Master Thesis

im Rahmen des
Universitätslehrganges „Geographical Information Science & Systems“
(UNIGIS MSc) am Interfakultären Fachbereich für GeoInformatik (Z_GIS)
der Paris Lodron-Universität Salzburg

zum Thema

„Nah-Echtzeit von Sensordaten im Webumfeld“

Übertragung von Flugzeug-Positionsdaten über SOS und Websocket

vorgelegt von

BSc Stefan Scheuber
103116, UNIGIS MSc Jahrgang 2013

Zur Erlangung des Grades
„Master of Science (Geographical Information Science & Systems) – Msc(GIS)”

Gutachter:
Ao. Univ. Prof. Dr. Josef Strobl

Binningen (CH), 13.07.15

Eigenständigkeitserklärung

Ich versichere diese Master Thesis ohne fremde Hilfe und ohne Verwendung anderer als der angeführten Quellen angefertigt zu haben, und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen ist. Alle Ausführungen der Arbeit, die wörtlich oder sinngemäss übernommen wurden, sind entsprechend gekennzeichnet.

Binningen, 13.07.15 _____

Danksagung

An dieser Stelle möchte ich mich bei meinem Arbeitgeber der Firma Atos AG bedanken, welcher mich zu diesem Studium ermutigte und mir während des gesamten Studiums hilfreich zur Seite gestanden hat.

Des Weiteren möchte ich mich ganz herzlich bei Prof. Dr. Josef Strobel und beim UNIGIS-Team der Universität Salzburg für die grossartige Unterstützung und Betreuung während des gesamten Studiums bedanken.

Mein innigster Dank gilt allen voran meiner Freundin Alessandra Brenta sowie meiner Familie, welche mich während dem gesamten Studium unterstützt und mir den Rücken freigehalten hat, damit ich mich voll und ganz aufs Studium wie auch auf die Masterarbeit konzentrieren konnte.

Kurzfassung

Die vorliegende Arbeit befasst sich mit dem Vergleich von zwei möglichen Varianten, um Sensordaten von einer Serverapplikation zu einer Webanwendung zu übertragen. Der Fokus des Vergleichs besteht darin, die Echtzeitfähigkeit und die Performanz der beiden Varianten anhand von Flugzeug-Positionsdaten zu untersuchen.

Die Variante 1 wurde mit Hilfe eines Sensor Observation Services (SOS) umgesetzt, welches ein Standard des Open Geospatial Consortiums darstellt. Ein SOS stellt eine Programmierschnittstelle zur Verfügung, über welche Sensordaten eingefügt und wieder ausgelesen werden können. Die Variante 2 wurde mit Hilfe von Websockets umgesetzt, was einem relativ neuen Netzwerkprotokoll entspricht, welches im Rahmen der HTML5 Initiative entwickelt wurde. Websockets ermöglichen eine Zweiwegkommunikation und werden als besonders performant angesehen.

Um eine Vergleichsbasis schaffen zu können, wurde ein System entwickelt, welches generisch für die Übermittlung von Sensordaten verwendet werden kann. Das System besteht aus einer Serverapplikation und einer Webanwendung. Die Serverapplikation wurde mit der Programmiersprache Java entwickelt und ermöglicht es, Daten zu empfangen, diese aufzubereiten und über die beiden Varianten zu verteilen. Die Webanwendung wurde in JavaScript geschrieben und ist fähig, Sensordaten über die beiden Varianten zu empfangen und auf einer Kartenkomponente darzustellen. Das System wurde konfiguriert um das Rufzeichen, die Höhe, die Ausrichtung und die Position von Flugobjekten aus einem ADS-B Empfänger, welcher an das System angeschlossen ist, auszulesen und auf der Webanwendung darzustellen.

Die Vergleichsmessungen haben ergeben, dass beide Varianten zuverlässig die Daten übertragen. Unterschiede zeigten sich jedoch bei den quantitativen Aspekten der Datenübertragung. So ist bei der Websocket-Variante die Latenzzeit kleiner, die übertragene Datenmenge geringer und der maximale Durchsatz grösser als bei der SOS-Variante. Websockets überzeugten somit bei den Vergleichsmessungen in den Punkten Echtzeitfähigkeit und Performanz. Basierend auf diesen Ergebnissen, wird für die Übermittlung von Nah-Echtzeit-Sensordaten in einer Webanwendung der Einsatz von Websockets empfohlen.

Abstract

Within this work two different solutions are presented to transmit data from a server- to a browser application. The main focus of this work is to compare the real-time capability and the performance of said solutions in regards to aircraft position data.

The first solution is based on a Sensor Observation Service (SOS), which constitutes a standard that has been published by the Open Geospatial Consortium. The SOS provides an application programming interface which allows persisting and querying sensor data. The second solution is based on Websockets, which is a relatively new network protocol. The Websocket protocol has been developed as part of the HTML5-initiative and allows a bidirectional communication. The performance of Websockets is considered to be particularly well.

To create a basis for comparison of the two solutions, a system has been developed which allows a generic transmission of sensor data. The system consists of a server- and a web application. The server application has been developed with the programming language Java and is able to receive, convert and prepare data, to finally transmit it using the two solutions. The web application has been developed with the programming language JavaScript and is able to receive the data through both solutions and to visualize the received data within a map component. The system has been configured to transmit the call sign, altitude, heading and position of aircrafts and to visualize them on a web application. The data is captured from an ADS-B receiver which has been connected to the system.

The comparative measurements have shown, that both solutions transmit data reliably. The comparison has further shown, that the Websocket solution has a smaller latency, the amount of data transmitted is reduced and the maximum message rate is greater than that of the SOS solution. Thus, the comparative measurements has demonstrated, that the Websocket solution has advantages in real-time capability and performance. Based on these results, it is recommended to use Websockets for the transmission of near-real-time sensor data to web applications.

Inhaltsverzeichnis

Eigenständigkeitserklärung....II
Danksagung....III
Kurzfassung....IV
Abstract....V
Inhaltsverzeichnis....VI
Abbildungsverzeichnis....VIII
Tabellenverzeichnis....IX
Abkürzungsverzeichnis....X
1. Einleitung....1
1.1 Motivation und Problemstellung....1
1.2 Forschungsfragen....2
1.3 Gliederung der Arbeit....3
2. Theoretische Grundlagen....4
2.1 ADS-B....4
2.2 Kommunikationsprinzipien....5
2.3 Technologie Übersicht....7
2.4 Datenformate....17
3. Konzeption des Systems....22
3.1 Zieldaten....22
3.2 Verwendung der Technologien....22
3.3 Übersicht Gesamtsystem....31
4. Umsetzung Integrationssystem....33
4.1 Verwendete Java-Bibliotheken....34
4.2 Domänendaten....37
4.3 Datenempfangs-Route....39
4.4 SOS-Route....45
4.5 Websocket-Route....52
5. Umsetzung Webclient....56
5.1 Verwendete JavaScript Bibliotheken....56
5.2 Allgemeiner Aufbau....58
5.3 Client Service....60
5.4 Map Controller....78
5.5 View....86
5.6 Style Service....86
5.7 MenuController....87
6. Versuchsaufbau....88
6.1 Netzwerk Switch....88
6.2 NTP-Server....89
6.3 ADS-B Empfang....89
6.4 Integrationssystem....91
6.5 SOS-Server....91
6.6 Webclient....92
7. Vergleichsmessungen....95
7.1 Übertragungsqualität....95
7.2 Minimale Latenzzeit....100
7.3 Übertragungsmenge....103
7.4 Meldungsdurchsatz....107
8. Schlussfolgerungen....111

8.1 Beantwortung der Forschungsfragen....111
8.2 Weitere Einsatzszenarien....113
8.3 Schlussfolgerungen und Ausblick....114
9. Literaturverzeichnis....116
A Anhang....119
A.1 InsertSensor....119
A.2 Bean-Konfiguration SOS-Variante....121
A.3 Integrationssystem Konfigurationsdatei....123
A.4 Client Konfigurationsdatei....124

Abbildungsverzeichnis

Abbildung 1: Sequenzdiagramm Pull....6
Abbildung 2: Sequenzdiagramm push-Kommunikation....7
Abbildung 3: Mögliches SOS-Szenario....13
Abbildung 4: Websocket Meldungsaufbau (Quelle: websocket.org)....16
Abbildung 5: Typen von Observation (Quelle: Cox, 2011)....19
Abbildung 6: SOS konzeptionelle Übersicht....23
Abbildung 7: Websocket konzeptionelle Übersicht....29
Abbildung 8: Konzeptionelle Übersicht Gesamtsystem....32
Abbildung 9: Übersicht Routingsystem....33
Abbildung 10: Klasse SBS1Message....37
Abbildung 11: Klassendiagramm Feature....38
Abbildung 12: Ablauf Datenempfangs-Route....40
Abbildung 13: Ablauf SOS-Route....46
Abbildung 14: Insert Observation SOS Konfiguration....49
Abbildung 15: ObservedPropertyConfiguration Implementierungen....50
Abbildung 16: Ablauf Websocket-Route....52
Abbildung 17: Client Übersicht....59
Abbildung 18: Client Service - Controller Kommunikation....62
Abbildung 19: Sequenzdiagramm SOS-Client Service....66
Abbildung 20: Ablaufdiagramm Client SOS Service....67
Abbildung 21: Client SOS-Service Verarbeitung neuer Daten....70
Abbildung 22: Sequenzdiagramm Websocket-Client Service....73
Abbildung 23: Ablaufdiagramm Client Websocket-Service....74
Abbildung 24: Client Websocket-Service Verarbeitung neuer Daten....76
Abbildung 25: Styling Flugobjekte....87
Abbildung 26: Komponenten des Testsystems....88
Abbildung 27: Benutzeroberfläche SOS-Variante....94
Abbildung 28: Benutzeroberfläche Websocket-Variante....94
Abbildung 29: Histogramm SOS Latenzzeit....101
Abbildung 30: Histogramm Websocket Latenzzeit....102
Abbildung 31: SOS Übertragungsmenge....104
Abbildung 32: Websocket Übertragungsmenge....105
Abbildung 33: Gesendete Datenmenge pro Variante....105
Abbildung 34: Übertragunsmenge Webclient....106
Abbildung 35: SOS Durchsatz....108
Abbildung 36: Websocket Durchsatz (Messung 1)....109
Abbildung 37: Websocket Durchsatz (Messung 2)....110

Bei Abbildungen ohne Quellenangabe handelt es sich um eigene Darstellungen.

Tabellenverzeichnis

Tabelle 1: OGC-Standards der SWE-Initiative (Botts et al., 2008).....9
Tabelle 2 OGC Begriffserläuterung (Quelle: Wikipedia).....10
Tabelle 3: SOS Basis-Service Funktionen.....11
Tabelle 4: SOS Enhanced Extension Funktionen.....12
Tabelle 5: SOS Transactional Extension Funktionen.....12
Tabelle 6: SOS Result Handling Extenstion Funktionen.....12
Tabelle 7: SOS Funktionalitäten.....14
Tabelle 8: Websocket Anwendungsbeispiele.....16
Tabelle 9: SBS-1 Properties.....18
Tabelle 10: Geometrietypen GeoJSON.....20
Tabelle 11: FeatureTypen GeoJSON.....21
Tabelle 12: Webclient Zieldaten.....22
Tabelle 13: SOS Sensor Eigenschaften.....24
Tabelle 14: SOS Sensor Properties.....24
Tabelle 15: SOS Result Typen.....27
Tabelle 16: GeoJSON Zieldaten Properties.....30
Tabelle 17: Auszug Camel Komponenten.....35
Tabelle 18: Konfigurationsmöglichkeiten Datenempfangsroute.....43
Tabelle 19: Konfigurationsmöglichkeiten SOS-Route.....48
Tabelle 20: SOS Property Konfiguration Datentypen.....50
Tabelle 21: Konfigurationsmöglichkeiten Websocket-Route.....54
Tabelle 22: OpenLayers 3 unterstützte Datenformate.....57
Tabelle 23: Client Basis-Service Attribute.....61
Tabelle 24: Basis-Service implementierte Funktionen.....63
Tabelle 25: Ablauf Observer Pattern im Client Service.....64
Tabelle 26: Client Basis-Service Funktionen Schnittstelle.....64
Tabelle 27: Konfigurationsmöglichkeiten Client SOS-Service.....72
Tabelle 28: Konfigurationsmöglichkeiten Websocket Client-Service.....78
Tabelle 29: Map Controller Initialisierungsvariablen.....79
Tabelle 30: Konfigurationsmöglichkeiten Applikation.....84
Tabelle 31: Konfigurationsmöglichkeiten OpenLayers3-Komponente.....85
Tabelle 32: Konfigurationsmöglichkeiten Gesamtapplikation.....86
Tabelle 33: Style Service Methodendefinition.....87
Tabelle 34: NTP-Server Hardware.....89
Tabelle 35: Port Konfiguration ADS-B Empfänger (dump1090).....90
Tabelle 36: Hardware Integrationssystem.....91
Tabelle 37: Hardware SOS-Server.....91
Tabelle 38: Hardware Clientsystem.....92
Tabelle 39: Ergebnisse Übertragunsmenge SOS.....104
Tabelle 40: Ergebnisse Übertragungsmenge Websocket.....105

Bei Tabellen ohne Quellenangabe handelt es sich um eigene Darstellungen.

Abkürzungsverzeichnis

Abkürzung	Bedeutung
ADS-B	Automatic Dependent Surveillance-Broadcast
API	Application Programming Interface
DOM	Domain Object Model
DSL	Domain Specific Language
DVB-T	Digital Video Broadcasting – Terrestrial
EDA	Event-driven architecture
FTP	File Transfer Protocol
GeoJSON	Geographical JavaScript Object Notation
GIS	Geoinformationssystem
GML	Geography Markup Language
GPS	Global Positioning System
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IETF	Internet Engineering Task Force
IoC	Inversion of Control
JAXB	Java Architecture for XML Binding
JMS	Java Message Service
JSON	JavaScript Object Notation
KML	Keyhole Markup Language
KVP	Key Value Pairs
LAN	Local Area Network
MVC	Model - View - Controller
MVVM	Model - View - ViewModel
MVW	Model - View - Whatever
O&M	Observations and Measurements
OGC	Open Geospatial Consortium
RFC	Request for Comment
SDR	Software Defined Radio
SOA	Service Oriented Architecture (Serviceorientierte Architektur)
SOAP	Simiple Object Access Protocol
SOS	Sensor Observation Service
SQL	Structured Query Language
SWE	Sensor Web Enablement
TCP	Transmission Control Protocol
TRL	Technology Readiness Levels
UCUM	Unified Code for Units of Measure
URI	Uniform Resource Identifier
USB	Universal Serial Bus
WFS	Web Feature Service
WMS	Web Map Service
W3C	World Wide Web Consortium
XML	Extensible Markup Language

1. Einleitung

In diesem einführenden Kapitel wird erläutert, was die Motivation für das Erarbeiten dieser Master Thesis ist und welche Forschungsfragen mit dieser Arbeit beantwortet werden sollen. Im letzten Unterkapitel wird die Gliederung dieser Masterarbeit beschrieben.

1.1 Motivation und Problemstellung

In einer Zukunftsprognose im Jahre 2011 hat Michael F. Goodchild die Echtzeit-Verarbeitung und -Darstellung von ortsbezogenen Daten als einen möglichen Trend von Geoinformationssystemen (GIS) aufgezählt (Goodchild, 2011). Goodchild schreibt von einem möglichen Wandel im GIS-Umfeld, bei welchem man vom Analysieren von statischen Daten in Richtung dynamischer Verarbeitung von Daten, die für Echtzeit-Entscheidungsfindung und -Überwachung verwendet werden können, übergeht.

Verschiedene Technologien wurden entwickelt, welche hinsichtlich einer Echtzeit-Verarbeitung von Daten eingesetzt werden können. Eine dieser Technologien ist der Sensor Observation Service (SOS), für welchen im Jahre 2007 die erste Spezifikation veröffentlicht wurde (Na and Priest, 2007). Ein SOS stellt eine spezifische Schnittstelle für das Persistieren und Laden von Sensorbeschreibungen und Sensordaten zu Verfügung. Die Möglichkeiten um Sensorbeschreibungen oder Sensordaten über den Dienst zu persistieren, werden mit diesem Dienst sehr generell gehalten. Jirka und Bredel haben im Jahr 2011 ein Nah-Echtzeitsystem vorgestellt, welches den SOS als zentrale Komponente einsetzt, um Positionen von Schiffen in einer Webanwendung über einen Web Map Service (WMS) darstellen zu können (Jirka and Bredel, 2011).

Eine andere Technologie, welche für die Übertragung von Echtzeitdaten geeignet ist, ist das Websocket-Protokoll. Die Spezifikation des Websocket-Protokolls wurde im Jahre 2011 veröffentlicht (Fette and Melnikov, 2011). Durch den Einsatz des Websocket-Protokolls wird eine Zweiwegkommunikation zwischen einem Server und einer Clientanwendung ermöglicht. Der Server kann somit jederzeit aktuelle Daten direkt zum Client senden, ohne dass dieser erst eine Anfrage tätigen muss. Websockets können in Webanwendungen verwendet werden und zeichnen sich dort als besonders performante Lösung gegenüber anderen Webtechnologien aus (Puranik et al., 2013). Pimentel und Nickerson haben im Jahre 2012 eine Websocket Lösung vorgestellt, mit welcher Echtzeitdaten eines Windsensors über das Websocket-Protokoll übertragen werden (Pimentel and Nickerson, 2012).

Für die Visualisierung von Nah-Echtzeitdaten bietet eine Browserlösung grosse Vorteile. Denn durch das Internet kann überall auf der Welt ohne spezielle Installation die Anwendung aufgerufen und ausgeführt werden. Da die Entwicklung von Webtechnologien rasant fortschreitet, ist es heutzutage auch möglich, Webanwendungen zu schreiben, welche vom

Verhalten und Funktionsumfang her Desktop Applikationen ähneln (Fraternali et al., 2010).

In der vorliegenden Arbeit soll ein System entwickelt werden, in welchem Flugzeug-Positionsdaten zu einer Webanwendung übertragen werden, wo diese anschliessend auf einer Kartenkomponente dargestellt werden. Dieses System soll für die Übertragung der Daten vom Serversystem zum Client auf zwei unterschiedlichen Varianten basieren. Die Übertragung soll in einer Variante über einen SOS und bei der anderen Variante durch die Kommunikation über das Websocket-Protokoll erfolgen.

Die Flugzeug-Positionsdaten werden durch Automatic Dependent Surveillance Broadcast (ADS-B) Meldungen direkt von Flugzeugen empfangen. Durch Umwandlung der empfangenen Daten sollen Messwerte wie Höhe, Geschwindigkeit, Richtung sowie die Position ausgelesen und über die beiden Varianten parallel übertragen werden. Mit Hilfe dieses Systems soll es möglich sein, die beiden Varianten miteinander zu vergleichen. Durch einen Parallelbetrieb soll eine Aussage über die Zuverlässigkeit der beiden Varianten gemacht werden können. Zudem sollen die Nah-Echtzeit Möglichkeiten, der maximale Meldungsdurchsatz und die Übertragungsmenge verglichen werden. Diese Vergleichswerte sollen dazu genutzt werden, eine Empfehlung geben zu können, in welchem Fall die eine oder die andere Variante bevorzugt werden soll.

1.2 Forschungsfragen

Wie in vorangehendem Unterkapitel erwähnt, soll in dieser Arbeit eine Lösung für die Übermittlung von Flugzeug-Positionsdaten sowohl über einen SOS wie auch über Websockets sowie deren Darstellung in einer Webanwendung erarbeitet werden. Um die Varianten möglichst vergleichbar zu machen, soll die Abweichung von der Codebasis lediglich beim serverseitigen Versenden und clientseitigen Empfangen unterschiedlich sein. Mit Hilfe von verschiedenen Vergleichsmessungen sollen die beiden Varianten verglichen werden um folgende Fragen zu klären:

- Können die Sensordaten zuverlässig über beide Varianten übermittelt und auf einem Webclient dargestellt werden?
- Wie wirkt sich der Einsatz von Websockets für die Kommunikation zwischen Server und Webclient auf die Echtzeitfähigkeit gegenüber der Kommunikation über einen SOS aus?
- Gibt es erhebliche Unterschiede bei der übertragenen Datenmenge beim Einsatz von Websockets gegenüber dem Einsatz eines SOS?
- Wie gross ist der maximale Durchsatz von ADS-B Meldungen, die mit den beiden Varianten übertragen werden können?

Diese Fragen sollen Aufschluss darüber geben, in welchen Fällen bei der Übertragung von

Nah-Echtzeitdaten der Einsatz von Websockets gegenüber dem Einsatz von einem SOS bedacht werden soll. Es ist davon auszugehen, dass durch die direkte Kommunikation zwischen Server und Client die Websocket-Variante erhebliche Performance-Vorteile gegenüber der SOS-Variante aufweist.

1.3 Gliederung der Arbeit

Zu Beginn der Arbeit wird auf die theoretischen Grundlagen der beiden Varianten eingegangen. Dabei werden Technologien und Datenformate beschrieben, welche innerhalb des Zielsystems eingesetzt werden. Im darauf folgenden Kapitel 3 wird konzeptuell vorgestellt, wie die Kommunikation innerhalb der beiden Varianten erfolgen soll und in welcher Form die Daten übertragen werden sollen. In Kapitel 4 wird die serverseitige Umsetzung der beiden Varianten, das Integrationssystem vorgestellt. Hierbei werden der generelle Aufbau wie auch einzelne Implementationspezifika erörtert. Kapitel 5 beschreibt, wie der Webclient aufgebaut ist, so dass sowohl über die SOS- wie auch über die Websocket-Variante Daten empfangen werden können.

In Kapitel 6 wird aufgezeigt, wie das Testsystem aufgebaut ist, welches für die Vergleichsmessungen verwendet wird. Es wird beschrieben, wie die zuvor erläuterten Komponenten eingesetzt und welche weiteren Netzwerkkomponenten benötigt werden, damit ADS-B Daten über beide Varianten verteilt werden können. Dieses Testsystem wird in Kapitel 7 für verschiedene Vergleichsmessungen bezüglich der Qualität und der Performance der Datenübertragung eingesetzt. Im letzten Kapitel werden Schlussfolgerungen aus den Vergleichsmessungen und aus den gewonnen Erkenntnissen gezogen und zusammengefasst dargestellt.

2. Theoretische Grundlagen

In diesem Kapitel werden die dieser Arbeit zu Grunde liegenden Kommunikationsprinzipien und die verwendeten Technologien erläutert. In Unterkapitel 2.1 wird beschrieben, über welche Technologie die Flugpositionsdaten von den Flugobjekten verbreitet werden und mittels welchen Methoden diese empfangen werden können. Im darauf folgenden Unterkapitel werden die Kommunikationsprinzipien und -abläufe der SOS- und der Websocket-Variante erläutert. In Unterkapitel 2.3 werden die Grundlagen des Websocket-Protokolls wie auch des SOS erläutert. Im letzten Unterkapitel der theoretischen Grundlagen wird auf die Datenformate, welche für die Übertragung verwendet werden, eingegangen.

2.1 ADS-B

Die in dieser Arbeit verwendeten Flugzeug-Positionsdaten werden mittels Automatic Dependent Surveillance-Broadcast (ADS-B) gesammelt. ADS-B ist ein Flugsicherungssystem bei welchem ein Flugobjekt die eigene Position über das Global Positioning System (GPS) bestimmt und über einen Transmitter publiziert. Diese Informationen können von anderen Flugzeugen oder von Bodenstationen empfangen und zur Situationseinschätzung verwendet werden. Ein grosser Vorteil von ADS-B gegenüber etablierten Luftüberwachungssystemen ist, dass ADS-B kosteneffektiv in Flugzeugen installiert und in Betrieb genommen werden kann (Richards et al., 2010). Das Versenden der Daten kann über einen umgerüsteten Mode-S Transmitter erfolgen, welcher bei zivilen Maschinen vorhanden und in Betrieb sein muss. Das Empfangen der Daten kann im Gegensatz zu etablierten Technologien auch einfacher und günstiger erfolgen (Schäfer et al., 2014). Da die ADS-B Daten unverschlüsselt veröffentlicht werden, können diese ohne grossen Aufwand abgefangen und decodiert werden (Costin and Francillon, 2012). Dies stellt einen gehörigen Nachteil in der Sicherheit dieses Systems dar, da dadurch Ansätze geboten werden, wie dieses System gestört oder gar manipuliert werden kann (Strohmeier et al., 2013).

Die grosse Neuerung von ADS-B gegenüber etablierten Luftüberwachungssystemen liegt in der Art, wie die Daten für die Luftraumüberwachung beschafft werden. Zu Beginn der Luftraumüberwachung wurden Primärradare für das Detektieren der Position von Flugzeugen eingesetzt. Über die Reflektion von Wellen der Flugzeuge wird hierbei die Position eines Flugobjekts bestimmt. Aktuell werden hauptsächlich Sekundärradare für das Beschaffen von Flugzeuginformationen eingesetzt. Sekundärradare senden im Unterschied zu Primärradaren Meldungen an die Flugobjekte, welche mittels installierter Transpondern (Transmitter-Respondern) dem Sender auf dessen Anfrage antworten. Hierdurch können die Position und weitere Informationen über das Flugobjekt bestimmt werden. Die Bodenstation muss jedoch dem Transponder des Flugobjektes vertrauen, dass dieser korrekt auf eine Anfrage antwortet (Costin and Francillon, 2012).

ADS-B sendet im Gegensatz zu einem Sekundärrader in einem Intervall (in der Regel alle 0.5 Sekunden) einen eindeutigen Identifizierungscode, die Positionsdaten und weitere Informationen auf der Frequenz 1090 MHz aus, ohne dass eine Anfrage an das Flugzeug gesendet werden muss. ADS-B Daten können bei optimalen Bedingungen bis zu einer Distanz von mehreren 100 km empfangen werden (Schäfer et al., 2014).

Im Jahre 2013 waren bereits 70-80 Prozent der kommerziellen Flugzeuge mit ADS-B Transponders ausgestattet. Ab dem Jahre 2015 müssen alle neuen Flugzeuge im europäischen Luftraum mit ADS-B ausgestattet sein. Ältere Flugzeuge müssen zwingend bis im Jahre 2017 nachgerüstet werden (Strohmeier et al., 2013). Im amerikanischen Luftraum muss ADS-B erst ab 2020 zwingend bei jedem Flugzeug in Betrieb sein (Command, 2012).

2.2 Kommunikationsprinzipien

In dieser Arbeit werden zwei verschiedene Varianten verwendet um Sensordaten zu übertragen. Die beiden Varianten unterscheiden sich grundlegend im technologischen Aufbau und im Speziellen in der Art, wie die Meldungen durch das System geschleust werden. Die beiden Varianten basieren auf zwei verschiedenen Kommunikationsprinzipien. Die SOS-Variante basiert auf dem *pull*-Prinzip, wohingegen die Websocket-Variante auf dem *push*-Prinzip basiert.

2.2.1 Pull-Prinzip

Als eine *pull* basierte Kommunikation wird ein Kommunikationsprinzip beschrieben, bei welchem der Empfänger die Kommunikation initiiert. Der Empfänger sendet bei dieser Technik eine Anfrage oder Meldung an einen Dienst, welcher auf die Anfrage antwortet (Deriu, 2011).

Typischerweise wird diese Art von Kommunikation in Webservices verwendet, welche insbesondere in serviceorientierten Architekturen (SOA) eingesetzt wird (Chandy and Schulte, 2007). SOA ist eine Softwarearchitektur, die unabhängige und wiederverwendbare Dienste mit einem allgemein verständlichen Protokoll anbietet. Verschiedene Dienste können zu neuen Diensten gekoppelt werden (Orchestrierung), oder direkt von einem Klienten aufgerufen werden. In vielen Fällen bieten die Dienste verschiedene Methoden an, um Daten zu erzeugen, abzufragen, zu verändern oder zu löschen. Die Aufrufe auf einen Dienst werden gemäss des *pull*-Prinzips ausgeführt.

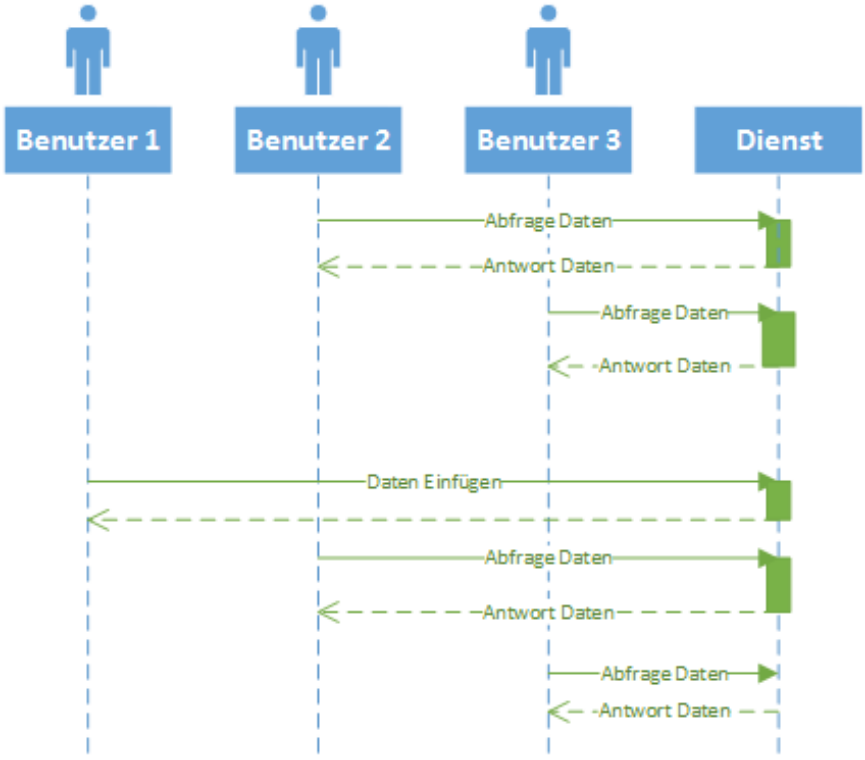

Abbildung 1: Sequenzdiagramm Pull

In Abbildung 1 wird anhand eines Sequenzdiagrammes ein Beispiel dargestellt wie die Kommunikation mit einem Webservice und drei Benutzern ablaufen kann. In diesem Beispiel sind Benutzer 2 und Benutzer 3 an aktuellen Daten interessiert und senden in einem bestimmten Intervall Anfragen an den Dienst um die neuesten Daten zu erhalten. Diese Methode um aktuelle Daten zu erhalten bezeichnet man als *polling*. In etwa der Hälfte der Sequenz fügt der Benutzer 1 Daten in den Dienst ein, die Benutzer 2 und 3 werden jedoch nicht automatisch über die neuen Daten informiert, weshalb diese beiden Benutzer erst nach dem nächsten Abfrageintervall die neuen Daten erhalten.

2.2.2 Push-Prinzip

Im Gegensatz zum *pull*-Prinzip werden beim *push*-Prinzip Informationen an den Empfänger übertragen, ohne dass dieser die Daten aktiv anfordern muss (Deriu, 2011).

Das Push-Prinzip spielt in der Informationstechnik bei der eventbasierten Architektur (EDA) eine zentrale Rolle. In einer eventbasierten Architektur wird üblicherweise eine Statusänderung (ein Event) mittels des *push*-Prinzips asynchron zu einem Event Processing System geschickt, welches diese verarbeitet. Man spricht hierbei von einer Notifikation. Wichtig ist, dass das Event Processing System nicht weiss, wann eine Meldung an das System gesendet wird und im Falle einer Event Notifikation diese sofort verarbeitet (Chandy and Schulte, 2007). Ein Empfänger muss sich initial bei einem Dienst registrieren, um von diesem

automatisch Daten zu erhalten.

In Abbildung 2 wird ein Sequenzdiagramm einer eventbasierten Architektur dargestellt, welche auf dem *push*-Prinzip basiert. Die Benutzer 2 und 3 registrieren sich initial bei einem zentralen Dienst. Nach der Registrierung werden Benutzer 2 und 3 automatisch über die von Benutzer 1 erzeugten Events benachrichtigt (Notifikation). Die Benutzer 2 und 3 werden so lange über Events notifiziert, bis sich diese vom Dienst wieder deregistrieren.

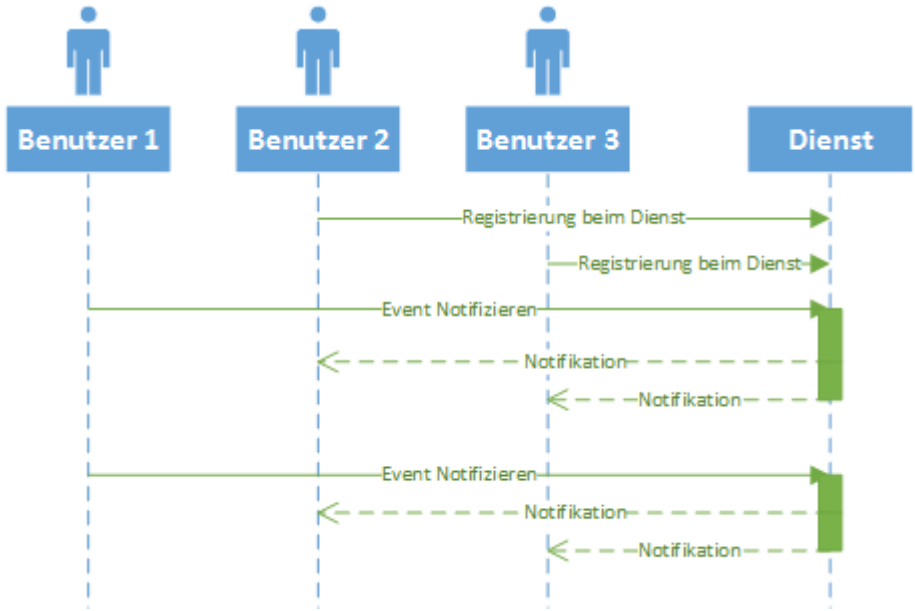

Abbildung 2: Sequenzdiagramm push-Kommunikation

2.3 Technologie Übersicht

Innerhalb dieses Unterkapitels werden die Technologien erläutert, welche für die Übertragung der Daten vom Serversystem zum Webclient verwendet werden. Für die Variante 1 ist dies der Sensor Observation Service (Unterkapitel 2.3.2), welcher im Rahmen der Sensor Web Enablement (Unterkapitel 2.3.1) entstanden ist. In Kapitel 2.3.3 wird auf das Websocket-Protokoll eingegangen, welches in der Variante 2 eingesetzt wird.

2.3.1 Sensor Web Enablement

Das Ziel vom Sensor Web liegt darin, möglichst viele Sensoren an ein Netzwerk anzuschliessen und die Daten für Applikationen zur Verfügung zu stellen. Dadurch, dass sich Sensor Technologien konstant verbessern, die Geräte kleiner, günstiger, intelligenter und energieeffizienter werden, werden Sensoren in immer weiteren Bereichen eingesetzt. Beispiele hierfür sind:

- Katastrophenmanagement
- Umweltüberwachung

- Präzisionslandwirtschaft
- Frühwarnsysteme
- Öffentliche Sicherheit
- Gesundheitsüberwachung

(Bröring et al., 2011)

Durch die unterschiedlichen Einsatzgebiete der Sensoren, den verschiedenen Arten von Sensoren und den Netzwerkprotokollen, welche sich teilweise stark unterscheiden, wurde vom Open Geospatial Consortium (OGC) die Sensor Web Enablement (SWE) Initiative ins Leben gerufen. Mit Hilfe der SWE-Initiative sollen Web zugängliche Sensor-Netzwerke und archivierte Sensordaten mithilfe von offenen Standards aufgefunden, den Zugriff auf die Daten und Sensoren sichergestellt und wenn möglich applikatorisch kontrolliert werden können. Dadurch soll sichergestellt werden, dass der Zugriff auf die Sensoren und Sensordaten mit offenen Standards vereinheitlicht wird. Die nachfolgende Tabelle enthält die Standards, welche der SWE-Initiative angehören:

Name	Abkürzung	Beschreibung
Observations & Measurements Schema	O&M	Dieses XML-Schema beschreibt Modelle, um Beobachtungen und Messungen von einem Sensor darzustellen (vgl. Unterkapitel 2.4.2).
Sensor Model Language	SensorML	Mithilfe dieses XML-Schemas können Sensorsysteme und Prozesse beschrieben werden. Es können Informationen zur Verfügung gestellt werden, um das Auffinden von Sensoren, die Position von Sensor-Beobachtungen, die Verarbeitung von low-level Sensor-Beobachtungen und die Auflistung von planbaren Fähigkeiten zu ermöglichen.
Transducer Markup Language	TransducerML oder TML	Ein konzeptionelles Model und XML Schema um Wandler zu beschreiben. Zudem wird das Echtzeit Streaming von Daten zu und von Sensor-Systemen unterstützt.
Sensor Observation Service	SOS	Dieser Standard beschreibt ein Standard Web Service Interface für das Anfragen, Filtern und Abrufen von Sensor-System-Informationen oder -Beobachtungen. Ein SOS kann als Vermittler zwischen einem Client und einer Sensordatenquelle oder einem Nah-Echtzeit Datenstrom verwendet werden.
Sensor Planning Service	SPS	Dieser Standard beschreibt ein Standard Web Service Interface für benutzergesteuerte

Name	Abkürzung	Beschreibung
		Anfragen und Beobachtungen. Ein SPS kann zwischen einem Client und einem Sensor Collection Management-Umfeld verwendet werden.
Sensor Alert Service	SAS	Dieser Web Service Standard kann für das Veröffentlichen oder zur Benachrichtigung von Alarmen von Sensoren verwendet werden.
Web Notification Services	WNS	Standard Web Service Interface für das asynchrone Übermitteln von Meldungen oder Alarmen von einem SAS oder SPS oder anderen Service Workflows.

Tabelle 1: OGC-Standards der SWE-Initiative (Botts et al., 2008)

Die SWE Standards sind bereits in der Version 2 verfügbar, wobei jedoch folgendes beachtet werden muss: Die Version 1 der SWE-Standards wurde bereits mehrfach von verschiedenen Institutionen implementiert und eingesetzt. Gemäss OGC ist die Version 1 genügend gereift um in einer produktiven Umgebung eingesetzt werden zu können und hat deshalb den höchsten technologischen Reifegrad (TRL-9) erreicht. Der technologische Reifegrad sagt aus, wie weit eine Technologie fortgeschritten ist, wobei TRL-1 die niedrigste technologische Reife aufweist. TRL-9 bezeichnet die höchste technologische Reife und beschreibt Systeme, die bereits erfolgreich umgesetzt und erfolgreich in wichtigen Systemen in Betrieb sind (Mankins, 1995). Die Version 2 der verschiedenen OGC-Standards wurde bereits implementiert und wird teilweise auch schon eingesetzt. Das OGC gibt den Reifegrad für diese Versionen mit TRL-6 an, was einem erfolgreichen Einsatz eines Systems oder Subsystem mit einem Prototypen in einem relevanten Umfeld entspricht (Hobona et al., 2013).

Das OGC verwendet im Rahmen des SOS und weiteren Projekten standardisierte Begriffe. An verschiedenen Stellen in dieser Arbeit werden diese Begriffe entsprechend verwendet. Die Verwendung in dieser Arbeit stützt sich auf die Interpretationen des OGC, welche in nachfolgender Tabelle aufgeführt sind:

Begriff	Beschreibung
Feature of Interest (FOI)	Das ~ repräsentiert das Geoobjekt, für das die Messwerte gelten und das von Sensoren gemessen wird. Über das FeatureOfInterest erfolgt in der Regel die Verortung (Georeferenzierung) der Messpunkte, d. h. das Geoobjekt besitzt Koordinaten (z. B. Länge/Breite und Höhe über NN). Die Festlegung des FOI hängt sehr stark vom Projekt ab und muss je nach Struktur gewählt werden.
Observation	Eine ~ liefert einen Messwert (Result) für die Eigenschaft (Phenomenon) eines observierten Objekts (FeatureOfInterest). Der Wert selbst wird durch einen Sensor oder Prozeduren erzeugt (Procedure). Ferner wurde das Phänomen zu einem bestimmten Zeitpunkt erfasst (SamplingTime) und der Wert an einem bestimmten Zeitpunkt erzeugt (ResultTime). Häufig stimmen die Werte überein, weshalb in der Praxis dann die SamplingTime als Zeitpunkt der Observation verwendet wird.
Offering	Ein ~ ist eine logische Gruppierung von miteinander in Bezug stehenden Observationen, welche gemeinsam von einem Dienst angeboten werden.
Phenomenon	Ein ~ (Phänomen) stellt eine Eigenschaft (physikalische Größe) eines Geoobjekts dar. (Lufttemperatur, Windgeschwindigkeit, Schadstoffkonzentration der Atmosphäre, Reflektierte Strahlung in bestimmten Frequenzband, etc.)
Procedure	Eine ~ (Prozedur) erzeugt den Messwert einer Observation. Dieses kann durch das Auslesen eines Sensors, Simulation oder auch einen numerischen Prozess geschehen.

Tabelle 2 OGC Begriffserläuterung (Quelle: Wikipedia)

2.3.2 Sensor Observation Service

Der SOS spielt in dieser Arbeit eine zentrale Rolle. In der ersten Variante wird dieser als zentrale Komponente für die Übertragung der ADS-B Daten eingesetzt.

In diesem Unterkapitel werden zum einen die Service-Funktionen erläutert, die in der SOS-Spezifikation enthalten sind. Zum anderen wird aufgezeigt, wie ein Server welcher einen SOS anbietet aufgebaut ist. Zum Ende des Unterkapitels wird die Funktionalität der SOS-Referenzimplementierung aufgelistet.

Service Funktionen

Die Spezifikation des SOS in der Version 1 wurde vom OGC im Jahre 2007 veröffentlicht und beschreibt einen Webservice, welcher eine Programmierschnittelle (API) für das Verwalten von und Zugreifen auf Sensordaten, im Speziellen von Beobachtungsdaten, anbietet (Na and Priest, 2007). Der SOS wurde so konzipiert, dass sowohl Sensoren die fix verortet (in-situ) sind (z.B. Wasserüberwachung) oder dynamische/mobile Sensoren (z.B. Satellitenbilder) unterstützt werden. Um Daten von einem SOS auslesen zu können, müssen Funktionen auf dem Service aufgerufen werden. Die Kommunikation erfolgt also nach dem Pull-Prinzip (vgl. Unterkapitel 2.2.1). Na und Priest spezifizierten in der Version 1 unter anderem folgende Funktionen:

Basis-Service Funktionen

Die Basisfunktionalität des SOS umfasst drei Funktionen, welche mittels HTTP (Hypertext Transfer Protocol) über Anfragen aufgerufen werden können. Die Requests werden als HTTP-

POST-Anfragen übertragen und tragen die Anfrage im Anfragekörper. Ein POST Request ist eine HTTP-Anfragemethode, bei welcher neben dem HTTP-Header zusätzlich Daten im Anfragekörper übertragen werden können. HTTP-POST Requests werden üblicherweise für das Hochladen von Dateien oder Übermitteln von Formularen verwendet.

Folgende drei Basisfunktionen, bei welchen es sich um Abfragen handelt die nur lesend auf den SOS zugreifen, müssen zwingend von jeder SOS-Server Instanz gemäss Spezifikation angeboten werden:

Funktion	Beschreibung
GetCapabilities	Wie auch andere OGC Standards, wie z.B. der Web Feature Service (WFS) oder der Web Map Service (WMS), umfasst auch der SOS die *GetCapabilities*-Funktion. Mit der *GetCapabilities*-Funktion kann ein Client Metadaten über den zugegriffenen Service anfordern. Die Antwort beinhaltet eine Beschreibung des Services in zwei Bereichen: Der *FilterCapabilities*-Bereich gibt an, was der Server für Query Parameter unterstützt. Der *Contents*-Bereich beschreibt die Daten, die vom SOS angeboten werden (Prozeduren, Sensoren, etc.).
DescribeSensor	Mit Hilfe der *DescribeSensor*-Funktion können detaillierte Metadaten über einen Sensor abgefragt werden. Für das Abfragen der detaillierten Sensor-Metadaten muss der Sensor Identifier als Parameter der Funktion übergeben werden. Alle Sensor Identifier die beschrieben werden können, werden bei der *GetCapabilities*-Funktion zurückgegeben.
GetObservation	Die *GetObservation*-Funktion ermöglicht es, die Beobachtungen von Sensoren abzufragen. Die Antwort wird typischerweise im O&M Format zurückgegeben. Die Antwort darf jedoch auch in einem anderen Datenformat zurückgegeben werden.

Tabelle 3: SOS Basis-Service Funktionen

(Na and Priest, 2007)

In der SOS-Spezifikation Version 1 wurden noch weitere Funktionen spezifiziert, welche optional von einer SOS-Server Instanz angeboten werden können. Diese wurden jedoch gemäss einer empirischen Studie von Tamayo aus dem Jahre 2011 nur sehr bedingt eingesetzt (Tamayo et al., 2011). Im Jahre 2012 wurden mit der Version 2.0 der SOS-Spezifikation grosse Teile der erweiterten/optionalen Funktionen umstrukturiert. Zudem wurden noch weitere Funktionen der SOS-Spezifikation hinzugefügt (Bröring et al., 2012).

In der Version 2.0 wurden zu den Basisfunktionen zusätzlich folgende Funktionen spezifiziert, welche alle optional von einem SOS angeboten werden können. Die Funktionen sind in Erweiterungspaketen (*Extensions*) strukturiert:

Enhanced Extension

Funktion	Beschreibung
GetFeatureOfInterest	Mit Hilfe dieser Funktion kann ein Feature-Of-Interest anhand des Identifiers abgefragt werden. Die Formatierung des Rückgabewertes erfolgt gemäss der übergebenen Formatierung (standardmässig GML).
GetObservationById	Die *GetObservationById* Funktion gibt eine Observation anhand des übergebenen Observation Identifiers zurück.

Tabelle 4: SOS Enhanced Extension Funktionen

Transactional Extension

Funktion	Beschreibung
InsertSensor	Mit Hilfe der *InsertSensor*-Funktion kann ein neuer Sensor auf der SOS-Instanz publiziert werden
UpdateSensorDescription	Die *UpdateSensorDescription*-Funktion kann dazu verwendet werden, die Beschreibung eines Sensors zu verändern
DeleteSensor	Mit der *DeleteSensor*-Funktion kann ein zuvor publizierter Sensor gelöscht werden. Beim Löschen eines Sensors werden auch alle dessen Beobachtungen gelöscht.
InsertObservation	Die *InsertObservation*-Funktion kann dafür verwendet werden, eine oder mehrere Beobachtungen zu einem publizierten Sensor einzufügen.

Tabelle 5: SOS Transactional Extension Funktionen

Result Handling Extension

Funktion	Funktion
InsertResultTemplate	Die *InsertResultTemplate*-Funktion fügt ein *Result Template* in die SOS-Instanz ein, welches dazu verwendet werden kann, Daten eines Sensors weniger wortreich mit einem Feature-Of-Interest in den Server einzufügen.
InsertResult	Wurde ein *Result Template* für einen Sensor mit einem *FeatureOfInterest* eingefügt, kann man mit der *InsertResult*-Funktion lediglich die angefallenen Daten in den Server einfügen.
GetResultTemplate	Gibt die Resultat Struktur und das Encoding vom angeforderten *ResultTemplate* zurück.
GetResult	Gibt die Result-Rohdaten gemäss den übergebenen Parametern zurück.

Tabelle 6: SOS Result Handling Extenstion Funktionen

SOS im Einsatz

Da in der SOS-Spezifikation lediglich die Service-Funktionen beschrieben werden, kann man sich die Verwendung eines solchen Services nur schwer vorstellen. Um zu verbildlichen, wie ein solcher Service eingesetzt werden kann, wird in diesem Unterkapitel ein mögliches Szenario aufgezeigt, in welchem ein SOS-Server verwendet und eingesetzt wird.

Im Zentrum von Abbildung 3 ist eine SOS-Server-Instanz abgebildet. Diese ist mit einer Datenbank verbunden, in welcher Messwerte und Metadaten des Sensors und seine Observationen hinterlegt sind. Wird von einem Client eine Anfrage an den SOS-Server gestellt, führt dieser entsprechende Structured Query Language (SQL) -Statements auf der Datenbank aus, um die gewünschten Ergebnisse dem Aufrufer zu liefern.

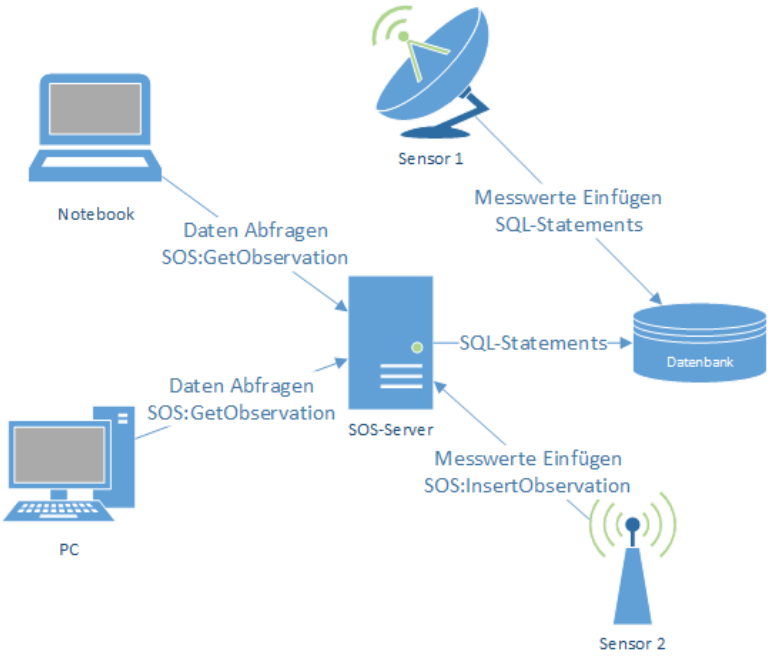

Abbildung 3: Mögliches SOS-Szenario

Der Sensor 1 fügt seine Messwerte mittels der Datenbanksprache SQL direkt in die Datenbank ein, da dieser nur über eine SQL-Schnittstelle und (noch) nicht über eine SOS-Schnittstelle verfügt. Der Sensor 2 führt *SOS:InsertObservation*-Anfragen des SOS-Server aus, um seine Messwerte in den SOS einzufügen. Der PC-Benutzer ist an den Messwerten interessiert welche am 15. Januar 2015 von Sensor 2 aufgezeichnet wurden. Er führt deshalb eine *SOS:GetObservation*-Anfrage mit dem entsprechenden Filter und dem gewünschten Rückgabeformat an den SOS-Server aus, welcher eine Abfrage für die gewünschten Daten auf der Datenbank durchführt und an den SOS zurückgibt. Der SOS gibt als Antwort auf die *SOS:GetObservation*-Anfrage die entsprechenden Resultate im gewünschten Format an den PC zurück.

Der Benutzer vom Notebook ist hingegen an den Messwerten von beiden Sensoren vom 06.

März 2015 interessiert. Da beide Sensoren unter dem gleichen Offering registriert sind, kann der PC eine *SOS:GetObservation*-Anfrage mit einem Filter auf das entsprechende Offering ausführen und erhält Ergebnisse von beiden Sensoren.

Funktionalität im Überblick

In diesem Absatz wird auf die Funktionalitäten der SOS-Implementierung von 52°North eingegangen, da diese in dieser Arbeit bei der SOS-Variante als zentraler Service eingesetzt wird. Die Version 4 dieser Implementierung wurde als Referenz-Implementierung für die Compliance Tests der Version 2.0 des SOS Standards verwendet ("52N Sensor Web Community - Sensor Observation Service," 2015). In folgender Tabelle werden die Funktionalitäten dieser Implementierung aufgelistet. Fett hervorgehobene Funktionen werden in der SOS-Variante, welche in dieser Arbeit erarbeitet wird, verwendet:

Bereich	Implementierte Funktionalitäten
SOS-Funktionen	- SOS-Version 1 (Basis-, und Erweiterte Funktionen) **- SOS-Version 2.0 (Basis-**, und Erweiterte Funktionen)
Datenhaltung	Anbindung an folgende Datenbanken möglich: **- PostgreSQL/PostGIS** - Oracle/Oracle Spatial - MySQL - Miscrosoft SQL Server - H2/GeoDB
Anfragetypen	Anfragen an den SOS können über folgende Datentypen oder Protokolle erfolgen: **- Plain Old XML (POX)** - Simple Object Access Protocol (SOAP) - Key Value Pairs (KVP) - JSON (JavaScript Object Notation)
GetObservation-Antwortformate	Der Rückgabewert einer *GetObservation*-Anfrage kann in folgenden Formaten erfolgen: **- JSON** - O&M 2.0 - WaterML-dr - WaterML

Tabelle 7: SOS Funktionalitäten

2.3.3 Websocket

In diesem Unterkapitel wird auf das Websocket-Protokoll eingegangen, welches im Rahmen dieser Arbeit für die Übertragung der ADS-B Daten in der Websocket-Variante verwendet wird.

Zu Beginn des Unterkapitels wird erläutert, was unter Websockets zu verstehen ist. Daraufhin

wird der Protokollaufbau kurz beschrieben. Zum Schluss wird aufgezeigt, was mögliche Anwendungsfälle für Websockets sein können.

Was sind Websockets?

Websocket ist ein Übertragungsprotokoll, welches eine bidirektionale Kommunikation zwischen einem Client und einem Server über eine einfache Transmission Control Protocol-Verbindung (TCP) ermöglicht. Dieses Protokoll wurde im Jahre 2011 von der Internet Engineering Task Force (IETF) als RFC 6455 standardisiert (Fette and Melnikov, 2011). Vom World Wide Web Consortium (W3C) wurde im Rahmen der HTML 5 Initiative daraufhin eine Programmierschnittstelle für den Einsatz von Websockets auf Webseiten spezifiziert (Hickson, 2011). Websockets zeichnen sich durch ihre Performance aus, da bis anhin auf Webseiten keine bidirektionale Kommunikation zu einem Server möglich war. Ein Zitat von Wang bringt es auf den Punkt:

"WebSocket makes real-time communication much more efficient . ... WebSocket saves bandwidth, CPU power, and latency. WebSocket is an innovation in performance." - (Wang et al., 2013)

Das Protokoll

Der Verbindungsaufbau zur Herstellung eines Websockets erfolgt, um kompatibel mit älteren Web Protokollen zu sein, über eine HTTP-Verbindung. Der anschliessende Wechsel von HTTP zu Websocket wird als "Handshake" bezeichnet. Hierfür sendet der Client, der sich mit einem Websocket Server verbinden will, einen gewöhnlichen HTTP-GET Request an den Server. Ein HTTP-GET Request enthält im Unterschied zu einem HTTP-POST Request nur einen Header und keinen Textkörper. Der Inhalt könnte folgendermassen aussehen:

```
GET                         /clientTrackData HTTP/1.1
Host                        192.168.2.36:8443
Accept                      text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding             gzip, deflate
Accept-Language             en-US,en;q=0.5
Cache-Control               no-cache
Connection                  keep-alive, Upgrade
Host                        127.0.0.1:8443
Origin                      http://localhost:9000
Pragma                      no-cache
Sec-WebSocket-Extensions    permessage-deflate
Sec-WebSocket-Key           LNwk+cjThRCvE5XqnuKlxA==
Sec-WebSocket-Version       13
Upgrade                     websocket
User-Agent                  Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:38.0) Gecko/20100101
                            Firefox/38.0
```

Falls der Server mit dem Request einverstanden ist, antwortet dieser wiederum mit einer normalen HTTP-Antwort. Der Header dieser Antwort könnte folgendermassen aussehen:

```
HTTP/1.1                Switching Protocols
Connection              Upgrade
Sec-WebSocket-Accept    8cHjWURjm3rjWTAghd93Yw3kUqk=
Upgrade                 WebSocket
```

Nach diesem Handshake bricht die HTTP-Verbindung ab und wird mit einer Websocket-Verbindung über dieselbe TCP/IP Verbindung aufgebaut. Ist die Verbindung hergestellt, können Daten direkt vom Client zum Server und vom Server zum Client gesendet werden. Jede Websocket-Meldung muss im Aufbau nur die in Abbildung 4 abgebildeten Anfordernisse erfüllen.

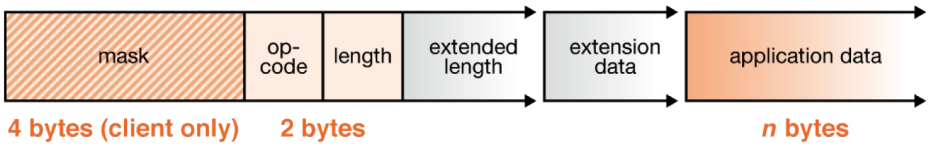

Abbildung 4: Websocket Meldungsaufbau (Quelle: websocket.org)

Verwendung

Websockets werden typischerweise auf Webseiten eingesetzt, um Notifikationen von Servern zu erhalten oder um von einem Webclient Daten zum Server zu senden. Websockets können jedoch auch ausserhalb vom World Wide Web eingesetzt werden wie z.B. in einer Desktop-Applikation zu Server-Verbindungen oder bei Server-zu-Server-Verbindungen. Über Websockets können sowohl binäre wie auch rein textuelle Daten übermittelt werden (Fette and Melnikov, 2011).

Im Unterschied zu gewöhnlichen HTTP-Anfragen muss der Client beim Einsatz von Websockets nur initial eine Verbindung zum Server aufbauen. Steht die Verbindung zum Server, kann dieser jederzeit eine Meldung zum Client senden, ohne, dass dieser erst eine Anfrage an den Server senden muss. Die Kommunikation erfolgt somit nach dem *push*-Prinzip (vgl. Unterkapitel 2.2.2).

In folgender Tabelle werden zwei Anwendungsfälle beschrieben, in welchen Websockets verwendet werden:

Anwendungsfall	Beschreibung
Fussball Liveticker	– Ein Client verbindet sich per Websocket mit dem Server – Der Client sendet per Websocket eine Meldung, dass er an Spiel XY interessiert ist – Der Server sendet nun alle Ereignisse zum Spiel XY über Websocket zum Client
Chat Applikation	– Client 1 und Client 2 verbinden sich jeweils mit dem Server – Client 1 sendet eine Nachricht über Websocket zum Server – Der Server sendet die Nachricht über Websocket an Client 2 weiter

Tabelle 8: Websocket Anwendungsbeispiele

Gemäss der Webseite "Can I use..."[1] unterstützten alle bekannten Browser (Chrome, Firefox, Safari und Internet Explorer) in der aktuellen Version das Protokoll RFC 6455 sowie die vom

1 Can I use... - http://caniuse.com/#feat=websockets [letzter Zugriff 11.1.2015]

W3C spezifizierte API. Somit können Websockets in allen weit verbreiteten Browsern verwendet werden.

2.4 Datenformate

Die Übertragung der Daten erfolgt nicht nur über zwei unterschiedliche Kommunikationsprinzipien und Technologien, sondern auch in Form von unterschiedlichen Datenformaten. In dieser Arbeit werden drei verschiedene Datenformate verwendet, die in unterschiedlichen Bereichen des Systems eingesetzt werden. Das SBS Socket Output-Format wird vom Programm des verwendeten ADS-B Empfängers erzeugt und als Input für das gesamte System verwendet. Das O&M Datenformat wird bei der Kommunikation zwischen dem Integrationssystem und dem SOS verwendet und beschreibt die verschiedenen Messwerte. Bei der Websocket-Variante werden die Daten als GeoJSON-Featuredaten übertragen.

2.4.1 SBS (BaseStation) Socket Output

Das SBS (BaseStation) Socket Output-Format ist ein Datenformat, welches erstmals von *kinetic-avionics* in einem Forumsbeitrag[2] beschrieben wurde. Es ist ein kommasepariertes Datenformat, welches für die Closed Source Software BaseStation[3] geschrieben wurde. Mittels BaseStation können ADS-B Daten empfangen und visuell dargestellt werden. Zudem verbreitet BaseStation typischerweise auf dem TCP Port 30003 die empfangen Flugzeug-Positionsdaten und weitere Daten über das Netzwerk im SBS-1 Socket Output-Format. Über das SBS Socket Output Format werden gemäss des erwähnten Forumbeitrages folgende Werte übertragen:

Parameter	Beschreibung
HexIdent	Eindeutige Mode-S ID vom Flugobjekt
Date message generated	Datum wann die Meldung erzeugt wurde
Time message generated	Zeit wann die Meldung erzeugt wurde
Callsign	Das Rufzeichen des Flugzeuges
Altitude	Relative Flughöhe in Fuss
GroundSpeed	Geschwindigkeit gegenüber dem Grund
Track	Flugrichtung des Flugobjekts abgeleitet von der Ost/West und Nord/Süd Geschwindigkeit in Grad (1-360°).
Latitude/Longitude	Die aktuelle Position des Flugobjekts
VerticalRate	Vertikale Steigung

2 Socket Output Format - http://www.kinetic-avionics.co.uk/forums/viewtopic.php?f=14&t=1402 [letzer Zugriff1 11.01.2015]

3 Basestation - http://www.kinetic-avionics.co.uk/basestationdownloads1.php#1 [letzter Zugriff 11.01.2015]

Parameter	Beschreibung
Squawk	Mode A Transpondercode
Alert (Squawk change)	Gibt an, ob sich das Squawk Flag geändert hat
Emergency	Gibt an, ob das Flugobjekt ein Notfall Signal aussendet
SPI (Ident)	Gibt an, ob der Special Position Identification aktiviert ist
IsOnGround	Gibt an, ob das Flugobjekt sich am Boden befindet

Tabelle 9: SBS-1 Properties

Auf einer weiteren Webseite[4] wurde dieses Übertragungsformat noch detaillierter ausgeführt und spezifiziert die einzelnen übertragenen Werte. Es werden unterschiedliche Meldungstypen über SBS-1 versendet, bei welchen jeweils unterschiedliche Werte befüllt sind. Nachfolgend wird eine SBS1-Meldung aufgelistet, welche neben dem Identifier, den Empfangs- und Sendezeitpunkt, die Geschwindigkeit und die Ausrichtung des Flugzeuges enthält:

```
MSG,4,5,211,4CA2D6,10057,2008/11/28,14:53:49.986,2008/11/28,14:58:51.153,,,408.3,146.4,,,64
,,,,,
```

Da das Format kommasepariert aufgebaut ist und die einzelnen Positionen für alle Meldungstypen jeweils die gleiche Bedeutung haben, ist das Umwandeln dieser Meldungen nicht aufwendig.

2.4.2 O&M

Der Observations and Measurements (O&M) Standard wird im SOS als zentrale Abhängigkeit verwendet, um Observations und Measurements in den SOS einzufügen oder um Observations und Measurements auslesen zu können. Der O&M Standard wurde im Jahre 2011 in der Version 2 in einer abstrakten Spezifikation beschrieben, in welcher das O&M Datenmodell der Version 2 beschrieben wird (Cox, 2011a). Unter anderem erläutert Cox, wie unterschiedliche Arten von Ereignissen als Observierungen beschrieben werden können. So enthalten Observationen jeweils ein Resultat. Zudem beschreibt eine Observation den zu beschreibenden Wert mit einer Referenz auf das Feature-Of-Interest, aus welchem das Resultat ausgelesen wurde.

4 Basestation Socket Data - http://www.homepages.mcb.net/bones/SBS/Article/Barebones42_Socket_Data.htm [letzter Zugriff 11.01.2015]

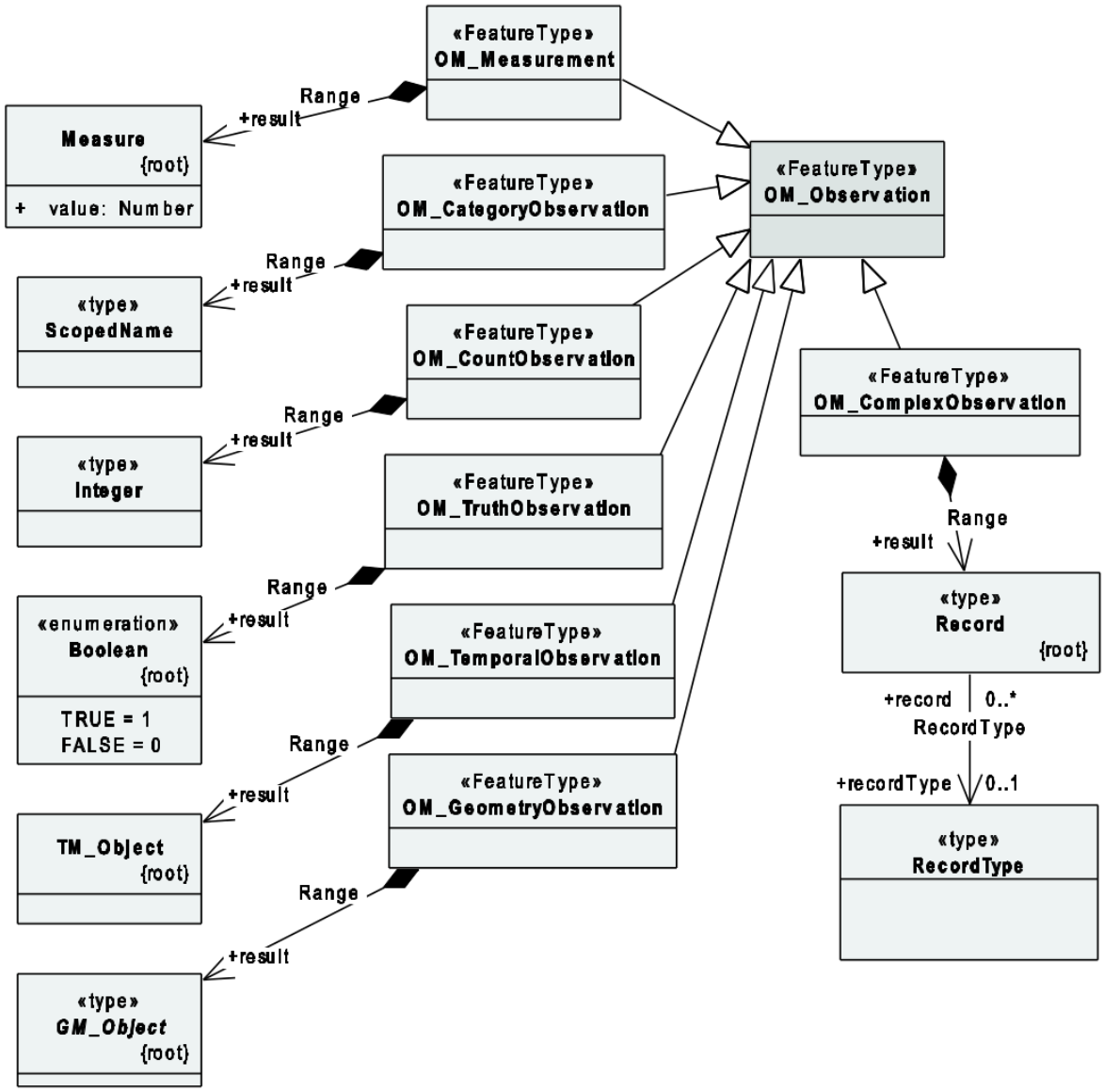

Abbildung 5: Typen von Observation (Quelle: Cox, 2011)

In Abbildung 5 sind die verschiedenen Observations-Typen aufgeführt. Für das Erzeugen von ADS-B Feature Daten sind neben einer normalen *Text*-Observation lediglich die *Geometry*- und die *Measurement*-Observations-Typen relevant. Eine *Text*-Observation beinhaltet, wie der Name vermuten lässt, einen textuellen Inhalt. Eine *Geometry*-Observation beinhaltet als Resultat eine Geometrie, wohingegen eine *Measurement*-Observation eine Gleitkommazahl als Resultat enthält und beschreibt.

In der O&M Implementierungs-Spezifikation beschreibt Cox weiter, wie dieses Datenmodell mit Hilfe von XML umgesetzt werden kann (Cox, 2011b). Eine mögliche Observation des Typs *Geometrie*-Observierung, die aus der umgesetzten SOS-Variante stammt, sieht folgendermassen aus:

```
<ns5:OM_Observation ns3:id="positionObservation">
      <ns3:boundedBy xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:nil="true"/>
      <ns5:phenomenonTime ns4:type="simple" ns4:href="#phenomenTime"/>
      <ns5:resultTime ns4:type="simple" ns4:href="#phenomenTime"/>
      <ns5:procedure ns4:type="simple"
        ns4:href="http://stue.ch/sensorobservation/procedure/flighttracking"/>
      <ns5:observedProperty ns4:type="simple"
        ns4:href="http://stue.ch/sensorobservation/observableProperty/position"/>
      <ns5:featureOfInterest ns4:type="simple"
        ns4:href="http://stue.ch/sensorobservation/foi/aircraft/IY9452"
        ns4:title="IY9452"/>
      <ns5:result xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:type="ns3:GeometryPropertyType" ns4:type="simple">
        <ns3:Point ns3:id="aircraftPosition">
          <ns3:pos srsName="http://www.opengis.net/def/crs/EPSG/0/4326">
              8.178279289215087 45.858138782052954
          </ns3:pos>
        </ns3:Point>
      </ns5:result>
</ns5:OM_Observation>
```

2.4.3 GeoJSON

GeoJSON ist ein Datenformat, welches es ermöglicht, geographische Datenstrukturen darzustellen. Die Spezifikation wurde im Jahr 2008 von Howard Butler veröffentlicht (Butler et al., 2008). GeoJSON basiert auf den Formatierungsregeln JSON, welche von der IETF (Crockford, 2006) und später von ECMA spezifiziert wurden. JSON wurde basierend auf der JavaScript Notation entwickelt. Die JSON-Spezifikation ist nicht sehr umfangreich, da für die JSON-Formatierung nur wenige Regeln befolgt werden müssen. Im Vergleich zum weitverbreiteten XML-Standard weist JSON bei der Übertragung von Daten als auch beim Verarbeiten der Daten signifikante Performance-Vorteile auf (Nurseitov et al., 2009).

Mit Hilfe von GeoJSON ist es möglich, geographische Daten wie eine Geometrie, ein Feature (Geometrie mit zusätzlichen Properties) oder eine Sammlung von Features darzustellen. Es werden folgende einfache Geometrietypen unterstützt:

Typ	Beschreibung
Point	Punktgeometrie
LineString	Liniengeometrie
Polygon	Polygongeometrie

Tabelle 10: Geometrietypen GeoJSON

Die einfachen Geometrien können zudem mehrfach in einem GeoJSON Objekt vorkommen. Für jeden einfachen Geometrie-Typ gibt es ein Multi-Geometrietyp, der mehrere Objekte des entsprechenden Typs enthalten kann. Eine *GeometrieCollection* kann mehrere Objekte enthalten, die verschiedenen Geometrietypen entsprechen (*MultiPoint*, *MultiLineString*, *MultiPolygon* oder *GeometryCollection*).

Sollen zu einer Geometrie zusätzlich Daten verknüpft werden, kann ein GeoJSON-Feature

verwendet werden. Ein Feature muss die Attribute *geometry* (welches ein Geometrie-Objekt enthalten muss) und *properties* (welches zusätzliche Werte zur Geometrie enthalten kann) aufweisen. Möchte man mehrere Features mittels GeoJSON formatieren, kann eine *FeatureCollection* erstellt werden:

Typ	Beschreibung
Feature	Geometrie mit Daten
FeatureCollection	Mehrere Features

Tabelle 11: FeatureTypen GeoJSON

Beispiel eines GeoJSON-Features:

```
{
  "type": "Feature",
  "geometry": {
    "type": "Point",
    "coordinates": [
      7.573060, 47.536420
    ]
  },
  "properties": {
    "address": "Schafmattweg 13",
    "city": "Binningen"
  }
}
```

3. Konzeption des Systems

Im Rahmen dieser Arbeit soll ein System entwickelt werden, mit welchem eine möglichst präzise Vergleichsmöglichkeit zwischen einer SOS- und einer Websocket-Variante für die Übertragung von Messwerten möglich ist. Mit Hilfe dieses entwickelten Systems sollen Sensordaten serverseitig empfangen und über SOS wie auch über Websockets übertragen werden, so dass diese auf einer Weboberfläche dargestellt werden können.

Grundsätzlich sollen beliebige Sensordaten durch das System geschleust werden können. In den folgenden Unterkapiteln wird jedoch darauf eingegangen, wie Daten die über einen SBS-1 Socket Output (vgl. Unterkapitel 2.4.1 SBS (BaseStation) Socket Output) empfangen werden, durch die beiden Varianten verarbeitet und verteilt werden. Zudem wird erläutert, wie das Gesamtsystem aufgebaut ist und wie die einzelnen Komponenten miteinander in Verbindung stehen.

3.1 Zieldaten

Da nicht alle über den SBS-1 Socket Output (vgl. Unterkapitel 2.4.1) versendeten Attribute auf der Weboberfläche ausgewertet werden sollen, müssen nicht alle Attribute zum Client übertragen werden. Dadurch reduziert sich die Menge der zu übertragenden Daten was sich positiv auf die Performance auswirken sollte. Folgende Attribute sollen durch das System nicht ausgefiltert werden:

Parameter	Typ	Einheit	Beispieldaten
HexIdent	Zeichenkette	-	4CA4E5
Date/Time message generated	Zeitpunkt	Unix Timestamp	1432233926692
Callsign	Zeichenkette	-	RJA1118
Altitude	Zahlenwert	Fuss	37000
GroundSpeed	Zahlenwert	Knoten	350.6
Track	Zahlenwert	Grad (0-360)	254
Longitude/Latitude	Position	WGS84-Position	7.573060/47.536420

Tabelle 12: Webclient Zieldaten

Die verschiedenen Attribute sollen möglichst unverändert mittels der entsprechenden Variante zum Webclient übertragen werden.

3.2 Verwendung der Technologien

In diesem Unterkapitel wird der Einsatz der beiden Technologien innerhalb des Gesamtsystems aufgezeigt. Im Speziellen wird darauf eingegangen, wie die Messwerte, welche zum Client übertragen werden (vgl. Unterkapitel 3.1 Zieldaten), mit den in beiden

Varianten verwendeten Technologien übertragen werden sollen.

3.2.1 SOS-Variante

In diesem Unterkapitel wird beschrieben, wie ein SOS eingesetzt werden soll bei der Umsetzung der SOS-Variante. Es wird erläutert, welche Service-Funktionen auf welche Art und Weise eingesetzt werden sollen, damit diese Variante funktionsfähig ist.

Diese Variante orientiert sich am Konzept, welches Jirka und Bredel erarbeitet haben, um Schiffsposition in Nah-Echtzeit zu visualisieren (Jirka and Bredel, 2011). Die empfangenen Daten werden wie auch bei Jirka und Bredel in einen SOS eingefügt, wodurch diese über eine standardisierte Schnittstelle wieder bezogen werden können. Für die Visualisierung der Daten setzen Jirka und Bredel einen WMS ein, in welchem die Schiffspositionen zur Visualisierung aufbereitet werden. Ein Client kann bei der Lösung von Jirka und Bredel Nah-Echtzeit-Bilder über den WMS beziehen und als Kartenlayer darstellen; die Karte wird in einem Intervall von 5 Sekunden neu vom WMS angefordert.

Im Unterschied zu dieser Umsetzung von Jirka und Bredel soll bei der Variante, welche für diese Masterarbeit umgesetzt wird, der Client die Messwerte direkt vom SOS beziehen. Dadurch wird angestrebt, die Menge der zu übertragenden Daten zu reduzieren, da nur Messwerte in textueller Form und nicht Bild- respektive Rasterdaten zum Client übertragen werden müssen. Zudem soll die Latenzzeit durch den direkten Aufruf beim SOS, im Gegensatz zu einer Orchestrierung über einen WMS, verringert werden.

Übertragung der Daten

Jede auf dem Server empfangene SBS1-Meldung soll umgehend an den SOS-Server gesendet werden. Wie in Abbildung 6 aufgezeigt ist, sollen bei der SOS-Variante die auf dem Server empfangenen SBS1-Meldungen mittels *SOS:InsertObservation*-Anfragen an den SOS-Server gesendet werden.

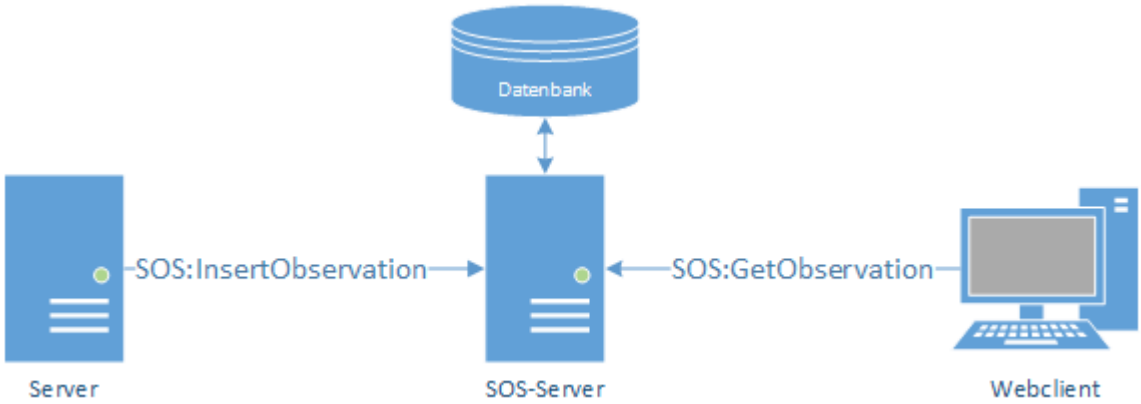

Abbildung 6: SOS konzeptionelle Übersicht

Die mit einer *SOS:InsertObservation* an den SOS-Server gesendeten Daten sollen auf dem

SOS-Server persistiert werden. Mit einer *SOS:GetObservation*-Anfrage sollen die Daten, welche auf dem SOS-Server gespeichert wurden wieder abgefragt und bezogen werden können. Bei der Implementierung dieser Variante soll der Webclient in einem definierbaren Intervall mittels *SOS:GetObservation*-Anfragen die jeweils aktuellen Daten vom SOS-Server auf den Webclient übertragen können.

SOS-Sensor

Um den SOS-Server für die Verarbeitung der beschriebenen Zieldaten (vgl. Unterkapitel 3.13.1 Zieldaten) vorzubereiten, muss der Empfänger der SBS1-Daten als Sensor auf dem SOS-Server registriert werden. Dadurch können die empfangenen SBS1-Datensätze auf dem SOS-Server mittels *SOS:InsertObservation*-Anfragen eingefügt werden. Beim Einfügen des Sensors kann angegeben werden, wie der Name des Procedure, in diesem Fall die Sensorbezeichnung, ist und zu welchem Offering der Sensor gehört. Zudem werden die verschiedenen Properties beschrieben, die durch den Sensor empfangen werden können.

Um den Sensor zu registrieren, muss eine *SOS:InsertSensor*-Anfrage (vgl. Anhang A.1 InsertSensor) ausgeführt werden. Der mit der *SOS:InsertSensor*-Anfrage registrierte Sensor wird mit folgenden Eigenschaften konfiguriert:

Eigenschaft	Wert
Procedure Identifier	http://stue.ch/sensorobservation/procedure/sbs-1
Offering Identifier	http://stue.ch/sensorobservation/offering/flighttracking

Tabelle 13: SOS Sensor Eigenschaften

Die Properties wurden wie folgt auf dem Sensor definiert:

Property	Identifier	Einheit
Callsign	http://stue.ch/sensorobservation/observableProperty/callsign	Zeichenkette
Altitude	http://stue.ch/sensorobservation/observableProperty/altitude	Feet
Speed	http://stue.ch/sensorobservation/observableProperty/speed	Knoten
Heading	http://stue.ch/sensorobservation/observableProperty/heading	Grad
Position	http://stue.ch/sensorobservation/observableProperty/position	Lon/Lat

Tabelle 14: SOS Sensor Properties

Die URL-Identifier zeigen auf nicht existierende Seiten. Sie werden lediglich zur eindeutigen Identifikation verwendet.

Anmerkung: Die Verwendung der Domain http://stue.ch resultiert daraus, dass diese vom Autor dieser Arbeit registriert ist. Es wird somit garantiert, dass die Identifier eindeutig sind.

SOS-Anfragen

Nachfolgend wird beschrieben, wie die Werte der SBS1-Meldungen in den SOS geschrieben werden sollen und wie diese wieder ausgelesen werden können.

Einfügen

Da eine SBS1-Meldung die aus dem SBS1 Socket Output stammt, mehrere Attribute enthalten kann, soll jedes Attribut der SBS1-Meldung in eine Observation konvertiert werden. Gemäss SOS-Spezifikation kann eine *SOS:InsertObservation*-Meldung mehrere Observationen beinhalten (Bröring et al., 2012). Deshalb sollen die Attribute einer SBS1-Meldung in einer *SOS:InsertObservation* zusammengefasst werden. Somit wird für jede empfangene SBS1-Meldung eine *SOS:InsertObservation*-Meldung erzeugt und an den SOS-Server gesendet. Da nicht jede SBS1-Meldung alle Attribute beinhaltet, soll eine Observation nur für jene Attribute erzeugt und übertragen werden, die einen Wert beinhalten. Dadurch wird die Menge der übertragenen Daten zum SOS-Server verringert. Zudem müssen auf dem SOS-Server weniger Daten in der Datenbank gespeichert werden. Jede dieser Meldungen wird an das gleiche Offering gesendet, wodurch alle Attribute unter dem gleichen Offering und somit unter der gleichen Procedure verfügbar sind.

Der Grundaufbau einer möglichen *SOS:InsertObservation* kann folgendermassen aussehen:

```
<?xml version="1.0"?>
<ns3:InsertObservation xmlns:ns1="http://www.opengis.net/ows/1.1"
 ...
xmlns:ns14="http://www.opengis.net/swe/2.0" version="2.0.0" service="SOS">
  <ns3:offering>http://stue.ch/sensorobservation/offering/ads-b</ns3:offering>
  <ns3:observation>
    <ns7:OM_Observation ns6:id="callsignObservation">
      ...
    </ns7:OM_Observation>
  </ns3:observation>
  <ns3:observation>
    <ns7:OM_Observation ns6:id="speedObservation">
      ...
    </ns7:OM_Observation>
  </ns3:observation>
  <ns3:observation>
    <ns7:OM_Observation ns6:id="altitudeObservation">
      ...
    </ns7:OM_Observation>
  </ns3:observation>
  <ns3:observation>
    <ns7:OM_Observation ns6:id="headingObservation">
      ...
    </ns7:OM_Observation>
  </ns3:observation>
  <ns3:observation>
    <ns7:OM_Observation ns6:id="positionObservation">
      ...
    </ns7:OM_Observation>
  </ns3:observation>
</ns3:InsertObservation>
```

Bei diesem Beispiel werden fünf Messwerte in einer *SOS:InsertObservation* übertragen. Der Aufbau der darin enthaltenen Observationen ist grundsätzlich bei allen Properties identisch:

```
<ns4:OM_Observation ns2:id="callsignObservation">
      <ns4:phenomenonTime ns3:type="simple">
             <ns2:TimeInstant ns2:id="phenomenonTime">
                    <ns2:timePosition>2015-05-26T20:52:12.834+0200</ns2:timePosition>
             </ns2:TimeInstant>
      </ns4:phenomenonTime>
      <ns4:resultTime ns3:type="simple" ns3:href="#phenomenonTime" />
      <ns4:procedure ns3:type="simple"
             ns3:href="http://stue.ch/sensorobservation/procedure/sbs-1" />
      <ns4:observedProperty ns3:type="simple"
             ns3:href="http://stue.ch/sensorobservation/observableProperty/callsign" />
      <ns4:featureOfInterest ns3:type="simple"
             ns3:href="http://stue.ch/sensorobservation/foi/aircraft/FO5606"
             ns3:title="FO5606" />
      <ns4:result xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xmlns:xs="http://www.w3.org/2001/XMLSchema"
             xsi:type="xs:string">PYR145</ns4:result>
</ns4:OM_Observation>
```

In der Implementierung dieses Konzeptes soll die *phenomenon*-Zeit (der Zeitpunkt wann das Ereignis aufgetreten ist) den Wert des SBS1-Message Date/Time *messageGenerated*-Attributes beinhalten. Da der *phenomenon*-Zeitpunkt dem *result*-Zeitpunkt entspricht, wird eine Referenz auf den *phenomenon*-Wert gesetzt. In einer *SOS:InsertObservation* soll nur die *phenomenon*-Time innerhalb der ersten Observation definiert sein. In den folgenden Observationen kann eine Referenz auf die *phenomenTime* gesetzt werden, da diese für alle Attribute identisch ist.

```
      ...
      </ns4:OM_Observation>
</ns9:observation>
<ns9:observation>
      <ns4:OM_Observation ns2:id="speedObservation">
             <ns2:boundedBy xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xsi:nil="true" />
             <ns4:phenomenonTime ns3:type="simple" ns3:href="#phenomenonTime" />
             <ns4:resultTime ns3:type="simple" ns3:href="#phenomenonTime" />
             ...
```

Dadurch wird die XML-Meldung wiederum verkleinert, was die zu übertragende Datenmenge reduziert. Das Procedure ist bei allen Messwerten identisch, da alle vom selben Sensor empfangen wurden. Das *observedProperty*-Tag ist je nach Property unterschiedlich. Die Werte werden gemäss des im Sensor definierten Property-Identifier gesetzt. Beim Feature-Of-Interest wird kein Feature angegeben, sondern lediglich eine Referenz auf das Flugobjekt vermerkt. Da alle Attribute des Feature-Of-Interest als Messwerte übertragen werden, genügt es, eine eindeutige Referenz für jedes Flugobjekt zu übergeben. Die Referenz zum Feature-Of-Interest beinhaltet den Mode-S des Flugobjekts, welcher in der SBS-1 Meldung als *hexIdent* bezeichnet wird. Mit Hilfe des Mode-S kann ein Flugobjekt eindeutig identifiziert werden (vgl. Unterkapitel 2.1 ADS-B).

Da die Messwerte von unterschiedlichem Typ sind, sollen die *result*-Tags je nach Typ unterschiedlich sein:

Result Type	Property	Beispiel
String	callsign	`<ns4:result xsi:type="xs:string">PYR145</ns4:result>`

Measurement	speed
	altitude
	heading
		`<ns4:result xsi:type="ns2:MeasureType" uom="[kn_i]">476.0</ns4:result>`
		`<ns4:result xsi:type="ns2:MeasureType" uom="[ft_i]">338.0</ns4:result>`
		`<ns4:result xsi:type="ns2:MeasureType" uom="deg">309.0</ns4:result>`

Geometry	position	`<ns4:result xsi:type="ns2:GeometryPropertyType"><ns2:Point ns2:id="aircraftPosition"><ns2:pos srsName="http://www.opengis.net/def/crs/EPSG/0/4326">11.047831559160208 45.49557764222335</ns2:pos></ns2:Point></ns4:result>`

Tabelle 15: SOS Result Typen

Auslesen

Vom Webclient soll in einem definierbaren Intervall eine Anfrage an den SOS-Server gesendet werden um neue Observationen auszulesen. Dies soll mit Hilfe von *SOS:GetObservation*-Anfragen erfolgen.

```
<?xml version="1.0" encoding="UTF-8"?>
<sos:GetObservation service="SOS" version="2.0.0"
      ...
      xsi:schemaLocation="http://www.opengis.net/sos/2.0
      http://schemas.opengis.net/sos/2.0/sos.xsd">
  <sos:offering>http://stue.ch/sensorobservation/offering/flighttracking</sos:offering>
  <sos:temporalFilter>
    <fes:During>
      <fes:ValueReference>phenomenonTime</fes:ValueReference>
      <gml:TimePeriod gml:id="t1">
        <gml:beginPosition>2015-05-27T15:18:30.434Z</gml:beginPosition>
        <gml:endPosition>2015-05-27T15:18:30.784Z</gml:endPosition>
      </gml:TimePeriod>
    </fes:During>
  </sos:temporalFilter>
  <sos:responseFormat>application/json</sos:responseFormat>
</sos:GetObservation>
```

In der *SOS:GetObservation*-Anfrage wird das Offering angegeben, von welchem die Observationen geladen werden sollen. Zudem wird in der Anfrage ein temporaler-Filter gesetzt. Dadurch werden alle Observationen als Antwort gesendet, welche sich zwischen dem *beginPosition*- und *endPosition*-Zeitpunkt befinden.

Bei einer nächsten Anfrage soll der aktuelle *startPosition*-Zeitpunkt auf die neueste r*esult*-Zeit

von etwaigen Ergebnissen gesetzt werden. Werden mit einem Request keine neuen Observationen empfangen, soll der *startPosition*-Zeitpunkt auf jenem des vorgängigen Request bleiben. Der *endPosition*-Zeitpunkt wird immer auf das aktuelle Datum gesetzt. Dadurch ist garantiert, dass man alle Observationen seit dem Beginn der ersten Abfrage empfängt unter der Voraussetzung, dass die Observationen sequenziell nach *phenomenon*-Zeit eingefügt werden. Da die Verarbeitung im Webclient in JavaScript erfolgt, eignet sich das JSON-Format als Antwortformat. In JavaScript können JSON-Daten einfach in JavaScript Objekte umgewandelt werden. Durch die Angabe des *response*-Formates auf "*application/json*" wird die Antwort des Servers im JSON Format zurückgegeben. Eine mögliche Antwort des Servers könnte folgendermassen aussehen:


```
{
  "request" : "GetObservation",
  "version" : "2.0.0",
  "service" : "SOS",
  "observations" : [
    {
      "type" : "http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_Measurement",
      "procedure" : "http://stue.ch/sensorobservation/procedure/sbs-1",
      "offering" : "http://stue.ch/sensorobservation/offering/flighttracking",
      "observableProperty" : "http://stue.ch/sensorobservation/observableProperty/heading",
      "featureOfInterest" : "http://stue.ch/sensorobservation/foi/aircraft/OA5626",
      "phenomenonTime" : "2015-05-27T15:18:30.593Z",
      "resultTime" : "2015-05-27T15:18:30.593Z",
      "result" : {
        "uom" : "deg",
        "value" : 16
      }
    },
    {
      "type" :
         "http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_GeometryObservation",
      "procedure" : "http://stue.ch/sensorobservation/procedure/sbs-1",
      "offering" : "http://stue.ch/sensorobservation/offering/flighttracking",
      "observableProperty" :
        "http://stue.ch/sensorobservation/observableProperty/position",
      "featureOfInterest" : "http://stue.ch/sensorobservation/foi/aircraft/PY8868",
      "phenomenonTime" : "2015-05-27T15:18:30.738Z",
      "resultTime" : "2015-05-27T15:18:30.738Z",
      "result" : {
        "type" : "Point",
        "coordinates" : [
          10.874314927293595,
          44.48931950733285
        ]
      }
    },
    {
      "type" : "http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_TextObservation",
      "procedure" : "http://stue.ch/sensorobservation/procedure/sbs-1",
      "offering" : "http://stue.ch/sensorobservation/offering/flighttracking",
      "observableProperty" :
        "http://stue.ch/sensorobservation/observableProperty/callsign",
      "featureOfInterest" : "http://stue.ch/sensorobservation/foi/aircraft/OA5626",
      "phenomenonTime" : "2015-05-27T15:18:30.593Z",
      "resultTime" : "2015-05-27T15:18:30.593Z",
      "result" : "IQQ862"
  }, ...
}
```

Die Antwort beinhaltet ausser dem Request Type, der Version und des Service alle Observationen im JSON Format. Alle Daten, die man beim Einfügen der Observation mitgegeben hat, können den einzelnen JSON-Observationen wieder entnommen werden. Anhand des Typs der Observation kann unterschieden werden, welche Messwerte die Observation beinhaltet. Durch das *observableProperty*-Attribut kann zudem festgestellt werden, welches Property empfangen wurde. Das *FeatureOfInterest* enthält die übergebene Referenz zum Flugobjekt mit dem eindeutigen Mode-S Bezeichner. Dadurch kann der Messwert einem Feature im Webclient zugewiesen werden.

Auf dem Webclient sollen somit die einzelnen Observationen anhand des Feature-Of-Interest Identifiers und des Observable-Property wieder als Feature Property zusammengesetzt werden. Falls auf dem Client bereits ein Feature mit dem entsprechenden Feature-Of-Interest besteht, soll das Feature auf dem Client mit den neuen Daten aktualisiert werden.

3.2.2 Websocket-Variante

In diesem Unterkapitel wird die Websocket-Variante beschrieben. Es wird erläutert, wie Websockets eingesetzt werden sollen, um die Werte der SBS1-Meldungen zum Webclient zu übertragen. Zudem wird aufgezeigt, was die Aufgaben der Websockets bei dieser Variante sind und wie die Daten formatiert werden sollen, damit diese auf dem Webclient möglichst einfach ausgewertet werden können.

Übertragung der Daten

In der Websocket-Variante sollen die auf dem Server empfangenen SBS1-Meldungen durch ein Websocket an den Webclient gesendet werden. Analog der Lösung von Pimentel und Nickerson sollen die Sensor-Messwerte auf dem Serversystem verarbeitet und anschliessend durch ein Websocket zu einem Client übertragen werden (Pimentel and Nickerson, 2012).

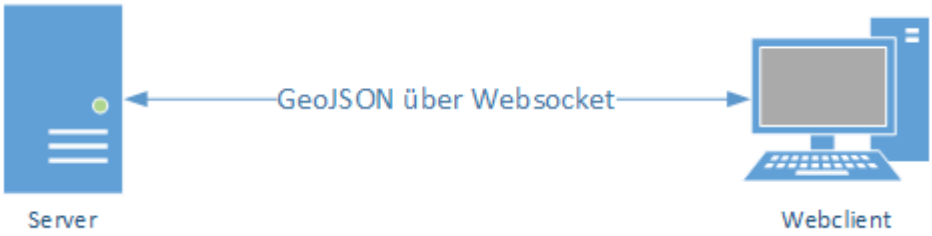

Abbildung 7: Websocket konzeptionelle Übersicht

Somit steht, wie in Abbildung 7 aufgezeigt, bei der Websocket-Variante der Client mit dem Server direkt in Verbindung. Bei der Implementierung dieser Variante sendet der Server nach dem Erhalt von neuen Daten diese umgehend zum Webclient. Dies ist durch die Verwendung von Websocket als Protokoll zwischen Server und Webclient möglich (vgl. Unterkapitel 2.3.3 Websocket). Der Webclient muss nur initial die Verbindung zum Server herstellen. Anschliessend kann der Server Daten direkt zum Webclient senden, ohne, dass dieser in

regelmässigen zeitlichen Abständen Abfragen auf dem Server durchführen muss.

Die vom Server gesendeten Daten sollen als GeoJSON formatierte Datensätze übertragen werden. Die Aufgabe des Servers in dieser Variante ist es, die SBS-1 Socket Output-Meldungen in GeoJSON-Zeichenketten umzuwandeln und diese in das Websocket zu schreiben.

Daten

Die SBS-1 Meldungen sollen im Server als GeoJSON-Feature Zeichenketten (vgl. Unterkapitel 2.4.3 GeoJSON) formatiert werden. Die Position des Flugobjektes entspricht der Geometrie des GeoJSON-Features und die restlichen Werte sollen als Properties dem Feature zugewiesen werden. Aus jeder empfangener SBS-1 Meldung wird ein GeoJSON-Feature erzeugt, welches umgehend vom Server in das Websocket geschrieben werden soll.

Die Geometrie soll als GeoJSON-Punkt Geometrie formatiert übertragen werden. Die anderen Messwerte werden dem Typ entsprechend als Properties dem GeoJSON-Feature zugefügt. Die verschiedenen Properties sollen folgendermassen übertragen werden:

Parameter	Typ	Beispiel
HexIdent callsign	Zeichenkette	`"hexIdent": "WP6203",` `"callsign": "REC968"`
Date/Time message generated	Zeitpunkt	`"messageGenerated": 1432834881844`
Altitude	Zahlenwert	`"altitude": 168`
GroundSpeed	Zahlenwert	`"groundSpeed": 643`
Track	Zahlenwert	`"heading": 108`

Tabelle 16: GeoJSON Zieldaten Properties

Eine mögliche vom Server erzeugte GeoJSON-Meldung welche alle Properties enthält könnte folgendermassen aussehen:

```
{
	"type": "Feature",
	"geometry":
	{
		"type": "Point",
		"coordinates":
		[
			10.464148374219123,
```

```
                    45.472322350414835
                ]
        },

        "properties":
        {
                "groundSpeed": 643,
                "callsign": "REC968",
                "messageGenerated": 1432834881844,
                "hexIdent": "WP6203",
                "heading": 108,
                "altitude": 168
        }
}
```

Enthält eine SBS1-Meldung keine Geometrie-Informationen, soll das GeoJSON-Feature dennoch erzeugt werden, jedoch ohne das *geometry*-Attribut. Beinhaltet eine SBS1-Meldung einen Messwert eines zu übertragenden Properties nicht, so soll das Property nicht aufgeführt werden. Der *hexIdent* muss jedoch zwingend gesetzt sein, da ansonsten die Werte keinem Flugobjekt zugewiesen werden können. Zudem muss der Zeitstempel der *messageGenerated* zwingend angegeben werden, da der Client ansonsten nicht weiss, wann die Meldung erzeugt wurde. Eine Meldung, die lediglich das *callSign*, den *hexIdent* und die *messageGenerated* enthält, könnte somit folgendermassen aussehen.

```
{
        "type": "Feature",
        "properties":
        {
                "callsign": "REC968",
                "messageGenerated": 1432834881289,
                "hexIdent": "WP6203"
        }
}
```

3.3 Übersicht Gesamtsystem

Unter Betrachtung der beiden Varianten soll das Gesamtsystem, wie in Abbildung 8 dargestellt, aufgebaut werden. Grundsätzlich kann es in vier Bereiche unterteilt werden.

Der erste Bereich befasst sich mit dem Empfang der ADS-B Daten, welche per Transponder von Flugzeugen publiziert werden. Da ADS-B Daten unverschlüsselt auf der Frequenz 1090 MHz von den Flugzeugen mittels deren Transmitter publiziert werden, können diese Daten mit dem Einsatz von wenigen Hilfsmitteln empfangen und decodiert werden (vgl. Unterkapitel 2.1). Die Daten werden vom ADS-B Empfänger über eine TCP-Verbindung im SBS1-Socket Output Format als Datenstrom zum Integrationssystem übertragen. Der zweite Bereich stellt das Integrationssystem dar, welches die vom ADS-B gesendeten Daten umwandelt und die Daten für die beiden Varianten SOS und Websocket vorbereitet. Die ADS-B Daten werden parallel an die beiden Varianten übertragen. Die beiden Varianten verarbeiten die Daten nun der Technologie entsprechend, so dass diese über das Netzwerk übertragen und auf einem Webclient dargestellt werden können.

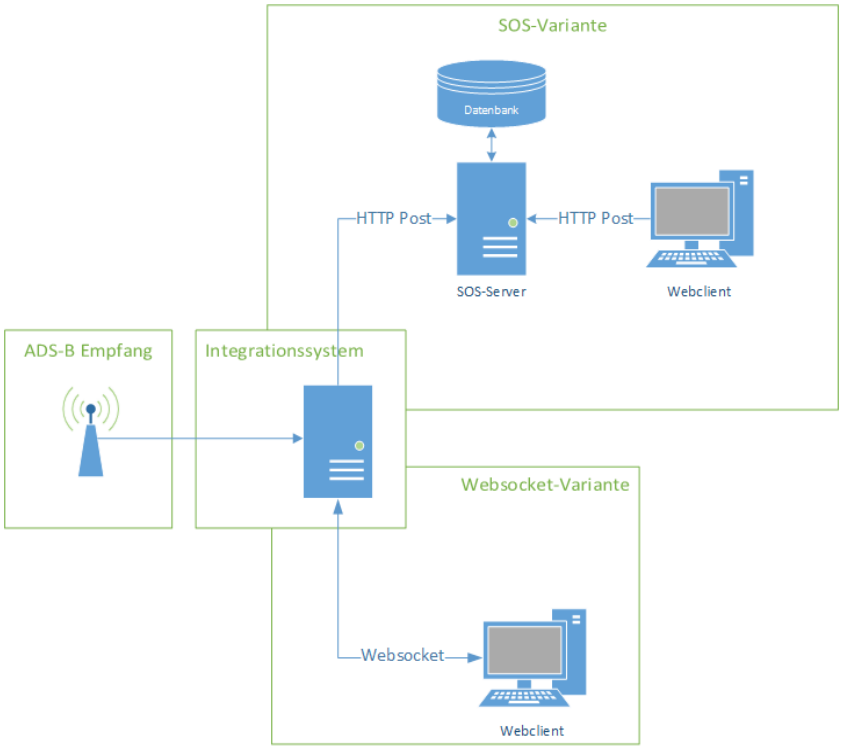

Abbildung 8: Konzeptionelle Übersicht Gesamtsystem

Im folgenden Kapitel wird darauf eingegangen, wie das Integrationssystem implementiert wurde, so dass die Daten über beide Varianten übertragen werden können. In Kapitel 5 wird die Umsetzung des Webclients genauer beschrieben.

4. Umsetzung Integrationssystem

Das Integrationssystem wandelt die vom ADS-B Empfänger gesendeten Daten um und verteilt sie über die beiden Varianten. Es basiert auf dem Open Source Framework Apache Camel (vgl. Unterkapitel 4.1.2 Apache Camel). Da Apache Camel ein Routingsystem ist, werden die einzelnen Verarbeitungsschritte als Routen bezeichnet. Das Integrationssystem hat eine Route, welche sich um den Datenempfang kümmert. In dieser Empfangsroute werden die SBS-1 Meldungen in Feature-Objekte umgewandelt. Zudem werden überflüssige Meldungen oder Werte ausgefiltert. Die Meldungen werden dann parallel über zwei weitere Routen verteilt, welche die Feature Daten gemäss entsprechenden Technologien weiterverarbeiten und versenden.

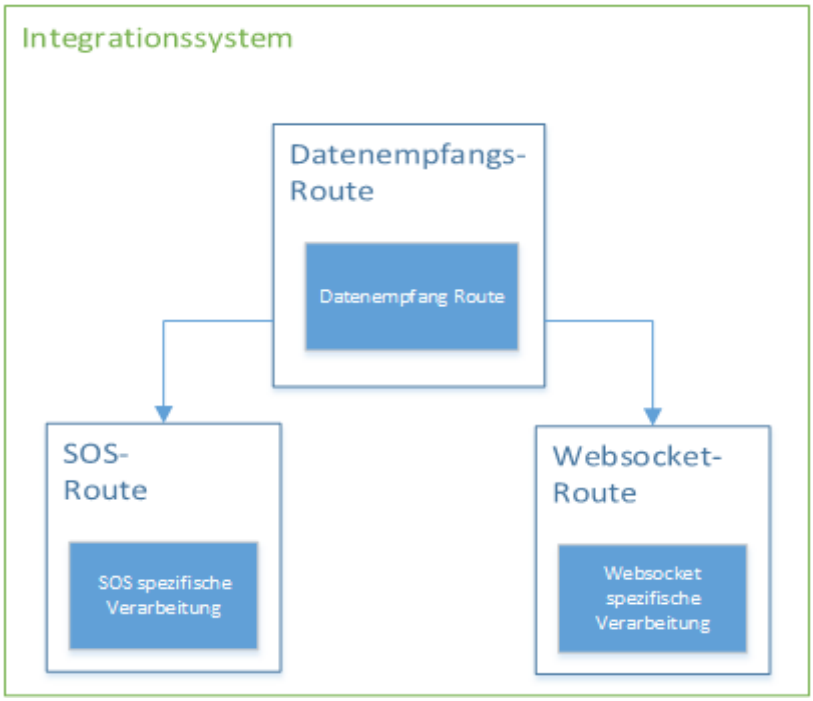

Abbildung 9: Übersicht Routingsystem

In Unterkapitel 4.1 wird auf die Java-Bibliotheken eingegangen, welche innerhalb dieses Integrationssystems verwendet werden. In Unterkapitel 4.2 werden die Java-Klassen beschrieben, deren Instanzen die Daten repräsentieren, welche im Integrationssystem empfangen und innerhalb der Routen verarbeitet werden. In den Unterkapiteln 4.3 Datenempfangs-Route, 4.4 SOS-Route und 4.5 Websocket-Route werden die einzelnen Routen im Detail beschrieben.

4.1 Verwendete Java-Bibliotheken

Die Implementierung des Integrationssystems basiert stark auf dem Spring Framework und auf dem Routingsystem Apache Camel. In diesem Unterkapitel werden diese beiden Rahmenwerke und weitere verwendete Bibliotheken vorgestellt. Zudem wird deren Einsatz innerhalb des Integrationssystems erläutert.

4.1.1 Spring Framework

Das Spring Framework[5] ist ein Open Source Rahmenwerk für Java, welches ursprünglich von Rod Johnson veröffentlicht wurde. Das Framework wird auch als Inversion of Control (IoC) Container bezeichnet und kann im Speziellen bei der Initialisierung von Objekten und deren Abhängigkeiten durch Dependency Injection sehr nützlich sein (Johnson, 2005).

Das Spring Framework ermöglicht es, Objekte (in Spring *Beans* genannt) innerhalb einer XML-Datei zu definieren. Die einzelnen in XML definierten Beans werden beim Initialisieren des Spring *Contextes* durch Spring erzeugt. Dabei werden alle Abhängigkeiten zwischen den Beans durch Spring aufgelöst, sofern dies möglich ist. Diese Art von Initialisieren der Objekte wird als Dependency Injection bezeichnet (Johnson, 2005). Innerhalb des Integrationssystems wird die Erzeugung von Objekten und deren Abhängigkeiten in Spring mit Hilfe von XML definiert.

4.1.2 Apache Camel

Apache Camel[6] ist ein Open Source Framework, welches für die Systemintegration verwendet werden kann. Der Fokus von Camel liegt darin, die Integration von verschiedenen Systemen möglichst einfach zu gestalten. Der Kern von Apache Camel ist ein System, welches das Erstellen von Routen ermöglicht. Eine Route kann als ein Regelwerk angesehen werden, wie Daten verarbeitet werden sollen. Eine Route besteht jeweils aus einer Datenquelle (in Camel *Producer* genannt) und einem Endpunkt (in Camel *Endpoint* genannt). Möchte man die Daten zwischen dem Producer und dem Endpoint zudem verarbeiten oder umwandeln, können beliebig viele Verarbeitungsroutinen (*Processor*) oder Datenumwandler (*TypeConverter*) eingefügt werden. Es gibt verschiedene Möglichkeiten, wie Apache Camel-Routen deklariert werden können. So kann dies über eine domänenspezifische Sprache (DSL) oder aber über eine XML-Konfigurationsdatei integriert in Spring geschehen (Ibsen and Anstey, 2010).

Komponenten

Apache Camel ist so konzipiert, dass das Framework mit wenig Aufwand erweitert werden kann. Es gibt mittlerweile einige Komponenten, welche für Apache Camel entwickelt wurden

5 Spring Framework - http://projects.spring.io/spring-framework/ [letzter Zugriff 25.06.2015]
6 Apache Camel - http://camel.apache.org/ [letzter Zugriff 25.06.2015]

und als *Producer* und oder als *Endpoint* verwendet werden können. Folgende Auflistung ist ein Auszug der Komponentenübersicht der offiziellen Camel Webseite[7]. Sie zeigt einige wenige Komponenten, welche von Camel unterstützt werden:

Komponente	Verwendung
File	Lesen und Schreiben von Dateien auf dem lokalen Dateisystem
SMTP	Senden und Empfangen von E-Mails
JMS	Lesen oder Schreiben über den Java Message Service
Websocket	Lesen oder Schreiben in ein Websocket
FTP	Herunter-/ oder Hochladen von Dateien auf/von einem FTP-Server
Dropbox	Herunter-/ oder Hochladen von Dateien auf/von Dropbox
HTTP	Ausführen von HTTP-Anfragen
IRC	Kommunizieren über IRC
JDBC	Ausführen von Datenbank Operationen und Queries
...	

Tabelle 17: Auszug Camel Komponenten

Beispiel-Route

Für diese Arbeit wurden alle Camel-Routen über Spring XML Konfigurationsdateien erzeugt. Aus diesem Grund wird an dieser Stelle nur auf diese Art der Routenerstellung eingegangen. In folgendem Beispiel wird eine Camel-Route mittels XML erzeugt:

```
...
<bean id="bufferGeometries" class="ch.test.MyBuffer">
      <property name="size">10</property>
</bean>
...
<camel:route autoStartup="true">
<camel:from   uri="dropbox://get?accessToken=XXX
              &clientIdentifier=XXX
              &remotePath=/sample_data.json" />
      <camel:convertBodyTo type="ch.test.message.GeoJSONFeature" />
      <camel:process uri="bufferGeometries" />
      <camel:to uri="file://buffered_sample_data.json" />
</camel:route>
...
```

Das Beispiel zeigt auf, wie mit wenigen Zeilen XML ein ganzer Workflow erzeugt werden kann. In diesem Beispiel wird eine Datei (sample_data.json) von einem Dropbox Account heruntergeladen. Dies erfolgt lediglich durch die URI-Angabe in der *from*-Anweisung. In der *convertBodyTo* wird der Inhalt der Datei umgewandelt. In diesem Fall soll der Inhalt in ein *ch.test.message.GeoJSONFeature* umgewandelt werden. In der Process-Anweisung wird danach ein Befehl auf dem *GeoJSONFeature* ausgeführt. Der Prozess *bufferGeometries*

7 Apache Camel: Components - http://camel.apache.org/components.html [letzter Zugriff 20.01.2015]

wurde vor der Route bereits in einem Spring Bean definiert. Damit soll ein Buffer um das *GeoJSONFeature* erstellt werden. Am Ende wird durch die Angabe des *to*-Elementes angegeben, wo die Route enden soll. Durch die Angabe des *file*-Prefixes weiss Camel, dass nun die Datei auf dem lokalen Dateisystem erzeugt werden soll, unter dem Namen *buffered_sample_data.json*.

4.1.3 JAX-B

Mit Hilfe der Programmierschnittstelle Java Architecture for XML Binding (JAXB)[8] ist es möglich, einzelne XML-Schema Dateien an Java-Klassen zu binden. So können aus einem XML-Schema Java Klassen generiert werden, welche instanziiert und mit Werten befüllt werden können. Es ist jedoch auch möglich aus speziell mit JAXB-Annotationen versehenen Java-Klassen ein XML-Schema zu generieren.

Die zu generierenden Java-Objekte können in Java erzeugt und mit Werten befüllt werden. Durch *Marshalling* können die Objekte in ein XML-Dokument überführt werden. Ein XML-Dokument kann wiederum durch *Unmarshalling* in Java-Objekte umgewandelt werden.

JAX-B wird im Integrationssystem in der Version 2.2.7 verwendet. Die Funktionen der JAX-B Programmierschnittstelle werden im Speziellen beim Erstellen der *SOS:InsertObservation*-Anfragen verwendet. Etliche Java-Schema-Klassen für die OGC-Standards sind als Open Source-Bibliotheken öffentlich verfügbar[9]. Für das Erzeugen von *SOS:InsertObservation*-XML-Anfragen wird die SOS v2.0 JAX-B Bibliothek verwendet. Durch das Instanziieren und mit Daten befüllen von diesen SOS v2.0 Java-Schema-Klassen kann in Java eine Abbildung von einer SOS:InsertObservation erstellt werden. Durch *Mashalling* dieser Daten wird eine XML-Repräsentation der *SOS:InsertObservation* erstellt, welche als Anfrage an den SOS-Server gesendet werden kann.

4.1.4 FasterXML/Jackson

Jackson ist eine Open Source Java Bibliothek, welche auf Github[10] weiterentwickelt wird. Analog JAX-B ist mit Jackson eine Datenbindung möglich. Im Unterschied zu JAXB bindet Jackson Daten nicht an XML-Schemas, sondern ermöglicht es, JSON-Objekte in Java-Objekte und Java-Objekte in JSON-Objekte umzuwandeln.

Um Java-Klassen mit Jackson in JSON-Objekte umwandeln zu können, müssen diese mit Java-Annotationen versehen werden. Die Feature-Klasse, welche im Integrationssystem verwendet wird (vgl. Unterkapitel 4.2.2 Feature), ist mit solchen Annotation versehen, so dass die Feature-Instanzen in GeoJSON-Objekte umgewandelt werden können. Im Integrationssystem wird Jackson in der Version 2.3.1 verwendet.

8 JAXB Reference Implementation - https://jaxb.java.net/ [letzter Zugriff 19.06.2015]
9 JAXB for OGC - http://www.ogcnetwork.net/jaxb4ogc [letzter Zugriff 15.06.2015]
10 Faster XML / Jackson - https://github.com/FasterXML/jackson [letzter Zugriff 19.06.2015]

4.2 Domänendaten

Innerhalb des Integrationssystems wurden Klassen erstellt und verwendet, welche die übertragenen Daten beschreiben. Instanzen dieser Klassen repräsentieren die empfangenen respektive die verarbeiteten Daten innerhalb des Integrationssystems.

4.2.1 SBS1Message

Die Klasse SBS1Message bildet die SBS1-Datensätze, welche vom ADS-B-Empfänger versendet werden, ab. Mit Hilfe dieser Klasse wird der SBS1-Socket Output innerhalb des Integrationssystems repräsentiert.

Eine *SBS1Message* enthält alle 22 Datenfelder, welche im SBS1 Socket Output-Datenformat enthalten sind (vgl. Abbildung 10), als Klassenattribute. Für jedes Attribut innerhalb der SBS1Message-Klasse gibt es entsprechende Zugriffsmethoden. Für das Auslesen enthält die Klasse für jedes Attribut eine Getter-Methode (z.B. die *getMessageType*-Methode). Um Daten einem Attribut zuzuweisen, gibt es für jedes Attribut eine entsprechende Setter-Methode (z.B. die *setMessageType*-Methode). Die Getter- und Setter-Methoden sind aus Platzgründen auf dem UML-Diagramm nicht abgebildet.

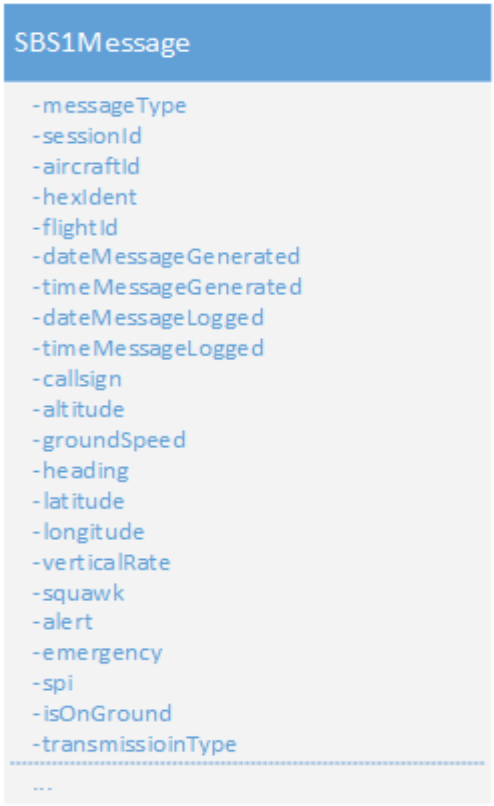

Abbildung 10: Klasse SBS1Message

Da im SBS-1 Datenformat nicht bei jeder Meldung alle Felder übertragen werden (vgl. Unterkapitel 2.4.1), können in einer SBS1Message einzelne Attribute leer sein. Anhand des Attributes *messageType* kann bei der Verarbeitung der Daten der Meldungstyp ausgelesen werden. Ist der Meldungstyp bekannt, kann bei der Verarbeitung auf die Attribute zugegriffen werden, welche beim entsprechenden Meldungstyp übertragen werden.

4.2.2 Feature

Für die Abbildung von Weltphänomenen wurde die Schnittstelle (Interface) Feature erstellt. Ein Feature enthält einen Typ und kann eine Geometrie sowie zusätzliche Properties enthalten. Der Aufbau des Features basiert stark auf dem GeoJSON-Standard (vgl. Unterkapitel 2.4.3) und enthält dieselben Attribute wie jene die im GeoJSON-Standard für Feature Daten angegeben sind.

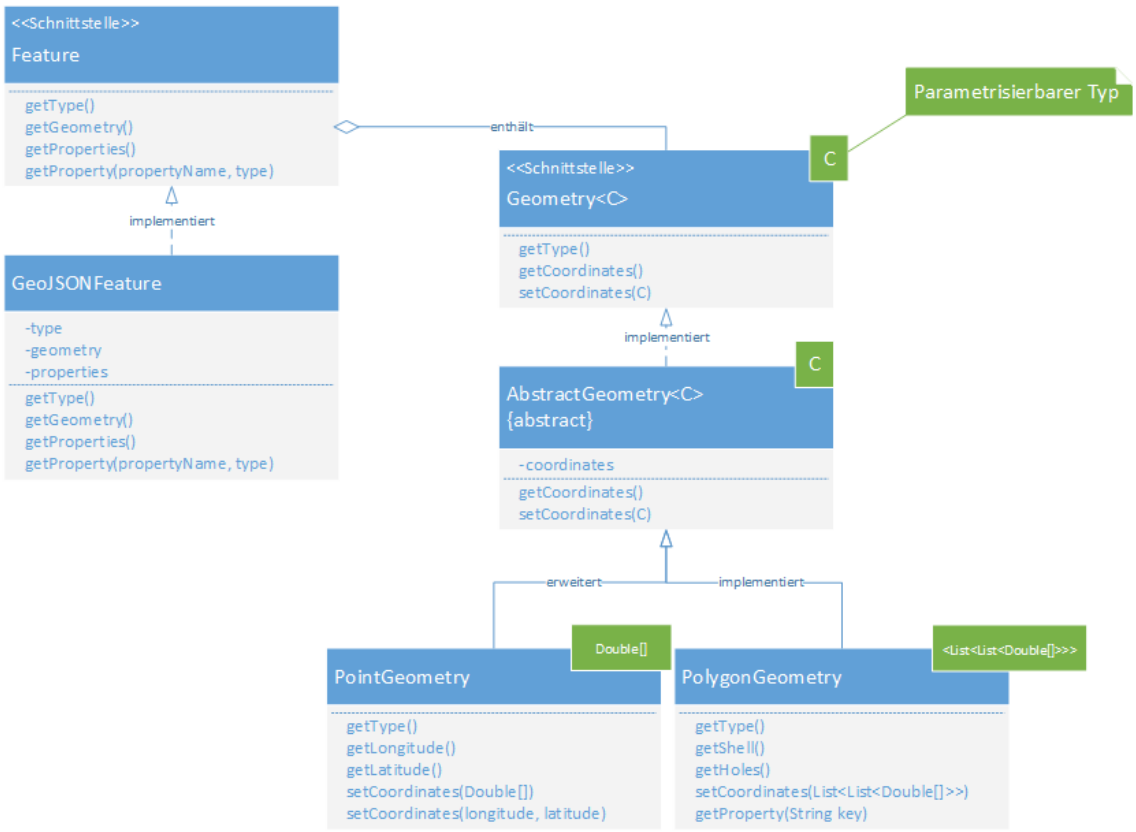

Abbildung 11: Klassendiagramm Feature

Wie der Abbildung 11 zu entnehmen ist, entspricht das Feature einer Schnittstelle, welche von der Klasse *GeoJSONFeature* implementiert wird. Der Name *GeoJSONFeature* wurde gewählt, um die Verbundenheit mit dem GeoJSON-Standard aufzuzeigen. Zusätzlich enthält die Klasse *GeoJSONFeature* verschiedene Jackson-Annotationen (vgl. Unterkapitel 4.1.4 FasterXML/Jackson), damit aus einem *GeoJSONFeature*-Objekt eine GeoJSON-Zeichenkette erzeugt werden kann.

Ein Feature kann eine Geometrieinstanz vom Typ *Geometry* enthalten, welcher wiederum als Schnittstelle definiert ist. Da die verwalteten Daten innerhalb einer Geometrie je nach Geometrie Typ unterschiedlich sein können, ist der Datentyp der Koordinaten parametrisierbar. Eine *AbstractGeometry* enthält den parametrisierbaren Datentyp *coordinates*. Die Klasse *PointGeometry* erweitert die *AbstractGeometry*-Klasse um die Schnittstelle *Geometry* komplett zu implementieren. Da die *PointGeometry* nur ein Koordinatenpaar speichern muss, wird in dieser Klasse der parametrisierbare Koordinatentyp auf ein Array mit Fliesskommazahlen (Double[]) gesetzt. Zusätzlich werden weitere Methoden implementiert um auf Koordinaten zugreifen zu können.

Eine weitere Implementierung des *AbstractGeometry* wurde für eine Polygon Geometrien (*PolygonGeometry*) erstellt, welche im Rahmen dieser Arbeit nicht verwendet wird, jedoch bei anderen Anwendungsfällen (z.B. für Regionen Filter) eine interessante Erweiterung

darstellen könnte.

Diese Feature-Repräsentation kann sehr generisch eingesetzt werden. Deshalb wird im Integrationssystem jeweils angestrebt, die Daten als Feature Daten zu verarbeiten, damit die Route nicht nur für ADS-B Daten sondern für jegliche Feature Daten verwendet werden kann.

4.3 Datenempfangs-Route

Die Aufgabe der Datenempfangs-Route ist es, sich mit dem ADS-B Empfänger zu verbinden, um die SBS1-Daten über den TCP-Kanal zu empfangen. Die empfangenen Daten sollen generalisiert werden. Hierfür werden die SBS-1 Objekte in Feature Objekte umgewandelt. Anschliessend sollen die Daten in diesem allgemein verständlichen Format an die SOS- wie auch an die Websocket-Route weitergeleitet werden. Es sollen nur Objekte weitergeleitet werden, welche neben dem *hexIdent*- und der *messageReceived*-Attribute zusätzliche, noch nicht erhaltene Werte enthalten. Objekte die keine neuen Daten beinhalten sollen ausgefiltert werden.

4.3.1 Ablauf

Beim Starten des Integrationssystems wird initial eine direkte TCP-Verbindung zum ADS-B Empfänger aufgebaut. Sobald TCP-Pakete empfangen werden, werden diese in die Datenempfangs-Route geschrieben. Empfangen werden kommaseparierte Zeichenketten, welche gemäss des SBS-1Socket Output (vgl. Unterkapitel 2.4.1) aufgebaut sind. Die Zeichenketten werden mittels Datenkonvertierung in ein Java-Objekt umgewandelt, welches für jede Spalte des kommaseparierten Datenformates ein entsprechendes Attribut beinhaltet. Das resultierende Java Objekt wird durch einen Umwandler in ein Feature-Objekt umgewandelt. Bei der Umwandlung in ein Feature-Objekt werden nur Werte in das Feature eingefügt, welche konfiguriert wurden. Dadurch werden bereits bei diesem Schritt Properties entfernt, welche nicht zum Client übertragen werden sollen (vgl. Unterkapitel 3.1 Zieldaten).

Der SBS1-Socket Output sendet teilweise Objekte mit dem gleichen Mode-S Identifier und exakt gleichem Empfangszeitpunkt mehrfach. Die Datenwerte können aber müssen nicht zwingend unterschiedlich sein. Es kann jedoch vorkommen, dass die gleichen Attribute mehrfach gesendet werden. Aus diesem Grund wird in einem nächsten Schritt geprüft, ob Properties mit dem gleichen Wert vom gleichen Mode-S Identifier und mit dem exakt gleichen Zeitstempel bereits verarbeitet wurden. Falls dem so ist, wird das Property auf dem Feature gelöscht. Im darauf folgenden Schritt wird geprüft, ob das Feature überhaupt noch neue Werte enthält oder nicht. Falls nicht, wird die Route für diese Meldung beendet. Falls das Feature neue Werte hat, wird in einem weiteren Schritt das Feature mittels *multicast*-Aufruf an die SOS-Route wie auch an die Websocket-Route gesendet. Durch den *multicast*-Aufruf wird die Ausführung der beiden Routen mit dem gesendeten Java Objekt als Input initiiert. Dies ermöglicht eine quasi parallele Verarbeitung der beiden Routen.

In Abbildung 12 wird der zuvor beschriebene Ablauf innerhalb der Datenempfangs-Route anhand eines Sequenzdiagrammes dargestellt.

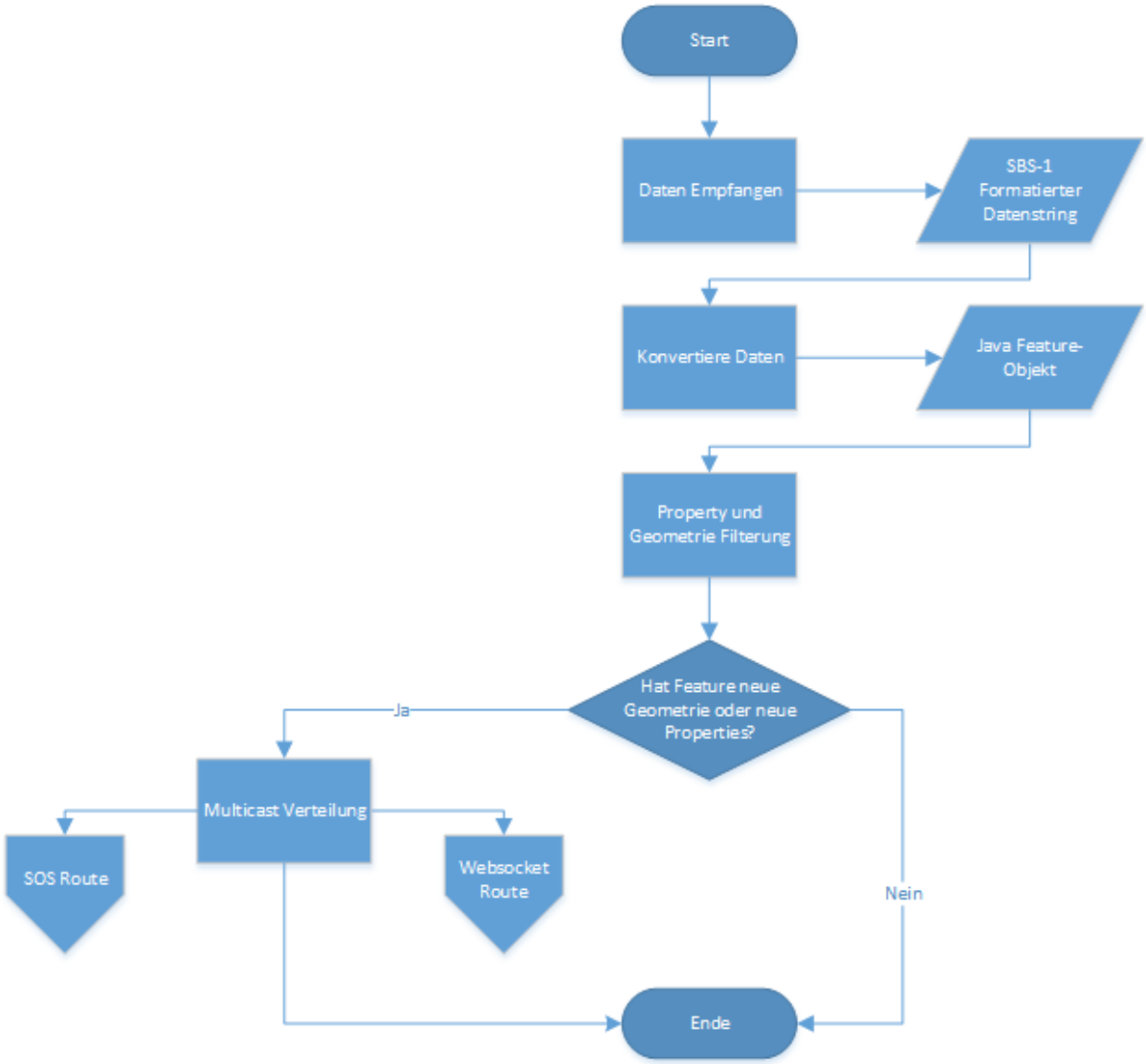

Abbildung 12: Ablauf Datenempfangs-Route

4.3.2 Implementierung

Die Datenempfangsroute basiert auf einem Github Projekt[11] von Andreas Kuhtz, mit welchem Daten von einer SBS-1 Basisstation empfangen werden können. Für den Aufbau der TCP-Verbindung zum ADS-B Empfänger setzt Kuhtz auf das Netty[12] Rahmenwerk, welches eine asynchrone und eventbezogene Verarbeitung von Netzwerkdaten ermöglicht. Der Verbindungsaufbau zum ADS-B Empfänger erfolgt mit Netty ausserhalb der Camel-Route.

11 camel-dvb-t - https://github.com/akuhtz/camel-dvb-t [letzter Zugriff 24.01.2015]
12 Netty:Home - http://netty.io/ [letzter Zugriff 25.01.2015]

Da die Verbindung zum ADS-B Empfänger nicht direkt mit dem Camel-Rahmenwerk erfolgt, werden die empfangenen Daten in einer Netty-Verarbeitungsroutine in eine Camel *direct*-Komponente geschrieben. Eine *direct*-Komponente kann innerhalb einer Route wiederum als Input wie auch als Output verwendet werden, um Daten innerhalb eines Camel Contextes über verschiedene Routen oder Verarbeitungsroutinen zu versenden.

Die Datenempfangs-Route wird folgendermassen im Camel-Context initialisiert:

```
...
<bean id="bindyDataformat"
  class="org.apache.camel.dataformat.bindy.csv.BindyCsvDataFormat">
  <constructor-arg value="ch.trackdata.sbs1route.message.SBS1Message" />
</bean>

<bean id="filterDuplicateValueProcessor"
  class="ch.trackdata.sbs1route.FilterDuplicateValueProcessor">
  <property name="cleanupInterval" value="${filter.duplicates.cleanupInterval}" />
  <property name="idProperty">
    <util:constant static-field=
     "ch.trackdata.sbs1route.message.TrackPositionMessage.HEX_PROPERTY_NAME"/>
  </property>
  <property name="comparableDateProperty">
    <util:constant static-field=
   "ch.trackdata.sbs1route.message.TrackPositionMessage.DATE_TIME_MESSAGE_GENERATED_NAME"/>
  </property>
</bean>

...
<camel:route autoStartup="true">
  <camel:from uri="direct:sbs1" />
    <camel:unmarshal ref="bindyDataformat" />
    <camel:process ref="sbs1Processor" />
    <convertBodyTo type="ch.trackdata.sbs1route.message.GeoJSONFeature" />
    <camel:process ref="filterDuplicateValueProcessor"></camel:process>
    <camel:when>
      <simple>${body.geometry} != null
        || ${body.properties.size} >= ${properties:filter.minproperties}</simple>
        <camel:log message="${body}" logName="RECEIVED_FEATURE" />
        <camel:multicast stopOnException="false">
          <camel:to uri="direct:websocket" />
          <camel:to uri="direct:sos" />
        </camel:multicast>
    </camel:when>
    <camel:otherwise>
      <camel:stop />
    </camel:otherwise>
</camel:route>
```

Die Rohdaten der ADS-B Verbindung werden von Netty an die Camel-Komponente *direct:sbs1* gesendet. Für die Datenempfangs-Route wird diese Komponente als Producer *direct:sbs1* als Datenquelle verwendet. Die Daten werden mittels dem Camel *BindyCsvDataFormat* in ein kommasepariertes Objekt der Klasse *SBS1Message* umgewandelt. Im *sbs1Processer* wird lediglich die Meldung extrahiert. Die von Kuhtz erstellte Route wurde für diese Arbeit um die folgenden Anweisungen erweitert:

Die *convertBodyTo*-Anweisung wandelt die SBS1-Daten um. Durch die Angabe des Typs auf *GeoJSONFeature* wird ein *Converter* gesucht, welcher die Umwandlung in *GeoJSONFeature*

ermöglicht. Ein entsprechender *Converter* wurde erstellt und in Camel registriert. Der *Converter* ist konfigurierbar und erlaubt es, dass nur gewisse Attribute in ein Feature übernommen werden.

Die darauf folgende *process*-Anweisung entfernt Properties, welche bereits übertragen wurden aus dem Feature. Hierfür wurde ein *camel:Processor* geschrieben, welcher die empfangenen Meldungen der letzten paar Sekunden in einem Cache führt. Beim Eintreffen einer neuen Meldung wird geprüft, ob das Feature mit dem Identifier und dem entsprechenden Erzeugungszeitpunkt bereits existiert. Existiert bereits ein solches Feature, wird dem neuen Feature alle *properties,* welche bereits übertragen wurden entfernt.

Im nächsten Schritt wird durch eine *camel:simple*-Anweisung geprüft, ob die Geometrie des Features ungleich *null* ist oder ob die Menge der Properties des Features grösser als oder gleich einem konfigurierbaren Wert ist. Ist dies der Fall, wird ein *multicast*-Aufruf gestartet. Der *multicast*-Aufruf ermöglicht es, die Meldungen an die Routen der zwei Varianten parallel weiterzuleiten.

Die empfangenen SBS1-Meldungen werden somit als *Feature*-Datensätze an die Websocket und an die SOS-Route übertragen.

4.3.3 Konfiguration

Die Datenempfangs-Route kann über zwei verschiedene Arten konfiguriert werden. Zum einen über die Konfigurationsdatei, welche jene Konfigurationen enthält, die je nach Laufzeitumgebung angepasst werden müssen. Zum andern über die Konfiguration eines Spring-Beans innerhalb des Application Contextes. Die Einstellung im Application Context beinhaltet Konfigurationsmöglichkeiten, welche sich nicht oder nur sehr selten ändern.

Konfigurationsdatei

Beim Starten des Integrationssystems wird jeweils eine Konfigurationsdatei (Dateiname: *sbs1route.properties*) ausgelesen, in welcher verschiedene Parameter der Route übergeben werden können. Die Datei muss sich im selben Verzeichnis befinden, aus welchem das Integrationssystem gestartet wird. Die Datenempfangsroute kann mit folgenden Einstellungsmöglichkeiten konfiguriert werden:

Property	Typ	Erläuterung
sbs1.enabled	Boolescher Wert	Wird dieser Wert auf *true* gesetzt, wird eine Verbindung zum angegebenen SBS1-Host initiiert um Meldungen vom SBS1-Host zu erhalten. Ist der Wert auf *false* gesetzt, wird keine Verbindung zum SBS1-Host aufgebaut. In diesem Fall werden keine Meldungen empfangen.

sbs1.host	Zeichenkette	Angabe des Hostnamens oder der IP-Adresse der SBS1 Station, zu welcher eine Verbindung aufgebaut werden soll (z.B. *sbs1station15*).
sbs1.port	Integer	Angabe des Netzwerk Ports auf welchem der SBS (BaseStation) Socket Output ausgegeben wird (z.B. 30003).
filter.minproperties	Integer	Definiert die Anzahl Properties die ein Feature haben muss, damit es an die SOS- und die Websocket-Route übertragen wird. Zu beachten ist, dass sowohl der Identifier als auch der *messageGenerated*-Zeitpunkt auch berücksichtigt werden müssen. Somit ist es sinnvoll, diesen Wert auf 3 zu setzen.
filter.duplicates.cleanupInterval	Integer	Definiert den Zeitraum, wie lange Feature Daten zwischengespeichert werden sollen. Nach Ablauf dieser Zeit werden die alten Objekte aus dem Cache gelöscht (z.B. 10000).

Tabelle 18: Konfigurationsmöglichkeiten Datenempfangsroute

Der Inhalt der Konfigurationsdatei zur Konfiguration dieser Route könnte folgenden Inhalt haben:

```
sbs1.enabled=true
sbs1.host=192.168.1.4
sbs1.port=30003

filter.minproperties=3
filter.duplicates.cleanupInterval=10000
```

Spring Bean

Wie bei der Implementierung der Route beschrieben, kann die Filterung der Properties konfiguriert werden. Die Konfiguration erfolgt über zwei Beans. Im *sbs1PropertyNamePredicate*-Bean kann ein Filter definiert werden, welcher entscheidet, ob ein Property bei der Umwandlung von einer SBS1-Message in ein Feature-Objekt übernommen werden soll oder nicht. Das Bean muss vom Typ *Predicate* sein, welches eine Klasse der Apache Commons Collection[13] ist. Ein *Predicate* muss eine Methode implementieren:

```
public interface Predicate<T> {
```

13 Apache Commons Collections - https://commons.apache.org/ [letzer Zugriff 25.06.2015]

```
    boolean evaluate(T object);

}
```

Bei der Umwandlung wird dem *Predicate* der Attributname zur Evaluation übergeben. Soll das Attribut in das Feature übernommen werden, soll *true* und falls nicht, soll die *evaluate*-Methode als Rückgabewert *false* zurückgeben. Die Konfiguration des Beans wurde so umgesetzt, dass nur Werte der gewünschten Zieldaten (vgl. Unterkapitel 3.1 Zieldaten) übernommen werden. Die Konfiguration des Beans sieht für das Vergleichssystem folgendermassen aus:

```
<bean id="sbs1PropertyNamePredicate" class="org.apache.commons...AnyPredicate">
  <constructor-arg index="0">
    <list>
      <bean class="org.apache.commons.collections4.functors.EqualPredicate">
        <constructor-arg index="0" value="hexIdent" />
      </bean>
      ...
      <bean class="org.apache.commons.collections4.functors.EqualPredicate">
        <constructor-arg index="0"  value="groundSpeed" />
      </bean>
      <bean class="org.apache.commons.collections4.functors.EqualPredicate">
        <constructor-arg index="0"  value="messageGenerated" />
      </bean>
    </list>
  </constructor-arg>
</bean>
```

Das *sbs1PropertyNamePredicate*-Bean wird als Predicate-Implementierung *AnyPredicate* konfiguriert. Ein *AnyPredicate* ist dann gültig, wenn eines der im *AnyPredicate* enthaltenen *Predicate* gültig ist. Falls alle ungültig sind, wird das Predicate als *false* evaluiert. Dem *Any Predicate* wird eine Liste von *Predicates* übergeben. Diese enthält für jedes Property der Zieldaten ein *EqualPredicate*. Dem *EqualPredicate* wird schliesslich der Propertyname übergeben. Das *EqualPredicate* gibt nun nur *true* zurück, falls der zu prüfende Wert dem übergebenen Propertynamen entspricht. Das *sbs1PropertyNamePredicate*-Predicate filtert somit alle Properties welche nicht in einem *EqualPredicate* definiert sind.

Das *sbs1ValuePredicate*-Bean ist das zweite Bean, welches für die Filterung der Attribute verwendet werden kann. Wie der Name bereits vermuten lässt, wird dieses Predicate auf die Werte der SBS1-Message angewendet. Somit kann anhand des Wertes entschieden werden, ob ein Property in das Feature eingefügt wird oder nicht. Wie im Unterkapitel Zieldaten beschrieben, sollen nur Werte in das Feature übernommen werden, welche nicht *null* sind. Das *sbs1ValuePredicate*-Bean soll somit als Predicate konfiguriert werden, welches auf ungleich *null* prüft:

```
<util:constant id="sbs1ValuePredicate"
  static-field="org.apache.commons.collections4.functors.NotNullPredicate.INSTANCE"/>
```

Das *NotNullPredicate*-Predicate, welches in der Apache Commons-Bibliothek enthalten ist,

gibt nur *true* zurück, falls der zu prüfende Wert ungleich *null* ist. Dadurch werden nur Properties in die Features übernommen, welche ungleich *null* sind.

4.4 SOS-Route

Mit Hilfe der SOS-Route sollen die Daten der *Feature*-Datensätze in *SOS:InsertObservation* XML-Statements umgewandelt und mittels eines Aufrufes an den SOS-Server gesendet und somit eingefügt werden. Die Umsetzung erfolgt gemäss des Konzepts der SOS-Variante, welches in Unterkapitel 3.2.1 vorgestellt wurde.

4.4.1 Ablauf

Der Ablauf der SOS-Route entspricht dem Sequenzdiagramm aus Abbildung 13. Der Route werden *Feature*-Objekte übergeben. Die Umwandlung in eine *SOS:InsertObservation* wurde so implementiert, dass konfigurierbar ist, welche Attribute eines Features in die Umwandlung einbezogen werden. In einem ersten Schritt werden die Featuredaten in ein Objekt umgewandelt, welches eine *SOS:InsertObservation* XML-Datenstruktur repräsentiert. Diese Objekte werden im folgenden Schritt durch *Marshalling* in eine Zeichenkette umgewandelt. Die entstandene Zeichenkette, entspricht nun einer *SOS:InsertObservation* XML-Repräsentation. Durch einen HTTP-Post Request wird nun diese Zeichenkette an den SOS-Server gesendet.

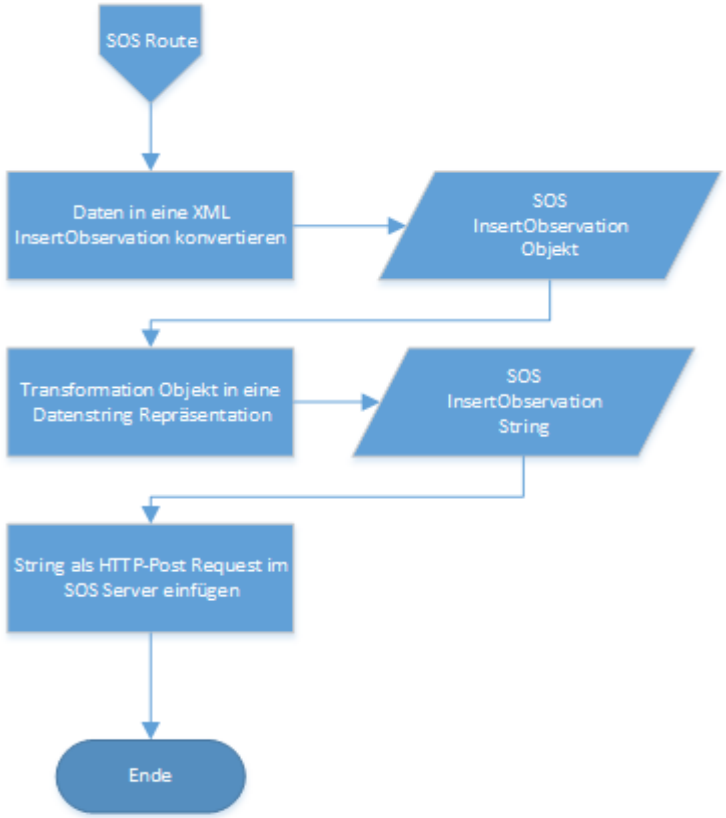

Abbildung 13: Ablauf SOS-Route

4.4.2 Implementierung

Die SOS-Route ist eine Eigenentwicklung, welche im Rahmen dieser Arbeit entstanden ist. Die SOS-Route empfängt die Daten, welche durch die Datenempfangs-Route in den Camel-*Endpoint direct:sos* geschrieben werden. In einem ersten Schritt wird mittels einer *when*-Anweisung die Konfigurationseinstellung, ob die SOS-Variante aktiviert ist oder nicht, überprüft. Nur falls die Variante aktiviert ist, werden die Feature-Objekte durch die *convertBodyTo*-Anweisung in *JAXBElement*-Objekte umgewandelt. Im darauf folgenden Schritt wird eine *marshall*-Anweisung aufgerufen, welche die zuvor erstellten JAXBElemente in einen XML-Text umwandeln soll. Durch die Angabe des *prettyPrint*-Parameter auf *false* wird sichergestellt, dass keine unnötigen Leerzeichen und Zeilenumbruche beim Umwandeln in die XML-Zeichenkette eingefügt werden und somit die Datenmenge der zu übertragenden Zeichenkette unnötig vergrössert wird.

Die darauf folgenden Befehle innerhalb dieser Camel-Route führen die jeweiligen Anfragen an den SOS-Server aus. Mittels eines *setHeader*-Befehls können Metadaten einer Meldung gesetzt werden. Die *CamelHttpMethod* wird auf POST gesetzt, damit ein HTTP-Post Request ausgeführt werden kann. Durch die Angabe des *Content-Types*, der bei der Meldung im

HTTP-Header aufgeführt wird, wird sichergestellt, dass der SOS über den Datentyp des Inhalts des Requests informiert ist.

```
<camel:route>
  <camel:from uri="direct:sos" />
  <camel:choice>
    <camel:when>
      <simple>${properties:sos.enabled}</simple>
      <convertBodyTo type="javax.xml.bind.JAXBElement" />
      <marshal>
        <jaxb id="myJaxb" prettyPrint="false" ... />
      </marshal>
      <setHeader headerName="CamelHttpMethod">
        <constant>POST</constant>
      </setHeader>
      <setHeader headerName="Content-Type">
        <simple>application/xml</simple>
      </setHeader>
      <log message="${body}" logName="SOS_INSERTOBSERVATION" />
      <to uri="sosEndpoint" />
    </camel:when>
      <camel:otherwise>
        <camel:stop />
      </camel:otherwise>
  </camel:choice>
</camel:route>
```

Zum Schluss wird durch die Angabe der *to*-Anweisung angegeben, welcher Camel-Endpoint als Datensenke dienen soll. In diesem Fall wurde ein Spring Bean mit dem Namen *sosEndpoint* definiert, welcher die URL und den Pfad des SOS beinhaltet. Durch die Umrandung von Zeichenketten mittels "${}" wird auf die Einstellungen innerhalb der Konfigurationsdatei verwiesen:

```
<camel:endpoint id="sosEndpoint"
  uri="http://${sos.host}:${sos.port}${sos.path}" />
```

4.4.3 Konfigurationsmöglichkeiten

Die SOS-Route kann wie auch die Datenempfangs-Route über die Konfigurationsdatei und über die Konfiguration eines Spring-*Beans* innerhalb des Application Contextes konfiguriert werden:

Konfigurationsdatei

Wie auch bei der Datenempfangsroute können in der Konfigurationsdatei *sbs1route.properties* Einstellungen für die SOS-Route vorgenommen werden. Diese Art der Konfiguration ist angedacht für die Einstellung von Attributen, welche sich oft ändern können (z.B. neuer Servername oder die SOS-Route temporär deaktivieren). Nachfolgend sind die verschiedenen Konfigurationsmöglichkeiten aufgelistet:

Property	Typ	Erläuterung
sos.enabled	Boolescher Wert	Wird dieser Wert auf *true* gesetzt, werden innerhalb der

		SOS-Route die von den SOS-Routen empfangenen Meldungen nicht weiterverarbeitet.
sos.host	Zeichenkette	Angabe des Hostnamens oder der IP-Adresse des Servers, auf welchem der SOS Dienst läuft (z.B. *sosserver1*).
sos.port	Integer	Angabe des Netzwerk Ports, auf welchem der SOS-Dienst auf dem angegebenen Host läuft.
sos.path	Zeichenkette	HTTP-Pfad auf dem SOS-Server, über welchen auf den SOS-Dienst zugegriffen werden kann (z.B. */sos/pox*)

Tabelle 19: Konfigurationsmöglichkeiten SOS-Route

Eine mögliche Konfiguration dieser aufgelisteten Einstellungen könnte in der *sbs1route.properties* Konfigurationsdatei folgendermassen aussehen:

```
sos.enabled=true
sos.host=127.0.0.1
sos.port=8080
sos.path=/52n-sos-webapp/sos/pox
```

Spring Bean

Wie bereits zuvor erwähnt, wurden die SOS-Routen so entwickelt, dass diese nicht nur für SBS1-Meldungen sondern für alle möglichen Feature-Meldungen verwendet werden können. Mit Hilfe dieser Spring Bean-Konfiguration kann eingestellt werden, in welche Offerings die *SOS:InsertObservation*-Anfrage geschrieben werden soll und welche Attribute eines Features als *ObservedProperty* in einer *SOS:InsertObservation*-Anfrage enthalten sein sollen.

Die Umwandlung eines Features in ein *JAXBElement* erfolgt über einen selbstgeschriebenen *Converter*. Der *Converter* referenziert das *insertObservationConfiguration*-Bean.

Dieses Bean kann im Application Context so angepasst werden, dass die gewünschten Attribute in die *SOS:InsertObservation* einbezogen werden. Das Spring-Bean muss eine Instanz der Klasse *InsertObservationSOSV2Configuration* beschreiben, welche mit dem Namen *insertObservationConfiguration* bezeichnet werden soll. Die Schnittstelle *ObserverPropertyConfiguration* enthält genügend Funktionen um die Umwandlung eines Feature Attributes in eine *ObservedProperty* Beschreibung durchzuführen. Eine *InsertObservationSOSV2Configuration* kann keine, eine oder mehrere ObservedProperties beinhalten.

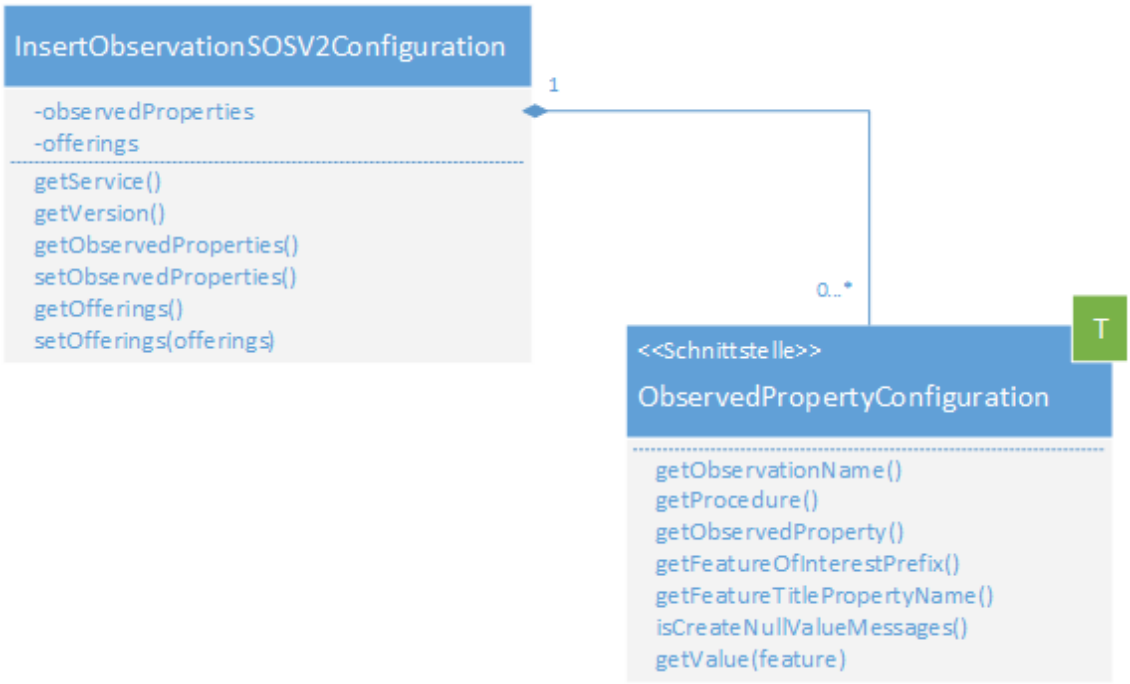

Abbildung 14: Insert Observation SOS Konfiguration

Für die Umwandlung der *Feature*-Objekte, welche aus der *SBS1Messages* stammen, sollen die in den Zieldaten (vgl. Unterkapitel 3.1 Zieldaten) beschriebenen Properties in den SOS geschrieben werden.

Für jedes dieser Attribute muss somit eine entsprechende *ObservedPropertyConfiguration*-Instanz in der Spring-Konfiguration erstellt werden. Das *insertObservationConfiguration*-Bean sieht für die Konfiguration der zuvor definierten Attribute folgendermassen aus:

```
<bean id="insertObservationConfiguration"
class="ch.trackdata.sbs1route.converter.insertobservation.InsertObservationSOSV2Configurati
on">
  <property name="offerings">
    <list>
      <value>http://stue.ch/sensorobservation/offering/ads-b</value>
    </list>
  </property>
  <property name="observedProperties">
    <list>
      <ref bean="callsignObservationConfiguration" />
      <ref bean="speedObservationConfiguration" />
      <ref bean="altitudeObservationConfiguration" />
      <ref bean="headingObservationConfiguration" />
      <ref bean="positionObservationConfiguration" />
    </list>
  </property>
</bean>
```

Wie dem Spring-XML Auszug zu entnehmen ist, wird für die Konfiguration der *observedProperties* auf andere Spring-Beans referenziert. Da die einzelnen Attribute unterschiedliche Datentypen umfassen (Zeichenkette, Zahlenwert oder Geometrie), müssen

diese Beans unterschiedliche Implementierungen der *ObservedPropertyConfiguration* -Schnittstelle verwenden. Im Rahmen dieser Arbeit wurden verschiedene Implementierungen erstellt, mit welchen diese unterschiedlichen Attribut-Typen beschrieben werden können (vgl . Abbildung 15).

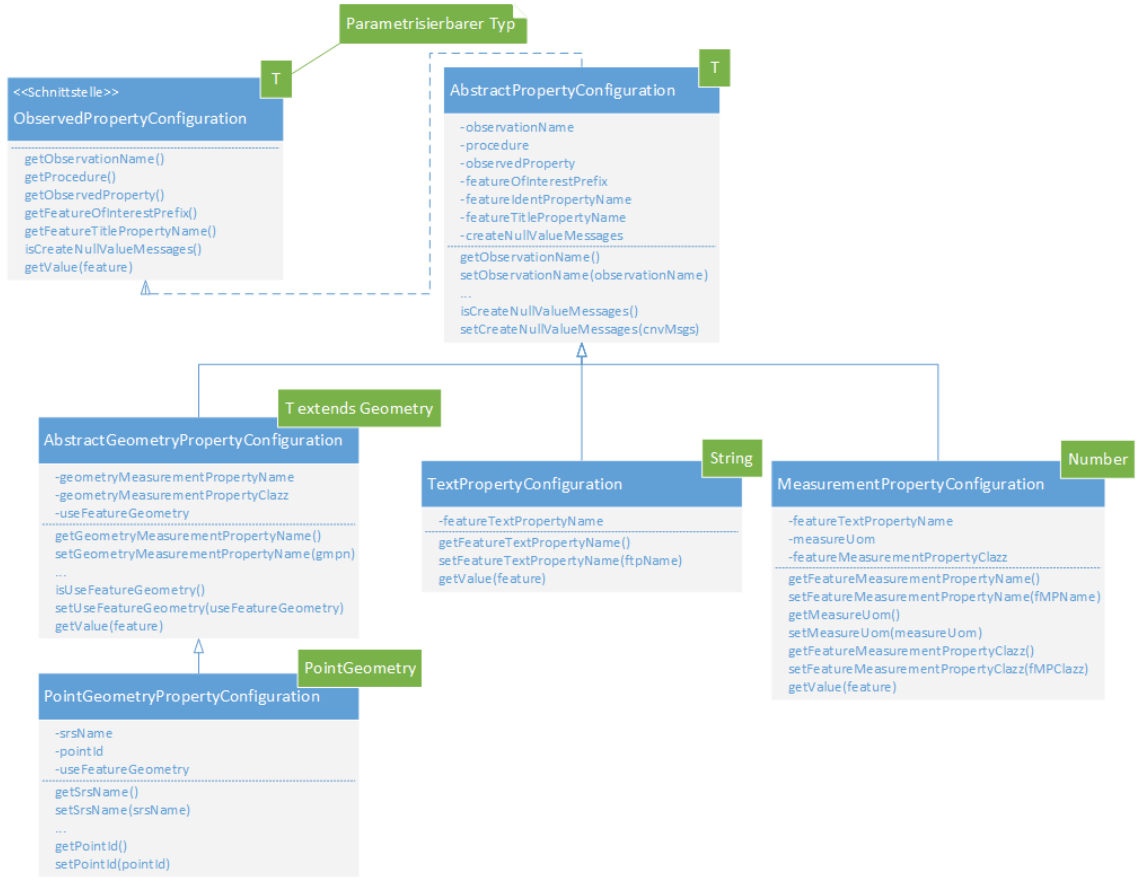

Abbildung 15: ObservedPropertyConfiguration Implementierungen

Es wurden bisher drei Implementierungen der *ObservedPropertyConfiguration* erstellt. Die verschiedenen Implementierungen sind für den jeweils angegebenen Datentyp geeignet. Als Beispiel werden die Attribute der Flugzeug-Positionsdaten aufgelistet, welche mit der entsprechenden Implementierung verwendet werden sollen:

Implementierung	Datentyp	Beispiel Attribute
TextPropertyConfiguration	Zeichenketten	Call Sign
MeasurementPropertyConfiguration	Zahlenwerte	Speed, Altitude, Heading
PointGeometryPropertyConfiguration	Punkt Geometrie	Geometrie

Tabelle 20: SOS Property Konfiguration Datentypen

Die Spring-Konfiguration für die angegebenen Attribute ist umfangreich und sieht für die verschiedenen Attribute ähnlich aus (vgl. Anhang A.2 Bean-Konfiguration SOS-Variante).

Aus diesem Grund wird exemplarisch nur die *ObservedPropertyConfiguration* für das Speed-Attribut genauer beschrieben. Das Bean wird mit der ID *speedObservationConfiguration* bezeichnet. Auf dieses Bean wurde unter anderem bereits im Code-Ausschnitt der *insertObservationConfiguration* referenziert:

```
<bean id="featureOfInterestPrefix" class="java.lang.String">
  <constructor-arg value="http://stue.ch/sensorobservation/foi/aircraft/" />
</bean>

<bean id="speedObservationConfiguration" class=
    "ch.trackdata.sbs1route.converter.insertobservation.MeasurementPropertyConfiguration">
  <property name="observationName" value="speedObservation" />
  <property name="procedure"
    value="http://stue.ch/sensorobservation/procedure/flighttracking" />
  <property name="observedProperty"
    value="http://stue.ch/sensorobservation/observableProperty/speed" />
  <property name="featureOfInterestPrefix" ref="featureOfInterestPrefix" />
  <property name="featureIdentPropertyName" value="hexIdent" />
  <property name="featureTitlePropertyName" value="hexIdent" />
  <property name="measureUom" value="[kn_i]" />
  <property name="featureMeasurementPropertyName" value="groundSpeed" />
  <property name="featureMeasurementPropertyClazz" value="java.lang.Integer" />
  <property name="createNullValueMessages" value="false" />
</bean>
```

Da die Geschwindigkeit in Knoten gemessen wird, wird ein *MeasurementPropertyConfiguration*-Bean für die *speedObservationConfiguration*, verwendet. Diese Klasse eignet sich, um gemessene Zahlenwerte als Observation darstellen zu lassen. Der Name der Observation wird als *speedObservation* bezeichnet und wird lediglich für die Unterscheidung der verschiedenen Observationen verwendet. Die Werte der Procedure und des Offerings werden gemäss des Konzepts der SOS-Route befüllt (vgl. Unterkapitel 3.2.1 SOS-Variante).

Durch die Angabe der verschiedenen Property-Namen (*featureIdent* und *featureTitle*) wird angegeben, wie der Name der Attribute auf den Features, von welchem die entsprechenden Informationen ausgelesen werden sollen, lautet. In diesem Fall wird als Identifier und für den Titel eines Feature-Of-Interest auf den pro Flugobjekt eindeutigen *hexIdent*-Wert verwiesen.

Durch die Angabe des *featureMeasurement*-Property-Namen und der Property-Klasse werden der Name und die Klasse des Attributs, welches die Messdaten enthält, ausgelesen. Innerhalb der Measurement Konfiguration wird in diesem Beispiel bei der Umwandlung von Feature zu einer *SOS:InsertObservation* der Wert auf folgende Art und Weise ausgelesen:

```
Integer value = feature.getProperty("groundSpeed", Integer.class);
```

Durch die Angabe der *measureUom* wird die Einheit der entsprechenden Messwerte gemäss des Unified Code for Units of Measure (UCUM) (Schadow and McDonald, 2009) angegeben. Die Geschwindigkeit wird bei Flugobjekten in Knoten gemessen.

Der letzte Konfigurationspunkt gibt an, ob für Attribute, bei welchen kein Wert gesetzt ist,

eine Observation erzeugt werden soll oder nicht. Da wir nur Elemente in den SOS einfügen wollen, die aktuell geändert wurden und somit im Feature nicht leer sind, wird dieser *createNullValueMessages*-Wert bei allen Property-Konfigurationen auf *false* gesetzt. Dadurch wird eine Observation für diesen Typ nur erstellt, falls das Property, welches umgewandelt werden soll, nicht einen *null*-Wert beinhaltet.

4.5 Websocket-Route

In der Websocket-Route werden die aus der Datenempfangs-Route erstellten *Feature*-Objekte in GeoJSON-Zeichenketten umgewandelt und in ein Websocket geschrieben. Die Umsetzung erfolgt gemäss dem in Unterkapitel 3.2.2 vorgestellten Konzepts zur Umsetzung der Websocket-Variante.

4.5.1 Ablauf

Innerhalb der Websocket-Route wird lediglich das *Feature*-Objekt in eine Zeichenkette umgewandelt. Diese Umwandlung von einem Java Objekt in eine Zeichenkette erfolgt durch einen registrierten *Converter*. Schlussendlich wird dieser resultierende Text als einzelne Meldungen in alle aktuell mit dem Integrationssystem verbundenen Websockets geschrieben.

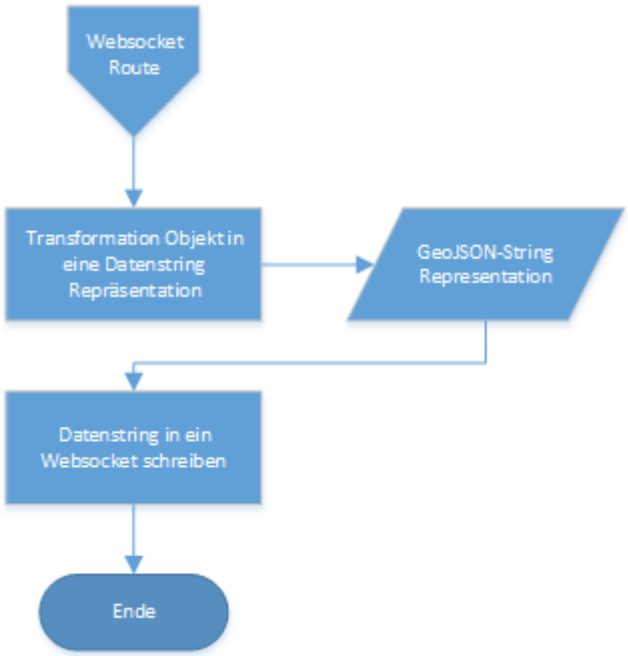

Abbildung 16: Ablauf Websocket-Route

4.5.2 Implementierung

Die Implementierung dieser Route ist weniger umfangreich als jene der SOS-Route, da die Umwandlung in GeoJSON nicht so aufwendig ist wie die Umwandlung in eine XML-*SOS:InsertObservation*.

Die URL zum Websocket Endpoint der Websocket-Route wird im Bean *websocketClientJSONEndpoint* beschrieben. Der Endpoint ist vom Typ *filterwebsocket,* was ein im Rahmen dieser Arbeit entstandener Endpoint-Typ ist. Der *filterWebsocket*-Typ erweitert den Endpoint-Typ Websocket mit der Möglichkeit, Meldungen abhängig von einem Zustand von einem über Websocket verbundenen Client zu Filtern. So kann ein Webclient über Websocket eine Meldung mit einem Filter an das Integrationssystem senden. Das Integrationssystem filtert dann alle Meldungen gemäss des übergebenen Filters, so dass dieser Client nur Daten empfängt, welche dem Filter entsprechen. Dieser *filterwebsocket*-Endpoint entstand als Prove-Of-Concept im Rahmen dieser Arbeit. Es wird jedoch für den Vergleich der SOS und der Websocket-Variante nicht benötigt. Aus diesem Grund wird nicht weiter auf diese Funktionalität eingegangen.

Wie auch bei der SOS-Route wird anhand eines Flags überprüft, ob die Meldungen an den Endpoint übertragen werden sollen. Ist das Flag auf True gesetzt, wird die Meldung in das *websocketClientJSONEndpoint*-Bean geschrieben. Im anderen Fall wird die Route direkt beendet.

```
<camel:endpoint id="websocketClientJSONEndpoint"
  uri="filterwebsocket://${websocket.host}:${websocket.port}${websocket.path}?
websocketStore=#websocketStore&amp;sendToAll=true" />
<camel:route>
  <camel:from uri="direct:websocket" />
  <camel:when>
    <simple>${properties:websocket.enabled}</simple>
    <camel:to ref="websocketClientJSONEndpoint" />
  </camel:when>
  <camel:otherwise>
    <camel:stop />
  </camel:otherwise>
</camel:route>
```

Beim Schreiben in das Websocket werden die Features automatisch in Zeichenketten umgewandelt. Hierfür muss jedoch ein entsprechender *Converter* in Camel registriert sein. Der Konverter, der hierfür erstellt wurde, ist von der Klasse *GeoJSONFeatureToStringConverter*. Da die Klasse *GeoJSONFeature* mit Jackson Annotationen (vgl. Unterkapitel 4.1.4 FasterXML/Jackson) ergänzt ist, kann diese mittels einem Jackson *ObjectMapper* umgewandelt werden.

```
ObjectMapper mapper = new ObjectMapper();
...
@Converter
public String convert(GeoJSONFeature<?> geoJSONFeature) {
  try {
    return mapper.writeValueAsString(geoJSONFeature);
  }
  catch (JsonProcessingException e) {
    //Do exception handling
  }
}
```

Die Klasse enthält einen Jackson Object Mapper, auf welchem das übergebene Feature mittels der *WirteValueAsString*-Methode in eine Zeichenkette umgewandelt wird.

4.5.3 Konfigurationsmöglichkeiten

In der Websocket-Route können nur wenige Attribute mittels Konfigurationsdatei konfiguriert werden. Dennoch ist die Websocket-Route generell einsetzbar. Die Klassen der in die Websocket-Route geschriebenen Objekte müssen lediglich mit Jackson-Annotationen ergänzt werden um als JSON-Objekt in das Websocket geschrieben zu werden.

In der Konfigurationsdatei *sbs1route.properties* können wie bei der Datenempfangsroute und der SOS-Routen Einstellungen zu der Websocket-Route vorgenommen werden. Die Einstellungen sind hauptsächlich für dynamische Konfigurationen angedacht, welche je nach Umgebung unterschiedlich sein können. Die folgenden verschiedenen Konfigurationsmöglichkeiten werden unterstützt:

Property	Typ	Erläuterung
websocket .enabled	Boolescher Wert	Wird dieser Wert auf *true* gesetzt, werden innerhalb der Websocket-Route die von den Websocket-Routen empfangenen Meldungen nicht weiterverarbeitet.
websocket.host	Zeichenkette	Angabe des Hostnamens oder der IP-Adresse des Servers, auf welchen sich Websocket Clients verbinden können. **Vorsicht:** Clients können sich nur auf diese angegebene IP-Adresse verbinden. Wird beispielsweise 127.0.0.1 (localhost) angegeben, kann von keinem anderen Client über das Netzwerk auf das Websocket zugegriffen werden.
websocket.port	Integer	Angabe des Netzwerk Ports auf welchen sich Clients mit dem lokalen Websocket verbinden können.
websocket.path	Zeichenkette	Pfad des Websockets auf welchen sich Websocket Clients verbinden können (z.B. */clientTrackData*)

Tabelle 21: Konfigurationsmöglichkeiten Websocket-Route

Eine mögliche Konfiguration dieser aufgelisteten Einstellungen könnte in der *sbs1route.properties* Konfigurationsdatei folgendermassen aussehen:

```
websocket.enabled=true
websocket.host=192.168.1.136
websocket.port=8443
websocket.path=/clientTrackData
```

Mit dieser Konfiguration können sich Clients auf folgende Websocket URL verbinden (sofern das Integrationssystem gestartet ist):

```
ws://192.168.1.136:8443/clientTrackData
```

5. Umsetzung Webclient

In diesem Kapitel wird die technologische Umsetzung des Webclients beschrieben, welcher im Rahmen dieser Arbeit entwickelt wurde um die Sensordaten empfangen und visualisieren zu können. In der Clientanwendung werden die durch das Integrationssystem verteilten Daten empfangen und auf einer Kartenkomponente visualisiert. Die Webanwendung kann sowohl Daten von einem SOS-Server beziehen sowie Websocket Daten empfangen. Im Unterkapitel 5.1 werden die JavaScript-Bibliotheken vorgestellt, welche im Webclient verwendet werden. Der Grundaufbau der Client-Applikation wird im Unterkapitel 5.2 erläutert. Das Unterkapitel 5.3 befasst sich mit der Umsetzung des Empfangs der Daten im Webclient. Im darauf folgenden Unterkapitel (5.4) wird der Map Controller vorgestellt, welcher für die clientseitige Aufbereitung der Daten zuständig ist. Im Unterkapitel 5.5 wird schliesslich erläutert, wie die Darstellung der empfangenen Daten umgesetzt wurde. Die darauf folgenden Unterkapitel 5.6 und 5.7 beschreiben, wie empfangene Daten unterschiedlich gerendert werden können und wie die Kommunikation des Menüs mit den Diensten und den Controllern erfolgt.

5.1 Verwendete JavaScript-Bibliotheken

Der Webclient basiert auf AngularJS. Über AngularJS ist der Aufbau der Applikation zu einem gewissen Grad gegeben. Für die Visualisierung der ADS-B Daten wird OpenLayers3 als Kartenkomponente eingesetzt. jQuery wird für die dynamische Manipulierung von HTML Seiten verwendet.

In diesem Unterkapitel werden diese drei wichtigen Bibliotheken erläutert. Es wird zudem beschrieben wo und in welchem Umfang die Bibliotheken eingesetzt werden.

5.1.1 AngularJS

AngularJS[14] ist ein Open Source Framework für JavaScript und wurde ursprünglich von Google entwickelt (und weiterentwickelt). Es hilft dem Nutzer clientseitige Webapplikationen zu schreiben. Das Framework ist so konzipiert, dass der Visualisierungscode vom Applikationscode getrennt werden kann. Die darzustellenden Daten werden in Modellen gehalten und werden in der Darstellungsebene, der View, schliesslich dargestellt. Diese Trennung des Applikationscodes wurde von Reenskaut erstmals 1979 als Model-View-Controller (MVC) bezeichnet (Krasner and Pope, 1988). Eine Weiterentwicklung dieses Paradigmas wurde als Model-View-ViewModel (MVVM) von Gossman im Jahre 2005 in einem Blogpost vorgestellt (Gossman, 2005). Der Unterschied vom MVC zum MVVM liegt darin, dass im MVVM der Controller durch ein zusätzliches Model ersetzt wird, welches von der View direkt verändert werden kann und das Model über die Änderung notifiziert. In AngularJS können grundsätzlich beide Paradigmen eingesetzt werden, wobei es dem

14 AngularJS - https://angularjs.org/ [letzter Zugriff 15.03.2015]

Entwickler offen gelassen wird, für welche Variante er sich entscheidet oder ob gar eine Kombination der beiden Varianten eingesetzt wird. Einer der AngularJS Hauptentwickler äusserte sich zu dieser Thematik mit folgender eindeutigen Aussage:

*"Having said, **I'd rather see developers build kick-ass apps** that are well-designed and follow separation of concerns, than see them waste time arguing about MV* nonsense. And for this reason, **I hereby declare AngularJS to be MVW framework - Model-View-Whatever**. Where Whatever stands for 'whatever works for you'."* - (Minar, 2012)

Seither wird AngularJS als ein Model-View-Whatever (MVW) Framework bezeichnet. Bei der Entwicklung des Webclients für diese Arbeit wurden sowohl Eigenheiten von MVC wie auch von MVVM verwendet.

5.1.2 OpenLayers 3

OpenLayer3[15] ist eine Open Source Bibliothek, welche in JavaScript geschrieben ist und für das Darstellen von Karten auf Webseiten verwendet werden kann. Mit Hilfe dieser Bibliothek können Raster- und Featuredaten aus verschiedenen Quellen geladen und als Layer auf einer Kartenkomponente dargestellt werden. In der folgenden Tabelle werden die unterstützten Datenquellen aufgelistet:

Rasterdaten	Featuredaten
OpenstreetMap	GeoJSON
Bing	TopoJSON
MapBox	Keyhole Markup Language (KML)
MapQuest	Geography Markup Language (GML)
OGC – WMS	OGC – WFS

Tabelle 22: OpenLayers 3 unterstützte Datenformate

OpenLayers3 wird im entwickelten Webclient für die Visualisierung der Messwerte verwendet. Da OpenLayers 3 GeoJSON out-of-the-box unterstützt, wird angestrebt, diese Funktionalität direkt zu nutzen um die empfangenen Features zu visualisieren.

5.1.3 jQuery

Die Open Source JavaScript Bibliothek jQuery[16] kann bei der Manipulierung von HTML Seiten hilfreich sein. Im Speziellen kann das Domain Object Model (DOM) von einer HTML-Seite manipuliert werden. So kann mit jQuery die Repräsentation einer HTML-Seite als ein Baum aus JavaScript-Objekten verändert werden. Dadurch ist es möglich, dass nach dem Laden einer Seite der Inhalt geändert, nachgeladen oder aktualisiert wird. jQuery ist

15 OpenLayers 3 - http://openlayers.org/ [letzter Zugriff 15.03.2015]
16 jQuery - https://jquery.com/ [letzter Zugriff 18.06.2015]

weitverbreitet: Gemäss einer Auswertung von w3techs[17] wird jQuery auf 64.6% Prozent aller Webseiten eingesetzt; falls auf der Webseite JavaScript verwendet wird sogar zu 95.2%. Im entwickelten Webclient wird jQuery für das Zusammenfügen von Objekten direkt verwendet. Indirekt wird es zudem durch AngularJS für das dynamische Laden von Seiteninhalten genutzt.

5.2 Allgemeiner Aufbau

Der Grundaufbau der Clientanwendung ist zu einem grossen Teil durch das verwendete Rahmenwerk AngularJS (vgl. Unterkapitel 5.1.1) gegeben. Grundsätzlich ist die Anwendung in Dienste (Services), Kontroller (Controller), Datenmodelle (Models) und Darstellungssichten (Views) unterteilt. Die Aufgabe der Dienste ist es, Funktionen zur Beschaffung von Daten zur Verfügung zu stellen. Die Controller verwenden die Dienste um Daten zu laden. Die Aufgabe der Controller ist es, die Daten für die Darstellungssichten aufzubereiten und in einem Model abzubilden. Auf den Darstellungssichten werden schliesslich die Modeldaten visualisiert.

In Abbildung 17 wird die Client-Webanwendung im Gesamtkontext des Systems aufgezeigt. In der Client-Webanwendung wurden zwei Implementierungen der Dienste entwickelt, den SOS Service und den Websocket Service. Die Services beziehen respektive empfangen Daten von den entsprechenden Server Diensten. Der SOS Service bezieht die Daten vom SOS-Server per Anfragen, der Websocket Service empfängt die Daten direkt aus dem Websocket, welches vom Integrationssystem zur Verfügung gestellt wird. Eine weitere Aufgabe der Dienste ist es, diese Daten in einem einheitlichen Format an die Map Controller zu übergeben. Dadurch, dass die Dienste die empfangenen Daten in ein einheitliches Format umwandeln, musste lediglich eine Map-Controller-Implementierung erstellt werden, da diese für beide Varianten verwendet werden kann. Wie in der Abbildung 17 dargestellt, befüllen Controller die Modelle der OpenLayers Komponente, welche wiederum die OpenLayers-Darstellungskomponente über Änderungen benachrichtigt. Da nur der Service bei den beiden Varianten unterschiedlich implementiert wurde, ist der direkte Vergleich der beiden untersuchten Varianten besser möglich. Würden unterschiedliche Implementierungen von den Controllern oder Views verwendet, könnte dies das Laufzeitverhalten der beiden Varianten zusätzlich beeinflussen.

17 W3techs - http://w3techs.com/technologies/overview/javascript_library/all [letzter Zugriff 18.06.2015]

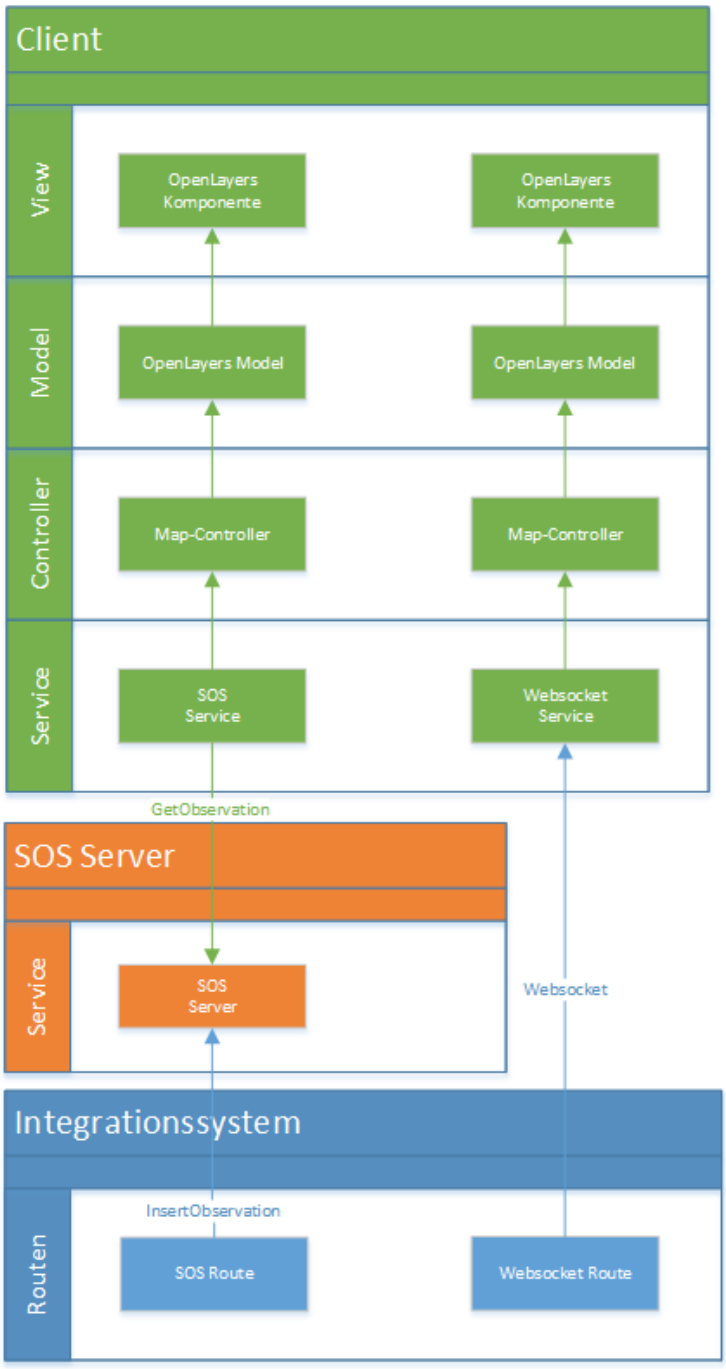

Abbildung 17: Client Übersicht

In der Websocket-Variante werden somit die Daten vom Empfang auf dem Integrationssystem bis zur Darstellung durchgereicht, ohne dass an einer Stelle nach neuen Daten nachgefragt werden muss. In der SOS-Variante werden auf der Clientseite mittels eines *SOS:GetObservation*-Service Aufrufs in regelmässigen Abständen neue Daten abgefragt.

5.3 Client Service

Die Aufgabe des Client Service ist es, eine Anbindung zum Integrationssystem über Websocket oder HTTP-Requests zum SOS-Server herzustellen und Daten über die enstprechende Variante zu empfangen. Empfangene Daten sollen innerhalb dieses Dienstes in ein einheitliches Format umgewandelt werden. Für die beiden Varianten wurde jeweils eine Service-Implementierung erstellt, welche auf dem gleichen Basis-Service aufbaut und dieselben Funktionen anbietet. Es wird somit das Kompostium Designpattern verwendet (Gamma et al., 1994). Dies hat den grossen Vorteil, dass die beiden Services analog verwendet werden können. Ein Controller oder ein anderer Dienst kann auf ein solches Objekt dieselben Funktionen aufrufen, ohne dass dieser Implementierungsdetails kennen muss. Innerhalb des Services werden je nach Implementierung unterschiedliche Funktionen aufgerufen. Im folgenden Unterkapitel wird der Basis-Service genauer beschrieben. Zudem wird erörtert, welche Funktionen der Basis-Service anbietet, damit der Map Controller nicht nach Daten *pollen* muss, sondern dass empfangene Daten direkt an den Map Controller übergeben werden.

Die darauf folgenden Unterkapitel 5.3.4 und 5.3.5 beschreiben, wie der Websocket Service und der SOS Service aufgebaut sind und wie diese die Funktionalität des Basis-Services nutzen.

5.3.1 Client Basis-Service

Der Basis-Service (*BaseService*) implementiert verschiedene Funktionen, welche wiederum von abgeleiteten Klassen genutzt werden können. Die Klasse umfasst drei verschiedene Properties, welche den Status des Services beschreiben:

Attribut	Beschreibung
enabled	Ein boolesches Wert, welcher anzeigt ob der Service aktiviert ist oder nicht.
connectionStatus	Eine Zeichenkette, welche den aktuellen Verbindungsstatus aufführt.
messageCount	Die Anzahl empfangener Meldungen

Tabelle 23: Client Basis-Service Attribute

Für diese drei Attribute bietet der Basis-Service *getter*- und *setter*-Funktionen an. Somit kann mit einem Aufruf auf den Service abgefragt werden, wie der aktuelle Stand des Services ist oder wie viele Meldungen bereits empfangen wurden. Da eine Änderung eines dieser Attribute möglichst rasch an interessierte Beobachter mitgeteilt werden soll, wurde das Observer Pattern (Gamma et al., 1994) innerhalb des Basis-Services implementiert. So können die interessierten Beobachter umgehend benachrichtigt werden, sobald sich ein Property ändert. Der Ablauf für dieses Pattern erfolgt nach demselben Prinzip wie das *push*-Prinzip (vgl. Unterkapitel 2.2.2 Push-Prinzip). Ein interessierter Beobachter kann sich mittels

einer *subscribe*-Funktion beim Basis-Service registrieren. Der Beobachter übergibt beim Registrieren eine Funktion, welche aufgerufen werden soll (Callback-Funktion). Ändert sich das Attribut, für welches sich der Beobachter registriert hat, wird die Callback-Funktion von allen registrierten Beobachtern dieses Attributes aufgerufen. Im nachfolgenden Sequenzdiagramm wird das Observer-Pattern anhand des Basis-Services aufgezeigt:

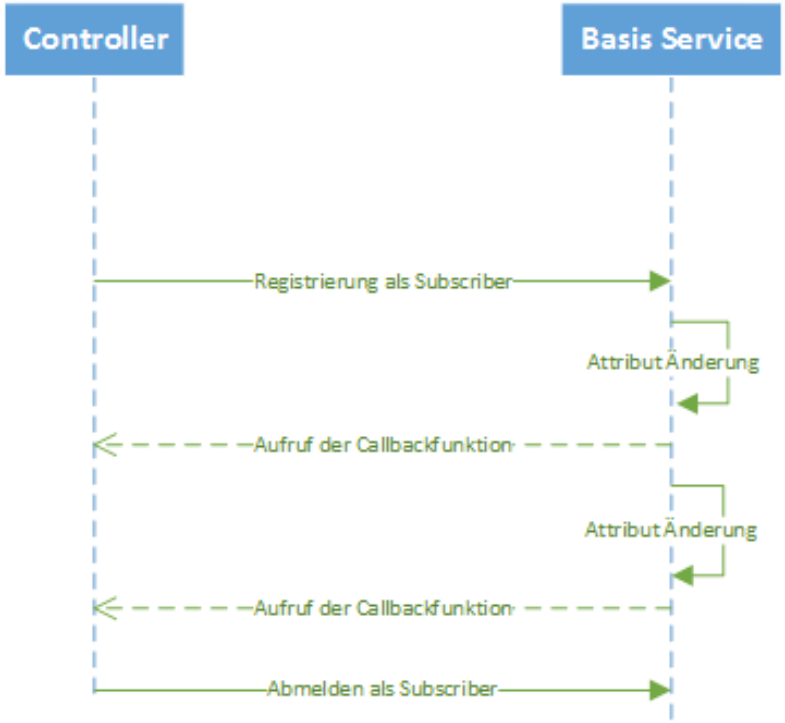

Abbildung 18: Client Service - Controller Kommunikation

Zusätzlich zu den bereits beschriebenen Attributen (*enabled*, *connectionStatus* und *messageCount*) können sich Beobachter auch mittels einer Callback-Funktion auf empfangene Meldungen (messages) registrieren. Somit wird der Beobachter immer benachrichtigt, sobald eine neue Meldung empfangen wurde. Die neue Meldung wird der Callback-Funktion übergeben, so dass diese vom Beobachter umgehend verarbeitet werden kann. Die empfangenen Meldungen werden ungleich der anderen Properties nicht im Basis-Service gespeichert, da die Datenmenge im Service rasant ansteigen würde, falls die Daten zwischengespeichert werden. Somit werden empfangene Daten im Service umgehend an interessierte Beobachter weitergereicht.

5.3.2 Implementierung des Basis-Services

Der Basis-Service enthält drei wichtige Funktionen, welche von abgeleiteten Klassen oder von anderen Objekten aufgerufen werden können. Diese drei Funktionen werden für die Umsetzung des Observer Patterns benötigt:

Funktion	Signatur	Funktion
subscribe	`function( propertyName, callback )`	Über die *subscribe*-Funktion kann sich ein Beobachter beim Basis-Service registrieren. Hierfür muss der Name des Properties, über welches man Informationen erhalten möchte und die Callback-Funktion, die ausgeführt werden soll, wenn das Property mit dem entsprechenden *property*-Namen sich geändert hat, angegeben werden
fire	`function( propertyName, newValue )`	Mit Hilfe der *fire*-Funktion werden Änderungen eines Properties publiziert. Durch den Aufruf der *fire*-Funktion werden alle Callback-Funktionen aufgerufen, welche sich für das Property mit dem entsprechenden *propertyName* interessieren. Die Callback-Funktion wird mit dem *newValue* als Parameter aufgerufen.
unsubscribe	`function( propertyName, callback )`	Mit der *Unsubscribe*-Funktion wird ein Beobachter eines Properties als Listener deregistriert. Es wird somit die übergebene Callback-Funktion vom entsprechenden Property mit dem übergebenen *propertyName* deregistriert und bei einem *fire*-Aufruf für das entsprechende Property nicht mehr aufgerufen.

Tabelle 24: Basis-Service implementierte Funktionen

Des Weiteren umfasst der Basis-Service zusätzliche Funktionen, die den Umgang mit dem Service vereinfachen. So existieren für die bereits erwähnten Properties (enabled, connectionStatus, messageCount, messages) zudem Helfer-Methoden für die in Tabelle 24 erwähnten Funktionen. So gibt es für jedes dieser Properties Funktionen für das Registrieren, Deregistrieren und Auslösen von *callback*-Funktionen. Die Bezeichnung ist jeweils folgendermassen aufgebaut:

```
function subscribe<PropertyName>(callback)
function unsubscribe<PropertyName>(callback)
function fire<PropertyName>(newValue)
```

Beispielsweise bietet der Basis-Service die Funktionen *subscribeMessageCount*, *unsubscribeMessageCount* und *fireMessageCount* an, welche intern die entsprechenden Funktionen aus Tabelle 24 aufrufen.

Der Ablauf des Observer Patterns im Basis-Service erfolgt nach folgendem Prinzip:

Schritt	Beschreibung	Aufruf im Code
1	Ein Objekt (z.B. ein Controller) möchte über alle Änderungen des Servicezustandes informiert werden.	

Schritt	Beschreibung	Aufruf im Code

2	Die Komponente registriert sich auf dem Basis-Service. Hierfür wird die *subscribe*-Funktion für das gewünschte Attribut aufgerufen. Als Übergabeparameter wird die *Callback*-Funktion mitgegeben, welche beim Ändern des Attributs aufgerufen werden soll.	`service.subscribeEnableState(`
		`  function(enabled) {`
		`  console.log("The enable"`
		`  + "state of the service"`
		`  + "is currently: "`
		`  + enabled);`
		`  });`

3	Der Service wird durch ein Drittobjekt deaktiviert. Innerhalb des Services wird die *setEnableState*-Funktion mit dem Attribut *false* aufgerufen, welches nach dem Setzen des Properties die *fireEnableState*-Funktion aufruft. Diese ruft wiederum die *fire*-Funktion auf.	`service.setEnableState =`
		`  function(enabled){`
		`  service.enabled = enabled;`
		`  service.fireEnableState(`
		`  service.enabled);`
		`  };`

4	Innerhalb der *fire*-Funktion werden alle Callbacks ausgeführt, welche auf das entsprechende Property registriert sind.	`this.callbackHandlers`
		`  [propertyName]`
		`  .forEach(function (item) {`
		`  item(newValue);`
		`  });`

5	Die in Schritt zwei übergebene Funktion wird ausgeführt. In der Konsole wird folgender Text ausgegeben	`The enable state of the` `service is currently: false`

Tabelle 25: Ablauf Observer Pattern im Client Service

Eine Service Implementierung, welche sich vom Basis-Service ableitet, kann somit auf die verschiedenen Helfer-Funktionen zugreifen. Da JavaScript eine dynamische Programmiersprache ist und keine Typensicherheit kennt, kann keine fixe Schnittstelle definiert respektive vorgegeben werden, was eine Basisklasse zur Erfüllung eines voll funktionsfähigen Services darstellt. Damit der Service voll funktionsfähig ist, sollten jedoch die folgenden Funktionen von einer abgeleiteten Klasse implementiert werden:

Funktion	Beschreibung
connect()	Die Connect-Funktion wird aufgerufen, wenn der Service mit dem Beschaffen der Daten beginnen soll
isConnected()	Gibt einen booleschen Wert zurück, welcher angibt ob der Service aktuell verbunden ist oder nicht.
disconnect()	Wird die Disconnect-Funktion aufgerufen, beendet der Service das Beschaffen der Daten.

Tabelle 26: Client Basis-Service Funktionen Schnittstelle

5.3.3 Konfigurationsmöglichkeiten

Die Konfiguration der Client Services und der Controller erfolgt über eine JavaScript Konfigurationsdatei (*config.js*). Die Konfiguration erfolgt durch die Definition von Konstanten.

```
angular.module('config', [])
              .constant('ENV', {
                     environment : 'development',
                     version : 1,
              })
```

Im obigen Beispiel wird die Konstante ENV mit einem JavaScript Objekt definiert, welches die Attribute *environment* mit dem Wert 'development' und *version* mit dem Wert '1' beinhaltet.

Die Konstanten (in AngularJS als *constant* bezeichnet) können anschliessend durch Angabe des Konstanten-Namens in die Services oder Controller injektiert werden.

```
angular.module('angularol3jsuiApp').controller('TestController', function(ENV) {
      var version = ENV.version;
      ...
});
```

Bei der Erzeugung dieses Testcontrollers wird die zuvor definierte Konstante *ENV* übergeben. Auf die einzelnen Attribute der ENV-Konstante kann nun nach Belieben zugegriffen werden.

Für die beiden Varianten (SOS und Websocket) existiert je eine Konstante:

- *sosConfig*
- *websocketConfig*

Beide Konstanten umfassen jeweils variantenspezifische Konfigurationsmöglichkeiten. Die Ausführung der Konfigurationsmöglichkeiten werden in den Unterkapiteln 5.3.4 SOS Client Service und 5.3.5 Websocket Client Service beschrieben.

5.3.4 SOS Client Service

Der SOS-Client Service erweitert den Basis-Service um die Funktionalität, Daten von einem SOS-Server zu laden, diese umzuwandeln und den Observern mitzuteilen.

Aufgaben

In regelmässigen Abständen sollen *SOS:GetObservation*-XML-Anfragen vom SOS-Client-Service an den SOS-Server gesendet werden (vgl. Unterkapitel 3.2.1 SOS-Variante). Die Antwort des SOS enthält alle Observationen, welche im angefragten Zeitbereich aufgetreten sind. Die einzelnen Observationen sollen nun so umgewandelt werden, dass diese vom Client-Controller verarbeitet werden können. Durch einen *fireMessages*-Aufruf sollen anschliessend

die neuen umgewandelten Datensätze allen Observern, die an den SOS-Messages interessiert sind (wie z.B. der Map Controller), bekannt gemacht werden.

Aufbau

Der SOS-Client Service ist nach dem Prinzip eines *pull*-basierten *polling*-Services aufgebaut. Der SOS-Client Service implementiert die vom Basis-Service geforderten Funktionen (vgl. Unterkapitel 5.3.2 Implementierung des Basis-Services).

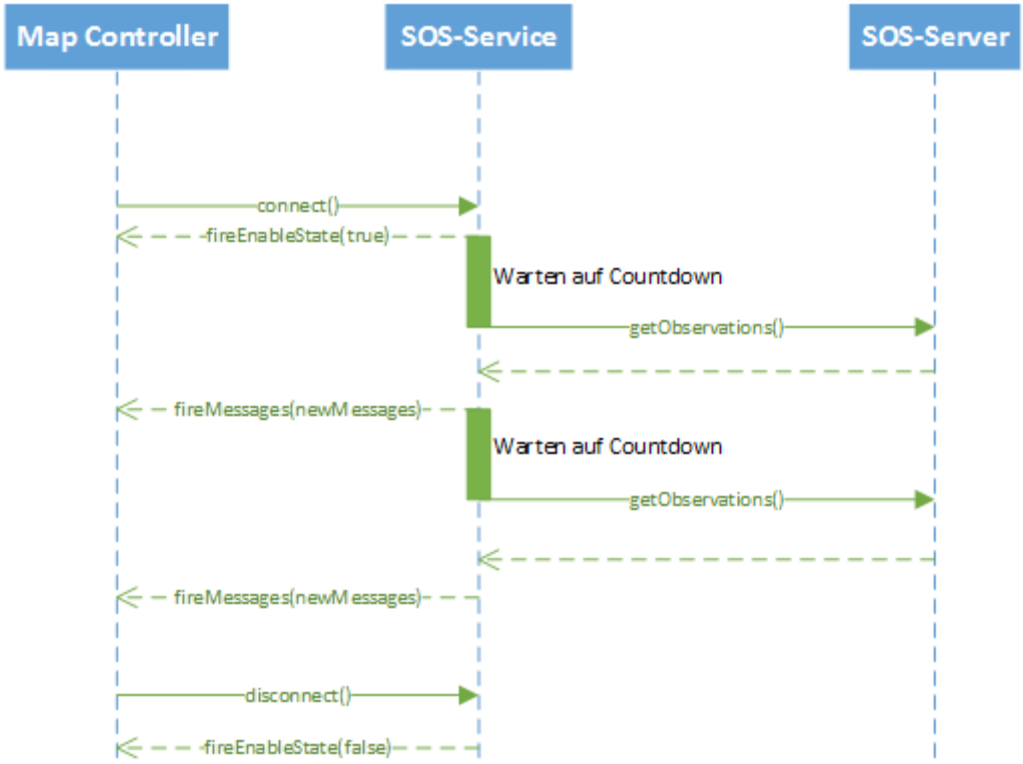

Abbildung 19: Sequenzdiagramm SOS-Client Service

Mit dem Aufruf der *connect*-Funktion wird im SOS-Service ein Countdown gestartet, welcher nach dessen Ablauf einen *getObservation*-Aufruf an den SOS-Server sendet. Falls der Serveraufruf korrekt ausgeführt wurde und der Client eine korrekte Antwort vom Server erhalten hat, werden die erhaltenen Daten vom Service verarbeitet und an die Observer verbreitet. Anschliessend wird der Countdown erneut gestartet. Dieser Vorgang wird solange wiederholt, bis die *disconnect*-Funktion aufgerufen wurde oder ein Fehler aufgetreten ist.

Implementierung

Der SOS Client-Service implementiert die Funktionen *Connect()*, *Disconnect()* und *isConnected()* auf die Art und Weise, dass die Daten nach einem bestimmten Intervall vom Server geladen werden können. Zusätzlich wird nachfolgend beschrieben, wie die Daten der *getObservation*-Anfrage in Features umgewandelt werden, so dass diese vom Map Controller in das OpenLayers3 Model geschrieben werden können.

Connect-Funktion

Die *Connect()*-Funktion ist die aufwendigste Implementierung der drei Funktions-Implementierungen des Basis-Services. Beim Aufruf wird folgender Ablauf durchgeführt:

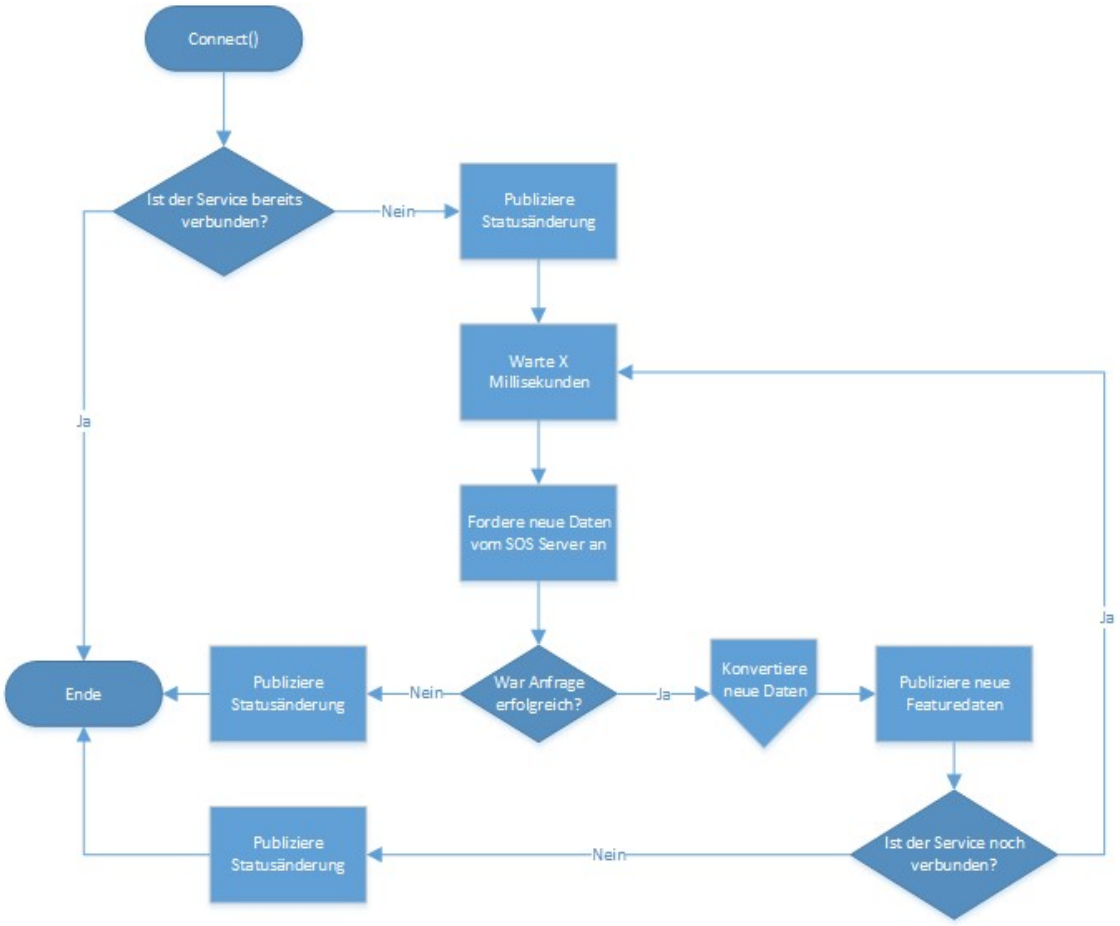

Abbildung 20: Ablaufdiagramm Client SOS Service

Die Abbildung 20 beschreibt den Ablauf, der erfolgt, nachdem die *connect*-Funktion beim SOS-Client Service aufgerufen wurde. Nach dem Aufruf der connect-Funktion innerhalb des Services wird mittels der *isConnected*-Funktion überprüft, ob der Service bereits verbunden ist. Ist dies nicht der Fall, wird der Status mittels der *setEnableState*-Funktion geändert, wobei die Observer des *EnableState*-Properties über den Statuswechsel informiert werden. Anschliessend wird die *timeout*-Funktion aufgerufen, welche das verzögerte Laden von Daten initiiert. Nach Ablauf des *timeouts* wird die *loadRemoteData*-Funktion aufgerufen. Die *loadRemoteData*-Funktion ist jene Funktion, welche den Zeitbereich berechnet, von welchem die Daten vom SOS-Server geladen werden sollen. Ist die Start- und Endzeit berechnet, wird ein Servicecall mit dem entsprechenden Zeitbereich gestartet. Das Startdatum wird gemäss des Konzepts der SOS-Variante (vgl. Unterkapitel 3.2.1 SOS-Variante) bei der ersten Ausführung auf das aktuelle Datum minus die Intervall-Dauer gesetzt. Wurden bereits Observationen empfangen, wird das Startdatum auf den Zeitpunkt des letzten Phenomenon-

Zeitpunkts gesetzt. Die Berechnung dieses Zeitpunktes erfolgt in der *convertToFeatures*-Funktion, die später im Abschnitt Umwandlung in Featuredaten beschrieben wird. Das Enddatum der zu ladenden Daten wird auf den aktuellen Zeitpunkt gesetzt. Ist die Datumsberechnung abgeschlossen, werden damit durch den Aufruf der *loadNewEntries*-Funktion die Daten mit dem berechneten Datumsbereich vom Service geladen.

```
function loadNewEntries(dateFrom, dateTo) {
      var request;
      ...
      request = $http({
        method: 'post',
        url: sosConfig.poxURL,
        data: '<?xml version="1.0" encoding="UTF-8"?>' +
        '<sos:GetObservation service="SOS" version="2.0.0"' +
        ...
      + '<sos:temporalFilter><fes:During>'
      + '<fes:ValueReference>phenomenonTime</fes:ValueReference>'
      + '<gml:TimePeriod gml:id="t1">'
      + '     <gml:beginPosition>' + dateFrom.toISOString() + '</gml:beginPosition>'
      + '     <gml:endPosition>' + dateTo.toISOString() + '</gml:endPosition>'
      + '</gml:TimePeriod>'
        ...
      + '<sos:responseFormat>application/json</sos:responseFormat>'
        '</sos:GetObservation>'
      });
      ...
      return request.then(handleSuccess, handleError);
```

Innerhalb der *loadNewEntries*-Funktion wird der HTTP-Request an den SOS-Server initiiert. Der übergebene Datumsbereich (*dateFrom* bis *dateTo*) wird innerhalb der SOS-Anfrage als Filter auf das *phenomenonTime*-Attribut mitgegeben. Zudem wird als Antwortformat in der Meldung *application/json* angegeben, so dass das Ergebnis des Service Aufrufs im JSON-Format empfangen wird (vgl. Unterkapitel 3.2.1 SOS-Variante Abschnitt SOS-Anfragen). Ist die Anfrage an den SOS-Server erfolgreich, werden die Daten in der *handleSuccess*-Funktion in Geo-Feature-Objekte, welche von Openlayers3 eingelesen und dargestellt werden können, umgewandelt. Die umgewandelten Objekte werden mittels der *fireMessages*-Funktion an die Observer publiziert.

Im Fehlerfall wird die *handleError*-Funktion aufgerufen, in welcher umgehend die *disconnect*-Funktion mit einer geeigneten beschreibenden Meldung aufgerufen wird.

Disconnect-Funktion

Die *disconnect*-Funktion prüft mittels der *isConnected*-Funktion, ob der Service aktuell verbunden ist. Ist der Service verbunden, wird das aktuelle Time-out beendet und gelöscht und der Zähler der empfangenen Meldungen zurückgesetzt.

Anschliessend wird der Status des Services mit der *setEnableState*-Funktion auf *false* gesetzt, wobei alle Observer über den neuen Status informiert werden. Zum Schluss wird diese, falls eine Meldung der *disconnect*-Funktion übergeben wurde, den Observern als Statusmeldung

mitgeteilt. Wurde keine Meldung der Funktion übergeben, wird den Observern lediglich die Statusmeldung "Disconnected" mitgeteilt.

```
service.disconnect = function(message) {
	if (service.isConnected()) {
		$timeout.cancel(timeout);
		service.resetMessageCount();
		timeout = null;
	}
	service.setEnableState(false);
	var connectionStatusMessage = message;
	if (!connectionStatusMessage) {
		connectionStatusMessage = 'Disconnected';
	}
	service.setStatus(connectionStatusMessage);
};
```

isConnected-Funktion

Die SOS-Implementierung der *isConnected*-Funktion prüft lediglich, ob die Variable *timeout* im Service gesetzt ist. Ist dies der Fall, wird von der Funktion *true* zurückgegeben (ansonsten *false)*.

```
service.isConnected = function() {
	return (angular.isObject(timeout));
};
```

Wie bereits bei der Beschreibung der *connect*- und *disconnect*-Funktion angedeutet, wird die *timeout* Variable in der *connect*-Funktion erzeugt und in der *disconnect*-Funktion gelöscht. Somit ist diese Variable ein gutes Indiz dafür ob der Service aktuell verbunden ist oder nicht.

Umwandlung in Featuredaten

An die Umwandlungsfunktion wird das Ergebnis der *handleSuccess*-Funktion übergeben. Dieses Ergebnis beinhaltet eine Liste mit Observation-Meldungen, welche im Format JSON empfangen und von AngularJS automatisch in JavaScript Objekte umgewandelt wurden. Diese sollen nun in GeoJSON-Objekte umgewandelt werden welche von OpenLayers3 verarbeitet werden können.

Jede dieser Observation-Meldungen enthält einen einzigen Messwert eines Feature-Of-Interests. Feature-Of-Interests können mehrfach vorkommen und Messwerte aus unterschiedlichen Zeitbereichen beinhalten. Um unterscheiden zu können was für ein Messwert die Observation beinhaltet, enthält die Observation-Meldung zudem ein *observableProperty*-Attribut, welches den Typ des Messwertes bezeichnet (vgl. Unterkapitel 3.2.1 SOS-Variante Abschnitt SOS-Anfragen). Der Ablauf für die Umwandlung von den empfangenen SOS-Observationen in Feature Daten erfolgt nach folgendem Ablauf:

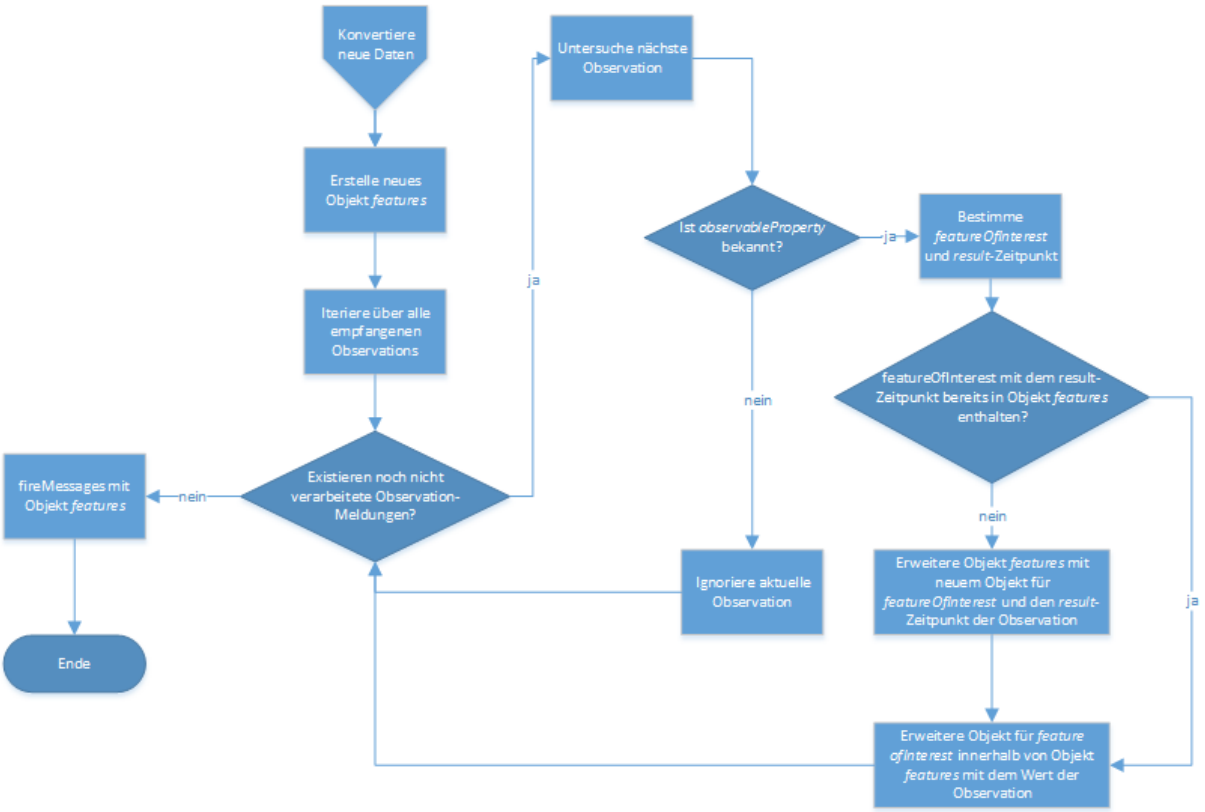

Abbildung 21: Client SOS-Service Verarbeitung neuer Daten

In einem ersten Schritt wird ein neues JavaScript Objekt erzeugt in welches die Messwerte gruppiert nach Feature-Of-Interest hinzugefügt werden. Anschliessend wird über die erhaltenen Observation-Meldungen iteriert. Für jede Observation wird das *observableProperty* bestimmt. Ist dieses konfiguriert (vgl. folgendes Unterkapitel), wird das Feature-Of-Interest und der r*esult*-Zeitpunkt von der Observation bestimmt. Existiert das Feature-Of-Interest für den entsprechenden *result*-Zeitpunkt noch nicht im *features*-Objekt, wird ein neues Feature für das ausgelesene Feature-Of-Interest mit dem *result*-Zeitpunkt der Observation erzeugt und dem *features*-Objekt hinzugefügt. Hierbei werden das *type*-, *properties*- und *id*-Attribut dem neuen *feature*-Objekt zugewiesen. Das *type*-Attribut wird auf den Wert "Feature" gesetzt, das *properties*-Attribut wird auf das JavaScript-Objekt gesetzt und das *id*-Attribut auf das Feature-Of-Interest. Zudem wird auf dem *properties*-Attribut das Attribut *messageGenerated* auf den Wert des *result*-Zeitpunktes und das *messageReceived* Attribut auf den aktuellen Zeitpunkt gesetzt. Dadurch wird der Grundaufbau eines GeoJSON-Objektes nachgebildet. Das *Id*-Attribut entspricht nicht dem GeoJSON-Standard, es wird jedoch in der OpenLayers3-Implementierung zur eindeutigen Identifizierung eines Objektes benötigt.

Anschliessend wird das neu erzeugte Feature oder das sich bereits im *features*-Objekt befindliche Feature Objekt mit dem ausgelesenen Messwert in der *updateFeature*-Funktion ergänzt.

```
function updateFeature(feature, observation) {
      var result = observation.result;
      var propertyType = sosConfig.properties[observation.observableProperty].type;
      var propertyName = sosConfig.properties[observation.observableProperty].name;

      if (propertyType === 'number') {
            feature.properties[propertyName] = result.value;
      } else if (propertyType === 'string') {
            feature.properties[propertyName] = result;
      } else if (propertyType === 'geojson') {
            feature[propertyName] = result;
      }
}
```

Bei der Zuweisung des Messwertes muss bekannt sein, was für einem Messwert-Typ der aktuelle Messwert entspricht. Mittels der SOS-Service Konfiguration (vgl. folgendes Kapitel) können die unterstützten *observableProperty* und deren Typen dem SOS-Service bekannt gemacht werden.

Bisher unterstützt die SOS-Service Implementierung folgende drei Typen von Observation-Messwerten:

- Number (Für Nummernwerte)
- String (Für Zeichenketten)
- GeoJSON (Für GeoJSON-Resultate)

Konfigurationsmöglichkeiten

Die Konfiguration des SOS-Client Services erfolgt über die *sosConfig*-Konstante der Konfigurationsdatei. Folgende Attribute werden von der Service Implementierung berücksichtigt:

Property	Typ	Erläuterung
sosConfig.poxURL	Zeichenkette	Definiert die URL zum SOS-Server. Es muss beachtet werden, dass die URL inklusive dem Pfad zum XML-Service angegeben wird.
sosConfig.updateInterval	Zahlenwert	Gibt die Dauer, wie lange beim *polling* zwischen verschiedenen Requests gewartet werden soll, in Millisekunden an.
sosConfig.procedure	Zeichenkette	Definiert die SOS-Procedure von welcher Observationen berücksichtig werden sollen.
sosConfig.offering	Zeichenkette	Bezeichnet das SOS-Offering von welchem Observationen berücksichtig werden sollen.
sosConfig.properties	Objekt	Die Properties Konfiguration beschreibt die Properties, welche bei der Umwandlung in Feature-Objekte berücksichtigt werden sollen.

Property	Typ	Erläuterung
		Für die Darstellung der ADS-B Daten wird die Konfiguration gemäss den Sensor Eigenschaften (vgl. Kapitel 3.2.1 SOS-Variante Absatz SOS-Sensor) konfiguriert. Es werden die Properties konfiguriert, welche als Zieldaten (vgl. Kapitel 3.1 Zieldaten) definiert wurden.

Tabelle 27: Konfigurationsmöglichkeiten Client SOS-Service

Eine mögliche Konfiguration welche für den Empfang von ADS-B Meldungen gemäss den definierten Zieldaten konfiguriert ist, ist als Anhang beigefügt (vgl. Anhang A.4 Client Konfigurationsdatei).

5.3.5 Websocket Client Service

Der Websocket-Client Service erweitert, wie auch der SOS-Client Service, den Basis-Service. Die Aufgabe vom Websocket-Client Service ist es, sich mit dem Websocket zu verbinden, die GeoJSON-Daten zu empfangen und diese nach erfolgreichem Empfang den Observern mitzuteilen.

Aufgaben

Der Websocket-Client Service verbindet sich mit dem vom Integrationssystem erstellten Websocket (vgl. Unterkapitel 3.2.2 Websocket-Variante). Nachdem der Service sich mit dem Websocket verbunden hat, werden die im Integrationssystem erzeugten GeoJSON-Nachrichten zum Client übertragen. Die einzelnen GeoJSON-Nachrichten sollen im Service angepasst werden, so dass diese vom Client-Controller verarbeitet und visualisiert werden können. Nach dem erfolgreichen Empfang einer neuen GeoJSON-Nachricht werden mittels eines *fireMessages*-Aufrufs die neuen Daten den Observern bekannt gemacht werden.

Aufbau

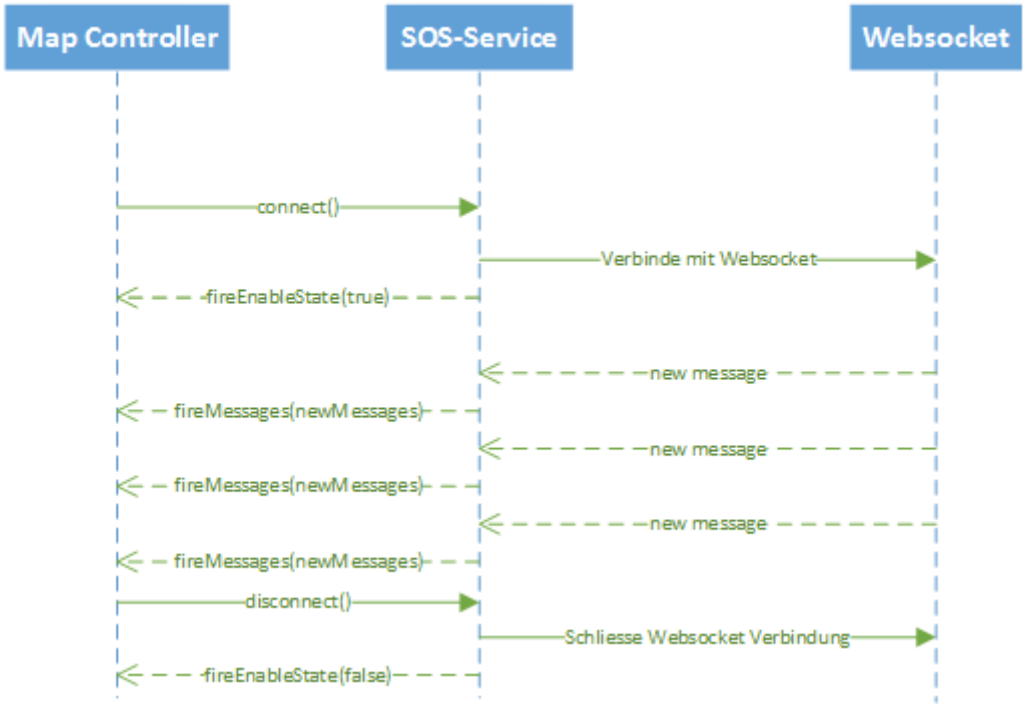

Abbildung 22: Sequenzdiagramm Websocket-Client Service

Der Websocket-Client Service ist nach dem Prinzip eines *pull*-Services konzipiert und implementiert die vom Basis-Service geforderten *connect*-, *disconnect*- und *isConnected*-Funktionen (vgl. Unterkapitel 5.3.2 Implementierung des Basis-Services). Mit dem Aufruf der *connect*-Funktion verbindet sich der Service mit dem Websocket. Konnte die Verbindung mit dem Websocket erfolgreich hergestellt werden, werden die empfangenen GeoJSON-Objekte ergänzt. Mittels der *fireMessages*-Funktion wird der Map Controller umgehend mit der ergänzten Meldung benachrichtigt. Der Service wartet somit immer, bis neue Daten empfangen werden und verarbeitet diese unmittelbar nach dem Empfang.

Implementierung

Die Implementierung des Websocket Client-Services ist verglichen mit jener des SOS Client-Services weniger umfangreich. Dies ist darauf zurückzuführen, dass sich die empfangenen Daten bereits im GeoJSON Format befinden und nur noch ergänzt und nicht umgewandelt werden müssen. Die Implementierung der drei vom Basis-Service geforderten Funktionen wurde folgendermassen implementiert:

Connect-Funktion

Der Ablauf der *connect*-Funktion ist die aufwendigste Implementierung der drei Funktionen.

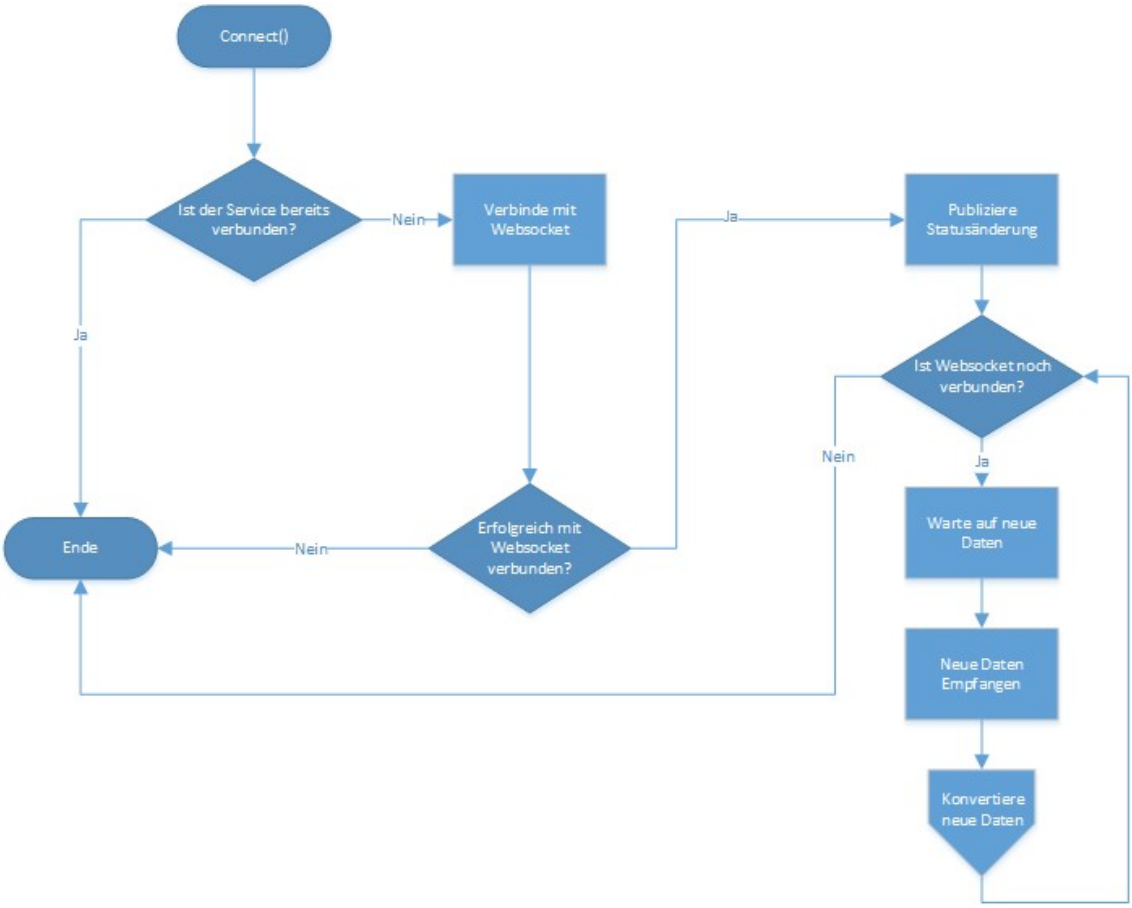

Abbildung 23: Ablaufdiagramm Client Websocket-Service

Beim Aufruf dieser Funktion wird folgender Ablauf durchgeführt:

Wie der Abbildung 23 zu entnehmen ist, wird nach dem Aufruf der *connect*-Funktion (wie bereits in der SOS-Variante) mittels der *isConnected*-Funktion überprüft, ob der Service bereits verbunden ist. Ist dies nicht der Fall, wird umgehend versucht das Websocket zu öffnen:

```
service.connect = function() {
      if (angular.isConnected()) {
            return;
      }
      var ws = new WebSocket(websocketConfig.url);
      ...
```

Dem Websocket werden nun drei Callback-Funktionen registriert, welche beim Eintreten des entsprechenden Events ausgeführt werden sollen.

```
      ...
```

```
	ws.onopen = function() {
		service.setStatus('Connected');
		service.setEnableState(true);
	};

	ws.onerror = function() {
		service.disconnect('Unable to open Connection');
	};

	ws.onmessage = function(message) {
		var geoFeatures = convertToFeatures(message);
		service.fireMessages(geoFeatures);
	};

	websocket = ws;
};
```

Falls das Websocket erfolgreich geöffnet werden konnte (*onopen*), wird die Statusänderung mitgeteilt. Im Fehlerfall (*onerror*) wird der Service durch den Aufruf der *disconnect*-Funktion geschlossen. Falls eine Meldung empfangen wurde, wird diese durch den Aufruf der *convertToFeatures*-Funktion in ein Feature geändert, welches vom Controller verarbeitet werden kann. Dieses *geoFeatures*-Objekt wird durch den Aufruf der *fireMessages*-Funktion den Observern mitgeteilt.

Zum Ende der *connect*-Funktion wird das erzeugte Websocket-Objekt (*ws*) der *websocket* -Variable zugewiesen. Dies stellt sicher, dass die *disconnect*- oder die *isConnected*-Funktion das richtige Websocket-Objekt ansprechen kann.

Disconnect-Funktion

Innerhalb der Websocket Service *disconnect*-Funktion wird geprüft, ob der Service aktuell verbunden ist. Ist dies der Fall, wird das Websocket durch den Aufruf der *close*-Funktion geschlossen. Der Zähler der empfangenen Meldungen wird im darauf folgenden Schritt zurückgesetzt.

Anschliessend werden alle Observer durch den auf der *setEnableState*-Funktion mit *false* gesetzten Parameter über das Trennen des Services informiert. Zum Schluss wird, falls eine Meldung der *discconnect*-Funktion übergeben wurde, diese Statusmeldung den Observern mitgeteilt. Wurde keine Meldung der Funktion übergeben, wird lediglich die Statusmeldung "Disconnected" den Observern mitgeteilt.

```
service.disconnect = function(message) {

	if (service.isConnected()) {
		websocket.close();
		service.resetMessageCount();
		websocket = null;
	}
	service.setEnableState(false);
	var connectionStatusMessage = message;
	if (!connectionStatusMessage) {
		connectionStatusMessage = 'Disconnected';
```

```
        }
        service.setStatus(connectionStatusMessage);
};
```

isConnected-Funktion

Die Websocket-Implementierung der *isConnected*-Funktion prüft ähnlich der SOS-Implementierung lediglich, ob eine Variable im Service gesetzt ist. In der Websocket-Variante wird dies anhand der *websocket*-Variable abgeklärt. Ist dies ein Objekt, wird *true* zurückgegeben (sonst *false)*.

```
service.isConnected = function() {
        return (angular.isObject(websocket));
};
```

Wie bei der *disconnect*- und *connect*-Funktion erwähnt, wird die *websocket*-Variable beim Verbinden des Services erstellt und beim Schliessen des Services gelöscht. Somit sagt diese Variable aus, ob das Websocket aktuell verbunden ist oder nicht.

Umwandlung in Featuredaten

Im Websocket-Client Service werden die empfangenen Daten nach folgendem Prinzip weiterverarbeitet, so dass diese als Feature publiziert dem Map Controller übergeben werden können:

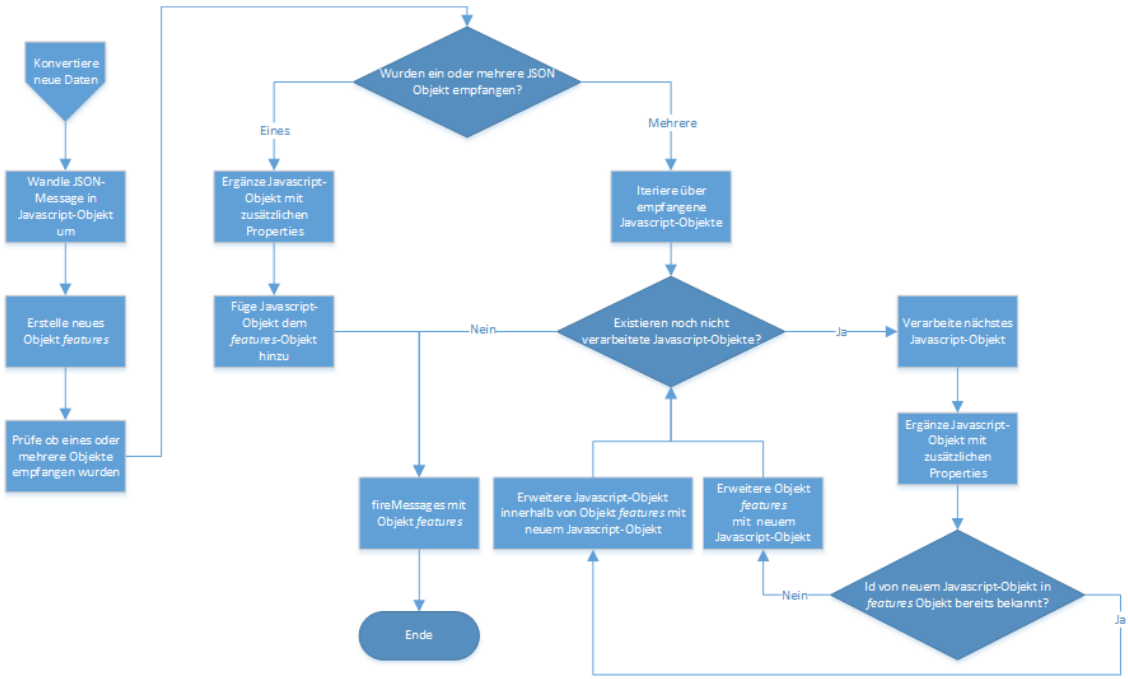

Abbildung 24: Client Websocket-Service Verarbeitung neuer Daten

In einem ersten Schritt wird die empfangene JSON-Zeichenkette in ein JSON-Objekt

umgewandelt. Wie auch bei der SOS Datenumwandlung wird in diesem Service ein neues JavaScript Objekt *features* erzeugt, in welches die Messwerte gruppiert nach einem Identifier hinzugefügt werden können. Anschliessend wird geprüft, ob das empfangene JavaScript-Objekt ein Array ist oder ob nur eine einzelne Meldung empfangen wurde.

Wurde nur ein Objekt empfangen, wird für das Objekt die *convertToFeature*-Funktion aufgerufen. Innerhalb der *convertToFeature*-Funktion wird dem Objekt das ID-Attribut auf den Wert des konfigurierten Properties gesetzt. Das *messageReceived* Feld wird auf das aktuelle Datum gesetzt. Das *messageGenerated* Feld wird auf den konfigurierten Property-Wert des empfangenen Features gesetzt.

```
function convertToFeature(jsonObject) {
  if (jsonObject.properties[websocketConfig.idProperty]) {
    var id = jsonObject.properties[websocketConfig.idProperty];
    if (id) {
      jsonObject.id = id;
      jsonObject.properties.messageReceived = new Date();
      if (jsonObject.properties[websocketConfig.messageGeneratedProperty]) {
        jsonObject.properties.messageGenerated = new Date(
            jsonObject.properties[websocketConfig.messageGeneratedProperty]);
      } else {
        jsonObject.properties.messageGenerated =
             jsonObject.properties.messageReceived;
      }
    }
    return jsonObject;
  }
}
```

Der Rückgabewert dieser Funktion wird dem *features*-Objekt hinzugefügt und wird dem Aufrufer der *convertToFeatures*-Funktion zurückgegeben.

Wurde ein Array übergeben, wird über die einzelnen Objekte iteriert. Für jedes Feature wird die *convertToFeature*-Funktion aufgerufen. Das Ergebnis dieser Funktion wird dem *features*-Objekt hinzugefügt. Falls bereits ein Objekt im *features*-Objekt mit der gleichen Id-exitstiert, werden die beiden Objekte mit Hilfe der jQuery-Funktion *$.extend* zusammengeführt. Die Funktion *$.extend* fügt den Inhalt von zwei Objekten zusammen. Durch die Angabe des ersten Parameters mit *true* wird das Objekt rekursiv zusammengefügt. Das heisst, die Objekte werden bis in alle Tiefen des Objektes miteinander zusammengefügt. Die Werte des zweiten Parameters werden zu jenen des Ersten hinzugefügt. Somit befinden sich nach dem Aufruf von *$.extend* im *features*-Objekt die aktuellsten zusammengefügten Daten. Die Möglichkeit mehrere Objekte in einer Websocket-Meldung zu senden wird im Integrationssystem derzeit nicht genutzt. Mit der aktuellen Integrationssystem-Implementierung werden empfangene SBS1-Meldungen umgehend als einzelnes GeoJSON Objekte über das Websocket versendet (vgl. 3.2.2 Websocket-Variante). Die Funktion mehrere Objekte mit einer Websocket-Meldung zu empfangen kann jedoch für andere Anwendungsfälle hilfreich sein.

Nachdem alle Objekte vom Array abgearbeitet wurden, wird das *features*-Objekt

zurückgegeben.

Konfigurationsmöglichkeiten

Die Konfigurationsmöglichkeiten des Websocket-Client Services sind weniger umfangreich als jene des SOS-Client Services und erfolgen über die *websocketConfig*-Konstante der Konfigurationsdatei. Die Konfigurationsmöglichkeiten sind jedoch darauf ausgelegt, dass der Websocket-Client Service für jegliche GeoJSON-Objekte benutzt werden kann. Neben der *URL* des Websockets können die Namen wichtiger GeoJSON-Properties konfiguriert werden.

Property	Typ	Erläuterung
websocketConfig.URL	Zeichenkette	Definiert die URL zum Websocket. Die URL muss inklusive Pfad zum Websocket angegeben werden.
websocketConfig.idProperty	Zeichenkette	Namen des GeoJSON-Properties, welches den Identifier Wert beinhaltet.
WebsocketConfig .messageGeneratedProperty	Zeichenkette	Namen des GeoJSON-Properties, welches den Zeit/Datumswert beinhaltet zu welchem die Nachricht erstellt wurde.

Tabelle 28: Konfigurationsmöglichkeiten Websocket Client-Service

Im Anhang ist eine mögliche Konfiguration für den Webclient angehängt (vgl. Anhang A.4 Client Konfigurationsdatei).

5.4 Map Controller

Der Map Controller ist das Bindeglied zwischen einem Client-Service und der Darstellungsebene. Die Hauptaufgabe des Map Controllers liegt im Verwalten der darzustellenden Featuredaten.

5.4.1 Ablauf

Für beide Varianten wird eine Map Controller-Instanz erstellt, welche den darunterliegenden Service observiert. Es wird für beide Varianten dieselbe Map Controller-Implementierung verwendet. Sobald ein Service neue Meldungen publiziert, werden diese verarbeitet. Bei der Verarbeitung werden die erhaltenen Features in ein OpenLayers3-Datenmodell gespeichert. Falls ein Feature bereits im Datenmodell zwischengespeichert ist, werden die Properties des zwischengespeicherten Features mit jenen des neu empfangenen Features aktualisiert. Es ist somit sichergestellt, dass die im OpenLayers3-Datenmodell befindlichen Features jeweils die aktuellsten Daten beinhalten. Vor dem Einfügen der Änderungen der Daten im OpenLayer3-Datenmodell werden die Features durch den konfigurierten StyleService (vgl. Unterkapitel 5.6) mit Style Attributen ergänzt.

In einem definierbaren Intervall wird überprüft, ob ein Feature seit der letzten Prüfung aktualisiert wurde. Ist dies nicht der Fall, wird das Feature aus den zwischengespeicherten Daten gelöscht. Dadurch sind nur Features zwischengespeichert von welchen unlängst Daten empfangen wurde. So wird sichergestellt, dass die Menge der darzustellenden Daten nicht konstant ansteigt und nur aktuelle Daten dargestellt werden.

5.4.2 Implementierung

Die Implementierung des Map Controllers entspricht einer *controller*-Implementierung von AngularJS. Für das Handling mit Openlayers wird eine spezielle Open Source AngularJS-Direktive[18] von David Rubert verwendet. Die *angular-openlayer-directive* bietet verschiedene Vereinfachungen an, um mit einer OpenLayers-Komponente in AngularJS zu interagieren. Einer Map Controller-Instanz werden beim Initialisieren verschiedene Variablen mitgegeben:

Variable	Funktion
dataService	Stellt eine Referenz auf den zu verwendenden Client Service dar. Dieser soll alle Funktionen des Basis-Services Implementieren (vgl. Unterkapitel 5.3.1)
implementationConfig	Stellt die variantenspezifische Konfiguration dar, die bereits in der Service Implementierung verwendet wird (entweder sosConfig oder websocketConfig). Diese enthält auch Konfigurationsmöglichkeiten, welche für den Map Controller relevant sind (vgl. Unterkapitel 5.4.3)
mapConfig	Diese Konfigurations-Konstante für die Kartendarstellung ist für beide Variante dieselbe, so dass die Karte für beide Varianten identisch aussieht (vgl. Unterkapitel 5.4.3).
olData	Das *olData*-Objekt enthält die OpenLayer3 spezifischen Attribute der *angular-openlayer-directive*. Unter anderem kann durch das *olData*-Objekt auf die OpenLayers-Komponente direkt zugegriffen werden
styleService	Der Service, welcher verwendet wird, um die empfangenen Features zu stylen (vgl. Unterkapitel 5.6)

Tabelle 29: Map Controller Initialisierungsvariablen

Die Anzahl Map Controller Instanzen und deren übergebene Parameter können über die Konfigurationsdatei definiert werden.

Beim Initialisieren einer neuer Map-Controller-Instanz werden unter anderem das Datenmodell und der Layer für die OpenLayers3-Komponente erzeugt:

```
var featureSource = new ol.source.Vector({
```

18 angular-openlayers-directive - https://github.com/tombatossals/angular-openlayers-directive [letzter Zugriff 26.04.2015]

```
      projection : implementationConfig.featureProjection
});

var featureLayer = new ol.layer.Vector({
      title : implementationConfig.featureLayerName,
      source : featureSource
});
```

Die Variable *featureSource* enthält somit im Map Controller das OpenLayers3-Datenmodell des Layers, welcher die empfangenen Feature Daten darstellen soll. Die *featureLayer*-Variable ist der Datenlayer, welcher in der OpenLayers3-Komponente visualisiert wird und stellt alle Features dar, welche in die *featureSource* geschrieben werden.

Zudem werden die *subscribeMessages*- und *subscribeEnableState*-Funktionen auf dem übergebenen Service mit jeweils einer Callback-Funktion aufgerufen, um über neue Meldungen und über Änderung des Service Status informiert zu werden.

```
    service.subscribeMessages(function (features) {
      $scope.applyRemoteData(features);
      ...
    });
```

Falls eine neue Meldung beim Controller eintrifft, wird umgehend die *applyRemoteData*-Funktion aufgerufen. Diese fügt die empfangenen Daten in das *featureSource*-Datenmodell ein.

```
    service.subscribeEnableState(function (enabled) {
      if (enabled) {
        ...
        $scope.startCleanupInterval();
      }
      else {
        $scope.stopCleanupInterval();
      }
    });
```

Im Falle einer Statusänderung wird geprüft ob der Service neu verbunden oder getrennt wurde. Bei einem Verbinden wird ein Intervall gestartet, welches nach einem definierbaren Zeitraum die *removeOldFeatures*-Funktion aufruft, die alte Featuredaten aus dem *featureSource*-Datenmodell löscht. Falls der Service bereits verbunden war, wird das Intervall gestoppt. Zudem wird beim initialen Verbinden des Services der Layer der OpenLayers3 Komponente hinzugefügt. Dies erfolgt über die *olData*-Variable, welche von der *angular-openlayer-directive* erzeugt wurde.

```
olData.getMap().then(function(map) {
      map.addLayer(featureLayer);
});
```

ApplyRemoteData-Funktion

Innerhalb der *ApplyRemoteData*-Funktion wird über die vom Service empfangenen Features iteriert. Für jedes empfangene Feature wird anschliessend die *addOrUpdateFeature*-Funktion mit dem entsprechenden Feature als Parameter aufgerufen.

Die Aufgabe der *addOrUpdateFeature*-Funktion ist es, das empfangene Feature im Openlayers3-Datenmodell hinzuzufügen oder zu aktualisieren.

```
$scope.addOrUpdateFeature = function(object) {
	var id = object.id;

	if (id) {
		var featureSourceFeature = featureSource.getFeatureById(id);
		if (featureSourceFeature) {
			$scope.$broadcast('featureChanged',
				updateFeature(featureSourceFeature, object));
		} else {
			var createdFeature = createFeature(object);
			featureSource.addFeature(createdFeature);
			$scope.$broadcast('featureAdded', createdFeature);
		}
	}
};
```

Falls sich im Datenmodell bereits ein Feature mit demselben Identifier wie jener des übergebenen Features befindet, wird die *updateFeature*-Funktion aufgerufen. Falls noch kein Objekt mit dem übergebenen Identifier in der Feature Source existiert, wird die *createFeature*-Funktion mit dem übergebenen Objekt aufgerufen. Anschliessend wird das erzeugte Objekt der *featureSource* hinzugefügt.

Die *$scope.$broadcast*-Funktion ist eine AngularJS-Implementierung des Observer Patterns, welche auf dem *$scope* Objekt aufgerufen werden kann. Die Funktionsweise ist analog der eigenen Implementierung im Service. Es werden somit durch den *$scope.$broadcast*-Aufruf interessierte AngularJS-Objekte informiert.

CreateFeature-Funktion

OpenLayers3 bietet die Klasse *ol.format.GeoJSON* an, mit welcher ein sich im GeoJSON-Format befindliches Objekt in ein OpenLayers-Vector Objekt umgewandelt werden kann. Der *readFeature*-Funktion muss dafür ein GeoJSON-Objekt übergeben werden. Zusätzlich können noch weitere Attribute übergeben werden. Diese können unter anderem Informationen zum Koordinatenreferenzsystem des übergebenen GeoJSON-Features wie auch des Ziel-Koordinatenreferenzsystem beinhalten. Da die weiteren Attribute für alle Umwandlungen dieselben sind, wurden diese als Klassenvariable definiert.

```
var featureAttributes = {
	dataProjection : implementationConfig.dataProjection,
	featureProjection : mapConfig.mapProjection
};
```

Das *dataProjection*-Attribut gibt an, was für ein Referenzsystem von den vom Service empfangenen Daten erwartet wird. Da die Daten je nach verwendetem Service ein anderes Referenzsystem haben können, wird dieses Attribut aus der implementierungsspezifischen Konfiguration ausgelesen. Das *featureProjection*-Attribut gibt an, in welchem Referenzsystem die Daten auf der OpenLayers-Komponente dargestellt werden. Da die Daten von allen Services auf der gleichen Kartenkomponente gerendert werden sollen, wird diese Einstellung aus der Kartenkonfiguration ausgelesen.

```
function createFeature(object) {
	var currentFeature;
	if (object.geometry) {
		var geoJsonFeature = $scope.olGeoJSONFormat.readFeature(object,
				featureAttributes);
		...
		currentFeature = geoJsonFeature;
	} else {
		currentFeature = new ol.Feature();
		currentFeature.setId(object.id);
		currentFeature.setProperties(object.properties);
	}
	currentFeature.setStyle(styleService.getStyle(currentFeature));
	return currentFeature;
}
```

Die *createFeature*-Funktion prüft, ob eine Geometrie innerhalb des übergebenen Objektes existiert. Ist dies der Fall, kann die GeoJSON-Format-Klasse für die Umwandlung verwendet werden. Enthält das neu empfangene Objekt keine Geometrie, wird ein neues *ol.Feature*-Objekt manuell erzeugt. Die ID und Properties müssen in dem Fall manuell gesetzt werden. In einem nächsten Schritt wird dem Objekt ein Style gesetzt. Hierzu wird der konfigurierte *StyleService* (vgl. Unterkapitel 5.6 Style Service) verwendet, welcher für das übergebene Objekt ein OpenLayers3-Style erzeugt und diesen dem OpenLayers3-Feature zuweist.

Das frisch erzeugte OpenLayers3-Vector Objekt wird zum Schluss als Funktionsrückgabewert zurückgegeben.

UpdateFeature-Funktion

Im Unterschied zur *addFeature*-Funktion werden in der *updateFeature*-Funktion lediglich die Properties und die Geometrie im Feature, welches sich bereits im *featureSource*-Datenmodell befindet, mit den Informationen des neuen Objektes überschrieben.

```
function updateFeature(featureSourceFeature, object) {
	var properties = featureSourceFeature.getProperties();
	$.extend(true, properties, object.properties);
	featureSourceFeature.setProperties(properties);
	if (object.geometry) {
		var geometry = $scope.olGeoJSONFormat.readGeometry(object.geometry,
				featureAttributes);
		featureSourceFeature.setGeometry(geometry);
		...
	}
	styleService.updateStyle(featureSourceFeature);
```

```
	return featureSourceFeature;
}
```

Die Properties des sich in der *featureSource* befindlichen Objektes wird mit Hilfe der *$.extend*-Funktion mit den neuen Daten zusammengefügt. Die *properties* des alten Features werden mit jenen des neuen Features aktualisiert. Somit befinden sich im properties-Objekt nach dem Aufruf der *$.extend*-Funktion die aktuellen, zusammengefügten Properties. Diese Properties werden dem *FeatureSource*-Feature zugewiesen.

Falls das empfangene Objekt ein Geometrie-Objekt beinhaltet, wird dieses mit Hilfe des OpenLayers Datenformats eingelesen. Da die verschiedenen Properties bereits zusammengefügt sind, wird in diesem Fall die *readGeometry*-Funktion aufgerufen, welche lediglich ein OpenLayers-Geometrie Objekt ohne Properties erzeugt. Dieses Geometrie Objekt wird darauf im *FeatureSource*-Feature als Geometrie zugewiesen.

Bevor das *FeatureSource*-Feature als Funktionsrückgabewert zurückgegeben wird, wird das Styling des *FeatureSource*-Features durch den *StylerService* aktualisiert.

Die Änderungen, welche an einem Feature Objekt beim Ändern der Attribute automatisch gemacht werden, werden zur OpenLayers3-Komponente notifiziert; so, dass diese umgehend über die Änderungen informiert ist und entsprechend die Karte und die Features neu zeichnen kann.

RemoveOldFeatures-Funktion

Die *RemoveOldFeatures*-Funktion wird in regelmässigen Abständen aufgerufen, sobald der Service verbunden ist.

```
$scope.removeOldFeatures = function() {
	var currentMillis = new Date();
	var oldestMoment = currentMillis - implementationConfig.cleanupInterval;
	featureSource.forEachFeature(function(feature) {
		var featureSeenDate = feature.get('messageReceived');
		if (oldestMoment > featureSeenDate) {
			featureSource.removeFeature(feature);
			$scope.$broadcast('featureRemoved', feature);
		}
	});
};
```

Innerhalb dieser Funktion wird über alle Features, die sich im *featureSource*-Datenmodell befinden, iteriert. Für jedes Feature wird das *messageReceived*-Attribut überprüft. Wurde ein Feature seit der Dauer des konfigurierten *cleanupInterval*-Parameters nicht geändert, ist der *messageReceived*-Zeitpunkt eines Features älter als der aktuelle Zeitpunkt minus der *cleanupInterval*-Dauer. In diesem Fall wird das entsprechende Feature aus dem Datenmodell gelöscht. Zudem werden durch eine Broadcast Meldung mit dem Namen *featureRemoved* etwaige interessierte AngularJS-Objekte über das Entfernen des Features informiert.

5.4.3 Konfigurationsmöglichkeiten

Einstellungen am Map Controller können mittels drei verschiedenen Konstanten vorgenommen werden:

- Über die *appConfig*-Konstante

- Mit Hilfe der *mapConfig*-Konstante

- Mit der variantenabhängigen Konfiguration (*sosConfig* oder *websocketConfig*)

Die *appConfig*-Konstante beinhaltet die Konfigurationen, welche die gesamte Applikation betreffen, so auch die Konfiguration, mit welchen Attributen die Map Controller initialisiert werden sollen. Einstellungen, die in der *mapConfig*-Konstanten definiert werden, beziehen sich auf die Darstellung der OpenLayers3-Komponente. Die *sosConfig*- respektive die *websocketConfig*-Konstanten beinhalten zusätzlich zu den bereits im Service verwendeten Attributen Einstellungsmöglichkeiten, die je nach Variante im Map Controller unterschiedlich sein können.

Die *appConfig*-Konstante enthält nur ein Property. Dieses Property ist jedoch von grosser Wichtigkeit und beschreibt wie viele Map Controller erzeugt und mit welchen Attributen diese initialisiert werden sollen:

Property	Typ	Erläuterung
appConfig.mapPages	JavaScript-Array	Konfiguration der Seiten, welche einen Map Controller verwenden. Neben einem eindeutigen Identifier kann der Anzeigename und die URL angegeben werden, unter welcher dieser Map Controller verwendet werden soll. Zudem kann die variantenabhängige Konfigurationsdatei (z.B. *sosConfig* oder *websocketConfig*) und der zu verwendende Daten-Service (*SOSService* oder *Websocket* Service) wie auch der Style Service (z.B. *AircraftStyleService*) angegeben werden.

Tabelle 30: Konfigurationsmöglichkeiten Applikation

Die Konfiguration *mapConfig*-Konstante umfasst folgende Konfigurationsmöglichkeiten, die das Verhalten der Openlayers3-Komponente beeinflussen:

Property	Typ	Erläuterung
mapConfig .mapProjection	Zeichenkette	Gibt an, nach welchem Koordinatenreferenzsystem die Karte und die Features gerendert werden sollen (z.B. "EPSG:3857" für eine Mercator Projektion).

Property	Typ	Erläuterung
mapConfig.olCenter	JavaScript-Objekt	Zentrum, auf welches beim Öffnen der Webseite gezeigt werden soll. Ein Zoomlevel kann angegeben werden, damit der darzustellende Kartenausschnitt besser eingestellt werden kann. Das folgende Beispiel zeigt auf das Zentrum der Schweiz mit einem Zoomlevel, welcher in der Regel die ganze Schweiz sichtbar macht: `"olCenter": { "lat": 46.801111, "lon": 8.226667, "zoom": 7 }`
mapConfig .olBackgroundLayer	JavaScript-Objekt	Mit dieser Einstellung kann die darzustellende Hintergrundkarte konfiguriert werden. Das folgende Beispiel zeigt, wie OpenStreetMap[19] als Hintergrundkarte konfiguriert werden kann: `"olBackgroundLayer": { "source": { "type": "OSM" } }`
mapConfig.olDefaults	JavaScript-Objekt	Mit den *olDefaults*-Einstellungen können weiterführende Einstellungen an der Openlayers-Darstellung vorgenommen werden; beispielweise, ob Zoom-Buttons dargestellt werden sollen oder ob mit der Maus gescrollt werden kann. Diese Einstellungsmöglichkeiten entsprechen jenen, die in der *angular-openlayers-directive* konfiguriert werden können

Tabelle 31: Konfigurationsmöglichkeiten OpenLayers3-Komponente

Über die variantenabhängige Konfigurationskonstante können zudem Einstellungen vorgenommen werden, welche die Daten oder Konfigurationen der jeweiligen Variante betreffen. Die Einstellungen können in der entsprechenden Service-Konstanten vorgenommen werden. Deshalb wird die Konfigurationskonstante in dieser Auflistung als *implementationConfig* bezeichnet:

19 OpenStreetMap – http://www.openstreetmap.org [letzter Zugriff 21.06.2015]

Property	Typ	Erläuterung
implementationConfig .cleanupInterval	Nummer	Bestimmt den Zeitraum in Millisekunden, wie lange ein Feature in der Openlayers-Komponente dargestellt werden soll, sofern das Feature nicht mit neuen Daten aktualisiert wird. Wird innerhalb dieses *cleanup*-Intervalls ein Feature nicht aktualisiert, wird es aus der Openlayers Darstellungkomponente gelöscht.
implementationConfig .featureProjection	Zeichenkette	Diese Einstellung hilft Openlayers, das Objekt im richtigen Referenzsystem einzulesen (z.B. "EPSG:4326" für WGS-84)

Tabelle 32: Konfigurationsmöglichkeiten Gesamtapplikation

Die sich im Anhang befindliche Konfiguration für den Webclient (vgl. Anhang A.4 Client Konfigurationsdatei) enthält die Konfiguration, wie sie teilweise für die Vergleichsmessungen verwendet wurde.

5.5 View

Die View ist eine HTML-Seite, die mit verschiedenen HTML und AngularJS spezifischen Elementen beschrieben wird. Innerhalb der View wird definiert, an welcher Stelle welches Element dargestellt werden soll. Durch die Verwendung der *angular-openlayers-directive* kann der View Code schlank gehalten werden. Für die Darstellung der OpenLayers3-Komponente muss lediglich folgendes Element für die Darstellung eingefügt werden.

```
<openlayers custom-layers="true" ol-defaults="olDefaults"
      ol-center="olCenter" height="100%">
      <ol-layer ol-layer-properties="olBackgroundLayer" />
</openlayers>
```

Durch die Angabe des Elementnamen *openlayers* wird die *angular-openlayers-directive*-Erweiterung angesprochen. Diese wandelt das *openlayers*-Element in eine OpenLayers-Komponente um. Die im Map Controller von den Konfigurationskonstanten ausgelesenen OpenLayers-Einstellungen (*olDefault*, *olCenter* und *olBackgroundLayer*) werden entsprechend der Verwendung der Variable als Konfiguration dem *openlayers*-Element übergeben.

5.6 Style Service

Ein Style Service erstellt einen OpenLayers3-Style für ein gegebenes Feature. Der Aufruf an den Style Service für das Erstellen eines Styles erfolgt aus dem Map Controller (vgl. Unterkapitel 5.4 Map Controller). Wird ein Feature im Map Controller aktualisiert, wird der Style Service erneut aufgerufen. Der Style Service kann in diesem Fall das Styling anpassen

sofern die Änderung das Styling beeinflusst. Der Style Service muss somit zwei Funktionen implementiert haben, damit dieser korrekt aus dem MapService aufgerufen werden kann:

Funktion	Beschreibung
getStyle(feature)	Gibt ein OpenLayers-Style zurück, welcher für das Rendering des angegebenen Features verwendet werden kann.
updateStyle(feature)	Aktualisiert das Style-Objekt innerhalb des übergebenen Features, damit der Style den Attributen des Features entspricht.

Tabelle 33: Style Service Methodendefinition

Im Rahmen dieser Arbeit entstand ein Style Service, welcher ein Feature mit einem Flugzeugsymbol darstellt und das Symbol entsprechend der Ausrichtung des Flugzeuges (*heading*-Property) darstellt. Zudem wird das Callsign, sofern eines empfangen wurde, süd-östlich des Flugzeugsymboles dargestellt. Der *StyleService* wird als *AircraftStyleService* bezeichnet. In Abbildung 25 wird das Ergebnis eines mit dem *AircraftStyleService* gestylten Features aufgezeigt.

Abbildung 25: Styling Flugobjekte

Falls andere Daten als Flugobjekte an den Webclient übertragen werden sollen, kann ein eigener *StyleService* implementiert werden, welcher die Daten fachlich korrekt darstellen kann.

5.7 MenuController

Der Menu Controller ist ein AngularJS-Controller für die Menüleiste. Der *MenuController* ist verantwortlich für das Verbinden und Trennen von den Services. Abhängig davon, welche Variante aktuell auf der Webseite dargestellt wird, soll beim Drücken von "Connect" der Websocket-Service oder der SOS-Service verbunden werden. Der Menu Controller kennt nur die Services, die Map Controller sind dem Menu Controller nicht bekannt. Um einen Service zu verbinden ruft dieser direkt auf dem Service die *connect*-Funktion auf. Ein Map Controller, der an einer Statusänderung eines Services interessiert ist, wird durch das Publizieren des Services über dessen Statusänderung informiert.

6. Versuchsaufbau

In diesem Kapitel wird erörtert, wie das Testsystem aufgebaut ist, welche Hardware eingesetzt wird und wie das in Kapitel 3 vorgestellte Konzept für die Vergleichsmessungen von ADS-B Daten konfiguriert ist. Am Ende des Kapitels wird der lauffähige Webclient anhand von Screenshots aufgezeigt und beschrieben.

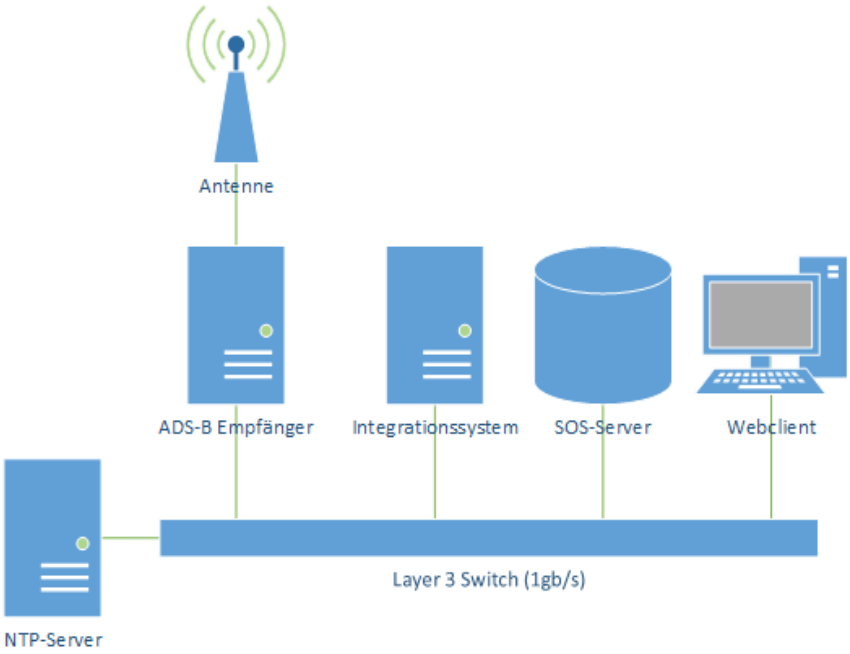

Abbildung 26: Komponenten des Testsystems

Das Testsystem ist gemäss Abbildung 26 aufgebaut. Die einzelnen Komponenten werden in den folgenden Unterkapiteln im Detail beschrieben.

Damit die Resultate vergleichbar sind, werden die Vergleichsmessungen für beide Varianten auf dem gleichen Testsystem durchgeführt. Dadurch, dass das Netzwerk ausschliesslich für die Vergleichstests verwendet wird, kann die gesamte Netzwerkbandbreite für die Vergleichsmessungen verwendet werden.

6.1 Netzwerk Switch

Die verschiedenen Komponenten sind über einen Netzwerkswitch miteinander verbunden. Die Hardware des im Testsystem verwendeten Netzwerkswitches entspricht einem *Netgear Pro Safe 8 Port Gigabit Smart Switch GS108T*[20]. Dieses Modell unterstützt Übertragungsraten von bis zu bis 1000 Mbps.

20 Netgear GS108T - http://www.downloads.netgear.com/files/GDC/GS108Tv1/gs108t_26march07.pdf [letzter Zugriff 03.05.2015]

Auf der Administrationsoberfläche des Netzwerkswitches können verschiedene Statistiken zur Netzwerknutzung ausgelesen werden. Unter anderem kann die Menge der übertragenen Bytes pro Netzwerkanschluss entnommen werden.

6.2 NTP-Server

Der NTP-Server ist eine Netzwerkkomponente, auf welchem ein Zeitdienst läuft. Dieser Dienst wird für eine adäquate Berechnung der Latenzzeiten verwendet.

6.2.1 Verwendete Hardware

Als Hardware wird ein Banana-Pi Board[21] verwendet.

Prozessor	ARM Cortex-A7 dual-core, 1 GHz, Mali400MP2 GPU
Arbeitsspeicher	1 GB
Netzwerkkarte	1000 Mbps

Tabelle 34: NTP-Server Hardware

6.2.2 NTP-Server

Eine zentrale Rolle beim Testsystem für die Berechnung der Latenzzeit hat der Network Time Protocol (NTP)-Server. Der NTP-Service wird für die Zeit-Synchronisierung der verschiedenen Komponenten verwendet. Das NTP Protokoll wurde von David L. Mills entwickelt. Das Synchronisieren der Zeit mittels NTP über das Internet (WAN) ist auf wenige Millisekunden genau (Mills, 1991). Beim Einsatz des NTP-Servers in einem lokalen Netzwerk kann die Genauigkeit in den Sub-Millisekunden Bereich verbessert werden (Huston, 2015). Durch das Betreiben eines eigens dafür vorgesehenen NTP-Server im lokalen Netzwerk und einer Datenrate von 1000 Mbps wird eine entsprechende Genauigkeit angestrebt. Alle Netzwerkkomponenten (Layer 3 Switch, Integrationssystem, SOS-Server und der Client-Rechner) haben den NTP-Server als einzigen NTP-Server konfiguriert. Dadurch kann sichergestellt werden, dass alle Netzwerkkomponenten auf circa eine Millisekunde genau die exakt gleiche Uhrzeit führen. Der NTP-Server wird auf der Linux Distribution Bananian[22] betrieben.

6.3 ADS-B Empfang

Wie in Unterkapitel 2.1 erwähnt, werden ADS-B Daten unverschlüsselt auf der Frequenz 1090 MHz von den Flugzeugen aus publiziert. Diese Daten können mittels Einsatz von wenigen Hilfsmitteln empfangen und decodiert werden. Als Hardware wird, abgesehen von einem Computer mit einer USB-Buchse, lediglich ein terrestrischer Digital Fernseh-Empfänger (DVB-T Empfänger) in Form eines USB-Sticks, welcher Software Defined Radio

21 Banana Pi - http://www.bananapi.org/ [letzter Zugriff 07.05.2015]
22 Homepage Bananian Linux - https://www.bananian.org [letzter Zugriff 19.06.2015]

(SDR) unterstützt, benötigt. Als *Software Defined Radio* wird ein Signalverarbeitungskonzept bezeichnet, bei welchem versendete Radio Wellen zu einem grossen Teil mit Hilfe von Software umgewandelt werden.

6.3.1 Verwendete Hardware

Für den Einsatz im Vergleichssystem wurde ein kostengünstiger DVB-Stick von GiXA Technology[23] verwendet, mit welchem SDR mittels GNU Radio möglich ist. Der USB-Stick wird auf einem kostengünstigen, kreditkartengrossen Computer, einem Raspberry PI, betrieben[24], welcher über einen 100 Mbps Netzwerkanschluss an das lokale Netzwerk (LAN) angeschlossen ist.

6.3.2 Verwendete Software

Der Raspberry PI wird mit der Linux-Distribution Raspian[25] betrieben.

Für den Empfang von Radiosignalen und für deren Verarbeitung wurde das Open Source Framework GNU Radio entwickelt. Die initiale Version wurde von Eric Blossom erstellt und veröffentlicht (Blossom, 2004). Damit ist es möglich Radio Signale in digitale Daten umzuwandeln. Für den Empfang von ADS-B Daten wurde basierend auf GNU Radio die Open Source Software *dump1090*[26] entwickelt, welche von Salvatore Sanfilippo unter dem Spitznamen *antirez* auf github entwickelt wird. Die Software *dump1090* wird für das Betreiben im Testsystem mit folgendem Befehl ausgeführt:

```
dump1090 --net --quiet
```

Der --*net*-Parameter wird verwendet, damit die Netzwerkfähigkeit von *dump1090* aktiviert wird. Durch die --quiet Angabe wird die Ausgabe in der Konsole unterdrückt. Ist diese aktiv, können über verschiedene Ports Daten empfangen oder versendet werden.

Port	Typ
30001	Über den Port 30001 können Mode-S Rohdaten an dump1090 gesendet werden.
30002	Über den Port 30002 können Mode-S Rohdaten, welche empfangen wurden, abgefragt werden. Die Daten werden nach dem Empfang sofort über diesen Port versendet.
30003	Verbindet ein Client sich mit diesem Port, erhält dieser Daten im SBS1 Socket Output-Format (vgl. Kapitel 2.4.1).

Tabelle 35: Port Konfiguration ADS-B Empfänger (dump1090)

23 GiXa DVB-T USB Stick - http://www.amazon.de/GiXa-Technology-Fernsehen-Bearbeitung-Fernbedienung/dp/B004W4INCM [letzter Zugriff 09.01.2015]
24 Raspberry Pi - https://www.raspberrypi.org/ [letzter Zugriff 10.01.2015]
25 Raspbian – http://www.raspbian.org/ [letzter Zugriff 10.01.2015]
26 dump1090 - https://github.com/antirez/dump1090 [letzter Zugriff 09.01.2015]

6.4 Integrationssystem

Das in Kapitel 4 vorgestellte Integrationssystem wird im Testsystem auf einem eigenen Rechner ausgeführt.

6.4.1 Verwendete Hardware

Der Rechner ist mit folgender Hardware ausgestattet:

Prozessor	Quadcore Intel(R) Core(TM) i7 CPU M640 @ 2.80GHz
Arbeitsspeicher	8 GB
Netzwerkkarte	1000 Mbps

Tabelle 36: Hardware Integrationssystem

6.4.2 Verwendete Software

Das Integrationssystem wird auf einem Ubuntu Linux[27] 12.04.5 LTS ausgeführt. Für das Ausführen des Integrationssystems wird das Java OpenJDK[28] in der Version 1.6.0_33 verwendet. Bei den verschiedenen Vergleichsmessungen wird die Konfiguration dieses Systems an die jeweilige Vergleichsmessung angepasst. Für die Filterung der Daten auf dem Integrationssystem wird die Konfiguration entsprechend den in Kapitel 3.1 definierten Zieldaten angepasst. Im Anhang wurde eine Standardkonfiguration des Integrationssystems (vgl. Anhang A.3 Integrationssystem Konfigurationsdatei) angehängt, wie sie für den ersten Vergleichstest (vgl. Kapitel 7.1 Übertragungsqualität) verwendet wurde.

6.5 SOS-Server

Auf dieser Netzwerkkomponente wird der SOS mit einer Datenbank für die Persistierung der Daten ausgeführt.

6.5.1 Verwendete Hardware

Prozessor	Quadcore Intel(R) Core(TM) i7 CPU Q720 @ 1.60 GHz
Arbeitsspeicher	6 GB
Netzwerkkarte	1000 Mbps

Tabelle 37: Hardware SOS-Server

6.5.2 Verwendete Software

Als Betriebssystem wird auf diesem Rechner Ubuntu Linux 14.04.2 LTS verwendet. Der SOS-Service wird in einem Apache Tomcat Applikationsserver mit der Version 8.0.9

27 Ubuntu Linux - http://www.ubuntu.com/ [letzter Zugriff 21.06.2015]
28 OpenJDK - http://openjdk.java.net/ [letzter Zugriff 21.06.2015]

ausgeführt. Als SOS wird die Implementierung von 52°North[29] in der Version 4.1.4 eingesetzt. Der Service wurde ohne spezielle Konfiguration installiert. Er wird mit einer PostgreSQL-Datenbank[30] in der Version 9.3 betrieben. Zudem wurde die Spatial-Erweiterung für die PostgreSQL-Datenbank, PostGIS[31], in der Version 2.1 installiert.

6.5.3 Konfiguration SOS

Dem SOS wurde gemäss des im Anhang (vgl. Anhang A.1) beigefügten *InsertSensor*-Statements einen Sensor eingefügt, so dass die Daten gemäss des definierten Konzepts korrekt an den SOS gesendet und ausgelesen werden können.

6.6 Webclient

Auf dem Webclient wird die in Kapitel 5 (Umsetzung Webclient) vorgestellte Webanwendung ausgeführt, welche die vom Integrationssystem versendeten Daten visualisieren kann.

6.6.1 Verwendete Hardware

Der Rechner, auf welchem die Webanwendung angezeigt wird, ist mit folgenden Hardwarekomponenten ausgestattet:

Prozessor	Quadcore Intel(R) Core(TM) i5 CPU M520 @ 2.40 GHz
Arbeitsspeicher	4 GB
Netzwerkkarte	1000 Mbps

Tabelle 38: Hardware Clientsystem

6.6.2 Verwendete Software

Das Betriebssystem dieses Rechners ist Ubuntu Linux 14.04.2 LTS. Die Webseite wird durch einen leichtgewichtigen Webserver zur Verfügung gestellt, welcher auf demselben Rechner gestartet wird. Hierfür wird GRUNT[32] in der Version 0.4.5 verwendet. Angezeigt wird die Webanwendung in einem Mozilla Firefox Browser in der Version 38.0.

6.6.3 Konfiguration Webclient

Konfiguriert wird der Webclient gemäss der im Anhang beigefügten Konfiguration (vgl. Anhang A.4).

In den Abbildungen 27 und 28 wird die Graphische Oberfläche dargestellt, wie sie beim Verwenden des Webclients erscheint. Obwohl die beiden Bilder identisch aussehen, wurden die Bilder parallel aufgenommen. Bei Abbildung 27 wurden die Daten über die Websocket-

29 52°North SOS - http://52north.org/communities/sensorweb/sos/ [letzter Zugriff 21.06.2015]
30 PostgreSQL - http://www.postgresql.org/ [letzter Zugriff 21.06.2015]
31 PostGIS - http://postgis.net/ [letzter Zugriff 21.06.2015]
32 Grunt – http://gruntjs.com [letzter Zugriff 07.05.2015]

Variante übertragen. Die dargestellten Flugobjekte in Abbildung 28 wurden hingegen über die SOS-Variante empfangen. Die Hintergrundkarte wurde bei beiden Varianten auf die OpenStreetMap-Standard Karte konfiguriert. Im oberen Bereich befindet sich die Menüleiste, in welcher die beiden Karten geöffnet werden können. Mit dem grossen hellblauen Button kann man den Service der gerade geöffneten Variante verbinden. Durch erneutes Drücken wird der Service wieder getrennt. In der oberen rechten Ecke wird die Anzahl der Meldungen dargestellt, die der entsprechende Service empfangen hat.

Die in den Abbildungen dargestellten Flugobjekte wurden mit einem eigenen ADS-B -Empfänger empfangen und wurden via dem Integrationssystem auf die beiden Webclients verteilt.

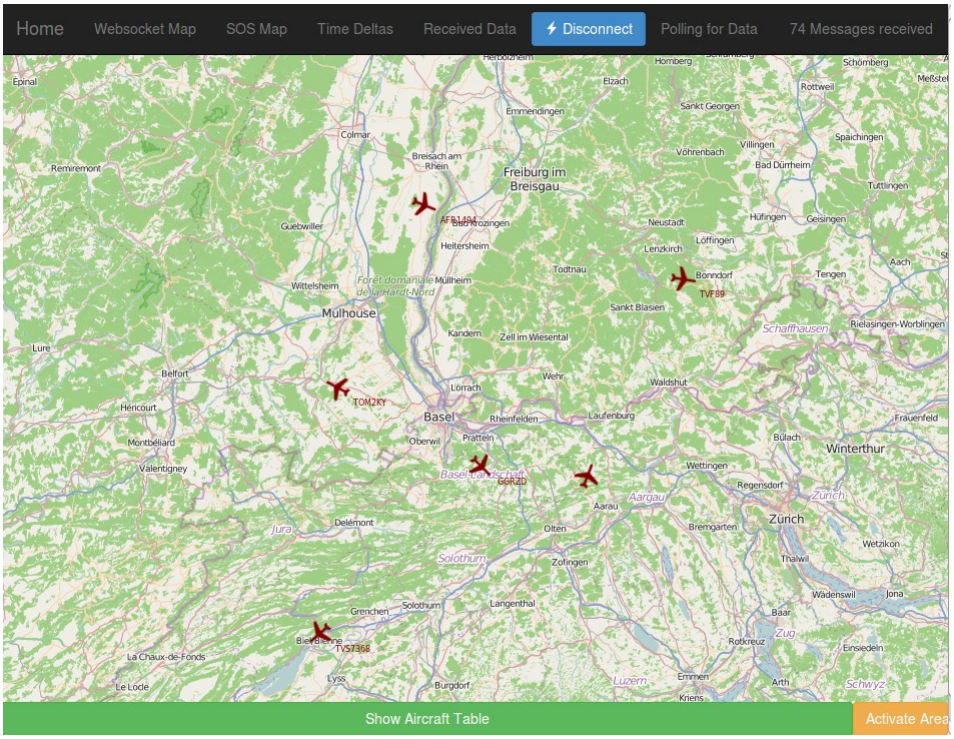

Abbildung 27: Benutzeroberfläche SOS-Variante

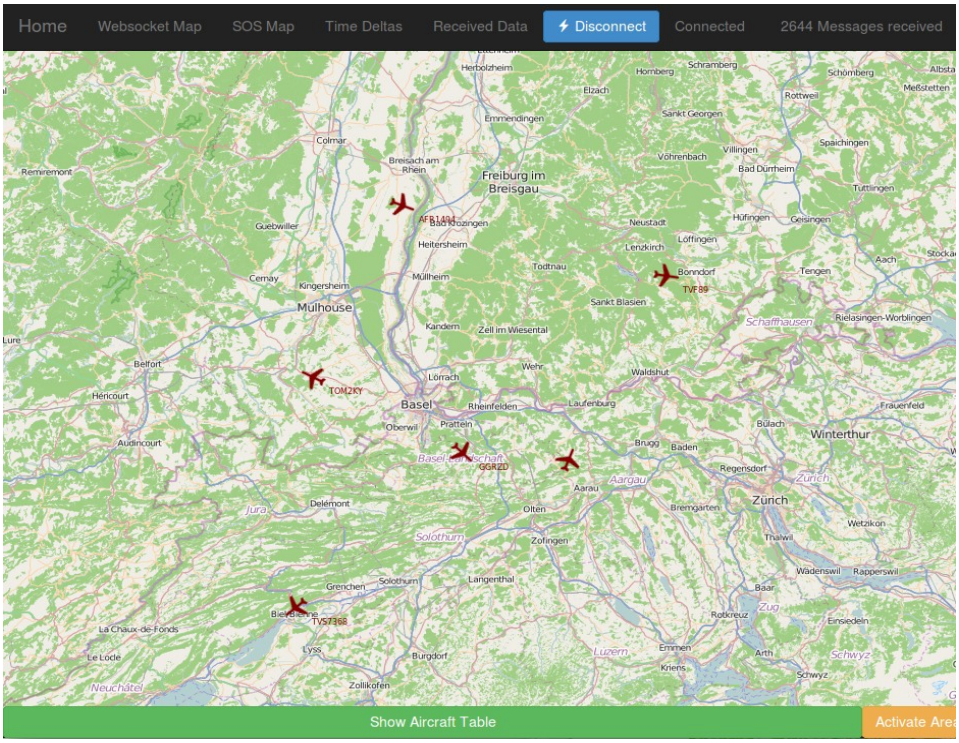

Abbildung 28: Benutzeroberfläche Websocket-Variante

7. Vergleichsmessungen

In diesem Kapitel werden die beiden Varianten anhand von verschiedenen Vergleichsmessungen miteinander verglichen. In Unterkapitel 7.1 wird die Qualität der übertragenen Daten verglichen. Damit soll untersucht werden, ob die Daten über beide Varianten korrekt übertragen werden. In den Unterkapiteln 7.2 – 7.4 werden Vergleichsmessungen durchgeführt, in welchen die Performance der beiden Varianten untersucht wird.

7.1 Übertragungsqualität

Mit dieser Vergleichsmessung wird überprüft, ob die an das Integrationssystem übertragenen Daten korrekt auf den Webclient übertragen werden. Es soll aufgezeigt werden, dass die Messwerte, welche über den SBS1-Socket Output auf dem Integrationssystem empfangen werden, über beide Varianten unverändert zum Webclient übertragen werden.

7.1.1 Ablauf

Bei diesem Versuch wird der ADS-B Empfänger 90 Minuten lang an das Integrationssystem angeschlossen. Auf dem Integrationssystem werden die empfangenen Daten im GeoJSON-Format in ein Logfile geschrieben. Auf dem Client wird die Webanwendung sowohl für die SOS- wie auch die Websocket-Variante gestartet. Beide Varianten werden somit parallel betrieben. Dadurch ist sichergestellt, dass exakt dieselben Daten über die beiden Varianten verteilt werden. Auf Clientseite werden die empfangenen Daten im Map Controller jeweils in einen HTML5-Localstore geloggt. Die Daten werden wie auch auf Serverseite im GeoJSON Format geloggt. Der Localstore kann nach Beendigung der Vergleichsmessung ausgelesen werden, so dass die Daten mit dem Serverlogfile verglichen werden können.

7.1.2 Datenanalyse

Die GeoJSON-Features auf dem Webclient weisen zusätzliche Attribute zu jenen auf dem Server auf (im Speziellen *id* und *properties-messageReceived*). Bei der SOS-Variante ist zudem das *hexIdent*-Property nicht gesetzt. Hingegen ist die *id* im Objekt mit der FeatureOfInterest URL befüllt. Auch die Sortierung der Attribute innerhalb des Objektes kann unterschiedlich sein. Als Beispiel wird dieselbe Meldung aus den drei Logfiles dargestellt:

Server:


```
{
	"geometry":	{
		"type": "Point",
		"coordinates": [7.68089, 47.41214]},
	"properties": {
		"hexIdent": "4B178D",
		"altitude": 14150,
		"messageGenerated": 1434438037171},
	"type": "Feature"
```

```
}
```

Websocket:

```
{
	"geometry": {
		"type": "Point",
		"coordinates": [7.68089,47.41214]},
	"properties": {
		"hexIdent": "4B178D",
		"altitude": 14150,
		"messageGenerated": 1434438037171,
		"messageReceived": 1434438037152 },
	"type": "Feature",
	"id": "4B178D"
}
```

SOS:

```
{
	"type": "Feature",
	"properties": {
		"messageReceived": 1434438037449,
		"messageGenerated": 1434438037171,
		"altitude": 14150 },
	"id": "http://stue.ch/sensorobservation/foi/aircraft/4B178D",
	"geometry": {
		"type": "Point",
		"coordinates": [7.68089, 47.41214]
	}
}
```

Darüber hinaus kann sich die Anzahl der geloggten Meldungen bei der SOS-Variante zum Server-Log unterscheiden. Dieses Phänomen tritt auf, wenn zwei SBS1-Meldungen für das gleiche Flugobjekt mit dem gleichen Timestamp aber mit unterschiedlichen Properties gesendet wird. Auf dem Server und in der Websocket-Variante werden hierfür zwei Meldungen geloggt. In der SOS-Variante hingegen erscheint das Feature mit allen Properties in einer Meldung.
Aus diesen Gründen wurde im Rahmen dieser Arbeit das Script *filterAndSort.js* geschrieben, welches lediglich die konfigurierten Werte aus den Features ausliest und doppelte Meldungen pro Zeitpunkt zusammenfügt. Zudem werden Meldungen mit dem gleichen Flugobjekt Identifier und dem gleichen Timestamp zusammengeführt. Das Resultat wird als JSON ausgegeben und kann wiederum in eine Datei gespeichert werden. Das Script kann über eine JSON Datei konfiguriert werden. Für die beiden Client-Logs sieht die Konfiguration folgendermassen aus:

```
{
  "values": {
    "properties": {
      "altitude": "altitude",
      "messageGenerated": "messageGenerated",
      "groundSpeed": "groundSpeed",
      "heading": "heading"},
    "id": "id",
    "geometry": {"coordinates": {"0": "longitude", "1": "latitude"}}
  },
```

```
  "identifier": [
    "id",
    "messageGenerated"
  ]
}
```

Im *values*-Feld können jene Werte mit einem Namen angegeben werden, welche im erzeugten Script enthalten sein sollen und es kann bestimmt werden, wie die Werte im neuen Script benannt werden sollen. Es werden somit die bereits erwähnten Zieldaten (vgl. Kapitel 3.1 Zieldaten) konfiguriert. Im *identifier*-Feld kann definiert werden, nach welchen Kriterien die Gruppierung erstellt werden soll. Somit werden die Objekte zuerst nach *id* und anschliessend nach *messageGenerated* gruppiert. Haben mehrere Objekte dieselbe *id* und denselben *Erzeugungszeitpunkt*, werden die Daten zusammengefügt.
Für das Server-Logfile muss das *hexId*-Property als Identifier ausgelesen werden. Deshalb wurde die Konfiguration folgendermassen angepasst:

```
{
  "values": {
    "properties": {
      "altitude": "altitude",
      ...
      "heading": "heading",
      "hexIdent": "id"},
    "geometry": {"coordinates": {"0": "longitude", "1": "latitude"}}
  },
  ...
}
```

Nachdem das Script für alle Objekte ausgeführt wurde, muss in der SOS-Variante lediglich der URL Prefix (*http://stue.ch/sensorobservation/foi/aircraft/*) bei der ID gelöscht werden. Dies kann mittels Suchen und Ersetzen erfolgen. Auf der Linux Kommandozeile kann dies mit dem *sed* Befehl erfolgen:

```
cat sos_log.json| sed 's/http\:\/\/stue\.ch\/sensorobservation\/foi\/aircraft\///g' >
sos_log_noidpref.json
```

Um die drei erzeugten JSON-Dateien vergleichen zu können wird schliesslich das Open Source Programm *jsondiffpatch*[33] verwendet, welches von Benjamín Eidelman auf GitHub weiterentwickelt wird. Das *jsondiffpatch* Programm gibt alle Zeilen aus, welche sich im JSON-Objekt unterscheiden.
Das *jsondiffpatch* Programm wird nun für zwei Vergleiche der vom *filterAndSort.js* erzeugten Logfiles durchgeführt. Im ersten Vergleich werden die Server Logfile-Daten mit jenen des SOS-Logfiles verglichen. Der zweite Vergleich wird vom Server-Logfile zum Websocket-Logfile durchgeführt. Ist das Ergebnis in beiden Fällen ein leerer Rückgabewert, so wurden bei beiden Varianten dieselben Datenwerte empfangen.

7.1.3 Ergebnisse

In einem ersten Versuch welcher lediglich etwas über 11 Minuten dauerte, um den Versuchsaufbau zu überprüfen, wurden 10'434 Meldungen vom ADS-B Empfänger über das

33 jsondiffpatch - https://github.com/benjamine/jsondiffpatch [letzter Zugriff 16.06.2015]

Integrationssystem für beide Varianten übertragen. Beim Evaluieren der Client Logfiles wurde ersichtlich, dass die SOS-Variante 2 Messwerte nicht erhalten hat. Nach eingehender Analyse konnte das Problem identifiziert werden: Es tritt auf, wenn für ein Flugobjekt mehrere Meldungen für den gleichen Zeitpunkt gesendet werden. Falls der Client just zu dem Zeitpunkt eine Anfrage an den SOS-Server sendet, bei welcher erst die erste Meldung in den SOS-Server eingefügt wurde, werden nur die Werte der ersten Meldung zum Client übertragen. Bei der nächsten Anfrage an den SOS ist der Zeitpunkt der zweiten Meldung nicht mehr im Abfragezeitraum, womit diese in der Abfrage nicht gefunden und zurückgegeben wird.

Für die Behebung dieses Problems wurden zwei Lösungen im SOS-Client Service evaluiert:

Lösung 1

Der Abfragezeitraum an den SOS-Server soll weiter in die Vergangenheit verschoben werden können. Dadurch hat das Integrationssystem genügend Zeit, die Werte in den SOS zu schreiben, bevor diese Abgefragt werden.

Vorteile:

- ✔ Es werden gleich viele Daten übertragen wie zuvor.

Nachteile:

- ✗ Die Aktualität der Daten auf dem Client leidet.
- ✗ Die Einstellung, wie gross die Verzögerung sein soll, ist nicht eindeutig zu bestimmen, da der Client nicht weiss, wie stark ausgelastet der Server ist und wie viele neue Daten aktuell zum Server gesendet werden. Wird der Parameter zu klein gewählt kann das oben beschriebene Problem erneut auftreten; wird der Parameter zu gross gewählt, leidet die Aktualität der Daten.

Lösung 2

Aktuell wird der Start-Zeitrange einer Anfrage an den SOS-Server jeweils auf den Wert der letzten empfangenen Meldung gesetzt, so dass diese bei der nächsten Abfrage nicht mehr enthalten ist. Eine Möglichkeit wäre es nun, den Start-Zeitrange so anzupassen, dass der Zeitpunkt der letzten erhaltenen Meldung bei der nächsten Abfrage noch enthalten ist. Damit würde in dem Fall die erste wie auch die zweite Meldung zum Client übertragen werden.

Vorteile:

- ✔ Die Aktualität der Daten wird nicht beeinträchtigt.

Nachteile:

- ✗ Es werden mehr Daten zum Client übertragen, da einige Meldungen mehrfach

übertragen werden.

Die beiden Lösungen wurden umgesetzt und können in der Client Konfigurationsdatei (vgl. Kapitel 5.3.3 Konfigurationsmöglichkeiten) über die folgenden Parameter eingeschaltet und konfiguriert werden:

```
{
"sosConfig":{
  "requestType": "application/xml",
  "jsonURL": "http://127.0.0.1:8080/52n-sos-webapp/sos/json",
  "poxURL": "http://127.0.0.1:8080/52n-sos-webapp/sos/pox",
  "updateInterval": 1000,
  "requestDelay": 0,
  "extendTimePeriod": true

...

}
```

Mit dem *requestDelay*-Parameter kann die Verzögerung der Requests an den SOS eingestellt werden: Ist der Wert 0, ist die Lösung 1 nicht aktiv. Der *extendTimePeriod*-Parameter aktiviert die Lösung 2.

Da die Lösung 2 weniger Nachteile aufweist und eindeutig konfiguriert werden kann, wurde der Vergleichstest mit aktivierter Lösung 2 erneut durchgeführt.

Insgesamt wurden in 90 Minuten 88'923 Meldungen über beide Varianten übertragen. Nach der Evaluation dieser Ergebnisse wurden keine Abweichungen der Werte festgestellt. Somit ist wird Übertragungsqualität sowohl bei der SOS- wie auch der Websocket-Variante als zuverlässig befunden.

7.2 Minimale Latenzzeit

In dieser Vergleichsmessung wird die minimale Dauer für die Übertragung von ADS-B Daten vom Integrationssystem bis zur clientseitigen Verarbeitung der Meldung im Map-Controller (vgl. Kapitel 5.4 Map Controller) in der Weboberfläche verglichen. Mit Hilfe dieses Vergleichs soll überprüft werden, ob die Erwartung, dass eine Push-Notifikation über Websocket an den Webclient erheblich schneller ist, als die Daten an einen SOS-Server zu *pushen* und die Daten vom Client aus mit einem Pull-Request abzufragen, in einem realen Experiment belegt werden kann.

7.2.1 Ablauf

Für diesen Versuch werden vom Integrationssystem simulierte ADS-B Meldungen übertragen. Der Vorteil beim Simulieren der Daten liegt darin, dass die Menge der Daten bestimmt werden kann. Der Versuch wird für beide Varianten einmal ausgeführt. Zunächst werden vom Integrationssystem simulierte ADS-B Daten über die SOS-Variante und anschliessend über die Websocket-Variante übertragen. Dadurch wird sichergestellt, dass die Belastung für das System bei beiden Varianten gleich gross ist und sich die beiden Varianten nicht gegenseitig beeinflussen. Bei dieser Vergleichsmessung wird pro Sekunde nur eine Meldung vom Integrationssystem erzeugt und mittels der getesteten Variante durch das System zur Visualisierung zum Webclient gesendet. Die Datenrate ist mit einer Meldung pro Sekunde bewusst klein gewählt worden, damit die Varianten nicht zu stark belastet werden und dadurch die minimale Latenzzeit beeinflusst würde.

Der Versuch wird bei beiden Varianten jeweils 60 Minuten lang mit den simulierten Daten ausgeführt. Es werden somit in etwa 3600 Meldungen je Variante zum Webclient übertragen.

Der SOS-Client wird so konfiguriert, dass nach dem Verbinden des Clients umgehend eine Anfrage an den SOS-Server gemacht wird. Sobald eine Anfrage vom SOS-Server zurückkommt, wird eine neue Anfrage vom Client aus initiiert. Dadurch wird versucht, die Latenzzeit für den SOS-Client minimal zu halten.

7.2.2 Datenanalyse

Da alle verwendeten Rechner über einen lokalen NTP-Server synchronisiert sind, sind die Uhren der Rechner im sub-Millisekunden-Bereich synchronisiert. Dies ermöglicht es, auf verschiedenen Rechnern erstellte Timestamps miteinander zu vergleichen. Auf den vom Integrationssystem simulierten ADS-B Meldungen wird das *messageGenerated*-Property auf den aktuellen Zeitpunkt gesetzt. Der Zeitpunkt wird in Millisekunden angegeben und repräsentiert die Anzahl vergangener Millisekunden seit dem 1. Januar 1970 um 00:00:00.000 der koordinierten Weltzeit (UTC). Auf dem Webclient werden die empfangenen Datensätze mit dem Empfangszeitpunkt ergänzt. Dieser Zeitpunkt wird dem Feature als

messageReceived-Property zugewiesen. Das *messageGenerated*- und *messageReceived*-Property wird auf dem Client zwischengespeichert, so dass diese nach dem Testdurchlauf ausgewertet werden können. Anschliessend werden die empfangenen Daten wiederum vom Map Controller aus in einen HTML5-Localstore geloggt. Für die Auswertung der Daten wird die Latenzzeit als Differenz vom *messageReceived*- zum *messageGenerated*-Property berechnet. Für die Auswertung werden die Daten aus dem Localstore in eine Datei geschrieben, aus welcher die *messageGenerated* und *messageReceived* Properties ausgelesen werden können. Mit Hilfe der berechneten Latenzzeiten sollen die beiden Varianten miteinander verglichen werden. Die Unterschiede werden dabei als erheblich betrachtet, wenn die Latenzzeiten sich um den Faktor zwei oder mehr unterscheiden.

7.2.3 Ergebnisse

Die durchschnittliche Latenzzeit der SOS-Variante, also die Zeit, die seit dem Erzeugen der Meldungen auf dem Integrationssystem vergeht bis zum Empfang der Meldung auf dem Clientsystem, lag bei dieser Vergleichsmessung bei 91.1ms. Die kürzeste Latenzzeit der SOS-Variante war bei 41ms. Der Maximalwert lag mit einem Faktor von etwa 1.5 zum Durchschnittswert bei 141ms. Wie dem Histogramm (vgl. Abbildung 29) zu entnehmen ist, sind die Werte in etwa normalverteilt. Der Peak mit mehr als 500 Meldungen liegt in etwa beim Durchschnittswert.

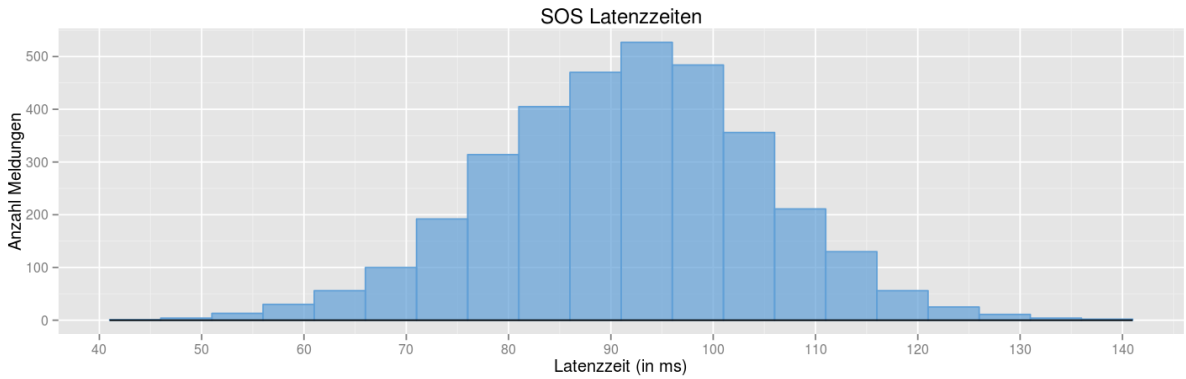

Abbildung 29: Histogramm SOS Latenzzeit

Bei der Websocket-Variante war die durchschnittliche Latenzzeit bei diesem Versuch mit 4.1ms deutlich tiefer als bei der SOS-Variante. Die minimalste Latenzzeit war mit dieser Variante bei einer Millisekunde. Die längste Latenzzeit bei der Websocket-Variante war mit 18ms noch immer fünfmal kürzer als die durchschnittliche Latenzzeit der SOS-Variante. Wie der Abbildung 30 entnommen werden kann, ist die Verteilung stark auf den Bereich von 2 -5 Millisekunden konzentriert. Die meisten Meldungen wurden mit einer Latenzzeit von 4 Millisekunden empfangen.

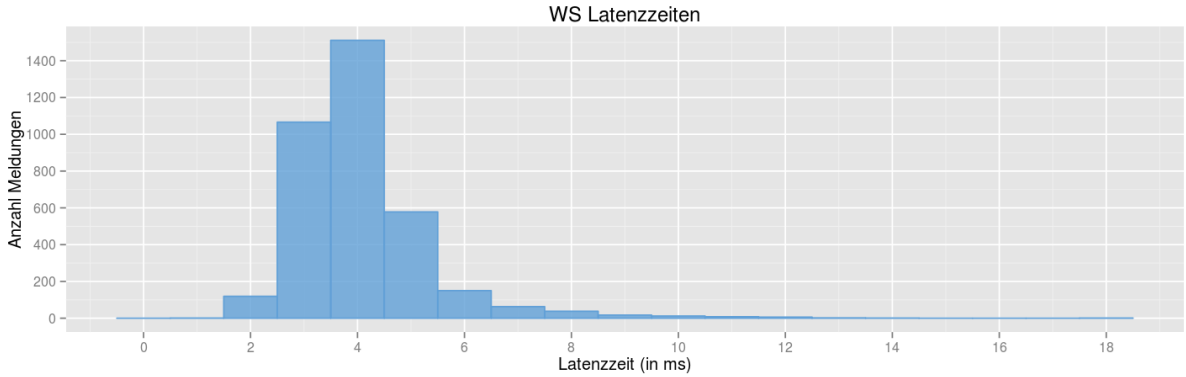

Abbildung 30: Histogramm Websocket Latenzzeit

Die durchschnittliche Latenzzeit der Websocket-Variante ist somit bei geringer Auslastung um den Faktor von etwa 22.5 kleiner als der durchschnittliche Wert der SOS-Variante. Werden die kürzesten Ergebnisse miteinander verglichen, so ist die Latenzzeit der Websocket-Variante sogar um etwa den Faktor 41 kürzer. Da alle Werte der etwa 3600 Meldungen bei der Websocket-Variante um mindestens den Faktor ~2.3 kürzer ausgefallen sind als jene der SOS-Variante wird dieses Resultat als erheblich angesehen.

7.3 Übertragungsmenge

Mit dieser dritten Vergleichsmessung wird die übertragene Datenmenge, welche pro Variante über das System in einem bestimmten Zeitraum versendet wird, verglichen. Die Erwartung ist, dass die Datenmenge bei den beiden Varianten aufgrund der verschiedenen Daten-Protokolle und -Formate stark unterschiedlich sein wird. Durch die Verwendung des wenig wortreichen JSON-Formates bei der bidirektionalen Websocket-Variante wird angenommen, dass die Übertragungsmenge bedeutend kleiner ist als bei der SOS-Variante. Bei der SOS-Variante werden für jedes Einfügen in den SOS-Server und bei jedem Auslesen jeweils zwei Meldungen versendet, da jeweils eine Anfrage und eine Antwort übertragen werden muss. Für das Einfügen in den SOS-Server und das Anfordern neuer Daten aus dem SOS wird zudem das XML-Datenformat verwendet, in welchem die Meldungen umfangreicher respektive ausführlicher beschrieben werden. Da üblicherweise nicht jeder Webclient einen eigenen Server respektive ein eigenes Integrationssystem hat, ist es relevant, dass die Übertragungsmenge vom respektive zum Webclient möglichst gering ist. Je kleiner diese Menge ist, umso weniger stark wird die Bandbreite des Servers respektive des Integrationssystems belastet. Aus diesem Grund wird bei diesem Vergleichstest die Client-Kommunikation gesondert betrachtet.

7.3.1 Ablauf

Wie auch bei der Berechnung der minimalen Latenzzeit wird der Test für beide Varianten nacheinander durchgeführt. Somit kann die Menge der übertragenen Bytes eindeutig einer Variante zugesprochen werden. Wiederum werden bei diesem Vergleich die Daten simuliert. So kann auch bei diesem Test die Meldungsmenge bestimmt und somit vergleichbar gemacht werden.

Bei dieser Vergleichsmessung wird in einem Abstand von etwa 200ms eine neue Meldung erzeugt und an den Client übertragen. Der Test wird pro Variante wiederum eine Stunde lang ausgeführt, womit etwa 18'000 Meldungen je Variante übertragen werden.

Der SOS-Client wird so konfiguriert, dass dieser pro Sekunde 5 Anfragen an der Server sendet. So sollte im Durchschnitt nach jeder Anfrage an den SOS ein neuer Datensatz empfangen werden. Durch ein längeres Abfrageintervall könnte die Datenmenge bei der SOS-Variante reduziert werden. Bei dieser Vergleichsmessung soll jedoch analysiert werden, wie gross die Datenmenge ist, falls jede Meldung umgehend zum Client gesendet wird und somit die Latenzzeit möglichst gering ist.

7.3.2 Datenanalyse

Vor dem Ausführen einer Messung wird auf dem Netzwerkswitch die Statistik der übertragenen Datenmengen zurückgesetzt. Die Datenmenge in der Statistik kann pro

angeschlossenem Gerät ausgelesen werden. Es kann sowohl die empfangene wie auch die gesendete Datenmenge bestimmt werden. Aus den Ergebnissen kann die Datenmenge ausgelesen werden, die der Webclient je Variante für den gesamten Test benötigte. Zudem kann die Datenmenge des Gesamtsystems bestimmt werden. Somit kann sowohl aufgezeigt werden, bei welcher Client Variante das Datenvolumen vom/zum Client weniger gross ist und wie gross das Datenvolumen des Gesamtsystems ist.

7.3.3 Ergebnisse

Gesamtsystem

Nach Durchführung der Vergleichsmessungen fiel beim Zusammenfügen der Resultate auf, dass die empfangenen Daten von den gesendeten Daten bei beiden Varianten abweichen. Diese Abweichungen zwischen empfangenen und gesendeten Daten kann darauf zurückgeführt werden, dass weitere Netzwerk-Komponenten (wie z.B. der NTP-Server) und weitere Dienste, die auf den Netzwerk-Komponenten laufen, das Netzwerk mit Daten belastet. Die Messergebnisse der SOS-Variante können der folgenden Tabelle entnommen werden:

	Empf. Daten	**Ges. Daten**
Integrationssystem	12'867 kB	71'781 kB
SOS-Server	100'374 kB	32'875 kB
Webclient	19'149 kB	28'424 kB

Tabelle 39: Ergebnisse Übertragunsmenge SOS

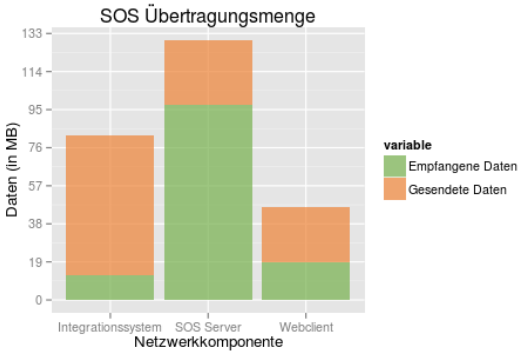

Abbildung 31: SOS Übertragungsmenge

Betrachtet man die Balken in Abbildung 31 ist erkennbar, dass die gesamte Datenmenge vom SOS-Server deutlich grösser ist als jene vom Integrationssystem oder dem Webclient, da der SOS-Server bei dieser Variante doppelt belastet wird.

Zudem ist die empfangene Datenmenge beim SOS-Server bedeutend grösser als die gesendete Menge. Dies kann darauf zurückgeführt werden, dass sowohl das Integrationssystem wie auch der Webclient Anfragen im XML-Format an den SOS-Server senden. Da die Antwort zum Webclient im JSON-Format ist, ist die empfangene Datenmenge beim Webclient geringer als die gesendete Datenmenge.

Die Messergebnisse der Websocket-Variante weichen jenen der SOS-Variante deutlich ab:

	Empf. Daten	Ges. Daten
SOS-Server	71 kB	14 kB
Integrationssystem	1'274 kB	4'891 kB
Webclient	5'245 kB	1'322 kB

Tabelle 40: Ergebnisse Übertragungsmenge Websocket

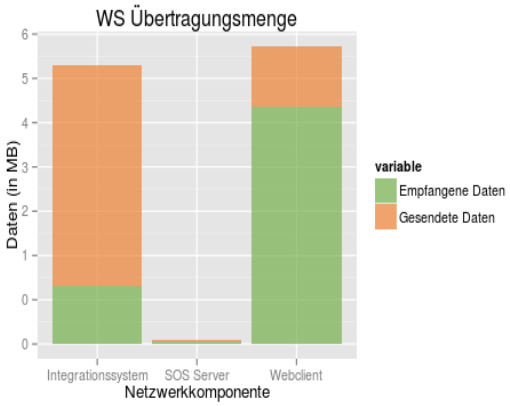

Abbildung 32: Websocket Übertragungsmenge

Da die Kommunikation zwischen Integrationssystem und Webclient direkt erfolgt, ist die Datenmenge des SOS-Servers verständlicherweise beinahe bei 0. Die gesendete wie auch die empfangene Datenmenge, die zwischen Integrationssystem und Webclient hin und her gesendet werden, ist in etwa gleich gross.

Um die Gesamtmenge der beiden Daten vergleichen zu können, werden nur die gesendeten Daten berücksichtigt. Ansonsten würden die übertragenen Daten doppelt berücksichtigt. In Abbildung 33 werden die gesendeten Daten der beiden Varianten gegenüberübergestellt. Bei der SOS-Variante wurden insgesamt ~130 MB versendet. Bei der Websocket-Variante wurden hingegen nur etwas mehr als ~6 MB übertragen. Die SOS-Variante hat in diesem Vergleichstest somit rund 21-mal so viel Daten versendet wie die Websocket-Variante.

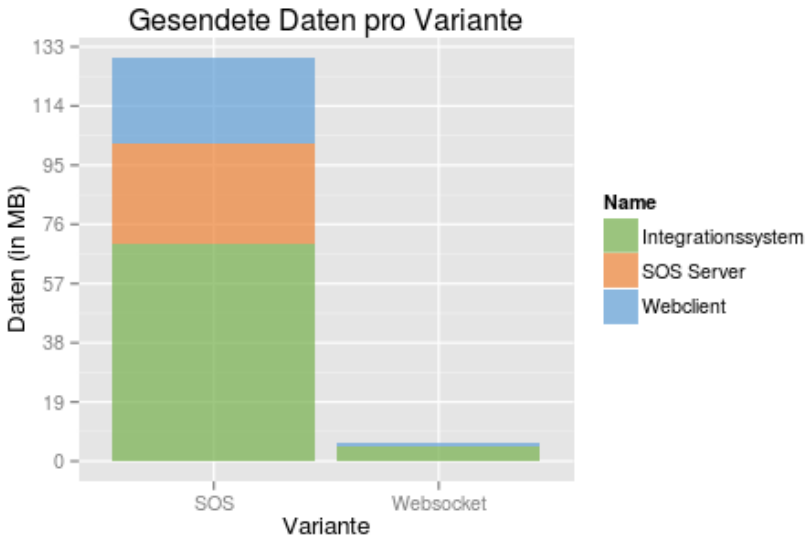

Abbildung 33: Gesendete Datenmenge pro Variante

Webclient

In Abbildung 34 wird der Webclient isoliert betrachtet. Der Unterschied von der SOS-Variante (~46.5 MB) zur Websocket-Variante (~6.4 MB) ist weniger gross als der Unterschied des Gesamtsystems. Dennoch benötigt ein einzelner Webclient bei der SOS-Variante gut 7-mal mehr Bandbreite als die Websocket-Variante. Bei der Websocket-Variante ist insbesondere die Menge der gesendeten Daten sehr gering.

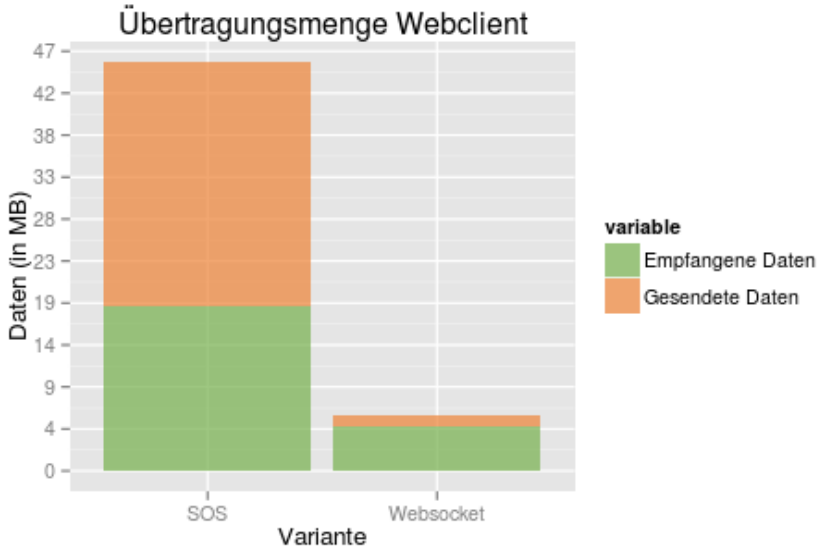

Abbildung 34: Übertragunsmenge Webclient

7.4 Meldungsdurchsatz

Mit diesem Versuch wird aufgezeigt, wie gross der maximale Durchsatz pro Sekunde der beiden Varianten innerhalb dieses Testsystems ist. Es soll belegt werden, dass durch die direkte Kommunikation mit dem Webclient der maximale Meldungsdurchsatz bei der Websocket-Variante erheblich grösser ist als jener der SOS-Variante.

7.4.1 Ablauf

Für diesen Versuch werden wiederum beim Integrationssystem ADS-B Meldungen erzeugt. Durch die Simulation der Daten kann die Anzahl der Meldungen pro Sekunde definiert werden. Der Versuch wird für beide Varianten einmal ausgeführt. Bei dieser Vergleichsmessung wird während einer Dauer von 30 Sekunden, pro Sekunde nur eine Meldung vom Integrationssystem erzeugt. Mit jeden weiteren 30 Sekunden wird die Anzahl der Meldungen pro Sekunde um eine erhöht. Dadurch wird die Menge der Meldungen konstant vergrössert. Der Versuch dauert 50 Minuten. Somit werden in der letzten Minute 100 Meldungen pro Sekunde vom Integrationssystem erzeugt. Auf dem Clientsystem wird darauf die Anzahl empfangener Meldungen pro Minute gemessen. Kann der Client die Menge an Meldungen nicht mehr verarbeiten, sinkt oder stagniert die empfangene Meldungsmenge pro Minute. Zudem wird die durchschnittliche Latenzzeit in den Zeiträumen betrachtet.

Der SOS-Client wird so konfiguriert, dass dieser theoretisch alle Meldungen umgehend empfängt. Das heisst, dass keine Verzögerung zwischen einer Antwort vom Server und einer neuen Anfrage gemacht wird. Somit wird nach dem Empfang einer Antwort vom SOS-Server umgehend eine neue Anfrage initiiert.

7.4.2 Datenanalyse

Die an den Webclient übertragenen Daten werden in einem Cache gesammelt. Nach jeweils 10 Sekunden wird der Cache ausgewertet. Für jede Auswertung werden alle empfangenen Features der aktuellen Periode berücksichtigt. Von den Features wird die durchschnittliche Latenzzeit, die Menge der empfangenen Meldungen und der Start- respektive der Endzeitpunkt der Periode in ein Logfile geschrieben. Mit Hilfe dieses Logfiles kann eingesehen werden, zu welchem Zeitpunkt der Meldungsdurchsatz unter Berücksichtigung einer möglichst geringen Latenzzeit am grössten war.

7.4.3 Ergebnisse

Wie dem Liniendiagramm in Abbildung 35 entnommen werden kann, skaliert die SOS-Variante bis 26 Meldungen pro Sekunde linear. Die Latenzzeit in diesem Zeitbereich verweilt in der gleichen Grössenordnung. Ab 26 Meldungen pro Sekunde bleibt die Anzahl verarbeiteter Meldungen in etwa konstant, die Latenzzeit wird ab diesem Zeitpunkt stetig

grösser. Bei 52 Meldungen pro Sekunde wird die Latenzzeit sprunghaft grösser. Gleichzeitig steigt die Anzahl verarbeiteter Meldungen leicht an. Ab diesem Zeitpunkt bleibt die SOS-Variante bei etwa 30 Meldungen pro Sekunde, wohingegen die Latenzzeit stetig grösser wird.

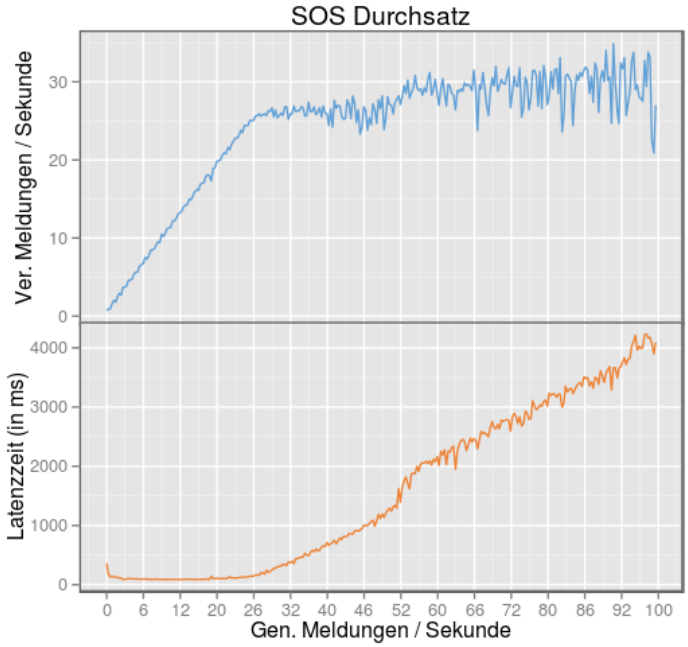

Abbildung 35: SOS Durchsatz

Wie dem Liniendiagramm der Websocket-Variante (vgl. Abbildung 36) entnommen werden kann, steigt die Anzahl der verarbeiteten Meldungen pro Sekunden bis zum Ende des Tests stetig an. Somit werden am Schluss bis zu 100 Meldungen pro Sekunde verarbeitet. Die Latenzzeit ist auch bei einem hohen Meldungsdurchsatz noch immer kleiner als 20 Millisekunden, was für eine sehr effiziente Übertragung spricht.

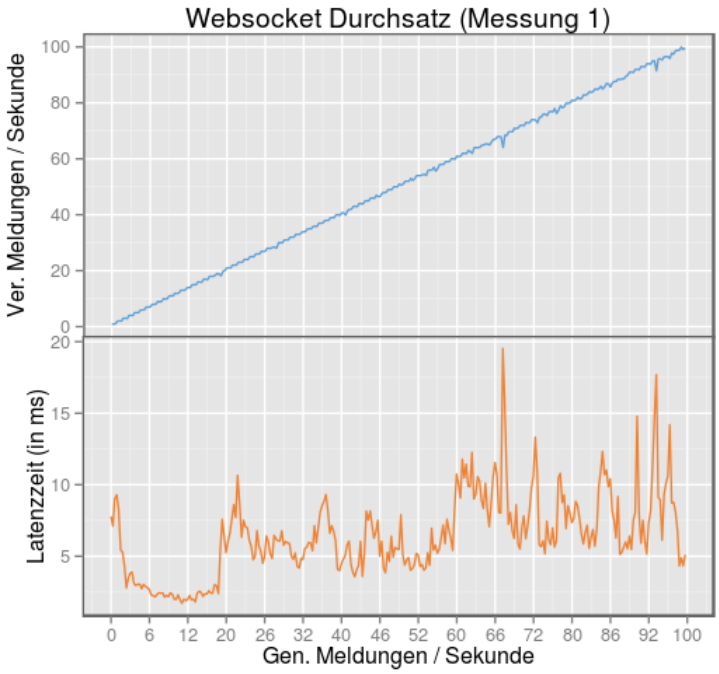

Abbildung 36: Websocket Durchsatz (Messung 1)

Da der maximale Durchsatz der Websocket-Variante mit dieser Messung nicht abschliessend bestimmt werden konnte, wurde der Test mit der vierfachen Meldungsmenge wiederholt. So startet diese Vergleichsmessung mit 4 Meldungen pro Sekunde. Nach Ablauf der ersten 30 Sekunden wird die Meldungsmenge auf 8 Meldungen pro Sekunde erhöht. Womit gegen Ende der Messung 400 Meldungen pro Sekunde gesendet werden. Das Ergebnis dieser Messung kann der Abbildung 37 entnommen werden. Die Websocket-Variante konnte bis 196 Meldungen pro Sekunde verarbeiten. Danach wurden auf dem Webclient keine Meldungen mehr empfangen. Eine Analyse des Problems hat ergeben, dass auf dem Integrationssystem die Meldungen bei diesem hohen Durchsatz nicht genügend schnell in das Websocket geschrieben werden konnten, so dass sich diese aufgestaut haben, bis der Buffer überlaufen ist. Die Websocket-Variante konnte somit bis zu 196 Meldungen pro Sekunde vom Integrationssystem bis zur Anzeige auf dem Webclient verarbeiten.

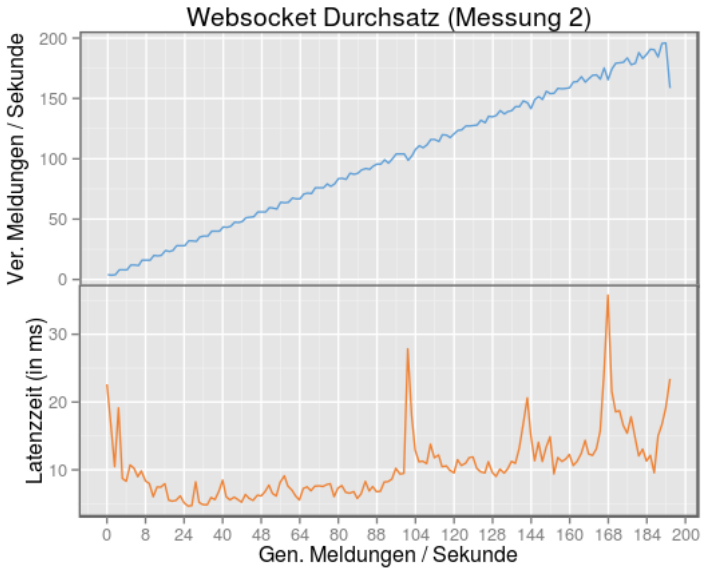

Abbildung 37: Websocket Durchsatz (Messung 2)

Die Websocket-Variante konnte somit bei diesem Aufbau des Systems gut 7.5-mal mehr Meldungen pro Sekunde verarbeiten als die SOS-Variante.

8. Schlussfolgerungen

In diesem Kapitel werden die initial gestellten Forschungsfragen beantwortet. Hierzu werden die wichtigsten Erkenntnisse und Ergebnisse dieser Arbeit zusammengetragen. Durch diese wird auch beschrieben, wie das Gesamtsystem in anderen Bereichen eingesetzt werden könnte. Die Arbeit endet mit einem Fazit und einem Ausblick, in welchem insbesondere Erweiterungen des Konzeptes beschrieben werden.

8.1 Beantwortung der Forschungsfragen

Im Rahmen dieser Masterarbeit wurde ein System erstellt, mit welchem Flugzeug-Positionsdaten sowohl über Websocket wie auch über SOS zu einem Webclient übertragen werden können. Wie den Kapiteln 4 (Umsetzung Integrationssystem) und 5 (Umsetzung Webclient) entnommen werden kann, basieren beide Varianten auf derselben Codebasis und nur die Übertragung der Daten wurde variantenspezifisch implementiert. Somit konnte ein System geschaffen werden, welches die Datenübertragung abstrahiert und damit einen Vergleich der beiden Varianten ermöglicht. Mit Hilfe dieses entwickelten Systems können die initial gestellten Fragen beantwortet werden:

Können die Sensordaten zuverlässig über beide Varianten verteilt und auf einem Webclient dargestellt werden?

Wie den Messergebnissen der Datenqualität entnommen werden kann, konnten die beiden Varianten in einem parallelen Betrieb ADS-B Livedaten bei einer durchschnittlichen Meldungsmenge von ~16.5 Meldungen pro Sekunde während 90 Minuten vom Integrationssystem mit Hilfe der jeweils verwendeten Technologie auf einen Webclient übertragen und dargestellt werden. Die mit dem ADS-B-Empfänger gemessenen Werte wurden sowohl in der Websocket- wie auch mit der SOS-Variante zu 100% korrekt übertragen (vgl. Unterkapitel 7.1 Übertragungsqualität). Es kann somit festgestellt werden, dass sowohl die Websocket- wie auch die SOS-Variante zur Übertragung von Daten, die von einem ADS-B Empfänger stammen, an eine Webanwendung zuverlässig eingesetzt werden können.

Wie wirkt sich der Einsatz von Websockets für die Kommunikation zwischen Server und Webclient auf die Echtzeitfähigkeit gegenüber der Kommunikation über einen SOS aus?

Um diese Fragestellung beantworten zu können, wurde eine Vergleichsmessung durchgeführt, mit welcher die Latenzzeiten der beiden Varianten unter geringer Auslastung untersucht wurden (vgl. Unterkapitel 7.2 Minimale Latenzzeit). Die Vergleichsmessung hat ergeben, dass bei der Websocket-Variante die Latenzzeit mit durchschnittlich ~4ms erheblich unter dem durchschnittlichen Wert von ~91ms der SOS-Variante liegt. Die kürzeste Latenzzeit, welche mit der Websocket-Variante erreicht wurde, lag mit 1ms sehr nahe bei der Echtzeit, wohingegen der kürzeste Wert der SOS-Variante mit 41ms deutlich über dem

Durchschnittswert der Websocket-Variante liegt. Die Websocket-Variante ermöglicht es somit dem Anwender, durch eine minimale Latenzzeit die Datenübertragung näher zur Echtzeit zu bringen als dies mit dem SOS möglich ist.

Gibt es erhebliche Unterschiede bei der übertragenen Datenmenge beim Einsatz von Websockets gegenüber dem Einsatz eines SOS?

Diese Frage muss differenziert betrachtet werden. Da bei der SOS-Variante die Kommunikation über einen SOS erfolgt, werden die Daten erst zu diesem Server übertragen von welchem sie vom Client abgefragt werden können. In einer Vergleichsmessung wird die übertragene Datenmenge vom Gesamtsystem und vom Client deshalb unter Berücksichtigung beider Aspekte betrachtet (vgl. Unterkapitel 7.3 Übertragungsmenge). Unter Betrachtung des Gesamtsystems hat die SOS-Variante gut 21-mal mehr Daten für die gleiche Anzahl der Meldungen übertragen als die Websocket-Variante. Unter gesonderter Betrachtung des Webclients wurde ein weniger grosser aber dennoch erheblicher Unterschied der übertragenen Datenmenge festgestellt. Der Webclient der SOS-Variante hat ~7-mal mehr Daten übertragen als der Webclient der Websocket-Variante.

Bei der Websocket-Variante wurden in der Gesamtheit unter Berücksichtigung einer gleichbleibenden Latenzzeit bedeutend weniger Daten übermittelt als bei der SOS-Variante.

Da die Websocket-Variante weniger Daten mit dem Server austauscht, ist die benötigte Bandbreite auch entsprechend kleiner. Somit wird die Verbindung zum Server weniger in Anspruch genommen, womit theoretisch bei der Websocket-Variante mit einem Server mit der gleichen Bandbreite ~7-mal mehr Clients bedient werden können als mit der SOS-Variante.

Wie gross ist der maximale Durchsatz von ADS-B Meldungen mit den einzelnen Varianten?

Mit Hilfe einer Vergleichsmessung (vgl. Unterkapitel 7.4 Meldungsdurchsatz) wurde evaluiert, wie viele Meldungen mittels der beiden Varianten verarbeitet werden können. Die SOS-Variante konnte bis 26 Meldungen pro Sekunde gut skalieren. Ein höherer Meldungsdurchsatz ging auf Kosten der Latenzzeit. Die Websocket Lösung konnte bis 196 Meldungen pro Sekunde gut skalieren. Ein höherer Meldungsdurchsatz konnte nicht erreicht werden. Mit der Websocket-Variante war der Meldungsdurchsatz unter Berücksichtigung einer kleinen Latenzzeit somit entscheidend grösser.

Die Websocket-Variante hat in den Vergleichsmessungen, in welchen Performance und Ressourcennutzung evaluiert wurden, innerhalb des Testsystems deutlich bessere Ergebnisse erzielt als die SOS-Variante. Für den Einsatz in einem Nah-Echtzeitsystem eignet sich somit die Websocket-Variante besser für dieses Einsatzgebiet. Der SOS bietet jedoch Funktionen an, welche in dieser Masterarbeit nur am Rande diskutiert wurden, da diese für diesen Vergleich nur von geringer Bedeutung sind. In anderen Einsatzszenarien könnten diese gegenüber der Websocket-Variante einen Vorteil bieten. Im Speziellen sind die Persistierung und

Filtermöglichkeiten hervorzuheben. Im SOS werden die Messwerte gespeichert und können für die Darstellung von historischen Daten wieder abgefragt werden. Die Abfragen an den SOS können komplexe Filterkriterien beinhalten und ermöglichen es, nur die Daten anzufordern, welche umgehend benötigt werden.

8.2 Weitere Einsatzszenarien

Die Übertragung vom Serversystem zum Webclient ist sehr generell gehalten und kann sowohl über Websocket oder SOS erfolgen. In dieser Masterarbeit wurde im Speziellen darauf eingegangen, wie das System am Beispiel von ADS-B Flugzeug-Positionsdaten betrieben und konfiguriert werden kann. Durch den generischen Aufbau des Systems können die zu übermittelnden Daten durch Konfigurationen angepasst werden, so dass Sensor-Messwerte auch von anderen Sensoren übermittelt werden können. Lediglich der Datenempfang auf dem Serversystem (vgl. Unterkapitel 4.3 Datenempfangs-Route) und das Rendering auf der Weboberfläche (vgl. Unterkapitel 5.6 Style Service) enthalten Code, der spezifisch für ADS-B respektive für Flugzeug-Positionsdaten ist.

Nachfolgend werden drei mögliche weitere Einsatzszenarien kurz vorgestellt, wobei die Liste nicht abschliessend ist und noch erweitert werden könnte. Es wurden möglichst unterschiedliche Beispiele gewählt. Die Übertragung der Daten könnte bei diesen Einsatzszenarien über beide Varianten erfolgen. Da jedoch die Websocket-Variante bedeutend performanter und Ressourcen schonender ist, wäre diese Variante bei den folgenden Einsatzszenarien vorzuziehen.

Flotten-Tracking

Ähnlich der vorgestellten ADS-B Variante könnte dieses System für das Flotten Tracking verwendet werden. Der Unterschied wäre in diesem Fall, dass die Autos, Kraftfahrzeuge, Schiffe, Flugzeuge, Fahrräder oder Fahrzeuge des öffentlichen Verkehrs den aktuellen Zustand und die Position selbst zum Integrationsserver übertragen, welcher die Anfragen umgehend verarbeitet und über die Routen weiterleitet. Über die Weboberfläche könnten somit der Zustand, die aktuelle Position und weitere Informationen der Fahrzeuge umgehend angezeigt werden.

Spatial Social Media

Ein weiteres Einsatzgebiet könnte sein, das Integrationssystem an eine öffentliche Schnittstelle von Sozialen Medien anzubinden, welche ortsbezogene Abfragen, wie beispielsweise die *Twitter geo API*[34], ermöglichen. Das Integrationssystem könnte auf diese Programmierschnittstellen zugreifen und die Ergebnisse an die angeschlossenen Clients übertragen.

34 Twitter Geo API - https://dev.twitter.com/rest/reference/get/geo/search [letzter Zugriff 30.06.2015]

Statische Sensoren

Das System ist so konzipiert, dass nicht nur ortsbezogene Daten übertragen werden können, sondern auch Messwerte, die nicht mit einem Ort verknüpft werden können oder von einem statischen Sensor stammen. Die Messwerte würden in der Websocket-Variante normal als JSON-Feature ohne g*eometry*-Attribut zum Client übertragen. In der SOS-Variante würden die Messwerte als normale Observationen in den SOS eingefügt werden. Für die Visualisierung müsste bei diesem Einsatzszenario jedoch eine neue View geschrieben werden, um die Daten in Nah-Echtzeit zu visualisieren.

8.3 Fazit und Ausblick

Es konnte ein System entwickelt werden, welches sehr flexibel für die Übermittlung von Messwerten zu einer Webanwendung verwendet werden kann. Wie den Ergebnissen der Vergleichsmessungen entnommen werden kann, ist sowohl die SOS- wie auch die Websocket-Variante fähig, die ADS-B Flugzeug-Positionsdaten in Nah-Echtzeit zuverlässig zu übertragen. Die Websocket-Variante weist jedoch bedeutende Performance-Vorteile auf. So ist die Latenzzeit kleiner, die übertragene Datenmenge geringer und der maximale Durchsatz grösser. Der Einsatz von Websocket eignet sich somit noch besser für die Übertragung von Nah-Echtzeitdaten. Durch die bidirektionale Kommunikation eignen sich Websockets nicht nur bei hohen Datenraten sondern auch bei geringer Meldungsmenge, da nur Meldungen versandt werden wenn tatsächlich Daten übertragen werden sollen. Bei der SOS-Variante muss hingegen ständig nach Daten *gepollt* werden. Somit wird der Einsatz einer Websocket-Lösung für die Übertragung von Nah-Echtzeitdaten zu einer Browseranwendung gegenüber einer SOS-Lösung empfohlen.

Das System könnte, um noch flexibler eingesetzt werden zu können, mit folgenden Erweiterungen ergänzt werden:

Websocket-Filter

Die Websocket-Variante könnte insofern erweitert werden, dass ein Websocket Client auch Anfragen an den Server senden kann, so dass beispielsweise Meldungen direkt auf dem Server für diesen Client ausgefiltert werden. So könnten nur Meldungen einer Region oder nur Features zurückgegeben werden, deren Attribute der Anfrage entsprechen (z.B. Flughöhe > 10000Fuss). Das Integrationssystem würde ab dem Zeitpunkt, wo die Anfrage beim Integrationssystem eingetroffen ist, nur Meldungen durch das Websocket zum Client übertragen, welche dem Filter entsprechen. Ein Prove-Of-Concept dieser Erweiterung wurde im Rahmen dieser Arbeit bereits erfolgreich umgesetzt (vgl. Unterkapitel 4.5.2 Implementierung). Eine spezifizierte Implementierung dieser Variante steht jedoch noch aus.

Hybride Lösung

Es ist auch denkbar, dass die Websocket- und die SOS-Variante gemeinsam in einer Lösung

eingesetzt werden. Da in der SOS-Variante die Daten in einer Datenbank persistiert werden, könnten nicht nur Nah-Echtzeitdaten sondern auch historische Daten auf einem Client dargestellt werden. Hierbei würden die beiden Varianten parallel betrieben. Ein Client könnte für die Echtzeitwiedergabe die Websocket-Variante verwenden. Durch Umschalten auf dem Client in den Wiedergabemodus könnte der Zeitpunkt angegeben werden, von welchem die Daten dargestellt werden sollen. Wurde dieser angegeben, führt der Webclient Anfragen an den SOS-Server aus, um die Daten vom angegebenen Zeitpunkt zu laden und diese zu visualisieren. Mit einer solchen hybriden Lösung könnten sich die beiden Varianten gut ergänzen und würden noch weitere Anwendungsmöglichkeiten generieren.

9. Literaturverzeichnis

52N Sensor Web Community - Sensor Observation Service [WWW Document], 2015. URL http://52north.org/communities/sensorweb/sos/ (accessed 6.18.15).

Blossom, E., 2004. GNU radio: tools for exploring the radio frequency spectrum. Linux J. 2004, 4.

Botts, M., Percivall, G., Reed, C., Davidson, J., 2008. OGC® sensor web enablement: Overview and high level architecture, in: GeoSensor Networks. Springer, pp. 175–190.

Bröring, A., Echterhoff, J., Jirka, S., Simonis, I., Everding, T., Stasch, C., Liang, S., Lemmens, R., 2011. New Generation Sensor Web Enablement. Sensors 11, 2652–2699. doi:10.3390/s110302652

Bröring, A., Stasch, C., Echterhoff, J., 2012. Ogc sensor observation service interface standard, version 2.0. OGC Doc.

Butler, H., Daly, M., Doyle, A., Gillies, S., Schaub, T., Schmidt, C., 2008. The GeoJSON format specification.

Chandy, K.M., Schulte, R., 2007. What is Event-driven Architecture (EDA) and Why Does it Matter? Retrieved May 20, 2009.

Command, A.M., 2012. Code of Federal Regulations, Title 14, Part 91, Automatic Dependent Surveillance: Broadcast (ADS–B) Out Performance Requirements to Support Air Traffic Control (ATC) Service, Final Rule, May 28, 2010. March 29.

Costin, A., Francillon, A., 2012. Ghost in the Air (Traffic): On insecurity of ADS-B protocol and practical attacks on ADS-B devices. Black Hat USA.

Cox, S., 2011a. OGC Abstract Specification - Observations and measurements.

Cox, S., 2011b. Observations and measurements-xml implementation. OGC Doc.

Crockford, D., 2006. The application/json media type for javascript object notation (json).

Deriu, U., 2011. Begriffsklärung: Pull- und Push-Prinzip | [WWW Document]. Begr. Pull-Push-Prinz. URL http://tirsus.com/blog/begriffsklarung-pull-und-push-prinzip/ (accessed 1.18.15).

Fette, I., Melnikov, A., 2011. The websocket protocol.

Fraternali, P., Rossi, G., Sánchez-Figueroa, F., 2010. Rich internet applications. Internet Comput. IEEE 14, 9–12.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1994. Design patterns: elements of reusable object-oriented software. Pearson Education.

Goodchild, M., 2011. Looking forward: Five thoughts on the future of GIS. ArcWatch.

Gossman, J., 2005. Introduction to Model/View/ViewModel pattern for building WPF apps. Viitattu 1410 2013 Saatavissa Httpblogs Msdn Combjohngossmanarchive20051008478683 Aspx.

Hickson, I., 2011. The websocket api. W3C Work. Draft WD-Websockets-20110929 Sept.

Hobona, G., Griffith, R., Botts, M., Bröring, A., Liang, S., Gates, C., Mandl, D., Howlett, E., Reed, C., McKenzie, D., Percivall, G., 2013. OGC® SWE Implementation Maturity Engineering Report.

Huston, G., 2015. Protocol Basics: The Network Time Protocol - The Internet Protocol Journal, Volume 15, No. 4. Internet Protoc. J. 15, 2–12.

Ibsen, C., Anstey, J., 2010. Camel in action. Manning Publications Co.

Jirka, S., Bredel, H., 2011. Building Tracking Applications with Sensor Web Technology. Geoinformatik 2011 Geochange.

Johnson, R., 2005. Introduction to the spring framework. TheServerSide Com 21, 22.

Krasner, G.E., Pope, S.T., 1988. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. J. Object Oriented Program. 1, 26–49.

Mankins, J.C., 1995. Technology readiness levels. White Pap. April 6.

Mills, D.L., 1991. Internet time synchronization: the network time protocol. Commun. IEEE Trans. On 39, 1482–1493.

Minar, I., 2012. MVC vs MVVM vs MVP. What a controversial topic that many developers can spend… [WWW Document]. URL https://plus.google.com/+IgorMinar/posts/DRUAkZmXjNV (accessed 6.19.15).

Na, A., Priest, M., 2007. Sensor observation service. Implement. Stand. OGC.

Nurseitov, N., Paulson, M., Reynolds, R., Izurieta, C., 2009. Comparison of JSON and XML Data Interchange Formats: A Case Study. Caine 2009, 157–162.

Pimentel, V., Nickerson, B.G., 2012. Communicating and displaying real-time data with WebSocket. Internet Comput. IEEE 16, 45–53.

Puranik, D.G., Feiock, D.C., Hill, J.H., 2013. Real-time monitoring using AJAX and WebSockets, in: Engineering of Computer Based Systems (ECBS), 2013 20th IEEE International Conference and Workshops on the. IEEE, pp. 110–118.

Richards, W.R., O'Brien, K., Miller, D.C., 2010. New air traffic surveillance technology. Boeing Aero Q. Quart. 2.

Schadow, G., McDonald, C.J., 2009. The Unified Code for Units of Measure. Regenstrief Inst. UCUM Organ. Indianap. USA.

Schäfer, M., Strohmeier, M., Lenders, V., Martinovic, I., Wilhelm, M., 2014. Bringing up OpenSky: a large-scale ADS-B sensor network for research, in: Proceedings of the 13th International Symposium on Information Processing in Sensor Networks. IEEE Press, pp. 83–94.

Strohmeier, M., Lenders, V., Martinovic, I., 2013. On the Security of the Automatic Dependent Surveillance-Broadcast Protocol. ArXiv Prepr. ArXiv13073664.

Tamayo, A., Viciano, P., Granell, C., Huerta, J., 2011. Empirical study of sensor observation services server instances, in: Advancing Geoinformation Science for a Changing World. Springer, pp. 185–209.

Wang, V., Salim, F., Moskovits, P., 2013. Introduction to HTML5 WebSocket, in: The Definitive Guide to HTML5 WebSocket. Springer, pp. 1–12.

A Anhang

A.1 InsertSensor

```
<?xml version="1.0" encoding="UTF-8"?>
<swes:InsertSensor xmlns:swes="http://www.opengis.net/swes/2.0" xmlns:sos="http://www.opengis.net/sos/2.0"
xmlns:swe="http://www.opengis.net/swe/1.0.1" xmlns:sml="http://www.opengis.net/sensorML/1.0.1"
xmlns:gml="http://www.opengis.net/gml" xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" service="SOS" version="2.0.0"
xsi:schemaLocation="http://www.opengis.net/sos/2.0 http://schemas.opengis.net/sos/2.0/sosInsertSensor.xsd
http://www.opengis.net/swes/2.0 http://schemas.opengis.net/swes/2.0/swes.xsd">
  <swes:procedureDescriptionFormat>http://www.opengis.net/sensorML/1.0.1</swes:procedureDescriptionFormat>
  <swes:procedureDescription>
    <sml:SensorML version="1.0.1">
      <sml:member>
        <sml:System>
          <sml:identification>
            <sml:IdentifierList>
              <sml:identifier name="uniqueID">
                <sml:Term definition="urn:ogc:def:identifier:OGC:1.0:uniqueID">
                  <sml:value>http://stue.ch/sensorobservation/procedure/sbs-1</sml:value>
                </sml:Term>
              </sml:identifier>
              <sml:identifier name="longName">
                <sml:Term definition="urn:ogc:def:identifier:OGC:1.0:longName">
                  <sml:value>Flight Tracking Procedure contains ADS-B Data.
                   The ADS-B receiver generates sbs-1 socket output data.</sml:value>
                </sml:Term>
              </sml:identifier>
              <sml:identifier name="shortName">
                <sml:Term definition="urn:ogc:def:identifier:OGC:1.0:shortName">
                  <sml:value>Flight Tracking</sml:value>
                </sml:Term>
              </sml:identifier>
            </sml:IdentifierList>
          </sml:identification>
          <sml:capabilities name="offerings">
            <swe:SimpleDataRecord>
              <swe:field name="Offering for ADS-B Sensor">
                <swe:Text definition="urn:ogc:def:identifier:OGC:offeringID">
                  <swe:value>http://stue.ch/sensorobservation/offering/flighttracking</swe:value>
                </swe:Text>
              </swe:field>
            </swe:SimpleDataRecord>
          </sml:capabilities>
          <sml:position name="sensorPosition">
            <swe:Position referenceFrame="urn:ogc:def:crs:EPSG::4326">
              <swe:location>
                <swe:Vector gml:id="STATION_LOCATION">
                  <swe:coordinate name="easting">
                    <swe:Quantity axisID="x">
                      <swe:uom code="degree"/>
                      <swe:value>7.573065</swe:value>
                    </swe:Quantity>
                  </swe:coordinate>
                  <swe:coordinate name="northing">
                    <swe:Quantity axisID="y">
                      <swe:uom code="degree"/>
                      <swe:value>47.536427</swe:value>
                    </swe:Quantity>
                  </swe:coordinate>
                  <swe:coordinate name="altitude">
                    <swe:Quantity axisID="z">
                      <swe:uom code="m"/>
                      <swe:value>310</swe:value>
                    </swe:Quantity>
                  </swe:coordinate>
                </swe:Vector>
              </swe:location>
            </swe:Position>
          </sml:position>
          <sml:inputs>
            <sml:InputList>
              <sml:input name="airtraffic">
                <swe:ObservableProperty definition="http://stue.ch/sensorobservation/observableProperty/airtraffic"/>
              </sml:input>
            </sml:InputList>
          </sml:inputs>
          <sml:outputs>
            <sml:OutputList>
              <sml:output name="callsign">
                <swe:Quantity definition="http://stue.ch/sensorobservation/observableProperty/callsign">
                  <gml:metaDataProperty>
                    <offering>
                      <id>urn:x-ogc:def:property:OGC::Callsign</id>
                      <name>Callsign text measurement</name>
                    </offering>
                  </gml:metaDataProperty>
                </swe:Quantity>
              </sml:output>
```

```
            <sml:output name="speed">
              <swe:Quantity definition="http://stue.ch/sensorobservation/observableProperty/speed">
                <gml:metaDataProperty>
                  <offering>
                    <id>urn:x-ogc:def:property:OGC::Speed</id>
                    <name>speed measurements</name>
                  </offering>
                </gml:metaDataProperty>
                <swe:uom code="[kn_i]"/>
              </swe:Quantity>
            </sml:output>
            <sml:output name="altitude">
              <swe:Quantity definition="http://stue.ch/sensorobservation/observableProperty/altitude">
                <gml:metaDataProperty>
                  <offering>
                    <id>urn:x-ogc:def:property:OGC::Altitude</id>
                    <name>altitude measurements</name>
                  </offering>
                </gml:metaDataProperty>
                <swe:uom code="[ft_i]"/>
              </swe:Quantity>
            </sml:output>
            <sml:output name="heading">
              <swe:Quantity definition="http://stue.ch/sensorobservation/observableProperty/heading">
                <gml:metaDataProperty>
                  <offering>
                    <id>urn:x-ogc:def:property:OGC::Heading</id>
                    <name>heading measurements</name>
                  </offering>
                </gml:metaDataProperty>
                <swe:uom code="deg"/>
              </swe:Quantity>
            </sml:output>
            <sml:output name="position">
              <swe:Quantity definition="http://stue.ch/sensorobservation/observableProperty/position">
                <gml:metaDataProperty>
                  <offering>
                    <id>urn:x-ogc:def:property:OGC::Position</id>
                    <name>position measurements</name>
                  </offering>
                </gml:metaDataProperty>
              </swe:Quantity>
            </sml:output>
          </sml:OutputList>
        </sml:outputs>
      </sml:System>
    </sml:member>
  </sml:SensorML>
 </swes:procedureDescription>
 <!-- multiple values possible -->
 <swes:observableProperty>http://stue.ch/sensorobservation/observableProperty/callsign</swes:observableProperty>
 <swes:observableProperty>http://stue.ch/sensorobservation/observableProperty/speed</swes:observableProperty>
 <swes:observableProperty>http://stue.ch/sensorobservation/observableProperty/altitude</swes:observableProperty>
 <swes:observableProperty>http://stue.ch/sensorobservation/observableProperty/heading</swes:observableProperty>
 <swes:observableProperty>http://stue.ch/sensorobservation/observableProperty/position</swes:observableProperty>
 <swes:metadata>
  <sos:SosInsertionMetadata>
   <sos:observationType>http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_Measurement</sos:observationType>
   <sos:observationType>http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_CategoryObservation</sos:observationType>
   <sos:observationType>http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_CountObservation</sos:observationType>
   <sos:observationType>http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_TextObservation</sos:observationType>
   <sos:observationType>http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_TruthObservation</sos:observationType>
   <sos:observationType>http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_GeometryObservation</sos:observationType>
   <!-- multiple values possible -->
   <sos:featureOfInterestType>http://www.opengis.net/def/samplingFeatureType/OGC-
OM/2.0/SF_SamplingPoint</sos:featureOfInterestType>
  </sos:SosInsertionMetadata>
 </swes:metadata>
</swes:InsertSensor>
```

A.2 Bean-Konfiguration SOS-Variante

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:camel="http://camel.apache.org/schema/spring"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:util="http://www.springframework.org/schema/util"
  xsi:schemaLocation="
       http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd
       http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd
       http://www.springframework.org/schema/util  http://www.springframework.org/schema/util/spring-util.xsd">


  <!-- Insert Observation configuration -->
  <bean id="featureOfInterestPrefix" class="java.lang.String">
    <constructor-arg value="http://stue.ch/sensorobservation/foi/aircraft/" />
  </bean>

  <bean id="callsignObservationConfiguration"
    class="ch.trackdata.sbs1route.converter.insertobservation.TextPropertyConfiguration">
    <property name="observationName" value="callsignObservation" />
    <property name="procedure" value="http://stue.ch/sensorobservation/procedure/sbs-1" />
    <property name="observedProperty" value="http://stue.ch/sensorobservation/observableProperty/callsign" />
    <property name="featureOfInterestPrefix" ref="featureOfInterestPrefix" />
    <property name="featureIdentPropertyName">
      <util:constant static-field="ch.trackdata.sbs1route.message.TrackPositionMessage.HEX_PROPERTY_NAME" />
    </property>
    <property name="featureTitlePropertyName">
      <util:constant static-field="ch.trackdata.sbs1route.message.TrackPositionMessage.HEX_PROPERTY_NAME" />
    </property>
    <property name="featureTextPropertyName">
      <util:constant static-field="ch.trackdata.sbs1route.message.TrackPositionMessage.CALLSIGN_PROPERTY_NAME" />
    </property>
    <property name="phenomenonTimePropertyName">
       <util:constant static-
      field="ch.trackdata.sbs1route.message.TrackPositionMessage.DATE_TIME_MESSAGE_GENERATED_NAME" />
    </property>
    <property name="createNullValueMessages" value="false" />
  </bean>

  <bean id="speedObservationConfiguration"
    class="ch.trackdata.sbs1route.converter.insertobservation.MeasurementPropertyConfiguration">
    <property name="observationName" value="speedObservation" />
    <property name="procedure"
      value="http://stue.ch/sensorobservation/procedure/sbs-1" />
    <property name="observedProperty"
      value="http://stue.ch/sensorobservation/observableProperty/speed" />
    <property name="featureOfInterestPrefix" ref="featureOfInterestPrefix" />
    <property name="featureIdentPropertyName">
      <util:constant
        static-field="ch.trackdata.sbs1route.message.TrackPositionMessage.HEX_PROPERTY_NAME" />
    </property>
    <property name="featureTitlePropertyName">
      <util:constant
        static-field="ch.trackdata.sbs1route.message.TrackPositionMessage.HEX_PROPERTY_NAME" />
    </property>
    <property name="measureUom" value="[kn_i]" />
    <property name="featureMeasurementPropertyName">
      <util:constant
        static-field="ch.trackdata.sbs1route.message.TrackPositionMessage.GROUND_SPEED_NAME" />
    </property>
    <property name="featureMeasurementPropertyClazz" value="java.lang.Integer" />
    <property name="phenomenonTimePropertyName">
       <util:constant static-field="ch.trackdata.sbs1route.message.TrackPositionMessage.DATE_TIME_MESSAGE_GENERATED_NAME" />
    </property>
    <property name="createNullValueMessages" value="false" />
  </bean>

  <bean id="altitudeObservationConfiguration"
    class="ch.trackdata.sbs1route.converter.insertobservation.MeasurementPropertyConfiguration">
    <property name="observationName" value="altitudeObservation" />
    <property name="procedure"
      value="http://stue.ch/sensorobservation/procedure/sbs-1" />
    <property name="observedProperty"
      value="http://stue.ch/sensorobservation/observableProperty/altitude" />
    <property name="featureOfInterestPrefix" ref="featureOfInterestPrefix" />
    <property name="featureIdentPropertyName">
      <util:constant
        static-field="ch.trackdata.sbs1route.message.TrackPositionMessage.HEX_PROPERTY_NAME" />
    </property>
    <property name="featureTitlePropertyName">
      <util:constant
        static-field="ch.trackdata.sbs1route.message.TrackPositionMessage.HEX_PROPERTY_NAME" />
    </property>
    <property name="measureUom" value="[ft_i]" />
    <property name="featureMeasurementPropertyName">
      <util:constant
        static-field="ch.trackdata.sbs1route.message.TrackPositionMessage.ALTITUDE_PROPERTY_NAME" />
    </property>
    <property name="featureMeasurementPropertyClazz" value="java.lang.Integer" />
    <property name="phenomenonTimePropertyName">
       <util:constant static-field="ch.trackdata.sbs1route.message.TrackPositionMessage.DATE_TIME_MESSAGE_GENERATED_NAME" />
    </property>
```

```
    <property name="createNullValueMessages" value="false" />
  </bean>

  <bean id="headingObservationConfiguration"
    class="ch.trackdata.sbs1route.converter.insertobservation.MeasurementPropertyConfiguration">
    <property name="observationName" value="headingObservation" />
    <property name="procedure"
      value="http://stue.ch/sensorobservation/procedure/sbs-1" />
    <property name="observedProperty"
      value="http://stue.ch/sensorobservation/observableProperty/heading" />
    <property name="featureOfInterestPrefix" ref="featureOfInterestPrefix" />
    <property name="featureIdentPropertyName">
      <util:constant
        static-field="ch.trackdata.sbs1route.message.TrackPositionMessage.HEX_PROPERTY_NAME" />
    </property>
    <property name="featureTitlePropertyName">
      <util:constant
        static-field="ch.trackdata.sbs1route.message.TrackPositionMessage.HEX_PROPERTY_NAME" />
    </property>
    <property name="measureUom" value="deg" />
    <property name="featureMeasurementPropertyName">
      <util:constant
        static-field="ch.trackdata.sbs1route.message.TrackPositionMessage.HEADING_PROPERTY_NAME" />
    </property>
    <property name="featureMeasurementPropertyClazz" value="java.lang.Integer" />
    <property name="phenomenonTimePropertyName">
       <util:constant static-field="ch.trackdata.sbs1route.message.TrackPositionMessage.DATE_TIME_MESSAGE_GENERATED_NAME" />
    </property>
    <property name="createNullValueMessages" value="false" />
  </bean>

  <bean id="positionObservationConfiguration"
    class="ch.trackdata.sbs1route.converter.insertobservation.PointGeometryPropertyConfiguration">
    <property name="observationName" value="positionObservation" />
    <property name="procedure"
      value="http://stue.ch/sensorobservation/procedure/sbs-1" />
    <property name="observedProperty"
      value="http://stue.ch/sensorobservation/observableProperty/position" />
    <property name="featureOfInterestPrefix" ref="featureOfInterestPrefix" />
    <property name="featureIdentPropertyName">
      <util:constant
        static-field="ch.trackdata.sbs1route.message.TrackPositionMessage.HEX_PROPERTY_NAME" />
    </property>
    <property name="featureTitlePropertyName">
      <util:constant
        static-field="ch.trackdata.sbs1route.message.TrackPositionMessage.HEX_PROPERTY_NAME" />
    </property>
    <property name="useFeatureGeometry" value="true" />
    <property name="srsName" value="http://www.opengis.net/def/crs/EPSG/0/4326" />
    <property name="pointId" value="aircraftPosition" />
    <property name="phenomenonTimePropertyName">
       <util:constant static-field="ch.trackdata.sbs1route.message.TrackPositionMessage.DATE_TIME_MESSAGE_GENERATED_NAME" />
    </property>
    <property name="createNullValueMessages" value="false" />
  </bean>

  <bean id="insertObservationConfiguration"
    class="ch.trackdata.sbs1route.converter.insertobservation.InsertObservationSOSV2Configuration">
    <property name="offerings">
      <list>
        <value>http://stue.ch/sensorobservation/offering/flighttracking</value>
      </list>
    </property>
    <property name="observedProperties">
      <list>
        <ref bean="callsignObservationConfiguration" />
        <ref bean="speedObservationConfiguration" />
        <ref bean="altitudeObservationConfiguration" />
        <ref bean="headingObservationConfiguration" />
        <ref bean="positionObservationConfiguration" />
      </list>
    </property>
  </bean>
</beans>
```

A.3 Integrationssystem Konfigurationsdatei

```
localaddress=192.168.1.136

websocket.enabled=true
websocket.host=192.168.1.136
websocket.port=8443
websocket.path=/clientTrackData

sos.enabled=false
sos.host=192.168.1.136
sos.port=8080
sos.path=/52n-sos-webapp/sos/pox

sbs1.enabled=false
sbs1.host=192.168.1.136
sbs1.port=30003
aggregator.cleanupInterval=30000

generator.enabled=true
generator.amountOfTracks=10
generator.updateInterval=250
```

A.4 Client Konfigurationsdatei


```
{
	"ENV": "production",
	"appConfig":
	{
		"mapPages":
		[{
				"id": "websocketMap",
				"displayName": "Websocket Map",
				"url": "/websocket-map",
				"config": "websocketConfig",
				"dataService": "WebsocketGeoJSONService",
				"styleService": "AircraftStyleService"
			},

			{
				"id": "sosMap",
				"displayName": "SOS Map",
				"url": "/sos-map",
				"config": "sosConfig",
				"dataService": "SOSJSONService",
				"styleService": "AircraftStyleService"
			}]
	},

	"sosConfig":
	{
		"requestType": "application/xml",
		"poxURL": "http://192.168.2.36:8080/52n-sos-webapp/sos/pox",
		"updateInterval": 500,
		"requestDelay": 0,
		"procedure": null,
		"offering": "http://stue.ch/sensorobservation/offering/flighttracking",
		"properties":
		{
			"http://stue.ch/sensorobservation/observableProperty/callsign":
			{
				"type": "string",
				"name": "callsign"
			},

			"http://stue.ch/sensorobservation/observableProperty/position":
			{
				"type": "geojson",
				"name": "geometry"
			},

			"http://stue.ch/sensorobservation/observableProperty/heading":
			{
				"type": "number",
				"name": "heading"
			},

			"http://stue.ch/sensorobservation/observableProperty/altitude":
			{
				"type": "number",
				"name": "altitude"
			},

			"http://stue.ch/sensorobservation/observableProperty/speed":
			{
				"type": "number",
				"name": "groundSpeed"
			}
		},
		"cleanupInterval": 15000,
		"featureLayerName": "SOS Tracks",
		"dataProjection": "EPSG:4326",
		"timeDeltaLoggerName": "sosDeltas",
		"enableTimeDeltaLogger": true,
		"receivedDataLoggerName": "sosData",
		"enableReceivedDataLogger": true
	},

	"websocketConfig":
	{
		"idProperty": "hexIdent",
		"messageGeneratedProperty": "messageGenerated",
		"url": "ws://192.168.2.5:8443/clientTrackData",
		"featureLayerName": "Websocket Tracks",
		"cleanupInterval": 15000,
		"dataProjection": "EPSG:4326",
		"timeDeltaLoggerName": "websocketDeltas",
		"enableTimeDeltaLogger": true,
		"receivedDataLoggerName": "websocketData",
		"enableReceivedDataLogger": true
	},
	"mapConfig":
	{
		"mapProjection": "EPSG:3857",
		"olCenter":
		{
```

```
                        "lat": 46.801111,
                        "lon": 8.226667,
                        "zoom": 7
                },
                "olBackgroundLayer":
                {
                        "source": { "type": "OSM" }
                },
                "olDefaults":
                {
                        "interactions": { "mouseWheelZoom": true          },
                        "controls":
                        {
                                "zoom": false,
                                "rotate": false,
                                "attribution": false
                        }
                }
        },
        "deltaConfig": { "persistInterval": 60000 },
        "receivedDataConfig": { "persistInterval": 10000 }
}
```