GANYMED_TOOLS_REV_A Dokumentation

Release 0.1

Dipl.-Ing. (FH) Andreas Gschossmann

14.07.2015

The Kalman Filter in its various forms is clearly established as a fundamental tool for analyzing and solving a broad class of estimation problems.

Leonhard McGee and Stanley Schmidt,
Ames Research Center, NASA

Abkürzungsverzeichnis

UAV	Unmanned Aerial Vehicle
UAS	Unmanned Aerial System
JSON	JavaScript Object Notation
BSD	Berkeley Software Distribution
LGPL	Lesser General Public License
I/O	Input/Output
MIT	Massachusetts Institute of Technology
UTF-8	8-Bit Universal Character Set Transformation Format
ASF	Apache Software Foundation
CAN	Controller Area Network

KAPITEL 1

Einleitung

1.1 Ziel der Arbeit

Für den Austausch sowie das Speichern und Lesen von Daten müssen diese in binärer Form, kompakt vorliegen. Beispielsweise ist dies bei der drahtlosen Übertragung von Daten über - in der Bandbreite beschränkte - Systeme notwendig. Es wird auf unterschiedlichen Systemen gearbeitet. Auf dem UAV befindet sich ein eingebettetes System, während auf am Boden im Webbrowser oder auf Betriebssystemen gearbeitet wird. Deshalb sind für die jeweiligen Systeme Anbindungen an unterschiedliche Programmiersprachen notwendig. Die Codierung der Daten nennt man Serialisierung und die Rückgewinnung der einzelnen Datentypen aus den binären Daten Deserialisierung. Um für dieses wiederkehrende Problem der Handhabung strukturierter Sensor- oder Steuerdaten eine möglichst generische Lösung zu schaffen, wird hier auf frei verfügbare Standards gesetzt. In dieser Arbeit soll ein Überblick über die zahlreichen verfügbaren Bibliotheken geschaffen werden. Hierfür werden eine selektierte Untermenge geeigneter Kandidaten untersucht und verglichen und deren jeweilige Stärken und Schwächen herausgestellt. Abschließend wird die Entscheidung getroffen, welche der verfügbaren Bibliotheken am besten geeignet ist.

1.2 Stand der Technik

1.2.1 Hintergrund

Für den Datenaustausch gibt es zahlreiche Formate, die für verschiedene Zwecke geeignet sind. **JSON** ist ein weit verbreitetes Dateiformat in Textform zum Datenaustausch. Es wird von vielen Systemen in nahezu allen Sprachen unterstützt. Da dieses Protokoll in Unicode codiert ist, ist das Footprint der übertragenen Daten sehr groß. Dies ist abträglich, wenn große Datenmengen mit niedriger Latenz übertragen werden sollen. Deshalb wurden von verschiedenen Stellen Formate entwickelt, deren Hauptentwurfskriterium Performanz ist (**Google Protocol Buffers**, **Cap'n Proto**, **MAVlink**, **Apache Thrift**).

1.2.2 Untersuchte Bibliotheken

Google Protocol Buffers wurde von Google für die eigene Infrastruktur entwickelt und erstmals 2008, in der BSD-Lizenz, der Open Source Community zur Verfügung gestellt. [1]

Cap'n Proto ist ein binäres Protokoll mit der selben Zielsetzung wie Google Protocol Buffers. Es wurde vom Hauptautor von Google Protocol Buffers, in der Version 2 - Kenton Varga - entwickelt. Laut Kenton Varga ist Cap'n Proto wesentlich schneller als Google Protocol Buffers. [7]

MAVlink ist eine leichtgewichtige Header-Only Bibliothek, die von Lorenz Meier zusammen mit Andrew Tridgell und James Goppert, im Rahmen des Pixhawk-Projekts [3] der ETH-Zürich entwickelt wurde. Die Software wurde 2009 erstmals unter der LGPL-Lizenz veröffentlicht. [8]

Apache Thrift ist ein Kommunikationsprotokoll, dessen Ursprung bei der Social-Network-Platform Facebook liegt. Es unterstützt sowohl binäre als auch textbasierte Protokolle und bietet die umfangreichste Sprachanbindung aller untersuchten Bibliotheken. Sogar *Erlang*, eine Websprache welche hohe Skalierbarkeit bietet (die derzeit häufig genutzte Alternative ist *JavaScript* in Verbindung mit *node.js*) wird unterstützt. [6]

1.2.3 Einige nicht betrachtete Bibliotheken

Einige Bibliotheken sind zwar verbreitet, doch auf deren Betrachtung wurde gänzlich verzichtet:

- Apache Etch
- **JSON**

Apache Etch [2] scheidet aus, da es nicht für den Einsatz auf einem eingebettetem System ausgelegt ist. **JSON** [5] überträgt Unicode-Zeichen. Dies macht den I/O-Stream zwar für Menschen unmittelbar lesbar, dabei ist dieser unnötig speicherintensiv. Da die Bandbreite in Funkübertragungssystemen begrenzt ist und viele Daten möglichst schnell und fehlerfrei übertragen werden sollen, werden in diesem Rahmen stattdessen nur binäre Protokolle untersucht.

1.3 Rahmenbedingungen

1.3.1 Hintergrund

Abbildung 1.1 [1] zeigt den Aufbau der Funkkommunikation wie er derzeit für das UAS im Sensorik-Applikationszentrum geplant ist. Hier wird ersichtlich, dass verschiedenste Systeme miteinander kommunizieren. Auf dem UAS, findet ein Embedded System, auf der Host-Seite ein Tablet oder ein PC Verwendung. Zwischen allen Systemen müssen Nachrichten austauschen werden. Auf den verschiedenen Plattformen werden unterschiedliche Programmiersprachen auf unterschiedlichem Software-Level verwendet. Deshalb ist es essentiell, dass eine sinnvolle und einheitliche Gestalt der Daten für die Kommunikation gefunden wird. Nur so kann die Kommunikation von vorneherein nachhaltig gestaltet und hoher Zeitaufwand einer nachträglichen Anpassung jener Systeme verhindert werden.

1.3.2 Anforderungen

Eine Anforderung ist die Möglichkeit der Serialisierung und Deserialisierung von Protokolldaten im Browser. Geplant ist eine Kommunikation über Websockets. Auf der Server-Seite wird unter Umständen auf

[1] Diese Abbildung stammt von Waldemar Sessler.

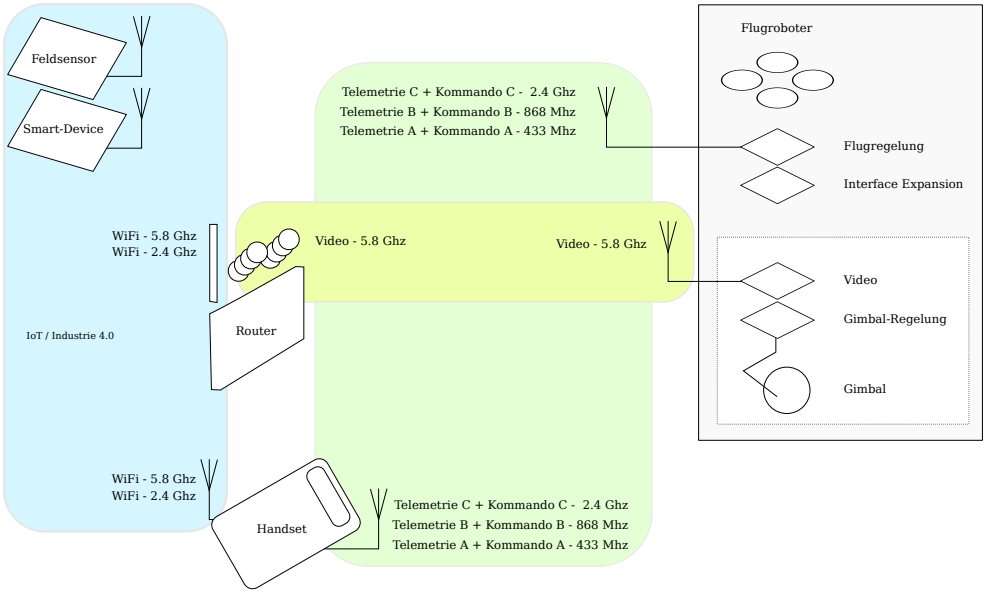

Abbildung 1.1: Funkkonzept des UAS

JavaScript in Verbindung mit Technologien wie *node.js* gesetzt [2]. Auch ist es zweckmäßig, dass die Bibliothek an einem Host-PC betrieben werden kann. Außerdem muss sie auf einem eingebettetem System (FlightControl) lauffähig sein. Hierfür muss die Bibliothek plattformunabhängig sein und verschiedene Programmiersprachen unterstützen.

Die Bandbreite von Funkübertragungssystemen ist begrenzt. Deshalb ist es wichtig, dass trotz eines kleinen Footprints des serialisierten, binären Datenstreams Funktionen zur Serialisierung und Deserialisierung möglichst geringe Laufzeiten aufweisen.

Alle Anforderungen sind in folgender Liste zusammengefasst:

- kompaktes binäres Potokoll
- schnelle Serialisierung/Deserialisierung
- generische Lösung
- flexible Protokollanpassung (Änderungen/Erweiterungen)
- schlanke Bibliothek, auch geeignet für den Betrieb auf eingebetteten Systemen
- Portabilität (Plattformunabhängigkeit)
- Verschiedene Sprachanbindungen (*C*, *C++*, *Python*, *JavaScript*)
- Webtechnologien (JavaScript, node.js)
- Lizenz erlaubt Nutzung

[2] Eine endgültige Entscheidung über die verwendete Technologie ist an dieser Stelle noch nicht gefallen. Diese Problematik, wird umfassender in der Abschlussarbeit von Waldemar Sessler untersucht.

Auf diese Anforderungen werden die jeweiligen Bibliotheken im Folgenden untersucht und verglichen.

KAPITEL 2

Vergleich der Bibliotheken

2.1 Diskussion

In folgender Tabelle sind die wichtigsten Eigenschaften der einzelnen Bibliotheken zusammengefasst. Neben den möglichen Sprachanbindungen aus Tabelle 2.2, wird auf diese Eigenschaften im Folgenden eingegangen.

Tabelle 2.1: Vergleich von Bibliotheken zur Serialisierung strukturierter Daten

	JSON	google protobufs	Cap'n Proto	MAVlink	Apache Thrift
Human Readable	*ja*	*nein*	*nein*	*nein*	*nein/JSON*
Protokoll	*UTF-8*	*binär*	*binär*	*binär*	*binär/JSON*
Lizenz	*MIT*	[4]	*BSD*	*LGPL*	*ASF 2.0*
Website	[5]	[1]	[7]	[8]	[6]
Checksumme	*keine*	*keine*	*keine*	*ITU X.25*	*keine*

Google Protocol Buffers unterstützt zwar keine Checksummen, doch können diese bei Bedarf selbst als eigene Nachricht implementiert werden. In der neuesten Version werden Google Protocol Buffers von *Ruby* unterstützt, was dessen Einsatz mit Webtechnologien ermöglicht. In diesem Rahmen ist jedoch die Verwendung von *JavaScript* gefordert, was von Google Procol Buffers nicht nativ unterstützt wird. Ein Vorteil von Google Protocol Buffers ist, dass es sogenannte *optional fields* unterstützt, das sind Daten, die nur bei Bedarf im Binärstream mitgesendet werden können. Außerdem wird in der neuesten Version *Go* unterstützt, dessen Verwendung für zukünftige Webprojekte von Vorteil sein könnte.

Cap'n Proto ist eine verbesserte Version von Google Protocol Buffers, welche von einem der Entwickler von Google Protocol Buffers geschrieben wurde. Es besticht durch die gute Performanz bei der (De-)Serialisierung und die vielen unterstützten Programmiersprachen. Es wird nur von privaten Quellen gepflegt, was ein Nachteil sein kann.

Apache Thrift besticht durch seine hohe Anzahl an Features und die untersützung von vielen Programmiersprachen. Es könnte beispielsweise durch die Nutzung von *Erlang* oder *node.js* hervorragend in ein skalierbares Webprojekt eingebunden werden.

MAVlink wurde im Gegensatz zu den restlichen Protokollen speziell für die Benutzung in eingebetteten Systemen mit Funksystemen mit eingeschränkter Bandbreite entwickelt. Der binäre Daten-Stream ist dabei sehr schlank und enthält eine Prüfsumme [1]. Außerdem wird im Gegensatz zu Google Protocol Buffers

[1] Die Prüfsumme ist die selbe, wie sie in den Standards *ITU X.25* und *SAE AS-4* (CRC-16-CCITT) verwendet wird. Sie wird in

auf dynamische Speicherverwaltung verzichtet, was es gut anwendbar auf Mikrocontrollern macht. Zwar unterstützt MAVlink *JavaScript* nicht nativ, trotzdem könnte der Vorteil der guten Einsetzbarkeit auf eingebetteten Systeme gegenüber den anderen Protokollen entscheidend sein.

Tabelle 2.2: Sprachanbindung ausgewählter Bibliotheken zur Serialiserung

	JSON	google protobufs	Cap'n Proto	MAVlink	Apache Thrift
C	○	○	○	●	●
C++	○	●	●	●	●
Python	○	●	○	●	●
Java	○	●	○	○	●
JavaScript	○	○	○	○	●

● Native Unterstützung in vollem Umfang
○ Nur bedingte Unterstützung durch verschiedene Autoren mit ungewisser Maintainance, möglicherweise mit eingeschränktem Funktionsumfang

Apache Thrift unterstützt neben den in Tabelle 2.2 genannte Programmiersprachen noch *PHP*, *Ruby*, *Erlang*, *Perl*, *Haskell*, *C#*, *Cocoa*, *node.js*, *Smalltalk*, *OCaml* und *Delphi*.

2.2 Fazit

Wie aus den Tabellen und den Ausführungen aus Kapitel *Diskussion* hervorgeht, decken alle Bibliotheken große Teile der Anforderungen ab. Einzig MAVlink sticht jedoch dadurch hervor, dass es für den Gebrauch auf eingebetteten Systemen optimiert ist. Da letztlich alle anderen Anforderungen, wenn auch manche mit etwas Integrationsaufwand, mit MAVlink umgesetzt werden können, ist dies der entscheidende Vorteil, den MAVlink gegenüber den anderen Bibliotheken bietet. Alle anderen Bibliotheken sind auf die Umsetzung auf Betriebssystemen ausgelegt. Häufig werden zwar Lösungen für eingebettete Systeme angeboten, jedoch nie nativ und mit eingeschränkter Funktionalität, so dass diese eher als Notlösung erscheinen.

Das einzige Feature, das MAVlink nicht unterstützt ist eine reine *JavaScript*-Sprachanbindung. Zwar können *JavaScript*-Sources erzeugt werden, diese basieren jedoch auf *node.js* und sind damit für die serverseitige Nutzung vorgesehen. Dieser Nachteil kann jedoch mit vertretbarem Aufwand beseitigt werden. Es können entweder die vorhandenen *JavaScript*-Klassen so umgeschrieben werden, dass sie auch ohne *node.js* im Browser lauffähig sind, oder C-Funktionen nach *JavaScript* portiert werden. Dies kann beispielsweise mit Emscripten geschehen. Die Attraktivität dieser Lösung liegt darin, dass der Nachteil fehlender Unterstützung von 64-bit Integer in *JavaScript* umgangen würde.

Außerdem ist MAVlink durch **CAN** und Luftfahrtstandards aus *SAE AS-4* (*CRC-16-CCITT*) inspiriert. Auch das macht es für UAV-Anwendugnen attraktiv.

SAE AS5669A dokumentiert. [8]

KAPITEL 3

Implementierung

Zu Testzwecken wurden einfache Beispiele zur Anwendung von MAVlink in verschiedenen Sprachen implementiert. Es wurde ein Python-Skript geschrieben, mit dem MAVlink und einige notwendige Abhängigkeiten installiert werden können. Danach können die Beispiel-Codes kompiliert und ausgeführt werden. Die Installation und Ausführung jener Codes wird im Anhang in den Kapiteln *MAVlink installieren* und *Ausführen des Sample Codes* beschrieben.

Alle weiteren Infos, zur Benutzung und Funktionsweise von MAVlink, finden sich unter [8].

KAPITEL 4

Ausblick

Für einen letzten Vergleich könnte noch der Vollständigkeit halber ein einfacher Benchmarktest in Bezug auf Serialisierungsgeschwindigkeit und Footprintgröße der Binärdatei durchgeführt werden. Hierfür war in diesem Rahmen keine Zeit. Trotzdem hat sich MAVlink, durch seine gute Integratierbarkeit in eingebette Systeme, als am besten geeignet herausgestellt.

ANHANG A

MAVlink Samples

Bemerkung: Alle nachfolgenden Schritte wurden unter Ubuntu 14.04 getestet.

A.1 MAVlink installieren

Herunterladen und entpacken:

```
mkdir myMAVtools && cd myMAVtools
scp -P 4444 gsa39665@hps.hs-regensburg.de:html/files/ganymedtools_v0.1.2015.tar.bz2 .
tar xvjf ganymedtools_v0.1.2015.tar.bz2
```

Für die lokale Installation von MAVlink und einiger Abhängigkeiten wurde ein Skript geschrieben. Dies kann mit folgenden Befehlen ausgeführt:

```
cd GanymedTools/mavlink
./setup.py --install
```

Nun muss noch die Pfadtvariable PYTHONPATH gesetzt werden, damit die Python-Tools von MAVlink systemweit ausführbar werden. Wird dieser Schritt vergessen, kompilieren unter umständen einige Codes nicht, da die nötigen Headers vom Makefile nicht erzeugt werden können. Auch dies kann mit dem genannten Installationskript gemacht werden:

```
./setup.py --setpath
```

A.2 MAVlink deinstallieren

Die Deinstallation von MAVlink wird mit folgendem Befehl durchgeführt:

```
./setup --uninstall
```

A.3 Ausführen des Sample Codes

Bemerkung: Folgende Befehle setzen vorraus, dass die Schritte aus Kapitel *MAVlink installieren* erfolgreich durchgeführt wurden.

Die Sample-Codes zur Implementierung von MAVlink in verschiedenen Sprachen befinden sich in den jeweiligen Unterordnern des Ordners *GanymedTools/src*. In allen Unterordnern befindet sich eine Datei namens *readme.rst*. Darin befinden sich Anweisungen, die jeweiligen Codes zu kompilieren und auszuführen.

Mit folgenden Befehlen können die C-Codes unter Linux kompiliert und ausgeführt werden.

Headers von MAVlink erzeugen:

```
cd GanymedTools/src/linux
make headers
```

Und schließlich können die Codes im selben Verzeichnis mit folgendem Befehl kompiliert werden:

```
make binary
```

A.4 Troubleshooting

Problem: Will man die JavaScript Sources mit *mavgenerate.py* erzeugen kann folgender Fehler erscheinen:

```
Errors occoured in mavgen:
[Errno 2] No such file or directory:
'./javascript/lib/jspack/jspack.js'
```

Lösung: Damit das Submodule *jspack* nachgeladen wird, müssen folgende Befehle ausgeführt werden:

```
git submodule init
git submodule update
```

Zwar tritt dann die Fehlermeldung immer noch auf, kann aber dann ignoriert werden. Dies geht aus folgendem Thread hervor:

https://groups.google.com/forum/#!topic/mavlink/iDKpDrqL9AI

Außerdem werden hier Möglichkeiten diskutiert, *JavaScript* ohne die Abhängigkeit von *node.js* durchzuführen.

Literaturverzeichnis

[1] Google Code. Protocol buffers. http://code.google.com/p/protobuf. [Online; accessed 15-März-2014].

[2] Apache Etch. Apache etch. http://etch.apache.org. [Online; accessed 15-März-2014].

[3] ETH-Zuerich. Pixhawk. https://pixhawk.org/start. [Online; accessed 15-März-2014].

[4] JSON. The json license. http://www.json.org/license.html. [Online; accessed 15-März-2014].

[5] JSON. Introducing JSON. json.org. [Online; accessed 15-März-2014].

[6] Apache TLP. Apache thrift. https://thrift.apache.org/. [Online; accessed 15-März-2014].

[7] Kenton Varda. Cap'n proto cerealization protocol. https://capnproto.org. [Online; accessed 15-März-2014].

[8] Lorenz Meier (ETH Zürich). Mavlink - micro air vehicle communication protocol. http://qgroundcontrol.org/mavlink. [Online; accessed 15-März-2014].