COORDENADORIA DE ENGENHARIA DE COMPUTAÇÃO

ALEX COVOLAN VIEIRA COELHO
GABRIEL GIOVANINI DE SOUZA

SISTEMA DE IOT PARA URGÊNCIAS NO TRÂNSITO

Sorocaba/SP

2017

Alex Covolan Vieira Coelho
Gabriel Giovanini de Souza

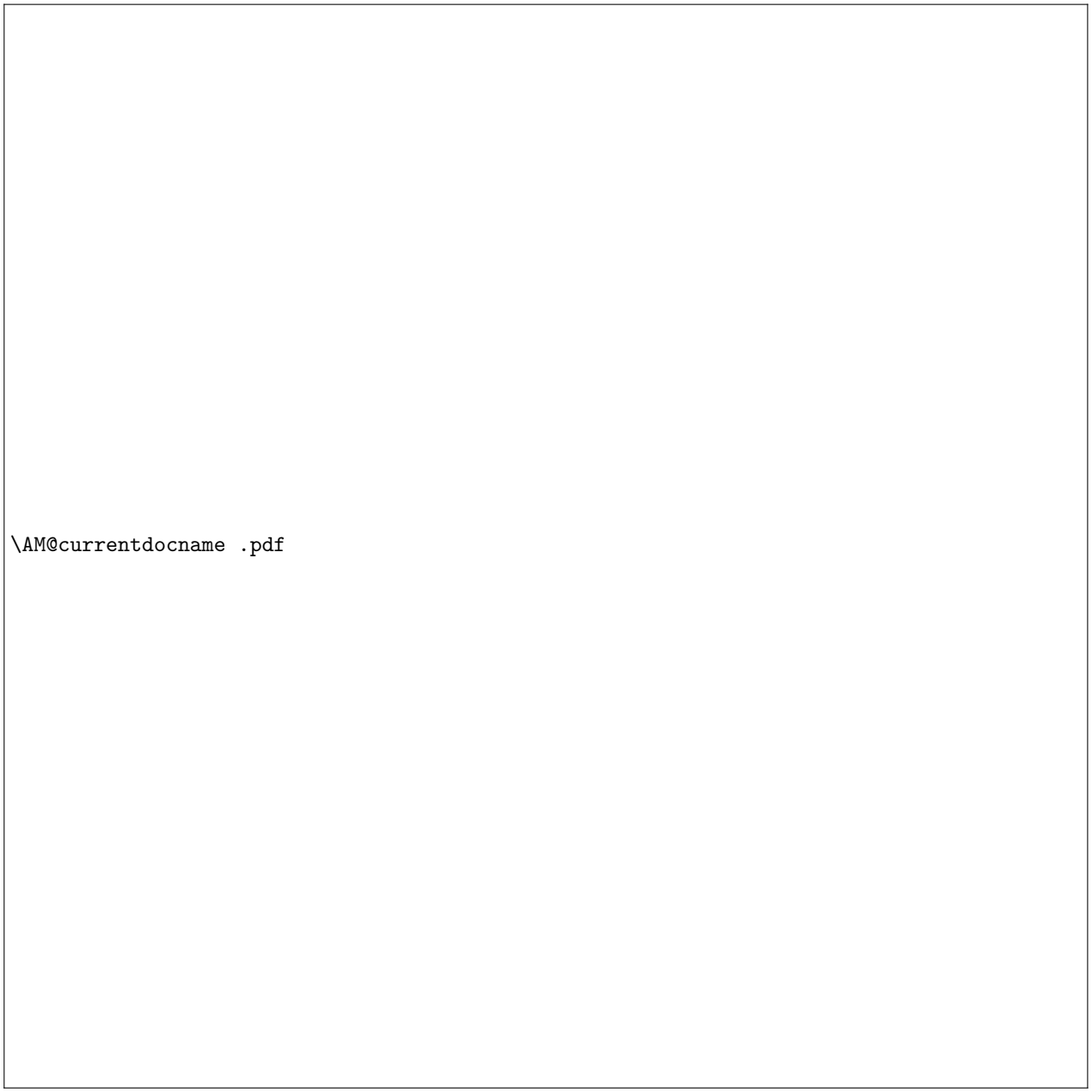
SISTEMA DE IOT PARA URGÊNCIAS NO TRÂNSITO

Trabalho de conclusão de curso apresentado à Faculdade de Engenharia de Sorocaba como exigência parcial para a obtenção do diploma de graduação em Engenharia de Computação.

Orientadora: Ma. Andréia Damasio de Leles

Sorocaba/SP

2017


\AM@currentdocname .pdf
.pdf

AGRADECIMENTOS

Primeiramente a Deus por nos proporcionar a vida e saúde necessária para estarmos concluindo mais uma jornada de nossas vidas, ao meu pai Luiz Coelho que não mediu esforços para me apoiar nessa jornada, me incentivando e me dando sustentação em todos os momentos de dificuldade, à minha mãe Cleusa Coelho a qual passou noites sem dormir preocupada por eu estar em uma cidade distante, mas que mesmo assim continuou me apoiando, pois ela sempre quis o meu melhor, ao meu irmão Alan Coelho que sempre esteve ao meu lado e sempre me admirou, à Sarah Fernandes a qual dedico todo meu amor, e sem a qual não sei como conseguiria concluir esta graduação, pois ela sempre esteve comigo nas minhas horas de estudo me ajudando, dando opiniões e transmitindo todo seu amor, além de me servir café. Dedico também a todos os colegas de faculdade, os quais mostraram total cumplicidade em todos os momentos desta jornada, em especial Rodolfo Cugler, Felipe Pereira e Bianca Correa que sempre formavam nossos grupos de trabalhos, além dos momentos de descontrações em que sempre estávamos juntos.

Agradeço também a aqueles que não estão completando esta jornada junto comigo, mas que em algum momento fizeram parte dela e levo comigo muitas recordações, em especial Thiago Kobayashi que se tornou um grande amigo na vida pessoal, mesmo não estando mais na mesma Faculdade. Não posso me esquecer também de todos meus colegas de trabalho que torceram por mim e me ensinaram coisas valiosas para minha carreira, deixo aqui um agradecimento especial para o pessoal do horário do almoço, em que sempre estiveram comigo para almoçar, bater papo e dar risadas.

Por fim agradeço o meu parceiro de TCC Gabriel Giovanini, que ao decorrer do trabalho se tornou um grande amigo e não me abandonou nem nos piores momentos, ficando acordado nas madrugadas ajudando para a finalização deste trabalho, tanto na documentação quanto no protótipo, além de me ensinar muitas coisas sobre a comunidade *Open Source*. Juntamente gostaria de deixar um agradecimento especial a nossa orientadora Andréia Leles que acreditou em nosso trabalho e em nossa capacidade de desenvolvê-lo nos ensinando o caminho a percorrer. Deixo aqui meu agradecimento geral a todos que direta ou indiretamente nos ajudaram para a conclusão desta jornada, nos apoiaram e torceram por nós, meu muito obrigado.

AGRADECIMENTOS

Primeiramente gostaria de agradecer meus pais Vanda Andreia e Nilton Steffen que desde o início sempre me incentivaram a continuar com meus estudos e buscar uma melhor qualidade de vida para meu futuro. Sem falar que sempre estiveram lá quando eu precisei de ajuda, tiveram paciência e me entenderam quando muitas vezes tive que estar ausente devido aos estudos.

Gostaria também de agradecer meus amigos Jonathan e Fabio que já estão formados mas que estiveram comigo por toda a duração deles no curso. Eles foram de grande companheirismo, não somente dentro da sala de aula como também em minha vida pessoal, onde até pouco tempo atrás, vários anos pudemos compartilhar o mesmo ambiente de trabalho através de várias fases de nossas carreiras. Gostaria também de fazer um agradecimento a todos os colegas de classes no qual tive o prazer de trabalhar junto durante todo esse tempo na faculdade. Um agradecimento especial ao meu parceiro de TCC Alex Coelho, que durante todos esse tempo junto no projeto tornou-se um excelente amigo no qual pude contar sem exitar não somente no TCC mas também fora dele. Além de ser uma pessoa que compartilha muitas das ideias da comunidade *Open Source*, tornando o trabalho de TCC uma experiência e aprendizado pouco menos estressante. Gostaria de agradecer muito a nossa orientadora Andréia Leles que acreditou no nosso projeto e dedicou de seu tempo a nos ajudar nessa reta final de nossa jornada.

O verdadeiro significado da vida é plantar árvores,
sob cujas sombras você não espera sentar.
(Nelson Henderson)

RESUMO

O presente trabalho apresenta um sistema de IoT que através dos conceitos de V2V e V2I, visa auxiliar unidades de emergências em meio ao trânsito, comunicando aos veículos que estão na mesma rota desta unidade, que ela passará por lá e eles devem abrir caminho, além disso possibilitar o armazenamento de dados para uma futura análise de *Big Data* sobre os veículos conectados a arquitetura desenvolvida, para que se obtenha assim, indicadores sobre o trânsito para que sejam tomadas decisões a fim de melhorar a segurança para quem nele trafega. Para tal desenvolvimento utilizamos de ferramentas novas no mercado, de forma a buscar um novo modelo de arquitetura que venha a ser robusta, escalável e de fácil manipulação e integração com outros projetos que possam vir a ser desenvolvidos. Tais objetivos foram alcançados de forma que quando realizados testes de *stress* e simulação a arquitetura se comportou de forma satisfatória apresentando respostas consideradas rápidas, validando assim a arquitetura, conseguimos a agilidade durante o desenvolvimento através das ferramentas utilizadas, como é o caso do encapsulamento de serviços em Docker e Clojure.

Palavras-chave: IoT. Big Data. Clojure.

ABSTRACT

The current project presents an IoT system that uses concept of V2V and V2I to help emergency units at the traffic, letting drivers know they are commuting through the same path of an emergency unity and that they need to make the path clear for that unit to get to its destination faster. And beyond that the project stores all the data gathered from that process for further analysis of the Big Data, so that we can measure some indicators that may help to improve the health and safety of the traffic. To accomplish it was used new tools and a new architecture that is scalable, robust, easy to use and it is gonna be capable of integrate with another systems that may be created in the future. The stress test report and simulations show that those objectives were achieved, it behaved satisfactorily, and achieved high data throughput . In summary it can be said that this architecture reached its goal using agile system like Docker and Clojure.

Keywords: IoT. Big Data. Clojure.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

IoT	*Internet of Things*
RFID	*Radio-Frequency IDentification*
V2V	*Vehicle to Vehicle*
V2I	*Vehicle to Infrastructure*
DOT	Departamento de Transportes do Estados Unidos
IP	*Internal Protocol*
TIC	Tecnologia da Informação e Comunicação
IDC	*Digital Universe Study*
HDFS	*Hadoop Distributed File System*
API	*Application Programming Interface*
JVM	*Java Virtual Machine*
Saas	*Software as a Service*
CQRS	*Command Query Responsibility Segregation*
JSON	*JavaScript Object Notation*
JRE	*Java SE Runtime Environment*
SaaS	*Server as a Service*

SUMÁRIO

1 INTRODUÇÃO

A internet das coisas (IoT) vem sendo muito difundida, ganhando cada vez mais força no mercado atual, se tornando uma tendência das novas aplicações que se tem desenvolvido, ela se caracteriza por ser a comunicação entre objetos dentro de uma rede de dados, de forma a resolver problemas e trazer facilidades para os usuários desta tecnologia, no trânsito ela vem sendo implantada nos computadores de bordo dos carros, como pode ser visto em carros de diversas marcas, em que pelo painel do veículo podem ser agendadas revisões periódicas do veículo, bem como realizar troca de informações com a central de ajuda da concessionária. (**??**)

O IoT ainda tem muito a ser explorado dentro do trânsito, pois hoje se tem carros que carregam consigo uma vasta gama de tecnologias com relação a sensores e dispositivos de segurança, mas ainda se tem pouco das tecnologias de IoT empregadas dentro deles, muito menos uma infraestrutura robusta para suportar veículos conectados a ela, mas é um ramo que está em crescimento e é questão de tempo para que estas tecnologias passem a surgir, muitos estudos e protótipos já se tem feito, como é o caso do carro autônomo, em que provavelmente passará a existir em um futuro próximo.

Tratando-se de trânsito, os congestionamentos tem se tornado um problema comum nas grandes cidades, dificultando assim a circulação de unidades de emergência que se deslocam para realizar o atendimento às vítimas, sendo que um minuto a mais ou a menos pode significar a vida de uma pessoa, esta situação é problemática e complexa de ser resolvida, pois mesmo que os motoristas abram caminho para a unidade de emergência isto demanda um certo tempo, sem contar que muitas vezes não é possível que se abra caminho para a passagem da mesma por questões do próprio congestionamento.

Tendo em vista esta problemática, o presente trabalho visa se utilizar do conceito de IoT para resolver este problema, de forma que existindo uma interface embarcada com acesso a internet nos veículos, estes recebam alertas antecipadamente de que uma unidade de emergência passará por aquela rota e ele deve abrir caminho. Para isso a unidade de emergência deve comunicar ao sair atender a vítima, qual a rota percorrerá e a infraestrutura se encarregará de informar os veículos que estiverem presente na mesma rota, esta interface embarcada também poderá enviar informações relevantes sobre o veículo para a infraestrutura, desta forma possibilitando diversas análises de dados.

Desta forma o trabalho visou a criação de uma arquitetura que seja robusta, escalável e distribuída o suficiente para suportar a grande quantidade de veículos que

trafegam nas vias públicas e realizaram troca de dados com a infraestrutura através do *Vehicle to Infrastructure* (V2I), um outro conceito dentro de IoT. Para isso buscou-se utilizar de novas ferramentas para o desenvolvimento desta arquitetura que também está relacionada com o *Big Data*, pois nela se tem um banco de dados exclusivo para este tipo de análise de dados, dentre elas Docker, Clojure, arquitetura CQRS, MongoDB e Datomic como bancos de dados, não se prendendo aos padrões tradicionais de *Big Data*.

A escolha do tema a ser abordado se deu pela filosofia do IoT, que é proporcionar facilidades e melhor qualidade de vida aos seus usuários, desta forma utilizá-lo no trânsito para resolver um problema complexo como este faz todo sentido, além de se tratar de uma causa muito nobre que são as vidas das pessoas.

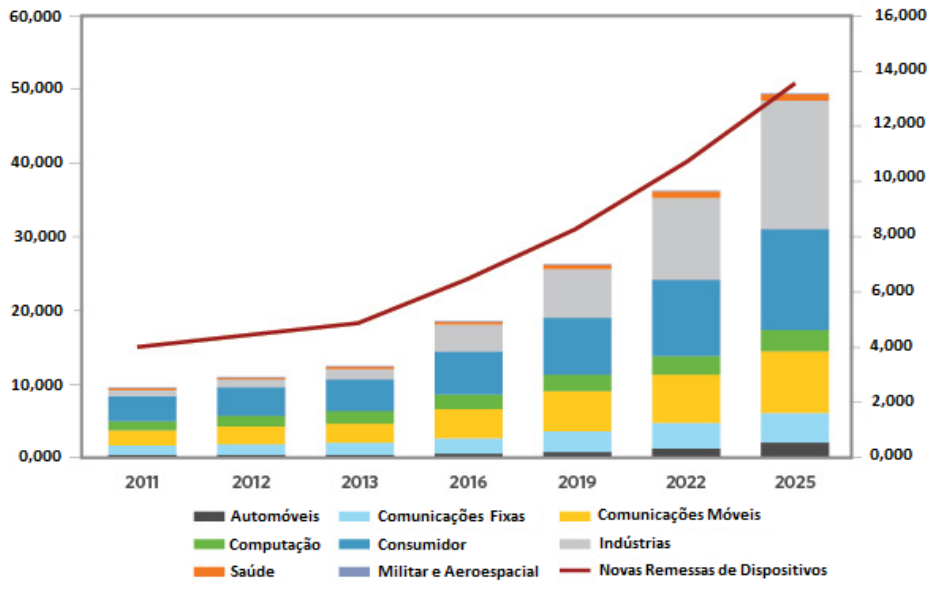
Tal trabalho se torna uma grande contribuição para a comunidade em geral, tanto a população que passará a ter um atendimento mais ágil no caso das emergências, vindo a salvar mais vidas através do rápido deslocamento das unidades de emergências no trânsito, seja uma ambulância ou bombeiro. Outro beneficiário de tal trabalho será a comunidade *Open Source* cuja a qual terá os códigos abertos para continuidade do projeto ou uso em outros subprojetos que possam se utilizar da arquitetura desenvolvida.

O trabalho está dividido em dois capítulos teóricos nos quais são apresentados todos os conceitos envolvidos no trabalho e juntamente com o funcionamento das tecnologias neles envolvidos, um capítulo explicando as ferramentas utilizadas, bem como a organização destas para o desenvolvimento da arquitetura, um capítulo de resultados onde é mostrado o que se conseguiu atingir através do uso da arquitetura, alguns casos de testes e a discussão destes resultados, e por fim a conclusão em que é mostrado os pontos relevantes sobre o trabalho.

2 INTERNET DAS COISAS APLICAÇÕES EM MOBILIDADE URBANA E SAÚDE

Internet das coisas, conhecido também como IoT, sigla que em inglês significa *Internet Of Things*, originou-se através de Kevin Ashton que em 1999 realizou uma apresentação na empresa Procter & Gamble (P&G), quando falava em se etiquetar eletronicamente os produtos da empresa através do uso de Identificador de Rádio Frequência (RFID), assunto que era recente na época. Desde então este paradigma tem sido muito discutido, principalmente no contexto atual, em que é possível notar um crescimento exponencial de tecnologias desenvolvidas neste sentido, como é mostrado na figura 2.1 que apresenta o aumento no uso de IoT mundialmente, fazendo uma estimativa até o ano de 2025.(**??**)

Figura 2.1 – Gráfico de crescimento do IoT entre os anos de 2011 a 2025



Fonte: **??**, **??** (Adaptado)

Tais dados se devem as consequências geradas pela emergência de tecnologias microeletrônicas, *wireless* (*Wi-fi*, *Bluetooth* e *ZigBee*), interfaces de comunicações móveis que se somaram às fixas já existentes e devido a formação de uma grande rede ubíqua capaz de conectar seres humanos com uma grande facilidade, possibilitando assim fornecer toda a base para a formação da IoT. (**??**)

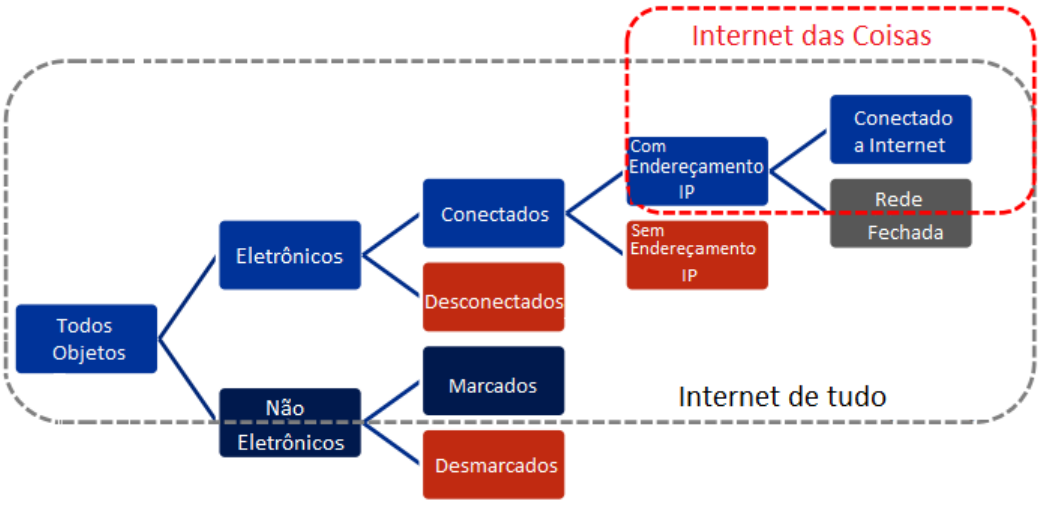
2.1 IOT

No conceito de IoT um terceiro elemento foi inserido nas redes pervasivas que se possui hoje em dia, os objetos, sendo assim dentro da rede é possível se ter a comunicação entre humano-humano, humano-objeto e objeto-objeto, desta forma é possível ter humanos se comunicando normalmente como já acontecia anteriormente, humanos definindo comportamentos para os objetos e recebendo dados dos mesmos e objetos trocando informações entre si disponibilizando dados a humanos, dados estes, úteis para tomada de decisões ou até mesmo para facilitar atividades do dia a dia.(**??**)

> Quando os objetos podem sentir o ambiente e se comunicar, eles se tornam ferramentas poderosas para entender coisas complexas e responder a elas com eficiência. Embora tais objetos inteligentes possam interagir com humanos, é mais provável que interajam ainda mais entre si automaticamente, sem intervenção humana atualizando-se com as tarefas do dia.(**??**, p. 2)

Tais objetos podem ser considerados como tudo que está na rede e possui um endereçamento *Internet Protocol* (IP), podendo interagir com outras interfaces endereçáveis dentro da mesma rede ou em outras através da internet, como mostra na figura 2.2. (**??**)

Figura 2.2 – Fluxograma da IoT



Fonte: **??**, **??** (Adaptado)

Esses objetos podem ser um automóvel, uma geladeira, uma câmera, um sensor de temperatura, entre muitas outras interfaces, o que importa é que elas estão interligadas pela internet tomando ações de forma automática sem a intervenção humana. Pode-se citar o exemplo de um senhor que sofre de mal de Alzheimer e mora

sozinho sendo que seus filhos não podem estar 24 horas por dia com ele, então os filhos decidem implantar sensores na casa do pai e pela vizinhança para que possam saber remotamente aonde ele está. Estes sensores estariam conectados a internet enviando dados para os filhos e emitindo alertas caso o pai saia de casa.(**??**)

Um outro exemplo de aplicação da IoT é apresentado na figura 2.3.

Figura 2.3 – Exemplo de aplicação da IoT

Fonte: **??**, **??** (Adaptado)

Para que as aplicações de IoT tenha este tipo de comportamento é necessário que se tenha uma infraestrutura para dar suporte a esses objetos, ela pode ser estruturada de diferentes formas utilizando diversas tecnologias, mas de modo geral para o funcionamento de um sistema de IoT é necessário que se tenha os objetos conectados na internet ou a uma rede local, que envie e receba dados da infraestrutura (banco de

dados ou armazenamento na nuvem) e os aplicativos que tem a função de gerenciar o sistema acessam e enviam os dados se comunicando diretamente com a infraestrutura, como é mostrado na figura 2.4 (**??**)

Figura 2.4 – Estrutura de um sistema de IoT

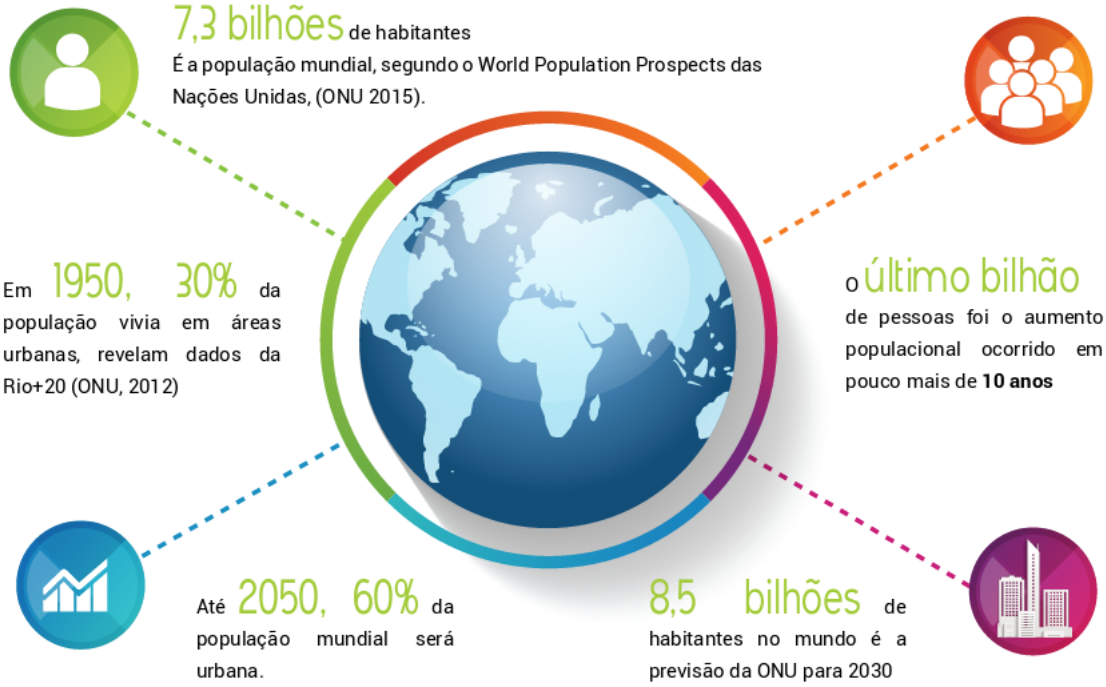
Fonte: **??**, **??** (Adaptado)

2.2 CIDADES INTELIGENTES

Com o grande crescimento da população como, é possível ver na figura 2.5, as cidades também vem crescendo, mas de forma desordenada e desigual, causando problemas como o esgotamento de recursos, aumento da desigualdade social, aumento das áreas de favela, sem contar o caos com relação a locomoção dentro das cidades. Diante deste cenário criou-se o conceito de Cidades Inteligentes (*Smart Cities*), que tem a finalidade de reinventar as cidades, ou seja, reestruturá-las, a fim de que não haja desperdícios, tornando-a uma cidade sustentável e que a cidade seja organizada da melhor forma possível, trazendo uma melhor qualidade de vida aos cidadãos que nela vivem.(**??**)

Para tal reestruturação das cidades, transformando-as em Cidades Inteligentes, espera-se contar com o auxílio da área de Tecnologia da Informação e Comunicação (TIC), ou seja, boa parte das mudanças nas cidades se deverá pelo uso da tecnologia, mais especificamente pelas tecnologias de IoT. Elas poderão ajudar na redução de gases poluentes, redução na quantidade de lixo gerado pela população, redução do

Figura 2.5 – Impacto populacional



Fonte: **??**, **??**

uso de recursos naturais, melhora da locomoção e segurança dentro das cidades, entre outras melhorias.(**??**)

> As maiores metrópoles do mundo têm adotado objetivos de tráfego e mobilidade para solucionar ou mitigar o problema de congestionamento com soluções de cidades inteligentes ativadas por Internet das Coisas (IoT), mas a mobilidade urbana não para em uma escolha contínua que consiste em se mover de A até B. (**??**, p. 1).

2.2.1 Mobilidade Urbana

Dentro do conceito de Cidades Inteligentes, a mobilidade urbana se deve grande atenção, pois no século XXI ela tem se tornado um desafio a ser resolvido dentro das grandes cidades, pois o crescente número de veículos particulares causa um inchaço no trânsito, dificultando assim a locomoção, principalmente em grandes centros urbanos (**??**). Desta forma dentro de uma Cidade Inteligente a tecnologia pode ser aplicada para solucionar ou ao menos ser um paliativo aos problemas existentes, tecnologia esta que poderia atuar diretamente no trânsito, implantada através de *smartphones* ou nos próprios carros, a fim de evitar acidentes, melhorar o fluxo, indicar rotas mais rápidas atuando diretamente na redução de gases poluentes e trazer maior facilidade aos motoristas. (**??**)

2.2.2 Saúde

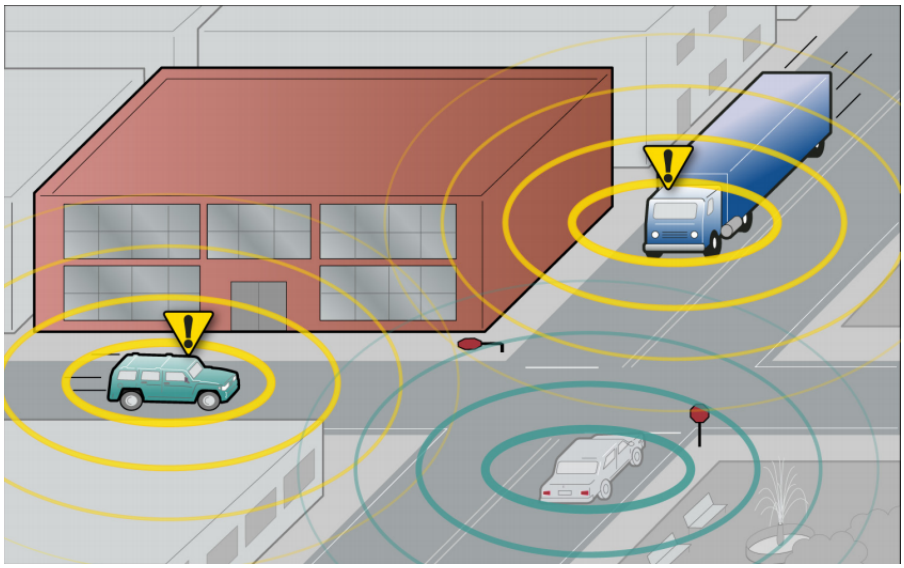
Outro setor que se deve bastante atenção é o da saúde, pois é a necessidade básica da população, e infelizmente ela é muito precária nos dias de hoje, por diversos motivos, principalmente os governamentais, mas através da tecnologia é possível que se mude este contexto. Com o uso de IoT é possível se criar aplicações na área da saúde que venha auxiliar no tratamento de doenças, cuidados com os pacientes, monitoramento e diagnósticos, transferência dos dados e colaboração, como por exemplo cadeiras de rodas inteligentes, Unidades de emergência conectadas, veículos de resposta, e hospitais, dentre tantas outras utilidades. (**??**)

Na área da saúde um grande desafio a se vencer é a de confiabilidade nos dados obtidos, pois um dado errado ou algo que se perca durante a transmissão pode representar a vida ou a morte de uma pessoa, visto que no futuro o uso de IoT na saúde será inevitável é necessário que se criem formas de manter esta tecnologia funcionando de forma segura e confiável. (**??**)

2.3 V2V

Veiculo para Veiculo, ou V2V, habilita carros a se comunicarem entre eles em uma tentativa de avisar motoristas sobre potenciais acidentes ou colisões. A base da tecnologia é usar uma onda de rádio de baixo alcance para permitir que os carros se comuniquem, podendo também que os carros enviem informações como localização, velocidade, direção, e também os estados dos freios, como mostra na figura 2.6.

Figura 2.6 – V2V simulação



Fonte: **??**, **??**

2.3.1 Regulamentação

Em dezembro de 2016 o Departamento de Transporte dos Estado Unidos (U.S. DOT) anunciou que esta trabalhando na regulamentação do uso da tecnologia em veículos de uso diário. O DOT diz que a tecnologia de rádio terá um alcance em média de 300 metros, e oferece um alcance maior que a abrangência de um radar ou câmera, em adição de não ser obstruídos por obstáculos ou outros veículos. O mesmo departamento acredita que a tecnologia poderá ser utilizada para avisar veículos sobre perigos eminentes, particularmente quando se está em uma conversão ou realizando a troca de faixa. Adicionalmente, o departamento diz que os carros com sistemas automáticos de direção (ou até mesmo carros completamente autônomos) se beneficiarão das informações fornecidas pelo sistemas V2V. (**??**)

2.4 V2I

Da mesma forma como acontece no caso do V2V, no paradigma de Veículo para Infraestrutura (V2I) os carros podem se comunicar, mas aqui a comunicação ocorre entre o carro e a infraestrutura, podendo receber instruções dela, assim como enviar instruções sobre as condições do veículo ou dados sobre o trânsito. A infraestrutura por sua vez, vem a ser as antenas que captam os dados do carro e os enviam para a nuvem, podendo usar estas informações transmitidas para se tomar decisões sobre o trânsito, analisar estatísticas, apontar trechos em que seja necessária a intervenção dos agentes de trânsito e trazer maior facilidade para o gerenciamento do mesmo. (**??**)

Além da administração por parte dos agentes organizacionais, é possível que estes enviem mensagens para os carros a fim de alertar sobre algo á frente ou passar alguma informação relevante ao motorista, desta forma é possível notar as grandes vantagens trazidas por esse tipo de conexão que pode evitar acidentes e melhorar as condições do trânsito. (**??**)

2.4.1 Aplicação

Como é apresentado em (**??**) hoje já é possível se ter exemplos da aplicação desta tecnologia, é o caso da empresa alemã Audi que está implantando nas próximas versões dos seus carros a tecnologia que ao parar em um semáforo inteligente, é exibido no painel do carro um temporizador indicando quanto tempo falta para abrir o semáforo, algo tido como não muito útil inicialmente, mas é só o começo do que há de vir, a ideia é avançar em busca de carros autônomos.

O funcionamento deste sistema se dá pela comunicação do carro com as centrais de tráfego, instaladas nos semáforos, estas por sua vez se comunicam com os servidores, os quais enviam a informação que o veículo necessita, ao receber

essas informações os veículos podem tomar ações, no caso de um carro autônomo (futuramente) ele poderia se preparar para uma parada no semáforo, trazendo maior economia de combustível.

Esta é uma tecnologia que tende a aumentar com o passar dos anos, pois no momento ainda é preciso que se reestruturem as cidades para que possa receber este tipo de tecnologia, como é o caso das centrais de tráfego, que atualmente não há este tipo de dispositivo instalado dentro das cidades, mas futuramente será algo necessário e que trará grandes benefícios a população podendo gerenciar o trânsito de forma inteligente evitando congestionamentos e gerenciando de forma mais eficiente o tempo dos semáforos.

2.5 FUNCIONAMENTO DE DISPOSITIVOS DE IOT

Quando se fala no uso de tecnologias IoT logo se pensa em sensores conectados na rede captando dados, sendo assim é preciso se detalhar o que são estes dispositivos e como funcionam.

2.5.1 Sensores

Sensor é o termo para designar um dispositivo sensível a algum tipo de energia do ambiente, podendo ela ser luminosa, térmica, cinética, relacionado a uma grandeza física como temperatura, pressão velocidade, corrente, aceleração, etc. Normalmente o sinal de saída de um sensor deve ser manipulado antes de sua utilização, geralmente através do uso de um amplificador, pois as tensões de saída após o dispositivo ser sensibilizado costumam ser baixas. (**??**)

2.5.2 Transdutores

É o termo designado para se referenciar o dispositivo que transforma um tipo de energia em outra, trabalham geralmente junto com os sensores transformando o impulso elétrico vindo dos sensores em valores digitais úteis dentro de um sistema de IoT. Um exemplo de transdutor é o alto-falante que converte o impulso elétrico em movimento mecânico necessário para reproduzir o som. (**??**)

3 SISTEMAS DE BIG DATA

Analisando o contexto atual, da nova era da tecnologia, se tem em média 100 bilhões de transações de cartão de crédito por dia acontecendo ao redor do mundo, cerca de 200 milhões de e-mails enviados a cada minuto. Em 2011 a Digital Universe Study (IDC) levantou a seguinte estatística: que a quantidade de dados gerado por mês em 2010 era de 1 exabyte, e ainda projetou que até 2020 esse número cresça 50 vezes mais. Sendo assim a cada dia se produz mais dados de todos os tipos possíveis, estruturados, não estruturados e em diferentes formatos, principalmente após o surgimento do IoT, como já mencionado no capítulo anterior a quantidade de dados gerados por estes dispositivos é imensa. A tabela 3.1 mostra algumas medidas de dados para comparação com os dados gerados mundialmente. (**??**)

Tabela 3.1 – Grandeza de dados

Medida	**Representação Numérica**	**Exemplo**
Byte	1	Único caractere
Kilobyte	1000	Uma sentença
Megabyte	1000000	20 slides do PowerPoint
Gigabyte	1000000000	10 livros
Terabyte	1000000000000	300 horas de video em boa qualidade
Petabyte	1000000000000000	350 mil fotos digitais
Exabyte	1000000000000000000	100 mil vezes a biblioteca do congresso
Zettabyte	1000000000000000000000	Difícil de exemplificar

Fonte: **??**, **??** (Adaptado)

Tendo em vista todos esses dados gerados que até então não serviam para grandes coisas a não ser momentaneamente, eram apenas dados soltos, passou-se a empregar sobre eles uma análise, agrupando dados similares e os processando a fim de que através deles fossem possíveis retirar indicadores, gerar *insights*, criar produtos ou serviços e embasar decisões. É aí que surge o conceito de *Big Data*, que de forma simplista representa uma grande massa de dados, mas está mais relacionado com a grande quantidade e variedade de informações que esta massa de dados pode gerar. (**??**)

O conceito de *Big Data* ganhou força nos anos 2000, uma época de grandes avanços tecnológicos. Uma aplicação prática de como este conceito pode ser empregado, foi na reeleição de Barack Obama em 2012 para a presidência dos Estados Unidos, quando sua equipe de tecnologia se utilizou do *Big Data* para gerar estatísticas sobre os comentários a respeito do presidente e assim construir estratégias políticas,

sendo possível analisar o comportamento de seus eleitores, para que servissem como indicadores de tomadas de decisões políticas pelo presidente. (**??**)

E este é um dos exemplos mais conhecidos, mas empresas ao redor do mundo tem adotado a implantação de sistemas de *Big Data*, a fim de conhecer o seu público, saber o que os seus clientes estão falando da sua empresa, a fim de buscar informações para o desenvolvimento de novos produtos, tudo isso para alavancar seus lucros, mudando a forma de relacionamento com o cliente e entrega do produto. Esses dados podem ser extraídos de diversos lugares, o Google por exemplo se utiliza da localização GPS das pessoas para que elas avaliem estabelecimentos visitados, para que avisos sobre o trânsito e notícias de interesse por parte do usuário sejam enviados a ele. Desta forma além de benefícios dentro de empresas e governos, o *Big Data* pode trazer também uma melhor qualidade de vida ao usuário, lhe apresentando maiores facilidades no dia a dia, contribuindo com a sustentabilidade e criação de cidades inteligentes. (**??**)

3.1 GESTÃO DOS DADOS

Com relação aos dados que são utilizados para compor o *Big Data*, estes podem ser advindos de diversas fontes e conter os mais variados tipos, podendo ser dados estruturados de forma relacional, documentos de registros de atendimento, fotos, vídeos, que podem ser gerados por máquinas, sensores, humanos, e na sua variedade podem conter dados de mídias sociais, dados gerados por fluxo de cliques de interação em *websites*, dados de localização gerado pelos dispositivos móveis, entre muitas outras fontes que ainda estão por vir.

O grande desafio tem se tornado agrupar esses dados e encontrar alguma forma com que eles façam sentido, para que a partir de então indicadores capazes de apontar estratégias possam ser gerados. Tendo em vista todos esses dados e a forma com que estão estruturados, nesta nova onda de gestão de dados, é impossível pensar em administrá-los de forma tradicional, para isso foi preciso evoluções não só em software mas também em hardware, além de rede e modelos de computação como virtualização e computação em nuvem. Só desta forma essa massa de dados pôde ser utilizada com maior eficiência. (**??**)

O *Big Data* não foi algo que surgiu agora, mas foi uma junção dos últimos cinquenta anos de evolução, o que se costuma dividir em três ondas de dados.

3.1.1 Primeira Onda

Em 1960 com a entrada da computação no mercado, os dados passaram a ser armazenados em arquivos simples, o que era pouco eficiente, pois para levantar dados

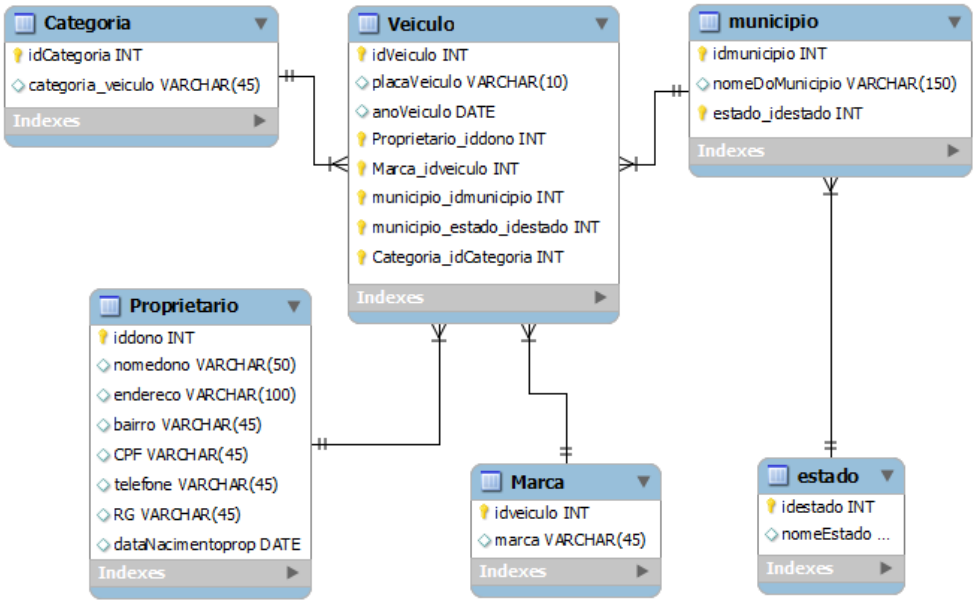
sobre clientes era necessário ir em busca de arquivos soltos, em 1970 com a criação do modelo relacional os dados precisaram ser convertidos para esse novo modelo, para isso utilizou-se de força bruta (trabalho manual), pois não havia uma forma de estruturar aqueles dados automaticamente.

3.1.1.1 Modelo Relacional

O Modelo Relacional vem da teoria dos conjuntos (álgebra relacional), representado por um conjunto de tabelas (entidade/relação), estas ainda se dividem em colunas (atributos) e linhas (tuplas), e são nessas tabelas que os dados são armazenados, tendo cada entidade separada em uma tabela, e estas tendo relações entre si através de colunas chaves, desta forma é possível realizar pesquisas em uma entidade isolada ou usar das relações entre elas para se pesquisar mais de uma entidade de uma única vez.

Quanto a entidade ela é a representação de um componente do mundo real sobre a qual se deseja guardar informações, elas são coisas significativas, como por exemplo: Cliente, Produto e Compra. Desta forma temos entidades que geram valor, armazenadas de forma tabular, tendo cada novo registro como uma nova linha dentro desta entidade e em cada linha uma sequência de atributos como sendo as colunas, a figura 3.1 apresenta um exemplo de organização no modelo relacional, com tabelas, atributos e relações. (**??**)

Figura 3.1 – Modelo Relacional



3.1.1.2 Linguagem SQL

Para trabalhar com o modelo relacional foi criado uma linguagem, chamada SQL (*Structured Query Language*), a qual geralmente eram escritas e executadas nos SGBDs (Sistema Gerenciador de Banco de Dados) e tinham a função de manipulação, definição e controle de dados, através de *querys*, que eram sentenças de comandos a fim de realizar uma ação sobre os dados armazenados. Ela mantinha um nível de abstração sobre os dados, além de ser útil para criar relatórios detalhados sobre o cliente, manter um controle de acesso aos dados, e todo o gerenciamento necessário, satisfazendo as exigências crescentes dos negócios. A figura 3.3 mostra uma *query* escrita em linguagem SQL que busca todos os dados da entidade veículo presente no banco de dados. (**??**)

Figura 3.2 – *Query* SQL

```
SELECT * FROM veiculo WHERE idVeiculo = 15;
```

3.1.1.3 Data Warehouse

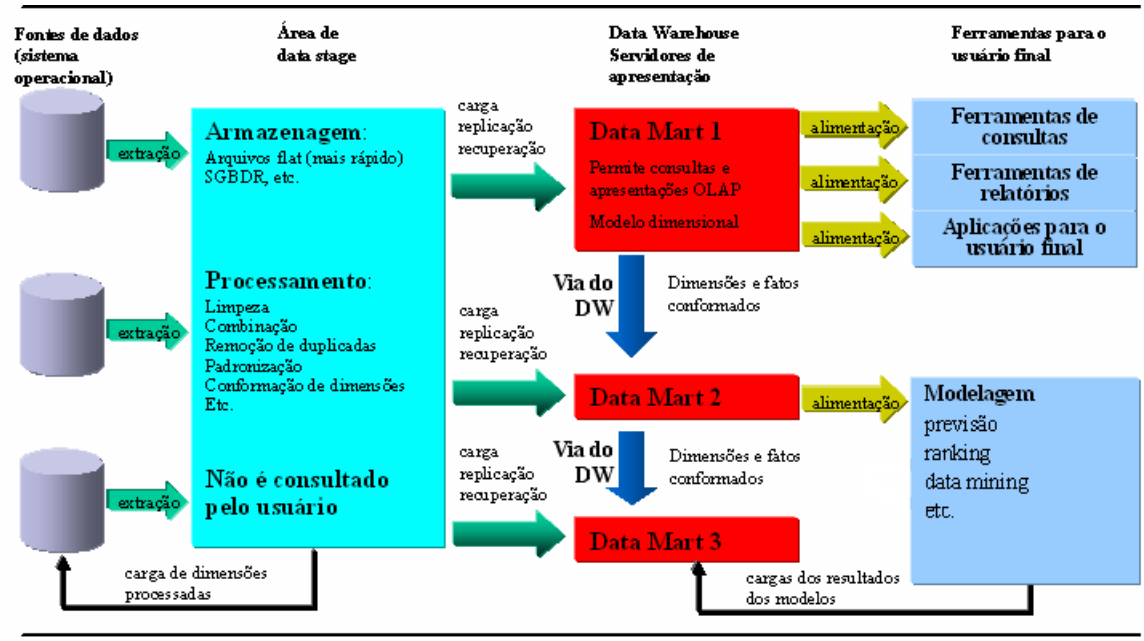
Com a adoção do modelo relacional, conforme os dados foram crescendo, criou-se em meados da década de 90 o *Data Warehouse* (DW) (armazéns de dados), que é um repositório de dados organizados por assunto, por exemplo em uma empresa de arrecadação de impostos, os assuntos que interessam a empresa são os cadastros de contribuintes e impostos a receber, sendo assim, esses dados são minerados do ambiente transacional da empresa (bancos que estão em produção na empresa), também conhecido como OLTP (*On-line Transaction Processing*), e vão para um DW (um banco a parte) conhecido como OLAP (*On-line Analytical Processing*).

Neste DW é preciso que se tenha integração com a base de dados original, a fim de padronizar os dados que se tem em ambas as bases de dados, é preciso também que ele seja não-volátil, para isso dentro de um DW, só são permitidas operações de consulta e exclusão, mantendo-o não volátil. Outra característica é que ele deve ser variável com o tempo, pois todo dado disponível dentro de um DW se refere a um determinado período de tempo, e esses dados não são atualizados em tempo real para que não afete o desempenho dos bancos transacionais.

Como resultado final de um DW se tem a obtenção de relatórios e estatísticas sobre aqueles determinados assuntos para o qual o DW foi criado, estes são alimentados pelos *Data Marts* que são divisões lógicas dentro do DW, geralmente dentro das empresas é a separação em departamentos, em que cada usuário verá

dados relacionados ao seu departamento. A figura 3 mostra a arquitetura básica de um DW. (**??**)

Figura 3.3 – Arquitetura de um *Data Warehouse*



Fonte: **??**, **??**

3.1.1.4 Banco de Dados Orientado a Objetos

Com a popularização de linguagens de programação orientadas a objetos, houve a necessidade de armazená-los da mesma forma com que eram manipulados, sendo assim criou-se o Banco de Dados Orientado a Objetos, uma alternativa ao modelo relacional que se mantêm forte até os dias de hoje. Partindo da premissa de que são armazenados objetos juntamente com seus atributos, mantendo seu estado de quando foram armazenados, o relacionamento entre os objetos se dá da mesma forma que no modelo relacional, se utilizando de chaves estrangeiras, e nas relações de "tabela-filha"neste tipo de armazenamento seria um atributo que tenha como valor outro objeto. Eles são armazenados serializados, sendo possível retornar ao seu estado quando são deserializados. (**??**)

3.1.2 Segunda Onda

Na década de 1990 com o surgimento da *web*, deixou de se armazenar apenas documentos, para se armazenar áudio e vídeo também, fazendo com que novas tecnologias surgissem para dar conta da nova demanda, criando assim na área da gestão de dados um modelo mais unificado, os metadados. (**??**)

3.1.2.1 Metadados

De forma simplista é possível dizer que metadados são dados que descrevem dados, ou seja ele mantêm uma descrição concisa sobre o tipo do dado armazenado, seja ele tabela, imagem, gráfico ou vídeo. O banco de dados se utilizam de metadados para armazenar o tipo de dado que contêm nas tabelas por exemplo, as classes de objetos são metadados dos objetos, entre muitos outros exemplo. Mas o que é importante destacar é que se os dados não estão documentados, é possível que haja inconsistências, além de maior dificuldade de manutenção desses dados. (**??**)

3.1.3 Terceira Onda

Na terceira onda está compreendido a era atual, em que a gestão de dados evoluiu para o *Big Data*, fase na qual se tem dados como foto, vídeo e áudio a serem armazenados, desta forma a alternativa encontrada foi virtualizar esses dados e armazená-los em nuvem, pois avanços com relação a velocidade de rede e confiança, removeram muitas limitações físicas da capacidade de administrar quantidades massivas de dados, alinhado com a sofisticação de memórias de computadores, possibilitando assim o avanço para essa nova era.

Muitas das tecnologias empregadas no *Big Data* existem a anos, como virtualização de dados, processamento paralelo, sistema de arquivos distribuídos e base de dados *in-memory*, mas se tem algumas das tecnologias que foram criadas agora para atender o que o *Big Data* se propõem a entregar, como o Hadoop e MapReduce. Não só na área privada, mas o ramo da ciência, pesquisa e o governo tiveram uma grande parcela no incentivo para o desenvolvimento dessas tecnologias, pois era necessário analisar o genoma humano, analisar os dados astronômicos e coletar dados para fins antiterroristas, o que são alguns dos exemplos de utilização nessas áreas.

Desta forma *Big Data* não vem a ser uma ferramenta ou tecnologia específica, mas sim um encontro de tecnologias diferentes que quando juntas promovem ideias certas, no tempo certo, baseadas nos dados certos, sejam eles vindos de qualquer fonte geradora de dados. As próximas evoluções tendem a ser através da junção de outras tecnologias que já existem e possam ser utilizadas de alguma forma a integrar com o *Big Data*. (**??**)

3.2 CARACTERÍSTICAS DO BIG DATA

Comumente são usadas três características para se descrever o *Big Data*, o que é chamado de 3Vs (volume, velocidade e variedade), ainda existem outras abordagens que tratam como 5Vs (volume, velocidade, variedade, veracidade e valor),

estas características mencionadas servem como pré-requisito para se caracterizar armazenamentos e análise de dados como *Big Data*.

3.2.1 Volume

A quantificação do volume de dados na computação é relativa, pois ela é dependente do tempo, o que hoje pode ser considerado um volume grande amanhã pode já não ser mais, por esse fato muitas mídias de armazenamento foram sendo substituídas por outras, pois a demanda de armazenamento cresce exponencialmente e novas formas para se armazenar dados precisam ser criadas. (**??**)

O *Big Data* tem a proposta de trabalhar com um volume de dados atualmente considerado grande o suficiente para que os modelos tradicionais não suporte o mesmo trabalho, dados como iterações em redes sociais, trocas de email diária, transações bancárias, entre outros. Estes são exemplos de dados hoje considerados muito grandes para se trabalhar de forma tradicional, até porque muitos desses dados não estão estruturados e trabalham com tipos de dados diferentes. (**??**)

3.2.2 Velocidade

> Você cruzaria uma rua vendado se a última informação que tivesse fosse uma fotografia tirada do tráfego circulante de 5 minutos atrás? Provavelmente não, pois a fotografia de 5 minutos atrás é irrelevante, você precisa saber das condições atuais para poder cruzar a rua em segurança. (**??**, p. 2)

É exatamente isso, nessa nova onda houve uma imersão de tecnologias *real time*, não que antes não existissem, mas com as novas tecnologias seu uso se tornou mais acessível, e para que ela funcione cumprindo seu propósito é necessário que haja velocidade, tanto no armazenamento, como na busca e na análise de resultados. É importante ressaltar que esse conceito de velocidade não está necessariamente relacionado com rapidez, mas sim com a exatidão do tempo de determinada ação, por mais que ela demore horas, o importante é ao final desse tempo pré estipulado, o resultado estar disponível. (**??**)

3.2.3 Variedade

Quando se fala de variedade dentro de *Big Data*, isto se refere a grande variedade de dados com que ele trabalha, além de virem de fontes diferentes, os mesmos possuem formatos diferentes, organizado em diferentes formas, por exemplo fotos, áudio, vídeo, informações de usuários, todas essas informações estarão organizados de forma estrutural, semi estrutural e não estrutural e o *Big Data* deve ser capaz de trabalhar com eles. (**??**)

3.2.4 Veracidade

Veracidade está diretamente relacionado com a velocidade, pois para os resultados obtidos serem verídicos, ou seja serem confiáveis, é necessário que ele pertença ao tempo que se pretende analisar, de nada vale uma análise de dados que gerou resultados de um tempo passado, é necessário o dado verídico no tempo esperado, caso contrário não será possível se utilizar dos resultados obtidos, por isso a velocidade influencia na veracidade. (**??**)

3.2.5 Valor

O valor está relacionado ao propósito de se fazer o uso do *Big Data*, pois antes da análise e obtenção de resultados, é necessário que se tenha questionamentos a serem respondidos através do uso do *Big Data*, ou seja, é necessário que o resultado obtido gere valor para o fim que ele está sendo usado, seja ele melhora no relacionamento da empresa com o cliente ou em áreas da ciência ou ainda do governo. (**??**)

3.3 PROCESSAMENTO DE DADOS EM TEMPO REAL

O termo *Big Data*, como já explicado anteriormente, é utilizado quando o grupo de dados que está sendo analisado é tão grande que não pode ser processado através de técnicas clássicas como utilização de banco de dados relacional (RDBMS ou *Relational Data Base Systems*. E a necessidade de velocidade em toda *pipeline* é acentuada ainda mais quando se pretende fazer processamento em tempo real dos dados obtidos.

Até 2014 uma das mais notáveis ferramentas para o gerenciamento de um grande volume de dados era o MapReduce, uma biblioteca desenvolvida pela Google que oferecia uma grande quantidade de ferramentas em um único pacote para lidar com muitos os desafios que se encontrava em sistemas *Big Data*. Logo após o anúncio do MapReduce pela Google saíram muitas bibliotecas *Open Source* com o mesmo objetivo, entre elas estavam Hadoop MapReduce e Haddop YARN.

Embora MapReduce faça com que o processamento de larga escala em dados complexos fiquem bem mais simples e eficiente ele ainda se trata de processamento em *batch*, e não era ideal para a demanda recente de processamento em tempo real de um alto volume de dados.

A solução para esse caso pode ser dividida em dois campos, a solução que tenta reduzir o gargalo das tecnologias como MapReduce e fazer com que ele execute seu trabalho mais rápido, e/ou uma segunda solução que provê meios para que as *queries* sejam executada tanto em banco de dados SQL quanto NoSQL. (**??**)

3.3.1 Computação em memoria

Muitos dos problemas tanto do MapReduce quanto do Hadoop é que seu sistema de processamento em *batch* não era otimizado para uma execução rápida, e sim para processar uma quantidade grande de informação.O Hadoop é muito dependente de um disco rígido para armazenar as informações isso adiciona mais um gargalo para o *startup* de uma tarefa *batch*.

Mesmo que a máquina esteja equipada com vários módulos de I/O, o problema ainda persistirá pois se trata de uma limitação do hardware e qualquer requisição de I/O ainda é muito lenta para operações em tempo real. Uma solução elegante seria armazenar toda a informação em sistemas de memória distribuída.

A memória pode oferecer uma alta quantidade de informação a mais de 10 GB/s. A latência no acesso a informação sendo na casa do nanosegundos enquanto a do disco fica na casa do milissegundo. Combinado com o preço da memória RAM, que vem ficando mais barato, torna o argumento de computação em memória muito mais viável. Há algumas soluções para computação em memória, entre elas estão; Apache Spark, GridGrain e XAP.

Computação em memória não significa que toda a informação tem que estar na memória, mas se somente uma parte da informação estiver em cache na memória já significaria um grande ganho de performance. (**??**)

3.3.2 Hadoop

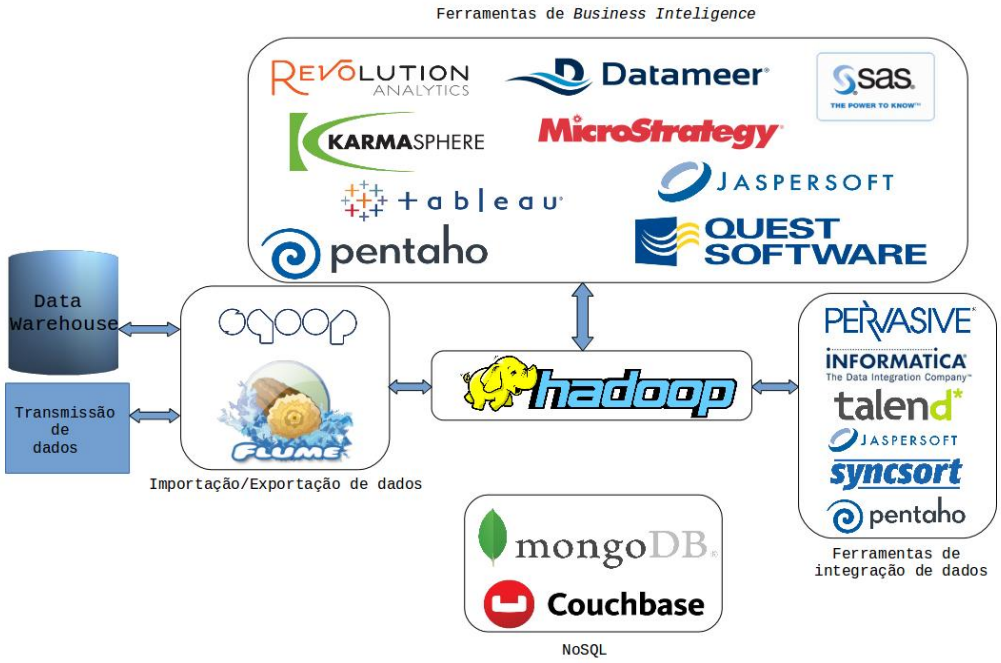
Hadoop em resumo é uma biblioteca para sistemas de processamento distribuído de grande quantidade de informações entre *clusters* de computares usando modelos simples de programação. Ele é desenvolvido para escalar de um simples computador para milhares, onde cada um pode oferecer sua própria unidade de computação e armazenamento. (**??**)

Hadoop pode resolver diversos problemas no campo de *Big Data* oferecendo velocidade e variedade como discutido nos 5Vs. O fluxo do Haddoop pode ser separado em 4 etapas: (**??**)

- **Captura dos dados:** as fontes de dados é extensiva e pode ser tando estruturada, semi-estruturada, e até *streaming* de dados (tempo real). Estendendo também para sensores e diversos dispositivos. Tendo integração com diversas fontes de dados como Flume, Storm, e outras fontes de dados do sistema Hadoop.
- **Processamento:** se trata de filtros e transformações que serão aplicados nos dados usando ferramentas baseada na lógica do MapReduce. Hadoop hoje

suporta uma quantidade grande de bibliotecas como MapReduce, Hive, Pig, Spark, Storm, etc como mostrado na figura 3.4.

Figura 3.4 – Sistemas que se integram com Hadoop.



Fonte: **??**, **??** (Adaptado)

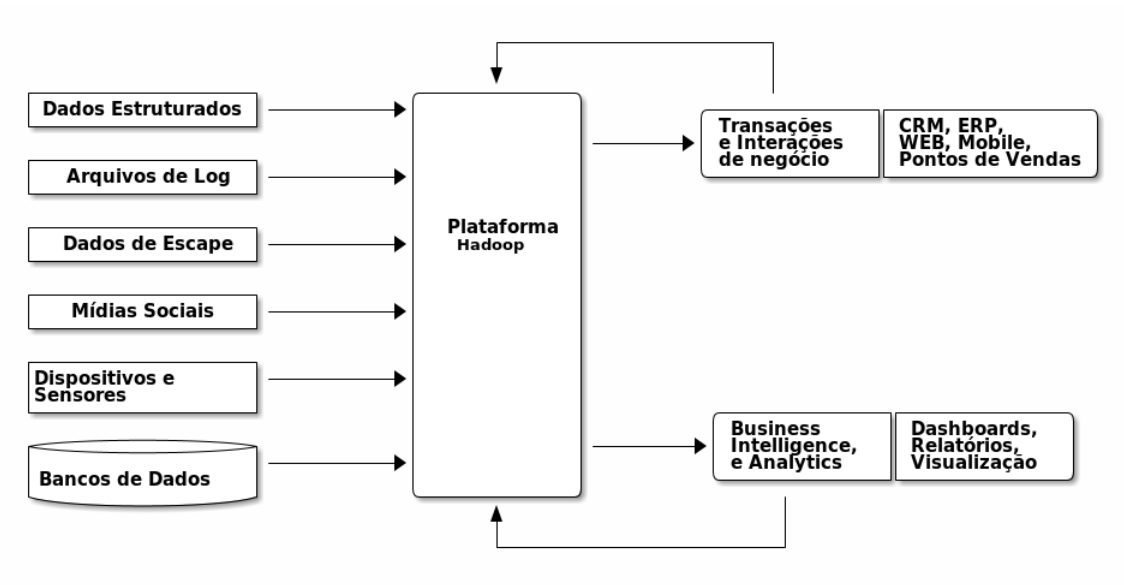
- **Distribuição:** após tratamento, os dados podem ser distribuídos para serem utilizados pela área de BI (*Business Intelligence*), e outros sistemas de análise podem consultar os dados.
- ***Feedback***: trata-se de analisar os dados e auditar para futuras melhoras no sistema *Big Data*.

A figura 3.5 mostra como o dado é capturado e então processado pelo Hadoop, e os resultados são utilizados em sistemas de BI. (**??**)

3.3.3 Apache Spark

Apache Spark é um sistema *cluster* de uso geral com foco em velocidade. Ele provê APIs em Java, Scala, Python e R, e um *engine* otimizada que suporta grafos de uso geral. Entres suas bibliotecas ele possui um grupo de ferramentas de alto nível incluindo Spark Sql, MLlib para aprendizado de maquina, GraphX para processamento de grafos e também Spark Streaming para processamento de dados em tempo real.

Figura 3.5 – Fluxo básico de informações do Hadoop.



Fonte: **??**, **??** (Adaptado)

Abstraindo o Spark em um alto nível pode-se dizer que cada aplicação Spark consiste de um *driver* que roda na função principal e executa diversas operações em paralelo em um *cluster*. Spark também provê entre seu grupo de ferramentas um *resilient distributed dataset* (RDD), que consiste em uma coleção de elementos particionados entre diversos nós de um *cluster*. Esses elementos particionados podem ser acessados de forma paralela. Normalmente os RDDs são criados a partir de um HDFS. Pode-se escolher também hospedar esse RDD em memória, deixando-o assim mais eficiente.

Uma segunda abstração são as variáveis compartilhadas que podem ser usadas em operações paralelas. Normalmente quando se inicia uma função em paralelo uma cópia das variáveis é usada em cada instância. Mas algumas vezes se quer compartilhar essas informações com todas as tarefas.

3.3.3.1 Spark Streaming

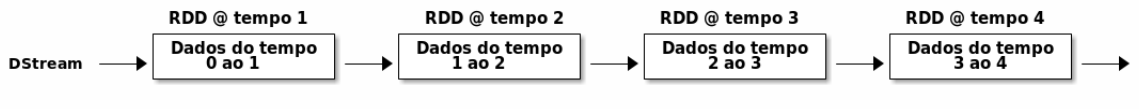
Spark Streaming é uma extensão do *core* do Spark Api que habilita escalabilidade, alta disposição, tolerância a falhas de *data streams*. Dados podem ser utilizados a partir de um grande número de fontes, como Kafka, Flume ou até mesmo *sockets* TCP como mostrado na figura 3.6. Esses dados podem ser processados utilizando algoritmos complexos que podem ser expressados em funções de alto nível como *map, reduce, join,* e *window*. Podendo ser aplicados outros algorítimos de aprendizado de máquina e processamento de grafos do próprio ecossistema Spark. (**??**)

Figura 3.6 – Exemplo de fonte de informação que o Spark Streaming suporta.

Fonte: **??**, **??**

Para o *streaming* de dados o Spark possui uma abstração chamada *discretized stream* ou *DStream*. *DStream* podem ser criado a partir das fontes citados acima (kafka, Flume, etc.) ou qualquer outra fonte utilizando operações de alto nível. Internamente *DStreams* são sequências de RDDs como demonstrado na figura 3.7.

Figura 3.7 – Representação de um *DStream*.



Fonte: **??**, **??** (Adaptado)

3.4 CASE DE SUCESSO COM BIG DATA

Big Data nos tempos atuais tem se tornado um diferencial no relacionamento das grandes empresas com seus clientes, pois ele garante uma nova experiência ao usuário, como é possível ver o exemplo da Netflix, grande produtora de conteúdos de entretenimento, como filmes e séries através de *streaming* de vídeo, em que cobra uma mensalidade de seus assinantes para acesso a estes conteúdos, com mais de 100 milhões de assinantes, eles se utilizam de um poderoso sistema de *Big Data* para indicar filmes e séries que sejam similares aos gostos dos usuários. Através deste sistema a empresa aprende informações sobre o seu cliente melhorando assim seu relacionamento para com o mesmo, já o cliente se sente bem por usar uma plataforma facilitada para os seus gostos.

Esta é uma das forma de como o *Big Data* vem sendo utilizado, muitas outras empresas o utilizam como o Google Now, Amazon, Waze e até o próprio governo

americano. Cada uma utilizando tal tecnologia focada na sua realidade, almejando diferentes objetivos, mas o que é possível notar em todas elas é que tal tecnologia tem mostrado bons resultados e ainda há muito a ser explorada antes de se tornar algo limitado como visto anteriormente com outras tecnologias. (**??**)

4 ESPECIFICAÇÕES E ARQUITETURA DO SISTEMA

Durante todo o desenvolvimento do sistema, buscou-se não utilizar dos modelos de arquitetura comumente empregados neste tipo de aplicação, que trabalham com grandes volumes de dados, um dos motivos vem a ser pela grande complexidade encontrada ao se trabalhar com módulos Apache como o Hadoop Common, Hadoop Distributed File System (HDFS), Hadoop YARN e Hadoop MapReduce, os quais dependendo de suas aplicações acabam por gerar dependências de outras ferramentas como Cassandra, Spark, ZooKeeper, entre outras que se mostram apesar de robustas, muito complexas e com uma curva de aprendizagem alta demais para serem usadas de imediato.

Outro motivo de grande valor para o desenvolvimento de uma nova arquitetura tem sido a empregabilidade de novas ferramentas utilizadas no lado *backend*, por assim dizer, relacionado a organização dentro dos servidores, conceitos de desenvolvimento e bancos de dados que tem ganhado força dentro do mercado e se mostradas bem aplicáveis em projetos com grandes volumes de dados e alta disponibilidade, além de possuir uma baixa curva de aprendizagem, se comparadas com as tecnologias tradicionais.

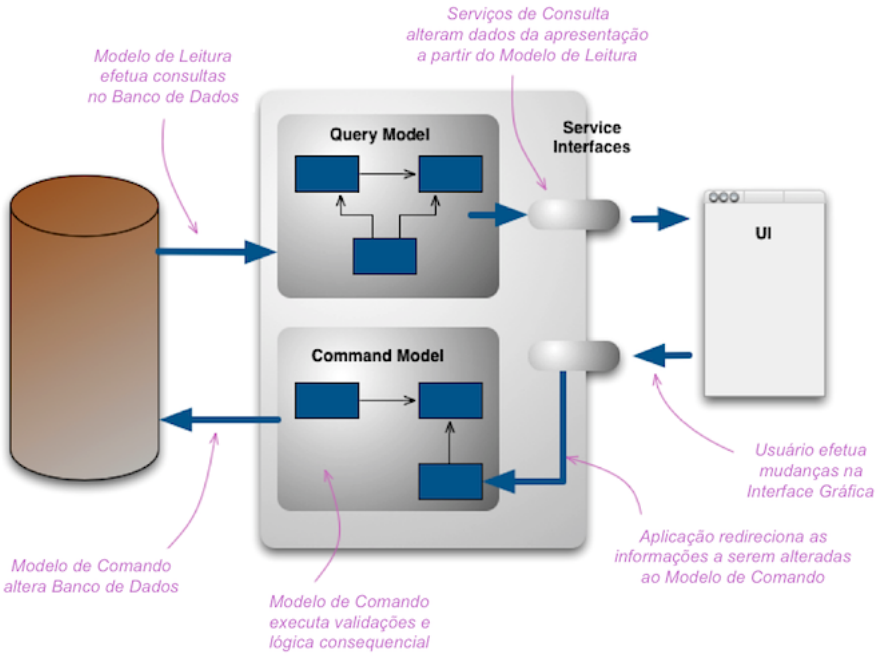
4.1 MICRO SERVIÇOS

Os Micros Serviços tem se tornado um padrão de projeto muito utilizado em aplicações de multiplataforma, pois através deste padrão este tipo de aplicação tem se tornado mais fácil de se desenvolver, demandando menos tempo. A ideia do padrão de Micro Serviços é desenvolver pequenos serviços autônomos que trabalham juntos, ou seja, um serviço possui uma única responsabilidade bem definida, mas para o funcionamento da aplicação por completa é necessário o uso de todos os serviços disponíveis dentro do domínio da aplicação. Sendo assim, um serviço pode ser um *endpoint* de uma *Application Programming Interface* (API), que pode retornar informações como todos os dados de uma tabela. (**??**)

4.2 CQRS

O *Command Query Responsibility Segregation* (CQRS) é um padrão de desenvolvimento que descreve como separar as operações de leitura e escrita em diferentes modelos, sendo que para a leitura é utilizada a *Query Model* e para outros comandos como atualização e inserção de dados é utilizado o *Command Model*, os quais compartilham da mesma base de dados, como mostra a figura 4.1.

Figura 4.1 – Arquitetura CQRS



Fonte: **??**, **??**

Este padrão foi desenvolvido tendo em vista que ao se criar novas regras de leitura e escrita, novas formas de representação das informações são criadas, o que já não faz mais sentido manter em um único modelo quando se fala em responsabilidade única, este é um padrão que adiciona grande complexidade ao projeto, portanto só deve ser utilizado caso as regras empregadas na leitura e escrita passem a ser complexas, não havendo um padrão fixo para todos os casos, é necessário adaptá-la ao problema que se deseja resolver. (**??**)

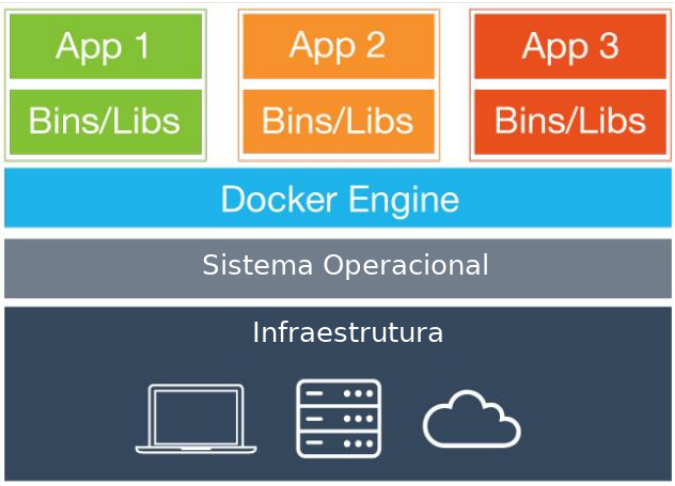
4.3 DOCKER ENGINE

O Docker é uma plataforma aberta criada com o intuito de tornar mais ágil o desenvolvimento, implantação e execução de aplicações em ambientes isolados, além de ser multiplataforma, tudo isso através do conceito de conteinerização e imagens, em que numa imagem se tem todos os arquivos necessários para a criação de um contêiner, este por sua vez, possui a aplicação encapsulada e funcionando.

Desta forma se tem uma grande flexibilidade para trabalhar com as aplicações, pois estas irão funcionar tanto em um notebook, quanto em um mainframe. Este conceito de contêiner se assimila ao processo de criação de máquinas virtuais, onde se tem todo o sistema operacional virtualizado, já o Docker realiza uma virtualização á nível do sistema operacional, mantendo um único kernel no *host* executando processos

(contêineres) de forma isolada, isso é possível através do uso de *namespaces*, fazendo com que um processo só tenha acesso a recursos de outro se isso for explicitamente configurado na criação dos ambientes. Para que não ocorra a exaustão de recursos no *host*, o Docker se utiliza de *cgroups* do kernel, responsável por criar limites de hardware para os processos, evitando o uso exagerado de recursos por apenas um processo. A figura 4.2 apresenta a organização a nível de sistema operacional de um *host* utilizando Docker. (**??**)

Figura 4.2 – Arquitetura Docker



Fonte: **??**, **??**

O Docker tem ganhado fama no mercado, principalmente pelos profissionais de infraestrutura, pois ele tem facilitado o trabalho destas pessoas, fazendo com que elas economizem tempo para realizar a instalação de um software, não necessitando mais se preocupar com todas as dependências e ferramentas necessárias para o funcionamento do software, basta apenas se utilizar uma imagem da aplicação e transformá-la em um contêiner em qualquer ambiente que se esteja. Comumente o Docker não é utilizado isoladamente, pois o que traz a ele grande facilidade ao se trabalhar com contêineres é a junção de todas as suas ferramentas, as quais serão descritas a seguir. (**??**)

4.3.1 Docker Compose

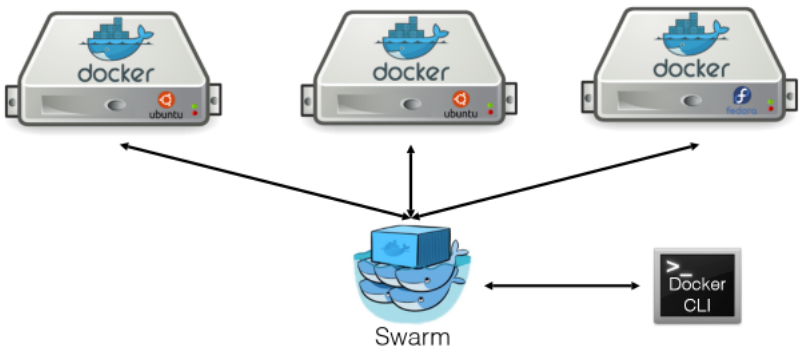
Com o aumento da complexidade do ambiente em que se está hospedada a aplicação, surgem as necessidades de se criar mais contêineres, tornando o seu gerenciamento um tanto quanto complicado, é neste cenário que se aplica o Docker Compose, uma ferramenta que trabalha juntamente com o Docker, ele se caracteriza por definir e executar múltiplos contêineres, pois através do uso de um arquivo que

descreve todos os contêineres e configurações necessárias para o funcionamento de uma aplicação por completa, o Docker Compose tem a capacidade de criá-los, reiniciá-los e desligá-los quando necessário, facilitando assim o uso de contêineres. (**??**)

4.3.2 Docker Swarm

Como visto até aqui, Docker Engine é o responsável pela arquitetura de criar e manter em funcionamento os contêineres, além de armazenar as imagens, o Docker Compose leva a responsabilidade da criação e remoção de múltiplos contêineres, facilitando o gerenciamento destes, por fim se tem o Docker Swarm a qual tem a função de gerenciar o *cluster* de contêineres, ele pode ser considerado como um orquestrador de contêineres como mostra a figura 4.3, especificando quais máquinas e contêineres farão parte deste *cluster*, sendo que muitos destes contêineres podem terem sidos criados a partir de um Docker Compose.

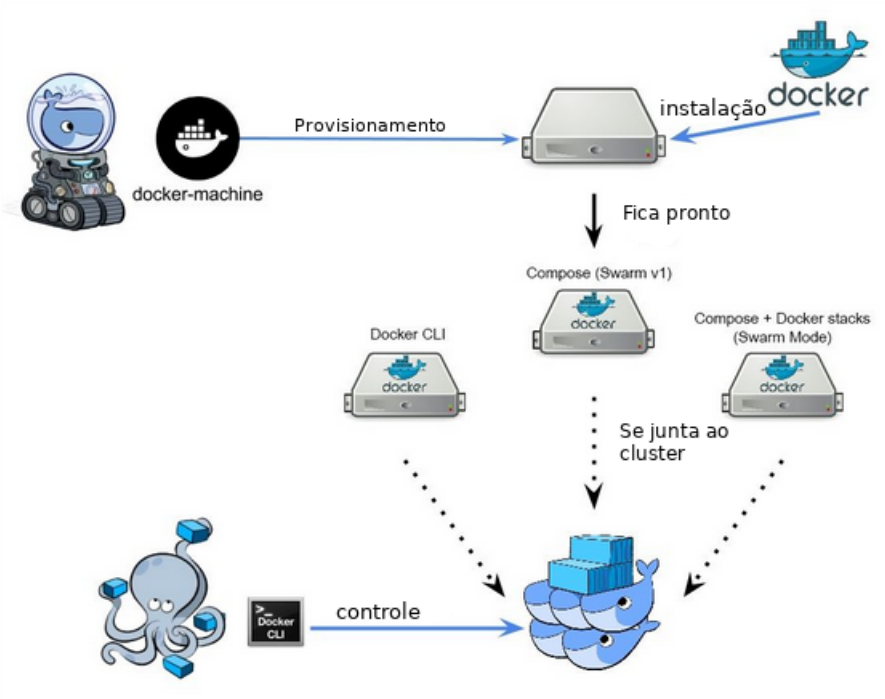
Figura 4.3 – Docker Swarm



Fonte: **??**, **??**

É o Docker Swarm que aplica todo o conceito de escalabilidade à aplicação conteinerizada, pois ela tem a capacidade de criar novas réplicas de contêineres de acordo com a necessidade, ou até os limites pré estabelecidos, é ele também que aplica o conceito de distribuição de cargas entre os contêineres que estão em funcionamento, podendo estes estarem na mesma máquina ou não, no caso de estarem em outra máquina o gerenciamento do Docker Swarm só é possível através da troca de *tokens* entre elas formando um *cluster* com contêineres homogêneos ou heterogêneos. É desta forma que é criado todo o ecossistema da arquitetura Docker, como pode ser visto na figura 4.4. (**??**)

Figura 4.4 – Ecossistema Docker



Fonte: **??**, **??**

4.4 REDIS

Redis é um banco de dados No-SQL destinado ao armazenamento em memória de dados no tipo chave-valor, utilizado para manter cache das aplicações, nesse cache podem ser armazenados *hashs*, *strings*, *bitmaps*, listas e *Hyperloglogs*, estes dados tem uma data de expiração após serem persistidos no Redis, por ser um banco de dados em memória, as ações de escrita e leitura se tornam mais rápidas do que uma leitura ou escrita em disco, dando maior performance ás aplicações nas quais ele é implantado. (**??**)

4.5 CLOJURE

Apresentada ao público em 2007 ela foi desenvolvida visando a construção de aplicações multitarefa, aproveitando os recursos da *Java Virtual Machine* (JVM), mas diferente do Java em que se escreve códigos orientado a objetos, em Clojure se tem códigos escritos em linguagem funcional, em que cada função é especializada em sua tarefa, sendo que a resolução de problemas se dá pela integração destas funções.

Algo que faz Clojure uma linguagem diferente das outras é que valores são imutáveis, isso resolve o problema que se tem de compartilhamento de memória quando usada várias *threads*, outra funcionalidade interessante é a criação de macros,

que é o equivalente a criar novas funcionalidades dentro da linguagem, algo muito útil para adequar a linguagem ao problema a ser solucionado. Além de tudo, assim como o Ruby possui o gerenciador de pacotes rake e o Java o Maven, Clojure possui o Leiningen, que gerencia os pacotes necessários para o seu funcionamento, além de disponibilizar um *prompt* de comando para teste de trechos de código, similar ao pip do Python. (**??**)

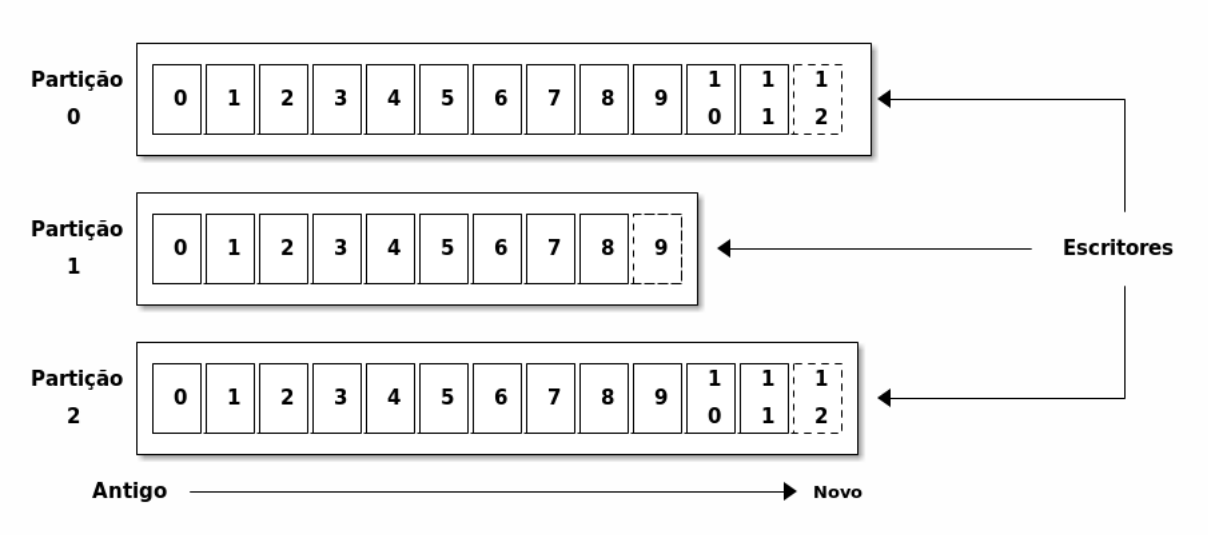
4.6 KAFKA

O Kafka se caracteriza por ser uma central de mensagens distribuídas, em que todas as informações passam por ele antes de alimentarem um banco de dados, Data Warehouse ou algum outro sistema de análise de dados, dentre as suas vantagens estão a sua rapidez, alta escalabilidade e redundância, permite uma grande quantidade de conexões persistentes dos usuários , além de apresentar uma grande resiliência tendo a capacidade de se recuperar após a ocorrência de falhas. Desta forma o Kafka se torna ideal para comunicação e integração de componentes em sistemas de larga escala.

Seu funcionamento se dá através de partições sobre tópicos diferentes (como se fossem mensagens relacionadas a cada assunto, ou ainda pode-se comparar as tabelas de um banco relacional) que podem estar em diferentes máquinas, estas partições armazenam uma fila de mensagens, tendo cada uma delas um número identificador imutável que segue a ordem da fila, um consumidor dessa fila de mensagens tem a possibilidade de ler qualquer mensagem que esteja na fila através o seu identificador, um consumidor também pode consumir mensagens de diversas partições como mostra na figura 4.5. As mensagens que chegam ao Kafka ficam armazenadas por um período de tempo configurável, não possuindo algo que informe se a mensagem foi lida ou não, apenas mantém pelo período estabelecido, caso o período seja menor ao período em que será feita a leitura pelo consumidor, essa mensagem será perdida

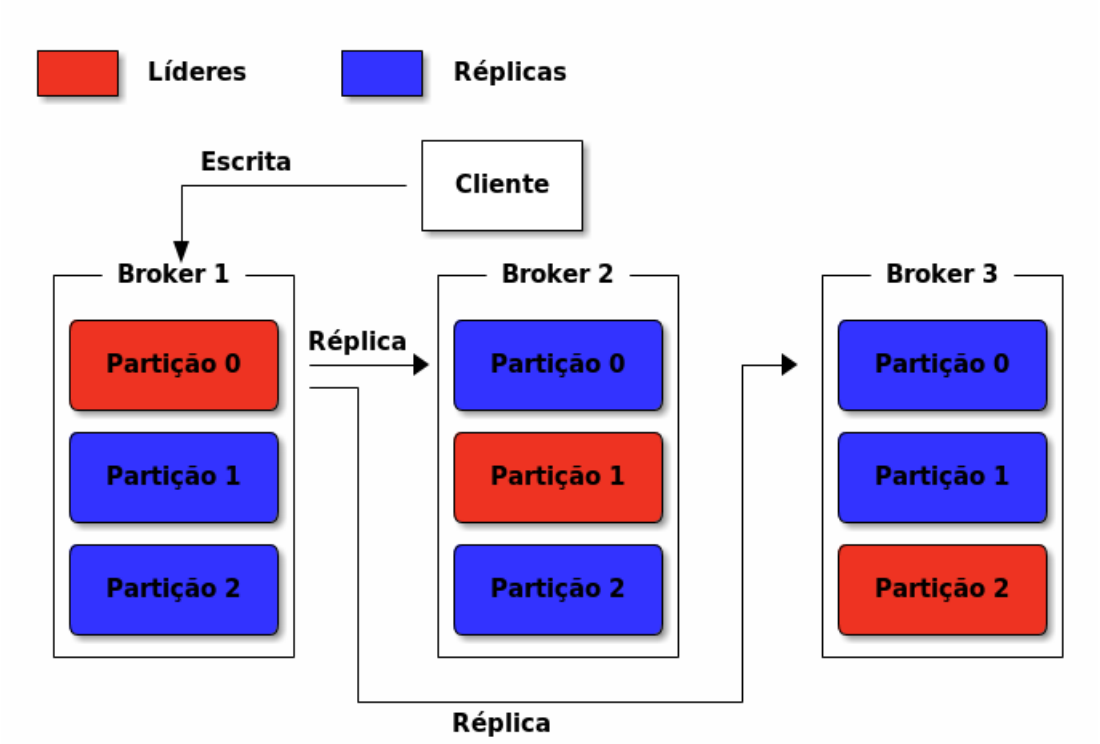
Além da organização das mensagens dentro de partições, se tem ainda a organização de partições dentro de *brokers*, onde se tem partições chamadas de líderes e réplicas, no processo de produção de mensagens como apresentado na figura 4.6 o produtor escreve na partição líder, esta então gera cópia nas partições réplicas que podem estar em outros *brokers* e em outras máquinas, desta forma se mantêm a consistência das mensagens e um alto nível de paralelismo, pois no processo de leitura o consumidor faz uma requisição para ler e a partição que está disponível retorna a mensagem para ele. (**??**)

Figura 4.5 – Partições do Kafka



Fonte: **??**, **??** (Adaptado)

Figura 4.6 – Processo de escrita do Kafka



Fonte: **??**, **??** (Adaptado)

4.7 MONGODB

MongoDB é um banco de dados orientado a documento, organizado em coleções de dados, aos quais se permitem operações de leitura e escrita, substituindo o conceito de linhas dos bancos relacionais por documentos, uma das grandes vantagens deste modelo é a fácil escalabilidade possível de se empregar, além da grande flexibilidade que se tem, pois não são pré-definidos identificadores, tamanhos ou tipos de dados para os documentos armazenados.

Dentre as funcionalidades dele se destacam a indexação através de texto o que mantêm uma boa performance na execução de *queries*, agregações complexas através de pedaços mais simples de código, tipos especiais de coleções suportando tamanhos fixos, suporta também o armazenamento de grandes documentos além de metadados. Seu funcionamento difere aos de bancos relacionais por não possibilitar as funções como *joins*, ou seja, para utilizá-la é preciso que se armazene dados completos em um único objeto, pois quando se tenta utilizá-lo similar a um banco relacional toda a *performance* da ferramenta quanto as buscas se perde. (**??**)

4.8 DATOMIC

Assim como o MongoDB ele é um banco de dados não relacional, mas com um conceito diferente, adaptado para as novas demandas de serviços em nuvem com arquitetura *Software as a Service* (SaaS), Datomic é um banco de dados orientado a fatos, com o grande diferencial de que dados nele persistidos se tornam imutáveis, isto não significa que não se pode alterar dados, mas o que acontece é um versionamento dos dados, desta forma é possível se ter um histórico do objeto persistido com todas as alterações já realizadas sendo a última versão o dado de maior veracidade.

Possui como características separação dos conceitos de leitura e escrita, forte garantia transacional na escrita, noções de imutabilidade expressas por bases de dados estritamente incrementais, utiliza para as consultas de dados a linguagem Datalog, como é mostrado um exemplo da linguagem no quadro 4.1 que é estruturada baseada em lógica permitindo consultas complexas incluindo *joins* inferidos. (**??**)

4.9 LOCUST

Locust é uma ferramenta *open source* desenvolvida em Python que visa realizar testes de carga e *stress* simulando conexões de usuários reais, para estes testes podem ser usados máquinas *slaves* tendo um teste de forma distribuída, a configuração é feita através de um *script* em Python também, onde são definidas as URLs a serem testadas, e por meio de uma interface gráfica WEB é possível configurar a quantidade

Quadro 4.1 – Exemplo da linguagem Datalog

```
        [:find ?cliente ?produto
         :where [?cliente :enderecoEntrega ?endereco]
                [?endereco :cep ?cep]
                [?produto :produto/peso ?peso]
                [?produto :product/preco ?preco]
                [(Entrega/estimativa ?cep ?peso) ?custoEntrega]
                [(<= ?preco ?custoEntrega)]]
```

Fonte: **??**, **??**

de usuários a serem simulados, quantidade de requisições a serem feitas, além de apresentar gráficos estatísticos sobre os testes. (**??**)

4.10 ARQUITETURA DO SISTEMA

Durante o desenvolvimento do projeto buscou-se utilizar do padrão de micro serviços, com cada um deles encapsulado dentro de um contêiner com responsabilidade única, sendo utilizado um arquivo Docker Compose para sua criação, e o Docker Swarm para realizar todo o gerenciamento de carga da aplicação. A escolha de se utilizar micro serviços conteinerizados se deu pela grande facilidade com que se pode criar, remover e escalar toda a aplicação, por exemplo, se em algum momento um serviço em específico está sendo mais utilizado que os demais, automaticamente podem ser criadas réplicas deste serviço para atender as demandas, sem contar o aumento do paralelismo que se pode ter com o uso de contêineres, pois cada um deles se apresenta como um processo isolado dentro de um servidor, tendo assim a velocidade que um sistema de *Big Data* espera, dentro do sistema em questão. (**??**)

Utilizou-se também do padrão CQRS para a execução das operações de escrita e leitura, utilizado por se tratar de um sistema ao qual as regras tendem a aumentar sua complexidade, e assim estando no padrão CQRS a aplicação já estará pronta para suportar estas demandas. Para conhecimento da arquitetura por completa será analisado o fluxo de dados seguido pelo dado objeto apresentado no quadro 4.2, o qual é criado assim que um carro é ligado e faz uma requisição para se conectar a infraestrutura (iniciar uma nova sessão) no serviço de *Command*.

Quadro 4.2 – Objeto criado ao se ligar um carro

```
{
        "hd_id": "abcd1234",
        "model": "zyz",
        "brand": "123"
}
```

4.10.1 *Command*

Operações de leitura, escrita e remoção passam por este serviço, que validará com as regras de negócio as informações recebidas e se passarem por esta fase, as elas serão inseridas em forma de texto na fila de mensagens do Kafka para serem inseridos no banco de dados juntamente com um identificador da sessão criada para este carro, o dado enviado ao Kafka é apresentado no quadro 4.3.

Quadro 4.3 – Objeto inserido no Kafka

```
{
        "command": "create_session",
        "session-id": "{uuid}",
        "hd-id": "abcd1234",
        "model": "zyz",
        "brand": "123"
}
```

4.10.2 *Worker*

Após a inserção da mensagem na fila do Kafka um outro serviço chamado de *Worker* que monitora se existem mensagens a serem lidas no Kafka, irá ler esta mensagem e salvá-las no banco de dados, neste caso por ser uma informação apenas de sessão será salvo apenas no MongoDB, mas para o caso de um novo cadastro de veículo ou alteração do mesmo, este serviço salvaria primeiramente no Datomic, o qual armazena dados históricos e posteriormente no MongoDB, o qual terá a informação imediata sobre o veículo, juntamente com esta informação de sessão o serviço de *tracking* estará atualizando dados referentes a localização do veículo, a qual será acrescentada no mesmo objeto apresentado no quadro 4.4.

Quadro 4.4 – Objeto inserido no MongoDB

```
{
        "session-id": "{uuid}",
        "hd-id": "abcd1234",
        "model": "zyz",
        "brand": "123",
        "location": {"x": 46.121, "y": 12.0213},
        "gas-lvl": 74
}
```

Esta estratégia de se salvar em dois bancos de dados faz com que Datomic fique armazenados todas as informações relevantes para se manter um histórico do veículo, mantendo a veracidade de dados sobre a entidade, pois o Datomic é um banco de dados que garante esta responsabilidade através do histórico e a impossibilidade de se alterar dados, já no MongoDB se tem os dados em tempo real e como pôde ser

visto a grande vantagem do banco não relacional está na rapidez com que esse dado poderá ser buscado, pois em um banco relacional provavelmente se teria uma tabela para localizações e uma para dados do veículo, necessitando de funções de agregação de dados para se ter o objeto completo, desta forma, no banco não relacional todas as informações já estão agrupadas e presentes em um único objeto, se ganhando muito em velocidade.

4.10.3 *Tracking*

O serviço de *tracking* é o responsável por receber informações do veículo, como posição e velocidade, dentre algumas informações sobre o estado do carro, e colocá-las na fila de mensagens do Kafka, para serem inseridos pelo serviço *worker* no MongoDB, juntamente com as informações de sessão do carro.

4.10.4 *Query*

Finalizado o processo de inserção de dados, o processo de leitura se dá pelo serviço de *query*, o qual recebe a requisição do usuário e busca dados diretamente no MongoDB retornando assim as informações requisitadas, consultas realizadas no Datomic estão relacionadas com as análise de dados do Big Data, pois é lá que se formará a grande massa de dados possíveis de serem analisadas com o decorrer do tempo.

4.10.5 Interface do Usuário

Esta vem a ser o dispositivo implantado no carro que tem a função de se ligar aos sensores do carro, a fim de mandá-las para a infraestrutura, juntamente com a localização instantânea do veículo. Esta interface não é contemplada pelo projeto, mas é essencial para o seu funcionamento prático.

4.10.6 Visão Geral

De forma geral se tem um fluxo de dados para inserção como visto na figura 4.7 e o fluxo de dados para leituras apresentadas na figura 4.8, a estrutura geral de toda arquitetura é representada na figura 4.9. Desta forma buscou-se por uma arquitetura que atenda a integração da interface de usuário, vinda de um dispositivo de IoT e que atenda também as características estipuladas pelo *Big Data*.

Figura 4.7 – Fluxo para inserção de dados pela arquitetura

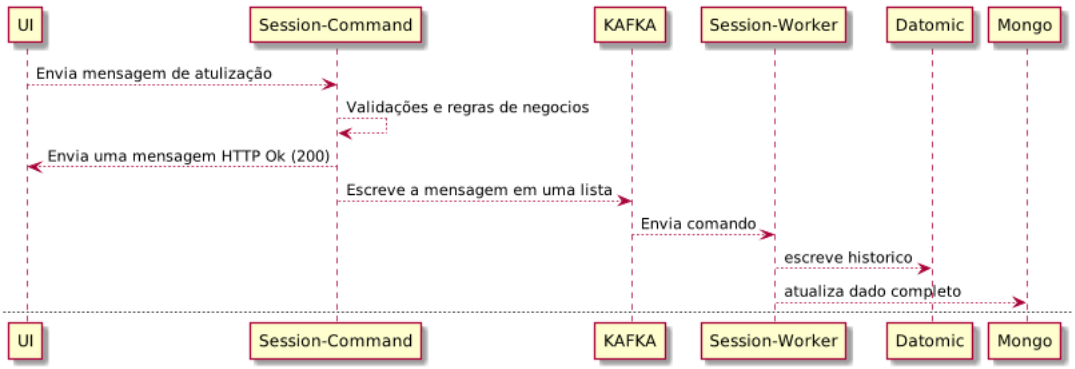


Figura 4.8 – Fluxo para leitura de dados pela arquitetura

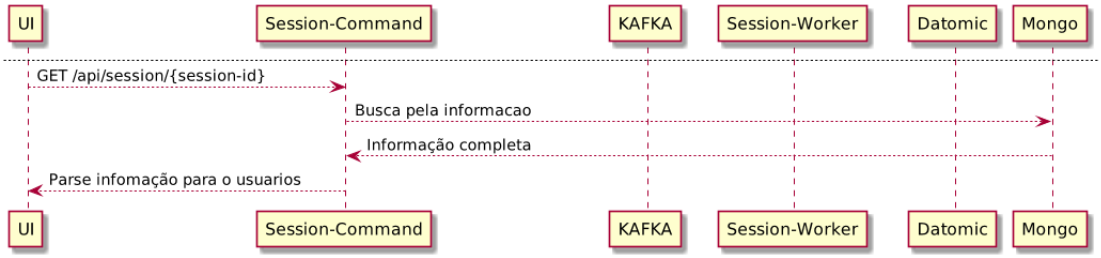
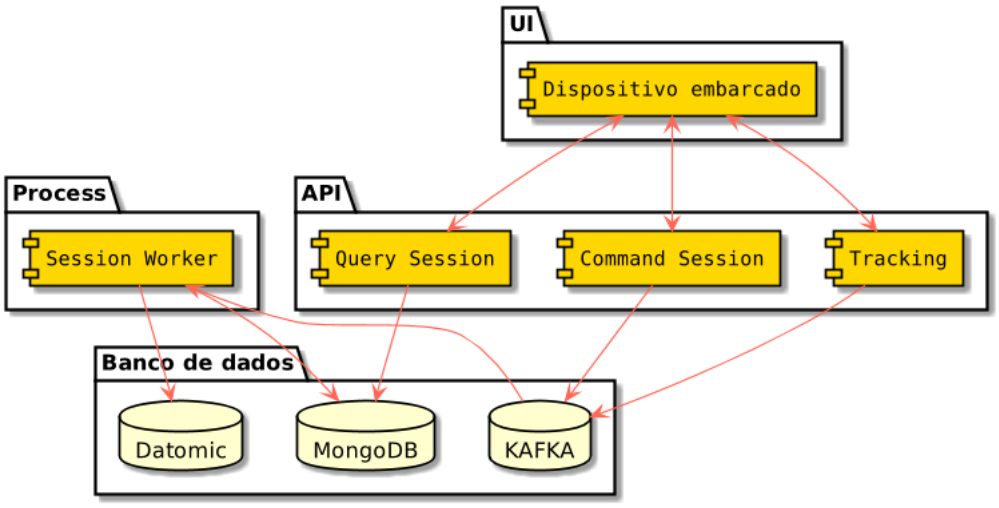


Figura 4.9 – Estrutura Geral da Arquitetura



5 RESULTADOS

Inicialmente para criação da arquitetura especificada na seção 4, criou-se o serviço de *Command*, o qual é a porta de entrada entre a interface de usuário e a arquitetura, este serviço tem a função de receber os dados da interface (dispositivo embarcado), validá-los e inserir na fila de mensagens, a qual foi montada se utilizando o Kafka, é através deste micro serviço que um carro terá a sua sessão criada (*Command Session*), informará dados sobre a localização (*Command Track*), ou ainda informações de alerta (*Command Warning*), para isso, como pode ser observado no apêndice A, se tem as validações de campos para cada um dos modelos de dados, no caso das informações referentes a sessão se tem informações sobre a marca, modelo, placa e proprietário, no modelo de dados de localização se tem o identificador da sessão, nível de combustível, latitude, longitude e velocidade, e no modelo de dados referentes a alertas o identificador da sessão e o alerta emitido pelo veículo. Todos estes dados são manipulados e recebidos no formato *JavaScript Object Notation* (JSON) por questões de velocidade e uso de memória. Ainda no apêndice A é apresentado todos os comandos de envio para a fila do Kafka de cada um dos modelos de dados.

Seguindo a arquitetura, se fez necessário a configuração da fila de mensagens (Kafka), a qual tem a função de armazenar uma informação por um determinado tempo até que ela expire depois de um tempo pré configurado. Para isso utilizou-se imagens Docker vindas de repositórios oficiais, sendo necessário um contêiner com a imagem do Kafka Zookeeper, o qual é o sistema responsável por gerenciar os *brokers* do Kafka, e outro contêiner com a imagem do Kafka Broker, no qual se tem os *brokers* propriamente dito, cada um funcionando em um contêiner separado.

O próximo passo que se tem dentro da arquitetura é a criação do micro serviço *Worker*, o qual tem a finalidade de ler dados da fila de mensagens e armazená-las nos bancos de dados, para isso este micro serviço foi criado em Clojure, contendo a conexão com o Kafka, além das conexões com os bancos de dados MongoDB e Datomic, o apêndice B mostra a retirada de dados da fila de mensagens e gravação dos dados nas duas bases de dados, de acordo com o tipo de informação contida na mensagem.

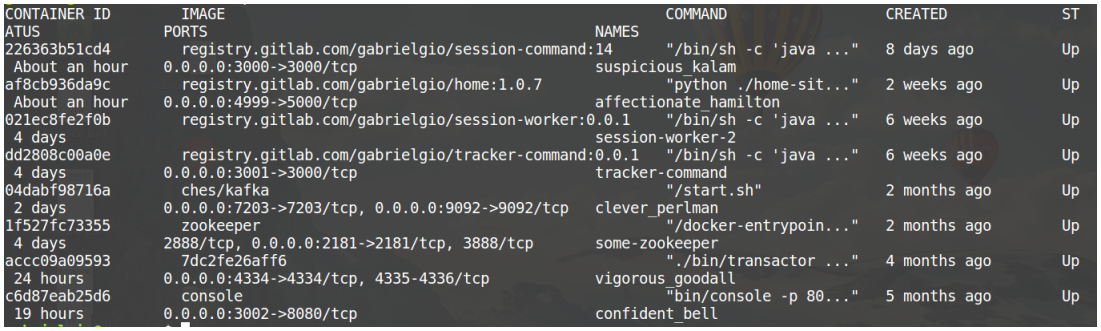
Por fim foram configurados os bancos de dados, os quais armazenam os dados vindos dos serviços de *Workers* e os mantém para consultas instantâneas no caso do MongoDB, ou consultas históricas no caso do Datomic. Para os dois bancos de dados utilizou-se de imagens Docker oficiais com a base de dados armazenadas no *host*, para que os dados fiquem disponíveis para todas as réplicas, fazendo a função de *pipeline* entre os contêineres, mesmo quando estes sejam replicados.

Tendo os dados armazenados, para o processo de leitura destes dados criou-se o micro serviço *Command Query* também em Clojure, o qual tem a função de ler os dados inseridos no MongoDB, sem passar pela fila de mensagens, diretamente ele realiza a função de busca no banco de dados de acordo com a informação requisitada, retornando para o usuário que requisitou a informação que passará a ser exibida na interface do usuário.

Todos os micro serviços utilizados dentro da arquitetura apresentam-se encapsulados dentro de contêineres, estes porém, possuem a base inicial vinda de uma imagem Docker oficial, no caso destes micro serviços utilizou-se a imagem Java com *Java SE Runtime Environment* (JRE) em sua versão 8, sendo assim, as alterações e configurações para o ambiente desejado se fizeram através do Dockerfile, que ao ser construído gera uma nova imagem com as alterações nele contidos, e para criação do contêiner basta especificar que a imagem a ser usada será esta nova imagem.

Tendo as imagens criadas, para facilitar o processo de instalação de toda a arquitetura no servidor desejado criou-se o arquivo Docker Compose, através dele são especificados os contêineres que serão instanciados, bem como as portas e variáveis de ambiente para cada micro serviço, é função dele manter o funcionamento dos contêineres e reiniciá-los em caso de falha, através dele também foi possível maior facilidade quando necessário parar ou reiniciar algum serviço. Como mostrado no apêndice C, são criados os contêineres para os micro serviços tendo conexões para os bancos de dados e a fila de mensagens, desta forma no servidor em que é instalado a arquitetura é possível observar o *cluster* de contêineres funcionando, como apresentado na figura 5.1.

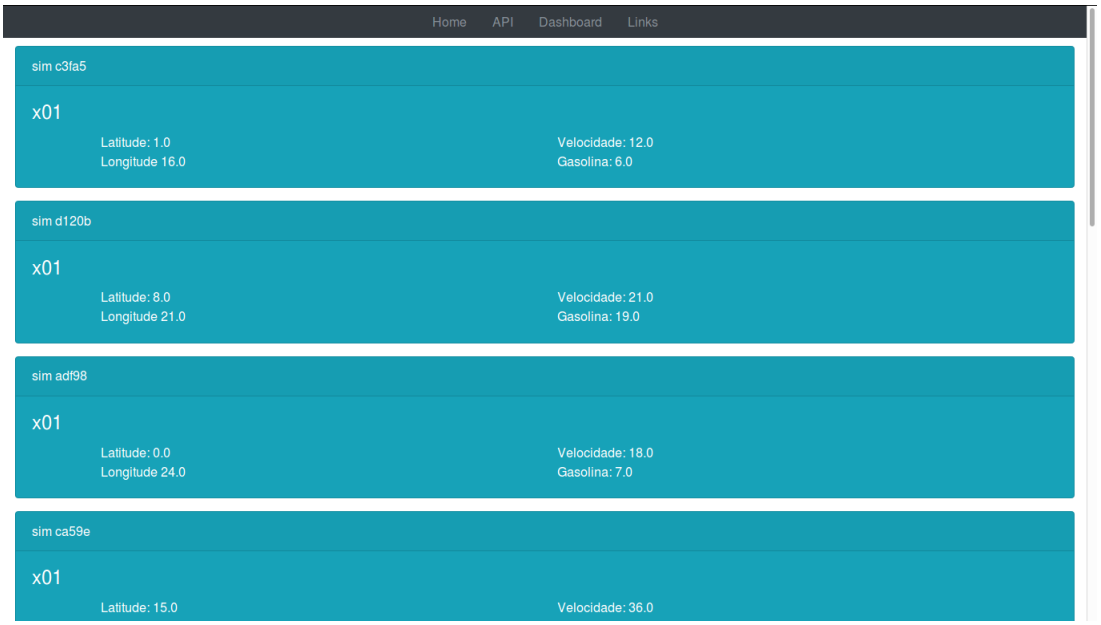
Figura 5.1 – Cluster de Contêineres



Para verificação do fluxo de dados e validação da arquitetura criada, desenvolveu-se uma aplicação em Python, a qual apresenta uma interface para visualização dos veículos conectados a infraestrutura em tempo real, além das informações enviadas através destas conexões, se utilizando de *sockets* para isso, tal aplicação se tornou útil para verificação das simulações que serão descritas na seção 5.1, pois através desta

foi possível a visualização dos dados simulados de forma simples, como pode ser visto na figura 5.2.

Figura 5.2 – Visualização dos clientes



5.1 TESTES DE SISTEMA

Para testes da arquitetura, realizou-se a instalação da mesma em um servidor no provedor de serviços *Server as a Service* (SaaS) Azure, com uma máquina de 8 GB de memória e 2 núcleos de CPU, então para testes de carga na arquitetura, bem como teste de integridade de dados mesmo sobre alta pressão, foi desenvolvido um *Mockup*, cujo o conceito é simular funcionalidades do sistema para que estas possam ser testadas de forma independente, sendo assim, dentro desta arquitetura os objetos *mock* foram os carros e unidades de emergência, os quais tiveram seu comportamento simulado via *software*, a fim de validar o armazenamento e fluxo de dados na arquitetura.

O *Mockup* realizou-se por meio de um micro serviço escrito em Clojure, apresentado no apêndice D, o qual cria uma quantidade de conexões pré-estabelecidas com o micro serviço *Session Command* e envia dados fictícios como a criação de uma sessão, localização e informações referentes a nível de combustível e alertas em cada conexão criada, destas informações, nível de combustível e campos texto na criação da sessão se fizeram de forma aleatória, já a latitude e longitude se fez de forma a apresentar rotas reais para que estes pudessem receber os alertas das unidades de emergência, como mostra no apêndice D, sendo assim, foi possível analisar o fluxo de dados dentro da arquitetura e verificar a veracidade dos dados armazenados, além de

seu volume, como mostra a figura 5.3 com dados armazenados no MongoDB e a figura 5.4 com dados do Datomic.

Figura 5.3 – Dados armazenados no MongoDB

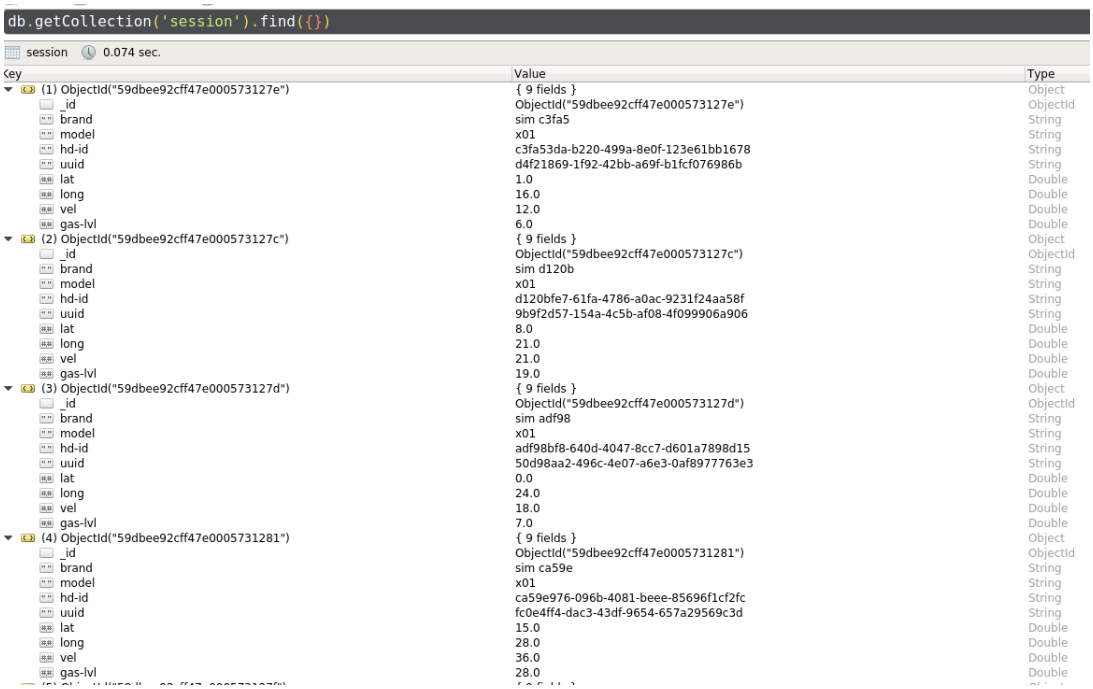
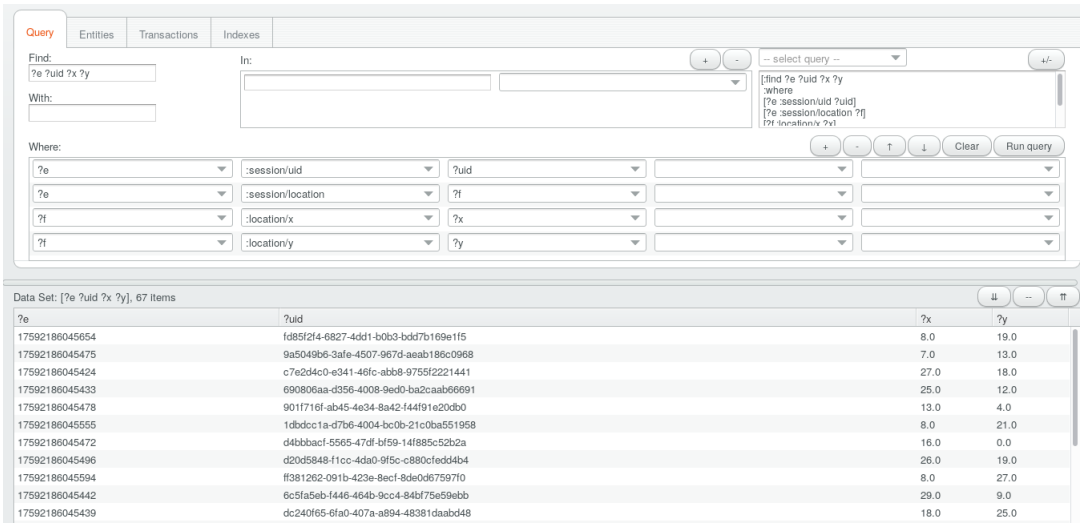


Figura 5.4 – Dados armazenados no Datomic



Para testes de *stress* se utilizou da ferramenta Locust, a qual teve a função de verificar a quantidade de usuários e conexões tal arquitetura suportaria, com os testes feitos se utilizando de uma máquina mestre e mais três escravas chegou-se aos resultados presentes na tabela 5.1, em que é apresentado as quantidades de usuários, tempo de resposta, quantidade de requisições e requisições por segundo, se utilizando

de até dois mil usuários, a cima disso o servidor líder e os escravos não suportavam simular mais usuários para gerar mais requisições na arquitetura, passando a falhar o servidor de testes e não a aplicação onde se encontra a arquitetura. Juntamente com a tabela a figura 5.5 apresenta um gráfico com as requisições e tempo de respostas.

Tabela 5.1 – Resultados de teste de *stress*

Usuários	**Requisições**	**Falhas**	**Respostas/s**	**Resposta(ms)**	**Requisições/s**
50	1077	0	22	26	10.72
100	2133	0	20	21	21.55
150	2200	0	20	21	32.49
200	3831	0	22	27	43.29
250	5140	0	22	22	53.43
300	6636	0	22	26	65.12
350	6813	0	20	26	77.75
400	8188	0	21	27	89.04
450	8833	0	23	26	99.77
500	11173	0	21	25	111.33
550	13415	0	21	24	122.2
600	12980	0	22	26	134.18
650	14046	0	22	25	145.03
700	14451	0	22	26	155.15
750	15740	0	21	23	168.54
800	18473	0	22	23	178.27
850	16620	0	21	22	188.64
900	30161	0	21	23	199.67
950	13915	0	21	22	210.73
1000	13716	0	21	22	220.92
1100	20624	0	7	7	241.8
1200	14092	0	7	7	266.02
1300	5854	0	8	7	288.79
1400	18807	0	8	7	309.91
1500	15493	0	8	7	333.22
1600	10259	0	8	7	360.37
1700	20056	0	8	7	377
1800	15885	0	8	7	401.51
1900	21434	0	8	8	427.13
2000	15587	0	8	8	444.88

5.2 DISCUSSÃO DE RESULTADOS

Através dos resultados apresentados é possível verificar que a arquitetura se adéqua ao propósito do seu desenvolvimento, apresentando uma alta escalabilidade e facilidade em sua implantação, através das ferramentas empregadas, quanto aos testes pode-se verificar que a arquitetura é válida quanto ao uso dela em uma análise

Figura 5.5 – Gráfico de resultados do teste de *stress*

de *Big Data* se utilizado o banco de dados Datomic, e se prova ter rapidez quando trabalhado com os dados do MongoDB.

A arquitetura se provou robusta mesmo quando funcionando em uma máquina de dois núcleos, possivelmente se utilizando de mais réplicas da fila de mensagens e do serviços de *worker* em outros servidores, os resultados seriam melhores, pois durante os testes o que mais sofria para o atendimento das requisições era os serviços de *worker* que por gravarem dados no banco se tornam mais lentos.

No teste de *stress* a arquitetura também se saiu bem, suportando uma pelo menos dois mil usuários conectados, que foi o número capaz de ser calculado pelos recursos que se possuíam para tal teste, sendo que esse número pode aumentar ao passo que se distribui mais a arquitetura em outros servidores, algo fácil de ser realizado, pois desde o início do trabalho o objetivo foi criar algo que se pudesse se escalar facilmente.

Com relação ao seu desenvolvimento, ferramentas como Docker e Docker Compose provaram ser muito úteis quando o assunto é realizar *deploy* de uma aplicação de forma rápida, diminuindo consideravelmente o tempo para instalação de todo o ambiente de produção, atingindo o objetivo de se ter uma arquitetura que fosse construída de forma ágil.

Visto os resultados de desempenho do servidor, é possível dizer que a arquitetura desenvolvida se enquadra em uma arquitetura de *Big Data*, pois possui a velocidade na obtenção de resultados, sendo capaz de se analisar dados praticamente em tempo real, quanto a veracidade dos dados, a escolha do banco Datomic garantiu isso de forma a não perder registros muito menos causar alterações indesejadas neles, além de manter uma base histórica dos dados armazenados, mesmo quando a base de dados passou a crescer gerando um grande volume.

Através do uso dos micro serviços notou-se também que a integração da arquitetura com qualquer outro dispositivo de IoT pode ser feita de forma fácil, pois nos próprios testes se teve um microsserviço em Clojure se conectando a infraestrutura e uma ferramenta de teste de *stress* escrita em Python, desta forma independente da tecnologia utilizada nas pontas, ela se adaptaria facilmente à arquitetura independente da tecnologia empregada no IoT.

6 CONCLUSÃO

O desenvolvimento do presente sistema possibilitou a criação de uma nova arquitetura para um sistema de *Big Data*, pois como apresentado, o sistema leva características dos 5Vs presente neste tipo de arquitetura, contendo um grande volume de dados, pois o sistema quando em operação gera uma massa de dados gigantesca vinda das milhares de conexões que se tem com todos os carros que estão no trânsito, as requisições são atendidas com uma alta velocidade, pois neste tipo de aplicação se torna inadmissível a demora na obtenção de uma resposta, e mesmo com o grande volume de requisições o sistema consegue manter a veracidade dos dados, tudo isso sem um alto consumo de recursos de *hardware*, já que este pode não estar concentrado em um único ponto, mas sim distribuído entre várias máquinas. Sendo assim, a arquitetura e o sistema proposto realmente atendem os resultados esperados com relação a urgências no trânsito e prova que poderia ser utilizado em outros estudos de caso.

Outro ponto que se esperava com a arquitetura era a fácil utilização da mesma, diferente do que ocorre com as ferramentas tradicionais com relação a instalação e uso propriamente dito, a arquitetura especificada por possuir ferramentas dedicadas a instalação e gerenciamento das instâncias se apresenta mais simples e rápida de ser manipulada e através dos micro serviços que também podem ser chamados de *Application Programming Interface* (API) quando analisado na visão da interface de usuário, a integração com outros sistemas os quais enviem ou consumam dados se torna fácil, mantendo a alta escalabilidade do sistema, tanto em relação a distribuição, como em relação aos dados armazenados.

Dentre as principais dificuldades durante a montagem da arquitetura especificada, está a dificuldade em se trabalhar com Clojure, apesar de todas as características e benefícios já mencionadas anteriormente, por ser uma linguagem recém lançada no mercado e não possuir uma comunidade ativa para a resolução de problemas encontrados com a linguagem, encontrar suporte e documentação para o desenvolvimento se tornou uma tarefa difícil, se fazendo necessário a análise puramente de *logs*, que por muitas vezes não eram específicos da linguagem, mas sim da JVM, o que tornou mais difícil a obtenção de soluções, embora torne o aprendizado mais rico, este tipo de análise demanda maior tempo.

Em uma primeira fase do projeto ao invés de Kafka como fila de mensagens utilizou-se o Redis, apesar de se mostrar eficiente, limitava a questão de escalabilidade e distribuição do sistema, pois ele vem a ser uma fila simples sem replicação das filas como acontece no Kafka, desta forma ele foi substituído pelo Kafka que apesar de mais

robusto também se apresentou mais complexo nas suas configurações, principalmente com relação ao encapsulamento em contêiner.

Como trabalho futuro para a arquitetura apresentada se tem o desenvolvimento da interface de usuário, que vem a ser o dispositivo embarcado presente nos carros, juntamente com a rede pela qual as informações trafegarão entre a infraestrutura e o veículo, para a escolha de ambas é necessário que se faça uma análise para a decisão de qual sistema de IoT deve ser utilizado, podendo se chegar ainda a conclusão de que a melhor forma seria o uso do celular do motorista conectado a rede 3G, pois a arquitetura permite comunicação com qualquer interface de usuário, provando sua interoperabilidade.

Por apresentar uma causa nobre e por escolha dos autores, todo o projeto está aberto em um repositório no GitHub (https://github.com/alexcvcoelho-gabrielgio) sob a licença GPL2, onde é possível encontrar todos os códigos desenvolvidos, configurações necessárias e *links* de referência para as tecnologias, sendo assim, o trabalho fica aberto a possíveis colaboradores ajudarem a definir os próximos passos e direções a serem tomadas com o projeto.

APÊNDICE A – *SESSION COMMAND*

```
(ns session-command.kafka.core
  (:require [kinsky.client :as client]
            [session-command.config :refer [env]]
            [clojure.core.async :as a]
            [jkkramer.verily :as v]))

(def s-validate (v/validations->fn [[:required [:brand :model :hd-id :
   command :uuid]]]))
(def t-validate (v/validations->fn [[:required [:session-id :gas-lvl :lat :
   long :vel]]]))
(def w-validate (v/validations->fn [[:required [:session-id :action]]]))

(mount.core/defstate conn
                     :start (client/producer {:bootstrap.servers (env :
                        kafka)}
                                             (client/keyword-serializer)
                                             (client/edn-serializer))
                     :stop (client/close! conn))

(defn push-session [session]
  (if (s-validate session)
    (client/send! conn "session" :session session)
    {:error "Value can be null"}))

(defn push-warn [warn]
  (if (w-validate warn)
    (client/send! conn "warn-warn" :session warn)
    {:error "Value cannot be null"}))

(defn push-track [track]
  (if (t-validate track)
    (client/send! conn "track" :track track)
    {:error "value cannot be null"}))
```

APÊNDICE B – *SESSION WORKER*

```
(ns session-worker.core
  (:gen-class)
  (:require [ext.kafka :as kf]
            [clojure.core.async :refer [go thread chan <!!]]
            [ext.router :as rt]
            [clojure.data.json :as json]
            [clojure.tools.logging :as log]))

(defn pull [item]
  (println (:command item) " " (:uuid item))
  (case (:command item)
    "create_session" (rt/save-session item)
    "warn" (rt/save-warn item)
    nil))

(defn loop-through [c]
  (loop [count 0]
    (try
      (println "LOOP" count)
      (kf/pop-session-async c)
      (doseq [i (<!! c)]
        (pull i))
      (catch Exception e
        (-> e print)))
    (recur (inc count))))

(defn -main [& args]
  (mount.core/start)
  (let [c (chan)]
    (loop-through c)))
```

APÊNDICE C – *DOCKER COMPOSE*

```
version: '3'
services:
  session-command:
      container_name: "session-command"
      environment:
        - KAFKA=gabrielgio.com.br:9092
      image: registry.gitlab.com/gabrielgio/session-command:0.0.1
      ports:
        - "3000:3000"

  tracker-command:
    container_name: "tracker-command"
    environment:
      - KAFKA=gabrielgio.com.br:9092
    image: registry.gitlab.com/gabrielgio/tracker-command:0.0.1
    ports:
      - "3001:3000"

  session-query:
    container_name: "session-query"
    environment:
      - MONGO=mongodb://remote:remote@gabrielgio.com.br:27017/main
    image: registry.gitlab.com/gabrielgio/session-query:0.0.1
    ports:
      - "3002:3000"

  session-worker-1:
    container_name: "session-worker-1"
    environment:
      - KAFKA=gabrielgio.com.br:9092
      - MONGO=mongodb://remote:remote@gabrielgio.com.br:27017/main
      - DATOMIC=datomic:sql://main?jdbc:mysql://gabrielgio.com.br:3306/
          datomic?user=remote&password=remote
    image: registry.gitlab.com/gabrielgio/session-worker:0.0.1
```

APÊNDICE D – *MOCKUP* DOS VEÍCULOS

```
(ns sim-worker.core
  (:gen-class)
  (:require [clojure.tools.cli :refer [parse-opts]]
            [clojure.core.async :as a :refer [go thread]]
            [clj-http.client :as client]
            [clojure.data.json :as json])
  (:import (java.util UUID)))

(def session-url "http://gabrielgio.com.br:3000/api/session")
(def track-url "http://gabrielgio.com.br:3001/api/track")

(defn uuid
  "Get a new UUID"
  []
  (str (UUID/randomUUID)))

(defn post-session
  "Start a session"
  [body]
  (client/post session-url {:form-params  body
                            :content-type :json}))

(defn call-api
  "Call tracker api to register sim's location"
  [item]
  (client/post track-url {:form-params  (dissoc item :delay)
                          :content-type :json}))

(defn get-randon-info []
  {:lat (rand-int 30)
   :long (rand-int 30)
   :vel (rand-int 100)
   :gas-lvl (rand-int 30)
   :delay 1})

(defn parse-json
  "Reads information from a json file and parse it"
  [n]
  (loop [x 0 ar []]
    (if (< x n)
      (recur (inc x) (conj ar (get-randon-info)))
      ar)))
```

```
(defn get-session
  "Starts a session from API"
  [{:keys [brand model hid]}]
  (let [res (post-session {:brand brand
                           :model model
                           :hd-id hid})]
    (:session-id (json/read-str (:body res) :key-fn keyword))))

(defn run-sim
  "Runs simulation"
  [sims session hid]
  (doseq [sim sims]
    (call-api (assoc sim :session-id session ))))

(defn run
  "Gathers informations, start simulation
  and start consuming api."
  []
  (let [hid (uuid)
        s (get-session (assoc {} :brand "sim 0001" :model "x01" :hid hid))]
    (run-sim (parse-json 100) s hid)))

(defn start-threads
  "Starts all threads"
  [n]
  (loop [v n]
    (if (> v 1)
      (do
        (go (run))
        (recur (dec v))))))

(defn -main
  "Start simulation"
  [& args]
  (let [{:keys [options]} (parse-opts args [["-n" "--n NUMBER" :parse-fn #(
     Integer. %)]])]
    (start-threads (:n options))))
```