GPS TRACKING WITH JAVA EE COMPONENTS

CHALLENGES OF CONNECTED CARS

Kristof Beiglböck

CRC Press
Taylor & Francis Group

GPS Tracking with Java EE Components

Challenges of Connected Cars

GPS Tracking with Java EE Components

Challenges of Connected Cars

By
Kristof Beiglböck

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed on acid-free paper
Version Date: 20180601

International Standard Book Number-13: 978-1-138-31382-8 (Hardback)
International Standard Book Number-13: 978-1-138-05494-3 (Paperback)

Library of Congress Cataloging-in-Publication Data

Names: Beiglböck, Kristof, author.
Title: GPS tracking with Java EE Components : challenges of connected cars / by Kristof Beiglböck.
Description: Boca Raton : Taylor & Francis, CRC Press, 2018. |
Includes index.
Identifiers: LCCN 2018014010| ISBN 9781138054943 (pbk. : alk. paper) |
ISBN 9781138313828 (hardback)
Subjects: LCSH: Vehicular ad hoc networks (Computer networks)--Data processing. |
Automatic tracking--Equipment and supplies. | Global positioning system. |
Java (Computer program language)
Classification: LCC TE228.37 .B45 2018 | DDC 629.2/72--dc23
LC record available at https://lccn.loc.gov/2018014010

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

Contents

Preface

The times of proprietary products dominating the software industry are over. Today more than 80% of all software applications are compiled of open source components and with the combination of Java and Open Source Software (OSS) alone there is practically nothing you can *not* program. The Apache Maven Build Tool has emerged to an automated project manager. The architect of a new application is in charge of composing the overall Project Object Model (POM). Once the project is defined each developer can create the software from scratch and begin to implement code with his domain knowledge.

Maven Central is hosting millions of artifacts and serving tens of million downloads per week, tens of billion per year, with growing demand. A Maven Server can easily be setup inside your company to support distributed software development for your team. Maven is machine automation of development, integration, testing, documentation and deployment. The infinite number of Java OSS Components is the modern challenge for software architects to compose an application.

Choosing the right OSS components already is a challenge and this book will demonstrate how to select Java and Java EE Components and compose the technical framework for your application before you start adding business code. Our business domain is GPS Tracking and we will see how to create individual components for a GPS Tracking System (GTS) and put them together to a running system. This process of picking JEE components and adding GTS features can be expressed in the pseudo equation

$$\text{JEE} + \text{GTS} = \text{JEETS}$$

where JEETS is pronounced like G.T.S., i.e. JEE T.S.

The book's index provides an overview over JEE and JEETS components. Instead of a Bibliography the book provides a 'Recommended Readings' Section of freely available Internet sources to be used while you are reading the JEETS code. In the recommended readings you can find book recommendations to explore each component in full depth – after you have gone through the book for a complete picture.

Instead of creating an application from scratch, which exceeds the limit and purpose of a book, we will work with the Open Source Tracking System Traccar. Traccar is a popular Java GTS that you can install out of the box and start tracking with your smartphone immediately. Traccar is a proven product, maintained very well and upgraded frequently by Anton Tananaev. Once Traccar is up and running we will take it as the running production system and put ourselves in the position to customize and replace GTS functionalities with new modules. This is a realistic situation for most professional developers.

We will analyze the Traccar GTS architecture to identify functional components like Device Communication Servers (DCS) and then create a JeeTS DCS component that can run stand alone or be embedded inside an Jee Application Server. With this approach you will learn how to decompose a monolithic application into independant modules and then apply integration technologies to put the pieces back together with explicit endpoints to run them as one system.

At the end of the book we will have created a Maven Repository with well aligned JeeTS components that you can use as seed components. Hopefully we will meet at `jeets.org` to create integration tests and improve the JeeTS components according to your requirements. You will able able to create your own Java components and combine them to a new application or with your running production system. You should also become aware of the paradigm shift from large Jee application servers to micro services that can be scaled individually. You will be able to create an Enterprise Application running on an AS or you can create many components and assemble them to one server backend application or both.

The Self Driving Car (SDC) is the latest global challenge of steering a real car with software. You could not navigate a car through a city without information about the traffic situation. GPS Tracking has to be taken to a higher level to be able to track many cars at the same time and provide the aggregated traffic situation to each car. This constellation is referred to as Connected Car Technology and we will look at the impact on classical tracking systems.

CHAPTER 1

Introduction and Overview

CONTENTS

1.1 GPS TRACKING WITH JAVA EE

Vehicle and fleet tracking has become a large industry over the last decade and the market offers innumerable trackers for different appliances. With the smartphone a new tracker type was introduced as a multipurpose well performing *Client* computer with full connectivity to *any* sensor.

On the *Server* different GPS Tracking Systems (GTS) have evolved to track people, assets, vehicles ... *any*thing. A vehicle tracking system can be used to observe a fleet and locate vehicles on a map in a browser frontend. The system can raise individual events to improve logistics, monitor fuel consumption, create travel reports etc. A transport company couldn't persist in the market today without a GTS and the software industry provides full fledged solutions to run a business.

Since the automotive industry has recognized tracking technologies as *the* live data source to actually direct cars through a complex environment the classical GTS architecture has to consider many new aspects. This book analyzes how the challenges of the Self Driving Car (SDC) exceed the limits of a classical GPS Tracking System. For a thorough analysis the widely accepted and well designed Open Source Traccar GTS [1] is reverse engineered in detail to dissect the major GTS constituents. Traccar is a monolithic application in Standard Java (JDK) running in a single JVM and we will re/model individual GTS components to reusable Java EE,i.e. JEE Modules.

The readers can witness the prototyping and modeling *process* and modify their own software. This book can help you to set up *your* tracking system by customizing the components. Every component is introduced in detail and includes a number of *design decisions* for development. The modules are combined into a customizable GTS – the JEE Tracking System JEETS . And they can be used as seed components to enrich existing systems including Java middleware and JEE application servers with live tracking. JEETS is a toolbox of GTS components combined to a bare bone level that you can also use in conjunction with Traccar.

1.2 THE CHALLENGE OF THE SELF DRIVING CAR

Navigation systems and digital maps have become extremely precise to localize a vehicle on a digital map with GPS coordinates. It was only a matter of time before tracking and mapping will be merged to actually guide the vehicle toward becoming a Self Driving Car (SDC). This challenge is currently changing the automotive industry from programming embedded software to hosting services and data crunching – in real time with really Big Data. The Internet has grown rapidly to serve people. Now it is being prepared to assist cars, sometimes within seconds, as they move.

The main goal in development of a self-driving car is to provide a calculated route and to actually control the car to follow it. At the first impression the SDC seems like a logical continuation of automotive developments. Existing intelligent Parking Assist Systems are noteworthy, since they actually take *complete control* over the car for a complex maneuver: heavy sensitive steering, driving forward and backward with high precision – *without human interaction*. This book will not explore the internal control of a vehicle but rather focuses on client server programming to provide useful information *fast*.

So the SDC is not really a revolutionary idea? Is it simply a normal evolution of technology? It is much more. The challenge is the live *coordination* of moving cars and the prediction of *coincidences* that could trigger events for cars approaching each other and avoid accidents.

The main difference of the SDC scenario to a classical GTS is that the latter only processes the tracking information of a single device! In the SDC scenario we need to evaluate submitted values of different vehicles and provide fast feedback to each vehicle as it moves. The system has to first evaluate and classify each vehicle message before comparing and aggregating different vehicles and finally supplying conclusions to each vehicle.

This SDC challenge requires a *remodeling* of a classical GTS to more independent components. We will combine these components on a higher level to form individual use cases which can be processed in parallel. In order to achieve this the JEETS will be modeled following some major guidelines:

Don't look into the past.
A classical GTS usually makes frequent use of database lookups to determine where the vehicle was previous to the submitted position.

Don't speculate on the future.
Some GTS keep up a connection dialog to wait for incoming events.

Don't wait for external resources.
The response time of an external resource is generally unpredictable. An external database represents a hardware resource and can easily become the main bottle neck of the complete system.

1.3 INTENDED AUDIENCE

Another main topic of this book is the reflection of JEE technology itself. Containers have diminished and can be created on the fly to execute a single method – and then be dropped again . . . JEE technology is slowly moving away from application servers to smaller independent Java (EE) components which are combined to a system. Many of these components will be introduced and applied for the creation of stand alone JEETS components.

In Java EE we usually speak of the architect and developer roles, which does not reflect the modern software development completely. The build tool and sophisticated test frameworks have become a core part of the development process. The Java EE platform is 'only' a higher level *specification* of many globally accepted API specifications related to each other, to the Java world and *the network*. Before actually starting an implementation the architect has to work as a software *composer* to create the built environment. All JEE client- and server modules developed in this book can be found in the GIT repository and can be compiled, tested, packaged and installed with Maven.

One typical problem to upgrade a standard Java GTS to a Java EE GTS is the direct connection to the hardware. Tracking messages arrive at the ports of a computer, a hardware. On the other hand in the JEE runtime environment, the application server implementation is designed for inversion of control (IoC) and the business logic should not include any code to interact with server, operating- nor file system. Otherwise scalability over different machines will not work!

Therefore this book can also help Java developers not only interested in GPS tracking, but in modern software design from many individual modules. The complete system design process is described bottom up beginning with the creation of data formats. The reader should be able to sense the fundamental importance of a persistence unit and a compatible network data format for data transmission, which actually pre/define the system and dictate each component's design. Since it is merely impossible to completely define data formats before creating a system we will describe how to create formats, which

can be modified during the development and according to new requirements to an existing system.

Since SDC development in the automotive industry is highly competitive and therefore taking place under non disclosure we will focus on applying Java EE technology to existing GPS tracking systems. By providing live tracking information to Entity Java Beans (EJBs) a GTS developer can easily break the isolation of his GTS and combine it with other systems of the enterprise. Just imagine clicking on a truck icon on the map frontend to retrieve the actual truck load from the warehouse and other logistic systems.

1.4 SKILLS / SKILL LEVEL

This book does *not* introduce Java EE and you are expected to...

install the JSE Traccar GTS

install the Postgres database (with PostGIS)

download / clone a GIT repository

apply the build tool Maven (clean compile test package install ..)

install the application server Wildfly

install ActiveMQ

configure the ActiveMQ resource adapter for WildFly

The book's source code and sample application are available to the reader on the book's website `jeets.org`. The book should be used as a hands-on instruction accompanying the source code. The reader is expected to download and compile the sources in a personal development environment. You can go through the appendixes all at once to setup your environment or you will be asked to setup required software by single appendixes as you read.

1.5 THE AUTHOR

The author is specialized on Geographic Data Processing and has worked for the automotive industry for more than a decade and witnessed the developments from GPS, digital maps, routing to navigation. The author has setup commercial GPS tracking systems with open source components and has been architect and developer in large automotive tracking projects. With the goal of a Self Driving Car GTS technologies were applied for real time tracking.

I

Data Formats and Relations

CHAPTER 2

Message Exchange

CONTENTS

2.1 GPS PROTOCOLS

The new challenge of handling crowd sourced data flow of smartphones, cars and connected things is the reduction of data to streams without losing information. In the context of the SDC every single bit and every millisecond eventually define the information limit to support every car – for a single highly frequented traffic node – as well and fast as possible.

In the course of this book we will develop different JEE components especially for GPS tracking purposes. To start the development from scratch we will look at the basic information bits of tracking: GPS coordinates. GPS was thoroughly introduced in the first book and broken down to the following fields needed to describe a point in time and space:

`longitude`: $]-180°, 180°]$ decimal degrees east/west +/-

`latitude`: $[-90°, 90°]$ decimal degrees north/south +/-

`altitude`: meters above (+) or below (-) see level (0) [default]

`timestamp`: UTC[1] time and date

[1] Universal Time Coordinated, formerly Greenwich Mean Time (GMT)

`event`: The cause of sending this GPS coordinate

Tracking information, i.e. events should *always* be tagged with a coordinate `GPS(lat,lon,alt,time)` to describe *what* has happened *where* and *when*. For the time being the event stands for any (sensor) information to be transferred to describe the meaning or cause of a transmission. The submitted fields are transformed into Plain Old Java Objects (POJO) to become entities of a tracking system. (see Figure 2.1).

Let's look at these four fundamental GPS fields to gain an understanding of different data formats and how much space, i.e. bits, they require. As Java developers we want to handle the submitted values defined by Java primitives or -types as the server target format. Although a 32-bit representation in a `float` would be sufficient lat, lon and alt are commonly used as 64-bit `double` primitives. A vital question in a tracking context is how to transfer the GPS data over the network with a minimum number of bits without losing precision?

For the ease of use many trackers transfer information in a human readable text (ASCII csv) format:

```
    keys       time          lon    lat    alt
  values    ..,170312123644,120589,490234,0,..
 formats       YYMMDDHHMMSS ddmmmm  ..
29 bytes      +---------+---------+---------
```

The first field represents a time stamp followed by lat and lon represented as digits and finally the altitude relative to sea level. This is useful for display purposes or to send tracking messages via SMS and many other human interactions. For an SDC scenario this format is a disaster.

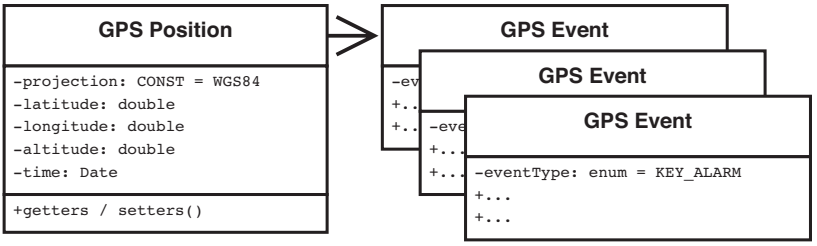

Figure 2.1 The server transforms the messages into related POJOs. GPS and events *can* be modeled in one or more system entities. Their relation *can* be modeled as `1:1` or `1:n` as needed.

TABLE 2.1 Integer Coordinates with a Precision Field

longitude	dir	integer	precision	double
121.1234567	East	+1211234567	7	+121.1234567
121.1234567	West	-1211234567	7	-121.1234567
21.1234567	East	+211234567	7	+21.1234567
21.1234	East	+211234	4	+21.1234

In a text format at least one byte is needed for every digit (or in this case for every character) and separator which adds up to 29 bytes. Also it is unclear whether the altitude of zero is a default value or real (unlikely). The lat and lon values are restricted to 6 digits to fix their overall precision.

From the data processing perspective it's a good idea to use integers instead of decimals, if you compare FLOPS and MIPS performances of computer systems. The problem emerging from the SDC is the precision of lat and lon. Digital maps were used to provide an accuracy in meters and were subsequently refined for ADAS[2]. Since this is still insufficient for autonomous driving a new type of map is coming up with centimeter precision: the HD Map for machine reading.

On the other hand not every location is provided in centimeter precision. Therefore we could define lat and lon as integers and add a precision field (see Table 2.1). The precision value 4 represents the divider 10^4, i.e. multiplier 10^{-4} and allows defaulting to a precision or specifying it explicitly for any coordinate. The space needed for java variables is a 16-bit `short` for the integer, a 8-bit `byte` for the precision and a 64-bit `double` for the targeted variable in the server code. Now we have a 24 bit representation for lat and lon, which is much better than the text values listed above.

Depending on your application you might even consider submitting a small route, i.e. a collection of related position infos. This opens the possibility to tag each position with small deltas for lat, lon, alt. Yet this is a very project dedicated approach which can also introduce risks. In a networking context we don't have Java types and there is still room for compression by looking at the single bits...

2.1.1 Message Format Encoding

If you buy a tracker you usually don't have the chance to define a protocol and you must reverse engineer the one provided. Therefore we will look at a few tracking protocol formats and some typical implementation problems they pose. Afterwards we'll define our own JeeTS data format with clear structures for fast prototyping.

[2] Advanced Driver Assistance System

We will now look at the `tk102` (or `tk103` etc.) as it is human readable (see [5]). A typical message `String` encoded by the tracker looks something like this:

```
008238008589BO01141129A2302.7532N07232.2461E000
                            .0092142349.381000000AL000000F1
```

The server application receiving the message has to know the format in order to parse it into the final variables of your application:

```
008238008589 - trackerid / mobile number
BO01         - alarm event
141129       - YYMMDD date
A            - valid data / V - invalid data
2302.7532    - latitude, format ddmm.mmmm
N            - N = north, S = south
07232.2461   - longitude, format (d)ddmm.mmmm
E            - E = east,  W = west
000.0        - speed k/m
092142       - UTC time HHMMSS
349.38       - altitude in meters (cm precision?)
1000000A     - bit representation of several events:
               power on/off/external, ignition on/off, ...
L000000F1    - mileage
```

We won't go into the details, yet it's worth noting that lat, lon and time have the format of NMEA sentences and indicate that these values are propagated from the GPS processor output. Here is a Decoding Algorithm into a Java target format which should provide an impression of a Decoding Process:

```
2302.7284 = latitude, format is ddmm.mmmm
        N = north, S = south
>>> (d)dd + mm.mmmm. >>> 23 + 02.7284
>>> 02.7284/60 = 0.0454733333333333
>>> 23 + 0.0454733333333333 = 23.04547333333333
>>> Then multiply the result by -1 if the direction is S
>>> as N so 1*23.04547333333333 = 23.04547333333333
```

This sample conversion of the latitude indicates processing time, which again is something we should not waste for the SDC scenario. It may seem picky, but we want a precise look at transferred data to identify the bottlenecks. Another transformation could be required to calculate the *local time* from the UTC time and geocoordinates etc.

You should be aware of the `altitude` source. If altitude is important for your application you should not rely on the value supplied by the GPS unit! The height is determined from the satellites and highly depends on their constellation. Every time the GPS switches to other satellites the altitude precision increases or decreases depending on the geometry. The original NAVSTAR GPS from 1984 simply wasn't made for this. If you need to rely on

accurate altitude values you should look for a tracker with a built in barometer. If you are building your own tracker you can get a barometer component with centimeter precision starting somewhere around 10 US dollars.

In the end every protocol is defined by groups (message definitions) of key-value pairs while the actual message is composed only of values. Due to data reduction client and server software have to agree on the (current) format of all sentences that define a protocol. The tk102 position message indicates some more challenges of data protocols.

If you take another look at the message string on page 10 it is obvious that the string length is vital information for decoding. If any value is missing, how could you decode the message? In the next chapter about TCP communication we will look at another GPS message string – this time with comma separated values (csv) which allows to submit empty fields as

```
"imei:359587010124900,,,13554900601,F,132909.397,,,"
```

A tracker specifies a certain protocol format for events triggered by external sensors or internal states like the battery level. The actual GPS processor protocol is usually the NMEA 103 format which was introduced for the very first GPS units around 1984. As we could see some trackers provide a setting to transfer NMEA sentences in a (complicated) raw format, like `(d)ddmm.mmmm`.

Later we will look at tracker architectures to understand how NMEA plays an important role inside the tracker controller to evaluate the GPS output and select high quality locations by taking accuracy, satellite constellation etc. into account. A tracker logic is responsible for selecting good events and formatting them into a tracker protocol format.

2.1.2 Message Type and ID

Sending a tracking message with a position and an event is the basis of tracking. Tracker protocols define different messages to submit event details. To set up a device communication software for a dedicated device you need to apply different value encoders and decoders repeatedly. Values for a certain key, i.e. latitude appears in different messages.

Commonly device communication software is designed for various messages of a tracker protocol and have many `switch .. case` constructs in their code to switch to message types and then field definitions. Many devices provide a header containing only message Id or place the Id as the first field in the message sentence. Many protocols tag the message Id with a `$`. As an example we'll look at some NMEA sentence formats

```
   GLL - Geographic Latitude and Longitude:
$GPGLL,4916.45,N,12311.12,W,225444,A,*1D
   GGA - essential fix data with 3D location and accuracy data:
$GPGGA,123519,4916.45,N,12311.12,W,1,08,0.9,545.4,M,46.9,M,,*47
```

to find that the messages `$GPGLL` and `$GPGGA` share some fields.

2.1.3 Message Catalogs

The `tk102` protocol format is helpful for a textbook and is sufficient to describe the process of encoding and decoding messages. For higher network traffic and smaller data sizes more sophisticated devices apply byte and bit manipulations for every data field.

To get an impression of a complex device you should have a look at Garmins Fleet Management Interface (FMI) [6]. FMI units provide a large number of features, i.e. messages used for fleet management solutions with navigation and messaging. Garmins FMI devices represent a connector for external truck sensors like a chat interface or a dash camera and much more.

The complete FMI functionality is specified by a large message catalog [7] where you will find message IDs for many different appliances:

A607 Waypoint Management and Driver Status

A611 Server to Client Long Text Message Protocol

A622 CAM 1.0 (Dash Camera Protocol)

The implementation details can be found in the Garmin Device Interface Specification [8] which has about 70 pages of technical details for encoding and decoding. We will not go into the specifics of extracting information from a decoded field and this specification can serve as a substitute for you to get a good impression of a complex device communication. The document describes data types, protocol layers and application protocols that you can study as needed.

Another good reason to look at the FMI devices in the SDC context is the fact that it is not a tracker! Although it does have a GPS unit and even more a digital map for navigation it does not hold a GSM unit. In order to communicate with a GTS or Fleet Management System the device has to be connected to a tracker with a GSM unit. The FMI device communicates with the tracker via physical protocols like the RS-232 or USB Standards. This is similar to the constellation of a tracker in a (self driving) car. The tracker is merely the device to submit messages and add a time and place stamp. We will look at some implementation details to deal with two devices a little later.

Now that we have a good idea of fields, messages and message catalogs we proceed to create a device communication server for dedicated message catalogs representing a device and its external sources or sensors.

2.2 TCP/IP COMMUNICATION

Generally we can assume that a tracking system is designed to receive live GPS information and telematics from a remote tracking device over the air (OTA). Since a modern tracking system should also be able to handle indoor tracking we don't want to restrict the system to OTA. The most general approach is to focus the Internet protocol TCP/IP and ignore the transport media – which is the actual benefit of TCP.

Transmission Control Protocol[3]

> The Transmission Control Protocol (TCP) is one of the main protocols of the Internet protocol suite. It originated in the initial network implementation in which it complemented the Internet protocol (IP). Therefore, the entire suite is commonly referred to as TCP/IP. TCP provides reliable, ordered, and error-checked delivery of a stream of octets between applications running on hosts communicating by an IP network.

For developers this information is concisely focusing on the implementation. We associate TCP communication with a port and an IP address on client- and on server side. And most important: an IP address always defines a hardware endpoint to software!

Port (computer networking)[4]

> In the internet protocol suite, a port is an *endpoint of communication in an operating system*. While the term is also used for hardware devices, in software it is a logical construct that identifies a specific process or a type of network service. A port is always associated with an IP address of a host and the protocol type of the communication, and thus completes the destination or origination address of a communication session. A port is identified for each address and protocol by a 16-bit number, commonly known as the port number.

For our software design we can simplify the complete communication by specifying `IP:port` combination at least for the server and, if available (i.e. online), for the client. You may miss the other common protocol UDP, which is also applied in various tracking devices. Anyway TCP/IP is a useful constellation we can work with and you can treat UDP implementations in a similar way. The data flow for the new system JeeTS can be simplified to:

```
Tracking Device > TCP > Tracking Server > JeeTS
```

[3] From Wikipedia, the free encyclopedia
[4] From Wikipedia, the free encyclopedia

This is a pragmatic model to start an implementation and as we move on we'll see how to transfer data back to the Self Driving Car via JEE services. Note that the 'tracking server' is actually a Device communication Server (DCS). A DCS represents a single component of the tracking server and is coded to communicate with well known messages of the trackers Message Catalog.

By looking only at the TCP communication we can also cover the fact that indoor tracking is usually based on various different technologies, since GPS is not available. Besides indoor tracking we should keep a new emerging industry in mind, the Internet of Things (IoT), Industry 4.0 and so on. In the end we only need values for time, longitude and latitude for tracking.

For indoor tracking the altitude is sometimes used in a different way. Abstractly speaking buildings have one or more floors with a constant altitude and the altitude might store the floor number, which is used to load its ground plan. Check out the Minneapolis Mall of America in Google maps to get an idea of map matched ground plans.

2.2.1 TCP with Standard Java

Java was invented as *the* network- and Internet programming language and Java objects in general can communicate directly via TCP (or UDP) by design. Distributed systems, like the tracker and tracking server, need reliable communication. A Java server application can bind a socket to a specified port and exchange information via a point-to-point channel. During this session data can be exchanged in both directions.

Some tracking systems simply keep up the connection to a car as long as possible although most of the time the connection is idle and virtual. This model is usually driven by the GMS provider fees. For an existing connection the provider charges the actual bytes being transfered while every new connection requires additional data packets. It is most expensive to transfer every tracking message over a new connection with much more overhead than the actual information.

The Java developer does not have to deal with TCP layers when using the `java.net` package. While TCP is a generic protocol many more customized protocols like `http` and `ftp` are implemented on top of TCP and specified by the port in the URL. The Java `URL` class offers many methods to deal with these protocols to establish connections and exchange data. A simple position message could be transmitted via http URL encoding:

```
http://host:5055/?id=401258&lat=49.158&lon=12.864&timestamp=192412
```

The other important class to know is the `Socket`. After establishing a client server connection each side communicates to a local socket which can be associated with an instance, a session or channel of the relevant software logic. Sockets provide the ability to establish and distinguish multiple client server connections at the same time.