Architecture Documentation

Project structure

The application code is separated into three parts.
Those three parts are represented by three packages, but in real life, they would actually represent a three separated applications.
`ota.monitoring.vehicle -` package represent a **mocked** application running on vehicle itself. It handles released packages downloading, installment, and bunch of notifications from vehicle to backend system. It listen on activemq for messages regarding new package release (activemq simulating real-life mediator. More about this later)

`ota.monitoring.backend -` package represent some of the functionality of backend layer, whom vehicle is communicate with. Backend send package release notification on vehicle, and receive responses and notification from the same. Along with vehicle communication, it is responsible for communication with ERP system as well, for getting new package releases.

`ota.monitoring.analitics –` package is related to analitics dashboard application, with use data from database, collected by backend application. It provides visual representation of data like,

- fleet selection
- fleet vehicle’s overview
- details of each vehicle like:
  - Vechile packages details
  - Geolocation of last 24 hours details
  - And could be also display another important data like: owner data, chargers locations, battery swapping support places, etc. etc.

On analitic dashboard page, you can also access a collaboration/deployment overall diagram regarding system architecture
The dashboard is available at http://localhost:6789/ota-monitoring/ once you start application.

Classes important to mention

I must mention several classes, even I put comments through the code.
`ERPPackageReleaseMockFactory –` is in charge to create mock data, and to simulate data gained from ERP system, related to new packages. The entire business logic of this class is devoted to that only (so not related to business logic of the app itself).
`VehicleGeolocationMockFactory –` creates geolocation of vehicles, simulating when app start like, vehicle has been in the past transmitted these data from itself to backend system.
`ChargingListener –` is actually observer, subscribed to battery charging done abstract event. Here I created it as a Jms listener, just to accent its purpose (nothing is creating battery charging event)
`ota.monitoring.shared.dto –` package actually contains object shared within an application, in order to not duplicate code for objects on vehicle and backend (so it is just to reduce code and not duplicate)
`GeolocationUpdateJob -` is class for sending a vehicle's geolocation coordinates. Here it is job, I disabled it because of obvious reason.
`@NoImplementedOperation –` I put on places which I just wanted to stress as a point of interest in this proof of concept, and not really start implementation.

System architecture overview

The architecture itself, real life without mocking, etc, can be checked at dashboard, or opening image: /ota-monitoring/src/main/resources/public/coll_deploy_overview.jpg

Package Release production – As denoted, ERP system would be responsible for generation of that data. In our code, I am simulating remote ERP, and generate some semi-ready data, related to package release.
In real world, our backend application would integrate with ERP, and get data produced by ERP, by one of following ways:

- MOM (middle oriented middleware) integration. ERP would send messages to broker (activemq, reabbit, etc) and our application would collect them from there
- Calling exposed web service via REST, or using SOAP messages for intercommunication
- Shared database, as the oldest and least viable solution, where our app will read data per some frequency, and digest them

In either solution, data obtain from ERP are more likely that would need to be adjusted (transformed and prepared) within first ESB tier, before processed by our app.
Regarding security aspect at this point, we would use **ssl transport layer** for communication, and **ip restriction range** of access as well.
Which solution we are going to apply, mainly depends of ERP capabilities.
In this situation, mom cluster would be choice of mine (if possible), because of easy scalability and high availability, and not only because of that but probably more clients, would be interested in such an event (package release), and by publishing such a topic, every part of interest could participate within (some statistical problem, or another department within a company).
I am not expecting hundreds of package releases every second, so this architecture would suit needs and requested **SLA** of this architecture part.

Backend application – or better say cluster (as depicted on diagram) consists of several machines, running application servers and backend application.
Cluster is introduce as well, to support high availability just as scalability in communication with any client (ERP, vehicle, mediator, etc)
The applications are using sql database cluster, to store data from system, just as a entity cache deployed at each instance. Database communication is also restricted for ip range, giving security for this communication.
Sensitive data would be strongly encrypted (either using AES or MD5 hash – depending, do we need to decrypted them back or not) before saved into database, hence achieving reliability as well.
Database layer itself, would use replication from main db, into slave for purpose of data high availability as well.
Communication with vehicle is happening two ways:

➔ From backend to Vehicle

➔From Vehicle to system

First one, I assume, uses some kind of mediator system for this communication. For example, if we are talking about android OS, we could use Google play for package release upgrade, or some provider (**data carrier**) which will transfer package release event along with payload, up to our targeted vehicle.
Whether it will be pull or push notification mechanism, depends on this mediator (if any).
If we are talking about custom implementation of vehicle software, I would prefer push technique, instead of client puling, in order to reduce unnecessary traffic volume on our server. So, once we have a ready release, just push it to vehicle of interest.
The reason I introduce mediator at first place is, separation of concern, since our system would not need to carry about delivery process, if we use google play, or itune as mediator (just publish app).
Here I am using activemq as mediator, just to depict situation.

Communication from Vehicle to backend system, regarding various notifications (package download status, release status) is going to be done via https as http secure layer. Encryption, and certification identity is something we need to maintain security within this communication, just as another things we should introduce like ip range server acceptance, etc.
In this situation, Vehicle talk to load balancing cluster, which further propagate requests to server, chosen with, let say – least used server load balancing strategy.
This architecture, Vehicle communication with load balancers cluster, and lb cluster communication with app cluster, support scalability, high availability just as performance.
We can easily do horizontal scaling of any mentioned hardware (load balancers, application servers), in case of vehicle number increase, in order to handle that enlarged traffic volume, without any downtime.

Analytic application, is using data collected by backend application, and more likely communicate to slave, in order to avoid traffic on master.

Monitoring system, as one of most important part of the any architecture, is present here as well, if we want to achieve certain level of any **service level requirements**.

We must be aware is machine goes down, if server goes down and no app is available, otherwise, our SLA is compromised. That is why we want monitoring system, with appropriate notification mechanism. Along with checking health status of the system, we can also monitor on performance of the system as well, and act in according to specific action, to improve it if needed.

Vehicle-OTA graphic, represent my understanding of vehicle system. We have a linux host, where packages are installed. Each package (let say **autopilot**) migh consisting of several features (`speed-limit", "time-limit-to-drive", "turn-on"`). Each feature gain data, and communicate with appropriate sensors, in order to express its nature (autopilot features must communicate with, for example, object and parking sensors, speed sensor, weather sensors, etc.)

Addressing dataset and analitics

Regarding range anxiety, we could conclude that, it is actually based on *gathering all information related to items which bring to alleviation of the range anxiety* such as:

- charging infrastructure,
- the development of higher battery capacity at a cost-effective price,
- battery swapping technology,
- use of range extenders,
- accurate navigation and range prediction

In our application we are collecting informations from the vehicle like **every hour geolocation** (`GeolocationUpdateJob.java`)and **battery charging event (**`ChargingListener.java`**).**

Every hour, our vehicle transmits notification to backend server, regarding current geo location. Since our vehicles are 24/7 on the network (each one of them has SIM card with 3G, plus they can be switched to wireless connection as well – my assumption), there would be scheduler within vehicle app, which will transmit this information.

Having that, our analytic application would easily compare those locations with range anxiety alleviation items (chargers, battery swapping), and would see distance between vehicle and each of them.
Of course, we would need first information about all chargers and places where battery swapping may occurs, either by collecting them in our system, or by contacting third party for that (maybe Tesla itself has exposed web service for that information).

Battery charging event is also one of the helpful features, because we could use that information for more precise analysis of range anxiety, since not we just have every hour geolocation info of particular vehicle (and we know how many battery volume it can empty out per hour), but also concrete date/time when charging happen just as with also geo location of that event along with how much battery has been charged in percentage.

All these information can easily facilitate analytic in determination of whether is there range anxiety should be present at vehicle owner OR whether it exists.
All mentioned range anxiety analysis is related to stationary factors.
Along with them, we have two more factors in our list as range extenders, and potentially higher capacity battery change.
Application covers this as well, since our vehicle model would tell us which range extenders it supports (and provide us with info like battery capacity fulfilling), just as we could implement field describing if range extenders is available at all for this specific model (a lot of things could be implemented in the app, but after end, goal is not create full flagged tesla monitor app, but proof of concept).
By development of higher capacity battery change, notification can also be sent to vehicle, informing those model which are capable for that, as well.

As you can see, range anxiety can easily be handled with concrete support for range anxiet allavation item management. Some of them this is implemented in the application, where for other one we would need specific information, but nevertheless, I think I described my point above.

Forecasting application features

So far, our application has been working with so called *transacted data* using *rdbms*. Vehicle communication with backend system has been limited to data processed within acid transactions context, assuring data consistency across persistence layer.
Processing package release status transmitted from vehicle, just as geolocation in this case (way and structure as well), are required to be processed transactionally (so either everything succeed regarding particular event (download/installment) or it fails, because we want to have utterly consistent state related to vehicle in database).
If for example, backend application while processing **downloading** event for specific vehicle **partially** succeed, then **installment** processing would be inconsistent as well, jeopardizing entire analytic system regarding determination of user experience degradation.

However, along those transactional data, which SQL databases provides perfect solutions, Internet of Things would mean collection of, sort of, metadata, related to core business logic.
We maybe decide generation of that metadata, based on the values of core data (like: maybe based on the transmitted geolocation coordinates and download status, can be derived correlation between location and success/failed downloading. That correlation represent example of metadata), or simply dispatching a more information and more frequent from the vehicle them self, like heartbeat event per minute, with payload representing health status of entire vehicle, collected from sensors, etc…
Such a data could be exponentially growth in our system, and since they are representing non functional information and huge volume of generated data, nosql databases, created just for that purpose, should play main role in the action.

Introducing at application level, so called *polyglot persistence programming (*point in code, where based on certain business logic applied on incoming data we decide which persistent layer should be used for storage), we can easily manage this situation:
Transacted data (core data) goes to SQL databases, and metadata – either generated by app itself, or coming from vehicle, are redirected to more suitable direction – nosql.

When we talk about big data and nosql, we must be familiar with CAP theorem, which at bottom line tell us that: in nosql ecosystem between consistency, availability and partition tolerance , only two of them can be achieved entirely.

Most suitable solution for this purpose of data persistence is probably column based databases like Cassandra and Hbase.
Reason for that is, they are actually designed to run in the big clusters (many database nodes), easy scaling cluster and improving persistence capacity, just as providing some built in features like versioning, compression and data expiration.
Simply adding new node in existing cluster, read/write operations are getting scaled by needs. So we do not need to worry, if vehicle production burst, we would just add - proportionally to expected increase amount of data, additional nodes.
Using column family based databases, we are also free of rigid schema constraint. In other words, we do not need setup, as mandatory to sql database, prior to saving data, since data simply can be in any form which is suitable for us (the consuming application).

Ecosystem where Hbase lives, is based on Hadoop platform, which provides distributed file system (HDFS) and another functionalities as mapreduce capability. Hbase/Hadoop maintain easy replication between nodes, keeping consistency as close as possible.
When I mentioned that Hbase scalability and fault tolerance, it is actually design in such a way that failure of node is acceptable and taken into account (heart beat is maintained between nodes via Zookepper).
From everything above, regarding CAP theorem, Hbase supports consistency (written value will be consistently replicated across other nodes within cluster) and partitioning .

Nevertheless, these forecasting seems to me natural extension of the application and functionality, because when we talk about Internet of Things or machine-to-machine data acquiring and generation, big data management is something which is inevitable.
(Visual representation about this part can be checked on diagram: *forecast_assumptation_diagram.jpg* as well).