

Bridge the gap or tunnel straight through!

How to connect private MQTT broker

Studiengang
 Autor
 Dozent
 Experte

Bachelor of Science in Computer Science
 Markus Salvisberg und Remo Meyer
 Prof. Dr. Reto Koenig
 Thomas Jäggi

Version 1.0 vom 15. Januar 2023

Abstract

In dieser Thesis zeigen wir, wie MQTT-Nachrichten zwischen privaten, öffentlich nicht erreichbaren MQTT-Brokern ausgetauscht werden können. Die Realisierung erfolgt hierbei ohne tiefgreifende Netzwerk-Konfigurationen wie VPN (Virtual Private Network) oder Port-Forwarding. Es muss auch kein Server im Internet gehostet werden. Die von uns entwickelte Lösung, das «Magic Message Portal», besteht aus zwei Teilen: dem «MQTT-Tunnel» und der «Secure Bridge». Beim MQTT-Tunnel wird mittels WebRTC eine Peer-to-Peer-Verbindung zum Austausch der Daten eingerichtet. Die MQTT-Nachrichten werden über einen WebRTC-Datenkanal direkt zwischen den Brokern in den unterschiedlichen lokalen Netzwerken ausgetauscht. Wenn der Aufbau eines MQTT-Tunnels nicht möglich ist, z.B. wegen zu restriktiven Firewall-Einstellungen oder ungeeigneten NAT-Verhaltens der Gateways, nehmen die Nachrichten den Weg über die Secure Bridge. Dabei werden sie über einen öffentlichen und somit potenziell unsicheren MQTT-Broker gesendet. Um die Vertraulichkeit, Authentizität und Integrität der MQTT-Nachrichten sicherzustellen, werden bei der Secure Bridge die Nutzdaten verschlüsselt.

Inhaltsverzeichnis

Abstract	iii
1 First Chapter: Tunnel und Bridge	1
1.1 Einleitung	1
1.2 Problemstellung und Ziele	1
1.3 Grundlagen	3
1.3.1 Kommunikation über den Gap	3
1.3.2 MQTT - Grundlagen	5
1.3.3 MQTT - Bridging	6
1.3.4 MQTT - Security	9
1.3.5 WebRTC - Grundlagen	10
1.3.6 WebRTC - Protokoll-Stack	11
1.3.7 WebRTC - Architektur	13
1.3.8 NAT-Implementierung und NAT-Traversal	14
1.3.9 Abgrenzung	16
1.4 Resultate	17
1.4.1 MQTT-Tunnel	17
1.4.2 Secure Bridge	23
1.4.3 The Magic Message Portal	27
1.4.4 Fazit und Ausblick	28
2 Second Chapter: Projektbeschrieb	31
2.1 Einführung	31
2.2 WebRTC Technologie-Stack	31
2.2.1 Ähnliche existierende Projekte und Lösungen	33
2.3 WebRTC Prototypen	35
2.3.1 aiortc Code-Basis	35
2.3.2 Prototyp mit Signaling via Copy&Paste	35
2.3.3 Signaling mit MQTT	36
2.3.4 Konzept Secure Signaling mit MQTT	37
2.3.5 Umsetzung Secure Signaling mit MQTT	39
2.3.6 Multichannel	41
2.3.7 Multiinitiator	43
2.4 Testumgebung	44
2.4.1 Virtual Machine BFH als Server	44
2.4.2 SSL Zertifikat für VM BFH	44

2.4.3	Inbetriebnahme Prototyp mit aiortc auf VM BFH	45
2.4.4	Erstellung Flatpak Applikation	45
2.5	Performanz-Tests Bridge vs. Tunnel	48
2.5.1	Durchführung	50
2.5.2	Resultate	51
2.6	Secure Bridge	53
2.6.1	Encryption-Client	54
2.6.2	Geschwindigkeits-Test	61
2.7	The Magic Message Portal	61
2.8	Dynamisches Bridgen	65
2.9	Projektplanung	67
2.10	Schlussbetrachtung	67
Literaturverzeichnis		71
Abbildungsverzeichnis		77
Tabellenverzeichnis		79
Listings		81
Glossar	84
Appendix A: Test setup		85
1	Introduction	85
2	Install and prepare Flatpak	86
3	Install Application Tunnel	86
4	Test the tunnel	86
4.1	Run initiator	86
4.2	Connect to the broker	87

1 First Chapter: Tunnel und Bridge

1.1 Einleitung

Wer mit wem welche Daten austauschen kann und soll sind offensichtliche Fragen im boomenden Zeitalter von «Internet of Things (IoT)» und Industrie 4.x. Nicht so offensichtlich ist die Antwort auf die Frage, *wie* die Daten geteilt werden können. Im Folgenden geht es um das Teilen von Daten mit dem MQTT-Protokoll über MQTT-Broker. Heutzutage ist es üblich, über VPN einen Zugang zu schaffen, so dass auf private MQTT-Broker von aussen zugegriffen werden kann. Diese Art der «privaten gemeinsamen Nutzung» erfordert jedoch tiefgreifende Netzwerk-Konfigurationen und wird daher auf der Ebene der Unternehmen oder einzelner Netzwerke verwaltet. In dieser Bachelorarbeit wollen wir der Frage nachgehen, ob schlankere Lösungen möglich sind, die direkt auf der Anwendungsebene eingesetzt und verwaltet werden können. Konkret werden die Möglichkeiten des «Bridgings», einer Funktionalität von MQTT-Brokern, untersucht. Darüber hinaus wird die Möglichkeit des «Tunnelns» von Daten mit Hilfe von WebRTC erforscht. Beide Varianten haben zum Ziel, die Vertraulichkeit, Authentizität und Integrität der Daten beim Austausch zu gewährleisten, ohne weitreichende Netzwerkeinstellungen vornehmen zu müssen. Es soll möglich sein, das Ganze auf der Anwendungsschicht ohne oder nur mit minimalen externen Abhängigkeiten zu verwalten.

In diesem Teil der Bachelorarbeit präsentieren wir das Resultat unserer Arbeit. Zunächst schildern wir das Ausgangsproblem und unsere Ziele anhand von einem Fallbeispiel. Danach werden die technischen Grundlagen erläutert, welche für das Verständnis nötig sind. Anschliessend werden die von uns erarbeiteten Lösungen beschrieben. Abschliessend ziehen wir ein Fazit zur Arbeit und geben einen Ausblick auf mögliche weitere Entwicklungsschritte.

1.2 Problemstellung und Ziele

Unsere Ausgangslage ist das in Abbildung 1.1 skizzierte Szenario. Bob hat sich zuhause ein «Smart Home» eingerichtet. Er hat diverse Sensoren und Aktuatoren installiert, welche Messwerte liefern und Prozesse steuern. Die Kommunikation zwischen den Akteuren findet mittels MQTT über einen lokalen Broker in seinem privaten lokalen Netzwerk (LAN) statt. Solange Bob zuhause ist, kann er alle

Sensoren und Aktuatoren über eine Smart-Home-Software auf seinem Notebook überwachen und steuern.

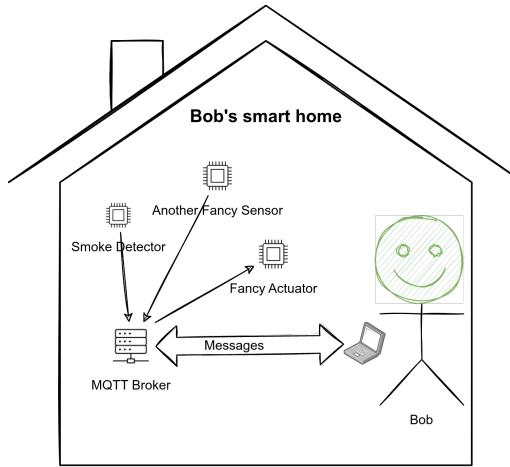


Abbildung 1.1: Ausgangslage

Sobald Bob aber sein lokales Netzwerk verlässt, zeigt ihm seine Smart-Home-Software keine Messwerte an und steuern kann er auch nichts mehr. Denn dazu müssten die Daten irgendwie den Weg durch das «Evil Internet» zu ihm finden. Bob ist mit dieser Situation nicht zufrieden, wie in Abbildung 1.2 gezeigt wird. Im übertragenen Sinne sind mit dem «Evil Internet» alle nicht vertrauenswürdige Server gemeint.

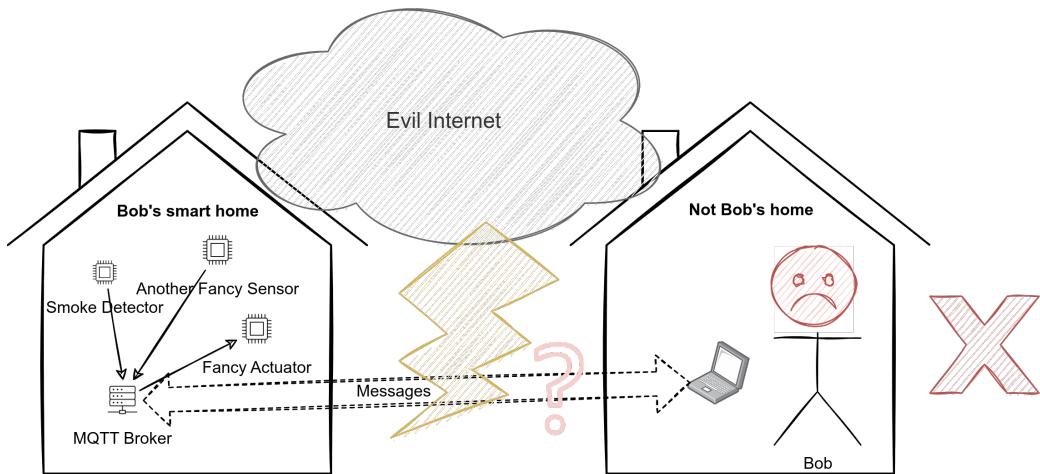


Abbildung 1.2: Problem

Das Ziel dieser Arbeit ist es, Bob zu helfen. Er soll, ohne komplizierte Netzwerk-konfigurationen vornehmen zu müssen oder selber einen Server im Internet zu hosten, Zugang zu seinem MQTT-System haben, auch wenn er sich nicht im ei-

genen LAN befindet. Das wollen wir ihm, wie in Abbildung 1.3 dargestellt, mit unserem «Magic Message Portal» ermöglichen.

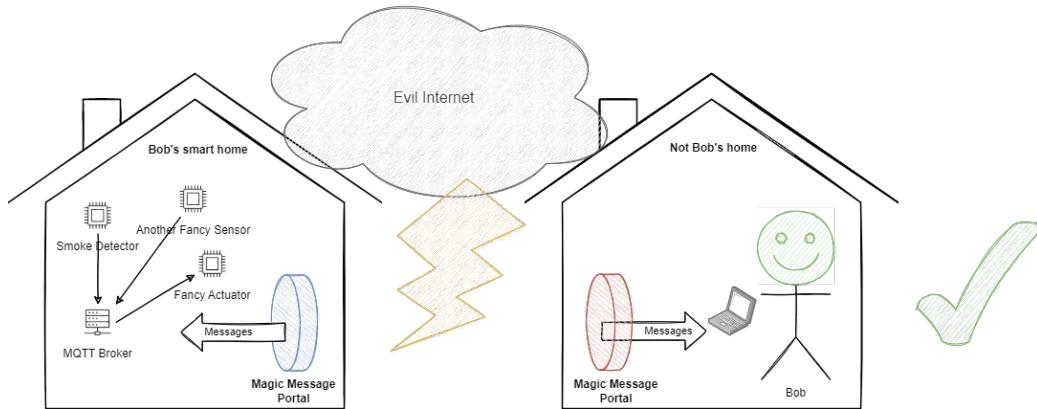


Abbildung 1.3: Ziel

Dabei sollen die Vertraulichkeit (*privacy*), Authentizität (*authenticity*) und Integrität (*integrity*) der MQTT-Nachrichten beim Senden von einem LAN in ein anderes sichergestellt werden. Unser «Magic Message Portal» soll dies auf Applikationsebene umsetzen. Erreichen wollen wir das, indem wir einen NAT-Traversaltunnel mit WebRTC erstellen (Abbildung 1.4), durch welchen wir die Nachrichten tunneln. Als Tunnel wird in einem Rechnernetz die Verwendung eines Netzwerkprotokolls bezeichnet, welches in ein anderes Protokoll eingebettet ist [1]. Falls es nicht möglich ist, einen Tunnel aufzubauen (siehe Kapitel 1.3.1), werden die Nachrichten Ende-zu-Ende verschlüsselt über einen öffentlichen MQTT-Broker gesendet (Abbildung 1.5). Dieses «Fallback»-Szenario bezeichnen wir als «Secure Bridge».

1.3 Grundlagen

1.3.1 Kommunikation über den Gap

Unter «Gap» verstehen wir ein nicht vertrauenswürdiges Netzwerk und meinen damit in der Regel das Internet. Grundsätzlich ist die Kommunikation über den Gap dank dem Internet-Protokoll (IP) kein Problem, sofern die IP-Adressen der Kommunikationspartner bekannt und erreichbar sind. Bei privaten IPv4 Netzwerken wird die Verbindung zum Internet aber in der Regel mittels NAT (Network Address Translation) gewährleistet. Bei NAT werden die lokalen IP-Adressen von privaten Subnetzen auf öffentliche IP-Adressen abgebildet. Dies bedeutet, dass Teilnehmer des privaten Netzwerks von ausserhalb nicht direkt erreichbar sind. Es gibt verschiedene Möglichkeiten um zu erreichen, dass zwei oder mehrere Rechner, die sich in unterschiedlichen lokalen Netzwerken hinter einem

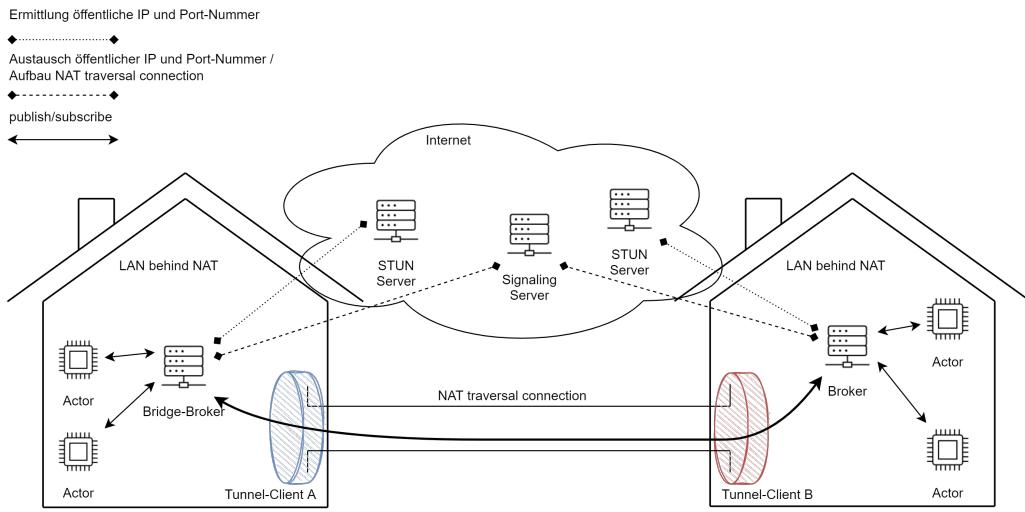


Abbildung 1.4: MQTT-WebRTC-Tunnel

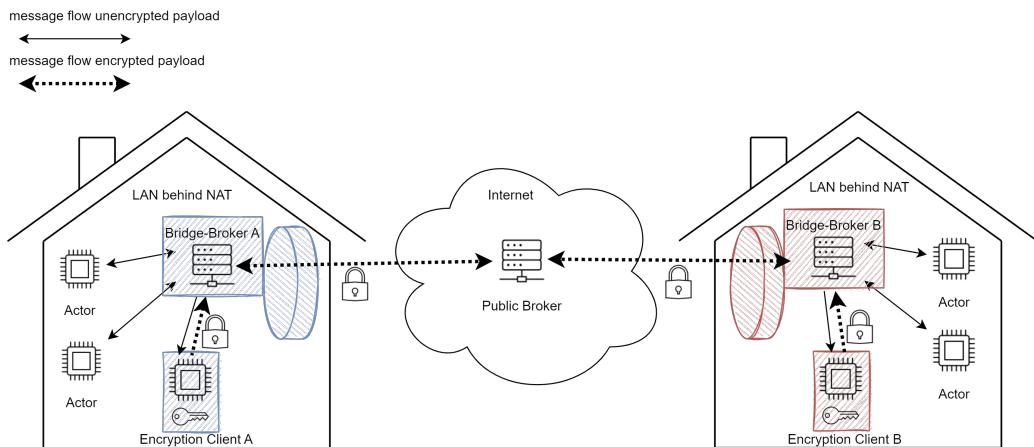


Abbildung 1.5: Secure-Bridge mit verschlüsselten Nutzdaten

NAT befinden, über das Internet miteinander kommunizieren können. Oft angewendet werden sogenannte VPNs. Bei VPN wird mittels Software ein logisches privates Netzwerk aufgebaut. Es gibt eine Vielzahl unterschiedlicher Protokolle und Anwendungen um virtuelle private Netzwerke einzurichten und es gibt zahlreiche VPN-Typen, z.B. End-to-Site-VPN (ein externer Client wird in ein Netz eingebunden), Site-to-Site-VPN (mehrere lokale Netzwerke werden zusammen-

geschaltet) oder End-To-End-VPN (ein Client greift auf einen anderen Client in einem entfernten Netzwerk zu). Zu jedem VPN-Typus gibt es jeweils noch diverse Sub-Typen. Bei allen Varianten muss Server-Infrastruktur betrieben werden, um ein eigenes VPN aufzubauen [2]. Daher kommt VPN für unsere Lösung nicht in Frage.

Eine weitere Möglichkeit wäre das sogenannte «Port-Forwarding». Beim Port-Forwarding wird der NAT-Router manuell so konfiguriert, dass er bestimmte Datenpakete an bestimmte lokale Rechner weiterleitet. Der NAT-Router richtet dabei eine feste Weiterleitung anhand von Port-Nummern (bzw. Portbereichen) und IP-Adressen der lokalen Rechner ein, über welche der lokale Rechner ausserhalb des Netzwerkes erreichbar ist [3]. Da Port-Forwarding einen Zugang zum Router und technisches Verständnis voraussetzt, kommt es für unsere Lösung ebenfalls nicht in Frage.

WebRTC hingegen ermöglicht es, eine Peer-to-Peer Verbindung zum Austausch von Daten einzurichten, ohne dass dazu Einstellungen am Router vorgenommen werden müssen oder selber ein Server gehostet werden muss. Damit erweist es sich als geeignete Technologie zum Umsetzen unseres Ziels in Form des «MQTT-Tunnels».

Es gibt allerdings Fälle, wo WebRTC nicht in der Lage ist eine direkte Verbindung zwischen zwei Clients aufzubauen, z.B. wenn UDP durch einen Paketfilter blockiert wird oder wenn einer der Clients hinter einem «Symmetric-NAT» ist [4] (siehe Kapitel 1.3.8).

Bei der Verwendung eines im Internet erreichbaren MQTT-Brokers hat man keine Probleme mit NAT-Traversal. Will man aber keinen eigenen Broker hosten, werden die Daten über einen öffentlichen und damit nicht vertrauenswürdigen und potenziell unsicheren Broker gesendet. MQTT-Nachrichten können zwar mittels TLS verschlüsselt werden, dies sichert aber nur die Kommunikation zwischen Client und Broker, aber nicht zwischen Client und Client (End-to-End) ab. Auf dem Broker sind die Nutzdaten weiterhin im Klartext vorhanden, was voraussetzt, dass die Broker-Software respektive der Broker-Hoster vertrauenswürdig ist. Um die Vertraulichkeit, Authentizität und Integrität der Nachrichten sicherzustellen, werden wir bei der «Secure Bridge» die Nutzdaten verschlüsseln, bevor sie dem Broker gesendet werden.

1.3.2 MQTT - Grundlagen

MQTT (Message Queue Telemetry Transport) ist ein offenes Netzwerkprotokoll, das spezifisch auf die Bedürfnisse von «Machine-to-Machine» (M2M) Kommunikation und für den Einsatz im Internet of Things zugeschnitten ist. Die MQTT Spezifikation ist ein OASIS (Organization for the Advancement of Structured Information Standards) Standard. Die aktuelle Version, MQTT 5.0, wurde 2019 ver-

öffentlicht [5]. Wie HTTP ist MQTT ein Protokoll aus der Anwendungsschicht und verwendet als Transportprotokoll TCP/IP, TLS oder WebSocket. Die Ports 1883 und 8883 sind bei der IANA (Internet Assigned Numbers Authority) registriert. MQTT funktioniert nach dem Publish/Subscribe-Muster. Clients können Nachrichten auf einem Broker (Server¹) veröffentlichen (*publish*) und Nachrichten abonnieren (*subscribe*). Der Broker agiert als Vermittler zwischen den Clients, er nimmt die von den Clients veröffentlichten Nachrichten an und leitet sie an die Clients weiter, welche sie abonniert haben. Die Publish/Subscribe Architektur ist in Abbildung 1.6 dargestellt.

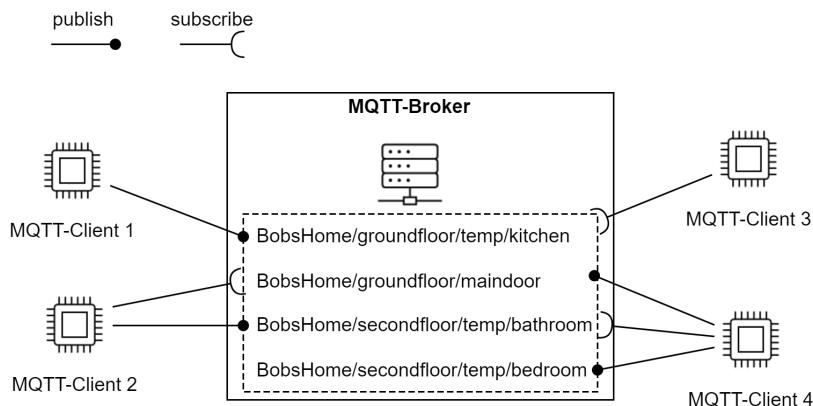


Abbildung 1.6: Publish/Subscribe-Architektur von MQTT

Das Publishen und Subscriben von Nachrichten läuft über eine hierarchische Topic-Struktur (siehe Abbildung 1.7). Ein Topic ist ein UTF-8 String der als Bezeichnung oder Identifikator des Kanals dient, über den die Nachrichten veröffentlicht und abonniert werden können. Im Topic-String wird das «/»-Zeichen zum Trennen der Ebenen (*levels*) innerhalb eines Topic-Baums verwendet. Clients können auf mehrere Topics publishen und subscriben. Dabei ist auch die Verwendung von Platzhaltern (*wildcards*) möglich. Mit dem «#»-Symbol kann man alle Topics, die in der Hierarchie-Ebene tiefer liegen, abonnieren. Das «+»-Zeichen ist ein Hierarchie-Ebenen Platzhalter. Das Topic ist neben dem eigentlichen Dateninhalt (*payload*) ebenfalls Teil einer MQTT-Nachricht.

Es gibt zahlreiche Implementationen von MQTT-Brokern und Clients, eine aktuelle Liste findet sich auf der MQTT-Webseite [6].

1.3.3 MQTT - Bridging

Mit einer Bridge können zwei oder mehrere MQTT-Broker miteinander verbunden werden, um selektiv Nachrichten bzw. Topics untereinander auszutauschen.

¹In der MQTT Spezifikation wird der Begriff «Server» verwendet. In der Literatur wird aber fast ausschliesslich der Begriff «Broker» gebraucht. Wir verwenden im Weiteren den Begriff Broker.

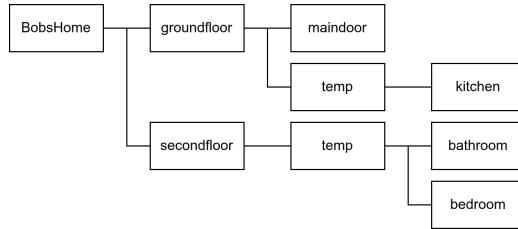


Abbildung 1.7: Beispiel einer Topic Struktur

Das Prinzip des Bridgings wird in Abbildung 1.8 illustriert. Zwei Broker sind jeweils mit mehreren Clients verbunden und bauen zwischen einander eine Bridge auf. So können Clients, welche nicht direkt mit dem gleichen Broker verbunden sind, trotzdem Nachrichten austauschen. Beispielsweise könnte der Bridge-Broker die Topics, auf die Client 1 published, auf Broker 2 bridgen, wodurch Client 5 diese Topics subscriben kann.

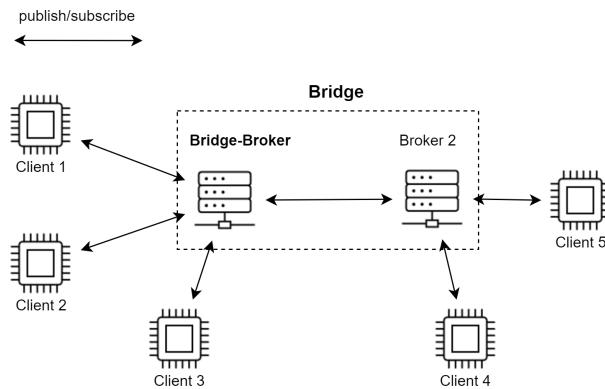


Abbildung 1.8: MQTT Bridge

Bridges sind nicht Teil der MQTT Spezifikation. In der Version 3.1.1 des Protokolls findet sich keine Erwähnung von *bridge*, *bridging* oder Ähnlichem [7]. In der Spezifikation von Version 5.0 werden an zwei Stellen Features genannt, welche im Hinblick auf die Implementation von *bridge applications* und *bridging* dienlich sein könnten [8, 73, 137]. Bridges werden aber von verschiedenen Broker-Implementationen angeboten, unter anderem auch von Mosquitto, HiveMQ, VerneMQ und EMQX. Bei Mosquitto wird zum Erstellen einer Bridge zwischen zwei Brokern die Konfigurationsdatei (*mosquitto.conf*) von einem der Broker, im Beispiel in Abbildung 1.8 «Bridge-Broker» genannt, angepasst. In der Vorlage der Konfigurationsdatei gibt es einen Abschnitt mit allen verfügbaren Einstellungen. Im Listing 1.1 werden die wichtigsten Zeilen und Optionen der Konfiguration gezeigt.

```
1 # =====
2 # Bridges
3 # =====
4 #connection <name>
5 #address <host>[:<port>] [<host>[:<port>]]
6 #topic <topic> [[out | in | both] qos-level]
7
8 # Beispiel 1
9 connection bridging-the-gap
10 address test.mosquitto.org:1883
11 topic # both 0
```

Listing 1.1: Auschnitt mosquitto.conf zu Bridges

Eine Bridge kann mit der «connection»-Option erstellt werden, wobei für jede Verbindung ein eindeutiger Name gewählt werden muss. Bei der «address»-Option muss die Adresse oder der Domainname des Ziel-Brokers angegeben werden. Mit der Option «topic» können die zu bridgenden Topics definiert werden. Die Verwendung von wildcards ist möglich. Mit den Werten *out*, *in* oder *both* kann angegeben werden, in welche Richtung das Topic gebridged wird. Mit *out* werden Topics auf den Remote-Broker exportiert, mit *in* vom Remote-Broker importiert und mit *both* in beide Richtungen ausgetauscht. Auch das QoS-Level kann festgelegt werden. Im Beispiel im Listing 1.1 werden alle Topics auf beide Broker gebridged mit QoS 0. Neben den oben Aufgeführten gibt es noch diverse andere Optionen, um das Verhalten der Bridge einzustellen. In den Kommentaren der Vorlagen-Konfigurationsdatei und auf der mosquitto.conf man page im Absatz «Bridges» [9] finden sich ausführlichere Erklärungen dazu.

Bei näherer Betrachtung stellt man fest, dass sich der Bridge-Broker zum Broker, auf den er die Bridge einrichtet (Broker 2 in Abbildung 1.8) wie ein «normaler» Client verhält. Er published und subscribed Nachrichten auf den anderen Broker.

Bridging - Einsatzszenarien

Im gängigen Anwendungsfall von MQTT basiert das System auf einem Broker, welcher im Internet erreichbar ist. Die Clients verbinden sich aus ihrem lokalen Netzwerk auf den Broker und müssen nicht im Internet exponiert sein (Abbildung 1.9).

Im Rahmen des Moduls Projekt 2 haben wir verschiedene Szenarien untersucht, in denen MQTT-Bridges eingesetzt werden können. Für diese Arbeit ist das folgende, in Abbildung 1.10 skizzierte Szenario relevant. Das MQTT-System ist jeweils pro lokales Netzwerk gekapselt. Die Clients verbinden sich auf den lokalen Broker. Die Verbindung von lokalen MQTT-Netzwerken erfolgt durch die Verwendung eines Brokers im Internet. Die lokalen Broker bridgen hierbei ausgewählte Topics auf den im Internet erreichbaren Broker und stellen so die Verbindung zwischen den gekapselten MQTT-Netzwerken sicher. Diese Architektur hat den

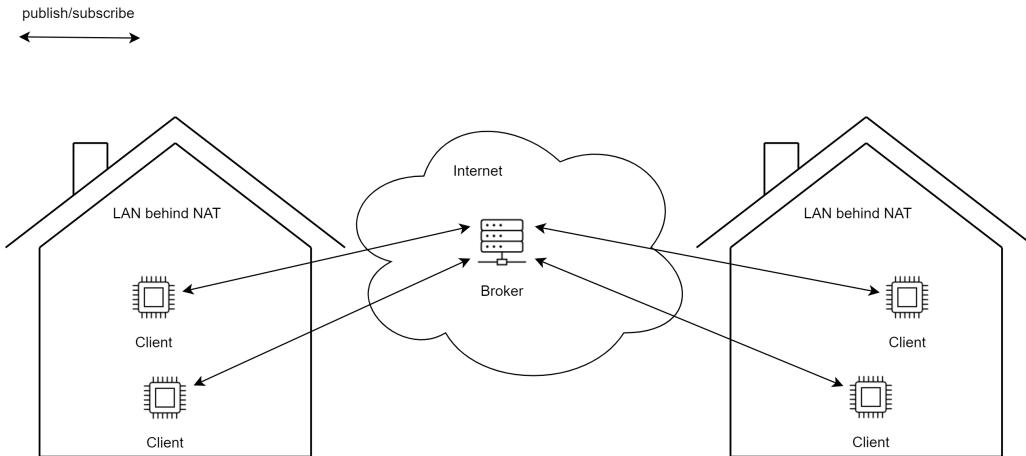


Abbildung 1.9: Zentraler Broker WAN

Vorteil, dass die lokalen MQTT-Systeme auch bei einem Unterbruch der Internetverbindung noch funktionieren.

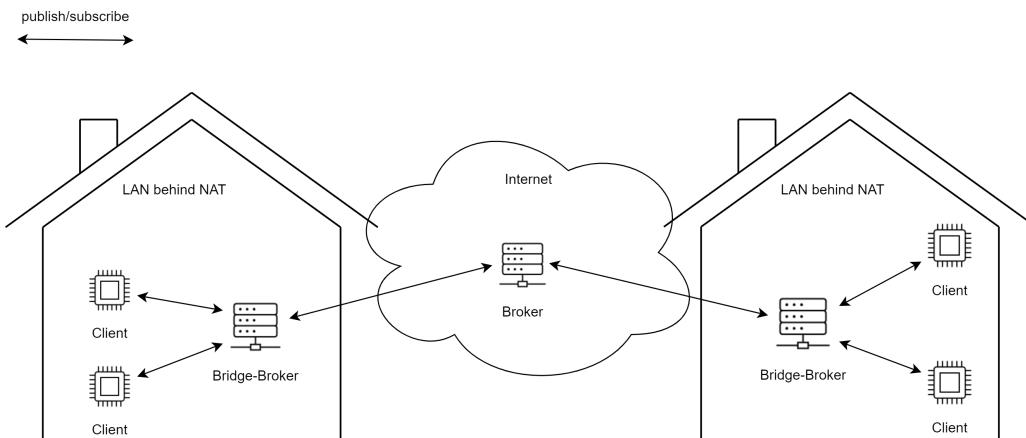


Abbildung 1.10: Lokale Bridge-Broker

1.3.4 MQTT - Security

Folgende Sicherheits-Features bietet das MQTT Protokoll standardmäßig an:

- ▶ **TLS** In einem nicht vertrauenswürdigen Netz ist die Verwendung von TLS zur Sicherung der IP-Verbindung empfohlen. Andernfalls können Passwörter, welche im Klartext übertragen werden, problemlos abgefangen werden.

- ▶ **Client-Authentifizierung** Die MQTT-Spezifikation sieht eine Authentifizierung Benutzername und Passwort vor. Diese können in der jeweiligen Implementation nach Belieben zur Authentifizierung verwendet werden. Beim Einsatz von TLS können sich die Clients alternativ mittels Zertifikats authentifizieren [7, 61].
- ▶ **Client-Zugriffsberechtigung** In der MQTT-Spezifikation ist keine konkrete Implementation der Verwaltung der Zugriffsberechtigung von Clients spezifiziert. Erwähnt ist jedoch, dass ein solcher Mechanismus, basierend auf den verfügbaren Merkmalen, realisiert werden kann (Benutzername, Client-ID, Hostname usw.) [7, 61].
- ▶ **Broker-Authentifizierung** Eine Authentifizierung des Brokers ist nur mittels TLS und Zertifikat vorgesehen [7, 61].
- ▶ **Enhanced Authentication von MQTT 5.0** Mit der MQTT-Spezifikation Version 5.0 ist die Enhanced Authentication eingeführt worden. Ziel dieser Erweiterung ist die Bereitstellung des Kommunikationsablaufs für die Verwendung von Authentifizierungsmethoden auf Basis von Challenge-Response-Protokollen [8, 107]. Enhanced Authentication wird zum Beispiel vom SMOKER-Protokoll verwendet, welches die Vergabe von Zugriffsberechtigungen durch den Client erlaubt. Ein Client kann die Berechtigungen hierbei nur für seine eigenen Topics vergeben [10].
- ▶ **Broker spezifische Features** Beim Mosquitto-Broker von Eclipse steht eine Access Control List (ACL) zur Verwaltung der Zugriffsberechtigungen zur Verfügung. Die Berechtigungen werden in diesem Fall auf Basis des Benutzernamens vergeben. Die Konfiguration von Benutzern und Passwörtern erfolgt im Password File und werden ebenfalls zur Client-Authentifizierung verwendet. Die Subscribe- und Publish-Berechtigungen sind in der ACL pro Benutzer Topic-spezifisch aufgeführt. Wildcards sind bei der Spezifikation der Topic-Freigabe zugelassen [9].

1.3.5 WebRTC - Grundlagen

WebRTC (Web Real-Time Communication) definiert eine Sammlung von Kommunikationsprotokollen und einer JavaScript Programmierschnittstelle (API), welche eine Echtzeitkommunikation zwischen zwei oder mehreren Rechnern ermöglichen. 2021 wurde WebRTC nach 10-jährigen Standardisierungsarbeiten vom World Wide Web Consortium (W3C) und der Internet Engineering Task Force (IETF) als offener Standard veröffentlicht [11]. Wesentlich beteiligt an der Entwicklung des WebRTC-Projekts waren und sind Google, Apple, Microsoft und Mozilla. Der Quellcode der Implementierung sowie die meisten Informationen über WebRTC sind frei verfügbar [12]. WebRTC wurde mit Fokus auf Anwendungen in

Browsern entwickelt. Das Ziel war es, Echtzeitkommunikation² zwischen Browsern ohne Plugins oder andere zusätzliche Softwareinstallationen zu ermöglichen. Heute unterstützt jeder bedeutende Browser WebRTC. Neben der JavaScript API für Browser ist für native Clients wie Android- und iOS-Apps eine Bibliothek verfügbar, welche die gleichen Funktionen bietet. Neben zahlreichen Webseiten und Webapplikationen gibt es auch einige sehr bekannte Applikationen die WebRTC verwenden, u.a. Google Hangouts, Facebook Messenger, WhatsApp, Discord oder Snapchat. Das WebRTC API besteht im Wesentlichen aus drei Teilen [13]; [14]:

- ▶ **MediaStream oder getUserMedia** Ermöglicht den Zugriff auf die Kamera und das Mikrofon eines Geräts, die Erfassung von Audio- und Videoströmen sowie deren Anschluss an eine RTC-Verbindung.
- ▶ **RTCPeerConnection** Bietet Methoden zum Herstellen, Aufrechterhalten, Überwachen und Schliessen der Verbindung zwischen einem lokalen Computer und dem entfernten Peer.
- ▶ **RTCDATAChannel** Kann für die bidirektionale Peer-to-Peer-Übertragung beliebiger Daten verwendet werden.

Dieses API ermöglicht es, Video- und Audiostreaming im Browser oder in nativen Applikationen mit wenigen Zeilen Code zu implementieren. Im Hintergrund werden dazu aber zahlreiche und vielfältige Technologien verwendet.

1.3.6 WebRTC - Protokoll-Stack

WebRTC verwendet eine Vielzahl von unterschiedlichen Protokollen. Die Abbildung 1.11 zeigt den von WebRTC verwendeten Protokoll-Stack [13] [15, 32].

- ▶ **IP** (Internet Protocol, RFC 791) In der Internetschicht wird das Internet Protokoll verwendet.
- ▶ **UDP** (User Data Protocol, RFC 768) Um bei Audio- und Videoübertragungen eine gute Qualität zu erreichen, ist die Übertragungsgeschwindigkeit zentral. Dabei ist es erlaubt, bestimmte Datenframes auslassen, um die Geschwindigkeit der Verbindung aufrechtzuerhalten. Es ist wichtiger, das aktuellste Datenpaket zu senden, als sicherzustellen, dass jedes auf der anderen Seite ankommt. Deshalb verwendet WebRTC UDP anstelle vom zuverlässigen, verbindungsorientierten TCP (Transmission Control Protocol) [16, 31-32].
- ▶ **ICE** (Interactive Connectivity Establishment, RFC 8445) ICE ermöglicht

²Echtzeit (real time): Daten werden unmittelbar nachdem sie generiert wurden, übermittelt, nicht in einem bestimmten Intervall oder erst dann, wenn sie abgefragt werden. Zwischen Generierung, Übermittlung und Empfang der Daten sollte eine möglichst geringe Latenzzeit bestehen

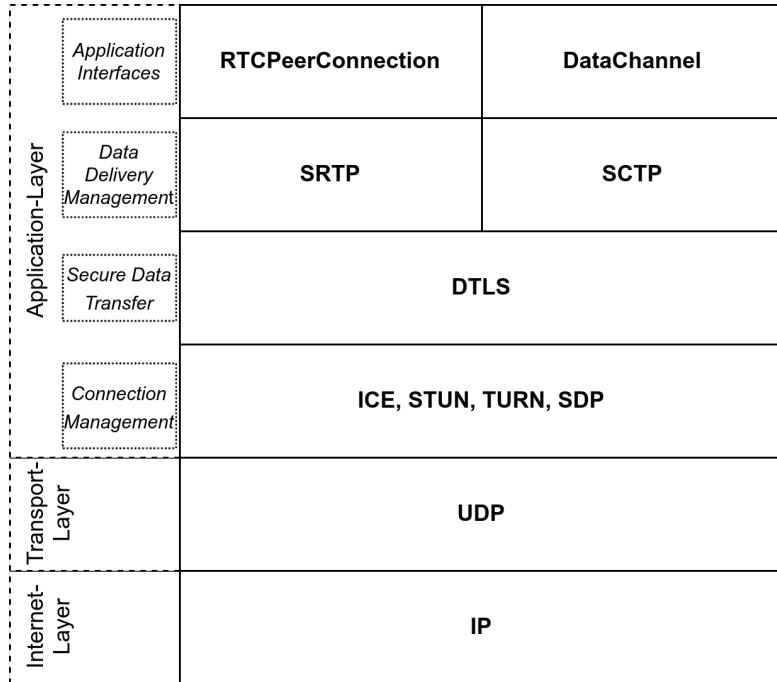


Abbildung 1.11: WebRTC Protokoll-Stack

NAT-Traversal für UDP basierte Kommunikation. Es verwendet STUN und TURN.

- ▶ **STUN** (Session Traversal Utilities for NAT, RFC 8489) Ermittelt die IP-Adresse und den Port der Endpunkten (Peers/Clients), welche ihnen vom NAT zugewiesen wurden. Es gibt öffentliche Server, welche STUN kostenlos anbieten, u.a. von Google (stun.l.google.com:19302) [17]. Zudem gibt verschiedene Open Source Implementationen wie coturn [18] oder STUNTMAN [19] wenn man den STUN-Server selber hosten möchte.
- ▶ **TURN** (Traversal Using Relays around NAT, RFC 8656) Dient als Ausweichlösung, falls der Aufbau einer direkten Verbindung nicht möglich ist. Der Kommunikationskanal zwischen den Peers wird dann über einen TURN-Relay-Server, der sich im öffentlichen Internet befindet, hergestellt. Es gibt mehrere Optionen für TURN-Server, sowohl für selbst gehostete Anwendungen (z.B. coturn, siehe Abschnitt zu STUN) als auch für online Dienste (gratis z.B. [20], kostenpflichtig z.B. [21]).
- ▶ **SDP** (Session Description Protocol, RFC 8866) Mithilfe vom SDP wird die Session-Identifikation übertragen und spezifiziert, welche Daten über die WebRTC-Verbindung übertragen werden.
- ▶ **DTLS** (Datagram Transport Layer Security, RFC 9147) Ermöglicht Verschlüsselung.

selung für UDP, basiert auf TLS (Transport Layer Security), dem Verschlüsselungsprotokoll zur Sicherung von TCP-Verbindungen.

- ▶ **SCTP** (Stream Control Transport Protocol, RFC 9260) Ermöglicht eine zuverlässige und geordnete Paketübertragung (analog zu TCP) auf Basis von UDP, welches verbindungslos ist.
- ▶ **SRTP** (Secure Real-Time Transport Protocol, RFC 3711) Ist die sichere (verschlüsselt und authentifiziert) Version von RTP (Real-time Transport Protocol). SRTP beschreibt, wie verschiedene Medien (Audio, Video) von einem Endpunkt zum anderen in Echtzeit übertragen werden können.

1.3.7 WebRTC - Architektur

WebRTC verwendet einen Peer-to-Peer (P2P) Ansatz. Die Parteien einer Kommunikation sollen nach Möglichkeit die Daten direkt untereinander austauschen, ohne auf einen Server als Mittelsmann angewiesen zu sein. Um die Probleme zu beheben, die durch NAT entstehen (siehe Kapitel 1.3.1), verwendet WebRTC das ICE Protokoll mit STUN und TURN [15]. ICE nutzt STUN zur Ermittlung der öffentlichen IP/Port-Kombinationen pro Client für mögliche direkte Verbindungen zwischen den Peers. Falls keine direkte Verbindung aufgebaut werden kann, sieht WebRTC die Möglichkeit vor, einen TURN-Server zu verwenden, über den die Daten ausgetauscht werden. Die durch STUN oder TURN ermittelten Verbindungsinformationen, ICE-Kandidaten genannt, werden zusammen mit weiteren wichtigen Informationen für den Verbindungsaufbau zwischen den Peers ausgetauscht. Wie diese Informationen beschrieben werden, wird im Session Description Protocol (SDP) spezifiziert. Der Austausch der SDP-Nachrichten mit den ICE-Kandidaten zwischen den Peers wird Signaling genannt. In Abbildung 1.12 wird der Ablauf beim Verbindungsaufbau zwischen zwei Peers über das Signaling aufgezeigt.

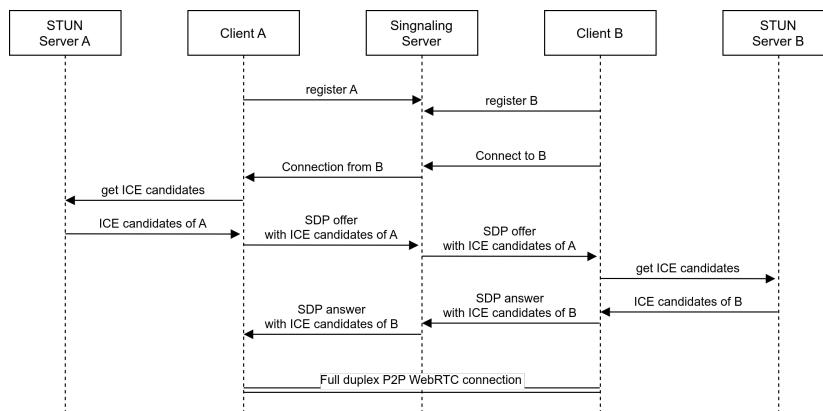


Abbildung 1.12: Ablauf Verbindungsaufbau WebRTC mit Signaling

Der Signaling Prozess gehört nicht zur Spezifikation von WebRTC. Es gibt verschiedene Möglichkeiten, diesen umzusetzen, u.a. mittels einfachem copy-and-paste, WebSockets, XMPP, AJAX oder MQTT. Grundsätzlich müssen nur die SDP-Nachrichten mit den ICE-Kandidaten zwischen den Peers ausgetauscht werden können. Über welchen Kanal dies geschieht lässt der WebRTC Standard offen. Sobald die Peers ihre ICE-Kandidaten und Verbindungsinformationen ausgetauscht haben, können sie eine direkte Verbindung herstellen.

Die Abbildung 1.13 zeigt eine Übersicht über die WebRTC Peer-to-Peer Architektur [15, 29].

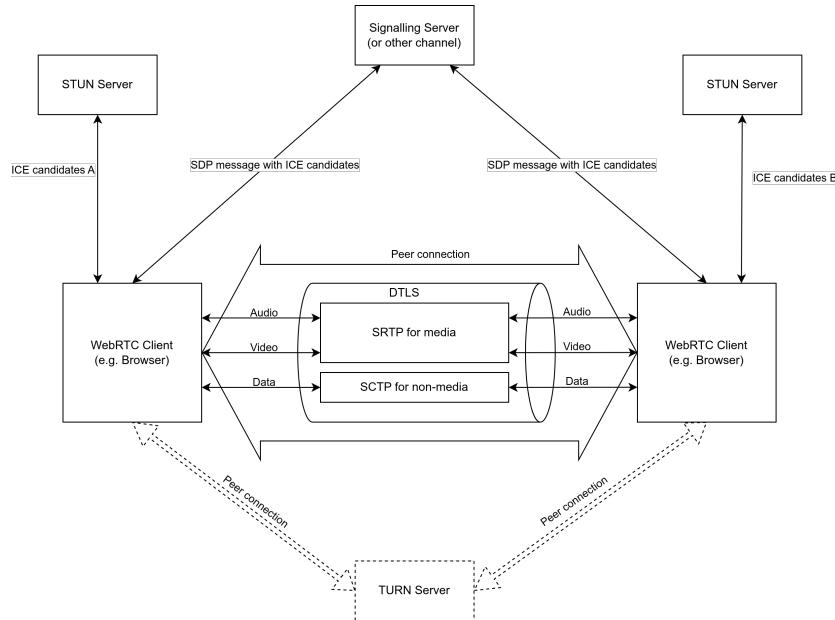


Abbildung 1.13: WebRTC Peer-to-Peer Architektur

1.3.8 NAT-Implementierung und NAT-Traversal

Beim obsoleten RFC 3489 «STUN» werden vier verschiedene NAT-Typen spezifiziert, welche die Kategorisierung von gängigen NAT-Implementationen erlauben sollen. Im Zusammenspiel mit einem STUN-Server kann ein Client herausfinden, welche NAT- und Firewall-Konfigurationen bei seinem Gateway vorhanden sind und feststellen, ob ein NAT-Traversal möglich ist. Die unterschiedlichen NAT-Typen sind nachfolgend beschrieben.

- **Full Cone NAT:** Bei dieser Konfiguration werden interne Adressen und Ports statisch auf externe Adressen und Ports übersetzt. Dies erlaubt auch einen Verbindungsaufbau durch einen externen Host. Diese Variante entspricht einer «Port Forwarding»-Konfiguration.

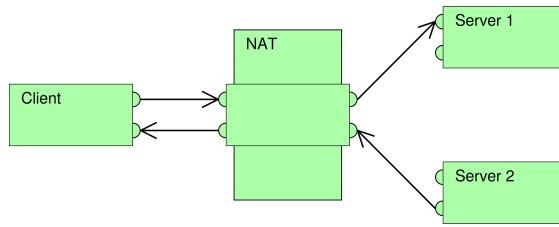


Abbildung 1.14: Full Cone NAT [22]

- ▶ **Restricted Cone NAT:** Bei dieser Konfiguration werden interne Adressen und Ports statisch auf externe Adressen und Ports übersetzt. Pakete von einem externen Host werden jedoch erst weitergeleitet, sobald dieser ein Paket vom internen Host erhalten hat.

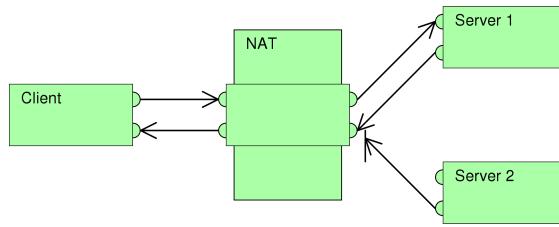


Abbildung 1.15: Restricted Cone NAT [22]

- ▶ **Port restricted Cone NAT:** Diese Konfiguration entspricht dem «Restricted Cone NAT» jedoch mit der weiteren Einschränkung, dass Pakete von einem externen Host nur weitergeleitet werden, falls dieser vorgängig vom selben internen Host und gleichem Port ein Paket erhalten hat.

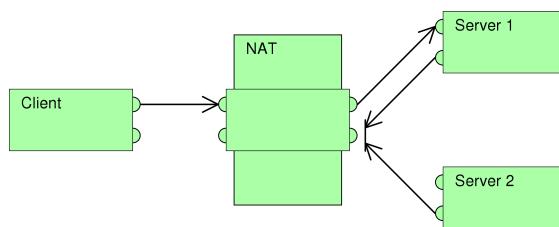


Abbildung 1.16: Port restricted Cone NAT [22]

- ▶ **Symmetric NAT:** Bei einem symmetrischen NAT wird für jede Verbindung zu einem externen Host ein neuer Mapping-Eintrag mit einer separaten ex-

ternen Port-Nummer erstellt. Die Verbindung muss vom internen Host initialisiert werden.

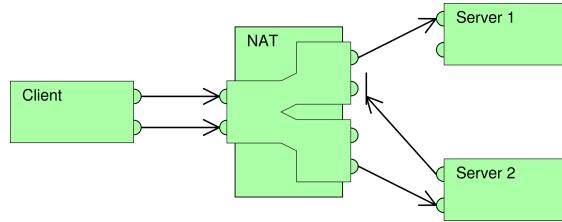


Abbildung 1.17: Port restricted Cone NAT [22]

Im aktuellen RFC 8489 für STUN wird jedoch auf diese Typisierung verzichtet, da die Klassifizierung nicht immer korrekt vorgenommen werden kann und sich die Charakteristik der NAT teilweise während dem Betrieb ändern kann [22]. Was jedoch weiterhin stimmt ist, dass mit einem «Symmetric NAT» kein NAT-Traversal mit WebRTC möglich ist. Dieser Umstand wird auch im RFC 4787 «NAT UDP Unicast Requirements» erwähnt. Im »REQ-1« heisst es, dass ein NAT ein vom Endpunkt unabhängiges Mapping verwenden muss, um den Empfehlungen zu folgen [23, 7]. In der ICE-Spezifikation (RFC8445) wiederum wird angegeben, dass ein NAT-Traversal funktioniert, sobald die Empfehlungen im RFC 4787 umgesetzt werden. Andernfalls wird der Einsatz eines TURN-Servers wahrscheinlich [24, 70].

1.3.9 Abgrenzung

Das Ziel dieser Arbeit ist es nicht, am Ende ein zur Veröffentlichung geeignetes Produkt zu haben. Vielmehr geht es darum herauszufinden, ob grundsätzlich mit dem gewählten Mittel (WebRTC, Secure Bridge) eine Lösung unseres Problems (Kommunikation über den Gap) möglich ist. Es handelt sich somit um eine explorative Arbeit mit einem «Proof of Concept» als Ziel.

Die Sicherheit der verwendeten kryptographischen Algorithmen wird im Rahmen dieser Arbeit nicht untersucht oder beurteilt. Auch gehen wir nicht auf die Sicherheit von WebRTC an sich ein. Eine eingehende Betrachtung dazu ist öffentlich zugänglich [25].

1.4 Resultate

1.4.1 MQTT-Tunnel

Der eine Teil unseres Magic Message Portal besteht aus dem «Application Layer Tunnel», mit welchem es möglich ist, eine direkte Verbindung zwischen einem MQTT-Client und einem MQTT-Broker herzustellen, obwohl sich diese beiden Teilnehmer nicht im selben LAN befinden. Der hier genannte MQTT-Client kann auch ein Broker sein, welcher eine Bridge-Verbindung erstellt. Der Application Layer Tunnel ermöglicht generell die Übertragung eines Applikationslayers, welcher auf TCP basiert, durch einen WebRTC-Tunnel. Da in unserer Anwendung MQTT der Applikationslayer ist, wird im weiteren Verlauf dieser Arbeit «MQTT-Tunnel» als Bezeichnung verwendet. In der Abbildung 1.18 sind die verschiedenen Teile dieser Applikation dargestellt, wobei die von uns implementierte Funktionalitäten blau hervorgehoben sind.

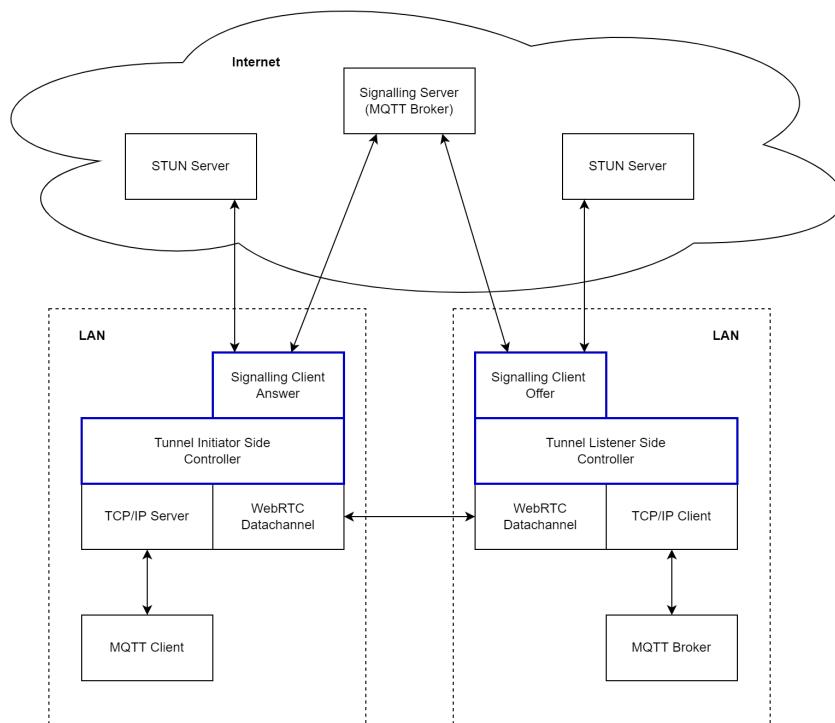


Abbildung 1.18: Übersicht Tunneling Client

Für eine Tunnelverbindung werden immer mindestens ein *Listener* und ein *Initiator* benötigt. Der Listener befindet sich im LAN mit dem MQTT-Broker, während der Initiator im LAN die Client-Seite abbildet. Nach dem Signaling mit Hilfe des Signaling-Clients wird eine WebRTC PeerConnection zwischen dem In-

initiator und dem Listener aufgebaut. Jede TCP-Verbindung auf den Initiator wird anschliessend über einen WebRTC Datenkanal getunnelt. In der Abbildung 1.19 ist dargestellt, wie der Tunnel auf den verschiedenen Transportlayern agiert. Im Prinzip werden die Daten vom Applikationslayer (blau), hier MQTT, ohne weitere Analyse oder Bearbeitung über den WebRTC Datenkanal übertragen. Auf der anderen Seite werden die entsprechenden Daten über TCP/TLS an das Ziel weitergeleitet.

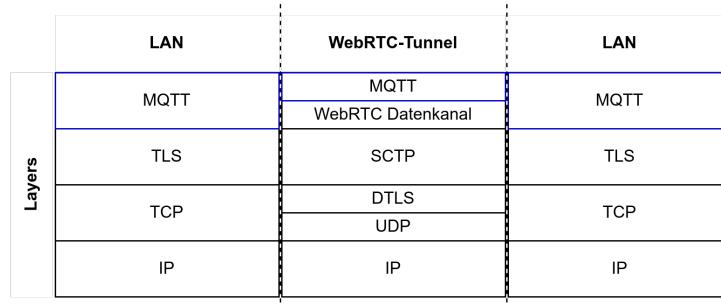


Abbildung 1.19: Verbindungsebenen MQTT-Tunnel

Wie bereits bei der Einführung zu WebRTC erwähnt, wird der Signaling-Prozess nicht durch WebRTC spezifiziert. Uns stellt sich daher die Frage, ob wir eine entsprechende freie Implementierung verwenden wollen, oder ob wir selbst eine Lösung erarbeiten. Viele der vorhandenen Lösungen basieren auf einem WebSocket-Server, welcher für das Pairing und die Authentifizierung zuständig ist. Wir wollten jedoch möglichst eine Applikation, welche ohne spezifische Server-Instanz auskommt und wo sich die Teilnehmer gegenseitig authentifizieren. Da für die Secure Bridge (siehe Kapitel 1.4.2) in jedem Fall ein MQTT-Broker vorhanden sein muss, erarbeiteten wir eine Lösung basierend auf MQTT.

Das Signaling ist bezüglich der Sicherheit der PeerConnection entscheidend und muss daher entsprechend verschlüsselt werden. Bei unserer Lösung sind die Signaling-Nachrichten mittels Public-Key-Kryptographie «End-to-End» verschlüsselt. In der Abbildung 1.20 ist der implementierte Signaling-Ablauf dargestellt.

Der Initiator muss zwingend den öffentlichen Schlüssel (*public key*) des Listenern kennen, damit das Signaling erfolgreich sein kann. Die öffentlichen Schlüssel werden jeweils in die MQTT-Topicstruktur integriert und dienen so gleichzeitig zur Identifikation der Teilnehmer. Grundsätzlich published der Initiator auf das Offer-Topic und subscribed auf das Answer-Topic, während Listener gegenteilig verfährt. Die Topic-Strukturen sind nachfolgend dargestellt.

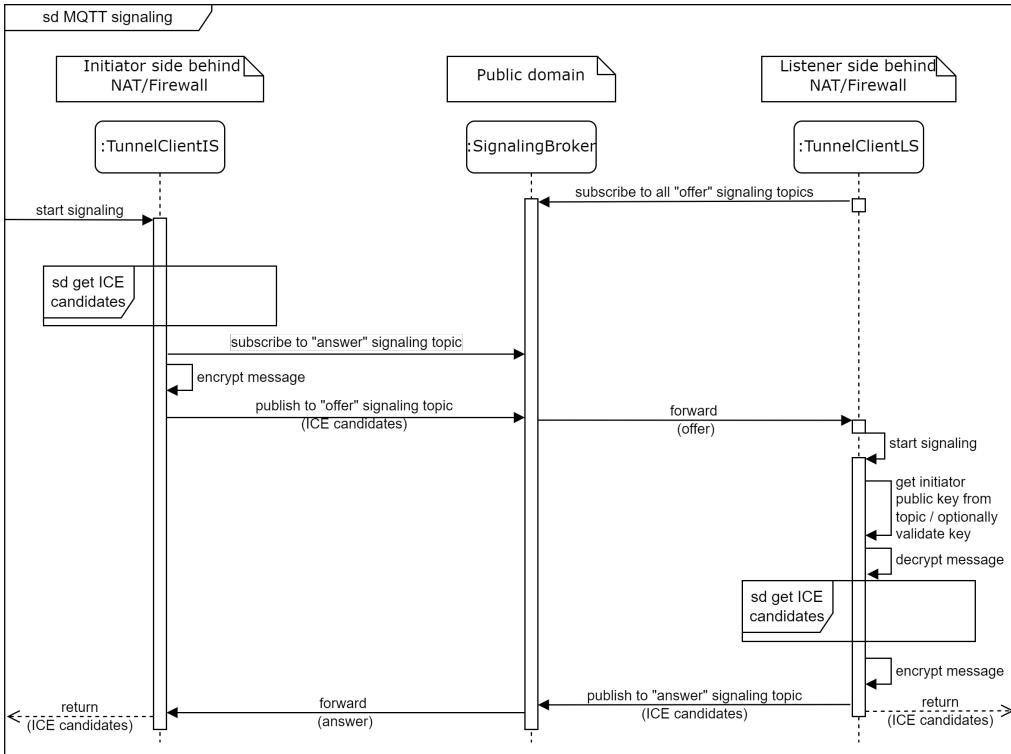


Abbildung 1.20: MQTT Signaling

Offer Topic

```
sig/<BrokerPubKeyBase32>/<ClientPubKeyBase32>/<ChannelId>/o
```

Answer Topic

```
sig/<BrokerPubKeyBase32>/<ClientPubKeyBase32>/<ChannelId>/a
```

Dem Listener kann eine Liste mit den öffentlichen Schlüsseln von zulässigen Initiatoren hinterlegt werden. Optional besteht jedoch auch die Möglichkeit, mit allen anfragenden Initiatoren eine PeerConnection aufzubauen. Damit der öffentliche Schluessel in die Topic-Struktur passt und keine MQTT-Sonderzeichen enthält, ist er Base32 codiert [26]. Für die Verschlüsselung wird PyNaCl [27] verwendet. PyNaCl ist eine Python API zur LibSodium [28], welche wiederum ein portabler und erweiterter Fork von NaCl (Salt) [29] ist. NaCl ist mit dem Ziel entwickelt worden, eine benutzerfreundliche, schnelle und sichere Open-Source Kryptographie-Bibliothek bereitzustellen. Für die Public-Key-Verschlüsselung wird der Curve25519-Algorithmus verwendet, welcher auf einer elliptischen Kurve beruht [30] [31] [27].

Die Application Tunnel Anwendung ist als Kommandozeilenwerkzeug realisiert

und kann gemäss dem API auf dem Listing 1.2 als «Listener», «Initiator» oder als «Key Generator (keygen)» aufgerufen werden.

```
1 flatpak run ch.bfh.ti.applic-tunnel -h
2 usage: Application layer tunnel [-h] {initiator,listener,keygen} ...
3
4 A program to tunnel a TCP application layer by a RTC data channel. The
5 signaling protocol is based on MQTT with message based end to end encryption.
6 A listener instance is waiting for initiators, which can be optionally
7 approved by a list of public keys.
8
9 positional arguments:
10    {initiator,listener,keygen}
11        initiator      run applic tunnel as initiator
12        listener       run applic tunnel as listener
13        keygen        generate a key pair
14
15 optional arguments:
16    -h, --help           show this help message and exit
```

Listing 1.2: Aufruf Application Tunnel

Mit Hilfe des «KeyGenerator» wird in einem spezifischen Ordner ein neues Schlüsselpaar generiert. Die Schlüssel werden jeweils als Text-Datei mit dem Schlüssel als Base32-Zeichenfolge abgelegt (Listing 1.3).

```
1 flatpak run ch.bfh.ti.applic-tunnel keygen -h
2 usage: Application layer tunnel keygen [-h] [-p PATH] [-v]
3
4 optional arguments:
5     -h, --help           show this help message and exit
6     -p PATH, --path PATH path to save the keys
7     -v, --verbose
```

Listing 1.3: Aufruf Application Tunnel «KeyGenerator»

Die API-Beschreibungen als Initiator und Listener sind in den Listings 1.4 und 1.5 dargestellt. Beim Initiator ist die Übergabe eines privaten Schlüssels optional. Falls kein Schlüssel definiert wird, erstellt die Applikation pro Aufruf ein temporäres Schlüsselpaar. Die Option «notls» bezieht sich jeweils auf den Transportlayer fürs Signaling. Die PeerConnection ist in jedem Fall verschlüsselt. Zu Testzwecken können die Signaling-Nachrichten unverschlüsselt übertragen werden. Hierzu muss die Option «unsecure» aktiviert werden. Bei einem produktiven System müsste diese Option zwingend entfernt werden, um Fehlbedienungen zu vermeiden.

```

1 flatpak run ch.bfh.ti.applic-tunnel initiator -h
2 usage: Application layer tunnel initiator [-h] [-u SIGUSER] [-p SIGPASSWORD]
3                                     [-t SIGTOPIC] [-n TUNNELNAME]
4                                     [--notls] [--unsecure] [-v]
5                                     [-pk PRIVATEKEY]
6                                     sigbrokername sigbrokerport
7                                     connectionip connectionport
8                                     listenerpubkey
9
10 positional arguments:
11   sigbrokername      hostname or ip of signaling broker
12   sigbrokerport      port on signaling broker
13   connectionip       connection ip
14   connectionport     connection port
15   listenerpubkey    Public key of listener as Base32 string
16
17 optional arguments:
18   -h, --help          show this help message and exit
19   -u SIGUSER, --siguser SIGUSER
20           signaling user name
21   -p SIGPASSWORD, --sigpassword SIGPASSWORD
22           signaling password
23   -t SIGTOPIC, --sigtopic SIGTOPIC
24           signaling topic
25   -n TUNNELNAME, --tunnelname TUNNELNAME
26           tunnel name
27   --notls            do not use TLS
28   --unsecure         use no application encryption mode (not recommended).
29           Activate on both sides.
30   -v, --verbose
31   -pk PRIVATEKEY, --privatekey PRIVATEKEY
32           Path to private key of initiator. Use random key if
33           not set.

```

Listing 1.4: Aufruf Application Tunnel «Initiator»

```

1 flatpak run ch.bfh.ti.applic-tunnel listener -h
2 usage: Application layer tunnel listener [-h] [-u SIGUSER] [-p SIGPASSWORD]
3                                         [-t SIGTOPIC] [-n TUNNELNAME]
4                                         [--notls] [--unsecure] [-v]
5                                         [-i INITIATORS]
6                                         sigbrokername sigbrokerport
7                                         connectionip connectionport
8                                         privatekey
9
10 positional arguments:
11   sigbrokername      hostname or ip of signaling broker
12   sigbrokerport      port on signaling broker
13   connectionip       connection ip
14   connectionport     connection port
15   privatekey         Path to private key
16
17 optional arguments:
18   -h, --help          show this help message and exit
19   -u SIGUSER, --siguser SIGUSER
20                           signaling user name
21   -p SIGPASSWORD, --sigpassword SIGPASSWORD
22                           signaling password
23   -t SIGTOPIC, --sigtopic SIGTOPIC
24                           signaling topic
25   -n TUNNELNAME, --tunnelname TUNNELNAME
26                           tunnel name
27   --notls             do not use TLS
28   --unsecure          use no application encryption mode (not recommended).
29                           Activate on both sides.
30
31   -v, --verbose
32   -i INITIATORS, --initiators INITIATORS
33                           Path to file with trusted initiators, trust every
                           initiator if not set

```

Listing 1.5: Aufruf Application Tunnel «Listener»

Wie aus den dargestellten Aufrufen ersichtlich ist, wird der Application Tunnel als flatpak-Applikation bereitgestellt. Flatpak ist eine Software zur Verteilung von Anwendungen unter Linux-Betriebssystemen [32]. In der Flatpak-Applikation sind alle nötigen Ressourcen gekapselt, was zu einer portablen Anwendung führt, welche unter den diversen Linux-Distributionen lauffähig ist.

Die Abbildung 1.21 zeigt eine mögliche Konfigurationen mit Application Tunnels und Bridge-Brokern. Der Hauptbroker A befindet sich im LAN A und ist über den Tunnel Client A (als Listener konfiguriert) erreichbar. Die Broker aus den anderen LAN bauen eine Bridge-Verbindung zum Hauptbroker auf. Dies geschieht jeweils über den lokalen Tunnel Client im Initiator-Modus. Ein Tunnel Client Listener kann Verbindungen zu mehreren Initiatoren gleichzeitig aufbauen und aufrechterhalten. Für jeden Initiator wird eine neue PeerConnection aufgebaut. Im LAN B ist exemplarisch dargestellt, dass ein Application Tunnel auch mehrere MQTT-Verbindungen resp. Clients handhaben kann. Der IoT Actor B3 stellt direkt eine Verbindung via Tunnel Client zum Broker A her. Pro getunnelte MQTT-Verbindung wird in der PeerConnection ein separater Datenkanal betrieben. Die

drei TCP-Verbindungen, welche über den Application Tunnel geführt werden, sind als TCP1, TCP2 und TCP3 gekennzeichnet.

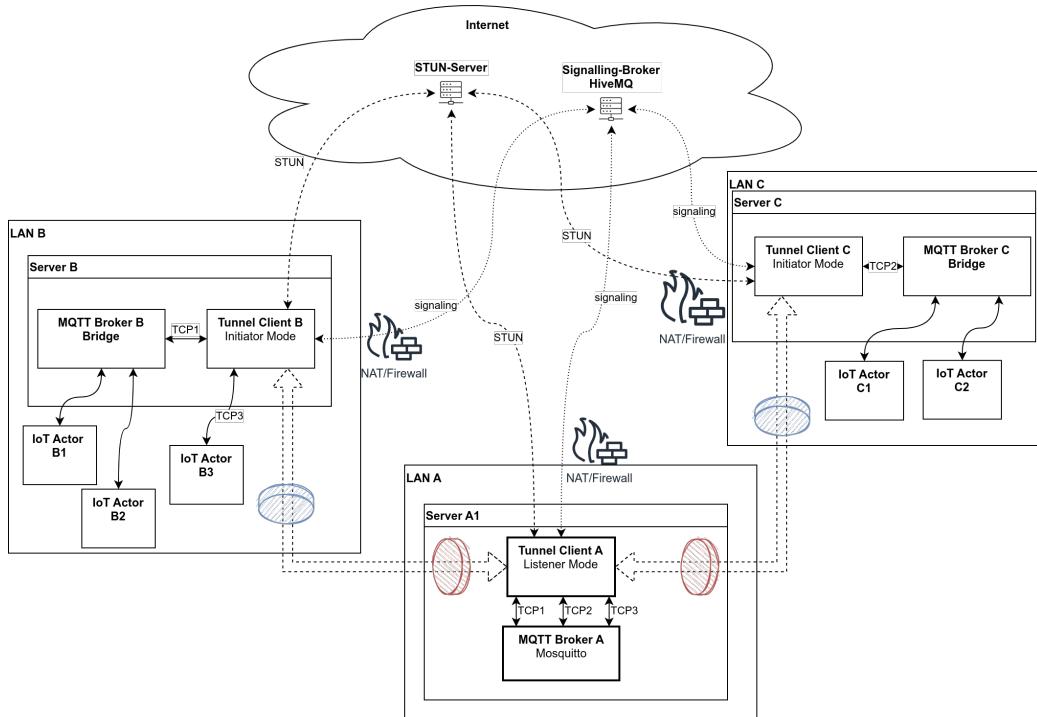


Abbildung 1.21: Mögliche Setups mit Application Tunnel

1.4.2 Secure Bridge

Wenn es nicht möglich ist, den Application Tunnel zwischen den Peers aufzubauen, können die Nachrichten, wie im Kapitel 1.2 eingeführt und in Abbildung 1.5 dargestellt, den Weg über die «Secure Bridge» nehmen. Dabei werden die Daten Ende-zu-Ende verschlüsselt über einen öffentlichen MQTT-Broker gesendet. Das «Magic Message Portal» wird in dem Fall aus einer Kombination von einem Bridge-Broker und einem «Encryption-Client» gebildet. Der Bridge-Broker ist für das Bridgen der gewünschten Topics auf bzw. von einen öffentlich erreichbaren Broker zuständig. Der Encryption-Client wird zum Ausführen der kryptographischen Funktionen eingesetzt. In Abbildung 1.22 wird anhand von einem Beispiel aufgezeigt, wie der Nachrichtenaustausch zwischen zwei lokalen Brokern mittels Secure Bridge funktioniert.

Der MQTT-Client A im LAN A published Nachrichten mit unverschlüsseltem Payload auf das Topic «securebridge/1». Der Encryption-Client subscribed dieses Topic, verschlüsselt den Payload der darüber erhaltenen Nachrichten und published sie auf das Topic «encrypted/securebridge/1». Dieses Topic wird vom loka-

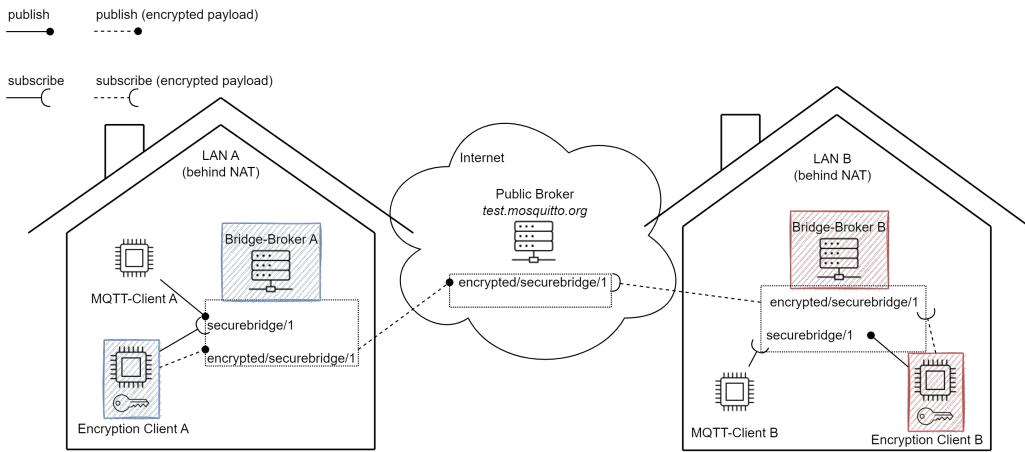


Abbildung 1.22: Nachrichtenaustausch zwischen lokalen Brokern mit der Secure Bridge».

len Bridge-Broker A auf den öffentlichen Broker gebridded. Der Bridge-Broker B subscribed darauf und holt so die Nachrichten in das LAN B. Dort subscribed sich der Encryption-Client B auf «encrypted/securebridge/1». Er entschlüsselt den Payload und published die Nachricht auf das Topic «example/1», welches vom MQTT-Client B subscribed wird. Auf diese Weise können grundsätzlich alle Topics vom Broker in LAN A auf den Broker in LAN B oder umgekehrt gespiegelt werden. Beim Starten des Encryption-Clients können die gewünschten Topics als Befehlszeilenargumente übergeben werden. Dabei wird auch festgelegt, ob der Client die «encrypt»-Rolle (wie im LAN A) oder die «decrypt»-Rolle ausführen soll. Die Nachrichten werden, wie im Beispiel gezeigt, so gepublisiert, dass dem ursprünglichen Topic ein neues first-level-Topic (erste Ebene in der Topic Hierarchie) vorangestellt wird. Dieses kann ebenfalls beim Programmaufruf mitgegeben werden. Das default first-level-Topic lautet «encrypted». In der decrypt-Rolle wird standardmäßig das bestehende first-level-Topic entfernt. So sind die Topics auf den beiden Brokern identisch. Es kann aber auch ein anderes first-level-Topic angegeben werden.

Die Secure Bridge Anwendung ist, wie der MQTT-Tunnel, als Kommandozeilenwerkzeug realisiert und wird auch als flatpak-Applikation bereitgestellt. Die Clients im obigen Beispiel sind mit den im Listig 1.6 gezeigten Befehlen gestartet worden.

```

1 # Encryption Client in LAN A
2 flatpak run ch.bfh.ti.secure-bridge encrypt test.mosquitto.org 1883 --keyfile
   secret_key.bin --notls
3
4 # Encryption Client in LAN B
5 flatpak run ch.bfh.ti.secure-bridge decrypt test.mosquitto.org 1883 --keyfile
   secret_key.bin --notls

```

Listing 1.6: Programmaufrufe für Beispiel in Abbildung 1.22

Die ersten drei Argumente geben die Rolle, den Broker und den Port an. Wird ein öffentlicher Broker verwendet, bei dem Benutzername und Passwort zur Authentifizierung angegeben werden müssen, können diese mit der «user» bzw. «user_pw» Option übergeben werden. Mit der Option «keyfile» wird der Pfad zu dem zur Ver- oder Entschlüsselung verwendeten symmetrischen Schlüssel angegeben. Eine andere Option ist die Verwendung eines Passwort, dieses wird mit «password» übergeben. Wenn andere Topics gebridged oder andere first-level-Topics benutzt werden sollen, können diese mit «s_topics» bzw. «p_topics» festgelegt werden. Falls die Verbindung zum öffentlichen Broker nicht mit TLS gesichert werden soll, wie in unserem Beispiel, muss die Option «notls» gesetzt werden. Das Listing 1.7 zeigt die Ausgabe der help-Funktion wo alle Befehlszeilenargumente beschrieben werden.

```

1 flatpak run ch.bfh.ti.secure-bridge -h
2 usage: Encryption Client [-h] [--user USER] [--user_pw USER_PW] [--keyfile KEYFILE]
   [--password PASSWORD] [--s_topics [S_TOPICS ...]] [--p_topic P_TOPIC] [--notls] [-v]
   {encrypt,decrypt} brokername brokerport
3
4 MQTT message encryption or decryption
5
6 positional arguments:
7   {encrypt,decrypt}   'encrypt': encrypts and publishes messages from and to specified
   topics. 'decrypt': decrypts and publishes messages from and to specified topics
8   brokername          hostname or ip of broker
9   brokerport          port on broker
10
11 options:
12   -h, --help           show this help message and exit
13   --user USER          user name
14   --user_pw USER_PW    user password for broker
15   --keyfile KEYFILE    Path to key-file
16   --password PASSWORD  secret password for encryption
17   --s_topics [S_TOPICS ...]
18           pass a list of topics to subscribe to; role 'decrypt': default
           value is encrypted/#role 'encrypt': default value is
           'securebridge/#subscribe to #' is not possible (creates
           loop, kills broker)
19   --p_topic P_TOPIC    Change first level of topic to publish messages to default
           behaviour if nothing entered: 'encrypt' role: add 'encrypted' as first level
           topic 'decrypt' role: remove current first level topic
20   --notls              do not use TLS
21   -v, --verbose        verbose output

```

Listing 1.7: Secure-Bridge CLI Optionen

Im Listing 1.8 wird die Konfiguration der beiden Bridge-Broker zum obigen Beispiel gezeigt. Im Beispiel werden nur zwei Broker verwendet. Grundsätzlich können aber beliebig viele Broker eingesetzt werden.

```
1  #--- Broker A ---
2  # connection name
3  connection securebridge
4  # addresses
5  address test.mosquitto.org:1883
6  # topics
7  topic securebridge/# out 1
8
9  #--- Broker B ---
10 # connection name
11 connection securebridge
12 # addresses
13 address test.mosquitto.org:1883
14 # topics
15 topic securebridge/# in 1
```

Listing 1.8: Konfiguration der Bridge-Broker A und B

Auf den lokalen Brokern liegen die Payloads unverschlüsselt vor, im Internet und auf dem öffentlichen Broker sind sie aber immer nur in verschlüsselter Form. Für die Verschlüsselung des Payloads werden Funktionen der PyNaCl-Library verwendet. Bei der «keyfile»-Variante kommt ein symmetrisches Verfahren zum Einsatz, bei dem sowohl die Ver- als auch die Entschlüsselung mit demselben geteilten Schlüssel durchgeführt wird. In der *encrypt()*-Funktion wird zur Verschlüsselung der «XSalsa20 stream cipher»-Algorithmus angewendet. Zusätzlich wird durch diese Funktion auch sichergestellt, dass die Payloads nicht unbemerkt manipuliert werden können. Dazu wird ein mit dem «Poly1305 MAC»-Algorithmus generierter MAC (Message Authentication Code) in die verschlüsselten Daten integriert und beim Entschlüsseln mit der *decrypt()*-Funktion geprüft. Damit bei Payloads mit identischem Inhalt nicht derselbe Verschlüsselungstext entsteht, verwendet die *encrypt()*-Funktion standardmäßig eine *Nonce*. Das ist eine Zeichenfolge, die bei der Verschlüsselung einmalig generiert und ein einziges Mal verwendet wird. Die Nonce wird zusammen mit dem verschlüsselten Inhalt und dem MAC als EncryptedMessage-Objekt zurückgegeben [27, 11-14]. Der Encryption-Client sendet dieses Objekt als Payload einer MQTT-Nachricht an den Broker. Durch diese kryptographischen Funktionen können die Anforderungen bezüglich Vertraulichkeit, Integrität und Authentizität der Daten erfüllt werden. Die Verteilung des Schlüssels auf die Clients wird nicht durch die Applikation vorgenommen sondern erfolgt über andere Kanäle (*out-of-band*), beispielsweise per Mail oder Memory-Stick. Mit dem «Secure Bridge Key Generator» stellen wir ein Kommandozeilenwerkzeug als flatpak-Applikation bereit, welches einen symmetrischen Schlüssel generieren kann. Das Listing 1.9 zeigt den Aufruf mit der help-Funktion.

```

1 flatpak ch.bfh.ti.secure-bridge-keygen -h
2 usage: Symmetric key generator [-h] path
3
4 Generate a symmetric key for encrypting MQTT payload
5
6 positional arguments:
7   path      Path to store keyfile to
8
9 options:
10  -h, --help  show this help message and exit

```

Listing 1.9: Secure-Bridge Key Generator Aufruf

In der «password»-Variante wird die Verschlüsselung mit einem Passwort vorgenommen. Dabei wird die *kdf()*-Funktion von PyNaCl verwendet. Diese leitet aus einem Passwort und einer zufällig generierten Zeichenfolge, *Salt* genannt, einen Schlüssel ab [27, 24-26]. Durch das Salt wird die Sicherheit des Schlüssels erhöht. Dieser Prozess wird als Schlüsselstreckung (*Key Stretching*) bezeichnet [33]. Der so generierte Schlüssel wird den gleichen Funktionen übergeben wie in der «keyfile»-Variante. Beim Senden des EncryptedMessage-Objekts muss zusätzlich das verwendete Salt mitgegeben werden, damit die *decrypt()*-Funktion die Nachricht entschlüsseln kann.

Die Probleme des NAT-Traversals stellen sich bei der Secure Bridge nicht. Der MQTT-Client, in dem Fall der Bridge-Broker, initiiert die TCP Verbindung zum öffentlichen Broker dessen Adresse bekannt ist. Dieser hält die Verbindung offen um das bidirektionale Senden und Empfangen von Nachrichten zu ermöglichen. Daher gibt es kein Problem mit Clients, die sich hinter einem NAT befinden [34, 12].

1.4.3 The Magic Message Portal

Mit Hilfe des WebRTC-Tunnels und der Secure Bridge Verschlüsselungapplikation kann nun in Kombination mit der Bridge-Funktionalität der MQTT-Broker ein «Magic Message Portal» konfiguriert werden. Bei diesem wird primär versucht, eine Bridge-Verbindung über den Application Tunnel aufzubauen. Falls dies scheitert, wird als Fallback eine Secure-Bridge aufgebaut. Das entsprechende Setup ist in der Abbildung 1.23 dargestellt.

Beim Broker A sind eine primäre Bridge Adresse (Applic-Tunnel-Client) und eine alternative Bridge Adresse (Public-Broker) konfiguriert. Falls die primäre Verbindung scheitert, wird die alternative Verbindung verwendet. Ist die alternative Verbindung aktiv, wird zyklisch versucht wieder auf die primäre Verbindung zu wechseln. Diese Funktionalität wird bereits durch den Mosquitto-Broker bereitgestellt [9]. Der Bridge des Broker B subscribed permanent auf das Secure-Bridge-Topic auf dem Public Broker.

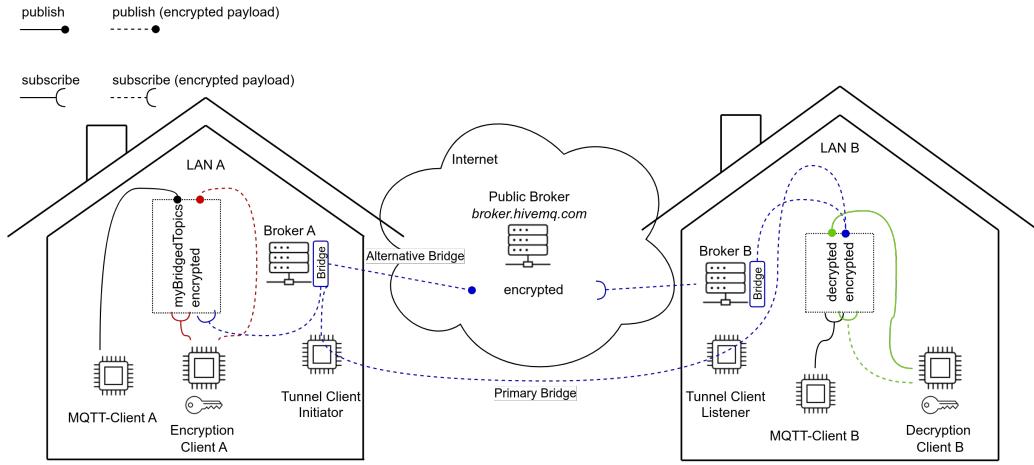


Abbildung 1.23: The Magic Message Portal

Bei diesem Aufbau existiert der Schwachpunkt, dass der Payload der Nachrichten auch verschlüsselt wird, falls die Verbindung über den sicheren Application Tunnel aufgebaut ist. Dies wäre nicht erforderlich und bindet somit unnötig Rechnerleistung.

Zurzeit kann auch nur eine unidirektionale Bridge ohne grössere Einschränkungen eingerichtet werden. Im zweiten Teil der Dokumentation sind die detaillierteren Überlegungen hierzu aufgeführt.

1.4.4 Fazit und Ausblick

Mit dem «Magic Message Portal» konnten wir einen Proof of Concept umsetzen und zwei Möglichkeiten aufzeigen, MQTT-Nachrichten verschlüsselt zwischen lokalen Brokern auszutauschen ohne dabei auf Server- oder Netzwerkebene tätig werden zu müssen. Der Funktionsumfang des «Magic Message Portal» müsste sicher auf ein mögliches Szenario hin optimiert werden. Je nach Verwendung ist eine Realisierung des gesamten Codes in einer effizienteren Programmiersprache angebracht (z.B. in Rust oder C++).

Für die allgemeine Verwendung des Applikationslayer-Tunnel fehlt noch eine graphische Benutzeroberfläche, um Tunnels einfacher zu administrieren. Auf der Listener-Seite wäre zum Beispiel ein Docker-Version sinnvoll.

Beim Tunneling mittels WebRTC wird ein relativ geringer Teil von WebRTC genutzt. Allenfalls macht die Realisierung einer abgespeckten Version ohne Mediendatenprotokolle (Audio/Video) Sinn, um unnötige Abhängigkeiten zu eliminieren. Es gibt bereits entsprechende Forks und Implementation, welche jedoch nicht aktiv gepflegt werden [35] [36].

Grundsätzlich kann mit der heutigen Implementation jeder Applikationslayer ge-

tunnelt werden, welcher auf TCP transportiert wird. MQTT ist somit nur der Anfang. Wir haben im Verlauf der Arbeit bereits SSH getunnelt, um den Fernzugriff auf einen Raspberry Pi zu gewährleisten.

Dies ist eine der wichtigsten Erkenntnisse aus der Arbeit: Es ist mittels WebRTC relativ einfach, einen Applikationslayer-Tunnel bereitzustellen. Für spezifische Problemstellungen bietet dieser Ansatz einen alternativen Lösungsweg an Stelle von VPN oder Port-Weiterleitung.

2 Second Chapter: Projektbeschrieb

2.1 Einführung

In diesem Teil der Arbeit beschreiben wir den Prozess, der zur im ersten Teil der Arbeit präsentierten Lösung geführt hat. Dabei zeigen wir, welche Arbeitsschritte wir geplant und ausgeführt haben. Wir führen aus, auf welche Probleme wir gestossen sind und wie wir diese gelöst haben. Zudem begründen wir Technologie-Entscheide und legen Überlegungen und Erkenntnisse aus der Recherche zu den verschiedenen Aspekten der Arbeit dar.

2.2 WebRTC Technologie-Stack

Eine erste Herausforderung bei der Einarbeitung in WebRTC ist der umfangreiche Technologie-Stack, der verwendet wird. Wir mussten uns mit diversen, uns bisher nicht bekannten Protokollen vertraut machen. Ein Problem, das hinzukommt ist, dass viele der Informationen, die man auf Webseiten und in Büchern findet, etwa 10 Jahre alt sind. Damals war WebRTC neu öffentlich zugänglich und es wurde viel darüber publiziert. Neuere Publikationen finden sich weniger. Die technische Entwicklung ist aber weitergegangen. Daher sind beispielsweise die Verweise auf die verwendeten Protokoll-RFCs veraltet. Das ist neben der Komplexität und dem Umfang der verwendeten Technologien ein möglicher Grund, warum nicht immer alle Angaben aus den verfügbaren Quellen stringent sind. Gerade wenn es um die allgemeine Architektur und das Zusammenspiel der Komponenten geht, ist es schwierig eine übersichtliche, korrekte Darstellung zu finden. Beispielsweise Grigorik [13], der sehr anschaulich in das Thema einführt und schöne Grafiken hat, zeichnet bei der Abbildung 2.2, WebRTC protocol stack, die DTLS Schicht nicht als durchgängig ein (also so, dass SRTP ohne DTLS möglich wäre) obwohl es bei Mozilla folgendes heisst: «All of the WebRTC related protocols are required to encrypt their communications using DTLS; this includes SCTP, SRTP, and STUN. » [37] Weinrank et al. [15] stellen den Protokollstapel zwar korrekt aber unserer Meinung nach weniger anschaulich dar. Wir haben schliesslich aus den beiden Abbildungen in [13] und [15] (siehe Abbildungen 2.1 und 2.2) unsere eigene Darstellung vom Protokoll-Stack zusammengestellt (siehe Abbildung 1.11).

Eine weitere Herausforderung war, dass WebRTC, wie das «Web» im Namen

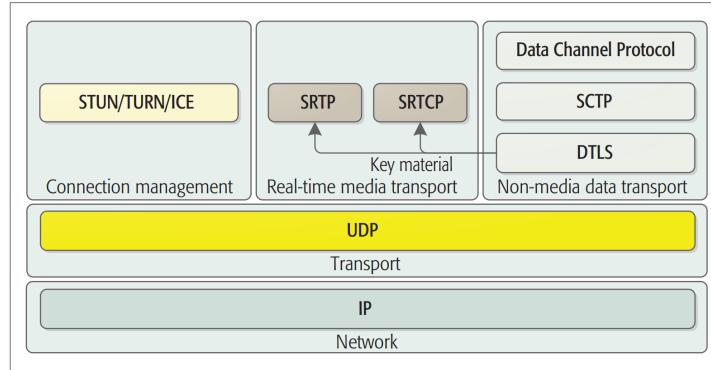


Abbildung 2.1: WebRTC Protokoll-Stack nach Weinrank [15]

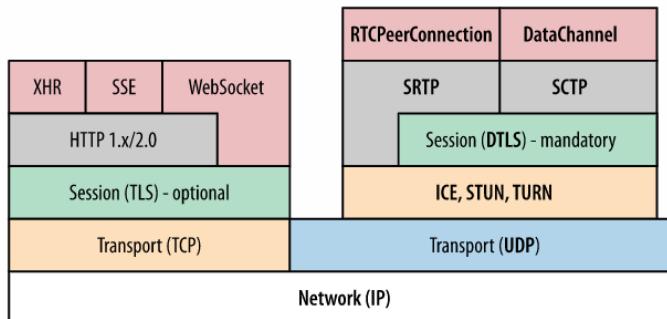


Abbildung 2.2: WebRTC Protokoll-Stack nach Grigorik [13]

schon vermuten lässt, primär für den Einsatz in Browsern ausgelegt ist. Jedenfalls sind die überwiegende Mehrheit der Informations-Webseiten, Tutorials und Sachbücher, die sich zum Thema finden, auf Implementierungen im Browser ausgelegt. Der Fokus liegt dabei oft auf der MediaStream API, bei der es um den Zugriff auf die Kamera und das Mikrofon sowie um die Manipulation der Audio- und Video-Daten geht. Für die Integration von WebRTC in native Applikationen gibt es vor allem für Android gute Beispiele und Tutorials [38]. Dazu, wie man WebRTC für native Desktop-Applikationen und mit Fokus auf den Datenkanal verwenden kann, finden sich deutlich weniger Informationen. Eine gut dokumentiert, verständliche Library zu finden, bei der sich die Beispiele auch kompilieren lassen, bedurfte einiges an Recherche und Tests. Ein Beitrag auf Stackoverflow hat uns einige mögliche Kandidaten geliefert [39]. Auch für unsere «Studiums»-Programmiersprache Java haben wir mit «webrtc-java» eine Library gefunden, welche die Entwicklung von RTC-Anwendungen für Desktop-Plattformen ermöglichen soll [40]. Diese Library wäre von der Programmiersprache her unsere erste Wahl gewesen. Leider gibt es keine Dokumentation dazu. Zudem haben wir die Beispiele aus dem Repository nicht zum Laufen gebracht. Daher war für uns webrtc-java keine Option. Von den möglichen Kandidaten aus dem erwähnten Stackoverflow Beitrag und weiterer Recherchen fiel unsere Wahl

schliesslich auf die Python Library «aiortc» [41]. Zum einen aufgrund der Programmiersprache, da wir uns schnell einig waren, Python gegenüber Rust, Go, C oder C++ den Vorzug zu geben. Zum anderen überzeugte uns die gut verständliche API und dass wir die Beispiele des offiziellen Repositories auf Anhieb zum Laufen brachten.

2.2.1 Ähnliche existierende Projekte und Lösungen

Im Zuge der Auseinandersetzung mit dem Problem haben wir nach bereits existierenden Lösungen gesucht. Dabei haben wir zunächst keine ähnlichen Projekte gefunden. Suchanfragen mit den Begriffen «WebRTC, MQTT, messages, send, DataChannel» und ähnlichen haben keine passenden Treffer geliefert. Hier kamen vor allem Resultate im Zusammenhang mit dem Signaling-Prozess, welcher in verschiedenen Projekten mit MQTT umgesetzt wird (z.B. beim IBM WebSphere Application Server [42]). Nachdem wir uns eingehender mit dem Thema auseinandergesetzt hatten und dadurch die dazugehörigen Begrifflichkeiten besser kannten und besser beschreiben konnten, was wir umzusetzen versuchten, sind wir auf ähnliche Projekte gestossen:

- ▶ **RTCTunnel**, build network tunnels over WebRTC: Eine Open Source Applikation die TCP über WebRTC tunnelt, geschrieben in Go [43].
- ▶ **RTC Tunnel**, RTC Tunneling for sockets: Ziel dieses Projekts ist es, SSH über WebRTC zu tunneln [44].
- ▶ **SSHX**, P2P SSH using WebRTC: Ebenfalls ein SSH-Tunnel Projekt, geschrieben in Go [45].

Dabei kommt das «RTC Tunnel»-Projekt unserem am nächsten. Es verwendet ebenfalls Python und die aiortc-Library. Auch wenn wir zunächst etwas enttäuscht waren, dass unsere Idee doch nicht neuartig war, bestätigte es uns zumindest in unserem eingeschlagenen Weg.

Synology QuickConnect

Auf die Frage von unserem Experten Thomas Jäggi hin, ob Synology für den Remote-Zugang zu seinen Produkten WebRTC verwendet, haben wir versucht dies herauszufinden. Synology ist einer der führenden Server und NAS-System Hersteller. Mittels «QuickConnect» ermöglicht Synology, dass sich Client-Anwendungen über das Internet mit einem NAS verbinden können, ohne dass Portweiterleitungsregeln eingerichtet werden müssen [46]. Im White Paper zu dieser Technologie sucht man den Begriff WebRTC und auch damit zusammenhängende wie STUN, TURN oder ICE zwar vergeblich, aber es finden sich sehr viele Ähnlichkeiten mit der WebRTC-Architektur. Bei QuickConnect wird

versucht, eine temporäre, direkte Verbindung für die Datenübertragung herzustellen, falls sich Client und NAS nicht im selben Netzwerk befinden und nicht direkt miteinander kommunizieren können. Dieses, im Paper «Hole Punching» genannte Verfahren entspricht dem, was das Signalling bei WebRTC ist. Falls das Hole Punching nicht erfolgreich ist, wird die Verbindung über einen «Relay Service» hergestellt, welcher einen Synology Relay-Server verwendet, analog zum TURN-Server bei WebRTC [46, 5-6]. Auch bei den Grafiken lassen sich deutliche Parallelen erkennen, wie man in den Abbildungen 2.3 und 2.4 gut sehen kann.

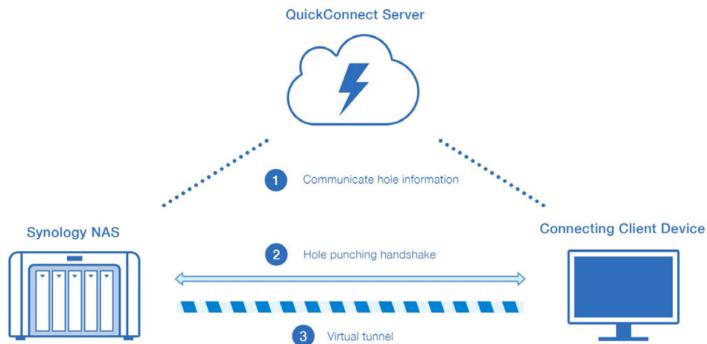


Abbildung 2.3: QuickConnect Hole Punching [46, 6]

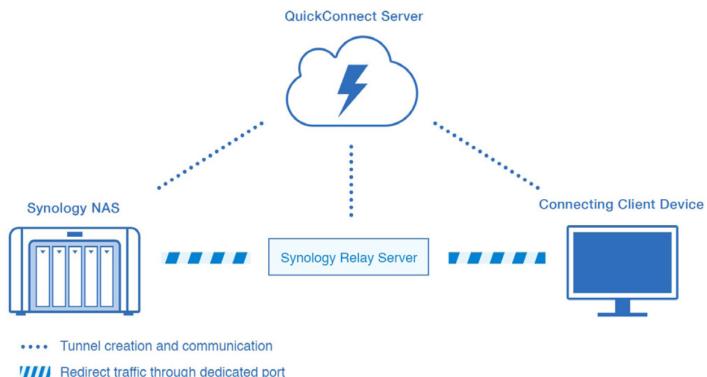


Abbildung 2.4: QuickConnect Relay Service [46, 6]

2.3 WebRTC Prototypen

2.3.1 aiortc Code-Basis

Um den MQTT-Tunnelclient Prototyp zu erstellen, haben wir als Code-Basis das Beispiel «datachannel-cli» aus dem das Python-Projekt «aiortc» verwendet [41]. Bei diesem wird ein WebRTC-Datankanal geöffnet. Anschliessend werden zyklisch Nachrichten mit einem Timestamp zwischen den beiden Seiten ausgetauscht. Für das Signaling kommt ein manuelles Kopieren und Einfügen der «Offer» bzw. der «Answer» zum Einsatz. Dies geschieht zu Testzwecken am einfachsten über zwei Terminals, in denen die Offer- und Answer-Anwendungen laufen. In Abbildung 2.5 ist der entsprechende Ablauf illustriert, wobei rot die Offer und blau die Answer markiert. In unserem Naming entspricht die «Offer» dem «Initiator» und die «Answer» dem «Listener».

```

[venv] /home/narusus@iwb002:/home/narusus/Documents/ScInformatik/Thesis/aiortc/examples/datamodel> $ python3 cli.py broker
[venv] /home/narusus@iwb002:/home/narusus/Documents/ScInformatik/Thesis/aiortc/examples/datamodel> $ python3 cli.py client

```

The sequence diagram illustrates the interaction between two terminals (red and blue) using the aiortc library. It shows the exchange of SDP offers and answers, followed by data message exchange.

- Terminal 1 (Red):**
 - Initializes a connection with `cli.py broker`.
 - Sends an offer (red box).
 - Receives an answer (blue box).
 - Exchanges data messages (red and blue boxes).
- Terminal 2 (Blue):**
 - Initializes a connection with `cli.py client`.
 - Sends an answer (blue box).
 - Receives an offer (red box).
 - Exchanges data messages (red and blue boxes).

Abbildung 2.5: aiortc datachannel-clı

2.3.2 Prototyp mit Signaling via Copy&Paste

Als «Proof of Concept» haben wir auf Basis des «datachannel-clı»-Beispiels einen Tunnel-Client erstellt, welcher den Ablauf gemäss dem Sequenzdiagramm in der Abbildung 2.6 implementiert. Hier existiert jeweils eine spezifische Listener-Seite des Tunnel-Clients und eine Initiator-Seite. Als Signaling wurde in dieser Phase die «Copy&Paste»-Methode verwendet.

Das Tunnellen der TCP-Nachrichten erfolgt ohne weitere Verarbeitung der Daten. Es werden daher die Bytes, welche über TCP empfangen werden, genau so über den Datenkanal auf die andere Seite gesendet und dort wieder über TCP weiter geleitet. Die Bestätigung, dass unser erarbeitetes Konzept funktioniert, erhielten wir überraschend schnell. Was die Implementationen vereinfachte, war die von aiortc verwendete asyncio-Bibliothek [47]. Mit asyncio können mit wenig Aufwand asynchrone Abläufe implementiert werden.

Dieser erste Wurf kann nur eine TCP-Verbindung tunneln. Dies geschieht, wie

im Sequenzdiagramm in Abbildung 2.6 hervorgehoben, über eine WebRTC Peer-Connection, welche einen WebRTC DataChannel beinhaltet.

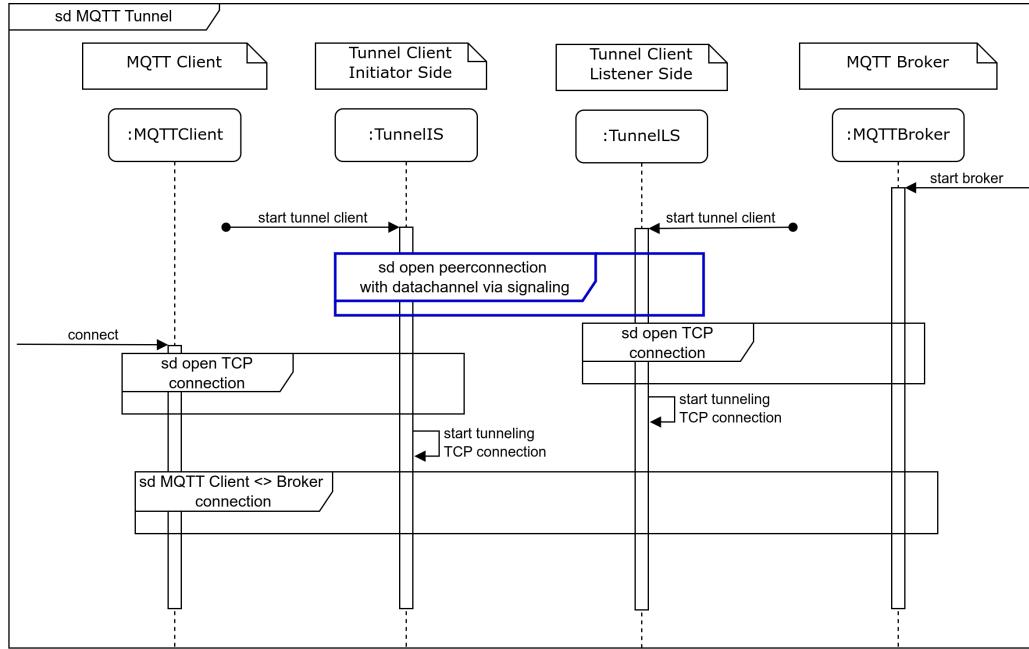


Abbildung 2.6: Sequenz MQTT-Tunnel

2.3.3 Signaling mit MQTT

Das Signaling als integrierte Lösung ist nicht in der WebRTC-Spezifikation beschrieben [48]. Daher existieren hier nur spezifische Lösungen, welche nicht allgemein verwendbar sind. Üblich sind zum Beispiel Lösungen mit einem Server, auf welchen sich die Clients via Websockets verbinden. Die Identifikation der Clients und das Weiterleiten der Signaling-Daten gehört dort zur Funktionalität des Servers. Wir wollten jedoch keine Serverapplikation entwickeln und entschieden uns daher für eine Lösung über MQTT. Bei dieser Lösung können übliche MQTT-Broker eingesetzt werden (z.B. Mosquitto, HiveMQ).

Auf die Dauer wurde das Testen mittels «Copy&Paste» umständlich. Daher haben wir eine erste einfache Variante eines MQTT-Signaling implementiert (Abbildung 2.7). Der Signaling-Instanz wird hierbei ein für die Verbindung spezifisches Topic übergeben. Zusätzlich müssen die Verbindungsangaben für den Signaling-Broker konfiguriert werden. Anschliessend werden die Offer und Answer Nachrichten zwischen den beiden Tunnelseiten ausgetauscht.

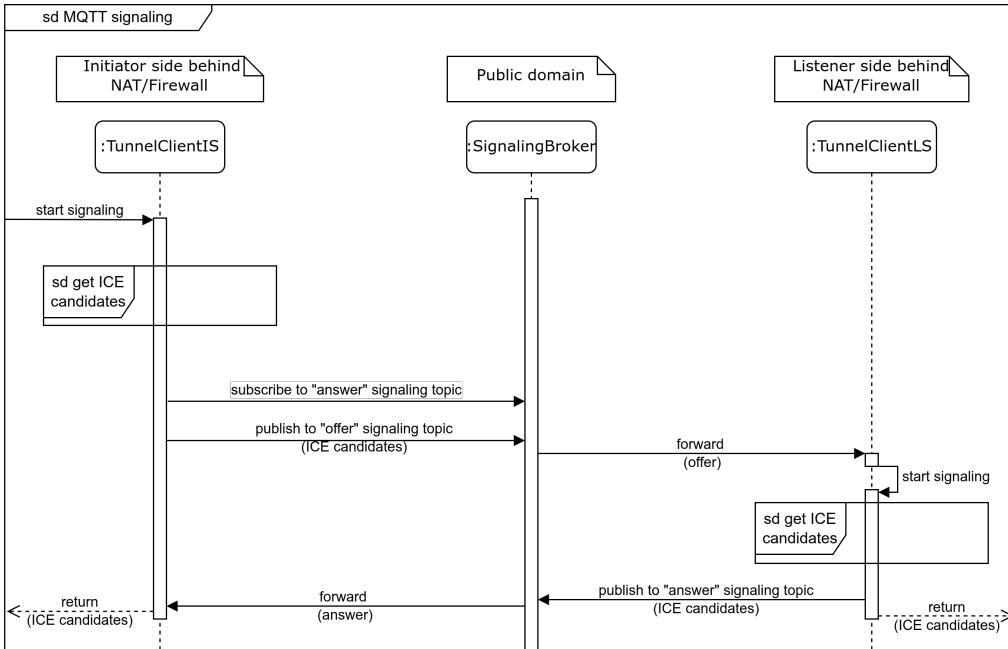


Abbildung 2.7: MQTT Signaling

2.3.4 Konzept Secure Signaling mit MQTT

Der Austausch der ICE-Kandidaten erfolgte bisher über den MQTT Broker unverschlüsselt. Ziel ist jedoch, eine Punkt-zu-Punkt-Verschlüsselung zu erreichen, damit prinzipiell auch bei unsicheren Brokern höchstens eine Denial-of-service-Attacke möglich ist. Das Signaling soll immer privat und authentisch erfolgen. Um dies zu erreichen, wird eine Public-Key-Verschlüsselung eingesetzt. Hierbei ist der Client-Seite der Public-Key der Brokerseite bekannt. Somit wird der Broker immer authentifiziert. Der öffentliche Schlüssel der Initiator-Seite wird jeweils von diesem publiziert. Optional kann bei der Listener-Seite definiert werden, dass nur bekannte Initiatoren eine Verbindung aufbauen können. Um Replay-Attacken zu verhindern, wird jeweils auch eine Nonce eingesetzt. Die Nonce wird von der Listener-Seite in fixen Abständen auf das Nonce-Topic publiziert. Die Initiator-Seite muss anschliessend die gültige Nonce der Listener-Seite in die Offer codieren. Neu wird von der Initiator-Seite eine eigene Nonce generiert und angefügt. Zusammen mit den ICE-Kandidaten werden die beiden Nonces verschlüsselt. Diese Nachricht wird auf das Offer-Topic publiziert, welches von der Listener-Seite abonniert wird. Zusätzlich wird der öffentliche Schlüssel der Initiator-Seite im Klartext an die Nachricht angefügt. Der Listener entschlüsselt anschliessend den verschlüsselten Teil und verifiziert die retournierte Nonce der Listener-Seite. Kann die Nachricht entschlüsselt werden und passt die retournierte Nonce, werden eigene ICE-Kandidaten ermittelt. Die ICE-Kandidaten und die übermittelte Nonce der Initiator-Seite werden in einer Nachricht verschlüsselt.

selt und auf das Answer-Topic publiziert, welches von der Initiator-Seite abonniert wird. Die Initiator-Seite entschlüsselt die Nachricht und verifiziert die retournierte Nonce. Falls alles passt, werden die übermittelten ICE-Kandidaten akzeptiert (Abbildung 2.8).

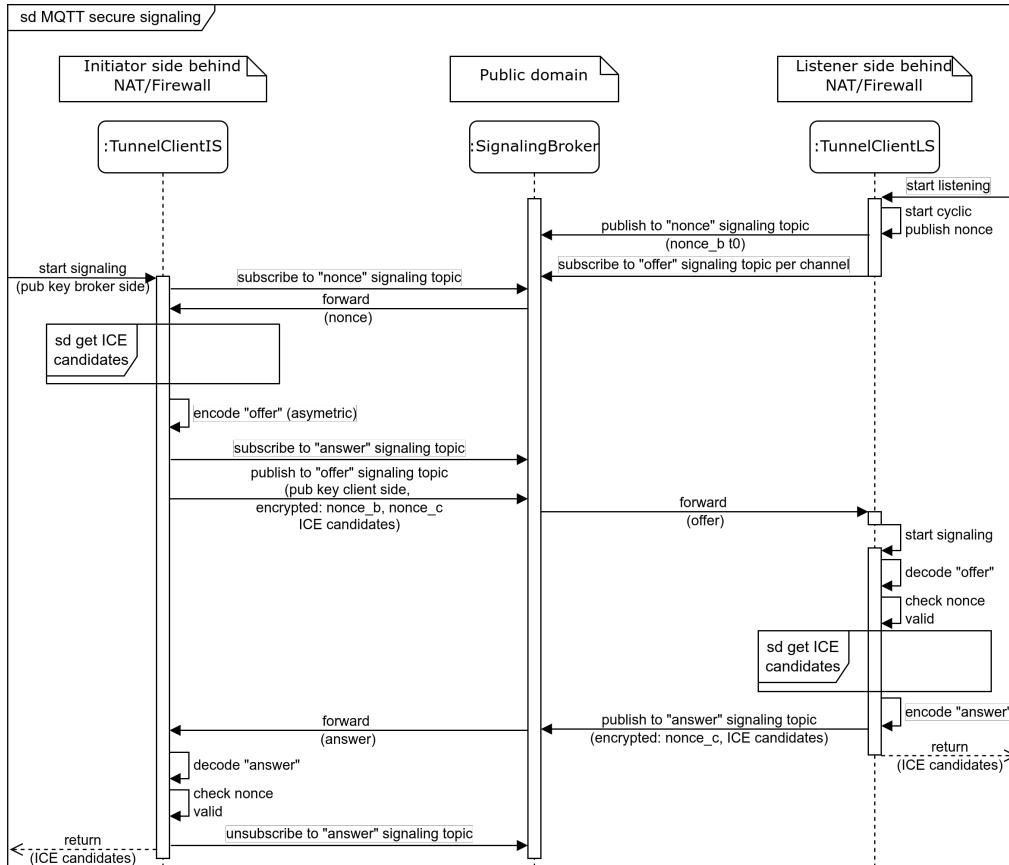


Abbildung 2.8: MQTT Secure Signaling

Nonce Topic

sig/<HashBrokerPubKey>/n

```

1 {
2   "nonce_b": "<nonce_b>"
3 }
  
```

Listing 2.1: Nonce Nachricht

Offer Topic

sig/<HashBrokerPubKey>/<HashClientPubKey>/<ChannelId>/o

```

1  {
2      "pub": "<public_key>",
3      "encrypted":
4      {
5          "nonce_b": "<nonce_b>",
6          "nonce_c": "<nonce_c>",
7          "ice":
8          {
9              "sdp": "<sdp_info>",
10             "type": "offer"
11         }
12     }
13 }
```

Listing 2.2: Offer Nachricht

Answer Topic

sig/<HashBrokerPubKey>/<HashClientPubKey>/<ChannelId>/a

```

1  {
2      "encrypted":
3      {
4          "nonce_c": "<nonce_c>",
5          "ice":
6          {
7              "sdp": "<sdp_info>",
8              "type": "answer"
9          }
10     }
11 }
```

Listing 2.3: Answer Nachricht

2.3.5 Umsetzung Secure Signaling mit MQTT

Nach den ersten Gehversuchen mit der Implementation einer Public-Key-Verschlüsselung mittels der Bibliothek PyNaCl stellte sich heraus, dass die Nonce bereits durch die Bibliothek der Nachricht angefügt wird. Aus diesem Grund ist die Integration einer Nonce von unserer Seite hinfällig. Einzig der Public-Key vom Initiator muss noch übermittelt werden.

Beim vorgängig erwähnten Konzept besteht bei den Public-Keys zudem eine gewisse Redundanz. Zum einen wird der Key in der Offer-Nachricht eingefügt (Listing 2.2), zum anderen ist dieser relevant für das Offer-Topic, da dort der Hash vorhanden ist. Eine simplere Lösung ist, an Stelle des Hashes direkt den Public-Key ins Topic zu integrieren.

Diese Anpassungen führten dazu, dass die Unterschiede zum Sequenzdiagramm des unverschlüsselten Signaling minimal ausfallen (Abbildung 2.9).

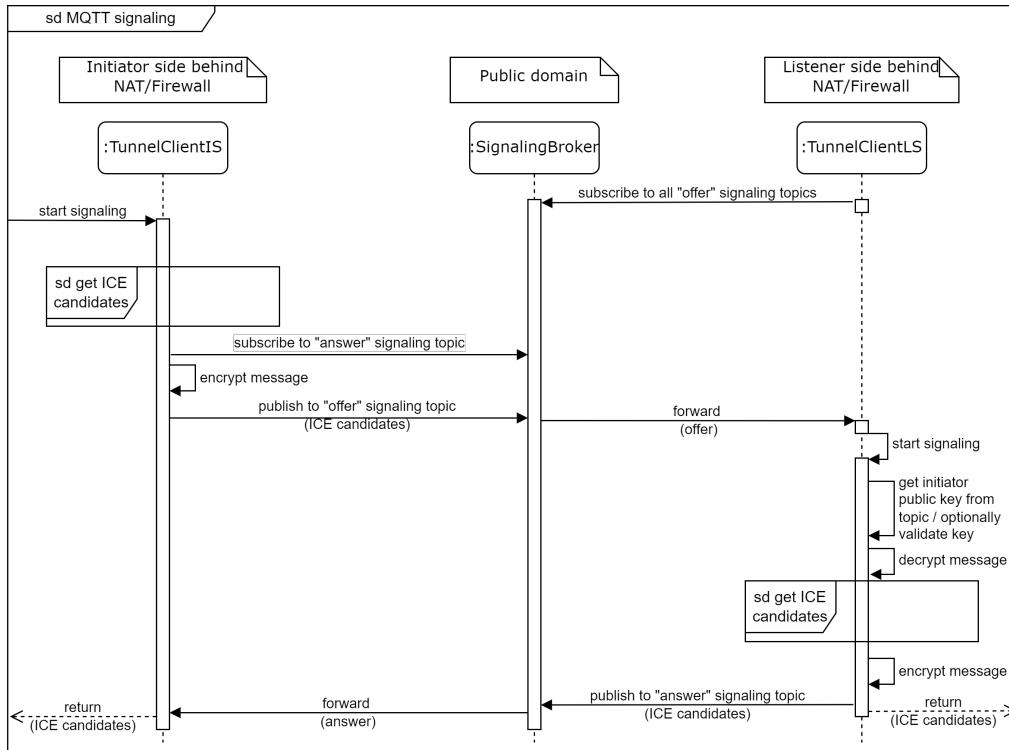


Abbildung 2.9: MQTT secure signaling

Eine offene Frage war anschliessend, in welchem Format der Public-Key im Topic integriert werden soll. Ein erster Gedanke war die Verwendung der Base64-Codierung [49]. Diese musste jedoch schnell verworfen werden, da im Base64-Zeichensatz die MQTT-Sonderzeichen «/» und «+» vorhanden sind. Dadurch kann ein in Base64 kodierter Schlüssel zu fehlerhaftem Subscriben und Publishen führen. Als naheliegende Alternative haben wir uns für die Base32-Variante entschieden, da hier keine MQTT-Sonderzeichen vorkommen [26].

Offer Topic mit Public-Key

`sig/<BrokerPubKeyBase32>/<ClientPubKeyBase32>/<ChannelId>/o`

Answer Topic

sig/<BrokerPubKeyBase32>/<ClientPubKeyBase32>/<ChannelId>/a

Mit einem MQTT-Explorer-Client lassen sich die Signaling-Nachrichten und Topics einfach visualisieren (Abbildung 2.10).

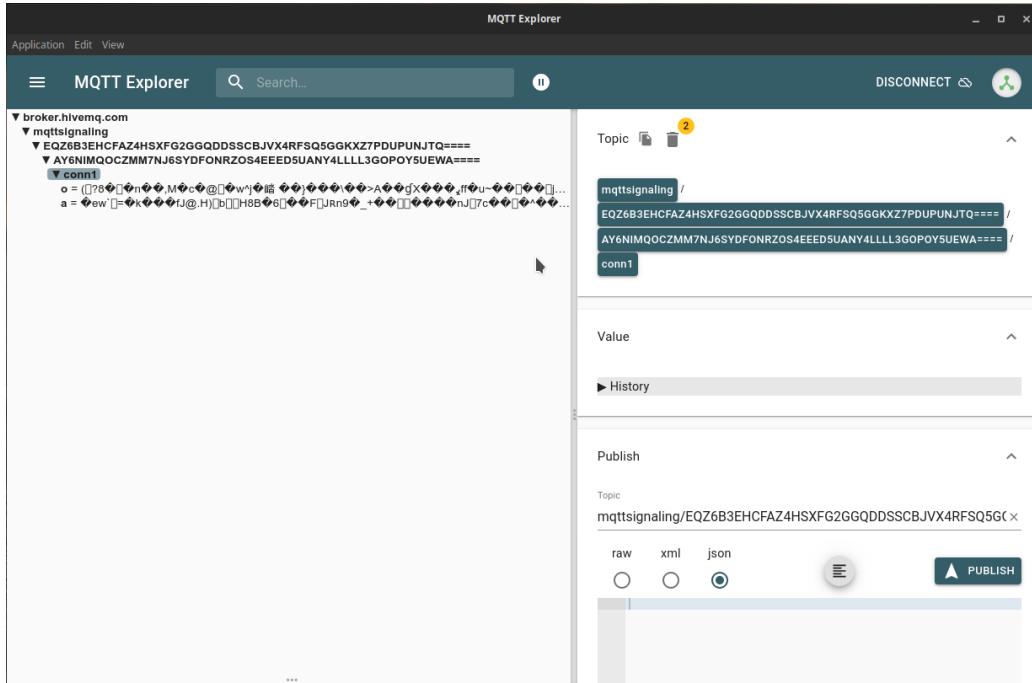


Abbildung 2.10: MQTT-Explorer: Secure signaling

Um die Signaling-Messages via MQTT-Explorer im Klartext zu visualisieren, ist in der Signaling-Klasse ein unsicherer Modus implementiert. Bei diesem werden die Nachrichten nicht verschlüsselt. Diese Funktionen wird bei einer finalen Version natürlich entfernt. Dieser Modus muss zwingend sowohl beim Initiator wie auch beim Listener aktiviert sein, damit dieser funktioniert.

2.3.6 Multichannel

Bei den ersten Implementationen des WebRTC-Tunnels konnte jeweils nur eine TCP-Verbindung getunnelt werden. Grundsätzlich ist es jedoch möglich mehrere TCP-Verbindungen als RTC-Datenkanäle über eine RTC-PeerConnection zu führen. Ein mögliches Einsatzszenario hierfür ist in der Abbildung 2.11 dargestellt. Bei diesem sind vier MQTT-Actors mit dem Broker im LAN A verbunden. Die zwei Actors 1 und 2 aus dem LAN B verwenden hierfür den WebRTC-Tunnel. Mit der Multichannel-Fähigkeit ist für dieses Szenario nur ein Tunnel nötig.

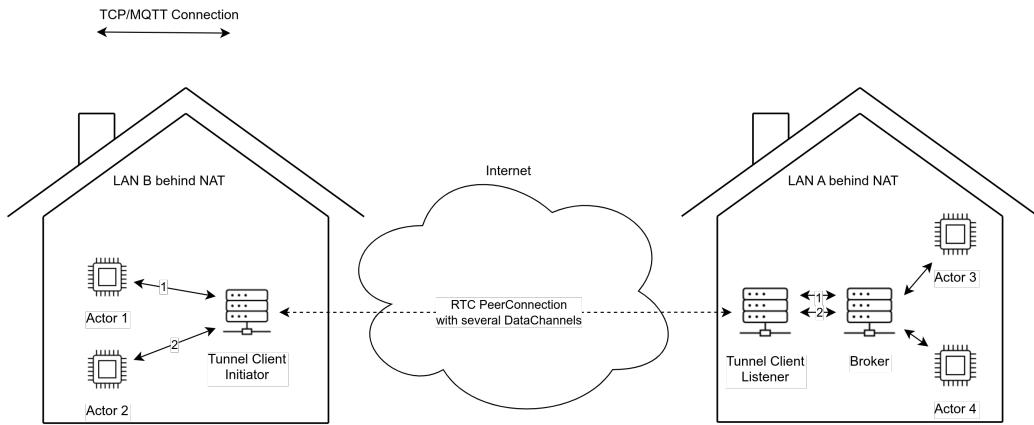


Abbildung 2.11: Szenario MQTT-Tunnel mit Multichannel

Das Sequenzdiagramm in der Abbildung 2.12 zeigt den grundsätzlichen Ablauf einer Multichannel-PeerConnection. Initial wird eine PeerConnection aufgebaut, jedoch noch kein DataChannel. Sobald sich auf der Initiator-Seite ein Client via TCP verbindet, wird ein neuer DataChannel aufgebaut. Auf der Listener-Seite wird eine neue TCP-Verbindung zum Broker erstellt, sobald ein neuer DataChannel etabliert wurde. Dieser Ablauf lässt sich nun beliebig oft wiederholen. Die Implementation dieses Multichannel-Tunnels stellte uns vor zwei grundsätzliche Probleme. Zum einen lässt sich keine PeerConnection ohne Daten- oder Medienkanal erstellen. Unsere pragmatische Lösung ist, bei der PeerConnection einen DataChannel zu erstellen, welcher nicht verwendet wird. Dieser könnte in einer späteren Phase auch als Administrationskanal verwendet werden, über welchen der Initiator und der Listener miteinander kommunizieren. Die zweite Herausforderung war, dass die ersten Nachrichten über den neuen Datenkanal gesendet werden, obwohl auf der Listener-Seite die TCP-Verbindung zum Broker noch nicht etabliert wurde. Bei unserer Lösung puffert die Listener-Seite alle Nachrichten, bis die TCP-Verbindung steht.

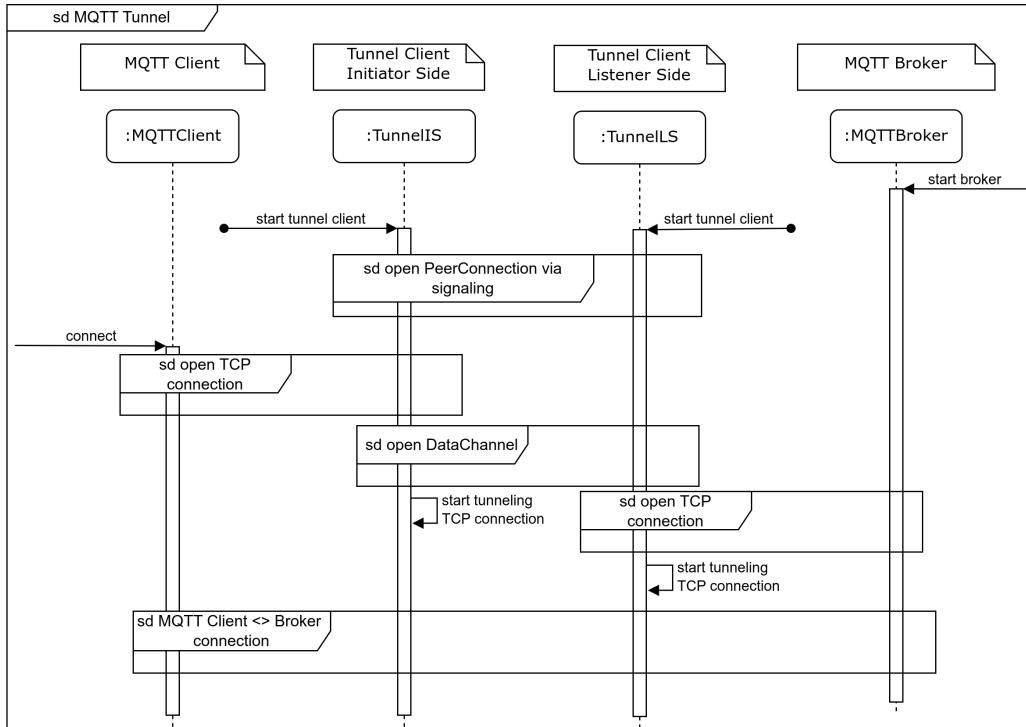


Abbildung 2.12: Sequenz MQTT-Tunnel mit Multichannel

2.3.7 Multiinitiator

Als Demonstrationsaufbau möchten wir einen Broker hinter einem NAT und einer Firewall bereitstellen, welcher über einen Listener erreichbar ist. Auf diesen können sich Testpersonen beliebig mit ihren MQTT-Clients via Initiator verbinden und unsere Lösung testen. Um dies möglich zu machen, muss ein Listener mit mehreren Initiatoren gleichzeitig umgehen können. Das entsprechende generalisierte Szenario ist in der Abbildung 2.13 dargestellt. In diesem läuft der Broker im LAN A. Die beiden Akteure 5 und 6 sind im LAN A und können sich somit auch direkt mit dem Broker verbinden. Die Akteure 1 bis 4 befinden sich jedoch in anderen Netzwerken und müssen sich jeweils via einem spezifischen lokalen Initiator mit dem Broker verbinden.

Im grundsätzlichen Ablauf ändert sich gegenüber dem Sequenzdiagramm in der Abbildung 2.12 nichts. Die Erstellung der PeerConnection wiederholt sich jeweils für jeden neuen Initiator. Die Signaling-Topics sind durch die Struktur mit den Public-Keys bereits eindeutig getrennt.

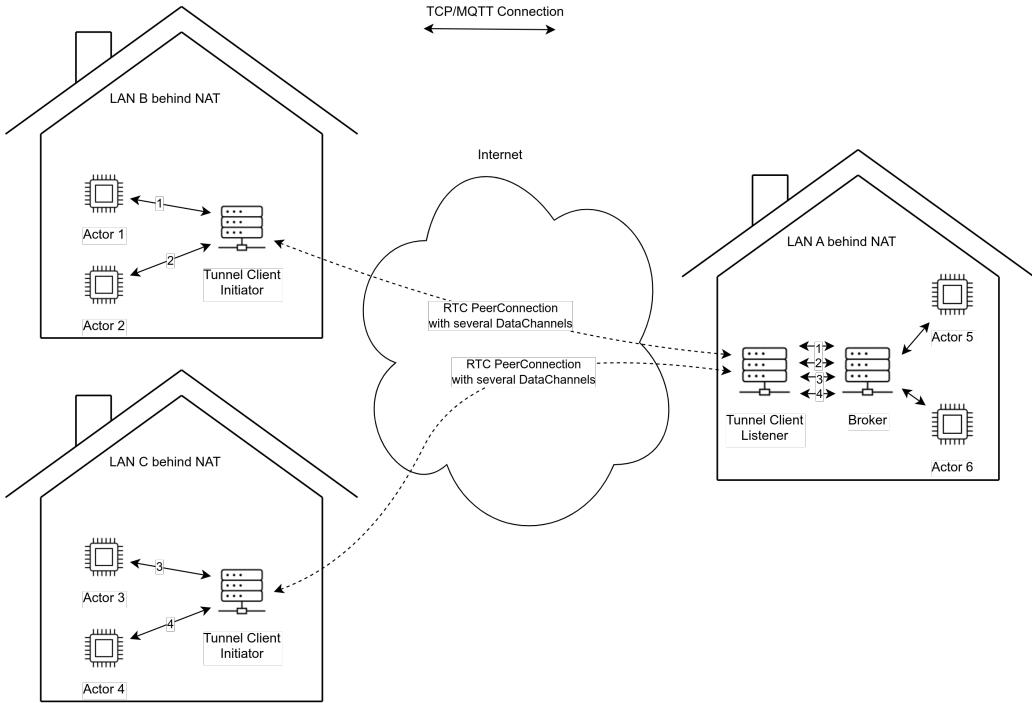


Abbildung 2.13: Szenario MQTT-Tunnel mit mehreren Initiator

2.4 Testumgebung

2.4.1 Virtual Machine BFH als Server

Für Tests, bei denen ein öffentlicher Server erforderlich ist, stellt die BFH uns virtuelle Maschinen (VM) zur Verfügung. Diese VM sind jeweils über eine öffentliche Adresse erreichbar. Unsere Testinstanzen laufen auf einer VM mit dem Server-Betriebssystem Ubuntu 18.04.6 LTS. Weiter läuft für die Tests eine Instanz eines Mosquitto-MQTT-Brokers, welche uns primär als Signalling-Broker dient.

2.4.2 SSL Zertifikat für VM BFH

Damit die Verbindungen zum VM der BFH mit TLS abgesichert werden können, haben wir ein Zertifikat durch zeroSSL.com erstellen lassen. Ursprünglich wollten wir das Zertifikat durch den verbreiteten Anbieter «Let's encrypt» erstellen lassen [50]. Leider ist dies dort nur mittels Domainname möglich [51]. Unsere VM ist jedoch nur über die IP-Adresse erreichbar. Nach einer Recherche sind wir auf den Anbieter «zeroSSL» gestossen, welcher ebenfalls kostenfreie Zertifikate ausstellt (90-Tage gültig) [52]. Für Server, welche nur über eine IP erreichbar sind,

steht nur das Validierungsverfahren mittels Webserver zur Verfügung. Hierbei muss für die initiale Validierung eine Verifikationsdatei auf dem Server publiziert werden, welche anschliessend von zeroSSL geprüft wird. Für diese Aufgabe haben wir auf der VM der BFH eine Apache-Webserverinstanz konfiguriert, welche eine entsprechende Webseite bereitstellen kann. Wir können somit die Verbindungen auf unseren Server mit einem gültigen Zertifikat absichern.

2.4.3 Inbetriebnahme Prototyp mit aiortc auf VM BFH

Damit aiortc lauffähig ist, muss zum einen das aiortc Paket über den Python-Paketmanager installiert werden, zum anderen müssen Bibliotheken für die Verschlüsselung (OpenSSL) und die Video- und Audio-Verarbeitung (z.B. FFmpeg) installiert sein. Für die Installation dieser Bibliotheken existiert auf der github-Seite von aiortc eine Anleitung [41]. Auf unserer VM der BFH war diese Installation aufwändiger, da auf dem vorhandenen Betriebssystem Ubuntu 18.04.6 LTS FFmpeg Version 3 in der Paketverwaltung vorhanden war. Gemäss der github-Seite ist jedoch mindestens Version 4 nötig. Zur Zeit der Arbeit war die Version 5 von FFmpeg die Neuste, daher haben wir die entsprechende Paketquelle hinzugefügt. Leider stellte sich heraus, dass aiortc zu diesem Zeitpunkt nicht mit der Version 5 von FFmpeg kompatibel war. Daher musste zwingend die Version 4 installiert werden.

Hier zeigt sich ein Schwachpunkt der Umsetzung mit WebRTC resp. aiortc: Die Funktionalität von WebRTC ist mit der komplexen Audio- und Videoübertragungsmöglichkeiten weitaus grösser, als für unseren Applikations- bzw. MQTT-Tunnel nötig ist. Dies kann, wie oben beschrieben, zu Problemen führen, welche durch für uns unnötige Komponenten verursacht werden.

2.4.4 Erstellung Flatpak Applikation

Als Reaktion auf die Probleme mit den FFmpeg-Versionen haben wir uns entschieden, eine Flatpak-Applikation zu bauen. Flatpak ist eine Lösung zur Softwareverteilung auf Linux-Betriebssystemen und ist auf allen gängigen Distributionen in den Paketquellen vorhanden. Um unabhängig von den im Betriebssystem vorhandenen Bibliotheken zu sein, werden bei jeder Flatpak-Applikation alle nötigen Abhängigkeiten in eine Sandbox eingepackt. Ein Nachteil dieses Vorgehens ist der erhöhte Speicherbedarf für die Installationen. Die Erwartung unsererseits ist, dass wir so unsere Applikation einfacher für erste Tests ausliefern können.

Die Verteilung von Flatpak-Applikationen erfolgt in der Regel über Webseiten, welche als Paketquellen eingebunden werden können. Die wohl bekannteste Paketquelle für Flappak-Applikationen ist flathub.org. Im Rahmen dieses Projekts, wollten wir die Pakete möglichst einfach bereitstellen. Nach einer Recherche haben wir uns entschieden, unsere Anwendung über gitLab Pages [53] bereitzustel-

len. Dies funktioniert, da die Flatpak-Daten lediglich auf einer statischen Webseite bereitgestellt werden müssen [54].

Leider hatten wir mit dem BFH gitLab-Server Probleme, da die Zertifikatsvalidierung durch die Flatpak-Applikation fehlschlug. Das exakte Problem hierbei konnten wir trotz Recherchen nicht ermitteln. Da die Verteilung der Applikation nicht unser Hauptfokus ist und die Lösung via gitLab-Pages nur temporär zum Einsatz kommen soll, haben wir als Alternative die Pages von GitHub verwendet (<https://salvm4.github.io/>). Die ausgearbeitete Applikation würden wir zum Beispiel auf flathub.com bereitstellen.

Die Flatpak-Applikation wird mittels einem Manifest definiert. In diesem Manifest sind die Basisumgebung, wie auch die Sourcedateien und die Kompilierstruktionen definiert (Listing 2.4). Ein wichtiges Detail ist «*finish-args*» mit der Option «*share=network*». Damit wird der Flatpak-Sandbox der Applikation der Zugriff auf das Netzwerk vom Host-System gewährt. Wie aus dem Manifest ersichtlich ist, benötigt unsere Applikation keine weiteren Zugriffe außerhalb der Sandbox.

```

1 app-id: ch.bfh.ti.applic-tunnel
2 runtime: org.freedesktop.Platform
3 runtime-version: '21.08' # do not use newer version, ffmpeg version 5 in 22.08 is
4 # not compatible with aiortc at the moment
5 sdk: org.freedesktop.Sdk
6 sdk-extensions:
7   - org.freedesktop.Sdk.Extension.rust-stable
8 command: runner.sh
9 finish-args:
10  # add network permission to sandbox
11  - --share=network
12 build-options:
13  build-args:
14    # allow network access during build process
15    - --share=network
16  append-path:
17    /usr/lib/sdk/rust-stable/bin # export rust complier path
18 modules:
19  - name: crc32c # Compile crc32c manual
20    buildsystem: cmake-ninja
21    sources:
22      - type: git
23        branch: main
24        url: https://github.com/google/crc32c.git
25  - name: applic-tunnel
26    buildsystem: simple
27    build-commands:
28      # show available versions
29      - rustc --version # log rust compiler version
30      - cargo --version # log cargo version
31      - ffmpeg -version # log ffmpeg version
32      - openssl version # log open ssl version
33      # install custom scripts

```

```

33   - install -D runner.sh /app/bin/runner.sh
34   - install -D applic_tunnel_main.py /app/applic_tunnel_main.py
35   - install -D applic_tunnel.py /app/applic_tunnel.py
36   - install -D mqtt_secure_signaling.py /app/mqtt_secure_signaling.py
37   - install -D applic_tunnel_utilities.py /app/applic_tunnel_utilities.py
38   - install -D signaling_interface.py /app/signaling_interface.py
39 build-options:
40 build-args:
41     # allow network access during build process
42     - --share=network
43 sources:
44     # add local sources
45     - type: file
46         path: runner.sh
47     - type: file
48         path: ../src/applic_tunnel_main.py
49     - type: file
50         path: ../src/applic_tunnel.py
51     - type: file
52         path: ../src/mqtt_secure_signaling.py
53     - type: file
54         path: ../src/applic_tunnel_utilities.py
55     - type: file
56         path: ../signaling_interface.py
57 # add python libraries from automatic generated file
58     - python3-requirements.json

```

Listing 2.4: Flatpak-Manifest

Eine weitere Herausforderung von Flatpak war, eine Version für unseren Testbroker installiert auf einem Raspberry Pi 4, zu paketieren. Da der Raspberry Pi einen ARM Prozessor verbaut hat, muss die Applikation spezifisch für diese Architektur erstellt werden. Für das Cross-Compiling auf einer x64 Maschine haben wir uns an verschiedene Anleitungen gehalten, wobei diese nicht sehr spezifisch sind [55] [56]. Prinzipiell müssen gemäss Anleitung die Virtualisierungssoftware QEMU [57] installiert und die entsprechenden fremden Binary-Formate registriert sein, damit diese über QEMU aufgerufen werden können. Unter Ubuntu können die nötigen Pakete gemäss dem Listing 2.5 installiert werden.

```

1 # install qemu and binfmt
2 sudo apt-get install qemu-system-arm qemu-user-static binfmt-support
3 # restart binfmt service
4 sudo systemctl restart systemd-binfmt.service

```

Listing 2.5: Ubuntu ARM cross building dependencies

Das Erstellen der Flatpak-Applikation erfolgt anschliessend durch die Spezifikation der Zielarchitektur (Listing 2.6). Leider haben wir das Cross-Compiling nicht auf Anhieb erfolgreich konfigurieren können. Das Kompilieren vom Rust-Code für die python3-cryptography Bibliothek scheiterte jeweils. Nach einigen

Recherchen und Analysen hat sich herausgestellt, dass das Problem nicht an der Rust-Installation oder dem Flatpak-Manifest lag, sondern schlicht an zu wenig Arbeitsspeicher. Das Cross-Complining des Flatpaks erfordert rund 25GB an Arbeitsspeicher. Falls zu wenig physikalischer Speicher vorhanden ist, kann dieser mit einer Swap-Partition erweitert werden.

```
1 flatpak-builder --arch=aarch64 --repo=path/to/repo --force-clean build
  ch.bfh.ti.applic-tunnel.yml
```

Listing 2.6: flatpak-builder cross compile

2.5 Performanz-Tests Bridge vs. Tunnel

In einem Test-Setup haben wir untersucht, wie sich die Performanz zwischen dem Tunneln über den WebRTC Datenkanal und dem «herkömmlichen» Weg über einen öffentlichen Broker unterscheidet. Die Messung geschieht über das Ermitteln der Paketumlaufzeit (*Round Trip Time*). Das Prinzip ist in Abbildung 2.14 dargestellt. Ein MQTT Client (*RTT Analyzer Client*) veröffentlicht eine Nachricht mit der aktuellen Zeit (*timestamp*) als payload auf ein spezifisches Topic, im Beispiel «*tunnelBridge/RTTRelay*». Ein zweiter Client (*RTT Mirror Client*) hat dieses Topic abonniert und veröffentlicht jede erhaltene Nachricht auf ein anderes Topic, in unserem Beispiel «*tunnelBridge/RTTEvaluation*». Dieses Topic wird wiederum vom Analyzer Client abonniert. Er vergleicht die Timestamp im Payload der erhaltenen Nachrichten mit der aktuellen Zeit und erhält so die Round Trip Time.

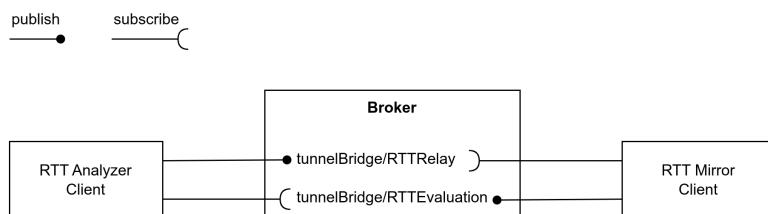


Abbildung 2.14: Prinzip Messung Round Trip Time

Abbildungen 2.15 und 2.16 zeigen den Versuchsaufbau, einmal mit dem Weg über einen Relay-Broker im Internet und einmal bei direktem Tunneling. Die Variante mit dem Relay-Broker dient als Referenz-Messung. Die Zeiten dieser Messungen werden mit den Zeiten der Tunnel Variante verglichen um festzustellen, ob diese im Vergleich zum Weg über einen öffentlichen Broker tatsächlich schneller ist.

Die Implementierung haben wir nach dem «Advanced use»-Beispiel von *asyncio-mqtt* vorgenommen [58]. Sowohl der Analyzer als auch der Mirror Client sind als

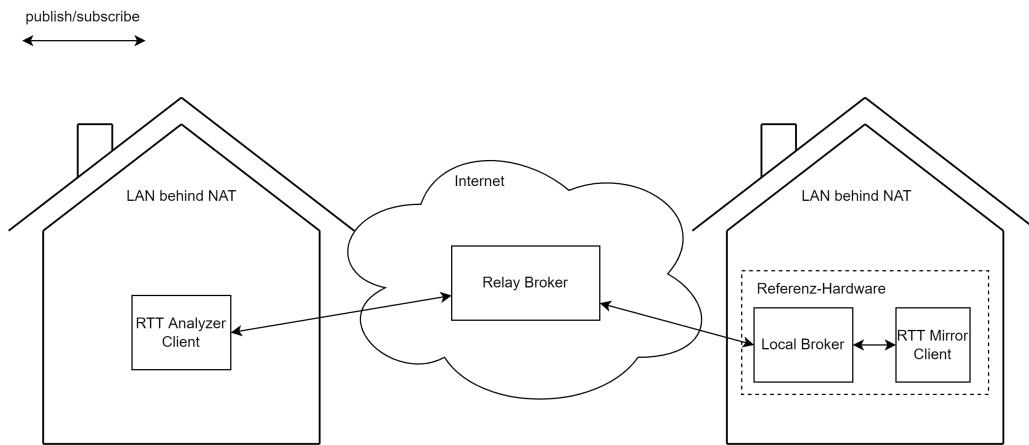


Abbildung 2.15: RTT Messung mit Broker

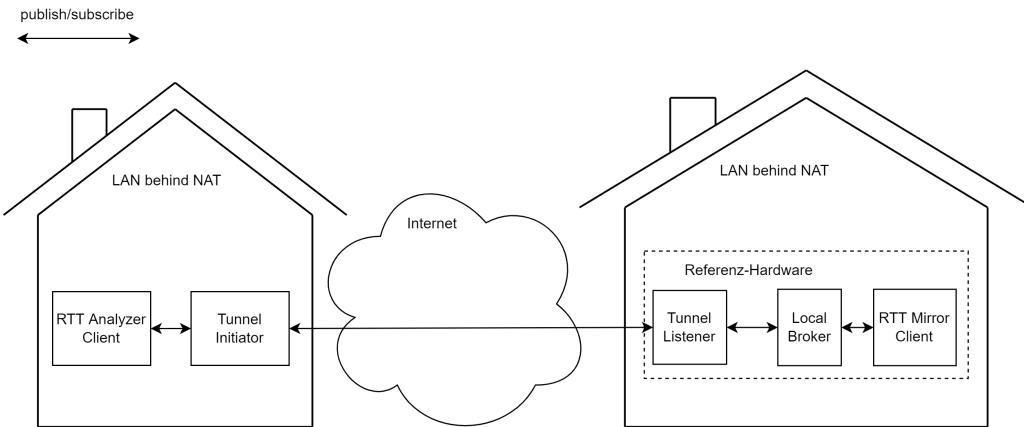


Abbildung 2.16: RTT Messung mit Tunnel

flatpak-Applikationen verfügbar. Sie können, wie im Listing 2.7 gezeigt, installiert werden.

```
1 flatpak --user install flatpak-salvm4 ch.bfh.ti.rtt-analyzer
2 flatpak --user install flatpak-salvm4 ch.bfh.ti.rtt-mirror
```

Listing 2.7: RTT Analyzer Tools Installation

Beim Aufruf über die Kommandozeile können den Clients neben broker-spezifischen Angaben auch die Topics angegeben werden, auf welche subscribed bzw. gepublished werden soll. Das Listing 2.8 zeigt die Aufrufe für das in Abbildung 2.14 gezeigte Beispiel.

```

1 flatpak run ch.bfh.ti.rtt-analyzer test.mosquitto.org 1883 tunnelBridge/RTTRelay
   tunnelBridge/RTTEvaluation --notls
2 flatpak run ch.bfh.ti.rtt-mirror test.mosquitto.org 1883 tunnelBridge/RTTEvaluation
   tunnelBridge/RTTRelay --notls

```

Listing 2.8: RTT Analyzer Tools Programmaufrufe

Die help-Funktion für den Analyzer Client ist im Listing 2.9 gezeigt. Für den Mirror Client ist der Aufruf beinahe identisch, bei ihm fehlt lediglich die Intervall-Option.

```

1 flatpak run ch.bfh.ti.rtt-analyzer -h
2 usage: RTT Analyzer Client [-h] [-u USER] [-p PASSWORD] [--message_interval
   MESSAGE_INTERVAL] [--notls] [-v] [-l] brokername brokerport s_topic p_topic
3
4 Round Trip Time evaluation analyzer client
5
6 positional arguments:
7   brokername           hostname or ip of broker
8   brokerport          port on broker
9   s_topic              topic to subscribe to
10  p_topic              topic to publish to
11
12 options:
13   -h, --help            show this help message and exit
14   -u USER, --user USER  broker user name
15   -p PASSWORD, --password PASSWORD
16                   broker password
17   --message_interval MESSAGE_INTERVAL
18                   interval messages are sent (in seconds), default is 1 second
19   --notls               do not use TLS
20   -v, --verbose         enable log to file
21

```

Listing 2.9: RTT Analyzer Tools help-Funktion Ausgabe

2.5.1 Durchführung

Die Referenz Tests mit dem Relay-Broker (siehe Abbildung 2.15) wurden mit folgendem Setup durchgeführt:

- ▶ **RTT Analyzer Client** Der Client, der die Nachrichten sendet und die RTT berechnet, läuft auf einem Linux Notebook im Heimnetz von Remo Meyer in Kiesen.
- ▶ **Relay Broker** Die Tests werden mit drei verschiedenen Brokern durchgeführt: test.mosquitto.org, broker.hivemq.com und mosquitto auf einer BFH VM unter 147.87.118.29.
- ▶ **Local Broker** Auf einem zweiten Linux Notebook, welches sich im Heimnetz von Markus Salvisberg in Bern befindet, ist mosquitto installiert und

eine Bridge eingerichtet. Für die Bridge-Konfigurationen werden im config-file die Einträge gemäss Listing 2.10 gemacht.

- **RTT Mirror Client** Auf dem selben Gerät wie der Local Broker läuft auch der Client, welcher die Nachrichten zurücksendet.

```

1 # connection name
2 connection rtt-test
3
4 # addresses
5 address test.mosquitto.org:1883
6 #address broker.hivemq.com
7 #address 147.87.118.29
8
9 # topics
10 topic tunnelBridge/RTTRelay in 0
11 topic tunnelBridge/RTTEvaluation out 0
12
13 #--- specific settings for HiveMQ ---
14 #bridge_protocol_version mqttv311
15 #try_private false
16 #notifications false
17 #bridge_attempt_unsubscribe false
18 #bridge_insecure true
19
20 #--- specific settings for mosquitto on BFH VM ---
21 #remote_username cedalo
22 #remote_password test

```

Listing 2.10: Bridge Konfiguration RTT für Tests

Die Tests mit dem Tunnel-Client wurden mit dem selben Setup durchgeführt.

2.5.2 Resultate

Die Ergebnisse der Messungen sind in der Tabelle 2.1 und in den Grafiken in den Abbildungen 2.17 und 2.18 ersichtlich. Alle gesammelten Daten sind in unserem Git-Repository abgelegt.

	Mittelwert	Max	Min
HiveMQ	0.135 915 799 s	0.640 798 33 s	0.050 804 853 s
Mosquitto	0.141 252 469 s	0.388 126 612 s	0.050 492 287 s
BFH VM	0.088 839 183 s	0.210 141 659 s	0.037 417 889 s
Tunnel	0.035 124 567 s	0.213 764 191 s	0.028 029 203 s

Tabelle 2.1: Round Trip Time Messungen Resultate

Die Resultate zeigen, dass der Weg durch den Tunnel deutlich schneller ist als über die Relay-Broker. Durch den Tunnel wurden Nachrichten im Schnitt fast

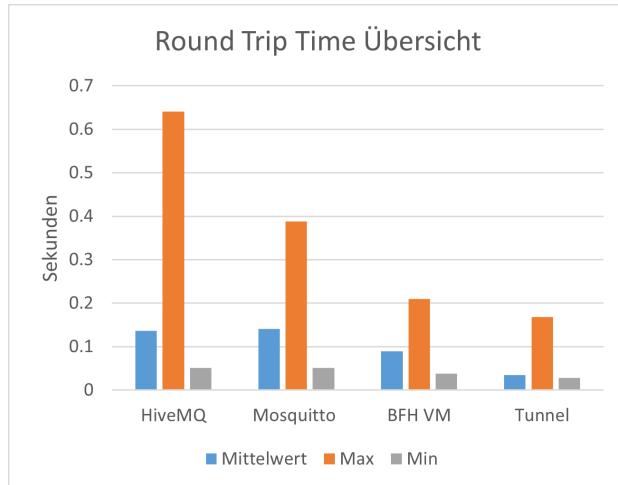


Abbildung 2.17: Round Trip Time Messungen Resultate
Übersicht

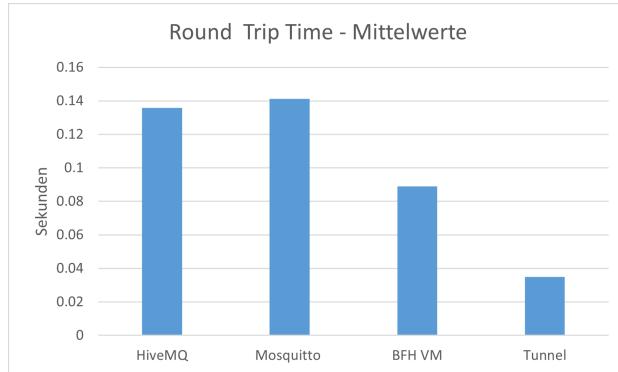


Abbildung 2.18: Round Trip Time Messungen Vergleich
Mittelwerte

4-mal schneller gesendet als über die Broker von HiveMQ oder mosquitto und 2.5-mal schneller als über einen selbst gehosteten Broker mit wenig Netzwerk-Traffic. Neben der durchschnittlichen Round Trip Time sind auch die Minimum- und Maximum-Werte deutlich besser als bei den Relay-Brokern. Dass es kaum Ausreisser nach oben gab zeigt, dass der Tunnel die Nachrichten konstant schnell übermittelt. Damit die Messungen noch aussagekräftiger wären, müsste man weitere und vor allem längere Testreihen durchführen. Interessant wäre auch zu sehen, wie sich der Tunnel in einem Belastungstest mit einer sehr grossen Nachrichtenlast schlagen würde.

Grundsätzlich sind die Resultate der Tests für uns sehr erfreulich. Sie zeigen, dass der Tunnel nicht nur funktioniert und einen Nutzen bezüglich Sicherheit bietet, sondern, dass er in Sachen Geschwindigkeit Vorteile bringt.

2.6 Secure Bridge

Falls keine direkt Verbindung zwischen zwei Tunnel-Clients mittels WebRTC Datenkanal möglich ist, haben wir als Fallback den Weg über einen im Internet erreichbaren MQTT-Broker vorgesehen. Wenn man diesen nicht selbst betreiben will oder kann, bietet es sich an, einen der zahlreichen öffentlich zugänglichen Broker zu verwenden. Weitere solche Broker findet man auf der github Seite von mqtt.org [59]. Falls der Weg über einen nicht selbst betriebenen und somit nicht vertrauenswürdigen Broker führt, besteht potenziell die Gefahr einer Verletzung der Vertraulichkeit. MQTT bietet zwar standardmäßig die Verbindungsicherung mittels TLS an, was die Verbindung zwischen Client und Broker absichert (siehe Kapitel 1.3.4), auf dem Broker bzw. dem Broker-Server sind die Nutzdaten jedoch weiterhin im Klartext vorhanden. Daher kann hier eine zusätzliche Verschlüsselung der Nutzdaten sinnvoll sein. Nachfolgend sind zwei Szenarien beschrieben, wie eine Nutzdatenverschlüsselung umgesetzt werden kann.

- ▶ **End-to-End Verschlüsselung** Bei der End-to-End Verschlüsselung erfolgt die Absicherung der Nutzdaten bereits bei den Actors. Jeder Actor muss daher in der Lage sein, Nachrichten zu ver- und entschlüsseln (Abbildung 2.19). Dadurch sind die Nachrichten selbst dann gegenüber Mitlesen und Manipulation geschützt, falls der Broker kompromittiert ist.

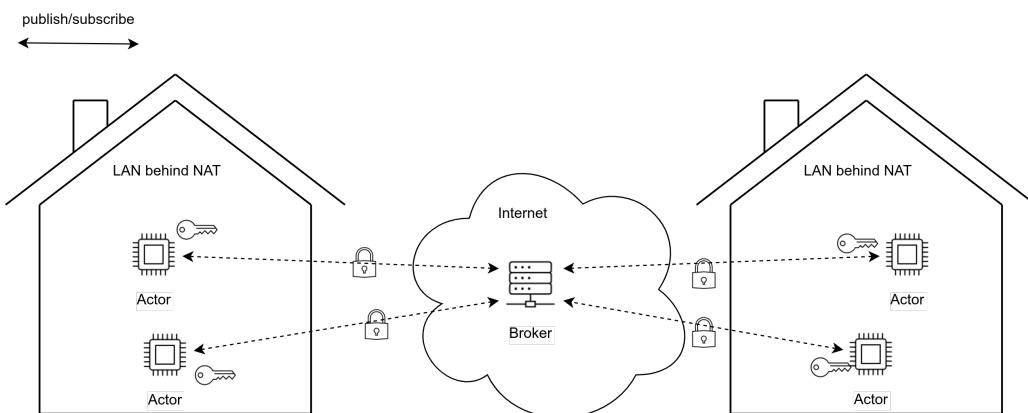


Abbildung 2.19: End-to-End Verschlüsselung

- ▶ **Nutzdatenverschlüsselung auf Bridge-Ebene** Falls die Actors nicht in der Lage sind, die Nutzdaten zu verschlüsseln oder der Schlüsselaustausch problematisch ist, kann die Verschlüsselung auch auf Bridge-Ebene erfolgen. Die Nutzdaten werden auf den lokalen Brokern verschlüsselt und via Bridge-Funktion auf einen nicht vertrauenswürdigen Broker publiziert (Abbildung 2.20).

Im Fall von Nutzdatenverschlüsselung auf Bridge Ebene kann zusätzlich auch

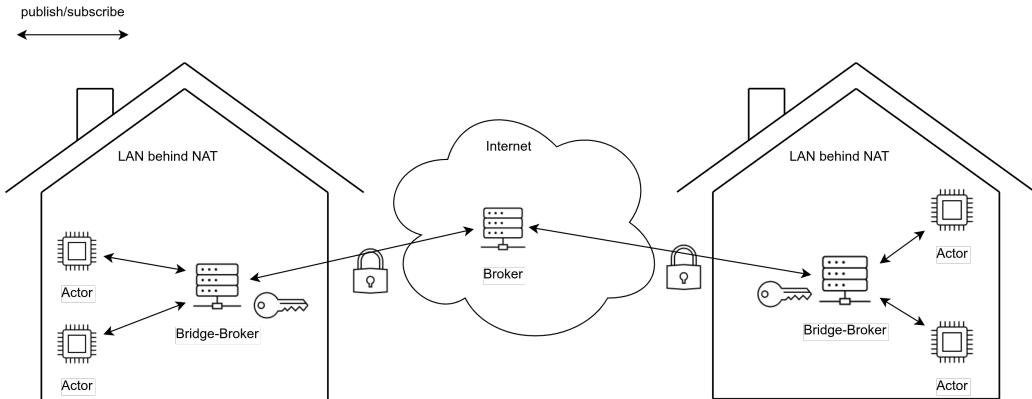


Abbildung 2.20: Nutzdatenverschlüsselung auf Bridge-Ebene

die Topic-Struktur verschleiert werden, indem die Broker diese in die Nutzdaten codieren und auf ein first-level Topic veröffentlichen respektive von diesem lesen. Intern wird die Topic-Struktur wieder interpretiert und die Nachricht auf das korrekte Topic gepublished (siehe Abbildung 2.21). Dieses Feature haben wir auch angedacht, aufgrund anderer Priorisierung und Zeitmangel aber nicht umgesetzt.

	Local Broker		Public Broker
Topic	sensors/kitchen/temp	Topic	bridgeID
Message	{"temp":25}	Message	{ "topic": "sensors/kitchen/temp", "message": {"temp":25} }

Abbildung 2.21: Verschleierung Topic-Struktur

Sowohl für die End-to-End Verschlüsselung als auch für die Nutzdatenverschlüsselung auf Bridge-Ebene benötigt man eine Software-Komponente, welche die Verschlüsselungs- und Entschlüsselungsfunktionen bereitstellt. Diese haben wir mit unserem «Encryption-Client» umgesetzt.

2.6.1 Encryption-Client

Prinzip

Mit der Implementierung eines Verschlüsselungs-Clients soll der Inhalt von MQTT-Nachrichten verschlüsselt werden und dadurch eine LAN-zu-LAN-Verschlüsselung erreicht werden. Abbildung 2.22 zeigt, wie wir uns die Ver- und Entschlüsselung von payloads mittels eines Encryption-Clients vorstellen. In Umgebung A abonniert der Encryption-Client die Topics, deren Nachrichten

verschlüsselt von Broker A auf Broker B gebridged werden sollen. Das sind alle Topics mit dem Basis-Level «topicsToBeShared» (`topicsToBeShared/#`). Er verschlüsselt die Nachrichten und publiziert sie auf dem gebridgten Topic. Dabei wird das Basis-Topic «topicsToBeShared» mit «`bridgedTopics`» ersetzt. In Umgebung B werden diese vom Encryption-Client abonniert (`bridgedTopics/#`). Er entschlüsselt die Nachrichten und publiziert sie mit dem Basis-Topic «`topicsToBeShared`». Diese werden in Umgebung B von den Clients abonniert.

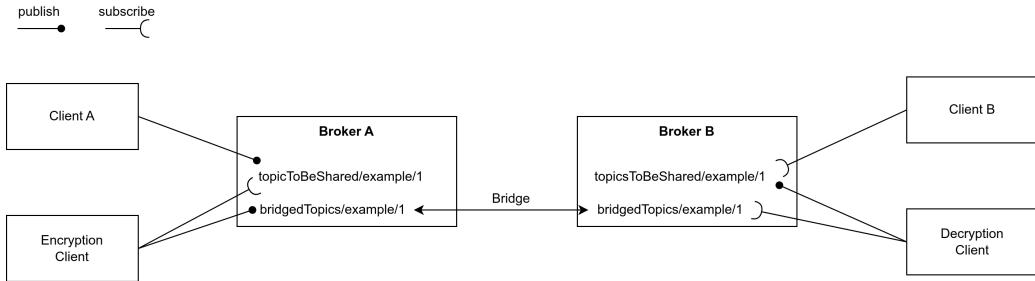


Abbildung 2.22: Prinzip Ver- und Entschlüsselung des Payloads von MQTT-Nachrichten

Die Verschlüsselung soll mit einem symmetrischen Verfahren umgesetzt werden, wobei alle Clients denselben zufällig erzeugten Schlüssel verwenden. Dieser wird in Form von einer Datei (*keyfile*) auf dem Client abgespeichert. Wir haben uns zum einen für symmetrische Verschlüsselung entschieden, da dies einfacher umzusetzen ist als ein asymmetrisches Verfahren. Zum anderen zeichnen sich die Algorithmen der symmetrischen Kryptografie durch eine höhere Geschwindigkeit aus, was für unseren Einsatzzweck zentral ist. Das Verteilen des Keyfiles geschieht «out of band», d.h. es wird nicht durch unsere Applikation verteilt. Wie das Keyfile auf den Client kommt wird nicht vorgegeben, möglich ist beispielsweise eine Austausch über einen USB-Stick oder als Anhang in einer E-Mail. Falls der Encryption-Client in einem grossen verteilten System mit vielen Clients zum Einsatz käme, müssten effizientere Möglichkeiten zum Austauschen des Keyfiles implementiert werden. Aber das Problem des sicheren Schlüsselaustauschs, «eines der ungelösten Probleme der Kryptografie» [60], ist nicht Teil dieser Arbeit.

Nachdem wir den grundsätzlichen Vorgang zur Ver- und Entschlüsselung festgelegt hatten, haben wir nach Open Source Kryptographie Libraries für Python gesucht. Dabei stand für uns die Benutzerfreundlichkeit im Vordergrund. Die Bibliothek sollte die aktuellen Versionen der kryptographischen Algorithmen implementieren, gründlich getestet, von einer vertrauenswürdigen Organisation veröffentlicht sowie einfach anzuwenden sein (*Secure by default*, ohne komplexe Konfigurationen). Als zwei mögliche Kandidaten haben sich «cryptography» und «PyNaCl» herausgestellt, wobei wir PyNaCl den Vorzug gegeben haben.

PyNaCl

PyNaCl ist eine API für Libsodium, einem Fork von NaCl (Networking and Cryptography library, ausgesprochen *salt*) und legt den Fokus auf Benutzerfreundlichkeit, Sicherheit und Geschwindigkeit [27, 1]. NaCl bzw. Libsodium wird unter anderem von Discord, KeePassXC, Threema und WordPress eingesetzt. Eine aktuelle Liste von Applikationen, Projekten und Firmen die Libsodium verwenden findet sich in der Libsodium Dokumentation [28]. Besonders gefallen hat uns, dass PyNaCl immer gleichzeitig verschlüsselt und signiert bzw. entschlüsselt und Signaturen verifiziert. Auf diese Weise können Angriffe verhindert werden, bei denen ein Angreifer den verschlüsselten Wert verändert. Wie bereits im ersten Teil erwähnt, wird beim Verschlüsseln automatisch eine Nonce verwendet, damit aus zwei oder mehr identischen, unverschlüsselten Texten (*Plaintexts*) niemals der gleiche verschlüsselte Text (*Ciphertext*) entsteht. Die Nonce muss nicht zufällig oder unvorhersehbar sein und muss auch nicht geheimgehalten werden. Sie darf zusammen mit dem Ciphertext abgelegt werden. Die einfachste Form einer Nonce wäre ein Zähler, der bei jeder Nachricht hochgezählt wird. Wichtig ist, dass sie nicht wiederverwendet wird, da dies einem Angreifer genügend Informationen liefern kann, um andere Nachrichten zu entschlüsseln oder zu fälschen [27, 13]. Die *encrypt()*-Funktion von PyNaCl kann automatisch eine zufällige Nonce generieren, was die Verschlüsselung sehr einfach macht. Als Rückgabewert der Funktion erhält man ein EncryptedMessage-Objekt, welches wir als Payload einer MQTT-Nachricht versenden können. Bei der Entschlüsselung wird die Nonce automatisch aus dem EncryptedMessage-Objekt ausgelesen und der Ciphertext bzw. die verschlüsselte Nachricht wird ganz einfach mit der *decrypt()*-Funktion entschlüsselt. Zusätzlich wird die Nachricht mit einer «authentication information», einem Message Authentication Code (MAC) versehen, welche unbemerkt Manipulationen verhindert [27, 12-14].

Entwicklung der Prototypen

Das in Abbildung 2.22 gezeigte Prinzip haben wir in einem ersten Prototypen versucht umzusetzen. Diesen haben wir im Stil des «Advanced use»-Beispiels von `asyncio-mqtt` umgesetzt [58]. `asyncio-mqtt` basiert auf dem viel verwendeten Eclipse Paho MQTT Python Client [61] und verwendet wie aiortc die `asyncio`-Bibliothek [47], wodurch mit wenig Aufwand asynchrone Abläufe implementiert werden können. `Asyncio-mqtt` erlaubt ein praktisches Client- und Message-Handling. Es müssen keine Callbacks implementiert werden und anstelle von Return Codes gibt es die `MqttError` Klasse. Die Verschlüsselung des Payloads ist dank der von PyNaCl zur Verfügung gestellten Funktion auch sehr praktisch und erfolgt in wenigen Schritten. Das Listing 2.11 zeigt die im Encryption-Client verwendeten Funktionen mit zusätzlichen Kommentaren zur Erklärung.

```

1  #--- Benötigte Libraries ---
2  import nacl.secret
3  import nacl.utils
4
5  #--- Encryption ---
6  # Schlüssel für die symmetrische Verschlüsselung, wird nicht im
7  # Encryption-Client selbst erstellt sondern übergeben
8  key = nacl.utils.random(nacl.secret.SecretBox.KEY_SIZE)
9  # Die SecretBox wird, zusammen mit dem Schlüssel, zum verschlüsseln verwendet.
10 box = nacl.secret.SecretBox(key)
11 encrypted_payload = box.encrypt(payload)
12 # Dabei werden automatisch Nonce sowie Authentifizierungsinformationen (MAC)
13 # mit gespeichert
14 assert len(encrypted) == len(message) + box.NONCE_SIZE + box.MACBYTES
15
16 #--- Decryption ---
17 # Nonce und MAC werden automatisch ausgelesen und die Nachricht entschlüsselt
18 # sofern der richtige key verwendet wird
19 box = nacl.secret.SecretBox(key)
20 # Wenn die Verschlüsselung manipuliert wurde oder ein anderer Fehler auftrat
21 # wird eine Exception geworfen
22 decrypted_payload = box.decrypt(payload)

```

Listing 2.11: Kryptographische Funktionen von PyNaCl

Nach der Implementierung eines ersten Prototyps haben wir uns die Frage gestellt, ob die Lösung mit den Basis-Topics, über welche die Selektion der zu verschlüsselnden und zu bridgenden Nachrichten geschieht, sinnvoll ist. Es ist eine zweckmässige und programmiertechnisch einfache Lösung, bietet aber wenig Flexibilität. Als nächsten Schritt haben wir deshalb eingerichtet, dass man über Befehlszeilenargumente Listen von Topics angeben kann, auf welche man subscribed und ein Basis-Topic, das den subscribed Topic-Strings zum publishen der Nachrichten mit dem verschlüsselten Payload vorangestellt wird. Eine Nachricht, die auf dem Topic «example/1» ankommt, wird z.B. auf das Topic «encrypted/example/1» gepublished. In diesem Schritt wurde auch hinzugefügt, dass die Angaben zum Broker über die Kommandozeile mitgegeben werden können. Das Listing 2.12 zeigt einen möglichen Programmaufruf und erklärt die Befehlszeilenargumente.

```

1 python3 securebridge.py encrypt 147.87.118.29 8883 cedalo test secret_key.bin --s_topics
2 topics/a/# topics/b/# --p_topic encrypted -v
3 # encrypt: Verschlüsselungs-Funktionalität starten. Der Entschlüsselungs-Modus wird mit
4 # "decrypt" gestartet
5 # 147.87.118.29: Broker IP
6 # 8883: Broker Port
7 # cedalo: Benutzername für Anmeldung bei Broker
8 # test: Passwort des Benutzers
9 # secret_key.bin: Pfad zur Datei in welcher der Schlüssel gespeichert ist
10 # --s_topics topics/a/# topics/b/#: das Tool subscribed die 3 angegebenen Topics (# als
11 # wildcard)
12 # --p_topic encrypted: Als Ziel-Topics der verschlüsselten Nachrichten wird den
13 # ursprünglichen Topics "encrypted" vorangestellt
14 # -v: im verbose logging Modus starten

```

Listing 2.12: Secure-Bridge Beispiel Programmaufruf

In Abbildung 2.23 sieht man, wie die originalen Nachrichten, welche auf die Topics «topics/a/test» und «topics/b/test» gepublished wurden, vom Encryption-Tool verschlüsselt und auf die Topics «encrypted/topics/a/test» respektive «encrypted/topics/b/test» gepublished wurden. Danach wurden sie vom decryption-Tool entschlüsselt und auf «decrypted/topics/a/test» respektive «decrypted/topics/b/test» gepublished. Ausserdem kann man sehen, dass der gleiche Plaintext in beiden Nachrichten in unterschiedlichen Ciphertext gewandelt wird.

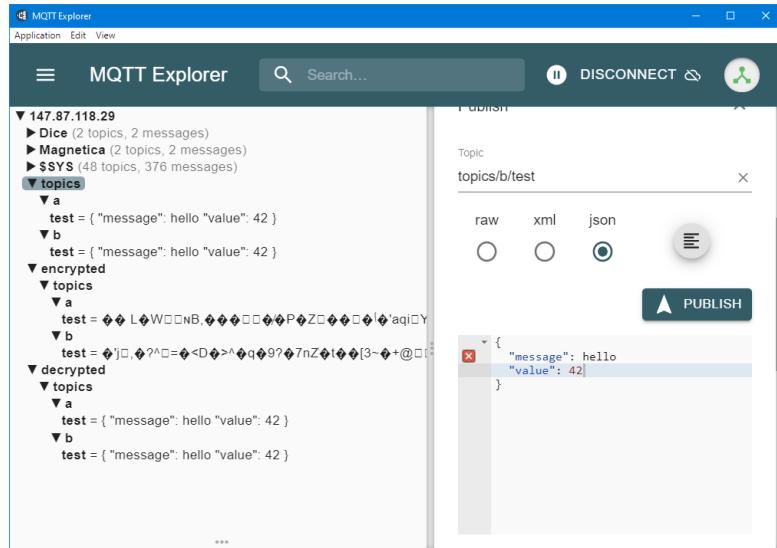


Abbildung 2.23: Cryptool Test mit MQTTEexplorer

Beim Entschlüsseln der Nachrichten wird wohl in der Regel auf das Voranstellen eines Topic-Basis-Levels verzichtet, da man die Nachrichten meistens auf einen anderen Broker mit gleichen Topic-Namen spiegeln möchte (wie im Beispiel im Kapitel 1.4.2). Daher ist im Programm der default-Wert für das Basis-Topic-Level ein leerer String, wodurch das erste Level im Topic-String (wir verwenden hier je-

weils «encrypted») entfernt wird. Für die Veranschaulichung der Funktionsweise des Tools im obigen Beispiel haben wir aber ein Topic-Basis-Level («decrypted») angegeben. So wird auch verhindert, dass ein endless-loop entsteht, wenn man encryption und decryption auf dem gleichen Broker macht.

Beim Testen des Prototypen haben wir festgestellt, dass wir das Subscriben auf «#» nicht erlauben dürfen. Subscript man darauf, werden die verschlüsselten Nachrichten, egal auf welches Topic auf dem Broker gepublished, auch wieder subscribed, wodurch ein endless-loop mit endloser Verschlüsselung entsteht. Wenn man alle Topics des Brokers über die Secure Bridge spiegeln will, muss man sie über ein gemeinsames first-level Topic führen und dieses dem Encryption-Client mit #-wildcard als Argument übergeben. Bei unserem Testaufbau des Magic Message Portals ist dies so realisiert.

Seit MQTT-5.0 besteht die Möglichkeit dem Broker mitzuteilen, dass bei überlappenden Publish- und Subscribe-Bereichen die Nachrichten nicht an den Absender gesendet werden, obwohl das Topic von diesem subscribed ist. Der einfachste Fall um das Verhalten zu beschreiben, ist ein Client A, welcher das Topic «MyLoopTopic» subscribed. Im Default-Fall erhält der Client A alle Nachrichten, welche er selbst auf «MyLoopTopic» published. Beim Subscriben kann der Client A nun die Option «No local» aktivieren [8]. Sobald diese Option aktiv ist, erhält der Client A keine Nachrichten mehr, welche er selbst published.

In unserem Fall müssten wir somit einen Client implementieren, bei welchem eine Instanz verschlüsselt und entschlüsselt. Dies ist nötig, damit die Ver- und Entschlüsselung mit der selben Client-Verbindung erfolgt und so der Loop auch bei einer bidirektionalen Verbindung zwischen den Brokern erfolgreich unterbunden wird. Da aktuell kein konkreter Bedarf hierfür besteht und der Abgabetermin naht, haben wir auf eine Realisierung verzichtet.

In einem nächsten Schritt haben wir eingebaut, dass die Verschlüsselung mit einem Passwort möglich ist anstelle von einem Keyfile. Die PyNaCl Library bietet Funktionen an, um aus einem Passwort einen Schlüssel zu generieren. Das Listing 2.13 zeigt das Code-Beispiel von PyNaCl[[27]], leicht angepasst und mit Kommentaren zu unserer Umsetzung ergänzt.

```

1  from nacl import pwhash, secret, utils
2
3  # message und password werden bei uns der Funktion übergeben
4  password = b'password shared between Alice and Bob'
5  message = b"This is a message for Bob's eyes only"
6
7  # Schlüsselableitungsfunktion (key derivation funktion)
8  kdf = pwhash.argon2i.kdf
9
10 # zufällig generiertes salt
11 salt = utils.random(pwhash.argon2i.SALTBYTES)
12
13 # zusätzliche key derivation parameters, müssen nicht so erstellt werden
14 # wir verwenden die default-Werte
15 ops = pwhash.argon2i.OPSLIMIT_SENSITIVE
16 mem = pwhash.argon2i.MEMLIMIT_SENSITIVE
17
18 # Schlüsselgenerierung
19 Alices_key = kdf(secret.SecretBox.KEY_SIZE, password, salt,
20                   opslimit=ops, memlimit=mem)
21 # bei uns:
22 Alices_box = kdf(secret.SecretBox.KEY_SIZE, password, salt)
23
24 # ab hier können die gleichen Funktionen wie bei der "Secret Key Encryption"
25 # verwendet werden, die nonce im Beispiel der PyNaCl Dokumentation muss
26 # nicht explizit erzeugt und übergeben werden, das passiert standardmäßig
27 box = secret.SecretBox(key)
28 encrypted = box.encrypt(message)
29
30 # die Nachricht und die KDF Parameter (bei uns nur das salt)
31 # werden gesendet, mit denselben Funktionen wird der
32 # Schlüssel hergeleitet und die Nachricht entschlüsselt
33
34 Bobs_key = kdf(secret.SecretBox.KEY_SIZE, password, salt)
35 Bobs_box = secret.SecretBox(Bobs_key)
36 received = Bobs_box.decrypt(encrypted)

```

Listing 2.13: Key derivation mit PyNaCl

Das Salt muss zusammen mit dem Ciphertext in der Nachricht mitgesendet werden. So kann der Empfänger mit seinem identischen Passwort denselben Schlüssel ableiten und den Payload entschlüsseln.

Diese Erweiterung des Encryption-Clients mit Passworteingabe hatte grössere Auswirkungen auf den Code. Da das Salt mit jeder Nachricht mitgesendet wird, musste die Funktion, welche die MQTT-Nachrichten erstellt, angepasst bzw. erweitert werden. Je nach dem, ob der Encryption-Client mit Keyfile oder Passwort gestartet wird, werden andere Funktionen ausgeführt. Auch der Programm Aufruf musste entsprechend angepasst werden. Das Resultat wird im ersten Teil der Arbeit gezeigt.

Wir haben festgestellt, dass die Verschlüsselung bei der Passwort-Variante deutlich langsamer ist als mit dem Keyfile. Ein Grund dafür kann die Schlüsselableitungsfunktion sein, welche zusätzlich zur Verschlüsselungsfunktion für jede Nachricht aufgerufen werden muss. Ein weiterer Grund könnte die zusätzlichen

Schritte zum Konvertieren des Payloads von Python Dictionary zu String zu JSON und umgekehrt sein. Falls die Passwortfunktion im produktiven Umfeld genutzt werden sollte, müsste die Performanz deutlich verbessert werden.

Die Secure Bridge und ein Tool zum Erstellen eines symmetrischen Schlüssels werden, ebenso wie der Application Tunnel, als flatpak-Applikation bereitgestellt. Sie können wie im Listing 2.14 gezeigten installiert werden. Die Programm-Aufrufe sind im ersten Teil der Arbeit gezeigt und erklärt worden.

```
1 flatpak --user install flatpak-salvm4 ch.bfh.ti.secure-bridge
2 flatpak --user install flatpak-salvm4 ch.bfh.ti.secure-bridge-keygen
```

Listing 2.14: Secure-Bridge Beispiel Programmaufruf

2.6.2 Geschwindigkeits-Test

Es wäre interessant, eingehender zu testen, wie gross die Geschwindigkeitseinbussen sind, wenn die Nachrichten verschlüsselt werden. Aus Zeitgründen und wegen anderer Prioritäten haben wir dies aber nicht durchgeführt.

2.7 The Magic Message Portal

Mit Hilfe des WebRTC-Tunnels und der Secure-Bridge-Verschlüsselungapplikation kann nun in Kombination mit der Bridge-Funktionalität der MQTT-Broker ein «Magic Message Portal» konfiguriert werden. Der angedachte Aufbau ist in der Abbildung 2.24 dargestellt.

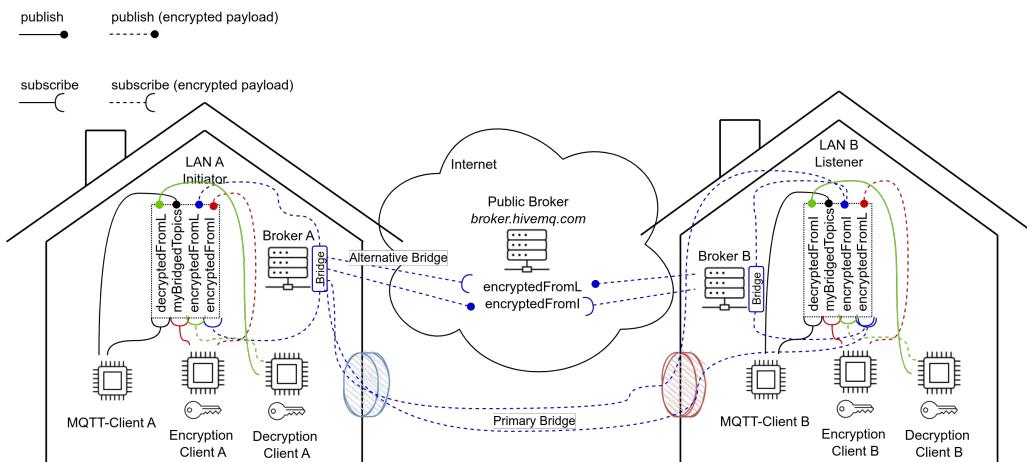


Abbildung 2.24: The Magic Message Portal

Neben dem «Application Tunnel», welcher hier nur symbolisch durch die Portal-Zeichen dargestellt ist, kommen auch der «Encryption Client» und die Bridge Funktionalität des Mosquitto-Brokers zum Einsatz.

Die beiden LAN sind als «Initiator» und als «Listener» bezeichnet, wobei sich diese Namensgebung an der Rolle des verwendeten Tunnelclient orientiert. Pro LAN ist die relevante Topic-Struktur für das Magic Message Portal dargestellt, bei welcher die Rollen jeweils abgekürzt geschrieben sind («I» für Initiator und «L» für Listener). Die Topic-Struktur des Relay-Brokers, welcher für den Fallback-Modus verwendet wird, ist ebenfalls mit dieser Notation dargestellt.

Die Idee hinter diesem Aufbau ist, dass der Mosquitto-Broker auf der Initiatoren-Seite autonom feststellt, ob die primäre Bridge- Verbindung durch den Applic-Tunnel nicht verfügbar ist und automatisch die alternative Verbindungen über den Public-Broker wählt. Dieses Funktionalität ist im Mosquitto-Broker bereits vorhanden. Dort können bei der Bridge-Konfiguration mehrere Broker-Adressen angegeben werden, welche als alternative Route verwendet werden. Ist die alternative Route aktiv, wird zyklisch versucht, auf die primäre Route zu wechseln. Die entsprechende Bridge-Konfiguration des Mosquitto-Brokers ist auf dem Listing 2.15 aufgeführt. Die detaillierte Beschreibung der Konfigurationen ist auf der Mosquitto-Homepage einsehbar [9].

```

1 # bridge name
2 connection secure-bridge
3
4 # addresses (primary: localhost:port of applic-tunnel-client, alternativ: public broker)
5 addresses 127.0.0.1:1884 broker.hivemq.com
6
7 # deactivate round robin on remote brokers (try to use first broker if possible)
8 round_robin false
9
10 # activate clean session
11 cleansession true
12
13 # keep alive interval in seconds (retry to connect to primary bridge address every x
14 # seconds)
14 keepalive_interval 5
15
16 # topics (publish and subscribe)
17 topic # in 1 encryptedFromListener/ encryptedFromListener/
18 topic # out 1 encryptedFromInitiator/ encryptedFromInitiator/

```

Listing 2.15: Bridge-Konfiguration Initator

Auf der Listener-Seite ist ebenfalls eine Bridge konfiguriert, welche ihrerseits die konfigurierten Topics auf den Public-Broker bridget (siehe Listing 2.16).

```

1 # bridge name
2 connection secure-bridge
3
4 # addresses
5 address broker.hivemq.com
6
7 # activate clean session
8 cleansession true
9
10 # keep alive interval in seconds
11 keepalive_interval 5
12
13 # topics (publish and subscribe)
14 topic # in 1 encryptedFromInitiator/ encryptedFromInitiator/
15 topic # out 1 encryptedFromListener/ encryptedFromListener/

```

Listing 2.16: Bridge-Konfiguration Lister

Wie die aktive Konfiguration beim Wegfallen der Tunnel-Verbindung aussieht, ist in der Abbildung 2.25 ersichtlich. Bei dieser Konfiguration existieren prinzipiell keine Einschränkungen.

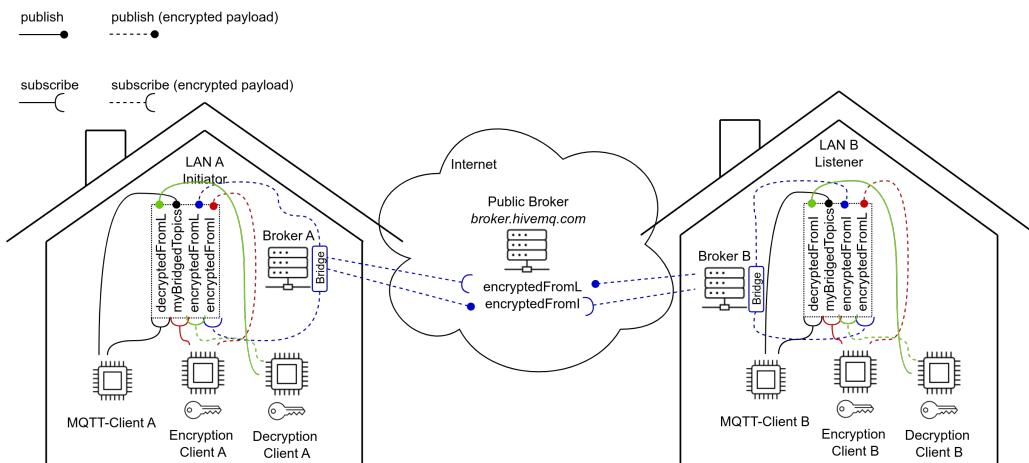


Abbildung 2.25: The Magic Message Portal (Fallback bridge)

Zwei Schwächen dieses Aufbaus sind beim Bridgen mit aktivem Tunnel vorhanden (Abbildung 2.26). Der erste Schwachpunkt ist, dass so die Payload der Nachrichten immer verschlüsselt werden, obwohl der Tunnel eigentlich eine sichere Verbindung auf dem Transport-Layer bereitstellt. Dies bedeutet, dass unnötig Rechenleistung eingesetzt wird.

Der grössere Schwachpunkt liegt jedoch bei der Bridge auf der Listenerseite. Da die Listenerseite nichts vom Status des Tunnels erfährt, bleibt die Bridge immer aktiv. Somit werden die Topics auch immer auf den Public-Broker gebridget, was spätestens beim Fallback berücksichtigt werden muss. Beim Fallback muss sich

die Bridge des Brokers A mit «Clean Session» auf die Topics des Public Brokers subscriben. Andererseits werden bereits durch den Tunnel gesendete Nachrichten nochmals verarbeitet. «Clean Session» bedeutet jedoch auch, dass Nachrichten zwischen dem Tunnelunterbruch und dem neuen Subscriben verloren gehen.

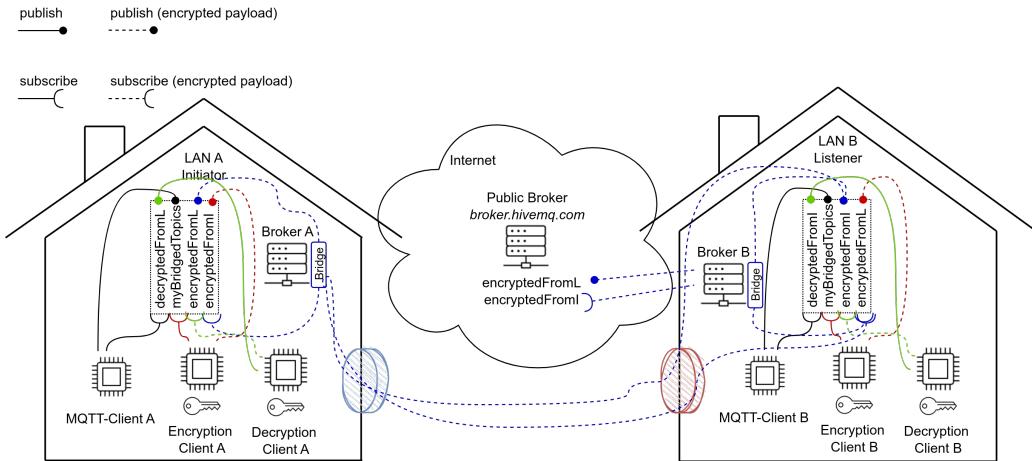


Abbildung 2.26: The Magic Message Portal (Primary bridge)

Wie vorgängig erläutert, existieren mit der aktuellen Konfiguration Schwachpunkte, welche je nach Einsatzszenario nicht akzeptabel sind. Hauptursache hierbei ist der asymmetrische Aufbau der Bridge, welche bei aktivem Tunnel im Einsatz ist. Bei diesem Szenario muss nur noch die Bridge vom Broker A aktiv sein, welche direkt auf dem Broker B subscribt und published. Somit wird die Bridge B überflüssig, was diese jedoch nicht feststellen kann.

Eine mögliche Lösung dieser Problematik ist in der Abbildung 2.27 skizziert. Hierbei werden die Bridges vom Broker A und vom Broker B immer benötigt. Der Trick hierbei ist, dass diese jeweils nur publishen. Auf dem Broker B wird ebenfalls via Tunnel eine primäre Bridge-Verbindung zum Broker A aufgebaut. Um dies zu realisieren, muss die Tunnelapplikation so angepasst werden, dass der TCP-Verbindungsauflauf von beiden Seiten möglich ist. Dies dürfte nach unserer Betrachtung relativ einfach realisierbar sein. Das bedeutet, dass neu auch der Initiator nach dem Erstellen der PeerConnection ein TCP-Serversocket öffnet und bei neuen Verbindungen ein Datenkanal erstellt. Die Initiatorseite müsste auch das Ziel kennen und bei Datenkanälen vom Initiator die getunnelten TCP-Nachrichten an das Ziel weiterleiten. In der Abbildung 2.28 sind die neuen Funktionalitäten grün hervorgehoben.

Aufgrund der fortschreitenden Zeit haben wir jedoch auf diese Implementation verzichtet.

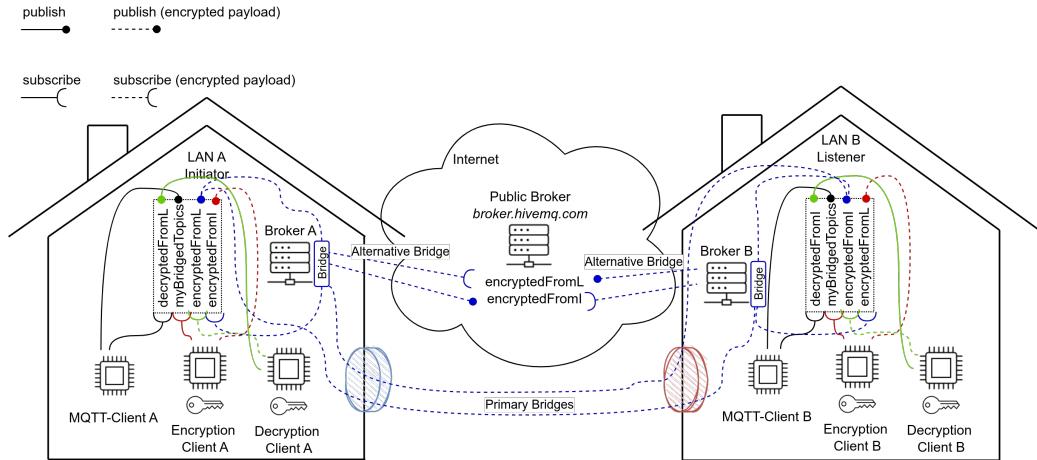


Abbildung 2.27: The Magic Message Portal (Advanced Tunnel)

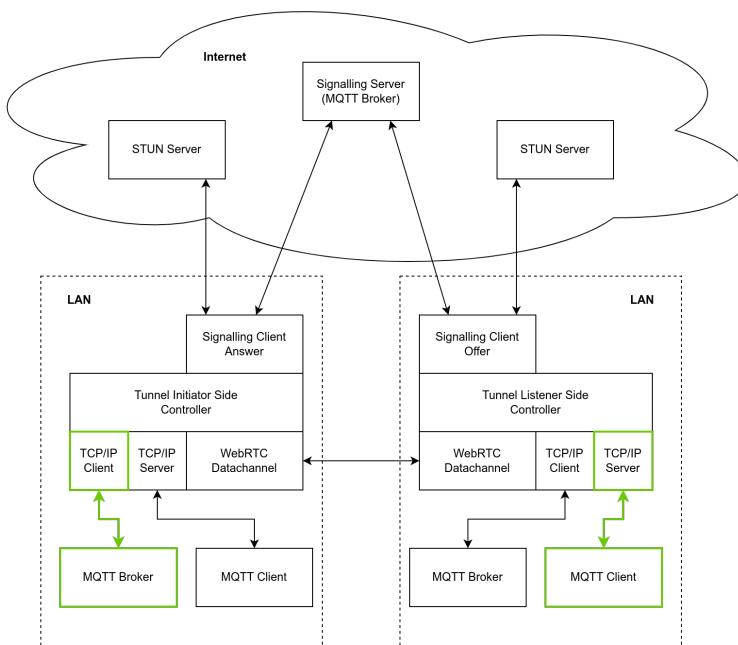


Abbildung 2.28: Tunnel mit symmetrischen Verbindungs möglichkeiten

2.8 Dynamisches Bridgen

Ausgehend von unseren Voraarbeiten im Projekt 2 haben wir auch die Implementierung eines Clients zum dynamischen Bridgen in Betracht gezogen. Die

im Master-Branch von Mosquitto umgesetzte Art des Bridgings ist statisch. Änderungen an der Konfiguration wirken erst nach einem Neustart des Brokers. Im dynamic-bridge-Branch, der auf GitHub verfügbar ist, ist es möglich, Bridges dynamisch zu erstellen, d.h. ohne den Broker neu starten zu müssen. Dazu verwendet Mosquitto spezielle BRIDGE-Topics, welche ähnlich funktionieren wie die SYS-Topics. Alle SYS-Topics beginnen mit \$SYS und werden verwendet, um Statusinformationen des Brokers anzusehen. Clients können sich auf Topics in der \$SYS-Hierarchie subscriben und so lesend auf die Informationen zugreifen. Die BRIDGE-Topics beginnen mit \$BRIDGE und sind schreib- und lesbar mit ACL-Schutz (Access Control List siehe Kapitel 1.3.4). Eine dynamische Bridge kann entweder in der Kommandozeile mit dem mosquitto_bridge-Tool erstellt oder gelöscht werden, oder indem Nachrichten auf das Topic «\$BRIDGE/new» bzw. «\$BRIDGE/del» gepublished werden [62]. Eine solche Funktionalität wäre wünschenswert, da sie ermöglichen würde, nur so viele Informationen auf den im Internet exponierten Broker zu publishen wie nötig. Dieses Szenario wird in Abbildung 2.29 dargestellt.

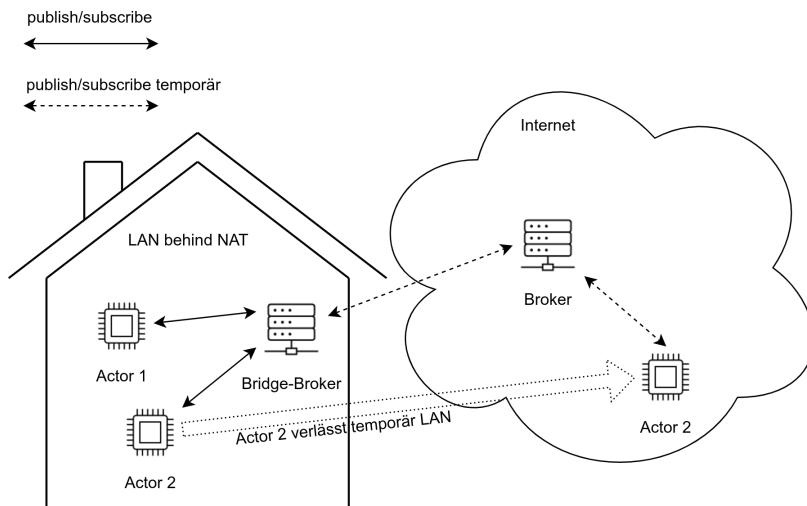


Abbildung 2.29: Bridge bei Bedarf

Angenommen Actor 2 sei eine Applikation zum Überwachen eines Systems. Solange er sich im lokalen Netzwerk befindet, wird die Bridge-Funktion nicht verwendet, das lokale MQTT-System läuft autonom. Falls Actor 2 jedoch das LAN verlässt, sollen trotzdem noch selektiv Statusinformationen von ihm konsumiert werden können. Der lokale MQTT-Broker erstellt in dieser Situation eine Bridge auf einen Broker im Internet, und spiegelt selektiv Topics. Der Actor 2 subscribt sich auf diese Topics und bleibt so über den Zustand des Systems informiert. Sobald er wieder im LAN ist, wird die Bridge abgebaut.

Selber eine dynamic-bridge-Funktionalität zu implementieren haben wir aber

bereits zu Beginn der Arbeit als tiefe Priorität eingestuft. Diese Funktionalität müsste unserer Meinung nach direkt im Broker implementiert werden. Unser «Magic Message Portal» würde sich aber sehr gut mit einer dynamic-bridge-Funktion kombinieren lassen.

2.9 Projektplanung

Die zeitliche und inhaltliche Projektplanung haben wir in einem über Teams geteilten Excel-Sheet festgehalten. In diesem haben wir die unterschiedlichen Teil-Projekte und Arbeiten aufgeführt, nach Priorität geordnet und zeitlich auf die zur Verfügung stehenden Semesterwochen aufgeteilt. Hier wurden nicht nur die inhaltlichen Aspekte, die zur Thesis gehören, aufgeführt, sondern insbesondere auch die Arbeit an der Dokumentation und den anderen geforderten «deliverables» (Präsentation, Seite für das Journal, Kurzfilm, Verteidigung). Dieses Excel-Sheet war sehr hilfreich dabei, den Überblick über die Arbeit zu behalten. Es ist in unserem Git-Repository abgelegt.

Während den Semesterwochen haben wir uns ein bis zwei Mal wöchentlich getroffen. Fixe Treffen fanden jeden Mittwoch statt, zum Austausch über den Stand der Arbeit und das weitere Vorgehen für die nächste Woche. Bei Bedarf kamen weitere Treffen oder Teams-Calls dazu, jeweils freitags oder samstags. Alle zwei bis drei Wochen trafen wir uns zudem mit unserem Betreuer, Reto Koenig, um das Projekt zu besprechen. Die Kommunikation im Team und die Aufgabenteilung hat für uns sehr gut funktioniert. Die ausgeführten Arbeiten haben wir in einem Arbeitsjournal in der gleichen Excel-Datei aufgeführt. Somit wussten wir laufend, wer woran gearbeitet und wie viel Zeit er dafür aufgewendet hatte.

Für die Thesis haben wir diverse Software Tools verwendet:

- ▶ Teams: Online-Calls, Dateiablage Projektplanung und Journal, Datei- und Ideenaustausch im Chat
- ▶ git und gitlab: Repo für Dokumentation und Softwarekomponenten
- ▶ LaTex: Erstellen der Dokumentation mit Template der BFH
- ▶ PyCharm: Entwicklungsumgebung für Softwarekomponenten
- ▶ draw.io: Zeichnen der Grafiken

2.10 Schlussbetrachtung

Ein «Proof of Concept» wurde mit dieser Arbeit erbracht. Der Application Tunnel mittels WebRTC war sicher der konzeptuell und programmiertechnisch schwierigere und unsicherere Teil der Arbeit. Hier war uns nicht von Anfang an klar, ob es

tatsächlich umsetzbar sein würde. Die Hürde WebRTC, mit all den unterschiedlichen Protokollen und Standards schien zunächst relativ hoch. Es dauerte auch eine Weile, bis wir uns in diesem RFC-Dschungel zurechtfanden. Es war aber gut zu merken, dass man sich nicht von zunächst sehr komplex erscheinenden Technologien abschrecken lassen sollte. Mit aiortc hatten wir dann eine zugängliche Library zur Verfügung, mit der wir die gewünschte Funktionalitäten umsetzen konnten. Mit den «Multi-Connection» und «Multi-Initiator» haben wir unser anfängliches Ziel sogar etwas übertroffen.

Beim Encryption-Client war die Ausgangslage etwas anders. Hier hatten wir an der Umsetzbarkeit wenig Zweifel. Dadurch haben wir diesen Teil der Arbeit allerdings etwas unterschätzt. Es zeigte sich, dass eine effiziente und brauchbare Umsetzung gar nicht so einfach ist. Die Tests zeigen, dass die vorliegende Implementation zu langsam ist, um produktiv eingesetzt werden zu können.

Die Bereitstellung der Applikationen über Flatpak war nicht direkt so geplant, erleichtert unserer Meinung nach jedoch den Einsatz als Technolgiedemonstrator.

Was uns teilweise schwer fiel, waren die Definition der API unserer Applikationsaufrufe und auch die Entscheidung, wie viele Funktionen implementiert werden sollen. Besonders da kein konkretes Anwendungsszenario vorlag, welches als Entscheidungsgrundlage dienen konnte. Wir haben versucht, einen für den produktiven Einsatz sinnvollen Funktionsumfang bereitzustellen, ohne dabei die Anwendung unnötig kompliziert zu gestalten. Eine hohe Komplexität kann bei der Verwendung als Technologie-Demonstrator bzw. Funktionsmuster hinderlich sein. Daher sind für viele Parameter mit default-Werten vorhanden, welche mittels der API optional überschrieben werden können.

Betreffend Arbeitsorganisation und Koordination im Team hat es für uns sehr gut gepasst. Durch den mindestens wöchentlichen Austausch konnten wir flexibel auf die jeweiligen Situationen reagieren und die vorgesehenen Arbeitsschritte bei Bedarf anpassen. Diese Arbeitsorganisation hat dem explorativen Ansatz der Arbeit unserer Meinung nach gut entsprochen. Was wir im Nachhinein wohl anders machen würden, wäre der Dokumentations-Prozess. Wir haben die Dokumentation zwar bereits während der Arbeit geführt, dies jedoch meist in fixen Arbeitsblöcken. Eine laufende Dokumentation und anschliessende Schlussredaktion wäre jedoch im Nachhinein der effizientere Ansatz gewesen.

Unser persönliche Anspruch an die Bachelorarbeit wurde in hohem Masse erfüllt. Die Arbeit weist eine hohe Vielfalt an Themen auf, was zu abwechslungsreichen Arbeitspaketen führte und extrem lehrreich war. Wir haben uns auf neue Themen eingelassen (WebRTC, Flatpak), jedoch auch Bekanntes vertieft (Python, MQTT).

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die hier vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Sämtliche Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, habe ich als solche kenntlich gemacht.

Hiermit stimme ich zu, dass die vorliegende Arbeit in elektronischer Form mit entsprechender Software überprüft wird.

15. Januar 2023



M. Salvisberg



R. Meyer

Literaturverzeichnis

- [1] wikipedia. Wikipedia artikel zu tunnel (rechnernetz). Online verfügbar unter: [https://de.wikipedia.org/wiki/Tunnel_\(Rechnernetz\)](https://de.wikipedia.org/wiki/Tunnel_(Rechnernetz)), abgerufen am 12.01.2023.
- [2] Elektronik Kompendium. Vpn - virtual private network. Online verfügbar unter: <https://www.elektronik-kompendium.de/sites/net/0512041.htm>, abgerufen am 19.11.2022.
- [3] Humboldt-Universität. Nat traversal. Online verfügbar unter: https://sarwiki.informatik.hu-berlin.de/NAT_Traversal, abgerufen am 19.11.2022.
- [4] Eytan Manor. An architectural overview for webrtc — a protocol for implementing video conferencing. Online verfügbar unter: <https://eytanmanor.medium.com/an-architectural-overview-for-web-rtc-a-protocol-for-implementing-video-conferencing-e2a914628d0e>, abgerufen am 19.11.2022.
- [5] MQTT.org. Mqtt specifications. Online verfügbar unter: <https://mqtt.org/mqtt-specification>, abgerufen am 07.10.2022.
- [6] MQTT.org. Mqtt software. Online verfügbar unter: <https://mqtt.org/software/>, abgerufen am 07.10.2022.
- [7] OASIS. Oasis mqtt version 3.1.1. Online verfügbar unter: <http://docs.oasisopen.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf>, abgerufen am 21.10.2022.
- [8] OASIS. Oasis mqtt version 5.0. Online verfügbar unter: [https://docs.oasisopen.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.pdf](http://docs.oasisopen.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.pdf), abgerufen am 21.10.2022.
- [9] Roger Light. mosquitto.conf man page. Online verfügbar unter: <https://mosquitto.org/man/mosquitto-conf-5.html>, abgerufen am 29.10.2022.
- [10] microfast GmbH. Smoker documentation. Online verfügbar unter: <https://smoker-doc.app.microfast.ch/docs/>, abgerufen am 29.10.2022.
- [11] w3c.org und ietf.org. Web real-time communications (webrtc) verändert die kommunikationslandschaft; wird eine empfehlung des world wide web consortium (w3c) und zu mehreren standards der internet engineering

- task force (ietf). Online verfügbar unter: <https://www.w3.org/2021/01/pressrelease-webrtc-rec.html.de>, abgerufen am 08.10.2022.
- [12] Google Developers. Echtzeitkommunikation für das web. Online verfügbar unter: <https://webrtc.org/>, abgerufen am 08.10.2022.
 - [13] Ilya Grigorik. Webrtc. Online verfügbar unter: <https://hpbn.co/webrtc/>, abgerufen am 08.10.2022.
 - [14] MDN contributors developer.mozilla.org. Webrtc api. Online verfügbar unter: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API, abgerufen am 08.10.2022.
 - [15] Felix Weinrank, Martin Becke, Julius Flohr, Erwin Rathgeb, Irene Rungeler, and Michael Tuxen. Webrtc data channels. *IEEE Communications Standards Magazine*, 1(2):28–35, 2017.
 - [16] Dan Ristic. *Learning WebRTC*. Packt Publishing, Birmingham, UK, 2015.
 - [17] Carlos Delgado. List of free functional public stun servers 2021. Online verfügbar unter: <https://ourcodeworld.com/articles/read/1536/list-of-free-functional-public-stun-servers-2021>, abgerufen am 08.10.2022.
 - [18] Contributors coturn. Coturn turn server. Online verfügbar unter: <https://github.com/coturn/coturn>, abgerufen am 09.10.2022.
 - [19] John Selbie. Stuntman open source stun server software. Online verfügbar unter: <https://www.stunprotocol.org/>, abgerufen am 09.10.2022.
 - [20] Next Path Software Consulting Inc. Open relay: Free webrtc turn server. Online verfügbar unter: <https://www.metered.ca/tools/openrelay/>, abgerufen am 09.10.2022.
 - [21] xirsys.com. Turn server cloud. Online verfügbar unter: <https://xirsys.com/>, abgerufen am 09.10.2022.
 - [22] wikipedia. Wikipedia artikel zu netzwerkadressübersetzung. Online verfügbar unter: <https://de.wikipedia.org/wiki/Netzwerkadress%C3%BCbersetzung>, abgerufen am 12.01.2023.
 - [23] Cullen Jennings Francois Audet. Rfc 4787: Nat udp unicast requirements. Online verfügbar unter: <https://www.rfc-editor.org/rfc/pdfrfc/rfc4787.txt.pdf>, abgerufen am 12.01.2023.
 - [24] Jonathan Rosenberg Ari Keranen, Christer Holmberg. Rfc 8445: Interactive connectivity establishment (ice). Online verfügbar unter: <https://www.rfc-editor.org/rfc/pdfrfc/rfc8445.txt.pdf>, abgerufen am 12.01.2023.
 - [25] NTT Communications. A study of webrtc security. Online verfügbar unter: <https://webrtc-security.github.io/>, abgerufen am 18.11.2022.

- [26] wikipedia. Wikipedia-artikel zu base32. Online verfügbar unter: <https://de.wikipedia.org/wiki/Base32>, abgerufen am 14.11.2022.
- [27] unknown. Pynacl: Python binding to the libsodium library, release 1.4.0. Online verfügbar unter: https://pynacl.readthedocs.io/_/downloads/en/1.4.0/pdf/, abgerufen am 28.10.2022.
- [28] unknown. Libsodium documentation. Online verfügbar unter: <https://doc.libsodium.org/>, abgerufen am 14.12.2022.
- [29] unknown. Nacl webpage. Online verfügbar unter: <https://nacl.cr.yp.to/>, abgerufen am 14.12.2022.
- [30] Daniel J. Bernstein. A state-of-the-art diffie-hellman function. Online verfügbar unter: <https://cr.yp.to/ecdh.html>, abgerufen am 14.12.2022.
- [31] wikipedia. Wikipedia-artikel zu curve25519. Online verfügbar unter: <https://de.wikipedia.org/wiki/Curve25519>, abgerufen am 14.12.2022.
- [32] unknown. Flatpak - homepage. Online verfügbar unter: <https://flatpak.org/>, abgerufen am 14.12.2022.
- [33] wikipedia. Wikipedia artikel zu schlüsselstreckung. Online verfügbar unter: <https://de.wikipedia.org/wiki/Schl%C3%BCsselstreckung>, abgerufen am 16.12.2022.
- [34] HiveMQ GmbH. *MQTT and MQTT 5 Essentials*. HiveMQ, Landshut, DE, 2020.
- [35] Ryo Kanbayashi. Github: aiortc-dc. Online verfügbar unter: <https://pypi.org/project/aiortc-dc/>, abgerufen am 21.12.2022.
- [36] WebRTC Magazine. Tiny webrtc datachannel implementation. Online verfügbar unter: <https://webrtcmagazine.wordpress.com/2015/09/30/librtcdt-tiny-webrtc-datachannel-implementation/>, abgerufen am 21.12.2022.
- [37] individual mozilla.org contributors. Dtls (datagram transport layer security). Online verfügbar unter: <https://developer.mozilla.org/en-US/docs/Glossary/DTLS>, abgerufen am 11.11.2022.
- [38] Google Developers. Capture media with webrtc. Online verfügbar unter: <https://developers.google.com/learn/pathways/webrtc-media-capture>, abgerufen am 08.10.2022.
- [39] Sean DuBois. How can i use webrtc on desktop application? Online verfügbar unter: <https://stackoverflow.com/questions/19078787/how-can-i-use-webrtc-on-desktop-application>, abgerufen am 19.10.2022.
- [40] Contributors webrtc java. Webrtc for desktop platforms running java. Online verfügbar unter: <https://github.com/devopvoid/webrtc-java>, abgerufen am 14.12.2022.

- [41] Contributors aiortc. github.com repository of aiortc. Online verfügbar unter: <https://github.com/aiortc/aiortc>, abgerufen am 16.10.2022.
- [42] IBM Corporation. Real-time communications. Online verfügbar unter: <https://www.ibm.com/docs/en/was-liberty/base?topic=overview-real-time-communications>, abgerufen am 19.10.2022.
- [43] Caleb Doxsey. Rtctunnel, build network tunnels over webrtc. Online verfügbar unter: <https://github.com/rtctunnel/rtctunnel>, abgerufen am 14.12.2022.
- [44] Przemyslaw Cimcioch. Rtc tunnel, rtc tunneling for sockets. Online verfügbar unter: <https://github.com/pcimcioch/rtc-tunnel>, abgerufen am 14.12.2022.
- [45] John Suu. Sshx, p2p ssh using webrtc. Online verfügbar unter: <https://github.com/suutaku/sshx>, abgerufen am 14.12.2022.
- [46] Synology Inc. Quickconnect. Online verfügbar unter: https://kb.synology.com/de-de/DSM/help/DSM/AdminCenter/connection_quickconnect?version=7, abgerufen am 16.12.2022.
- [47] The Python Software Foundation. asyncio – asynchronous i/o. Online verfügbar unter: <https://docs.python.org/3/library/asyncio.html>, abgerufen am 19.10.2022.
- [48] Google Developers. Webrtc - signaling. Online verfügbar unter: <https://webrtc.org/getting-started/peer-connections>, abgerufen am 19.10.2022.
- [49] wikipedia. Wikipedia-artikel zu base64. Online verfügbar unter: <https://de.wikipedia.org/wiki/Base64>, abgerufen am 14.11.2022.
- [50] Internet Security Research Group (ISRG). Let's encrypt homepage. Online verfügbar unter: <https://letsencrypt.org/>, abgerufen am 19.10.2022.
- [51] Let's encrypt community. Discussion: Ssl on a ip instead of domain. Online verfügbar unter: <https://community.letsencrypt.org/t/ssl-on-a-ip-instead-of-domain/90635>, abgerufen am 19.10.2022.
- [52] Stack Holdings GmbH. Zerossl homepage. Online verfügbar unter: <https://zerossl.com>, abgerufen am 19.10.2022.
- [53] GitLab B.V. Gitlab pages. Online verfügbar unter: <https://docs.gitlab.com/ee/user/project/pages/>, abgerufen am 05.11.2022.
- [54] Alexander Larsson. Maintaining a flatpak repository. Online verfügbar unter: <https://blogs.gnome.org/alexwl/2017/02/10/maintaining-a-flatpak-repository/>, abgerufen am 05.11.2022.
- [55] im Issue auf github ersichtlich diverse. Document how to cross-compile.

Online verfügbar unter: <https://github.com/flatpak/flatpak/issues/5>, abgerufen am 05.11.2022.

- [56] Librem5 documentation. Cross-building flatpaks. Online verfügbar unter: <https://github.com/flatpak/flatpak/issues/5>, abgerufen am 05.11.2022.
- [57] QEMU. Qemu - homepage. Online verfügbar unter: <https://github.com/flatpak/flatpak/issues/5>, abgerufen am 05.11.2022.
- [58] Frederik Aalund. asyncio-mqtt 0.13.0 - mqtt client with idiomatic asyncio interface. Online verfügbar unter: <https://pypi.org/project/asyncio-mqtt/>, abgerufen am 19.10.2022.
- [59] Contributors mqtt.org. public brokers. Online verfügbar unter: https://github.com/mqtt/mqtt.org/wiki/public_brokers, abgerufen am 08.10.2022.
- [60] Elektronik Kompendium. Kryptografische protokolle / verschlüsselungsverfahren. Online verfügbar unter: <https://www.elektronik-kompendium.de/sites/net/0908071.htm>, abgerufen am 09.12.2022.
- [61] Contributors paho.mqtt.python. Eclipse paho™ mqtt python client. Online verfügbar unter: <https://github.com/eclipse/paho.mqtt.python>, abgerufen am 01.12.2022.
- [62] Roger Light. github - eclipse mosquitto (dynamic-bridge branch). Online verfügbar unter: <https://github.com/eclipse/mosquitto/tree/dynamic-bridge>, abgerufen am 21.10.2022.

Abbildungsverzeichnis

1.1	Ausgangslage	2
1.2	Problem	2
1.3	Ziel	3
1.4	MQTT-WebRTC-Tunnel	4
1.5	Secure-Bridge mit verschlüsselten Nutzdaten	4
1.6	Publish/Subscribe-Architektur von MQTT	6
1.7	Beispiel einer Topic Struktur	7
1.8	MQTT Bridge	7
1.9	Zentraler Broker WAN	9
1.10	Lokale Bridge-Broker	9
1.11	WebRTC Protokoll-Stack	12
1.12	Ablauf Verbindungsaufbau WebRTC mit Signaling	13
1.13	WebRTC Peer-to-Peer Architektur	14
1.14	Full Cone NAT [22]	15
1.15	Restricted Cone NAT [22]	15
1.16	Port restricted Cone NAT [22]	15
1.17	Port restricted Cone NAT [22]	16
1.18	Übersicht Tunneling Client	17
1.19	Verbindungsebenen MQTT-Tunnel	18
1.20	MQTT Signaling	19
1.21	Mögliche Setups mit Application Tunnel	23
1.22	Nachrichtenaustausch zwischen lokalen Brokern mit der Secure Bridge»	24
1.23	The Magic Message Portal	28
2.1	WebRTC Protokoll-Stack nach Weinrank [15]	32
2.2	WebRTC Protokoll-Stack nach Grigorik [13]	32
2.3	QuickConnect Hole Punching [46, 6]	34
2.4	QuickConnect Relay Service [46, 6]	34
2.5	aiortc datachannel-cli	35
2.6	Sequenz MQTT-Tunnel	36
2.7	MQTT Signaling	37
2.8	MQTT Secure Signaling	38
2.9	MQTT secure signaling	40
2.10	MQTT-Explorer: Secure signaling	41
2.11	Szenario MQTT-Tunnel mit Multichannel	42

2.12	Sequenz MQTT-Tunnel mit Multichannel	43
2.13	Szenario MQTT-Tunnel mit mehreren Initiator	44
2.14	Prinzip Messung Round Trip Time	48
2.15	RTT Messung mit Broker	49
2.16	RTT Messung mit Tunnel	49
2.17	Round Trip Time Messungen Resultate Übersicht	52
2.18	Round Trip Time Messungen Vergleich Mittelwerte	52
2.19	End-to-End Verschlüsselung	53
2.20	Nutzdatenverschlüsselung auf Bridge-Ebene	54
2.21	Verschleierung Topic-Struktur	54
2.22	Prinzip Ver- und Entschlüsselung des Payloads von MQTT-Nachrichten	55
2.23	Cryptool Test mit MQTTEexplorer	58
2.24	The Magic Message Portal	61
2.25	The Magic Message Portal (Fallback bridge)	63
2.26	The Magic Message Portal (Primary bridge)	64
2.27	The Magic Message Portal (Advanced Tunnel)	65
2.28	Tunnel mit symmetrischen Verbindungsmöglichkeiten	65
2.29	Bridge bei Bedarf	66
1	Test environment	85
2	Connection settings	87
3	Connection open	87

Tabellenverzeichnis

2.1 Round Trip Time Messungen Resultate	51
---	----

Listings

1.1	Auschnitt mosquitto.conf zu Bridges	8
1.2	Aufruf Application Tunnel	20
1.3	Aufruf Application Tunnel «KeyGenerator»	20
1.4	Aufruf Application Tunnel «Initiator»	21
1.5	Aufruf Application Tunnel «Listener»	22
1.6	Programmaufrufe für Beispiel in Abbildung 1.22	25
1.7	Secure-Bridge CLI Optionen	25
1.8	Konfiguration der Bridge-Broker A und B	26
1.9	Secure-Bridge Key Generator Aufruf	27
2.1	Nonce Nachricht	38
2.2	Offer Nachricht	39
2.3	Answer Nachricht	39
2.4	Flatpak-Manifest	46
2.5	Ubuntu ARM cross building dependencies	47
2.6	flatpak-builder cross compile	48
2.7	RTT Analyzer Tools Installation	49
2.8	RTT Analyzer Tools Programmaufrufe	50
2.9	RTT Analyzer Tools help-Funktion Ausgabe	50
2.10	Bridge Konfiguration RTT für Tests	51
2.11	Kryptographische Funktionen von PyNaCl	57
2.12	Secure-Bridge Beispiel Programmaufruf	58
2.13	Key derivation mit PyNaCl	60
2.14	Secure-Bridge Beispiel Programmaufruf	61
2.15	Bridge-Konfiguration Initiator	62
2.16	Bridge-Konfiguration Lister	63
1	Add flathub repository	86
2	Install application tunnel	86
3	Run initiator client	86

Glossar

Actor MQTT-Client mit einer spezifischen Anwendung, beispielsweise einen Sensor oder Aktuator.

Bridge Die «Bridge»-Funktionalität von MQTT-Brokern. Erlaubt das publishen und subscriben von Topics unter Brokern.

Broker Ein MQTT-Server, welcher für die Verteilung von Nachrichten an MQTT-Client zuständig ist.

DTLS Datagram Transport Layer Security: Ein Verschlüsselungsprotokoll zur Sicherung von UDP-Verbindungen.

Firewall Ein Sicherungssystem für Netzwerke oder einzelne Rechner. Kann den Zugang zum oder vom Netzwerk / Rechner einschränken um unerlaubten Zugriff zu verhindern.

ICE Interactive Connectivity Establishement: Dient dem Peer-to-Peer-Verbindungsaufbau und verwendet unter anderem TURN und STUN.

IoT Internet of things: Ist ein Sammelbegriff für die Technologien zur Vernetzung von virtuellen und physikalischen Objekten mittels Informations- und Kommunikationstechniken.

IP Internet Protocol: Das Internetprotokoll implementiert die Internetschicht des TCP/IP-Modells bzw. die Vermittlungsschicht im OSI-Modell.

LAN Local Area Network: Ein lokales Rechnernetzwerk z.B. Heimnetzwerk oder Unternehmensnetzwerk. Typischerweise verwenden die Netzwerke private Netzwerkadressen (IPv4).

MQTT Message Queuing Telemetry Transport: Ein offenes Netzwerkprotokoll, optimiert für die Übertragung von Nachrichten im IoT Umfeld.

NAT Network Address Translation: NAT wird verwendet, um private Netze mit dem Internet zu verbinden wobei private IP-Adressen auf eine einzige öffentliche IP-Adresse abgebildet werden.

Peer-to-Peer Verbindung zwischen zwei Netzwerk-Clients ohne Verwendung eines Servers.

PoC Proof of Concept: Belegt prinzipielle Durchführbarkeit eines Vorhabens. Positiver oder negativer Machbarkeitsnachweis.

publishen Publizieren im Sinne des englischen «to publish» im Zusammenhang mit MQTT.

SCTP Stream Control Transport Protocol: Ein Protokoll zur zuverlässigen und geordneten Paketübertragung basierend auf UDP.

SDP Session Description Protocol: Dient zur Beschreibung der Session-Informationen für eine WebRTC-Verbindung.

SRTP Secure Real-Time Transport Protocol: Wird verwendet zur verschlüsselten Übertragung von Audio- und Videodaten in Echtzeit.

STUN Session Traversal Utilities for NAT: Dient zur Ermittlung der Öffentlichen IP-Adressen und Ports einer UDP-Verbindung. Zusätzlich werden die vorhandenen NAT- und Firewall-Typen analysiert.

subscriben Abonnieren im Sinne des englischen «to subscribe» im Zusammenhang mit MQTT.

TCP Transmission Control Protocol: Ein zuverlässiges, verbindungsorientiertes Netzwerkprotokoll, welches IP als Vermittlungsschicht verwendet.

TLS Transport Layer Security: Ein Verschlüsselungsprotokoll zur Sicherung von TCP-Verbindungen.

Topic Hierarchische Einstufung von Nachrichten bei MQTT.

TURN Traversal Using Relays around NAT: Dient als Ausweichlösung, falls bei einer WebRTC-Verbindung keine direkte Kommunikation möglich ist. Die Datenpakete werden hierbei durch einen öffentlichen TURN-Server vermittelt.

UDP User Datagram Protocol: Ein verbindungsloses Netzwerkprotokoll, welches IP als Vermittlungsschicht verwendet.

VPN Virtual Private Network: Ein virtuelles privates Netzwerk, welches als Transportmedium ein bestehendes öffentliches Kommunikationsnetz verwendet.

WebRTC Web Real-Time Communication: Eine Sammlung von Kommunikationsprotokollen zur Echtzeitkommunikation zwischen zwei Client-Rechnern.

Appendix A: Test setup

1 Introduction

Our prototype application can create an application tunnel between two different LANs. The tunnel is established by a WebRTC data channel connection. A basic test setup with MQTT as application layer is shown on figure 1 with the minimum configuration in blue. A MQTT broker with listener client is hosted on a Raspberry pi by the students for the duration of the thesis and presentation (in green). Feel free to connect to these test broker. The public broker of HiveMQ is used for signaling.

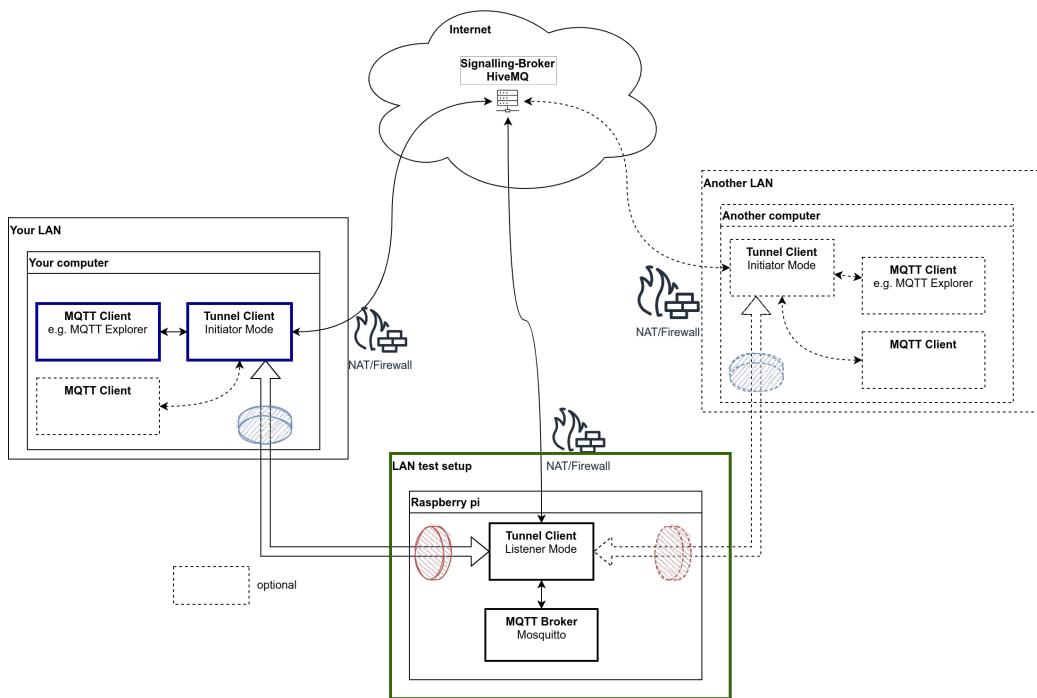


Abbildung 1: Test environment

2 Install and prepare Flatpak

The application tunnel software is provided as a flatpak package. Install flatpak¹ on your linux distribution and add flathub repository to download basic dependencies afterwards (listing 1).

```
1 flatpak --user remote-add --if-not-exists flathub  
https://flathub.org/repo/flathub.flatpakrepo
```

Listing 1: Add flathub repository

3 Install Application Tunnel

The flatpak of the application is provided on a github page. Add the corresponding repository and install the application according to the listing 2. Use the help function to get the information of the options set on the following commands.

```
1 flatpak --user remote-add flatpak-salvm4  
https://salvm4.github.io/flatpak-salvm4.flatpakrepo  
2 flatpak --user install flatpak-salvm4 ch.bfh.ti.applic-tunnel
```

Listing 2: Install application tunnel

4 Test the tunnel

4.1 Run initiator

The listener is identified by his public key. Run the initiator with the key mentioned in listing 3 to connect to the test setup listener. There is no initiator authentication activated on listener side. So every initiator can connect to the test setup.

```
1 flatpak run ch.bfh.ti.applic-tunnel initiator broker.hivemq.com 1883 127.0.0.1 1883  
RVC45WSASZAGKAJZT24NC3GTCEENN7FCZ03QC7Z5S2ZSQ0L22UXQ==== --notls
```

Listing 3: Run initiator client

The broker configured on the listener side is now accessible on the initiator side on the port 1883.

¹setup flatpak: <https://flatpak.org/setup/>

4.2 Connect to the broker

Test the connection to the broker. Use therefore a arbitrary MQTT client software e.q. «MQTT explorer»² (figures 2 and 3).

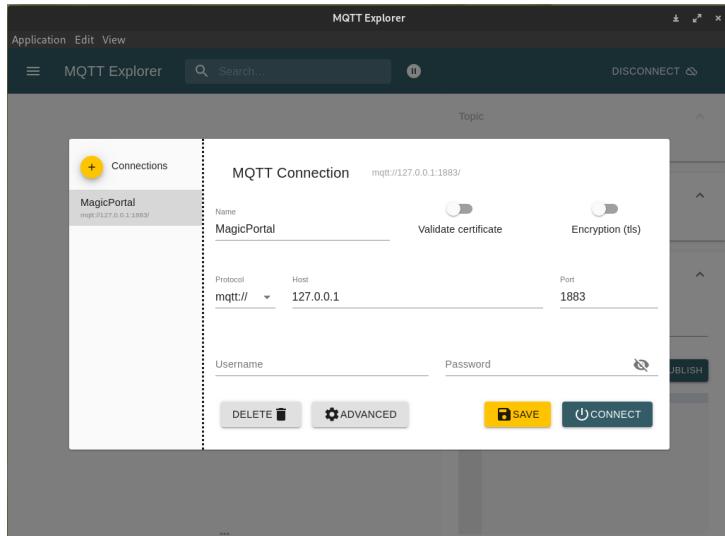


Abbildung 2: Connection settings

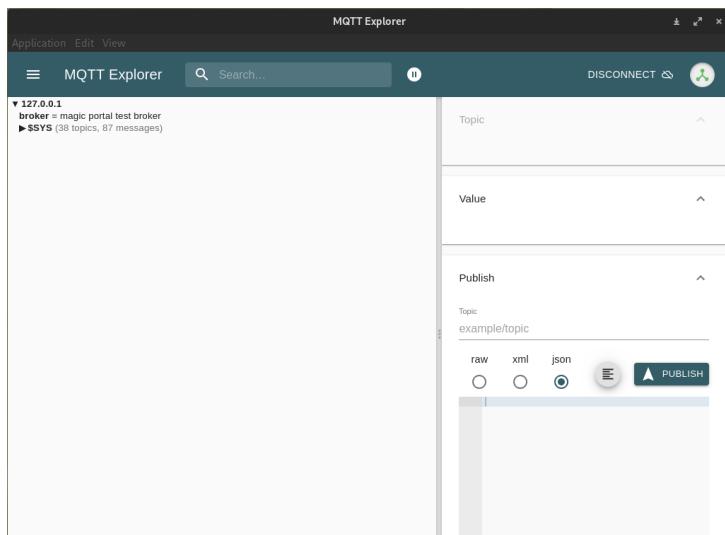


Abbildung 3: Connection open

²MQTT explorer: <https://mqtt-explorer.com/>