Towards a global traffic control (dispatcher) algorithm - interface prototype design

Jonathan Beebe

Jonathan Beebe Ltd., UK, jonathan@jonathanbeebe.com

Abstract. This paper presents an overview of the design and development of a prototype Global Dispatcher Interface (GDI) for the control of a group of lifts. The role of the dispatcher is to assign passenger calls to the optimal lift in a group, as decided by a dispatcher algorithm. The GDI is independent of the underlying algorithm, which may be distributed remotely, and provides a standard means through which all interactions with the dispatcher may occur. To warrant the "Global" appellation the GDI must support any of the currently available, as well as anticipated, call station modes, types and configurations of cars, topology of control equipment and buildings. The design process is a continuation of a recognised Software Development Lifecycle, centred on Use Cases in a UML model, the initiation of which is covered in a previous paper. Significant diagrams from the model are presented and discussed to illustrate the evolution of the prototype design. One of the requirements, resulting from analysis of the Use Cases, identifies that the GDI design must be compatible with a publish-and-subscribe architecture and a RESTful interface is selected for this purpose. Where possible, the prototype design uses open standards with an emphasis on demonstrating those aspects that are specific to lift system dispatcher operation, while attempting to demonstrate independence from implementation details such as programming language, network protocols, etc. The Standard Elevator Information Schema is particularly relevant and fulfils these objectives. The operation of the working prototype, in conjunction with simulated lifts and passengers, is presented as a validation of the design.

Keywords: Global dispatcher, Standard Elevator Information Schema, group control, design, prototype, REST, API, UML

1 INTRODUCTION

While definitions of standard group control algorithms have been documented [1], the reality of group control to date is that manufacturers have created proprietary designs that are inextricably linked to their own lift equipment. The result is that it is not possible accurately to compare or predict the effects on performance of different control policies during the design phase of a building, or in advance of a refurbishment of the lifts. The benefit of a standard interface is that it would make it possible to supply the dispatching capability in a component form that could be "plugged" into any group of lifts that conforms to the interface.

Secondly, a dispatcher design which has been configured and validated using simulation can be transferred directly into a physical installation with confidence, if both the simulator and the real lifts use the same standard interface.

Additionally, as lifts become better integrated with the other services of so-called "smart buildings" and with the introduction of applications that allow passengers to register requests for lift travel via a variety of channels [2] including personal mobile devices, an interface that allows simplified and standardised secure access to the group call assignment mechanism becomes increasingly desirable.

A previous paper [3] analysed the requirements for a Global Dispatcher Interface (GDI) via which a group of lifts could be controlled. Current trends and possible future developments in lift group controller technology were reviewed so that the identified requirements are sufficiently broad and flexible to avoid the analysis becoming prematurely outdated. The paper presented a structured statement of the requirements that a GDI must satisfy, followed by an analysis of those

requirements using the Requirements Capture and subsequent Analysis phases of a light-weight Software Development Lifecycle (SDLC) [4]. The outputs of these initial phases of the process are

- Passenger (user perspective) use cases.
- Dispatcher (system perspective) use cases
- Requirements catalogue
- Domain object catalogue

These outputs take the form of a UML Model plus supporting report documentation generated from the model and, because of their number and complexity, are published separately from the paper, which can only present the key features and conclusions. The report documents can be found at the project website [5]. The model has been developed and is maintained via a specialized tool[6] which supports the entire SDLC.

The current paper continues the SDLC process with a discussion of the design and development of a functioning prototype. By definition, a software prototype [7] is not intended to be deployed in a live situation serving real users (i.e. passengers, maintainers, managers), rather it is intended to demonstrate the viability of delivering a variety of key functional capabilities, while other characteristics may be only partially implemented or completely omitted. At the conclusion of prototype evaluation, the software should be archived and the design and development phases should be completely reiterated but from that point onwards, with the additional requirements of security, performance, robustness and cost fully accounted for in the design.

In addition to the GDI, the prototype demonstration system consists of a configurable simulator (a commercial product [8]) of passengers and of lift car activity. New gateway software has been developed to provide a more realistic representation of lift and call registration activity, which in real life (as opposed to a simulation program) are enacted as independent asynchronous activities.

Whilst the subject of the current paper is the design of the GDI prototype, the discussion includes some details of the operation of the prototype lift and landing call gateway software, where it is helpful to illustrate the sequence of intended interactions with the interface.

An important point, presented in [3], relates to the preferred use of open standards to provide the generic hardware and software, which are of themselves not specific to lift systems. Thus the discussion can concentrate on those considerations which are specific to lift systems. In response, the GDI prototype sets out to demonstrate the delivery of dispatching functionality supported by an infrastructure built of as many interoperating, heterogeneous and open technology standards (e.g. programming language, network protocol, etc) as it is practicable to include.

2 SOME RELEVANT TERMS

The current paper makes frequent use of terms that are relevant in the domain of lift control:

- Landing call (LC)
- Cars
- Algorithm

These and other terms are defined in Section 2 of [3] however two key terms are copied here:

Standard Elevator Information Schema (SEIS) [9] – is a standard for communicating static information, such as configuration details, dynamic information such as current floor and registered calls and events such as call registrations and car trips, between all manner of systems and users of passenger lifts. It comprises a set of definitions of complex and simple data types and the structures in which they may be used. This paper makes frequent references to the schema, both for data types of parameters passed in messages and also for the internal data structures of the dispatcher interface,

and these are indicated by text in CamelCase, which (for the digital format of this document) includes a hyperlink to the definition on the website where the schema is published.

Global Dispatcher Service – is a standard, non-proprietary mechanism for assigning a landing call to the lift car most suited to serving that call. It therefore includes an element - the "dispatcher" - which can produce the optimal assignment decision. The GDI encapsulates the dispatcher and is the route by which all access to the Global Dispatcher Service is made.

3 GDI ANALYSIS

During the analysis phase of the SDLC, the detailed description text of the system use cases (the use case "story" or "flow") provides the basis for developing a more detailed diagrammatic definition of the sequences of interactions that must occur between the collaborating domain objects. This is achieved through the development of sequence diagrams which elaborate the messages passed between the objects as the use case proceeds. A separate sequence diagram [10] is developed for each significant alternative route ("scenario") [11] through the use case (often the result of different outcomes from an If-Then-Else like decision). For example:

> *"When the assigned car arrives at the call origin floor, if the dispatcher has not been informed of the passenger's destination floor the passenger's call is then deleted from the list of current calls. However, if the destination floor is known then the call is retained but its status is changed to "Answered".*

While there was insufficient space in [3] to include all of these sequence diagrams, they are available at [5] and an example is included for illustration in the following section. It is then through the elaboration of sequence diagrams that the design phase of the SDLC can be commenced. During the design phase further sequence diagrams are produced but now showing the collaborations between the software components which will be implemented rather than abstract domain objects. This paper is concerned with the design of the Global Dispatcher Interface only but the diagrams also consider the operation of the dispatcher itself to ensure that all of the dispatcher requirements are supported by the interface. The full set of design sequence diagrams is maintained at [12].

3.1 Example of Analysis Phase System Use Case - Assign Call

3.1.1 Assign Call -Sequence Diagrams

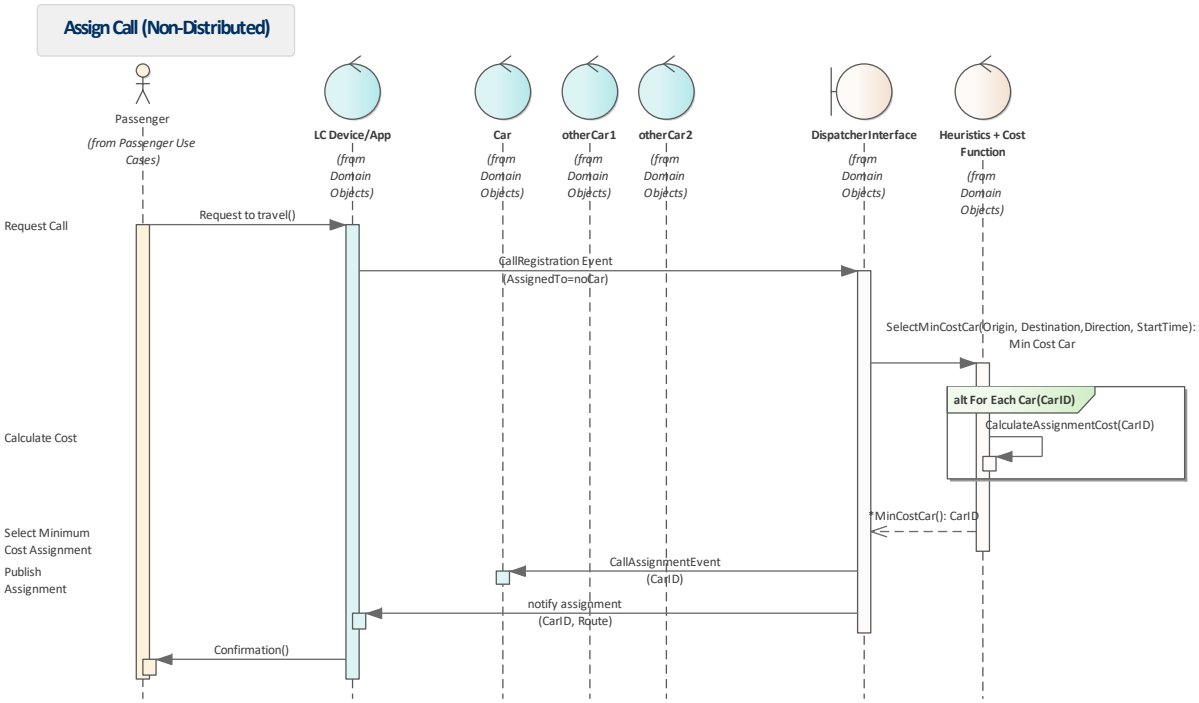


Figure 1 Assign Call (Single Dispatcher)

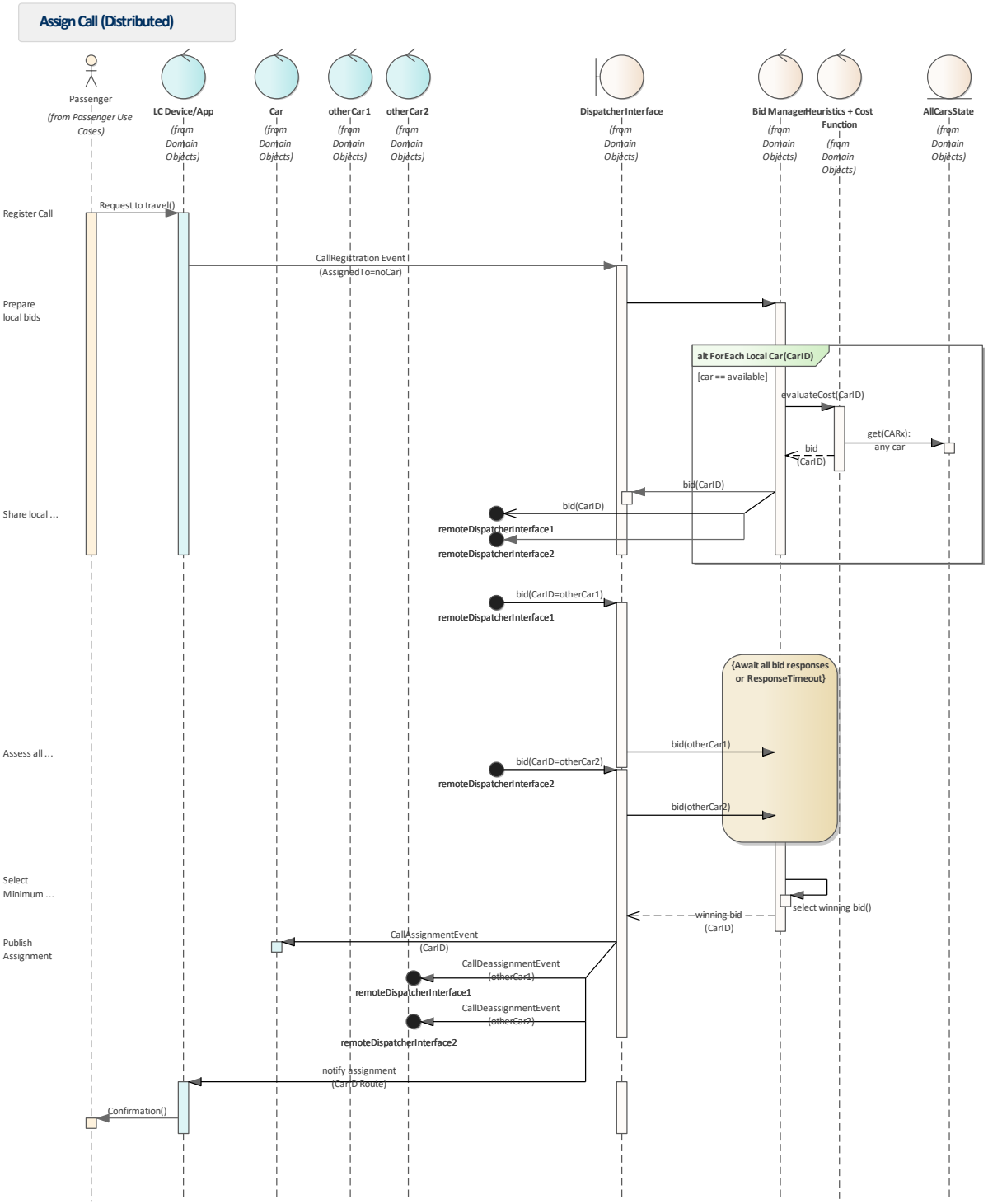


Figure 2 Assign Call (Scenario 1 - Distributed dispatcher)

3.1.2 Assign Call – Use Case Story (detailed description)

The paragraphs of this section are an extract of the description text of the Assign Call system use case. They outline the flow of different Scenarios (non-distributed and distributed), and Alternative Path 1, which covers exceptional behaviour (acknowledgement not received).

3.1.2.1 Register Call

Any client (including a passenger signalling device) may make a request to travel. The request includes the following information:

- call floor
- call direction or destination floor
- registration time

in the form of a CallRegistration event.

N.B. *Inclusion of registration time means this request may (re)occur at any time after the initial call registration event and may therefore be a request for a call to be re-assigned or simply a delayed request if no car was available for assignment earlier. In the case of re-assignment the dispatcher will have a record of the currently assigned car and this information may influence the result of the new assignment, though the decision would be a characteristic of the specific dispatcher algorithm and not of the dispatcher interface.*

At the discretion of the specific dispatcher algorithm (not the dispatcher interface) an Assign Call request may initiate the re-assignment of other previously assigned calls, which will result in multiple executions of the Assign Call use case (i.e. once for each call to be re-assigned).

3.1.2.2 Calculate Cost

The dispatcher algorithm calculates the cost of assigning the call to each of the registered and available client cars according to its own internal algorithm design. The term "cost" is not restricted to a purely financial cost and may be evaluated in terms of one or more criteria such as

- waiting time,
- system response time,
- energy consumed,
- etc

as a function of the increase in the value of that parameter after the call has been assigned compared to the cost before it was assigned to the car. The algorithm may include penalties or incentives that are derived from a logical analysis which will modify the simple cost.

If an overriding criterion is included in the algorithm such as never assigning a call that would cause the car to become overloaded then that car will be marked as "blocked" for this calculation.

N.B. *Car availability for assignment may be determined by a variety of properties such as:*

- *the operating mode of the car*
- *whether the car is able to service the call floor(s)*

However, the availability decision is a characteristic of the specific dispatcher algorithm and not of the dispatcher interface.

3.1.2.3 Scenario 1 - Distributed Dispatcher

The dispatcher may be distributed as a number of collaborating instances, each responsible for a unique set of one or more registered client cars. In this case each dispatcher instance will respond with a "bid" for each client car, which is the cost of assignment for adding the call to the travel plan of the car. A period is defined (internally by the dispatcher service) during which bids may be made.

3.1.2.4 Select Minimum Cost Assignment

The available car offering the minimum cost of assignment (bid in the case of a distributed dispatcher) is selected as the assigned car.

3.1.2.5 Publish Assignment

The selected minimum cost assignment is broadcast via the dispatcher interface to all registered clients and includes (but is not limited to):

- the CallRegistration call event updated with the cost of assignment, where cost analysis is made in terms of the cost-function that is specific to the algorithm used by the dispatcher

plus optionally:

- a CallAssignment event where the AssignedTo element is populated with the minimum-cost assignment details.
- a TravelPlan if the dispatcher is configured to hide assignments so that the car will by-pass an assigned call until it becomes the next landing call for the car.
- if the call was previously assigned to a different car the response will also include a CallDeassignment event.

3.1.2.6 Alternative Path 1 - No Acknowledgement

The Assignment is only considered to be complete after a positive acknowledgement has been received from the dispatcher responsible for the assigned car. If no acknowledgement or a negative acknowledgement is received from an assigned client car then the call assignment will be repeated with that car excluded.

3.2 Analysis Phase System Use Case – Cancel Call

A complimentary sequence diagram and use case story has been developed for the Cancel Call system use case and can be viewed in the GDI Analysis report [5]. This use case is initiated when the car approaches the call floor with a registered call for the floor. The use case offers alternative paths for both direction and destination landing calls as well as car call cancellations. However, the description text is omitted here for brevity.

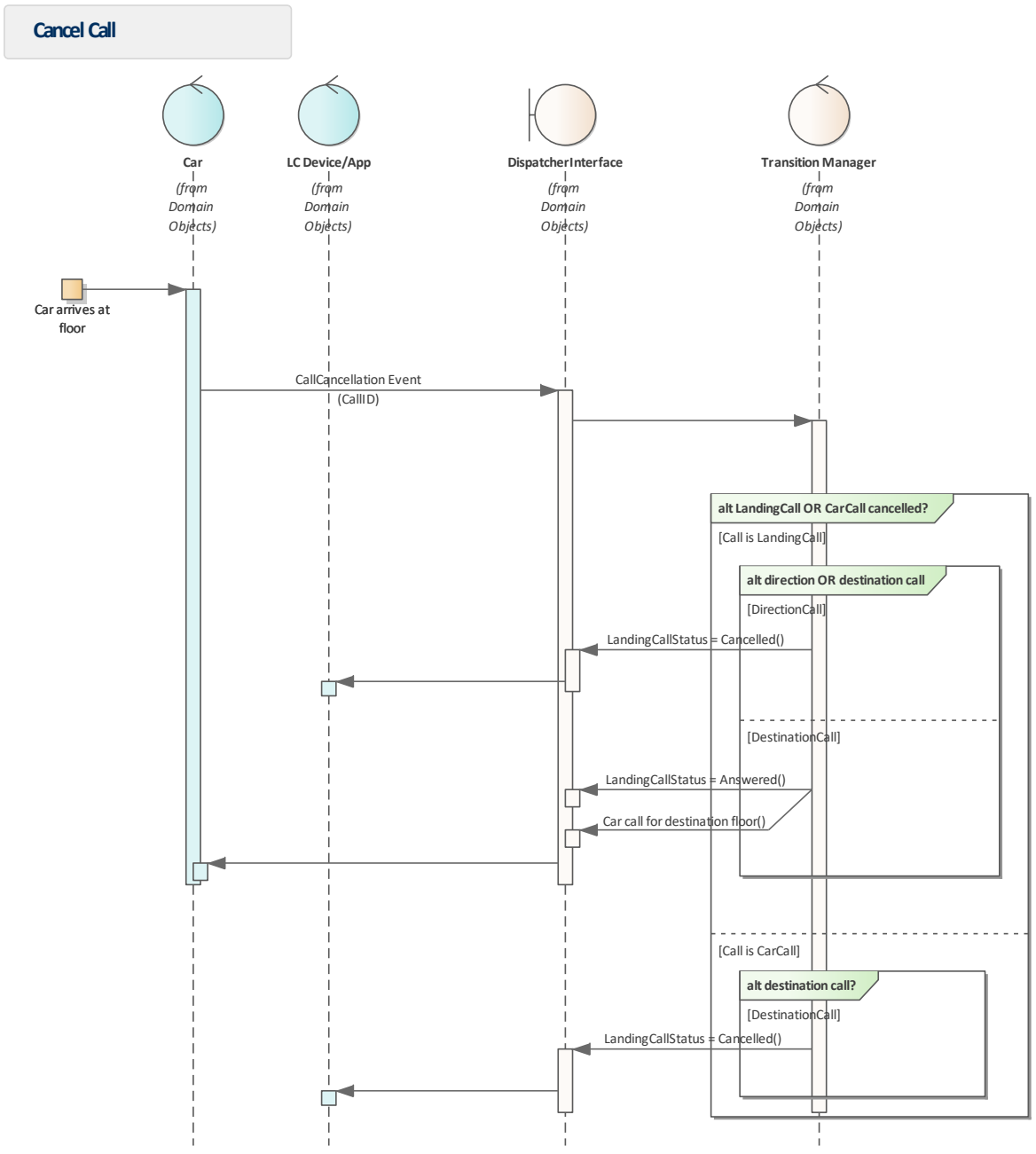


Figure 3 Cancel Call

4 GDI DESIGN

The outputs of the design phase of the SDLC are:

- Sequence Diagrams
- Class library definitions

These will be the necessary inputs for the subsequent software development phase – in this case development of the prototype.

4.1 Design Requirements

During the elaboration of the sequence diagrams some further design requirements are identified.

4.1.1 Publish and Subscribe Architecture

It is clear from the analysis sequence diagrams the assignment resulting from a request to travel must be communicated not only to the source of the request (LC device/application) but also most importantly to the assigned car (and possibly to all other cars as well). Until the assignment is made, the cars are unaware that a call has been registered. So a mechanism is required that will notify the car(s) without them having to continually poll the dispatcher service "just in case". This mechanism is provided by the Publish-And-Subscribe[13] messaging pattern. With this pattern, any number of active elements (cars, LC devices, etc) may request (usually, though not uniquely, during start-up) the GDI to publish a list of observable information sources against which they may submit a subscribe request. The GDI will subsequently send a message to all subscribers each time an event (usually a change of state) occurs associated with the information being observed (originally described as the Observer software design pattern[14]).

4.1.2 Dispatcher Interface as a "Notice-Board"

An examination of the system use case sequence diagrams reveals that the messages being passed to and from the GDI describe information events that are defined in terms of SEIS, which is a schema that defines an information model. The GDI acts like a central notice-board where the elements of the lift system simply post their current status, under specific subject headings. Coupled with the publish-and-subscribe architecture, it becomes like a social-media notice-board where subjects of interest can be "followed" (ie subscribed to). This is a very important property of the GDI since none of the lift system elements is assuming to understand or maintain expectations of the operation (or even the existence) of any other element which may receive its messages. In software engineering this characteristic is referred to as 'separation of concerns'[15].

The resulting interface is compatible with any dispatcher algorithm technique from dynamic sectoring, to neural networks based on cost functions and therefore the inevitable debate is avoided about which parameters must be passed in any call to the interface.

This mode of interaction allows an enormous amount of flexibility in the configuration and component architecture of lift systems which may use the GDI. Figure 4 illustrates some of the many possible configuration options.

Algorithm Behind Interface

The use case sequence diagrams in section 3.1.1 show the cars and landing call devices sending messages to the dispatcher interface with the dispatcher algorithm (Heuristics + Cost Function) located "behind" the interface, implying a simple function call from the algorithm to the information maintained by the dispatcher interface. This configuration supports Master/Slave and Simple Hierarchy interaction modes mentioned in [3].

Algorithm as Subscriber

In some configurations it may be more appropriate for the algorithm to be implemented simply as another subscriber to the dispatcher interface (ie the Notice-Board) leaving nothing "behind" the interface. This particular configuration allows the implementation of the distributed dispatcher (as illustrated by Figure 2) and supports Assignment Bidding, Master/Slave and Simple Hierarchy interaction modes mentioned in [3].

All Functionality Behind Interface

On the other hand in some circumstances it may be preferred to have all elements of the lift system control software implemented as a single software component that sits "behind" the interface. In this case the interface would operate simply as a reporting mechanism via which data logging and status monitoring equipment could be connected. This configuration may support Master/Slave and/or Simple Hierarchy interaction mode but this aspect will depend on the particular implementation and the result will be proprietary.

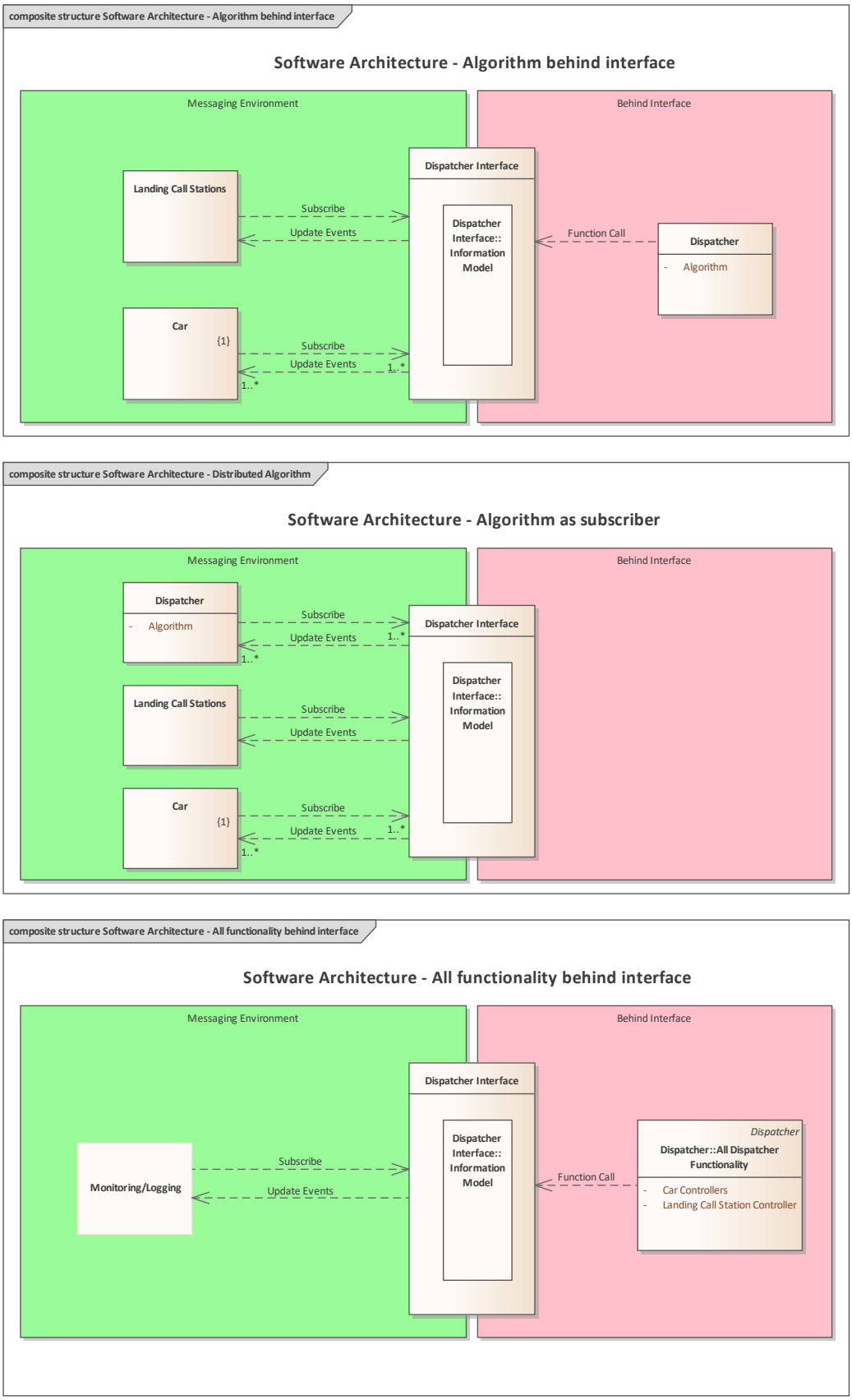


Figure 4 Some Software Architecture Configurations

4.1.3 RESTful Interface

It has already been noted that messages to and from the GDI represent information events that are defined in terms of SEIS. Therefore the messages do not make calls to specific functions of the dispatcher, or any other aspect of operation of passenger lifts. Instead, the same set of standard generic functions (called "methods") can be requested from each supported node of the information model.

These functions might be implemented, for example, as a 'RESTful'[16] interface (using a variety of available programming technologies and languages). Each node in the SEIS information model against which the GDI exchanges event messages, becomes a networked resource (URI)[17] potentially offering the same (though not necessarily all) four methods:

- POST
- GET
- PUT
- DELETE

Furthermore, nodes which have a multiplicity greater than 1 (i.e. lists) must support queries using the properties and referenced links of the node.

N.B. *The interface stores and retrieves the current state of the elements of the information model in a similar manner to a relation database with tables having stored procedures, where the functions would be named Create, Read, Update and Delete (CRUD)[18].*

Another valuable characteristic of REST is its independence of any network topology (e.g. proxies, gateways, firewalls, etc) so it is scalable. If required, a single instance of the GDI might therefore support a number of groups of lifts, located in the same building, across a campus or a further distribution where the dispatcher might be made available via the Internet as a cloud service.

The GDI prototype complies with all 6 REST constraints – see [16]. Additionally, access permissions to each published node of the SEIS information model must be considered:

4.1.4 Access rights

The GDI should implement an overall security policy to restrict access (including subscription) to authorised clients only. Access rights will be established during execution of the Registration use case but this is not discussed further in the current paper since it is not specific to the task of dispatching of lifts.

4.1.5 Create/Update access restricted to "owned" resources

The ability of a client to create and update resources via the GDI is limited to those resources that are "owned" by the client. Thus a car may update any attributes of its own CarDynamicData or CarStaticData but not those of another car.

Landing call devices may create (POST) a new LandingCall in the list but the resulting LandingCall is owned by the dispatcher. A car may create/update (PUT) its own Bid element in a LandingCall.

It is a matter of internal design of the dispatcher whether LandingCalls are deleted or retained when their Status becomes Cancelled and should be considered as part of the greater discussion of data logging and retention[2].

5 PROTOTYPE DEMONSTRATOR

The purpose of the prototype is to demonstrate and validate the ability of the GDI design to support the functionality that is particular to the task of lift system dispatching. To achieve this objective it is necessary to build a complete and realistic environment in which the dispatcher interface can operate. Such an environment needs to have access either to more than one real groups of

operational lifts or to a variety of configurations of simulated groups, though with the current prototype only one group will be active at a time. The different configurations allow the prototype to be driven by both direction and destination passenger call stations and to demonstrate different numbers of cars and patterns of passenger demand and floors served.

More general technical considerations, such as network performance, security, robustness, etc., are addressed only in as much as it is necessary to achieve a realistic and operational prototype. A thorough examination and design of such attributes must be addressed during an eventual commercial product development, when inevitably, the currently available technologies and standards are likely to have evolved or been superseded. The prototype environment is illustrated in Figure 5.

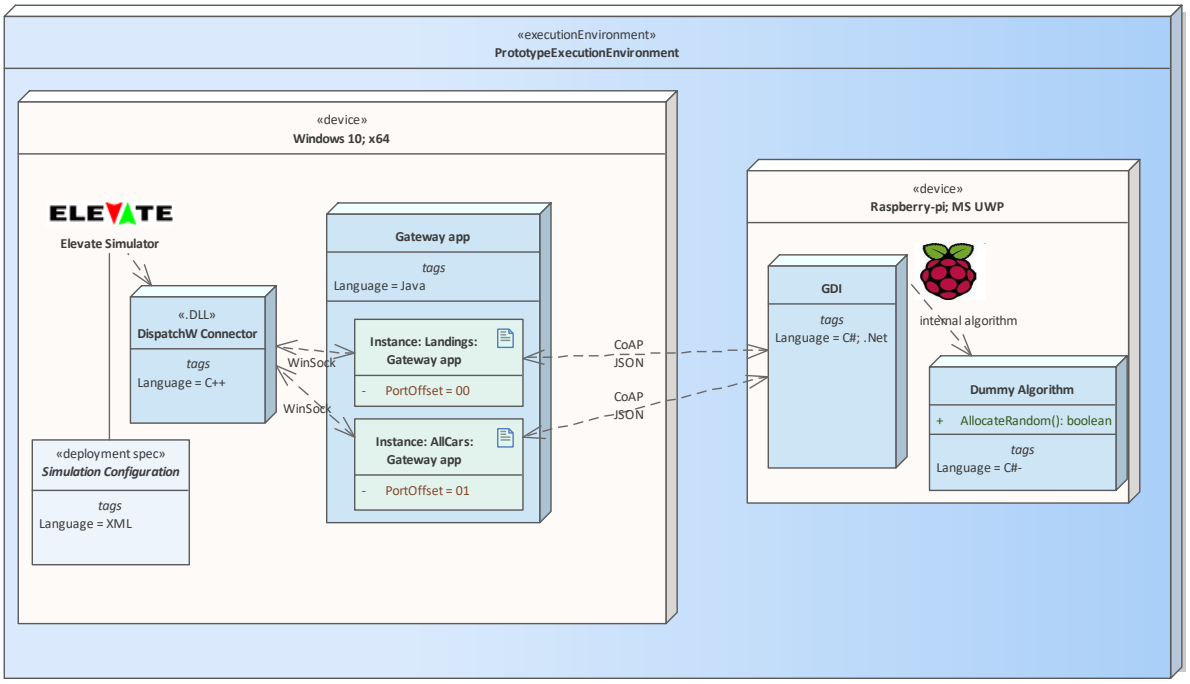


Figure 5 Prototype Implementation

In an attempt to demonstrate that the prototype design is not dependent on specific technologies, the selection of network protocols, software frameworks, programming languages, computer hardware and operating system environments has been chosen to be as diverse and heterogeneous as possible.

5.1 Lift System Simulator

For this prototype, an accurate lift system simulation [8] of lift cars, passengers and call stations has provided a realistic, flexible and permanently accessible solution.

The simulator allows car movement and door operation to be represented accurately and reported against simulated time. Many different passenger demand profiles may be investigated, with the possibility of passenger calls being registered through direction or destination call stations and including a mixture of either type at different landings in the same building. Furthermore, and of great importance to the prototype, the dispatcher algorithm, which controls the assignment of landing calls may be specified as being provided via a system that is external to the simulator.

The following sections describe the various software components that have been developed to complete the prototype environment.

5.2 Simulator Connector .DLL

The simulator provides the option to connect an external dispatcher algorithm via a single function call to an external user-defined software component, where all relevant information is passed as parameters of the call. The call is executed once during each simulated time interval.

However, for the purposes of demonstrating the prototype GDI it is more realistic to create the impression of cars and call stations operating as independent, asynchronous processes and communicating with the GDI through separate channels. Therefore, in this case the user programmable connection software ("DispatchW Connector" component in Figure 5 - a Microsoft Windows .DLL) has been developed which doesn't itself contain the algorithm but instead simply splits the information from the simulator function call, according to its subject matter, into separate streams of data events (using Microsoft sockets API – WinSock[19]). Each stream is allocated a different network port so that it appears to be communicating data from a separately connected device. The prototype demonstrates this by splitting landing call station events into one stream (port offset=00) and all car related events in a second stream (port offset=01). However, it is equally possible, with no changes to the software code, to deploy a separate instance of the same Connector component for each car (having port offsets = 01, 02 .. number of cars).

5.3 Landing Call-station and Car Gateway Application

In a further attempt at realism, a Gateway software application has been developed to undertake a variety of transformations of the data events received from the simulator and similarly for information being returned in the opposite direction. In order to demonstrate the "global" applicability of the GDI we must consider that any part of the lift equipment interacting with it may not be able to produce the necessary information, at the appropriate time or in a suitable format. It may be that some manufacturers would integrate such a gateway with their equipment, thereby maintaining the confidentiality of their own intellectual property. Others may prefer to delegate the development of gateway software to a third party. A similarly flexible approach is specified for the connection of lift systems for data monitoring by the National Standards Committee of the People's Republic of China[20].

For the prototype, a single Gateway application component has been developed which "listens" to the events and then interacts in an appropriate manner with the GDI. A separate execution instance of the Gateway application is launched to listen to each port (landings and cars) which is supplied with data event messages by the simulator Connector, so reinforcing the impression of asynchronous operation of the different active elements of the lift system. The gateway is written as a Java application which allows it to run in a very wide variety of operating environments. It communicates with the GDI using the Eclipse Californium CoAP library[21].

5.4 Global Dispatcher Interface Executable

The GDI itself is written in C# and executes on a separate computing device – a Raspberry Pi running the Windows IoT core on the Universal Windows Platform (UWP).

The GDI software is based on a Windows .Net library implementation[22] of CoAP[23] – Constrained Application Protocol – which is specifically designed to minimise processing and communication demands and which:

- supports a RESTful interface and
- enables discovery of resources through the "/.well-known/" URI

- supports subscription to observable resources
- supports a number of message payload formats including JSON, XML and plain-text, which may be used concurrently and interchangeably in a single implementation.

Whilst there are several available alternatives to CoAP, it was chosen because it offers the above capabilities and because the computing power and network bandwidth available to such an application, probably running in the lift motor room, are likely to be 'constrained'. However, an eventual commercial product may well employ a different open standard protocol.

CoAP messages have a similar format to HTTP messages used by web-browsers to access a web server – each request includes:
- an 'address' (URI)[17] which identifies the exact location of a particular resource that is the object of the request,
- an optional 'query' string which acts as a filter for the data of the resource and
- an optional data 'payload' which can carry data to or from the requested resource.

The GDI CoAP message payloads conform to the Standard Elevator Information Schema (SEIS) where element tag names in the JSON content have been abbreviated into two-letter acronyms in order to make more efficient use of the network resources – so for example the AssingedTo element of the CallAssignment event message is abbreviated to "La" (i.e. Lift assigned).

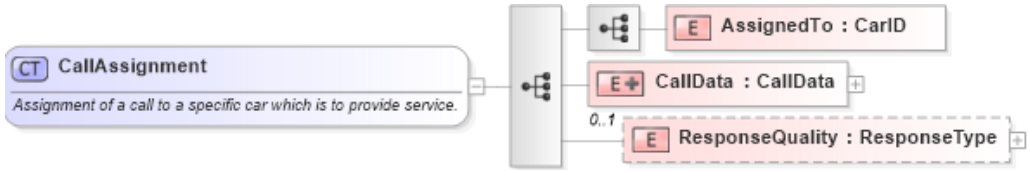


Figure 6 Schema of CallAssignment event

The following is an example of a CallAssignment as the JSON payload of a CoAP message:

{"La":2,"St":1,"Fr":1, Dr":"UP","Tr":39838.7,"Dn":0.0,"Ct":0.0,"Rk":0, "Id":43029267}

NB SEIS offers two representations for many types of change in dynamic data:
- DynamicDataType contains the data relevant to a particular node in the information model.
- LogEventType contains just enough data to describe the change that has occurred so is generally more efficient in data logging applications.

For messages directed to the REST interface it is most appropriate to use the DynamicDataType representation as the destination resource is a node in the information model. Conversely, content of responses to CoAP GET requests (including from observed resources) are more appropriately formatted as LogEventType, since the observers do not offer a REST interface. Response messages to subscribers do not expect a REST interface and so do not contain a method code (eg POST, etc).

A dummy dispatcher algorithm has been included, which appears to sit "behind" the GDI interface but can be configured to be inactive in the case where a different dispatcher algorithm interacts as a subscriber to the REST interface like any other element of the system (see Figure 4).

5.5 Global Dispatcher Sequence Diagram

Now that the components of the prototype have been defined, the design process continues by developing the sequence diagrams illustrating their interactions. For the purposes of this paper it is

only necessary to present an example of a design sequence diagram – direction call registration and assignment (Figure 7). The UML model contains the full set of sequence diagrams [12].

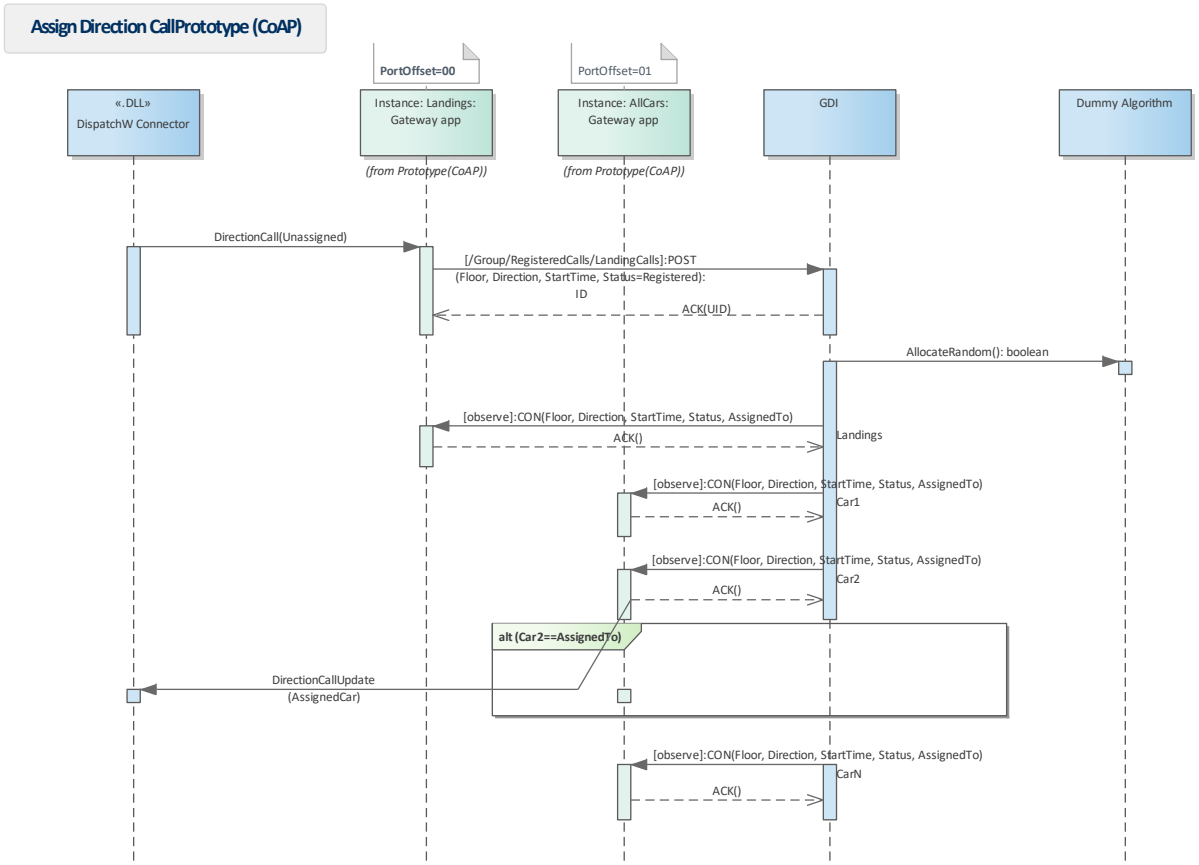
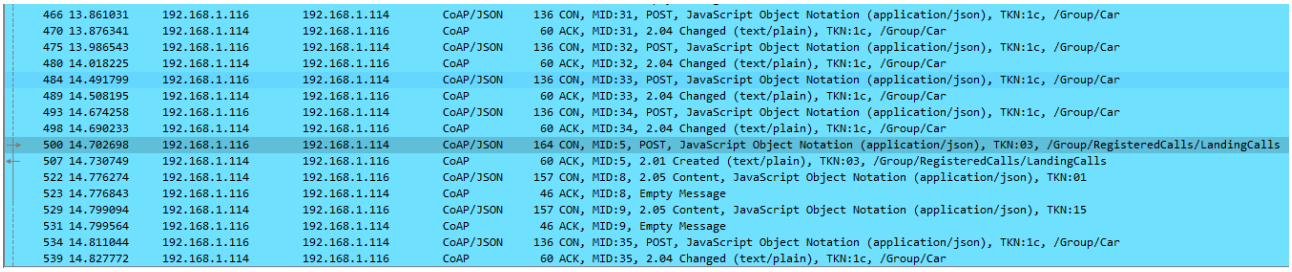


Figure 7 Prototype - Direction Call Registration and Assignment

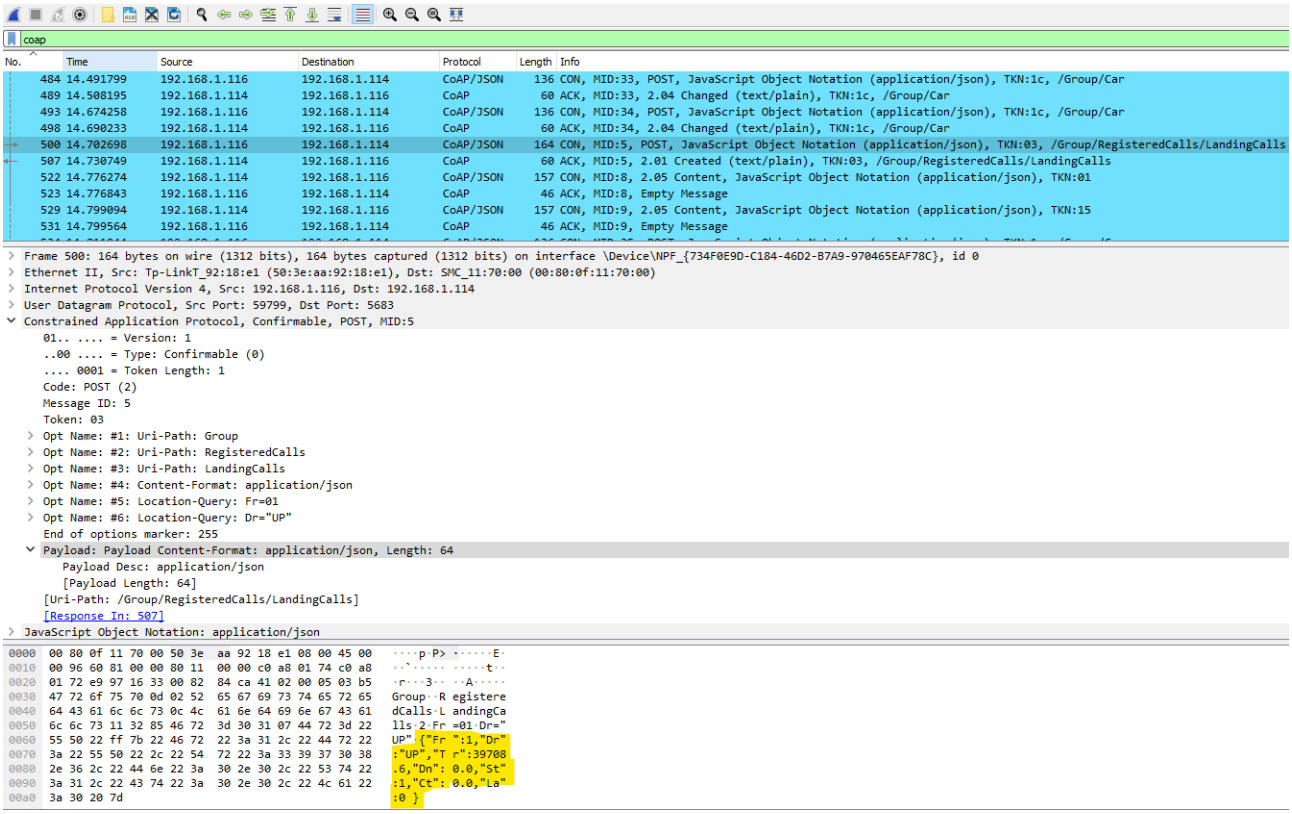
6 PROTOTYPE OPERATION - VALIDATION

Basic GDI operation can be demonstrated via the dynamic graphical display of the simulation program, where passenger arrivals and resulting car movements in response to assigned passenger landing calls can be observed. However, a more detailed examination of operation can be made with the aid of a network "sniffer" application [24] which can detect and log the CoAP request and response messages that replicate the sequence diagrams of the design (see example in Figure 8). As noted in [3], it is normal for a group of lifts to be executing multiple instances of different system use cases concurrently and asynchronously so the messages of the sequence diagrams will be mingled but they can be separated in the sniffer log by filtering out specific landing call IDs and message IDs.



No.	Time	Source	Destination	Protocol	Length	Info
466	13.861031	192.168.1.116	192.168.1.114	CoAP/JSON	136	CON, MID:31, POST, JavaScript Object Notation (application/json), TKN:1c, /Group/Car
470	13.876341	192.168.1.114	192.168.1.116	CoAP	60	ACK, MID:31, 2.04 Changed (text/plain), TKN:1c, /Group/Car
475	13.986543	192.168.1.116	192.168.1.114	CoAP/JSON	136	CON, MID:32, POST, JavaScript Object Notation (application/json), TKN:1c, /Group/Car
480	14.018225	192.168.1.114	192.168.1.116	CoAP	60	ACK, MID:32, 2.04 Changed (text/plain), TKN:1c, /Group/Car
484	14.491799	192.168.1.116	192.168.1.114	CoAP/JSON	136	CON, MID:33, POST, JavaScript Object Notation (application/json), TKN:1c, /Group/Car
489	14.508195	192.168.1.114	192.168.1.116	CoAP	60	ACK, MID:33, 2.04 Changed (text/plain), TKN:1c, /Group/Car
493	14.674258	192.168.1.116	192.168.1.114	CoAP/JSON	136	CON, MID:34, POST, JavaScript Object Notation (application/json), TKN:1c, /Group/Car
498	14.690233	192.168.1.114	192.168.1.116	CoAP	60	ACK, MID:34, 2.04 Changed (text/plain), TKN:1c, /Group/Car
500	14.702698	192.168.1.116	192.168.1.114	CoAP/JSON	164	CON, MID:5, POST, JavaScript Object Notation (application/json), TKN:03, /Group/RegisteredCalls/LandingCalls
507	14.730749	192.168.1.114	192.168.1.116	CoAP	60	ACK, MID:5, 2.01 Created (text/plain), TKN:03, /Group/RegisteredCalls/LandingCalls
522	14.776274	192.168.1.114	192.168.1.116	CoAP/JSON	157	CON, MID:8, 2.05 Content, JavaScript Object Notation (application/json), TKN:01
523	14.776843	192.168.1.116	192.168.1.114	CoAP	46	ACK, MID:8, Empty Message
529	14.799094	192.168.1.114	192.168.1.116	CoAP/JSON	157	CON, MID:9, 2.05 Content, JavaScript Object Notation (application/json), TKN:15
531	14.799564	192.168.1.116	192.168.1.114	CoAP	46	ACK, MID:9, Empty Message
534	14.811044	192.168.1.114	192.168.1.116	CoAP/JSON	136	CON, MID:35, POST, JavaScript Object Notation (application/json), TKN:1c, /Group/Car
539	14.827772	192.168.1.116	192.168.1.114	CoAP	60	ACK, MID:35, 2.04 Changed (text/plain), TKN:1c, /Group/Car

Figure 8 Sniffer log of CoAP messages



coap

No.	Time	Source	Destination	Protocol	Length	Info
484	14.491799	192.168.1.116	192.168.1.114	CoAP/JSON	136	CON, MID:33, POST, JavaScript Object Notation (application/json), TKN:1c, /Group/Car
489	14.508195	192.168.1.114	192.168.1.116	CoAP	60	ACK, MID:33, 2.04 Changed (text/plain), TKN:1c, /Group/Car
493	14.674258	192.168.1.116	192.168.1.114	CoAP/JSON	136	CON, MID:34, POST, JavaScript Object Notation (application/json), TKN:1c, /Group/Car
498	14.690233	192.168.1.114	192.168.1.116	CoAP	60	ACK, MID:34, 2.04 Changed (text/plain), TKN:1c, /Group/Car
500	14.702698	192.168.1.116	192.168.1.114	CoAP/JSON	164	CON, MID:5, POST, JavaScript Object Notation (application/json), TKN:03, /Group/RegisteredCalls/LandingCalls
507	14.730749	192.168.1.114	192.168.1.116	CoAP	60	ACK, MID:5, 2.01 Created (text/plain), TKN:03, /Group/RegisteredCalls/LandingCalls
522	14.776274	192.168.1.114	192.168.1.116	CoAP/JSON	157	CON, MID:8, 2.05 Content, JavaScript Object Notation (application/json), TKN:01
523	14.776843	192.168.1.116	192.168.1.114	CoAP	46	ACK, MID:8, Empty Message
529	14.799094	192.168.1.114	192.168.1.116	CoAP/JSON	157	CON, MID:9, 2.05 Content, JavaScript Object Notation (application/json), TKN:15
531	14.799564	192.168.1.116	192.168.1.114	CoAP	46	ACK, MID:9, Empty Message

```
> Frame 500: 164 bytes on wire (1312 bits), 164 bytes captured (1312 bits) on interface \Device\NPF_{734F0E9D-C184-46D2-B7A9-970465EAF78C}, id 0
> Ethernet II, Src: Tp-LinkT_92:18:e1 (50:3e:aa:92:18:e1), Dst: SMC_11:70:00 (00:80:0f:11:70:00)
> Internet Protocol Version 4, Src: 192.168.1.116, Dst: 192.168.1.114
> User Datagram Protocol, Src Port: 59799, Dst Port: 5683
v Constrained Application Protocol, Confirmable, POST, MID:5
    01.. .... = Version: 1
    ..00 .... = Type: Confirmable (0)
    .... 0001 = Token Length: 1
    Code: POST (2)
    Message ID: 5
    Token: 03
  > Opt Name: #1: Uri-Path: Group
  > Opt Name: #2: Uri-Path: RegisteredCalls
  > Opt Name: #3: Uri-Path: LandingCalls
  > Opt Name: #4: Content-Format: application/json
  > Opt Name: #5: Location-Query: Fr=01
  > Opt Name: #6: Location-Query: Dr="UP"
    End of options marker: 255
  v Payload: Payload Content-Format: application/json, Length: 64
      Payload Desc: application/json
      [Payload Length: 64]
    [Uri-Path: /Group/RegisteredCalls/LandingCalls]
    [Response In: 507]
> JavaScript Object Notation: application/json
```

```
0000  00 80 0f 11 70 00 50 3e  aa 92 18 e1 08 00 45 00   ····p·P> ·····E·
0010  00 96 60 81 00 00 80 11  00 00 c0 a8 01 74 c0 a8   ··`····· ·····t··
0020  01 72 e9 97 16 33 00 82  84 ca 41 02 00 05 03 b5   ·r···3·· ··A·····
0030  47 72 6f 75 70 0d 02 52  65 67 69 73 74 65 72 65   Group··R egistere
0040  64 43 61 6c 6c 73 0c 4c  61 6e 64 69 6e 67 43 61   dCalls·L andingCa
0050  6c 6c 73 11 32 85 46 72  3d 30 31 07 44 72 3d 22   lls·2·Fr =01·Dr="
0060  55 50 22 ff 7b 22 46 72  22 3a 31 2c 22 44 72 22   UP"·{"Fr ":1,"Dr"
0070  3a 22 55 50 22 2c 22 54  72 22 3a 33 39 37 30 38   :"UP","T r":39708
0080  2e 36 2c 22 44 6e 22 3a  30 2e 30 2c 22 53 74 22   .6,"Dn": 0.0,"St"
0090  3a 31 2c 22 43 74 22 3a  30 2e 30 2c 22 4c 61 22   :1,"Ct": 0.0,"La"
00a0  3a 30 20 7d                                       :0 }
```

Figure 9 Landing Call registration POST message

6.1 Example – Landing call registration and assignment messages

The following is an example of the interactions of call registration followed by call assignment for a direction call (as per diagram Figure 7), as recorded in the sniffer log *(*see Figure 9*)*:

NB The format of the sniffer log display window is:

- *Top pane: List of messages in ascending time order (blue background) with one message selected (grey background).*
- *Middle pane: Description of the properties and elements of the message selected in the top pane.*
- *Bottom pane - Hexadecimal and ASCII dump of the message selected in the top pane.*

Figure 9 shows, in the top pane, the landing call registration POST message which is CONfirmable with a payload in JSON. In the middle pane, Opt #1 to Opt #3 specify the address of the node (URI) to which the message is directed: “Group/RegisteredCalls/LandingCalls” and the Location-Query parameters floor: 01 and direction: UP in Opt #5 and #6 respectively. The payload (ie the actual

JSON) has been manually highlighted in yellow in the bottom pane of the screenshot. Note that there is no assigned lift at that point - “La”:00. Also, there is no Id: field in the payload since this is a POST request message and so it is the responsibility of the GDI to generate a unique ID as it creates a new element in the REST list.

Because the POST message was identified as CONfirmable, the GDI responds with an ACK to confirm receipt (see Figure 10). The unique ID of the newly created LandingCall is included as plain text (highlighted by hand in yellow in the bottom pane). Note that at this point the call is still not yet assigned to a lift.

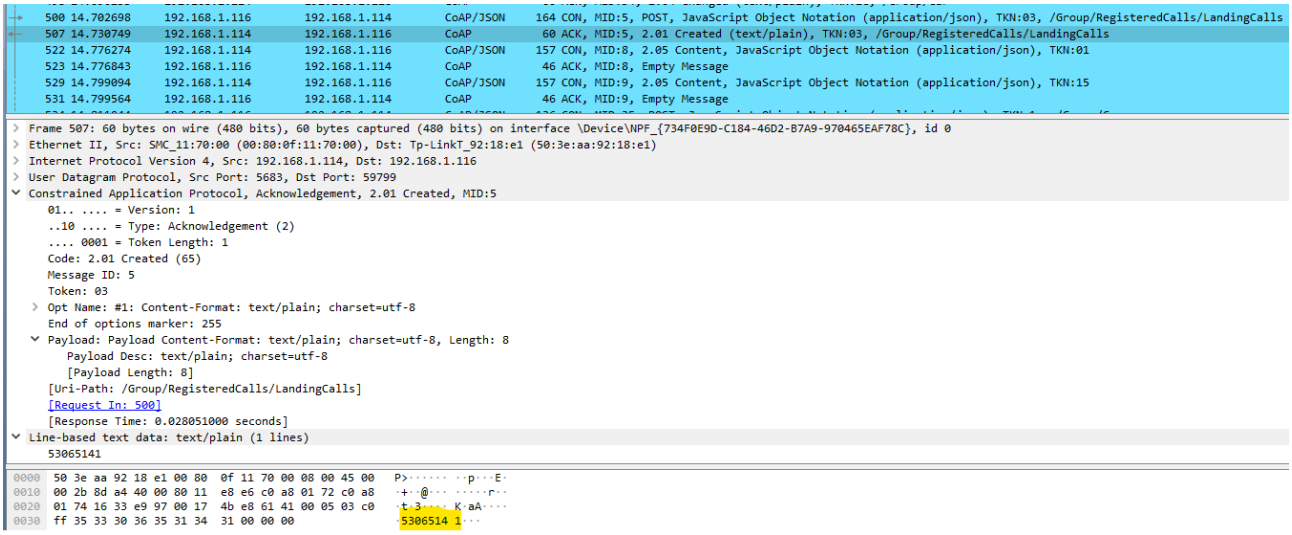


Figure 10 Dispatcher ACKnowledge of new LC including unique ID (highlighted in yellow)

The current version of the prototype includes a dispatcher service “behind” the GDI (see Figure 4) so in this case the process of invoking and receiving the response of the dispatcher is non-standard and implementation-specific as follows:

On receipt of a newly registered and unassigned landing call, the GDI invokes its internal algorithm in a concurrent thread and then continues its standard operation to await further update/query messages from registered clients or direct notification from the algorithm that the assignment has been completed. In the current prototype the algorithm simply assigns cars at random and then updates the internal information model of the GDI with the ID of the assigned car (this ID was notified by the client car in a PUT message during registration). The Algorithm returns a value of true to the GDI to indicate a successful assignment.

The GDI responds in the standard manner to this assignment update of the information model and sends a new CON message with a JSON payload (see Figure 11, in particular “La”:1 highlighted) to all subscribed observers of the node /Group/RegisteredCalls/LandingCalls where the updated landing call, now assigned, is held in the information model (in this case there are two subscribed observers – the Landings gateway and the AllCars gateway – see Figure 5). This last message is not a POST since the destinations do not support REST interfaces.

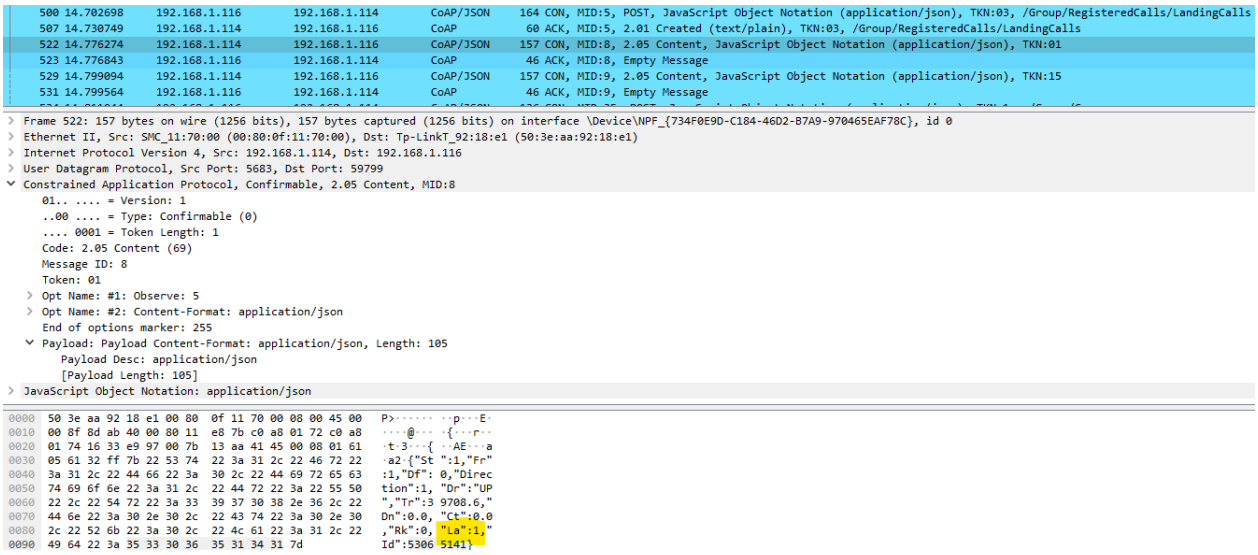


Figure 11 Dispatcher informs any observing cars of assignment (highlighted in yellow)

Each of the observers, but particularly the assigned car and the landing call station (important in the case of a destination call) will respond to the CON messages with ACKs. Network traffic could be reduced in the case of multiple car gateway instances, if any instance handling cars that were not assigned to the call were sent NONconfirm messages which do not require an ACK response.

The gateway and GDI components of the prototype can also be configured to generate copious logs of the processing they execute to assist system test and debugging. This logging is not intended to be enabled during normal operation.

7 PROTOTYPE FURTHER WORK

In the prototype, it is the simulator that manages internally the lifecycle of destination calls as they evolve from landing calls to pseudo car calls, with the effect that the cars are only indirectly controlled by the dispatcher. This is unlikely to reflect reality, where probably it will be the task of the car gateway to translate the change of call status, from Registered to Answered, into the generation of the appropriate car call and subsequently to status Cancelled. Therefore this aspect of the remaining work involves changes to the Gateway app rather than the GDI itself. It may be possible to circumvent the automatic behaviour of the simulator by directly monitoring passenger arrivals (this information is available) and configuring the simulator as if for direction calls.

The second tranche of further work would see the implementation of a multi-instance distributed dispatcher service handling assignments through bids in a sorted list. This is an important aspect of the GDI validation, but also represents a major enhancement of the prototype.

The requirements and analysis refer to the role of a car's TravelPlan in the interaction messages passed via the GDI. This will be considered for inclusion as an observable resource in a future version.

8 CONCLUSIONS

The design of a Global Dispatcher Interface has been presented using standard software design methods. Starting with a set of analysis use cases and associated catalogue of requirements, sequence diagrams illustrating the interactions between software components have been developed to document the design of the GDI. A prototype demonstration environment has been built, implementing the GDI design, interoperating via custom gateway software, with a realistic software simulation of passengers and lifts in order to validate the design. The operation of the prototype in an example of assigning a newly registered call has been discussed in detail and is evidenced with a discussion of the resulting messages which are received and generated by the GDI. The ability to support group dispatching of both direction and destination calls in a distributed software architecture has been demonstrated through the working the prototype.

The material presented in this paper is part of an ongoing research and development project and has yet to be implemented commercially. The author welcomes comments and questions, via the Editor, regarding possible improvements, errors and omissions. The next iteration of the prototype will explore and validate the potential offered by a distributed dispatcher interface.

End Note - Security

The introduction to this paper noted that the prototype, which is its subject, does not address general requirements of the GDI that are not specific to the domain of lift systems dispatching. However, it is important to stress in these concluding remarks that security must be placed at the forefront of considerations when developing a commercial product embodying the GDI. Even where the GDI is not connected directly to external networks, it is nonetheless capable of acting as an unintended route for malicious agents to gain access to the lifts or other external systems. Therefore a full risk assessment must be carried out, on a regular basis (extending throughout the product lifetime) and any exposed risks mitigated by regular updates. Refer to *CIBSE Guide D. 2020 Transportation Systems in Buildings* for a more detailed discussion [2].

REFERENCES

[1] G. C. Barney, G.C. and Al-Sharif, L. (2016) Elevator Traffic Handbook, Second edition, Chap.12, Routledge, Abingdon UK, 2016, ISBN 978-1-138-85232-7. https://doi.org/10.4324/9781315723600

[2] CIBSE-Ch14. (2020) . *CIBSE Guide D. 2020 Transportation Systems in Buildings*. Chap 14, The Chartered Institution of Building Services Engineers.

[3] Beebe, J (2018). "*Towards A Global Traffic Control (Dispatcher) Algorithm - Requirements Analysis*", Transportation Systems In Buildings, University of Northampton, Available from: http://journals.northampton.ac.uk/index.php/tsib/article/view/147. http://dx.doi.org/10.14234/tsib.v2i1.147

[4] SDLC(2021) see Software development process, Wikipedia, Available from: https://en.wikipedia.org/wiki/Software_development_process.

[5] Beebe, J (2021), "*Analysis products*"; https://dispatcher.std4lift.info/

[6] Sparx (2021), Sparx Systems Enterprise Architect. Available from: https://www.sparxsystems.com/

[7] Software Prototyping (2021), see Software prototyping, Wikipedia, Available from: https://en.wikipedia.org/wiki/Software_prototyping.

[8] Peters Research (2021). Elevate™ traffic analysis and simulation software. Available from: https://www.peters-research.com/index.php/elevate/about-elevate.

[9] Beebe, J. (2021)"*Standard Elevator Information Schema*", https://www.std4lift.info/ .

[10] Alhir, Sinan Si.(1998), "UML in a nutshell"; pp85-94 "*Sequence Diagrams*", O'Reilly & Associates, Inc., Sebastopol CA, USA, 1998, ISBN 1-56592-488-7.

[11] Bitner, K and Spence, I. (2008), Use Case Modelling, pp196, "*What is a Scenario*", Addison-Wesley, London, 2008, ISBN 02011709139.

[12] Beebe, J (2021), "Design products"; https://dispatcher.std4lift.info/GlobalDispatcher-PrototypeDesign.pdf

[13] Publish-Subscribe (2021), "*Publish–subscribe pattern*", Wikipedia, Available from:: https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern

[14] E. Gamma; R. Helm; . Johnson; J. Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley ISBN 0-201-63361-2.

[15] SeparationOfConcerns (2021). Separation of concerns. Wikipedia, Available from: https://en.wikipedia.org/wiki/Separation_of_concerns

[16] REST (2021) Wikipedia. Representational State Transfer [Internet]. Wikipedia, Available from: https://en.wikipedia.org/wiki/Representational_state_transfer

[17] URI (2021) Wikipedia. Uniform Resource Identifier. Wikipedia, Available from: https://en.wikipedia.org/wiki/Uniform_Resource_Identifier

[18] CRUD (2021) Wikipedia. Create, read, update and delete. Available from: https://en.wikipedia.org/wiki/Create,_read,_update_and_delete

[19] WinSock (2021), Microsoft, "*Winsock Network Protocol Support in Windows*". Available from : https://docs.microsoft.com/en-us/windows/win32/winsock/network-protocol-support-in-windows

[20] PRC (2021) National Standards Committee of People's Republic of China. GB/T 24476-2017 - Specification for internet of things for lifts, escalators and moving walks. 2018. Available from: https://www.chinesestandard.net/PDF/English.aspx/GBT24476-2017

[21] Californium (2021), Eclipse Foundation, "*Eclipse Californium*". Available from: https://www.eclipse.org/californium/

[22] Waher, P. (2018), *Mastering Internet of Things,* Chap 10 *The Controller,* Packt Publishing Ltd. (www.packtpub.com),ISBN 978-1-78839-748-3.

[23] CoAP (2014). Internet Engineering Task Force (IETF), "*The Constrained Application Protocol (CoAP)*", RFC 7252. Available from: https://tools.ietf.org/html/rfc7252

[24] Wireshark (2021), Wireshark.org. Available from https://www.wireshark.org/