LeetFleet

A Web of Things Fleet Management System

Ian Foster	Daniel Gresak	Tomás Kelly	Jörg Striebel
14332591	21201539	21207079	21205773

Synopsis:

System: Our team designed and will demonstrate a scalable architecture for fleet (vehicle) management using a combination of Web of Things over HTTP for the vehicles with an Actor based Akka system for fleet management. While a fully-fledged system is beyond the scope of this project, this system will act as a proof of concept for a fleet management system combining the aforementioned technologies.

Domain: The Internet of Things (IoT) has facilitated the uptake in consumer internet connected devices but it also brings complexities as the number of these devices increases. Communicating with different IoT devices usually requires different applications. There is not a single "common protocol" spoken by every device – each proprietary language is different. Connecting devices to the internet and giving them IP addresses is just one step towards the Internet of Things as it facilitates data exchange. The Web of Things – or WoT – bridges the protocol gap by using and adapting Web protocols over HTTP to connect anything in the physical world and give it a presence on the World Wide Web.

Our team designed a system that incorporates currently available WoT technologies for fleet management. Such a system could seamlessly interact with vehicles exposed using WoT. The availability of this and other systems of this kind may act as an incentive to manufacturers to introduce support for WoT protocols to their vehicles.

For our fleet management service to be successful it requires an accessible WoT directory service, so vehicles can be discovered and then managed. As the update of WoT technology increases, such directory services may be published by manufacturers themselves, or by vehicle leasing agencies, vehicle rental agencies etc..

Functionality: In the real world each vehicle has a manufacturer assigned VIN. Our prototype system has assigned each vehicle a unique four-digit number. A vehicle, on startup, registers with a local ad-hoc directory service ("The Directory").

Our fleet management application discovers available vehicles using The Directory. Each discovered vehicle is assigned an appropriate Fleet Id. The vehicle's state is mirrored in the Fleet Management system so it can be inspected by Fleet Manager personnel using a Web Client. It is possible for a Fleet Manager to remotely control some aspects of the vehicle (using the web client), via the Fleet Management system and the WoT infrastructure.

Technology Stack

Web of Things

The continuous growth of IoT solutions has led to an heterogeneous technology landscape with diverse hardware and software implementations at the edge layer of the IoT ecosystem (Dautov & Song, 2019). The Web of Things (WoT) aims to overcome the interoperability and usability challenges of the Internet of Things (IoT).

The Core Components of Web of Things

The key building blocks of the W3C WoT standardization are as follows (Kawaguchi et al., 2020):

- **The Web of Things (WoT) Thing-Description (TD)**, which describes the metadata and so-called interaction affordances (properties, actions, events) of a physical or virtual entity. This entity is referred to as a Thing.
- **The Web of Things (WoT) Binding Templates** comprise a reusable collection of blueprints used for mapping the so-called interaction affordances with the underlying communication protocols (e.g. MQTT, HTTP etc.)
- **The Web of Things Scripting API**[1] represents the W3C Web of Things implementation in Node.js and supports multiple protocol bindings. The API facilitates implementing the Thing application logic and the node.js server serves as the WoT runtime for the so-called servient which can host Exposed- and Consumed-Things.

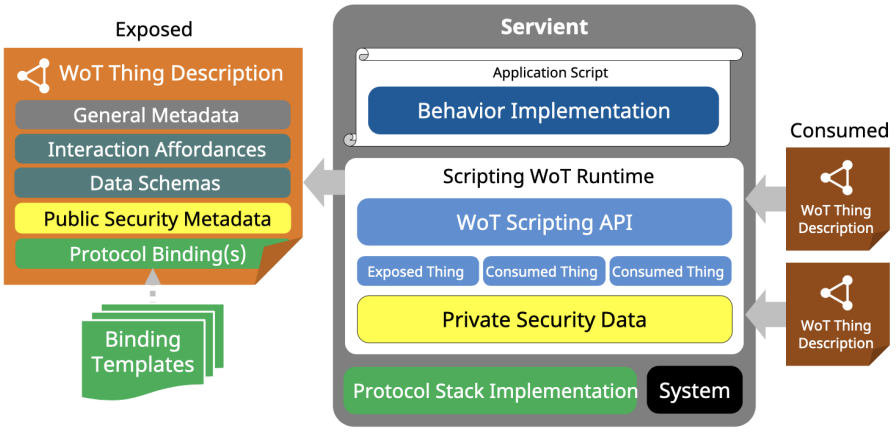

Fig 1. Implementation of a Servient using the native WoT API (Kawaguchi et al., 2020)

- **The Web of Things (WoT) Security and Privacy Guidelines** describe and discuss issues to be considered when implementing systems in W3C-WoT. Given this project is for educational purposes only, no security measures were applied.

The Web of Things standardisation describes two components for each “Thing” within the network - the Exposed Thing and the Consumed Thing. As described above, the Exposed Thing is the “actual” entity that the TD describes, which in this example are simulated vehicles. It is a software representation of a thing abstraction that provides a WoT interface for interfacing with its features (Kawaguchi et al., 2020). The communication between these two components is handled by so-called WoT-Interaction Affordances, which is an abstraction layer based on properties, events and actions as illustrated in Fig. 2. Consumed Things represent a remote thing consumed by a consumer, serving as the interface for applications to access the remote Thing (Kawaguchi et al., 2020). By parsing and processing a TD document, a consumer can generate a Consumed Thing instance (Kawaguchi et al., 2020).

As seen in Fig. 2, the bottom layer represents physical objects with sensory data. The next layer above shows the Consumed Things that contain the consumed and accessible TDs, written in JSON-LD (or JSON-Linked Data),

[1] https://github.com/eclipse/thingweb.node-wot

abstracting away the communication protocol to access the Exposed Thing. The top layer applications that access this information and interact with it by communicating over the internet (shown by the typical narrow-waist representation of internet communication protocols) with the Consumed Things.

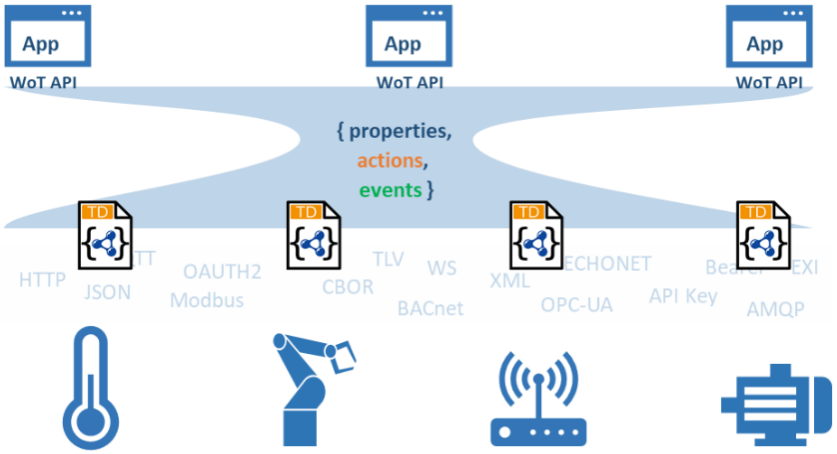

Fig 2. A representation of the WoT architecture at a very high conceptual level (W3C, n.d.)

Thing Directory

The Thing Directory services make local devices implemented as Things accessible to cloud applications by registering their metadata (Kawaguchi et al., 2020). In other words it provides a web interface for registering TDs and looking them up, or even utilising more advanced SPARQL queries for filtering TDs.

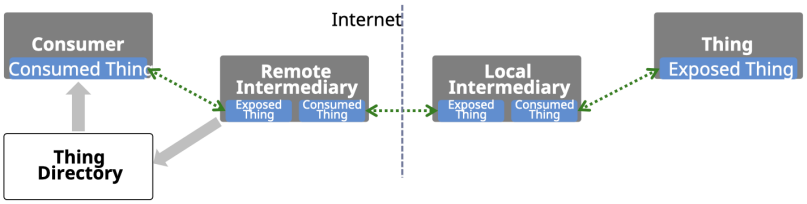

Fig 3. Cloud Service with a Thing Directory (Kawaguchi et al., 2020)

Akka

The following characteristics of Akka allow developers to solve difficult concurrency and scalability challenges in an intuitive way (*Akka Actors Quickstart With Java · Lightbend Tech Hub*, n.d.):

Event-driven model — Actors perform work in response to messages. Communication between Actors is asynchronous, allowing Actors to send messages and continue their own work without blocking to wait for a reply.

Strong isolation principles — Unlike regular objects in Java, an Actor does not have a public API in terms of methods that you can invoke. Instead, its public API is defined through messages that the actor handles. This prevents any sharing of state between Actors; the only way to observe another actor's state is by sending it a message asking for it.

Location transparency — The system constructs Actors from a factory and returns references to the instances. Because location does not matter, Actor instances can start, stop, move, and restart to scale up and down as well as recover from unexpected failures.

Lightweight — Each instance consumes only a few hundred bytes, which realistically allows millions of concurrent Actors to exist in a single application.

Akka Clustering

An Akka Cluster allows for building distributed applications, where one application or service spans multiple nodes. A cluster consists of a set of nodes joined together through the Cluster Membership Service. We elected to use the latest version of Akka clustering (Akka cluster typed for Akka 2.7.0)

Akka HTTP

The Akka HTTP module implements a full server and client-side HTTP stack on top of Akka-actor and Akka-stream (*Introduction - Akka HTTP*, n.d.). We used this to facilitate the Web of Things bridge and Web Client communication with the Akka system . We created API endpoints based on the application needs. When these endpoints are called, we can use the Akka HTTP to start up or access the actor services available, responding to each request with the appropriate response.

Django

Django was selected as a simple back-end framework to serve web content to the client. JavaScript was used to create user flow within the website instead of Django templates as this would limit the calls to our server making the application easily scalable (as the client's browser was responsible for the API calls to our actor system). Django also allowed us to control the cross-origin resource sharing policy which meant that our webpage would allow information from the other Akka HTTP server to manipulate data within the client page.

Redis

Redis is a simple in-memory key-value data-structure store, often used as a cache, database or message broker. Our team used Redis (via the JEDIS client library) as a proxy for a data store. Our application is a demonstration of a distributed system infrastructure and does not store any state. Using Redis as a proxy for a permanent data-store allows us to suggest where storage code would go and to “show” the pseudo-stored values handled by our Akka Fleet Management system.

Docker

Containerisation of this application and its different components was an important feature of development. Containerisation is a means to encapsulate a software application or its component(s) to ensure it runs in the exact same fashion regardless of the underlying physical infrastructure of the machine being used (IBM, 2019). This decision made collaboration on the development of different components and their integration easier by harmonising the development environment of each subcomponent of the system. It was decided to use Docker as the containerisation system for the application given the team’s prior experience using this technology and the ease of using Docker’s built-in networking support. Entities using the WoT architecture are designed to communicate over the internet (Guinard, 2017) so a successful prototype requires a suitable TCP/IP connected environment.

System Overview

Our team's initial plan was to build a system entirely using Java. Our initial research suggested use of the *SANE Web of Things Servient*[2] implementation would be a good potential choice. Based on the technologies we wanted to showcase we arrived at a preliminary design, as follows:

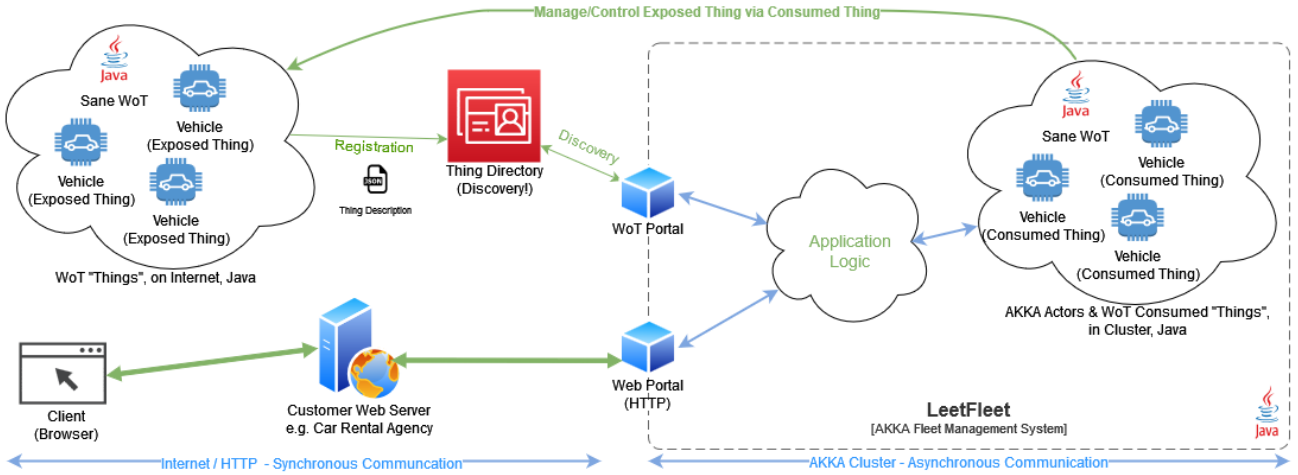

Fig 4: System Overview - Initial Design

We very quickly realised however that the SANE WoT Java project was not stable and was significantly out of date. Addressing the many dependency issues we encountered just to build the project - when we were not guaranteed it would support the functionality we required - was not possible in the assignment timeframe.
Our team faced the choice of either aborting and choosing another project - or compromising our design. The solution we chose was to use the *Eclipse Thingweb node-wot*[3] reference implementation and to build our system with an extra 'bridge' stage, as follows:

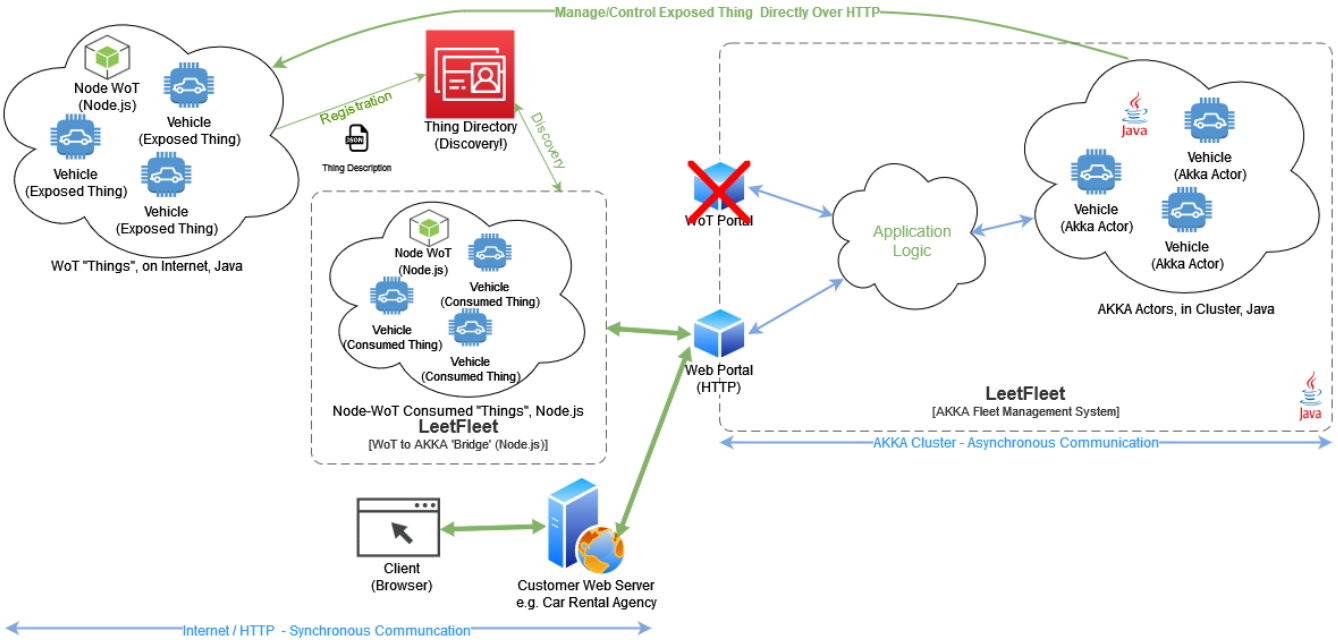

Fig 5: System Overview - Compromise Design

The new entity - 'the bridge' - is also implemented in node.js. This new entity will discover the exposed vehicles using the supplied directory service. It will use the Node-WoT framework to consume the Exposed Vehicle Devices

[2] https://github.com/sane-city/wot-servient
[3] https://github.com/eclipse/thingweb.node-wot

and - by monitoring properties we are interested in - communicate property changes to the Akka system. This has significant disadvantages as the consumed device will be outside the core application - but will still work.

Web of Things

Exposed Thing - Smart Vehicle

Within the Leetfleet system, the Exposed Things are simulations of smart vehicles. The Exposed Things are designed using a base class written in Typescript, allowing for an object-oriented approach to the class design. The TD is embedded in this base class. This class also contains methods to simulate the vehicle and register the Exposed Thing with the Thing Directory, as illustrated in Fig 6. To ensure fault tolerance, the Exposed Thing keeps trying to register itself with the directory in ten second intervals until it does so successfully. The registered TD does not contain any actual values for properties, just the means to access them (i.e. the hyperlinks to GET that information).

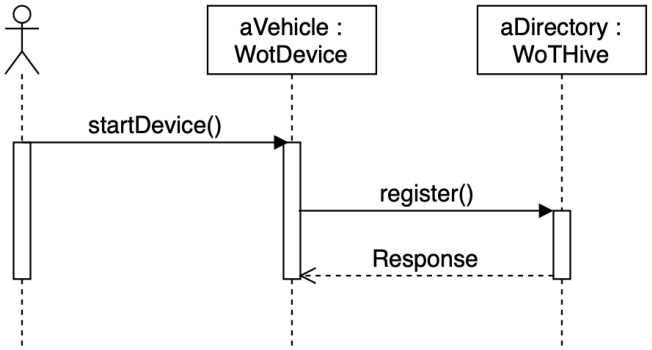

Fig 6. Sequence of starting an Exposed Thing and registering its TD with the WoTHive directory

Thing Directory - WoTHive

The *WoTHive*[4] Thing Directory would have been a key contributor to the event-driven nature of the WoT architecture. Unfortunately, the API of this directory had not been fully implemented at the time of creating the WoT bridge, particularly subscribing to events such as “TD created” or “TD updated”. The former event would have enabled an event-driven creation of consumer servient-instances for Consumed Things.

As a workaround, the bridge component of the Leetfleet system is designed to periodically query the thing directory and consume any vehicles that are present in the directory that are not yet consumed as explained below. Using the Thing Descriptions for the Exposed Things, the bridge component creates Consumed Things directly for each Exposed Thing.

Intermediary - WoT-Bridge

The bridge component of the application was originally not envisioned in the design as the intended means of creating Consumed Things was based on the *SANE Web of Things Servient* project, implemented in Java. We had hoped to build a hybrid *Consumed Thing / Akka Actor* to seamlessly integrate the two main elements of the application.

However, after determining the native Java implementation of creating Consumed Things within the Akka backend system was not a viable option for this system, it was required to create a means of communicating with the Akka backend actor system regarding any updates observed by the Consumed Thing. As such, a “bridge” program was created, which periodically queries the thing directory for a list of Thing Descriptions (TD). The WoT bridge also maintains a local cache of Consumed Things, based on the unique ID values of the things themselves. As illustrated in

[4] https://github.com/oeg-upm/wot-hive

Fig. 7, upon starting the application the bridge fetches all TDs stored in the thing directory and deletes all outdated TDs, which for this prototype were classed as any TDs present in the directory before the bridge program had been started.

However, periodically polling the TD list from the directory and deleting outdated TDs was a workaround as ideally the WoT bridge would have utilised the event-driven functionality of the WoTHive directory by subscribing to "*TD created*" events. As already mentioned, this feature was not available at the time of implementation.

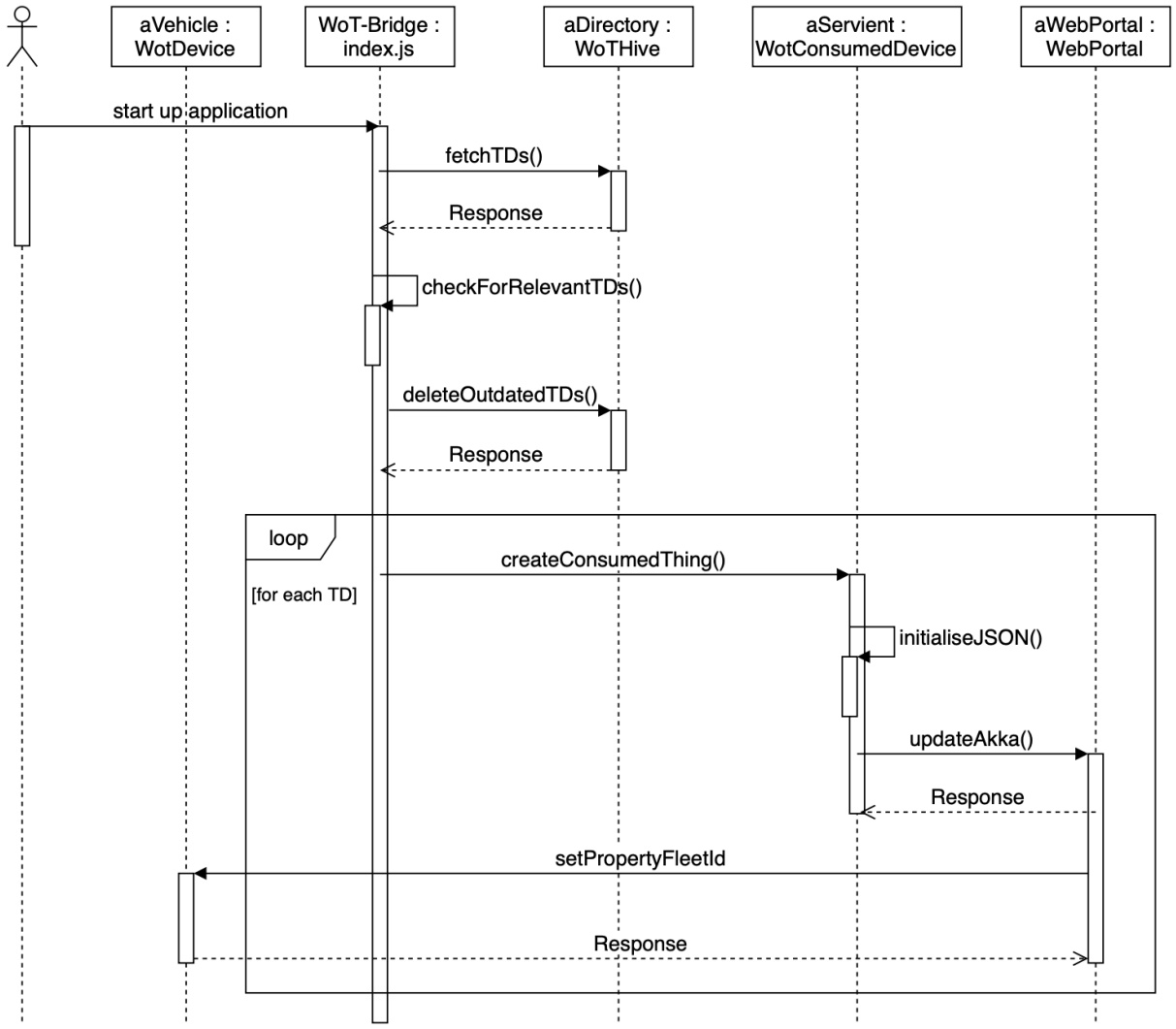

Fig 7. Sequence of the WoT-Bridge creating servient instances (Consumed Things) for each TD

As shown in Fig. 7 the WoT bridge program creates a separate consumer servient for each new Consumed Thing, a servient being a server that can simultaneously act as a client. Each Consumed Thing then interacts directly with the Akka system. One drawback of this approach is the lack of redundancy. The WoT-bridge became a single point of failure within the system as losing this component could break all communications between the Exposed Things and the backend system. This was an unfortunate side-effect of requiring a bridge component in the first place. Given this individual point of failure, precautions like additional replication of this component using a cloud deployment tool like Kubernetes could have mitigated this issue. Distributing the workload across these replicated components would also have been possible using Kubernetes and a load balancer, making the use of Kubernetes a very desirable addition to this system.

Consumed Thing - Interface to the Fleet-Management System

The Consumed Thing in this project was designed to observe certain properties of the Exposed Thing for demonstration purposes, *e.g.* mileage, and communicate updates to these values to the Akka system via the

WebPortal program. The Consumed Thing also subscribes to events on the Exposed Thing, *e.g.* a low tyre pressure event, which are triggered when the tyre pressure property of the vehicle becomes less than a certain threshold value. This was designed to send messages to Akka and then perform some simple logging.

The main tasks of the Consumed Thing is observing property changes and subscribing to events of the Exposed Thing. On each property change or event, the Consumed Thing communicates these updated values to the Akka system in JSON format. Unlike observing property changes and listening to events, the Consumed Thing is not involved in invoking actions. Actions such as locking/unlocking the door are directly invoked by the Akka system on the Exposed Thing itself.

The ideal solution would have been having the Consumed Thing inside the Akka system process the TD itself which would have allowed the backend to manipulate the vehicle directly. The bridge was designed to create a JSON message to send to the Akka system, containing the current values of the properties of the Exposed Thing. The JSON message contains the vehicle's "*Fleet Id*" which is used to route the message to appropriate Fleet Manager actor and finally on to the relevant Vehicle.

LeetFleet Fleet Management System - Akka

The actor system used for Fleet Management was developed in the latest version of Akka Typed (Akka 2.7.0) for Java. *The m*ain Fleet Management system logic is implemented as a scalable distributed Actor System. Akka supports native transparent clustering (for seamless horizontal scaling) and is designed for developing performant, scalable, maintainable and available applications. Akka exposes an Actor-based Concurrency Model as a high-level API allowing us, as developers, to write concurrent and parallel applications without threads, locking and other issues. (*Akka Actors Quickstart With Java · Lightbend Tech Hub*, n.d.)

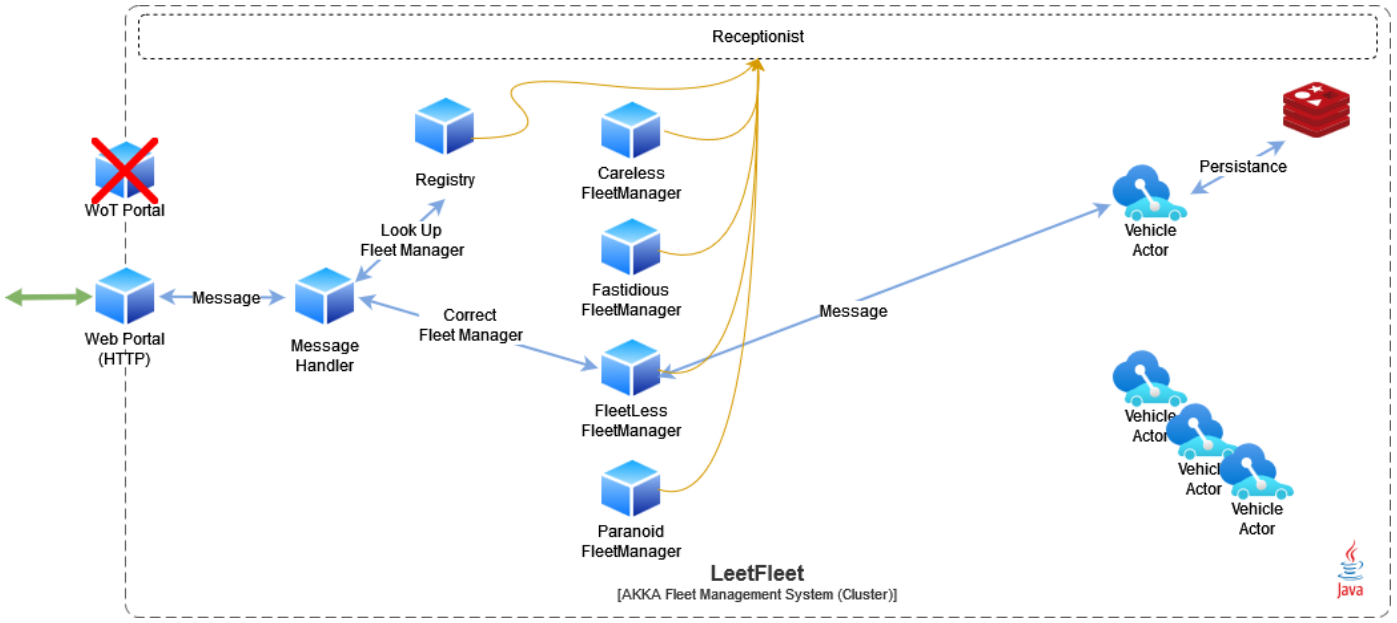

Fig. 8: LeetFleet Akka Overview

The system is split into three broad components in the Akka cluster; the Registry, the WebPortal and a suite of Fleet Managers. Our demonstration system includes Fleet Managers as follows:

Fleet Manager Name	Careless	Fastidious	Fleetless	Paranoid
Vehicle ID Range	0 - 2499	2500 - 4999	5000 - 7499	7500 - 9999

Fig. 8 shows 'The Receptionist'. When an actor needs to be discovered by another actor but one is unable to put a reference to it in an incoming message, one can use the Akka Receptionist. The Receptionist works in a cluster, an actor registered to the Receptionist will appear in the Receptionist of the other nodes of the cluster. Unfortunately,

the Receptionist does not scale up to any number of services or very high turnaround of services, prompting us to introduce the Registry actor.

- The WebPortal is an Akka HTTP module - a full server- and client-side HTTP stack on top of Akka-actor and Akka-stream.
    - On startup, the WebPortal subscribes to Receptionist notifications for the Registry (and for 'VehicleWebQuery' - one of the message handlers described below).
    - Each message received is forwarded to an appropriate Message Handler (below)
- The Registry is the actor responsible for storing the Actor References for each Fleet Manager.
    - On startup the Registry registers is existence with the Receptionist
    - The Registry also subscribes to all Fleet Manager notifications.
    - Message Handlers lookup Fleet Manager actor references in the Registry before forwarding
- The Fleet Managers are responsible for storing the Actor References for any registered vehicles in their fleet.
    - On startup each Fleet Manager registers it's existence with the Receptionist
    - Each vehicle that belongs to a Fleet Manager has an actor created in the cluster on initial connection (and is 'persisted' in REDIS).
    - The Fleet Manager forwards the relevant message to the Vehicle actor.

The Web Portal - Akka HTTP

The Akka HTTP section of the application was used to provide API endpoints to access the services that the Akka system provides. The main functions include retrieving or updating a vehicle, listing of fleet managers and listing of vehicles managed by a particular manager.

The endpoints used included:

- WoT endpoint to communicate updated thing
- List fleets endpoint
- List all vehicles from one fleet
- Description of one specific vehicle
- Post request for client to remotely control aspects of a vehicle (e.g. lock/unlock doors)

Once these endpoints were requested, Akka HTTP would then allow a handler function to send the information to the correct actor and wait for a response. Once the response is received, then the onSuccess function extracts the results of the handler function and passes it into the inner 'complete' function. This creates a HTTP response for the client or web of things to handle. We also implemented a timeout which would send an internal server error. This was done in case an infinite loop occurred.

The Akka HTTP framework included functionality which could take in the payload from a post request and a parameter from the url called.

Once the message is correctly routed, the information is parsed if needed and sent to the Web Portal Guardian. The Web Portal Guardian has methods to send the information to the correct fleet manager, or send a broadcast message to all fleet managers which will then ignore the message if the vehicle is not one of theirs. We created the Akka HTTP to be lightweight, and for this reason, the functionality of the service is all controlled within the different fleet managers.

The above basic infrastructure forms the basis of our Fleet Management system. The WebPortal stores no state. In the event of a failure it can be restarted with minimal loss ('in flight' messages) .The Registry - the actor responsible for storing the Fleet Manager actor references - is lightweight (the only state it stores is a Fleet Manager Id -> Actor Reference hash table) and extremely resilient. In the event of a failure, the Registry can simply be restarted and will load all relevant actor references from the Receptionist.

Message Handling

The LeetFleet fleet management system provides services to both WoT enabled vehicles and to Fleet Managers via a Web Portal. The initial design of two separate portals for traffic from either the WoT devices or externally was modified to incorporate a WoT 'Bridge' (implemented in Node.js). All traffic to the backend system is now handled by a single Web Portal. This portal expects and processes three classes of message:

1. A Vehicle WoT event. This type of message is typically a state update event generated by a Vehicle (*e.g.* an updated odometer reading). Our simulated vehicles generated events of this nature at regular intervals, the running LeetFleet system presented for the assignment logs a moderate stream of these messages.
2. A Vehicle Web event. This type of message is typically a *requested state update* generated by a Web Client. These events are always user generated. They flow into the LeetFleet system which then remotely activates an action or modifies a property (over HTTP) on the relevant Exposed Thing (*i.e.* the door lock actions).
3. A 'Query From the Web Client'. This type of event occurs when the Web Client requests information about the vehicles in a fleet. In a fully fledged system there would be many such messages. Our demonstration system consists of a few notional ones sufficient to demonstrate our architecture.

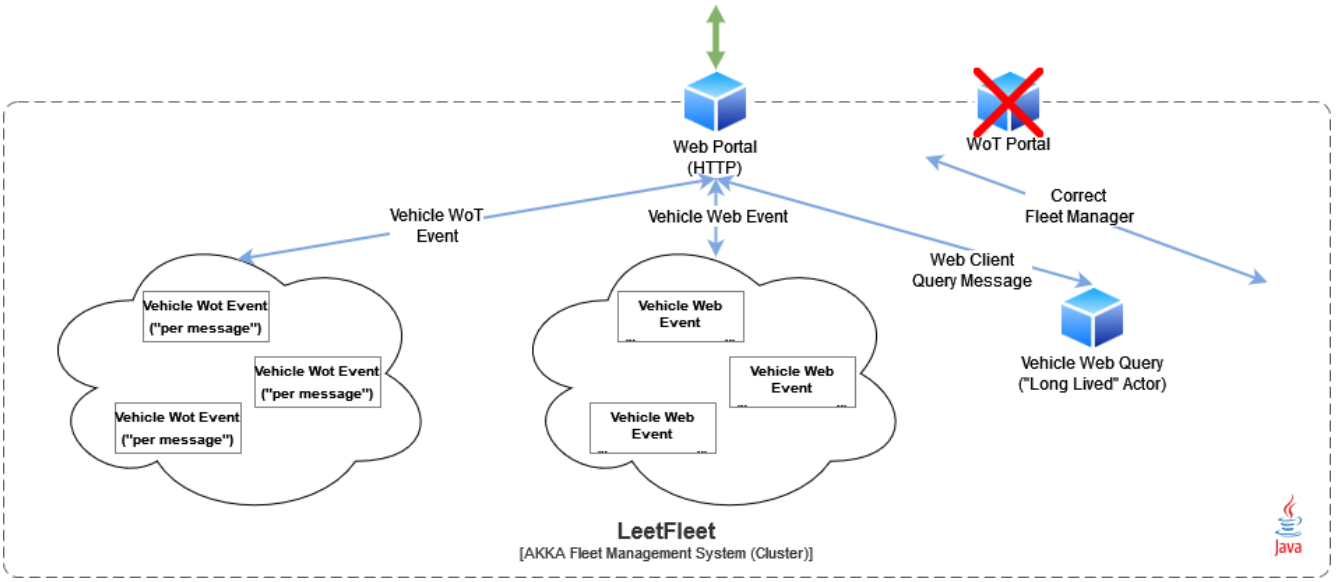

Fig. 9: LeetFleet Message Handling

1. Vehicle WoT event

When a "Vehicle WoT Event" arrives (as a JSON payload in a POST message to URI "*<leetfleethost>/wot*") the LeetFleet system creates an anonymous disposable actor to manage the activity required to deal with this request. The Web Portal delegates all responsibility for this single synchronous HTTP Request-Response cycle to this temporary actor. A message containing the JSON payload for the event is forwarded to the "Vehicle WoT Event" actor and the response that will be returned to the user is a java 'CompletionStage' - a type of promise that will be evaluated when the temporary actor has completed its work (or when the actor times out to a configurable delay - currently set to 5s).

The "Vehicle WoT Event" actor handles all the messaging required to look up the registry, pass the message to the appropriate fleet manager, etc. (meaning - most importantly - the Web Portal is not doing this work).

Initial Connection

One 'special case' handled by the Vehicle WoT Event handler is an initial communication by a Vehicle. On initial communication the vehicle will not have been assigned a 'Fleet Manager Id'. In this case the Vehicle WoT event will forward the received event to ALL Fleet Managers. This one-off overhead ensures that the initial communication will reach the appropriate Fleet Manager. All other Fleet Managers will ignore this message. The appropriate Fleet Manager will create a Vehicle Actor for this Vehicle and store an actor reference to the Vehicle in its own internal

map. In a real system we would load this Vehicle from persistent storage too, updating the stored model if appropriate.

2. Vehicle Web event

Vehicle Web events (a requested Vehicle state update from the Web Client) are handled in a very similar fashion to Vehicle WoT events. The two event types were kept separate as we envisioned several scenarios where 'requested state changes' would be handled differently to 'reported state changes'.

3. Query From Web Client

Traditional Web Server functionality for the Client Application is provided in a more conventional manner by a permanent actor (the 'VehicleWebQuery' actor). This actor assigns each request an ID and while passing messages for different web requests keeps track of them by monitoring the ID passed around.

Scalability and Fault Tolerance

The Web Portal delegates all responsibility for synchronous communications to disposable actors for updates to Vehicles. This was the central concept in our design, to remove the workload for handling individual request/response transactions from a single Actor and to spread it out across multiple Actors. This approach is ideally suited to horizontal scaling as more resources added to the cluster should directly translate to increased service capacity.

Speed of message handling is ensured by dividing the workload among each of the Fleet Manager actors. These actors could be further subdivided easily: by VIN range or by geographic area etc.. Once again, this leverages the ability of the cluster to horizontally scale. Choosing an appropriate level to model the Fleet Managers is more a question of good domain knowledge than of technical flexibility.

The key actors in the LeetFleet system are all restartable in the event of failure. If the Registry fails, it will pick up all appropriate references from the Receptionist on restart. Similarly the Fleet Managers. We envision deploying a system like LeetFleet on a Kubernetes cluster to further improve resilience.

Future work:

Our current implementation of the Web Portal would result in "in flight losses" on failure. The impact of such a failure could be significantly reduced by registering the Web Portal with the Receptionist and then introducing a confirmation message from the WebPortal to the message handler. The message handler could then detect a portal failure (on timeout) and look up the new Web Portal reference as required.

Introducing a proper persistence layer (our REDIS layer is used purely for demonstration purposes) would allow Fleet Managers to recover Vehicle state on startup (rather than waiting for first communication).

There are several improvements that could be made (what happens if a Vehicle communicates with "*the wrong Fleet ID*"? This could be handled by adding support for flood messages from one Fleet Manager to all other Fleet Managers. Etc.. Now that the base system is in place and functioning, introduction of support for these scenarios is relatively trivial.

Load testing the Vehicle Web Query actor to compare how its performance degrades relative to one of the 'per-event' actors across a range of clusters of increasing capacity would give a good indication of how successful the 'per-event' actor strategy we employed has been..

Client

The main function of the client is to display the current vehicle statuses to the application users. This is done by making API requests to the actor system and then displaying the vehicle JSON to the client in a clear and concise way. The user can then also send messages to an individual vehicle within their fleet. The example we implemented allows the user to lock or unlock the vehicle doors remotely through the client webpage.

The user is first presented with a dropdown menu of all the fleets we have. This is dynamically produced and will automatically update if fleet managers are added or removed in the Actor system. We decided that authentication is out of the scope of this project, but in a real world scenario the user would have a login and registration page before gaining access to a fleet, which can be implemented easily with Django's authentication system.

After the user chooses their fleet from the list, then they are shown the vehicle description and status of every vehicle in their fleet. They can then click on a single vehicle which will allow them to send requests to the Actor system which will then update both the Actor system's vehicle data as well as the Web of Things vehicle. This is discussed in more detail under the aforementioned headings.

Our team decided to do the majority of the functionality of the client in JavaScript. This is because it allows the client to be portable as well as scalable as the user's browser will be making most of the API calls to get information. The technology can also easily be changed to any other type of back-end framework or server/load balancer if this was later needed. We also used Djangos built in server, which could easily be updated to a more production level server and load balancer technology such as uWSGI and/or NginX if the application is ever used in production.

We also needed to change the CORS policy within the client and Akka HTTP to allow our client webpage to be manipulated with data from the Akka HTTP portal. All information is received by using endpoints provided by Akka HTTP.

Contributions & Reflections

Ian Foster

- I designed the first draft of the client using plain Javascript to simply access urls that were created available from the backend system to test the output
- I researched the Thing directory with the help of Jörg before deciding on using the WotHive implementation that had a stable, dockerised version available (see https://github.com/oeg-upm/wot-hive)
- I worked with Jörg to create the bridge program that would dynamically create the Consumed Things for each Exposed Thing that was registered in the Thing directory
- I worked with Jörg to create the base class for each Consumed Thing to handle the logic of updating Akka with a JSON representation of each simulated vehicle as it updated
- I performed manual integration testing alongside all other team members to debug any features with remaining issues at the conclusion of the project

Reflections

I personally found the networking components of this project the most interesting but simultaneously the most challenging. Much of my own debugging time was spent on issues around messages being passed from a component of the system I was working to another component of the application. For example, the loss of the discover feature from the thing directory became more and more apparent in terms of the scope of functionality that was lost as the project progressed. Implementing dynamic and efficient networking in applications is something that previously was of less interest to me versus business logic but now that I understand the scope of the former and its impact on an application, I will definitely endeavour to learn more about this area in the future. Another area that I would have liked to improve upon in the code is the bridge program. It, in its current form, is very vulnerable to failure and represents a major vulnerability in the entire system. Given the opportunity, I would have liked to improve the design of the bridge to make this component more resilient to failures and errors to improve the system's overall fault tolerance.

Overall, I greatly enjoyed working on this project primarily due to the opportunity to work with several new technologies and frameworks that I had never interacted with before. My primary focus during the development cycle was on the WoT components of the project, specifically the Consumed Thing and the bridge program to handle communicating information to the Akka backend system from the WoT "side" of the project. I learnt a lot about WoT and the philosophy behind the design of this standardisation and given that I had no prior experience with IoT prior to this project, this simulated version was a nice introduction to some ideas around IoT development.

The novelty of the technology in this project, while very interesting, was also I think the main challenge for the development team and the main element that took away from the project. This is to say that it was necessary to devote large amounts of time to the understanding and research of these new technologies, which was excellent for personal development and for learning outcomes from the project but hampered the actual development of the project itself in some ways. A primary example for myself was the time required to understand the concept of the WoT standardisation meant that I had less time to implement other parts of the project I had wanted to include, like Kubernetes deployment. The reduced timeframe I had left myself for this element then led to that feature of the project having to be abandoned with less time spent on it than would otherwise have been the case and so regaining some of that research time may have allowed for enough debugging and development time to have included some of those originally desired features of the project. While my own time-management skills also played a role in this loss of time I'm sure, it does merit considering, at least for myself, how much novel technology I take on for a given project and how to more accurately assess the time requirements of learning new technologies moving forward.

Daniel Gresak

- I worked very closely with Tomás to design and create the Akka system.
- I researched with Tom, the new Akka typed version, which differs in the way messages need to be implemented. I implemented the routing of the JSON which was sent from the client to Akka (endpoint: localhost:8080/web). The JSON had to be parsed and used to create a vehicle object, which was sent to the web portal guardian.
- I researched and implemented the routes within Akka HTTP. This included ensuring the right data was passed in to the Akka system and that the response was sent out once the Akka system was finished processing the request.
- I researched and implemented the first version of the communication with our redis docker image. I tried using Jedis with the JSON type. Tomás later changed this to using the string JSON type.
- I created the client from the first draft that Ian created. I implemented all of the website frontend including all api calls to the actor system, user flow within the single page website and data management within the front end. I also made sure that we didn't get CORS errors from requesting information from a different server (Akka http) within the single web page.
- I created the docker file for the client and made sure it worked with the docker-compose file.
- I helped team mates with debugging which got me involved in all aspects of the application, including the Web of Things section.

Reflections

My biggest challenge with the project was understanding and implementing the newer version of Akka (Akka Typed) as opposed to Akka classic. Although the main idea is very similar, the implementation is quite different. There is also not much information on sites such as stack overflow in relation to the new Akka version and any information that was there was mostly implementing it in scala. I overcame this by doing my best to fully understand the technology and looking carefully at the documentation. This involved a lot of thought and collaboration with Tom to find answers to any issues that arose as the documentation didn't specifically address all issues so we had to read between the lines. This challenged my problem solving skills which will benefit me greatly in my future career.

I would have used another data store technology other than redis as Akka does not natively support redis. This could have provided better persistence as redis is better used for caching. This could be implemented in future work without much change to the initial application.

I also would have looked at the technology we used more carefully before going ahead. We were ambitious with our stack as we had a group of 4 people. I am happy with how the application turned out, but I felt we spent more time on the project than we needed to as we used lots of new technologies which took time to understand and implement. I think we could have used Web of Things with classic Akka or rest which would have meant we could have a similar application with more functionality as part of the application could be implemented faster.

We learnt that Akka, once set up and understood, is very efficient at executing multi threaded behaviours and relatively easy to set up. One limitation of Akka is that because it's asynchronous, we needed to create timeouts and use CompletionStage to make it compatible with synchronous HTTP messages. Of course this limitation is true of any asynchronous technology and this comes with benefits of speed and efficiency.

Redis was also new to us and we learnt that it works well as a quick way of storing key value pairs, but found that using another database technology would have been better for persistence and would be easier to implement if it was natively set up to be used with Akka.

Tomás Kelly

- Initial application design
    - Build an Akka cluster using the latest Akka syntax as a feasibility exercise - understanding the importance of seed-nodes and their role in a cluster.
    - Build an Akka Http module using the latest Akka syntax as a feasibility exercise
    - Design the message passing pattern used in the Akka system (per-event actors)
- In collaboration with Jörg investigate the feasibility of using the SANE WoT Servient Java API. We concluded it would not be possible to build this in the timeframe provided and after confirming that using a non-Java solution was acceptable for the assignment, opted to use the Node-wot implementation.
- In collaboration with Daniel explore the new Akka Typed (Java) system. This involved but was not limited to:
    - developing an understanding of the typesafe configuration system used by Akka
    - learning how to bootstrap a guardian actor and spawn child actors from there
    - learning how to integrate REDIS as an information cache for the Akka system. Little use is made of REDIS in our demonstration system (save to showcase the *'ability'* to persist data if required)
    - developing an understanding of the new message handling approach used by Akka
    - developing an understanding of the lambda based syntax exposed by the API
    - developing an understanding of Behaviours and how Akka Actors are completely defined by them
    - use of the Receptionist and the service key, subscription service it provides
    - use of Timers under the new API
- In collaboration with Daniel I spent a considerable amount of time deliberating design choices for the Akka implementation. With little or no experience in the new API it was difficult to choose optimum design patterns outside those showcased in the Akka documentation.
- Design the 'disposable per-event' actor model and implement - this is the key 'innovation' presented in the Akka system to allow the Fleet Management system to scale horizontally.
- Testing on the Akka system
- Integration testing (significant)
- In collaboration with the team, testing on the WoT and Client components.

Reflections

General

The Akka Actor system is an appealing and performant system - but learning the (new) API from scratch was a significant undertaking for an assignment of this timescale. The effort in learning the Akka API and developing the skills to exploit it would be more suited to a "term-long" project than to a technology demonstration. Choosing two such new technologies (both *WoT* and *Akka Typed (Java)*) was not optimal for this assignment. Our compromise design did enable us to complete the project as we first envisioned - but came at the cost of yet more extra work as our team had to take on the Typescript and Javascript involved in the Node-wot implementation.

Design

The scale of the proposed system was large for this assignment. Upon completion there are many design choices we simply did not have time to complete or investigate. Getting the WoT Consumed Thing into the Akka Java actor system being chief amongst them. If repeating the project I would not choose to use REDIS - it's not natively supported in Akka. Using a natively supported persistence layer (e.g. MySQL) would have involved less work and have facilitated more functionality.

Future Work

It would be trivial to expand the client's functionality to allow managers to send more types of requests to vehicles (allowing greater granularity of control). There are many optimisations - such as introducing a confirmation message from Web Portal to Vehicle Event (discussed above) - that would make the system more resilient to failure.

Jörg Striebel

- I researched the WoT architecture with its numerous implementations listed by the W3C - Web of Things Consortium.
- In collaboration with Tomás Kelly, I investigated the feasibility of using the “SANE Web of Things Servient” project, a W3C Web of Things implementation in Java inspired by the “Eclipse Thingweb node-wot” project.
- I investigated and experimented with the “Eclipse Thingweb node-wot” implementation in Node.js after we decided that utilising the “SANE” project was unfeasible.
- I designed, implemented and tested in JavaScript the initial version of the Exposed Thing and the Consumed Thing. This Exposed Thing version already contained the logic and emulation of the smart vehicle (mileage increase, tyre pressure loss, etc.), but it did not register itself with a Thing Directory. Instead, it directly communicated with the Consumed Thing. See “leetfleet/vehicle/legacy_scripts”.
- To meet the scalability requirement, I refactored the Exposed Thing to make it reusable by utilising TypeScript along with node-wot as a Node.js dependency. TypeScript enabled using an object-oriented approach. I also implemented a method to register the Exposed Thing (vehicle) with a thing directory.
- In collaboration with Ian Foster, I investigated different TD directories and started experimenting with the most promising directory, the WoTHive directory.
- I designed and implemented the alpha version of the WoT bridge (see index.js) to dynamically spin up servient-instances for Consumed Things. Subsequently, I collaborated with Ian Foster on refining it.
- I dockerized the “vehicle” and “wot_bridge_directory” services.
- Finally, I supported the system integration and testing of the overall application.

Reflections

One major challenge I faced in this project was utilising the cutting-edge technology standard Web of Things, particularly trying to take advantage of the Web of Things implementation in Java, known as “Smart Networks for Urban Participation (SANE), a research project developed by some students from the University of Hamburg. Unquestionably, this Java implementation would have made the current WoT bridge (workaround) obsolete by allowing to create Consumed Things directly within the backend of the fleet management system. However, this complex project appeared to be still under development and included multiple dependency issues at the time of the investigation. Moreover, the complete lack of documentation made it extremely challenging to understand its utilisation, not to mention resolve the dependency issues. As an alternative, the reference implementation “Eclipse Thingweb node-wot” in Node.js seemed more mature as it ran stable and provided better documentation with hands-on examples. Another significant challenge was designing and creating the WoT bridge since the event-driven features of the chosen WoTHive thing directory, such as the “TD-created” event, were not incorporated at the time of realising this project. The lack of this feature required polling the directory periodically to fetch the stored TDs and sort out the outdated ones.

If I could start again, I would better exploit the discovery mechanisms of the Web of Things by creating a slightly different use case scenario. For example, a care-sharing fleet management system where the client can search for a specific size of car available in its area based on the GPS location.

Having already been exposed in other projects to the Internet of Things (IoT), this project provided a great opportunity to explore the benefits of the Web of Things, chief among its interoperability and abstraction from underlying communication protocols. Nevertheless, the Web of Things standard still appears primarily used in academia and has yet to be widely spread across the industry.

References

Akka Actors Quickstart with Java · Lightbend Tech Hub. (n.d.). Lightbend Tech Hub. Retrieved December 4, 2022, from https://developer.lightbend.com/guides/akka-quickstart-java/index.html

Akka Typed (Java). (n.d.). Akka Quickstart with Java. Retrieved January 4, 2023, from https://doc.akka.io/docs/akka/current/typed/index.html?_ga=2.150310582.1260975877.1672852130-1766490521.1666377730

Containerization Explained. (2019, May 15). IBM. Retrieved December 29, 2022, from https://www.ibm.com/uk-en/cloud/learn/containerization

Dautov, R., & Song, H. (n.d.). Towards IoT Diversity via Automated Fleet Management. *MDE4IoT/ModComp@ MoDELS*, *1*(1), 47-54.

Guinard, D. (2017, April 8). *What is the Web of Things? – Web of Things*. Web of Things. Retrieved December 29, 2022, from https://webofthings.org/2017/04/08/what-is-the-web-of-things/

Introduction - Akka HTTP. (n.d.). akka.io. Retrieved 12 26, 2022, from https://doc.akka.io/docs/akka-http/current/introduction.html

Kawaguchi, T., Matsukura, R., & Lagally, M. (2020, April 9). *Web of Things (WoT) Architecture*. W3C. https://www.w3.org/TR/wot-architecture/

W3C. (n.d.). *Documentation - Web of Things (WoT)*. W3C. https://www.w3.org/WoT/documentation/