Conference Paper Title*

*Note: Sub-titles are not captured in Xplore and should not be used

1st Justin Frommberger
Interaktionstechnik und Design
Hochschule Hamm-Lippstadt
City, Country
email address or ORCID

2nd Jonas Gerken
Interaktionstechnik und Design
Hochschule Hamm-Lippstadt
City, Country
email address or ORCID

3rd Benedikt Lipinski
Interaktionstechnik und Design
Hochschule Hamm-Lippstadt)
Soest, Deutschland
benedikt.lipinski@stud.hshl.de

4th Phillip Wagner
Interaktionstechnik und Desgin
Hochschule Hamm-Lippstadt
City, Country
email address or ORCID

***Abstract*—This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract.**

***Index Terms*—component, formatting, style, styling, insert**

I. Server

In diesem Kontext spielt der Server eine sehr wichtige Rolle in der Kommunikation zwischen den ausführenden Parts des Projektes. Durch den Server und seine Strukturen wird letztendlich erst eine Plattform geschaffen, die allen Fahrzeugen und den Kunden (Usern) eine Möglichkeit bietet, eine Verbindung untereinander zu schaffen und weitere Aufgaben zu erledigen. Konkret waren die Aufgaben des Servers:

Anmeldung von Clients
Das nächstgelegene Fahrzeug finden
Jedem Client eine eindeutige ID zuordnen
Übermittlung der Fahrzeugdaten an den Kunden
Interne Verarbeitung einer Fahrzeugbestellung

A. Anmeldung von Clients

Aufgabe des Servers ist es, eine Anmeldung von Clients zu ermöglichen, um einerseits nur aus dem Pool der aktuell aktiven, freien Fahrzeuge auszuwählen und andererseits, einer unbekannten Menge an Fahrzeugen und Kunden die Möglichkeit zu bieten, am Angebot teilzuhaben. Zu den Clients gehören sowohl die Kunden (User), wie auch alle Fahrzeugtypen. Damit sind alle Servicefahrzeuge mit den Unterkategorien: Police, Firefighter, Ambulance gemeint und zuletzt auch Fahrzeuge der Kategorie Taxi.
Um auf eine unbekannte Anzahl an sich zu registrierenden Clients reagieren zu können, muss der initiative Schritt durch den Client erfolgen. Zur Registrierung sendet der Client eine Nachricht mittels MQTT-Protokoll über die Adressen des Users oder der Fahrzeugtypen mit dem ersten Teil ”adresse = hshl/mqtt_exercise/” und der Endung des jeweiligen Fahrzeugtypen, also: ”user,taxi,police,firefighter,ambulance” an den Server. Dieser verarbeitet die Nachricht bei Erhalt in der Funktion

```
def receive()
        ....
        messageprocessing(temp)
        ....
```

wobei die Aufgabe der Funktion *receive()* eher die generelle Verarbeitung der MQTT-Nachricht darstellt und nicht die Zuordnung der Nachricht zu einen bestimmten Anwendungszweck. Dieser wird anschließend durch den Aufruf der Funktion *messageprocessing* und eine Teilung des übergebenen Arrays in die Informationen zu Inhalt und Adresse der Nachricht erreicht. Um ein Auslesen der Nachricht überhaupt zu ermöglichen ist es notwending, die empfangene Nachricht erst in das Json-Format zu codieren, um anschließend die Vorteile einer Verarbeitung mit Json-Datasets zu nutzen. Dies wird mit der Textzeitle

```
def messageprocessing(msg)
        json.loads(str(msg[1]))
```

erreicht.
Zur besseren Identifizierung und Klassifizierung als Registrierungs-Nachricht, wird in dieser die ID des Clients durch das Wort ”register” ersetzt. Dies dient- wie schon erwähnt- einerseits der besseren Einordnung und andererseits half das Lesbar-Halten von Nachrichten für Menschen bei der Entwicklung ungemein. Einen negativen Einfluss auf den erfolgreichen Ablauf der Registrierung des Clients hat dies nicht, da eine ID erst mit Antwort des Servers an den Client vergeben wird. Die Nachricht, die ein Client zur Registrierung/Anmeldung senden muss, sieht wie folgt aus:

```
    data={
    ”id”: ”register”,
    ”name”: name,
    ”coordinates”:coor
    }
```


Identify applicable funding agency here. If none, delete this.

Zudem wird in der internen Verarbeitung ein neuer Kanal für die weitere Kommunikation mit dem Client geschaffen, sodass eine direkte Kommunikation mit diesem möglich ist, ohne dass andere Clients hierdurch beeinträchtigt werden. Die Adresse des neuen Kanals wird unter Zuhilfenahme der durch den Server in den Funktionen *regestrationUser(data)* und *registrationCar(data, type)* vergebenen ID geöffnet und mittels einer Nachricht auf dem Kanal "hshl/mqtt_exercise/user/back" an diesen zurück gesendet.
Anhand des Quellcodes für das Registrieren des Users wird gezeigt, wie die Vergabe einer neuen ID und das Einspeichern des Clients in den Server funktioniert. Dies ist ganz ähnlich für das Vorgeben bei der Registrierung von Fahrzeugen mit dem einzigen Unterschied, dass in diesem Fall mittels einer Separierung durch den Übergabewert "type" die einzelnen Fahrzeuge unterschieden werden können. Des weiteren wird bei der Registrierung der Fahrzeuge noch der Status "free" vergeben.
Zuerst wird durch den Aufruf der Funktion *findid(user)* die kleinste noch freie ID aus der Liste der angemeldeten Fahrzeuge gesucht, indem die höchste vergebene ID gesucht wird und um einen Zähler höher zurückgegeben wird. Anschließend wird in der Methode *registrationUser(data)* durch ein weiteres Durchlaufen der Liste geprüft, ob bereits ein User mit demselben Namen vorhanden ist und bei negativem Ergebnis in die Liste aller User eingetragen.
Der User bekommt abschließend auf dem Rückkanal eine Nachricht mit seiner eindeutigen ID.

B. Bestellen eines Fahrzeugs

Nach erfolgreicher Registrierung ist es für den Kunden möglich, Fahrzeuge zu bestellen und für Fahrzeuge ist es möglich, durch einen Kunden bestellt zu werden. In diesem Fall spielt nun der Server eine verbindende Rolle, indem er eine Anfrage des Kunden entgegennehmen kann und diese an ein von ihm ausgewähltes Fahrzeug weiterleitet. Die Wahl des passenden Fahrzeugs trifft hierbei der Server, da nur er die Positionen aller Teilnehmer kennt und somit das nächstgelegene Fahrzeug auswählen kann.
Eine Anfrage durch einen Kunden wird unter dem im Vorfeld bei der Regestrierung neu geöffneten Kanal in Verbindung mit der ID des Kunden und einer Nachricht mit einem Inhalt, der Informationen über den Kunden, den Koordinaten des Kunden und dem gewünschten Fahrzeugtyp, gestartet. Beispielsweise kann durch den Kunden $ID : 0, Name : Peter, Koordinaten : 2, 4$ unter der $Adresse =$ hshl/mqtt_exercise/user/[ID] mit folgender Nachricht ein Taxi bestellt werden

```
data = {
        "type": "taxi",
    "id": id,
    "coordinates": coordinates
    }
```

Intern verarbeitet der Server die Anfrage des Kunden zuerst, indem er aus dem Type die richtige Liste an die Funktion *findnextcar(gpsUser, car)* übergibt, die das nächstgelegene Fahrzeug des Typs Taxi durch die Koordinaten, die durch den Kunden übermittelt wurden, findet.
Nach erfolgreicher Ermittlung des nächstgelegenen Fahrzeugs wird dem Kunden der Datensatz des Fahrzeugs auf dem Kanal hshl/mqtt_exercise/user/"id"/order/back , wobei "id" die kundenspezifische ID des Antragstellers darstellt und somit nur der Kunde Antwort erhält, der in diesem Moment auch ein Fahrzeug bestellt hat. Als weiteren wichtigen Schritt ist Sorge dafür zu tragen, dass die Einhaltung der Anforderung **F-S02** durch die Anwendung von **F-S08** nicht verletzt wird und nur Fahrzeuge vergeben werden, die den Status "free" besitzen. Das wird erreicht indem der Status eines freien Fahrzeugs durch den Status "busy" ersetzt wird, wenn er an einen Kunden verschickt wird.

1) Finde das nächstgelegene Fahrzeug: Um das Fahrzeug zu finden, das dem Kunden am nächsten gelegen ist, muss zuerst die vorhandene Karte abgebildet werden können. Dies wird erreicht indem Planquadrate eingezeichnet werden. Durch die festgelegten Planquadrate lässt sich anschließend durch Iterieren der Planquadrate und Abgleichen der Liste der Fahrzeuge das gesuchte Fahrzeug finden. So in der vereinfachten Theorie, allerdings entstehen bei der Umschlüsselung auf den konkreten Fall diverse Probleme, die das Finden des nächstgelegenen Fahrzeug und nicht irgendeines Fahrzeugs komplizierter gestalten als zu Anfang angenommen.
dDurch den Umstand, dass der Mittelpunkt der Suche bedingt durch den variablen Standort des Kunden nicht zwingend der Mittelpunkt der Karte darstellen muss, kam als optimale Lösung ein Ring-artiges Erweitern der Suche um den Standort des Kunden infrage. Dieses wurde mittels Schleifeniteration geschaffen.
Die erste Schleife mit der Laufvariable "k" gib an, um welchen Ring um den aktuellen Standort des Kunden es sich handelt. Die zweite Schleife steht für die y-Achse der Karte und gibt somit die Reihen der Quadrate an. Sie iteriert von einem Startpunkt an, der sich auszeichnet durch die Gegenzahl von der Ringstufe k minus einem Zähler $i_{starpunkt} = (k-1)*(-1)$ bis zu dem Endpunkt der durch k plus einem Zähler beschrieben werden kann $i_{endpunkt} = k + 1$. Die x-Achse wird durch einen weiteren Schleifendurchlauf abgebildet und spiegelt die Spalten der Karte wider. Die x-Achsen-Schleife läuft vom Startpunkt $j_{starpunkt} = i$, also der linken oberen Ecke des aktuellen Rings, bis zum Endpunkt $j_{endpunkt} = k+1$, also der rechten unteren Ecke des aktuellen Rings. Eine letzte Schleife vergleicht letztendlich die angegebene Position mit der Liste der angemeldeten Fahrzeuge ab, um zu prüfen, ob sich an der aktuellen Koordinate ein Fahrzeug befindet.

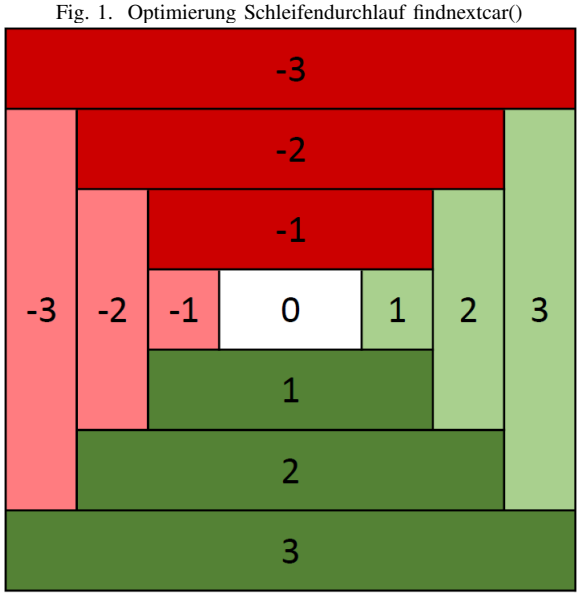
```
def findnextCar(gpsUser,car):
    cancle = 0
    for k in range(int(gpsUser.split(";")[0]),5):
        for i in range((k-1)*(-1), k+1):
            print("i ist:"+str(i))
            for j in range(i, k+1):

                print("j ist:"+str(j))
                for c in range(0,len(car)):
                    print("search for car: "
                                + str(car[c]))
```

...

Ein Nachteil den der beschriebene Code innehat ist allerdings, dass bei jedem Durchlauf nur um einen Ring erweitert wird. In Wirklichkeit wird hingegen nicht ein Ring geprüft, sondern ein Quadrat jedes mal um einen Ring erweitert. Dies hat zum Nachteil, dass bei jedem Durchlauf auch jedes mal bereits geprüfte Felder ein weiteres mal geprüft werden, was einen deutlich hemmenden Einfluss auf die Laufzeit des Programmes hat und letztendlich auch ein überflüssiges Unterfangen ist. Demnach ist bei einer möglichen Optimierung darauf zu achten, bereits geprüfte Felder nicht ein weiteres Mal zu überprüfen. Eine mögliche Umsetzung wäre, die vorhandenen Schleifen für die x- Achse und die y- Achse in jeweils zwei Schleifen aufzuspalten, wobei in diesem Fall die erste Schleife den negativen Bereich auf der linken und oberen Seite des Standortes prüfen würde und die zweite Schleife den rechten und unteren positiven Bereich der Karte. Anschließend

Fig. 1. Optimierung Schleifendurchlauf findnextcar()



wird als letzter Schritt innerhalb der letzten Schleife noch abgeglichen, ob sich ein freies Fahrzeug mit den passenden Koordinaten auf der entsprechenden aktuell geprüften Position befindet, beispielhaft dargestellt für den Bereich $-y, +x = \{-, +\}$. Aus der Tabelle I-B1 kann in Verbindung mit den aktuellen Schleifen-Variablen die aktuell zu prüfende Position bestimmt werden, wie in Beispiel Positionsabgleich zu sehen. Zudem wird noch sicherheitshalber ein Counter erstellt, der einen exit nach drei Durchläufen ohne Ergebnis ermöglicht.

```
#Bespiel Positions-Abgleich#
...
elif int(car[c][2].split(";")[0]) == -i
and int(car[c][2].split(";")[1]) == j:
        if car[c][3]== "free":#Requirements: F-S11
        return car[c]
    else:
        c=c-1
        cancle = cancle+1
```

TABLE I
ABGLEICH-TABELLE FÜR DIE POSITIONSBESTIMMUNG DER FAHRZEUGE

–,–	–,-	–,=	–,+	–,++
-,–	-,-	-,=	-,+	-,++
=,–	=,-	=,=	=,+	=,++
+,–	+,-	+,=	+,+	+,++
++,–	++,-	++,=	++,+	++,++

Als Rückgabewert wird der Datensatz des Fahrzeugs, das dem Kunden am nächsten liegt, an den Aufruf zurück gegeben. Dieser kann anschließend an den Kunden übermittelt werden.

C. Ankunft am Ziel

Für den Server ist durch die Vermittlung des nächstgelegenen Fahrzeugs an den Kunden der Hauptteil seiner Aufgabe erledigt und das Primärziel erreicht. Demnach wurde dem Kunden auf effiziente Weise ein Fahrzeug vermittelt und im Falle einer Buchung eines Service-Fahrzeugs im Optimalfall sogar Leben gerettet. Allerdings stehen nicht unbegrenzt Fahrzeug-Ressourcen zur Verfügung, weswegen eine einfache Nutzung dieser im realen Umfeld logischerweise niemals Anwendung finden würde. Deswegen ist es zwingend erforderlich, eine Möglichkeit zu bieten, Fahrzeuge fortlaufend an Kunden zu vermitteln.
Da aufgrund der zu Beginn des Projektes beschriebenen Systemumgebung der Server seinen Einfluss an den georderten Fahrzeug mit der Übermittlung der Fahrzeugdaten zum Kunden an diesen die Kontroll-Gewalt abgibt [**?**], [**?**, ?] ist es dem Server erst durch Initiative des Kunden möglich, Fahrzeuge wieder in den Status "free" zu nehmen. Dieses Verfahren bietet auf eine erste logische Betrachtung die Vorteile, dass solange der Kunde nicht aussteigt, ein Fahrzeug auf keinen Fall frei werden kann. Dies kommt der Praxis sehr nahe und bildet deswegen sehr gut die Realität ab. Ein Nachteil für dieses Vorgehen ist allerdings, dass durch die Abgabe der Kontroll-Gewalt der Server keinen Einfluss mehr auf das Fahrzeug hat, solange es nicht durch die Nachricht des Kunden freigegeben wird. Sollte nun der Kunde aussteigen und keine Nachricht an den Server senden, oder sollte diese verloren gehen, so wird dieses Fahrzeug auf unbegrenzte Zeit als belegt verbucht bleiben und somit nicht genutzt werden können.
Eine Lösung des Problems wäre ein Timer, der nach einer bestimmten Zeit eine Anfrage an jedes belegte Fahrzeug sendet, ob dies immer noch durch den Kunden belegt ist. Des Weiteren könnte eine Rückanmeldung eines Fahrzeugs auch auf Nachricht des Fahrzeugs umgesetzt werden, sodass das Fahrzeug sich in dem Moment des Freiwerdens beim Server zurück anmeldet. Allerdings würde der verwendete Kerncode auch in beiden Optimierungsfällen ähnliche Funktionen aufweisen.
Eine Nachricht zur Freigabe eines Fahrzeugs wird mittels einer Nachricht über den Kanal "hshl/mqtt_exercise/user/"id"/status/reset" und dem Inhalt "type" , "id" des Users und der id des Fahrzeugs "idCar" an den Server gesendet. Dieser

ist nun in der Lage durch den Aufruf der Funktion $statusReset(int(js[”idCar”]), str(js[”type”]))$
mittels Fahrzeugtyp und dessen ID, dem benutzten Fahrzeug wieder den Status ”free” zuzuordnen, ein weiterer wichtiger Schritt wird mittels Funktionsaufruf $requestPosition(js[”idCar”], js[”type”],$
$str(findcarname(js[”idCar”], str(js[”type”])))$
vollzogen. Angefragt wird der aktuelle Standort des freigegebenen Fahrzeugs, um in fortlaufenden Buchungen wieder die Möglichkeit zu bieten, dem Kunden sein nächstgelegenes Fahrzeug zu Verfügung zu stellen. Demnach sendet der Server dem gerade freigegebenen Fahrzeug über den Kanal ”hshl/mqtt_exercise/get_position” eine Anfrage mit dem Inhalt ”id” des Fahrzeugs und dem dazugehörigen Namen und erhält im besten Fall eine Antwort über den Kanal ”hshl/mqtt_exercise/set_position mit Informationen über ”id” des Fahrzeugs, seines ”typs” und zuletzt der aktuellen ”coordinates”. Diese Informationen werden dann in der passenden Liste des Typs für das aktuelle Fahrzeug aktualisiert.

```
data = {
”type”: cartype,
”id”: id,
”idCar”: idCar,
}
```

D. Periodische Abfrage der Fahrzeugposition

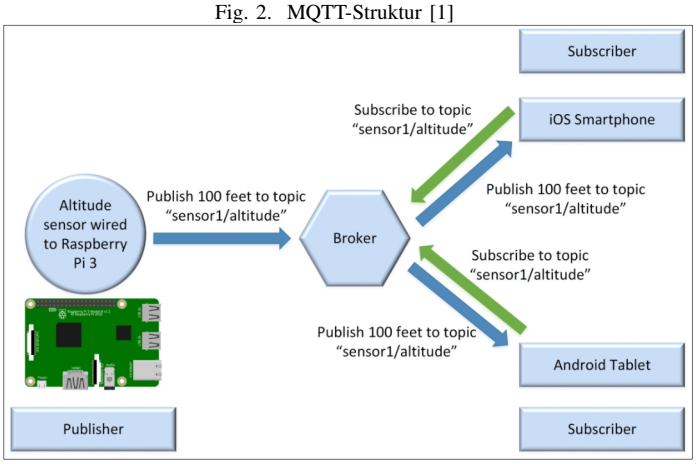
In der Realität stehen Fahrzeuge wie Taxen oder Polizeifahrzeuge selten still. Aus diesem Grund ist die Wahrscheinlichkeit ziemlich hoch, dass diese Fahrzeuge nach ihrer Anmeldung im Laufe der Zeit ihre Position verändern. Deswegen erscheint es nur sinnvoll, in regelmäßigen Abständen deren Position zu kontrollieren- beziehungsweise abzufragen- um immer zu garantieren, dass das nächst gelegene Fahrzeug ausgewählt werden kann.
Problematisch ist hierbei, dass die Menge an zu überprüfenden Fahrzeugen einen erheblichen Zeitaufwand in Anspruch nimmt und somit nicht so schnell wie möglich auf Anfragen eines Clients reagiert werden kann, oder im schlimmsten Fall sogar Nachrichten verloren gehen können. Aufgrund dessen ist es notwendig, die Abläufe von Nachrichtenverarbeitung und -abfrage der Position voneinander zu trennen. Genutzt wurde hierfür die Möglichkeit, mittels Python Erweiterung **threading** auf Multi-Threading zu setzen und somit die Aufgaben in zwei verschiedenen Prozessen aufzusplitten. Dies hat zwar zur Folge, dass die Auslastung des ausführenden Systems ein wenig höher ist, allerdings sind selbst Einplatinen-Computer schon seit Längerem mit Mehrkern-Prozessoren ausgestattet, was jegliches bedenken bezüglich der Auslastung ihm Rahmen dieses Projektes hinfällig macht.

E. Verbesserungen

II. MQTT

Das **Message Queuing Telemetry Transport**, oder kurz **MQTT**, ist ein Protokoll, welches für die Kommunikation zwischen Maschinen bestimmt ist. Somit eignet sich dieses Protokoll auch im Wesentlichen für seinen Einsatz im Anwendungsbereich der Automatisierung und ganz besonders im Bereich des Internet der Dinge (**IoT** Internet of Things). Da aufgrund seines Aufbaus Geräte mit wenig Akkukapazität so gut wie keine eigene Rechenleistung erbringen müssen, um Nachrichten zu empfangen oder senden zu können, ist vor allem bei Akku-betriebenen Geräten ein Vorteil zu erkennen. Erreicht wird dies durch den generellen Aufbau des Protokolls mittels Subscriber/ Publishern und einem zentralen Broker, wobei der Broker der verwaltende und Daten haltende Part ist und einem Server nahe kommt. Zudem gibt es die Subscriber, die Nachrichten empfangen können- ein Beispiel hierfür wären Aktoren. Die Publisher hingegen kommen zum Beispiel als Sensoren zum Einsatz, sie teilen der Netzwerkstruktur ihre Informationen mit. Die Einordnung als Subscriber oder als Publisher ist aber in keinem Fall exklusiv, sodass Geräte auch in einem hybriden Modus sowohl versenden als auch empfangen können, das wäre zum Beispiel bei einem Smart Home-Server der Fall.
Ein ganz wesentlicher Vorteil des MQTT-Protokolls liegt in seiner Struktur als Publishing and Subcribe Verfahren. Dadurch, dass das Senden der Nachrichten nicht als Direktverbindung der Clients untereinander, sondern mit dem Umweg über den Broker geschieht, sind die Geräte in der Lage, Nachrichten sicher zu empfangen, auch wenn sie gerade nicht auf Nachrichten-Empfang stehen. [1] Jedoch bringt

Fig. 2. MQTT-Struktur [1]



dieses Verfahren auch einen gravierenden Nachteil mit sich. Aufgrund dessen, dass Nachrichten nicht in einer eins-zu-eins-Übertragung miteinander geteilt werden, müssen Nachrichten auf eine andere Art und Weise zu ihrem Ziel gelangen. Fatal wäre hier zu glauben, auf eine Zuordnung der Nachrichten verzichten zu können. Da gerade in einem Smart Home viele ähnliche Geräte verbaut werden und somit auch viele ähnliche Werte versendet werden, wäre eine einfache Zuordnung nun nicht mehr ohne weiteres möglich. Abhilfe schafft im MQTT-Protokoll ein Nachrichtenfilter mit einer Art Kanäle, die die Geräte ”abonnieren” können, um nur relevante Daten zu erhalten [1]. Diese Kanäle haben Ähnlichkeit mit Adressen, geben Filter und Subfilter für die Nachrichtenverteilung an

und werden nach dem Schema: Erdgeschoss/Rolladensteuerung/Fenster1/Motor erstellt, um ein Beispiel für die Ansteuerung des 1. Fensters im Erdgeschoss zu nennen.
Vorteile bietet MQTT des weiteren dank der Möglichkeit, das IP-Protokoll und somit TCP und TLS zu verwenden, was der Nachrichtenübertragung einen gewissen Schutz gegen das Verlieren von Nachrichten bietet, was gerade in kritischen Systembereichen einen Mehrwert hat. Und dank der Verschlüsselung mittels TLS bietet es einen Schutz gegen das zugreifen von unbefugten.

REFERENCES

[1] Gastón C. Hillar, MQTT essentials, a lightweight IoT protocol : the preferred IoT publish-subscribe lightweight messaging protocol, Packt Publishing, Birmingham, United Kingdom, 2017