden wird die Verwendung der entwickelten Akteure beschrieben. Da die Einrichtung aller Akteure und der Simulationsumgebung ein etwas umfangreicherer Prozess ist, befindet sich auf der beiliegenden DVD eine virtuelle Maschine. Diese enthält bereits die Simulationsumgebung, die verwendeten Bibliotheken und die Software für die vier Akteure. Im Folgenden wird daher der Ablauf des Szenarios anhand der virtuellen Maschine erklärt. Eine Anleitung zum Erstellen der virtuellen Maschine beziehungsweise zum Einrichten des Szenarios außerhalb einer virtuellen Maschine, finden sich am Schluss von Anhang C.

Verwendung der virtuellen Maschine

Die virtuelle Maschine basiert auf InstantContiki[1], wurde allerdings auf die aktuelle Ubuntu LTS-Version 14.04 aktualisiert. Die VM wurde mit VirtualBox[2] und im VMWare-Player[3] getestet. Die VM liegt auf der DVD komprimiert vor und benötigt entpackt etwa 9 GiB Festplattenspeicher. Der VM müssen mindestens 1024 MiB Arbeitsspeicher zugewiesen werden. Außerdem sollte das Fenster der VM mindestens eine Auflösung von 1024x768 haben. Das Netzwerk der VM sollte auf NAT gestellt werden, ein Internetzugang in der VM ist aber nicht erforderlich. Um eine bessere Integration in das Host-System zu erreichen, sollten in der VM die "Gasterweiterungen" installiert werden. Das Passwort für den Standardbenutzer `user` lautet `dcafdcaf`.

In der VM befinden sich auf dem Desktop Icons für alle Programme, die für den Ablauf und die Auswertung notwendig sind. Sowohl die Contiki-Umgebung als auch die eingeschränkten Akteure liegen im Home-Verzeichnis des Standardnutzers. Die folgende Tabelle zeigt die Verzeichnisse, die konkret verwendet werden.

Außerdem wurden in der virtuellen Maschine bereits Zertifikate für CAM, SAM und ROP erzeugt und durch die CA von SAM signiert. Die Zertifikate befinden sich im

[1] InstantContiki: http://www.contiki-os.org/start.html (abgerufen am 08.06.2015)
[2] VirtualBox: https://www.virtualbox.org/ (abgerufen am 08.06.2015)
[3] VMWare-Player: https://www.vmware.com/de/products/player (abgerufen am 08.06.2015)

Komponente	*Verzeichnis*
Cooja-Contiki	`/home/user/contiki`
DCAF-Contiki	`/home/user/dcafcontiki`
Client	`/home/user/dcafcontiki/examples/dcaf/client`
Server	`/home/user/dcafcontiki/examples/dcaf/server`
CAM	`/home/user/dcafcamsam/cam`
SAM	`/home/user/dcafcamsam/sam`
Border-Router	`/home/user/dcafcontiki/examples/ipv6/rpl-border-router`
Webinterface	`/home/user/dcafcamsam/sam/src/http/html`

Tabelle C.1: Verzeichnisstruktur in der virtuellen Maschine

Verzeichnis `/home/user/dcafcamsam/certs`. Die Erzeugung der Zertifikat wird im nächsten Abschnitt von Anhang C beschrieben. Das Zertifikat von ROP wurde in der VM bereits im Browser Firefox als Client-Zertifikat hinterlegt. Der Browser schlägt die Verwendung des Zertifikats beim Aufruf des Webinterfaces vor.

Ablauf des Szenarios

Auf dem Desktop der VM befinden sich zwei Szenario-Dateien, die in Cooja geladen werden können:

1. `cooja_dcaf_1_commissioning.csc` enthält:
   1.1 Gerät (1) Border-Router
   1.2 Gerät (2) DCAF-Server
2. `cooja_dcaf_2_procedure.csc` enthält:
   2.1 Gerät (1) Border-Router
   2.2 Gerät (2) DCAF-Server
   2.3 Gerät (3) DCAF-Client

Die erste Datei wird für den Handover des Servers verwendet. Die zweite Datei enthält alle für den DCAF-Ablauf nötigen eingeschränkten Akteure. Die zweite Datei könnte auch zum Handover verwendet werden, allerdings versucht der Client in dieser permanent eine Verbindung zum Server aufzubauen, weshalb den Ausgaben des Handovers nicht gefolgt werden könnte. Die zweite Datei wird sowohl für die Kommunikation zwischen Client und Server als auch für das Widerrufen von Ticket benutzt.

Hinweis: Der Border-Router stellt für die Kommunikation von Cooja mit der Außenwelt das `tun`-Interface zur Verfügung. Jedes Mal, wenn die Cooja-Simulation neu gestartet wird (über den Reload-Button), wird das `tun`-Interface wieder entfernt. Daher muss bei jedem Neuladen der Simulation der Border-Router neugestartet werden. Da auch die nicht eingeschränkten Geräte die vom `tun`-Interface bereitgestellte IP `aaaa::1` benutzen, müssen auch CAM und SAM bei einem Neustart der Simulation neu gestartet werden.

CAM und SAM werden über das Terminal gestartet. Die ausführbaren Dateien befinden sich im jeweiligen `build/` Verzeichnis unterhalb des Verzeichnis' des Akteurs. Im Webinterface von SAM wird auf den einzelnen Unterseiten eine Hilfe für das Anlegen der, für das Szenario nötigen, Ressourcen auf SAM angeboten. Mit dieser ist es möglich, die für das Szenario nötigen Daten in die Formulare des Webinterfaces mit einem Klick hinzuzufügen.

Im Folgenden werden die, für den Ablauf des Nutzungsszenario nötigen, Schritte beschrieben. Zunächst werden die Akteure eingerichtet, anschließend findet das Commissioning des Server statt. Im Anschluss wird dann der DCAF-Protokollablauf und das Widerrufen von Tickets beschrieben.

1. Akteure starten
   - 1.1 Cooja starten über Desktop-Icon und Szenario `cooja_dcaf_1_commissioning.csc` öffnen (noch nicht auf `Start` klicken).
   - 1.2 Border-Router Starten, um IPv6-Netz mit Prefix `aaaa::/64` aufzumachen.
     - 1.2.1 Mit Terminal ins Verzeichnis `/home/user/dcafcontiki/examples/ipv6/rpl-border-router` wechseln.
     - 1.2.2 Starten des Border-Routers durch `make connect-router-cooja` (sudo PW: `dcafdcaf`).
   - 1.3 CAM starten: Mit Terminal ins Verzeichnis `/home/user/dcafcamsam/cam/build` und CAM mit `./cam` starten.
   - 1.4 SAM starten: Mit Terminal ins Verzeichnis `/home/user/dcafcamsam/sam/build` und SAM mit `./sam` starten.
   - 1.5 Browser (Firefox) öffnen
     - 1.5.1 Webinterface unter Adresse `https://[aaaa::1]:8080/` aufrufen.
     - 1.5.2 Gegebenenfalls TLS-Client-Zertifikat auswählen, falls Browser Dialog hierzu öffnet und es nicht selbst auswählt. Das Client-Zertifikat wurde bereits im Browser hinzugefügt und wird von SAM akzeptiert, weil es von SAMs CA ausgestellt wurde und weil der Fingerprint für ROP in der Config-Datei steht.
2. Handover/Comissioning des Servers
   - 2.1 Starten des, in Cooja geladenen Szenarios `cooja_dcaf_1_commissioning.csc` durch einen Klick auf den `Start-Button`. Speed Limit auf 100 %, maximal 200 %, stellen, sonst kann man den Ausgaben nicht mehr folgen.
   - 2.2 Browser mit Webinterface wieder öffnen und dort unter "Server" auf den Button `Commissioning` klicken. Es erscheint ein Formular zum Eingeben der Daten des in die Sicherheitsdomäne einzuführenden Servers.
   - 2.3 Die Daten können entweder manuell eingegeben werden oder für das Nutzungsszenario automatisch hinzugefügt werden. Dafür ist rechts oben in der Ecke ein kleiner Button, der ein Fenster öffnet. In diesem befinden sich auf den verschiedenen Seiten des Webinterfaces die für das Szenario nötigen Aufgaben. So können mit einem Klick alle für das Szenario notwendigen Da-

ten hinzugefügt werden. Auf der Commissioning-Seite kann das Formular komplett über den Button gefüllt werden

2.4 Nachdem das Formular ausgefüllt wurde, kann der Ablauf, mit einem Klick im Webinterface auf `Start Commissioning`, gestartet werden.

2.5 SAM versucht nun den Server zu erreichen. Dies kann in Cooja und im Terminal von SAM beobachtet werden.

2.6 Wenn das Commissioning erfolgreich war, zeigt das Webinterface dies an und leitet den Nutzer automatisch zur Seite weiter, auf der der neue Server als Objekt der Autorisierung hinzugefügt werden kann.

2.7 Hier kann ebenfalls über das Fenster rechts oben das Formular vollständig automatisch für das Szenario ausgefüllt werden.

2.8 Der Server wurde erfolgreich hinzugefügt. Weiter geht es mit der 2. Cooja-Datei, die auch den eingeschränkten Client als Akteur enthält und somit den DCAF-Protokollablauf ermöglicht.

3. DCAF-Protokollablauf

3.1 Cooja-Szenario `cooja_dcaf_2_procedure.csc` öffnen.

3.2 Border-Router, CAM und SAM neustarten.

3.3 Webinterface öffnen. Der Server wurde im vorherigen Schritt hinzugefügt, daher jetzt nur noch CAM und die Zugriffsregel.

3.4 In Webinterface auf *CAM -> Add CAM*. Dort entweder manuell Daten angeben oder wieder über das Dropdown-Menü rechts oben, welches vorgefertigte Daten für das Szenario enthält. Abschließend auf `Add CAM` klicken.

3.5 In Webinterface auf *Access Rules -> Add Rule*. Dort ebenfalls entweder manuell eine Regel anlegen oder über das Menü oben. Für den ersten Schritt des Szenarios wird eine implizite Zugriffsregel benötigt, die den vollen Zugriff auf den Server gestattet.

3.6 Das Cooja-Szenario kann nun gestartet werden. Die Geschwindigkeit wieder maximal auf 200 % einstellen beziehungsweise eingestellt lassen. Nach 30 Sekunden (Cooja-Zeit, bei 200 % Geschwindigkeit 15sek etc.) beginnt der Client mit der Unauthorized Resource Request Message. Der weitere Ablauf passiert automatisch. Sobald ein Ticket ausgestellt wurde, wird dies im Webinterface unter `Tickets` angezeigt. Der Client sendet im Folgenden alle fünf Sekunden eine Anfrage zum Server. Die Ausgabe kann in Cooja im Fenster “Mote Output” angesehen werden.

4. Widerrufen eines Tickets

4.1 Das Ticket kann im Webinterface mit einem Klick auf `Revoke` auf der Ticket-Seite widerrufen werden. Es wird aus den Tickets gelöscht und zu den `Revocations` hinzugefügt. Dort kann der Fortschritt der Übermittlung der Revocation an den Server beobachtet werden.

5. Ändern einer Zugriffsregel

5.1 Dem Nutzungsszenario entsprechend muss die Zugriffsregel in eine explizite Regel geändert werden. Hierfür unter `Access Rules` das Stift-Icon der Regel anklicken und explizit die eine Ressource des Servers auswählen und als Zugriffsmethode nur `GET` auswählen. Abschließend die Regel speichern.

5.2 Damit der Client sich ein neues Ticket ausstellen lässt, muss das Cooja-Szenario neu gestartet werden (Reload-Button). Auch der Border-Router, CAM und SAM müssen daher neu gestartet werden. Das Szenario kann anschließend direkt wieder gestartet werden. Der Client fängt nach 30 Sekunden wiederum mit dem Protokollablauf an und bekommt diesmal von SAM ein explizites Ticket ausgestellt.

Zertifikate

Um Zertifikate für die Authentifizierung zwischen CAM und SAM und zwischen SAM und ROP zu erstellen, wurde `openssl` verwendet. Folgende Schritte waren notwendig:

Selbst-signierte ClientAuthority für ROP/SAM erstellen:

```
openssl genrsa -des3 -out ca.key 4096
openssl req -new -x509 -days 365 -key ca.key -out ca.crt
```

Zertifikat-Requests für SAM, ROP und CAM erstellen und mit der CA signieren:

```
openssl genrsa -des3 -out rop.key 4096
openssl req -new -key rop.key -out rop.csr
openssl x509 -req -days 365 -in rop.csr -CA ca.crt -CAkey ca.key -set_serial 01 -
    out rop.crt

openssl genrsa -des3 -out sam.key 4096
openssl req -new -key sam.key -out sam.csr
openssl x509 -req -days 365 -in sam.csr -CA ca.crt -CAkey ca.key -set_serial 01 -
    out sam.crt

openssl genrsa -des3 -out cam.key 4096
openssl req -new -key cam.key -out cam.csr
openssl x509 -req -days 365 -in cam.csr -CA ca.crt -CAkey ca.key -set_serial 01 -
    out cam.crt
```

Key+Zertifikat von CAM in einer .pem-Datei zusammenfassen, da Mongoose dieses Format erwartet:

```
cat sam.key sam.crt > sam.pem
```

Zertifikat und Key von ROP für den Browser aufbereiten

```
openssl pkcs12 -export -clcerts -in rop.crt -inkey rop.key -out rop.p12
```

Manuelle Einrichtung

Im Folgenden wird beschrieben, wie das gesamte Szenario auf einem Rechner direkt oder in einer virtuellen Maschine eingerichtet werden kann. Bei der Verwendung der beiliegenden virtuellen Maschine sind diese Schritte nicht erneut auszuführen.

```

# Repositories:
Contiki + Server + Client:
```

```
    git clone git@gitlab.informatik.uni-bremen.de:bergmann/dcaf.git Branch tha
CAM + SAM:
    git@gitlab.informatik.uni-bremen.de:tha/dcaf-cam-sam.git Branch master
tinydtls x86:
    git@gitlab.informatik.uni-bremen.de:tha/tinydtls.git Branch tha-x86
tinydtls contiki:
    git@gitlab.informatik.uni-bremen.de:tha/tinydtls.git Branch tha-contiki
libcoap contiki:
    git@gitlab.informatik.uni-bremen.de:tha/libcoap.git Branch tha-contiki
libcoap x86:
    git@gitlab.informatik.uni-bremen.de:tha/libcoap.git Branch master
    commit #d077f1c4f56c97999a1b4ab3d201f91592d00214

# Contiki und angepasste Libs einrichten

## contiki
$ cd
$ git clone git@gitlab.informatik.uni-bremen.de:bergmann/dcaf.git dcafcontiki
$ cd dcafcontiki
$ git checkout tha
$ cd apps

## tinydtls in contiki
$ git clone git@gitlab.informatik.uni-bremen.de:tha/tinydtls.git tinydtls/
$ cd tinydtls
$ git checkout remotes/origin/tha-contiki
$ autoreconf
$ ./configure --with-contiki --without-debug

## libcoap in contiki
$ mkdir libcoap
$ git clone git@gitlab.informatik.uni-bremen.de:tha/libcoap.git libcoap/
$ cd libcoap
$ git checkout remotes/origin/tha-contiki
$ autoreconf
$ ./configure --with-contiki --without-debug

## Client und Server kompilieren
$ cd ~/dcafcontiki/examples/dcaf/client
$ make TARGET=wismote
$ cd ~/dcafcontiki/examples/dcaf/server
$ make TARGET=wismote

## tinydtls x86 in der VM
$ cd ~ && mkdir tmp && cd tmp
$ git clone git@gitlab.informatik.uni-bremen.de:tha/tinydtls.git
$ cd tinydtls
$ git checkout tha-x86
$ autoreconf
$ ./configure --without-debug
$ make
$ sudo make install

## libcoap x86 in der VM
$ cd ~/tmp
$ sudo apt-get install libjson0 libjson0-dev
$ git clone git@gitlab.informatik.uni-bremen.de:tha/libcoap.git
$ cd libcoap
$ git checkout d077f1c4f56c97999a1b4ab3d201f91592d00214
```

```
$ autoreconf
$ ./configure --without-debug
$ make
$ sudo make install

## SAM und CAM hinzufügen und einrichten
$ sudo apt-get install liburiparser-dev libcurl4-openssl-dev
    libcurl4-openssl-dev liburiparser-dev json-c build-essential
$ cd ~/tmp
$ wget http://www.digip.org/jansson/releases/jansson-2.7.tar.gz
$ tar xvfz jansson-2.7.tar.gz
$ cd jansson-2.7
$ ./configure
$ make check && make
$ sudo make install
$ sudo ln -s /usr/local/lib/libjansson.so.4 /usr/lib/libjansson.so.4

$ cd ~
$ git clone git@gitlab.informatik.uni-bremen.de:tha/dcaf-cam-sam.git dcafcamsam

## CAM
$ cd ~/dcafcamsam/cam
$ make
Binary und Config wird nach ./build/ kopiert. Vorhandene Config in build/
    wird überschrieben!

## SAM
$ cd ~/dcafcamsam/sam
$ make
Binary und Config wird nach ./build/ kopiert. Vorhandene Config in build/
    wird überschrieben!

# Browser konfigurieren
- Mongoose hat bei der Verwendung von TLS und einer eigenen CA zur
    Authentifizierung einen Bug der bei mehreren parallelen HTTPs-Verbindungen
    auftritt und bei dem Requests verloren gehen können. Daher muss im Browser
    die maximale Anzahl paralleler Zugriffe auf 1 begrenzt werden. Einen Fix
    für den Fehler scheint es bisher nicht zu geben
- Firefox starten. In Adresszeile about:config eingeben. "Ill be careful" klicken
- in Filter eingeben: "network.http.max-persistent-connections-per-server"
    und auf "1" stellen
- Zertifikat von SAM hinzufügen: (doch nicht, sondern Ausnahme hinzufügen)
    + Prefences -> Advanced -> Certificates -> View Vertificates
    + Im Tab "Servers" auf Import und das Zertifikat von SAM auswählen:
        /home/user/dcafcamsam/certs/sam.crt (Kein PW)
- Client-Zertifikat hinzufügen (ROPs Zertifikat)
    + Im selben Fenster auf "Your Certificates"
    + Import: /home/user/dcafcamsam/certs/rop.p12 (Kein PW)
- Cert und Preferences schließen
```

Anhang D

Daten-DVD

Dieser Arbeit liegt eine DVD bei, die die Arbeit selbst in digitaler Form, alle in der Arbeit verwendeten Abbildungen, die virtuelle Maschine und die Quelltexte der vier Akteure enthält.