Pós-Graduação em Ciência da Computação

Jorge Cavalcanti Barbosa Fonsêca

GiTo: Uma arquitetura baseada em políticas para coordenação de processamento de eventos complexos na Web of Things

Recife
2018

Jorge Cavalcanti Barbosa Fonsêca

GiTo: Uma arquitetura baseada em políticas para coordenação de processamento de eventos complexos na Web of Things

Este trabalho foi apresentado à Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

Área de Concentração: Redes de Computadores e Sistemas Distribuídos

Orientador: Carlos André Guimarães Ferraz
Co-Orientador: Kiev Santos da Gamaa

Recife
2018

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

F676g Fonsêca, Jorge Cavalcanti Barbosa
GiTo: uma arquitetura baseada em políticas para coordenação de processamento de eventos complexos na *Web of Things* / Jorge Cavalcanti Barbosa Fonsêca. – 2018.
117 f.: il., fig., tab.

Orientador: Carlos André Guimarães Ferraz.
Tese (Doutorado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2018.
Inclui referências.

1. Redes de computadores. 2. Sistemas distribuídos. I. Ferraz, Carlos André Guimarães (orientador). II. Título.

004.6 CDD (23. ed.) UFPE - CCEN 2020 - 157

Jorge Cavalcanti Barbosa Fonsêca

Gito: Uma arquitetura baseada em políticas para coordenação de processamento de eventos complexos na Web of Things

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Doutora em Ciência da Computação.

Aprovado em: 14/09/2018.

_____

Orientador: Prof. Dr. Carlos André Guimarães Ferraz

BANCA EXAMINADORA

_____

Prof. Dr. Nelson Souto Rosa
Centro de Informática /UFPE

_____

Prof. Dr. Ricardo Massa Ferreira Lima
Centro de Informática / UFPE

_____

Prof. Dr. Divanilson Rodrigo de Sousa Campelo
Centro de Informática / UFPE

_____

Prof. Dr. Fabio Moreira Costa
Instituto de Informática/ UFG

_____

Profa. Dra. Thais Vasconcelos Batista
Departamento de Informática e Matemática Aplicada/UFRN

Às minhas filhas Júlia e Helena, que me ensinaram a amar de maneira incondicional. Que este trabalho sirva de estímulo em seus estudos.

AGRADECIMENTOS

O doutorado é um caminho longo e árduo. Vários desafios nessa caminhada. Alguns simples e outros não. Porém, todos levam à momentos reflexivos. Assim como todas as conversas, deliberações e discussões com pessoas que marcam a construção de um trabalho como este. Esta tese é fruto de um esforço incomensurável, feito por alguém, que assim como todos os alunos de doutorado, não conhecemos o seu dia a dia. E todos merecem respeito e aplausos por ter alcançado este objetivo.

Agradeço de forma incondicional ao amor divino, que na esfera que vivemos é apresentado como pequenas e simples ações que nos rodeia. Ao amor das filhas Júlia e Helena, que de tamanha intensidade pode ser comparado aquele propagado por Deus. À Juliana, que como esposa e mãe, permite-me viver todo esse amor como Família, entidade a qual considero prioridade maior na vida aqui na Terra.

Agradeço aos meus familiares: meu pai Ivo, minha mãe Ana, meus irmãos Júnior e Viviana, minha avó Hilda e vovô Tota, pelo meu crescimento pessoal e por eu ser a pessoa que sou. Obrigado pelo esforço realizado durante toda a minha vida educacional. Em especial ao meu pai, que nasceu em um sítio no interior de Pernambuco, de origem humilde, mas que conseguiu alcançar os seus objetivos. Essa luta influenciou, ainda influencia, e sempre influenciará em toda a minha vida como exemplo de dedicação diante dos meus ideais. Aos meus sobrinhos, em especial Guiga, meu companheiro desde pequeno e que tive a honra de ajudar na orientação dos seus primeiros passos.

Aos meus orientadores, Carlos Ferraz, que repito aqui o agradecimento do meu mestrado, e afirmo novamente que ele, além de possuir tamanha sabedoria, é dono de um humor ímpar e uma elegância que merece ser replicada. E Kiev, que com conhecimento técnico e objetivo, muito me orientou nas importantes tomadas de decisão.

Por fim, agradeço à todos os amigos e demais membros da família que torceram, rezaram e vibraram com essa conquista.

RESUMO

A proliferação de dispositivos com poder de comunicação criou um novo paradigma tecnológico conhecido como *Internet of Things* (IoT), que através da integração com as aplicações da Internet, fez surgir a *Web of Things* (WoT). Atualmente, soluções da WoT compartilham seus fluxos de dados com a nuvem em busca de ganho de performance no processamento das informações. Essa abordagem pode não ser ideal para soluções que tenham requisitos de tempo real, uma vez que precisam de rápido retorno das informações. Considerando que um dispositivo da WoT pode possuir conectividade, memória e processador suficientes para colaborar na análise dos dados por ele produzidos, então ele pode decidir qual o melhor local para processá-los, reduzindo assim o tempo de resposta das notificações geradas pelo processamento. Baseado nesta premissa, esta tese apresenta a *GiTo*, uma arquitetura baseada em políticas para coordenação de Processamento de Eventos Complexos (CEP) na WoT, integrando as camadas da *Mist*, *Fog* e *Cloud*, através de protocolos já difundidos na Web. CEP é uma das principais técnicas de análise de fluxo de dados, e vem se destacando pela sua capacidade de detectar padrões complexos a partir de eventos simples. Neste trabalho, foi realizada uma experimentação envolvendo uma análise de performance comparando as execuções nas três camadas da arquitetura, com intuito de exercitar o CEP em situações com diferentes volume de dados. Com isso, limites do processamento foram identificados e a arquitetura foi refinada. Os resultados mostram que é possível realizar CEP localmente, porém, é preciso uma dinâmica de execução onde os dispositivos tenham a autonomia para tomar decisões em tempo real se devem ou não realizar o *offloading* para servidores remotos. Com o uso da *GiTo* foi possível estabelecer políticas no processamento, que foram respeitadas durante toda a análise dos dados. Esse dinamismo foi observado quando a arquitetura foi executada com os componentes de *offloading* ativos.

Palavras-chave: Processamento de Eventos Complexos. Internet das Coisas. Mist Computing. Fog Computing. Web das Coisas. Arquitetura baseada em Políticas.

ABSTRACT

The dissemination of devices able to communicate rise a new technological paradigm called the Internet of Things (IoT). IoT integrated with the Web created another paradigm called Web of Things (WoT). Electronic devices are in our daily life, they captures our information (such as route, time, etc.) and they publish them as a data stream in remote servers to support decision-making. However, in real-time scenarios, this approach may not be appropriate since they need complete data processing in a low response time. In the context of WoT, many devices have connectivity, memory, and processing power enough to analyze (partially or totally) their data. However, in some cases, such devices are also able to decide the best place to process this data, thereby reducing notification response time. Based on this premise, this thesis presents *GiTo*, a policy-based architecture to coordinate Complex Event Processing (CEP) in WoT. It integrates Mist, Fog, and Cloud layers through protocols already disseminated on Web. Among data flow analysis techniques, CEP has gained prominence in detecting complex patterns from simple events. An experiment was carried out with a performance analysis and execution with *GiTo* regarding the three layers (Mist, Fog, and Cloud). The results show that it is possible to perform CEP locally, however, it requires a flexibility to perform some executions. So, devices may have some autonomy to take real-time decisions whether or not offload their data to remote servers. Through GiTo architecture, it was possible to define policies to CEP coordination, which were guaranteed thought data analysis.

Keywords: Complex Event Processing. Internet of Things. Web of Things. Edge Computing. Fog Computing. Policy-based Architecture.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
AWS	*Amazon Web Services*
CEP	*Complex Event Processing*
CoAP	*Constrained Application Protocol*
CPU	Unidade Central de Processamento
DCEP	*Distributed CEP*
DMTF	*Distributed Management Task Force*
DSMS	*Data Stream Management System*
DSP	*Data Stream Processing*
DSPS	*Distributed Stream Processing System*
ECA	*Event-Condition-Action*
EPA	*Event Processing Agent*
HTTP	*Hypertext Transfer Protocol*
ICMP	*Internet Control Message Protocol*
IETF	*Internet Engineering Task Force*
IFP	*Information Flow Processing*
IFPS	*Information Flow Processing System*
IMD	Instituto Metrópole Digital
IoT	*Internet of Things*
LAN	*Local Area Network*
MANET	*Mobile Adhoc Network*
MAPE-K	*Monitor-Analyze-Plan-Execute over a Knowledge base*
PDP	*Policy Decision Point*
PEP	*Policy Enforcement Point*
PMT	*Policy Management Tool*

PoC	Prova de Conceito
PR	*Policy Repository*
QoS	*Quality of Service*
REST	*Representational State Transfer*
RNP	Rede Nacional de Pesquisa
RTT	*Round Trip Time*
SGBD	Sistema de Gerenciamento de Banco de Dados
SQL	*Structured Query Language*
TIC	Tecnologia da Informação e Comunicação
UDP	*User Datagram Protocol*
XML	*Extensible Markup Language*
WoT	*Web of Things*
WWW	*World Wide Web*

SUMÁRIO

1 INTRODUÇÃO

Na década de 90, o pesquisador *Mark Weiser* descreveu um cenário futuro no qual elementos especializados de *hardware* e *software*, conectados por cabos, ondas de rádio e infravermelho seriam tão ubíquos que ninguém notaria suas presenças. Após quase três décadas, ganha força o conceito de Internet das Coisas (IoT), trazendo consigo parte deste pensamento visionário e incentivando a inovação em diversos aspectos, especialmente na transformação de objetos em entidades eletrônicas comunicantes e inteligentes (WEISER, 1999).

A primeira grande mudança provocada pela *IoT* é que não será mais adotada a "visão individual" dos *endpoints* atuais, como *smartphone* e *tablet*. Os sistemas serão cada vez mais onipresentes e capazes de tomar decisões baseadas em características do ambiente no qual estão inseridos, sendo necessário observar as entidades numa "visão sistêmica", onde existirão, por exemplo, sensores e atuadores se comunicando entre si e também com a Internet em busca de um objetivo em comum (BONOMI et al., 2014).

Em 2014, a Cisco fez um projeção estimando ter cerca de 50 milhões de dispositivos conectados em 2020 (CISCO, 2014). Porém, com o crescimento recente, novas previsões para o mesmo ano já colocam esse número na casa de 200 bilhões, projetando assim uma proporção de 26 objetos inteligentes por ser humano na Terra (LUETH, 2018) (INTEL, 2018). Esses produzirão um grande volume de dados, criando o *The Big Data Bang*, em uma analogia com a teoria do *Big Bang* (MCNEELY; HAHM, 2014).

A IoT, apesar de viabilizar novos cenários e oportunidades, traz também dificuldades, como heterogeneidade de dispositivos, escalabilidade, custo de processamento, segurança, gerenciamento dos dados, entre outros (PIRES et al., 2015). Em particular, a interoperabilidade entre dispositivos e a análise dos dados produzidos se apresentam como desafios tanto para academia como para a indústria (MENDES et al., 2018) (PIRES et al., 2015). Fabricantes criam suas próprias plataformas dotadas de protocolos proprietários, dificultando assim a comunicação entre produtos de marcas diferentes (BRöRING et al., 2017).

Atualmente, vários dispositivos já utilizam a infraestrutura da Internet compartilhando os dados coletados com servidores remotos (*cloud computing*) em busca de maior poder de processamento. Essa técnica de utilizar a capacidade computacional de outros dispositivos é conhecida como *offloading* computacional (JUNG et al., 2017) (STOJANOVIC et al., 2014). Todavia, apesar desta abordagem melhorar o poder de processamento para sistemas de análise de dados, pode ser que os requisitos de tempo real das aplicações não sejam alcançados de forma satisfatória, uma vez que os dados são disponibilizados através da rede, processados e enviados de volta para o dispositivo (MUNIR; KANSAKAR; KHAN, 2017).

De fato, uma onda emergente de aplicação da IoT requer suporte à mobilidade, ciência de contexto e baixa latência, necessitando assim a presença de servidores na vizinhança

como solução para seus requisitos de tempo-real, caracterizando a *Fog Computing*. Essa estratégia descentraliza os recursos da nuvem, deixando-os mais próximos dos dispositivos da IoT e reduz latência de comunicação entre eles (BONOMI et al., 2012).

Os dispositivos da IoT, ao compartilhar os dados entre si ou com servidores remotos, criam fluxos contínuos de informação, que são processados sem interrupção. A necessidade de processar estes fluxos tem sido vista como um grande desafio na IoT, principalmente pela falta de um padrão de comunicação (MORENO et al., 2018) (PUSCHMANN; BARNAGHI; TAFAZOLLI, 2017). Neste contexto, empresas como a Intel, Google, Cisco, entre outras, e pesquisadores da área discutem em relação à possibilidade de troca de informações e comunicação entre os diferentes dispositivos IoT (INTEL, 2018).

Neste sentido, surge então a *Web Of Things* (WoT), propondo que padrões já utilizados na Web, como *Hypertext Transfer Protocol* (HTTP), *Representational State Transfer* (REST), sejam reusados, adaptados e integrados à Internet das Coisas, facilitando assim a troca de informações entre dispositivos e servidores remotos (GUINARD; TRIFA; WILDE, 2010) (NEUMANN et al., 2016). A WoT é considerada como um refinamento da IoT através da integração de coisas inteligentes não apenas à camada da Internet (infraestrutura de rede), mas também à Web (camada de aplicação) (GUINARD; TRIFA; WILDE, 2010).

Utilizando a WoT alinhada a técnicas específicas para análise de fluxos de dados, é possível processar os dados provenientes dos dispositivos. Entre as técnicas de processamento de fluxo de informação, o Processamento de Eventos Complexos (CEP) é um dos modelos mais consolidados e completos (LUCKHAM, 2008) (CUGOLA; MARGARA, 2012). Os sistemas CEP visualizam os dados como notificações de eventos provenientes de múltiplas fontes e usam regras para descrever padrões de eventos simples (*low-level events*), para, quando detectados, produzirem eventos complexos (*high-level events*). Imaginando, por exemplo, um evento de temperatura alta, que isolado pode não ser relevante, porém, quando associado ao evento de fumaça, pode produzir um evento de alerta de incêndio.

Com a evolução dos dispositivos da IoT, que além de conectividade, podem possuir recursos computacionais que permitam suportar protocolos utilizados na Web, é possível efetuar CEP localmente. Essa abordagem de trazer o processamento da nuvem de volta para o próprio dispositivo enfatiza o uso do novo paradigma chamado de *Mist computing*. Esse é considerado uma evolução da *Fog*, e distribui a computação dos dados entre as entidades da IoT, descentralizando ainda mais a tomada de decisão das ações (YOGI; CHANDRASEKHAR; KUMAR, 2017) (PREDEN et al., 2015). A *mist* é também conhecida como *edge computing* (CHOOCHOTKAEW et al., 2017) (DAUTOV; DISTEFANO, 2017). Porém, a terminologia *edge*, na literatura, é muitas vezes utilizada para representar o conceito de *Fog* (OTTENWäLDER et al., 2014) (YI; LI; LI, 2015b). Assim, este trabalho utilizará o termo de *mist* para representar os dispositivos da IoT que processam as informações localmente ou com seus pares, sem precisar realizar o *offloading* computacional para um servidor externo.

1.1 Motivation

A *IoT* propõe que sistemas computacionais sejam formados por diversos agentes produtores e/ou consumidores de informações, permitindo o desenvolvimento de diversas soluções baseadas na aquisição e análise de dados. Porém, tal demanda exige uma dinâmica de execução que garanta tomar decisões e agir dentro de um tempo planejado (FRANCIS; MADHIAJAGAN, 2017) (SHIH, 2018)(BENNACEUR et al., 2014).

Com intuito de entender melhor essa dinâmica da IoT no contexto de processamento de dados, é apresentado um cenário real como motivação.

1.1.1 Cenário Motivador

O cenário motivador é baseado no sistema de monitoramento de trânsito (SMT) descrito por (BONOMI et al., 2014) e (OSANAIYE et al., 2017). Nesse sistema, cuja funcionalidade principal é a prevenção de acidentes, os veículos compartilham com servidores remotos informações como localização, velocidade, posições do volante e dos pedais de freios e aceleração, criando um fluxo contínuo de dados. Os servidores processam os dados analisando o risco de acidentes. Caso essa possibilidade exista, os veículos são notificados através de um alerta informando a situação. Porém, os veículos nem sempre terão conexão com os semáforos continuamente, além de que, acidentes podem acontecer em frações de segundo. Contextualizando o SMT para a proposta deste trabalho, o processamento dos dados dos veículos pode ser feito de três maneiras, como ilustrado na Figura 1.

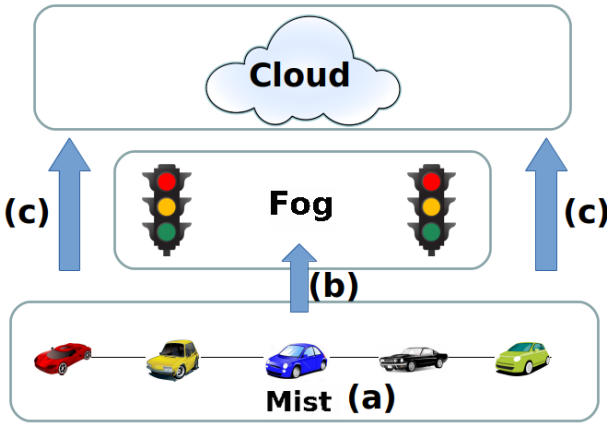


Figura 1 – Processamento dos dados (a) na *Mist*, (b) na *Fog* e (c) na *Cloud*

No cenário (a), o processamento dos dados é realizado nos próprios veículos através do compartilhamento de informações entre eles. Em (b), os veículos realizam o *offloading* computacional para a Fog, onde os dados são analisados, e em (c), esse procedimento alcança a nuvem. Porém, as três abordagens podem ser adotadas em conjunto, com a análise dos dados sendo realizada em todas as camadas ao mesmo tempo, reduzindo os problemas citados. Um dispositivo pode ou não realizar um processamento, dependendo

da sua capacidade computacional e do volume de dados que recebe, logo, é preciso ter mais de um fluxo operacional disponível. Por exemplo, o SMT poderia ter a análise dos dados iniciada no veículo. Caso o consumo dos recursos do computador embarcado nele seja elevado, o *offloading* para a névoa (*Fog*) ou para nuvem (*Cloud*) pode ser realizado continuando o processamento dos dados e minimizando a carga no automóvel.

1.2 Problem Statement

A sociedade passa por mais uma revolução digital, que apesar de provocar uma sobrecarga de informação, vem oferecendo possibilidades de melhoria na qualidade de vida dos indivíduos. Cenários como os descritos na seção anterior mostram como a Internet das Coisas e a tecnologia por trás deste conceito vieram para auxiliar e facilitar o dia a dia das pessoas (MIRANDA et al., 2015). De fato, é preciso estar cada vez mais próximo das pessoas e dos dispositivos para prover uma resposta contextualizada e com uma latência menor (ARKIAN; DIYANAT; POURKHALILI, 2017). Para que isso seja viável, é necessário que os dispositivos e/ou entidades localizadas nas extremidades das redes possam se comunicar e ter poder de decisão, possibilitando responder de maneira rápida e eficaz, principalmente em situações críticas (MUNIR; KANSAKAR; KHAN, 2017). Porém, faltam regras claras e diretas que orientem e garantam a tomada de decisão adequada em relação a quando e onde processar as informações, assim como definir qual a infraestrutura apropriada que viabilize o desenvolvimento dessa nova categoria de aplicações.

1.2.1 Pergunta de Pesquisa

Com base nos conceitos apresentados e no cenário mencionado, surge uma questão (*PP*):

Como é possível fazer processamento de eventos complexos ***nas coisas*** *da Internet das Coisas?*

Existe uma lacuna que ainda é pouco explorada. Os atuais sistemas de processamento de dados em tempo real possuem deficiência no suporte para consumidores e sensores dinâmicos, onde a ciência de contexto e o tempo de resposta são críticos e essenciais para qualidade do serviço (NALLAPERUMA et al., 2017).

Essa pergunta leva a uma secundária, que também se relaciona ao problema:

- *PP2*: Quais fatores influenciam a tomada de decisão de um dispositivo participar ou não do processamento das informações?

1.3 Hipótese

Considerando que a Internet das Coisas é uma tecnologia que vai produzir dados que precisam ser consumidos em um tempo cada vez mais curto, que a WoT reduz o problema de interoperabilidade entre os dispositivos e que nem sempre uma conexão com um servidor remoto para envio e recebimento de eventos vai existir, a hipótese que conduz este trabalho é: *Se os dispositivos da WoT puderem colaborar na análise dos dados por eles produzidos e decidir qual o melhor local para o processamento dos eventos, então as notificações pode ser geradas com um tempo de resposta satisfatório para a maioria das aplicações.*

1.4 Objetivos

Com intuito de resolver o problema mencionado, a ideia principal da tese é investigar se, quando um sistema CEP leva em consideração a integração da *Mist*, da *Fog* e da *Cloud* através de dispositivos com diferentes poderes computacionais, assim como as características do ambiente ao redor deles, o processamento dos dados pode ser executado com um tempo de resposta menor, além de fornecer uma informação com melhor qualidade para as aplicações. Porém, nem todos os dispositivos conectados a uma determinada rede estarão aptos a colaborar em um mesmo instante de tempo, pois cada um possui informações contextuais específicas, como capacidade de processamento, latência de rede, nível de bateria, entre outros. Surge então a necessidade de termos um conjunto de políticas para coordenar esse tipo de sistema.

Baseado na necessidade de políticas para CEP, este trabalho se propõe a investigar, definir e avaliar uma estrutura baseada em regras, que, quando distribuída, otimize o processamento de eventos a partir do recebimento de dados. Assim, o principal objetivo da pesquisa é definir **uma arquitetura orientada a políticas para coordenação de processamento de eventos complexos na Web das Coisas**. A arquitetura resultando deste trabalho é chamada de *GiTo* e representa um codinome para *Global IoT*.

1.4.1 Objetivos Específicos

Os objetivos específicos desse trabalho são:

1. Propor uma arquitetura baseada em regras que, uma vez definidas, possa coordenar o processamento de dados em tempo real através de dispositivos da IoT;

2. Implementar uma Prova de Conceito (PoC) de acordo com a definição da arquitetura;

3. Desenvolver uma aplicação cenário utilizando a PoC como base para realizar uma análise de desempenho e provar a hipótese do trabalho.

1.5 Metodologia

Este trabalho utiliza uma metodologia de pesquisa científica baseada numa abordagem metodológica Hipotético-Dedutivo (NEWMAN; BENZ, 1998). Nesse método, a investigação científica visa construir e testar uma possível resposta ou solução para um problema (ANHEMBI, 2015). A Figura 2 mostra a metodologia adotada.

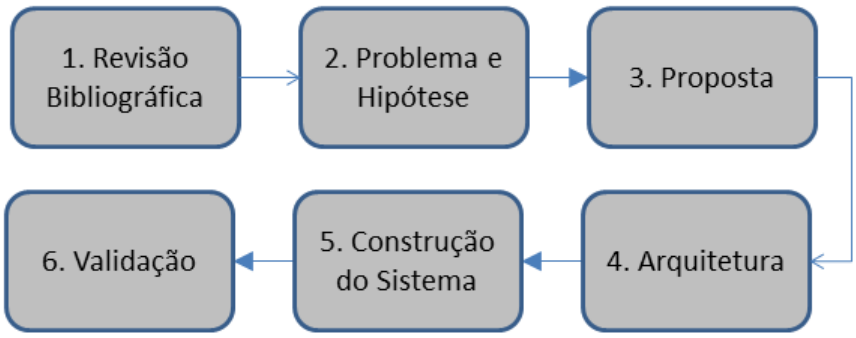


Figura 2 – Metodologia

Uma vez realizada a revisão da literatura, definiu-se o problema, objetivos com base em na hipótese apresentada. Depois, a arquitetura proposta foi elaborada e implementada. Por fim, um experimento de análise de performance foi realizado, assim como a integração total da arquitetura considerando as 3 camadas (*mist*, *fog* e *cloud*).

1.6 Contribuições

As principais contribuições desta pesquisa são:

1. Uma arquitetura baseada em políticas para coordenação de CEP na IoT, envolvendo a *Mist* com apoio da *Fog* e *Cloud*;

2. Mostrar a viabilidade de utilizar CEP nas coisas da Internet das Coisas sugerindo que tipo de processamento deve ou não ser executado neles.

1.7 Escopo Negativo

Essa seção descreve o que este trabalho não se propõe a pesquisar, analisar ou resolver. A lista seguinte enumera alguns itens relacionados à temática abordada e que não fazem parte do escopo investigado, uma vez que a pesquisa prioriza às questões relacionadas à camada de plataforma/*software*.

1. Infraestrutura de *hardware*, tanto dos sensores/atuadores como de servidores, assim como questões relacionadas à virtualização destes dispositivos na *Fog* e na *Cloud*;

2. Questões relacionadas a protocolos de comunicação e interoperabilidade com e entre os dispositivos, assim como itens relacionados à segurança e privacidade em dispositivos de Internet das Coisas;

3. Realização de medições de métricas relacionadas ao consumo de bateria. Apesar de que, do ponto de vista de *software*, toda e qualquer implementação será desenvolvida visando o menor consumo de bateria possível.

1.8 Organização do Documento

As demais partes deste documento estão organizadas da seguinte maneira:

- Capítulo 2 apresenta os conceitos fundamentais utilizados e as principais abordagens relacionadas a este trabalho, assim como os trabalhos relacionados;
- Capítulo 3 detalha a solução proposta e apresenta a arquitetura *GiTo* e seus componentes;
- Capítulo 4 mostra como foi construída a PoC;
- Capítulo 5 apresenta a metodologia e o planejamento da experimentação da arquitetura;
- Capítulo 6 detalha os resultados obtidos e faz uma discussão sobre a solução final;
- Capítulo 7 conclui o trabalho e aponta possíveis trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo tem como objetivo apresentar os principais conceitos que fundamentaram a proposta da solução, descrevendo-os brevemente.

2.1 Hierarquia de Camadas

A literatura mostra que os dispositivos da IoT são divididos em três camadas, como ilustrado na Figura 3 (BONOMI et al., 2014) (MUNIR; KANSAKAR; KHAN, 2017). A camada 1 (Mist) integra os dispositivos próximos dos usuários finais e produtores de dados, como por exemplo sensores de temperatura, telefones móveis, automóveis, monitores cardíacos. A camada 2 (*Fog)* apresenta servidores/dispositivos entre a camada 1 e a camada 3 (Nuvem). Esta última, contém servidores com alta capacidade de processamento/armazenamento, e sempre envia e recebe dados das outras camadas.

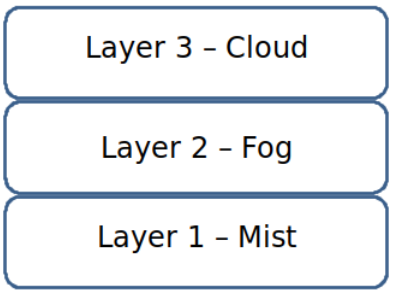


Figura 3 – Hierarquia de camadas

A camada de um determinado dispositivo depende da aplicação onde ele está inserido. Por exemplo, um *smartphone* pode atuar como produtor de dados (camada 1) ou como um servidor próximo à extremidade (camada 2), agregando, processando e filtrando informações.

No restante do texto deste documento, o termo "camada" é usado para representar um nível horizontal da hierarquia composta pela Nuvem, *Fog* e *Mist*, como ilustrado na Figura 3. Já os termos "dispositivo" ou "entidade", representam as "coisas" da IoT.

2.2 Internet of Things (IoT)

A disponibilidade da Internet em todo lugar vem favorecendo a produção de equipamentos dotados de conectividade. Esses, além de acesso à Internet, podem também estabelecer conexões entre si (MIRANDA et al., 2015). Nesse caminho, o atual conceito da Internet como uma infraestrutura de rede que permite que usuários tenham acesso a todo tipo de

conteúdo, dá espaço à visão de entidades inteligentes interconectadas (WEISER, 1999). Essa visão possibilitará a criação de um novo conjunto de aplicações, habilitando novos modos de trabalho, interação e entretenimento, mudando de fato a vida das pessoas (BORGIA, 2014; GUBBI et al., 2013).

O termo Internet das Coisas emerge dessa inovação onde todos os objetos físicos se tornaram também objetos eletrônicos, tornando-os inteligentes e contribuindo de maneira transparente numa visão global. O conceito IoT dá início à implementação das colocações feitas por (WEISER, 1999) quase três décadas atrás.

Com a IoT, os sistemas poderão se adaptar continuamente conforme alterações nas características/atributos dos ambientes onde são executados. As entidades conectadas fornecerão dados relacionados ao ambiente em que estão inseridas, permitindo análises que levarão a tomadas de decisão adequadas. Esse cenário habilita um novo conjunto de aplicações. (ATZORI; IERA; MORABITO, 2010) apresenta, através da Figura 4, um levantamento dos principais domínios e cenários interessados no uso da IoT.

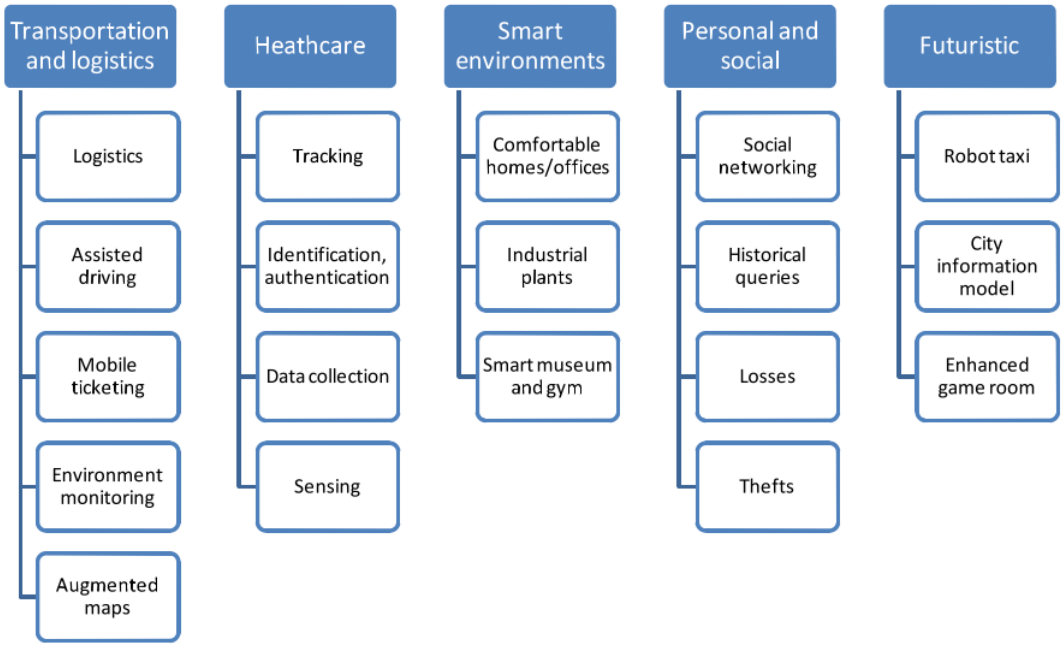


Figura 4 – Internet das Coisas: domínio e cenários de aplicações - Fonte: (ATZORI; IERA; MORABITO, 2010)

Na Figura 4 observa-se o setor de logística como uma categoria promissora para aplicação de IoT. Em 2015, a empresa *DHL* (líder mundial no setor de logística (LOGWEB, 2018)) e a CISCO lançaram em conjunto um Relatório de Tendências mostrando o potencial do setor no contexto de IoT (CASTRO, 2015).

Contudo, a IoT, apesar de viabilizar novos cenários e oportunidades, também traz desafios. Por exemplo, um automóvel conectado na IoT pode ser alvo de um ataque remoto, onde o *hacker* pode assumir o controle do veículo, deixando o motorista passageiro do seu próprio carro. (WIRED, 2015) realizou um teste onde dois hackers invadem o sistema de

um automóvel da marca *Jeep* e assumem todo o controle do veículo. Situações como essas surgem como barreiras a serem resolvidas, e apresentam-se como diferenciais atrativas para todos.

2.2.1 Desafios

A Internet das Coisas está causando uma revolução no setor de Tecnologia da Informação e Comunicação (TIC), movendo a atual Internet, que conecta dispositivo ao usuário, para um mundo onde todos os objetos são conectados uns aos outros e também com humanos (ZORZI et al., 2010) (YAN et al., 2008). Toda essa mudança traz inúmeras possibilidades e vantagens, contudo, apresenta vários desafios (MIORANDI et al., 2012), dentre eles:

1. Heterogeneidade de dispositivos: a IoT é caracterizada por integrar os mais variados dispositivos, com diferentes capacidades de processamento e comunicação;
2. Escalabilidade: objetos conectados em uma infraestrutura global, gerando problemas de escalabilidade em vários níveis;
3. Troca de dados ubíqua: a IoT precisa viabilizar a comunicação dos objetos de maneira transparente;
4. Otimização de energia: com os dispositivos processando mais dados, o consumo de energia se torna ainda mais crítico;
5. Localização e rastreamento: na IoT as entidades são consideradas móveis e com poder de comunicação. Assim, é possível saber suas localizações em qualquer instante de tempo;
6. Auto-gerenciamento: as entidades precisam possuir capacidade autônoma de reagir ao contexto que estão inseridos;
7. Gerenciamento de dados e semântica de interoperabilidade: Com todos os objetos conectados, o volume de dados tende a aumentar consideravelmente. É preciso saber gerenciar esses dados provenientes de fontes e formatos distintos (GUBBI et al., 2013);
8. Segurança e privacidade: com o poder de decisão nas entidades, questões sobre segurança e privacidade se tornam críticas e emergenciais.

Diante dessas dificuldades, (BONOMI et al., 2012) apresentou o conceito de *Fog Computing* como uma solução para alguns desses desafios.

2.2.2 Web of Things

Atualmente, várias soluções de IoT publicam e expõe seus dados e funcionalidade através de sistemas proprietários fortemente acoplados. Essa abordagem aumenta o problema de interoperabilidade causado pela heterogeneidade de dispositivos. Vários sistemas com poder de comunicação trabalham de forma "isolada", não estabelecendo comunicação com outros ao seu redor, nem mesmo com aplicações da Internet (GUINARD; TRIFA; WILDE, 2010). Neste contexto, surge a Web of Things, cuja proposta é reusar, adaptar e integrar padrões já utilizados na Web nos dispositivos embarcados. Por exemplo, embarcando *web servers* nas *things* é possível reduzir o esforço na integração entre dispositivos.

A Figura 5 mostra a diferença entre protocolos utilizados na IoT e WoT. Apesar de consumir mais recursos, a WoT reduz o problema da interoperabilidade (BLACKSTOCK; LEA, 2013).

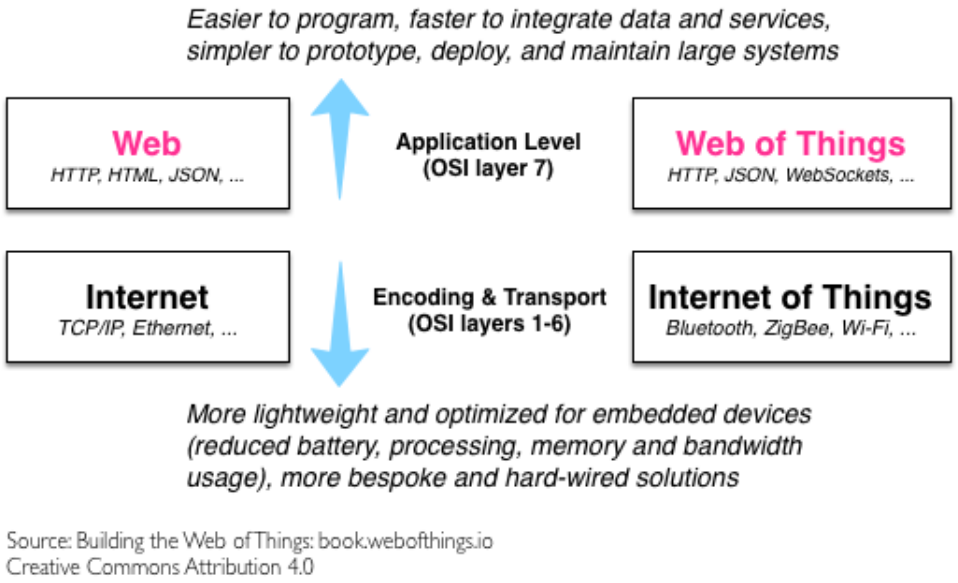


Figura 5 – Protocolos: IoT vs. WoT

2.2.3 *Mist* e *Fog Computing*

A computação em Nuvem atende os requisitos de aplicações executadas no modelo cliente-servidor de maneira satisfatória. Porém, são consideradas ineficientes em sistemas que demandam por requisitos que exijam baixa latência e alto grau de mobilidade. Na IoT, além da comunicação ser considerada uma necessidade, o suporte à mobilidade é essencial (HAJIBABA; GORGIN, 2014). Visando minimizar essa lacuna, (BONOMI et al., 2012) apresentou o conceito de *Fog Computing* como sendo uma plataforma virtualizada que provê poder computacional, armazenamento e servidores de rede localizados entre os servidores na Nuvem e os dispositivos que ficam na extremidade da rede. A Figura 6 ilustra o papel da computação *Fog*.

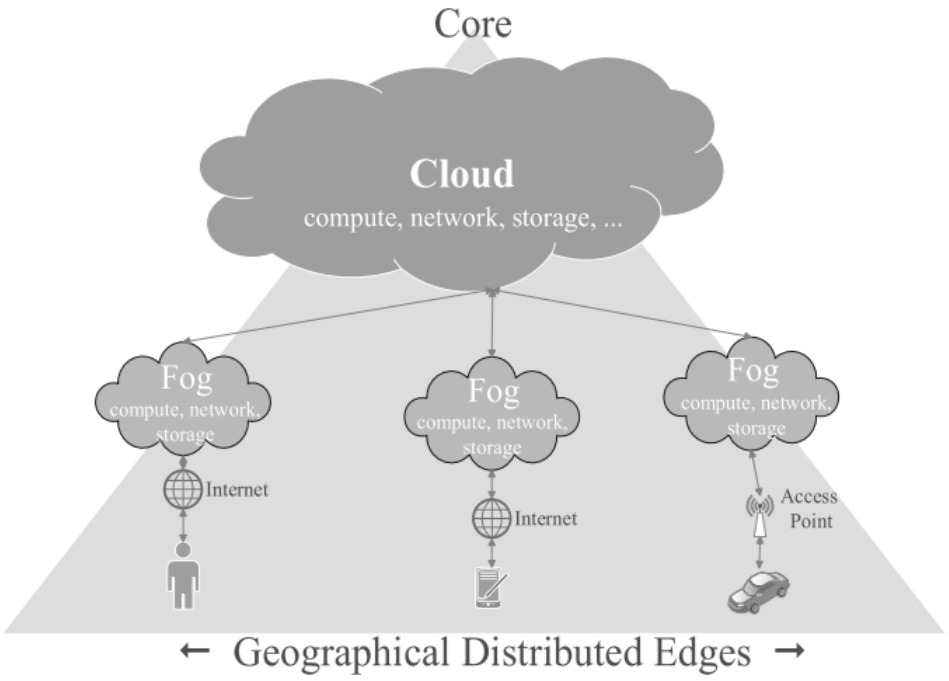


Figura 6 – Arquitetura da computação *Fog* - Fonte: (HAJIBABA; GORGIN, 2014)

A computação *Fog* acontece fora da Nuvem, assegurando que os serviços da Nuvem possam ser estendidos até as extremidades da rede, de maneira parcial ou total. Logo, a ideia principal da computação *Fog* é distribuir os dados e processamento para perto do usuário final, reduzindo a latência, melhorando o *Quality of Service* (QoS) e eliminando possíveis problemas relacionados à conexão e transferência de dados. Devido à sua distribuição geográfica, esse novo paradigma é considerado apto para processar dados em larga escala e analisá-los em tempo real, suportando computação móvel e análise de fluxo de dados (YI; LI; LI, 2015a).

Porém, a *Fog Computing* não atende todos os cenários da IoT (MUNIR; KANSAKAR; KHAN, 2017). Embora o ganho apresentado pela *Fog* seja promissor, os dispositivos localizados na *fog* (ex. servidores *edge*, roteadores inteligentes, estações base, etc.) podem não ser capazes de atender à requisitos de desempenho, taxa de transferência, consumo de energia e tempo de resposta da nova geração de aplicativos da IoT (MUNIR; KANSAKAR; KHAN, 2017). Surge então a *Mist computing* (YOGI; CHANDRASEKHAR; KUMAR, 2017) (PREDEN et al., 2015). A *Mist* tem como proposta levar a computação para a verdadeira borda da rede: os dispositivos (ex. sensores e atuadores).

Na *Mist*, os dispositivos possuem ciência do contexto onde estão inseridos, e possuem acesso às informações necessárias para sua própria adaptação e reconfiguração. As regras e funcionalidades das aplicações não podem ser estáticas, assim como os provedores de conteúdo. Os dispositivos devem dinamicamente descobrir onde os dados se encontram e executar suas aplicações.

2.3 *Information Flow Processing* (IFP)

Atualmente, várias aplicações que processam dados precisam gerar uma resposta em um curto espaço de tempo, algumas vezes em tempo real (ANICIC et al., 2012). Por exemplo, sistemas de monitoramento de trânsito e sistemas de controle de cargas em empresas que trabalham com logística precisam rastrear as informações em tempo de real. Os tradicionais Sistema de Gerenciamento de Banco de Dados (SGBDs) dificilmente atendem à esse tipo de requisito.

Diante de cenários onde requisitos de tempo real são indispensáveis, um novo conjunto de aplicativos distribuídos vem ganhando destaque: os *Information Flow Processing System* (IFPS) (CHAKRAVARTHY; JIANG, 2009). Apesar de ter suas raízes nos SGBDs, os *IFPs* diferem em vários aspectos. Os conceitos de *timeliness* e fluxo de processamento (*streams*) justificam a existência dessa nova categoria de sistemas (CUGOLA; MARGARA, 2012).

A demanda por sistemas IFP vem atraindo o interesse de várias comunidades científicas. Detecção de fraude em transação de cartão de crédito, gerenciamento de inventário, rastreamento de encomendas por *RFID* e aplicações financeiras que traçam tendências de mercado são exemplos de sistemas de áreas distintas (BANDARA et al., 2015). Em comum a esses exemplos, tem-se que o processamento dos fluxos acontece das extremidades para o centro do sistema. A Figura 7 apresenta uma visão genérica de uma arquitetura IFP.

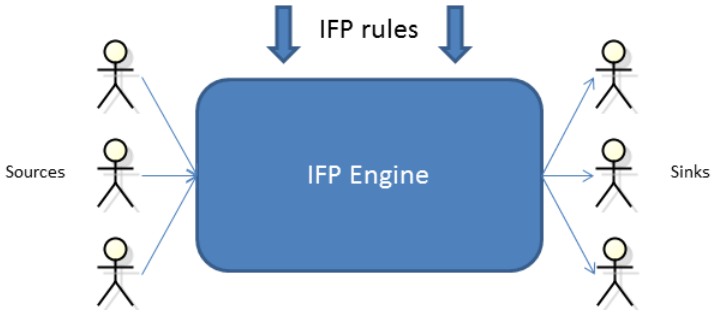


Figura 7 – Arquitetura de um sistema IFP

As entidades que criam os fluxos de informações são chamadas de *sources* ou produtores. O engenho IFP é um conjunto de máquinas que operam de acordo com regras pré-definidas. Essas regras definem o processamento dos fluxos em função de suas semânticas, identificando os tipos de dados e parâmetros que se deseja detectar. O sistema produz como saída novas informações, que uma vez enviadas ao interessados (*sinks* ou consumidores), podem ser descartadas. Um IFP é capaz de processar, filtrar e agregar dados provenientes de diferentes fluxos ao mesmo tempo.

O termo IFP está relacionado com diferentes tecnologias. Para superar as limitações dos SGBDs, os sistemas de banco de dados ativos foram desenvolvidos. Esses, através das regras *Event-Condition-Action* (ECA) (MCCARTHY; DAYAL, 1989), disparam ações

quando certas condições são alcançadas. Porém, os SGBDs ativos também funcionam com dados persistidos, prejudicando o desempenho, principalmente, quando o número de regras extrapola um certo limiar, tornando-os inviáveis para os sistemas IFPs (JOVANOVIC et al., 2015). Para superar essa limitação, principalmente quando é preciso processar eventos externos como fluxo de dados, foram criados os *Data Stream Management Systems* (DSMS). Esses não trabalham com tabelas nem armazenamento, nem assumem uma ordem de chegada dos dados. Os DSMSs processam os dados assim que eles chegam através de consultas contínuas (*continuous queries*). Essas consultas, uma vez configuradas no sistema IFP, produzem como resultados um novo fluxo de dados baseado nos parâmetros informados. As consultas são executadas continuamente até que sejam removidas.

Os DSMSs foram modelados como uma extensão dos SGBDs. Como consequência, não atendem às necessidades dos IFPs por completo. Eles produzem respostas às consultas e as atualizam continuamente a partir de mudanças no fluxo de entrada. Detecção e notificação de padrões complexos, envolvendo relações de ordem e sequência não são cobertos pelos DSMSs. Para superar tal limitação, foram criados os sistemas de processamento de eventos complexos (CEP) (LUCKHAM, 2008).

2.3.1 *Complex Event Processing*

CEP é uma tecnologia para processamento de fluxo de dados baseada em eventos que provê um modelo de processamento assíncrono para detecção em tempo real de determinadas situações de interesse (LUCKHAM, 2008). Um sistema CEP tem como entrada um ou mais fluxos de dados (*streams*) brutos produzidos por fontes distintas, como sensores de uma casa inteligente e carros que trafegam pela rodovia, e são processados de maneira contínua, gerando novos eventos. Estes eventos derivados são enviados como notificações para os consumidores interessados (BAPTISTA et al., 2013).

As situações de interesse são modeladas como regras CEP. Essas regras, quando configuradas em um engenho CEP, são analisadas continuamente durante o processamento dos *streams*. Atualmente, vários engenhos dão suporte à linguagens de escrita de regras baseadas em *Structured Query Language* (SQL) (ESPER, 2016), (CQL, 2016). As regras utilizam operadores aplicados aos eventos recebidos, buscando correlação entre eles, gerando eventos complexos a partir da combinação dos eventos simples. Eventos simples correspondem a eventos que não são compostos por outros eventos (CUGOLA et al., 2015).

As regras CEP não só definem a semântica associada ao engenho CEP, como também determinam o papel dos agentes de processamento de eventos *Event Processing Agent* (EPA). Esses agentes representam os nós da rede que colaboram na análise dos dados. Dependendo da localização e distribuição das tarefas, um EPA pode, além de processar, realizar filtros, agregação, divisão ou composição de dados (ETZION; NIBLETT, 2010).

Como exemplo de uma regra CEP, a Listagem 2.1 apresenta uma consulta realizada para calcular a média de temperatura de 10 sensores, agrupadas por id e prioridade (JUN;

CHI, 2014).

```
1        select avg(temperature) as aTemperature, id, priority
2        from Sensor.win:length_batch(10) group by id, priority
```

Listing 2.1 – Exemplo Consulta CEP - média de temperaturas

Pelo exemplo da listagem 2.1, é possível notar a semelhança com a sintaxe utilizada nas consultas SQL.

2.3.2 CEP como uma extensão do modelo *publish-subscribe*

O modelo de processamento de eventos complexos (CEP) tem sua base no modelo *publish-subscribe*. CEP estende essa abordagem incrementando linguagem de consulta para considerar padrões de eventos complexos que envolvem a ocorrência de múltiplos eventos e as relações de ordem e sequência entre eles (EUGSTER et al., 2003).

Sendo o modelo *publish-subscribe* baseado na troca de mensagens assíncronas, os sistemas CEP podem ser executados de maneira distribuída, formando assim um CEP distribuído (*Distributed CEP* (DCEP). Considerando as camadas apresentadas na Figura 3, um DCEP pode ser executado de maneira horizontal, em apenas uma camada, ou de maneira vertical, ortogonal a todas as três camadas.

2.3.3 Arquitetura CEP

A Figura 8 apresenta uma arquitetura abstrata de um IFPS, especificamente de um IFPS que utilizada a técnica de CEP (CUGOLA; MARGARA, 2012).

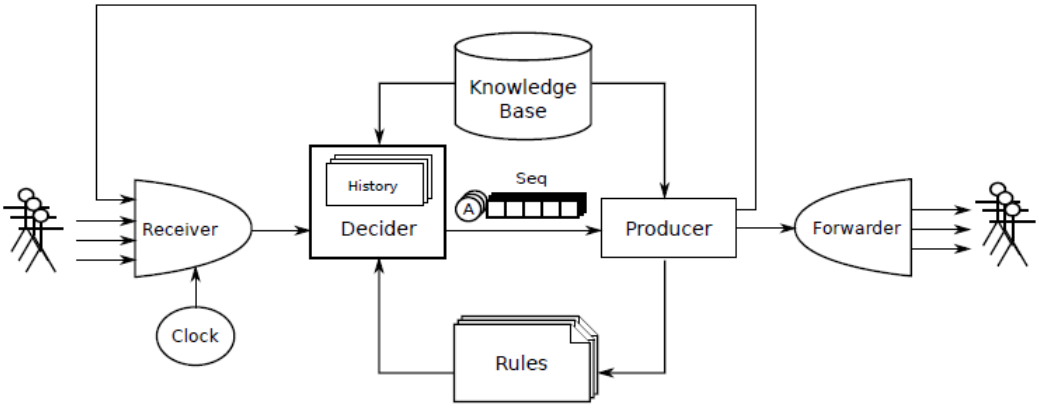


Figura 8 – Arquitetura de um sistema CEP - Fonte: (CUGOLA; MARGARA, 2012)

Os fluxos de dados alcançam o sistema através do *Receiver*. Esse componente é responsável por gerenciar os fluxos de entrada, repassando os dados para o *Decider* como um demultiplexador. Esse último é responsável por filtrar os dados, comparando-os com

as condições estabelecidas nas consultas. Quando detectado, o evento é repassado para o *Producer* que executa uma ação, por exemplo, criar um evento composto a partir dos eventos simples detectados. Por fim, o novo evento é repassado para o *Forwarder* que o encaminha para os consumidores. Eventos compostos produzidos pelo sistema CEP podem retornar para o próprio engenho como um evento simples, criando assim um novo fluxo de dados interno do sistema.

Entende-se por evento simples, um dado que é enviado ao engenho CEP para ser processado, seja proveniente de um sensor, ou mesmo através de um processo de retroalimentaçao do engenho. Evento composto é um evento gerado pelo CEP a partir de eventos simples.

2.4 Contexto Computacional

Atualmente, os sistemas computacionais atuam com base em especificações pré-configuradas, sem levar em consideração características do ambiente de execução, dos indivíduos presentes nesse ambiente e das peculiaridades dos atores que interagem com o sistema. Por exemplo, se um fazendeiro e um estilista realizam uma consulta em um engenho de busca procurando pela palavra "manga", provavelmente ambos receberão a mesma resposta. Se esse engenho de busca conhecesse as necessidades individuais de cada ator do sistema, poderia gerar respostas priorizando detalhes de manga, a fruta, na consulta realizada pelo fazendeiro (VIEIRA; TEDESCO; SALGADO, 2009). Segundo (GARTNER, 2015), contexto estará presente nas mais diversas áreas de TIC, sendo cada vez mais necessário a sua existência nos sistemas.

Contexto é definido de diversas maneiras por diferentes pesquisadores (DEY, 2001) (ABOWD et al., 1999) (BREZILLON, 1999) (GROSS; PRINZ, 2003) (VIEIRA; TEDESCO; SALGADO, 2009). Neste trabalho, optou-se por seguir a definição de (DEY, 2001):

"*Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves.*"

Da qual, pode-se extrair a informação de elemento contextual, que segundo (VIEIRA; TEDESCO; SALGADO, 2009):

"*Um* ***elemento contextual*** *é qualquer dado, informação ou conhecimento que pode ser utilizado para definir um contexto. O elemento contextual pode ser identificado quanto à sua periodicidade de atualização e classificado como estático ou dinâmico.*"

Um elemento contextual pode ser classificado quanto à sua aquisição, como primário ou secundário. Um elemento primário é qualquer informação obtida sem a realização de operações de fusão de dados. Ou seja, é o dado bruto recebido do produtor. Já o elemento

secundário é aquela informação que pode ser calculada a partir de elementos de contexto primários.

Contexto é essencial na construção de sistemas auto-gerenciáveis. Esses precisam reagir de acordo com a variação do contexto dos elementos gerenciados, seja um elemento interno ou externo ao sistema.

2.4.1 Sistema Sensível ao Contexto

Com base nas definições de elemento contextual e contexto, (VIEIRA; TEDESCO; SALGADO, 2009) define os sistemas sensíveis ao contexto:

"***Sistemas Sensíveis ao Contexto*** *são aqueles que gerenciam elementos contextuais relacionados à uma aplicação em um domínio e usam esses elementos para apoiar um agente na execução de alguma tarefa. Esse apoio pode ser alcançado pelo aumento da percepção do agente em relação à tarefa sendo executada ou pelo provimento de adaptações que facilitem a execução da tarefa.* "

Para aplicar contexto a um sistema, é preciso executar etapas para capturar e processar os itens de contexto, tomar decisão com base nessa análise para depois executar uma ação. (ABOWD et al., 1999) definem um processo para facilitar a compreensão das dificuldades inerentes à construção de sistemas sensíveis ao contexto. O processo é formado por 5 etapas, como ilustrado na Figura 9.

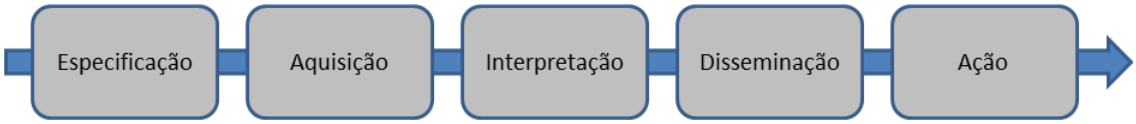


Figura 9 – Etapas do processo de Contexto

1. Especificação: visa especificar o problema a ser resolvido e propor uma solução de alto nível;

2. Aquisição: determinar quais sensores são necessários para prover o contexto. Esses sensores podem ser dispositivos eletrônicos ou informações textuais encontradas nas redes sociais;

3. Interpretação: análise das informações recebidas na aquisição;

4. Disseminação: prover métodos para apoiar a disseminação do contexto a uma ou mais aplicações, possivelmente remotas;

5. Ação: envolve verificar a utilidade da informação adquirida e executar o comportamento ciente do contexto.

Essas etapas auxiliam no entendimento do contexto, moldando o sistema para reagir à mudanças no ambiente de maneira apropriada.

2.5 Sistemas Auto-gerenciáveis

Aplicações distribuídas em larga escala requerem diferentes formas de adaptação. Uma das formas mais simples de adaptação é a configuração de parâmetros operacionais aplicados no sistema antes do início de sua execução. Por outro lado, sistemas mais sofisticados apresentam uma configuração dinâmica e flexível, uma vez que permitem alterar valores dos parâmetros em tempo de execução. O algoritmo de eleição de líder em grupos de máquinas pertencentes a uma mesma rede é um exemplo de algoritmo para coordenação em sistemas distribuídos. Esse algoritmo permite que um processo ou máquina seja determinado para executar tarefas específicas dentro do grupo, como por exemplo, compartilhar dados com um servidor remoto. Caso o líder seja desconectado da rede, é possível eleger um novo líder a partir da execução do algoritmo (HUNT et al., 2010).

Os sistemas distribuídos atuais exigem cada vez mais adaptação a partir de mudanças do ambiente e dos usuários, resultando em um aumento de processamento na administração desses sistemas. Como consequência desse comportamento, os sistemas precisam de módulos capazes de se auto-adaptar a partir dessas mudanças. Uma abordagem comum para garantir o auto-gerenciamento é adicionar componentes específicos cuja função é monitorar o comportamento dos sistemas, e reagir em tempo de execução caso um estado inválido seja alcançado (GARLAN; SCHMERL; STEENKISTE, 2004).

A funcionalidade de auto-gerenciamento implica em uma série de aspectos envolvendo diferentes dimensões, e pode ser dividida em quatro habilidades: auto-configuração, auto-cura, auto-proteção e auto-otimização. Sistemas considerados auto-gerenciáveis são chamados de sistemas autonômicos, e são regidos por instruções de alto-nível (AGRAWAL SERAPHIN CALO, 2008). Uma das maneiras de se alcançar esse nível de gerenciamento é através da aplicação do *loop* de controle *Monitor-Analyze-Plan-Execute over a Knowledge base* (MAPE-K) (BRUN et al., 2009) (AGRAWAL SERAPHIN CALO, 2008). A Figura 10 apresenta o *loop* MAPE-K.

O *loop* de controle permite que o sistema se torne auto-consciente em relação à qualidade das suas operações, e pró-ativo em relação a possíveis problemas. Esse nível de auto-gerenciamento é alcançado através do monitoramento e análise da execução do sistema. Quando problemas são encontrados, o sistema define e executa uma estratégia de adaptação, alterando o estado dos elementos gerenciados.

2.5.1 Políticas

Os sistemas baseados em políticas são capazes de reagir à situações indesejáveis, mantendo seu estado sempre consistente e fiel às suas especificações. Ou seja, são sistemas auto-

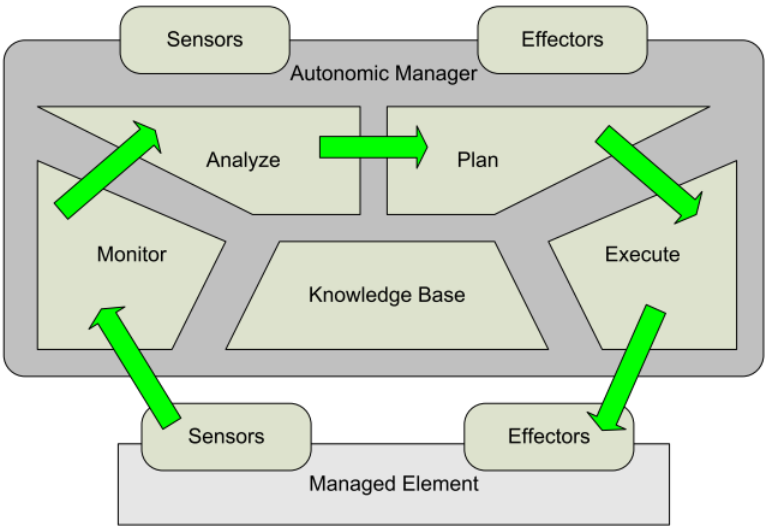


Figura 10 – MAPE-K: *Feedback loop control*

gerenciáveis e por isso são considerados autonômicos (PEDIADITAKIS et al., 2012).

Na área da computação, política descreve regras, regulamentos, objetivos gerais ou planos de ação. Com objetivo de evitar ambiguidades, e também seguir uma definição única, este trabalho seguiu o conceito de políticas introduzido por (AGRAWAL SERAPHIN CALO, 2008).

Primeiramente, é preciso entender quatro conceitos base, para depois ser apresentada a definição de política:

1. O **Alvo/objeto** da política, chamado de sistema alvo, pode ser um simples dispositivo como um *smartphone* ou um *notebook*, ou alguma entidade mais complexa, como um sistema reserva de passagem aérea ou mesmo um servidor de banco de dados localizado na Nuvem;

2. Um **atributo** é uma característica do sistema alvo, podendo ter um determinado valor. Um sistema alvo possui um conjunto de **atributos**;

3. Um **estado** é representado como um conjunto de valores dos atributos do sistema alvo em determinado instante de tempo;

4. O **comportamento** do sistema alvo é definido como um conjunto ordenado e contínuo de estados, onde a ordem é imposta pelo tempo.

Considere um sistema S e $B(S)$ como o conjunto de todos os possíveis comportamentos que o sistema S pode exibir. A definição de política é:

"Uma política é um conjunto de restrições que podem ser aplicadas sobre os possíveis comportamentos $B(S)$ de um sistema alvo S; Ou seja, uma política define um subconjunto de $B(S)$ como comportamentos válidos de S."

Para o sistema ter um uma transição de estados, é preciso ter **ações** implementadas. Essas ações não participam da definição da política em si, porém, se um sistema não executa nenhuma ação, não faz sentido implementar políticas nesse sistema.

2.5.2 Tipos de Políticas

(AGRAWAL SERAPHIN CALO, 2008) também apresenta 4 tipos de políticas, conforme lista abaixo.

1. *Políticas de configuração*: especificam restrições definidas por configuração que devem ser satisfeitas pelo sistema alvo em todo e qualquer estado que ele possa assumir.

2. *Políticas de métricas*: especificam restrições de métricas que precisam ser satisfeitas pelo sistema durante a sua execução, independente do seu estado.

3. *Políticas de ação*: especificam uma ou mais ações que precisam ser executadas quando o estado do sistema alvo alcança determinada restrição.

4. *Políticas de alerta*: é um tipo de política de ação que tem como objetivo enviar notificações sobre o estado do sistema.

2.5.3 Arquitetura de Políticas

O ponto de partida para construir um sistema de gerenciamento baseado em políticas é definir uma arquitetura que atenda às necessidades de auto-gerenciamento. O *Internet Engineering Task Force* (IETF) / *Distributed Management Task Force* (DMTF)[1] possui uma arquitetura composta por quatro componentes, como mostra a Figura 11.

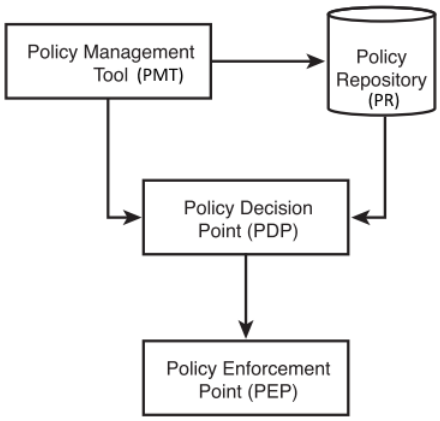


Figura 11 – Arquitetura de Políticas - IETF/DMTF

[1] https://www.dmtf.org/ - Grupo comercial de *software* com intuito de simplificar e gerenciar tecnologias acessíveis por rede.

O componente *Policy Management Tool* (PMT) provê um mecanismo para criação e gerenciamento das políticas. Essas são armazenadas no *Policy Repository* (PR), permitindo consultá-las quando necessário. O *Policy Decision Point* (PDP) é responsável pelo processo de tomada de decisão, examinando as políticas armazenadas no PR, aplicando-as em determinadas circunstâncias e determinando o que for necessário para garantir as políticas. Já o componente *Policy Enforcement Point* (PEP) é responsável por monitorar a execução do sistema, garantindo que as políticas não serão violadas.

O uso de políticas está relacionado à informações do sistema em execução. Sempre que o ambiente ou seus atores mudam seus estados, é preciso verificar se as políticas definidas não foram violadas. Essas características formam o contexto do sistema. Esse conceito é essencial no mundo da IoT, uma vez que a mobilidade dos dispositivos é constante e o volume de dados produzidos é variável.

2.6 Trabalhos Relacionados

A computação em Nuvem é um modelo eficiente no estilo *pay-as-you-go* para realizar as mais diversas operações, deixando seus usuários livres dos detalhes de *hardware* e *software* (BONOMI et al., 2014). Porém, essa liberdade torna-se um problema para a nova geração de aplicações, especificamente para a *Internet of Things*. Esses novos sistemas requerem suporte à mobilidade e geo-localização, ciência de contexto e baixa latência de comunicação (GUAN, 2017) (ARKIAN; DIYANAT; POURKHALILI, 2017).

Diante destes problemas, vários estudos (detalhados em seguida) tem apresentado estratégias para trazer de volta a computação para a borda da rede, e com isso, estar fisicamente mais próximos dos dispositivos *endpoints* e dos usuários.

Esta seção apresenta os trabalhos relacionados à esta tese, buscando entender o panorama das pesquisas efetuadas na área de processamento de fluxo de dados sensíveis a contexto na IoT.

2.6.1 *Data Stream* e Auto-gerenciamento

Com o interesse das mais variadas áreas no processamento de fluxo de dados, trabalhos de diferente linhas de pesquisa tem sido realizados. Em (MADSEN; ZHOU; CAO, 2017), (XU et al., 2014) e (LIU et al., 2017) são apresentadas estratégias de balanceamento de carga de acordo com o volume de dados e o tráfego de rede encontrado. Ferramentas que trabalham com fluxo de dados conhecidas no mercado são estendidas através da inserção de contexto. Porém, o poder computacional (informações de *hardware*) dos dispositivos não é considerado nas análises.

Em (CARDELLINI et al., 2018), foi desenvolvida uma extensão da ferramenta *Apache*

Storm[2], que explora o uso de políticas através da inserção de regras de contexto. O trabalho se destaca por adaptar os componentes de uma das principais ferramentas do mercado, criando uma balanceamento de carga auto-configurável durante sua execução com base no consumo dos recursos da máquina. O *Storm* é uma ferramenta não customizada para dispositivos com restrições de CPU e memória, e por isso não é possível executá-la na *Mist*.

2.6.2 *Fog Computing*

A computação na névoa tem se destacado como um dos principais paradigmas relacionados à IoT, onde a alocação de recursos, assim como o monitoramento e orquestração dos serviços ainda são desafio (WEN et al., 2017). A Figura 12 ilustra a estratégia comum aos trabalhos encontrados (detalhados abaixo), onde o processamento dos dados é realizado na camada 2, com intuito de produzir uma resposta contextualizada e em menor tempo para os dispositivos da *Mist*.

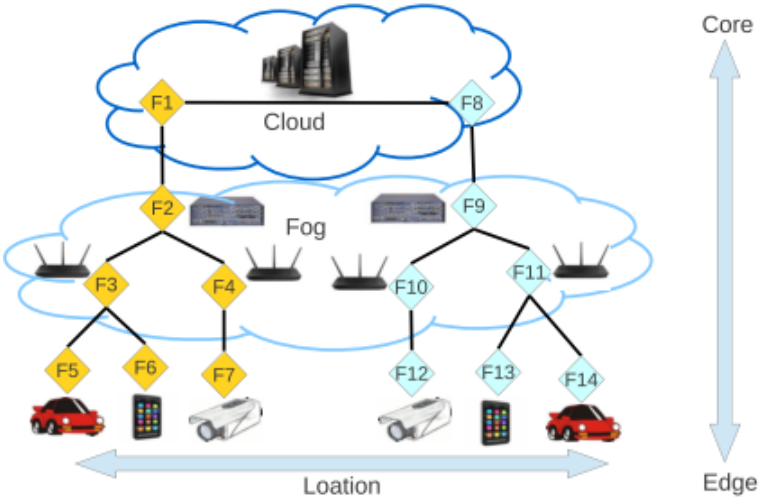


Figura 12 – DSPS Distribuído Móvel - Fonte: (HONG; LILLETHUN, 2013)

(MUNIR; KANSAKAR; KHAN, 2017) apresentam o *IFCIoT*, um *framework* capaz de processar dados na *Fog*. Além do módulo de dados, o *IFCIoT* possui um componente responsável pela reconfiguração da carga de trabalho nos servidores *edge*. O artigo não apresenta detalhes deste módulo, mas menciona o monitoramento dos picos de processamento em busca de uma melhor configuração de *hardware* para atender os requisitos das aplicações. Numa abordagem similar, (WEN et al., 2017) introduzem um modelo para análise de dados na *Fog*, baseado na divisão de tarefas e alocação de máquinas virtuais, com uma boa relação custo-benefício no provisionamento de recursos limitados. O trabalho apresenta uma arquitetura que, apesar de permitir que consumidores coletem dados

[2] http://storm.apache.org/

diretamente dos sensores, todo o gerenciamento e adaptação dos recursos é realizada na *Fog*.

(SKARLAT; BACHMANN; SCHULTE, 2018) enfatizam a distribuição adaptativa do processamento, mas, sempre excluindo os dispositivos da camada 1 da análise dos dados, e realizando *offloading* para a *Fog* processar as informações. Os dispositivos móveis não executam o processamento de fato. O conceito de mobilidade empregado nesses sistemas surge a partir da migração de componentes entre os servidores da *Fog*, que, numa visão global, acompanham a mobilidade das entidades. (OTTENWäLDER et al., 2014) propõe um configurador de consultas CEP (*query reconfigurator*). O sistema monitora as localizações dos dispositivos consumidores e produtores, assim como o desempenho do sistema como um todo. Uma vez alteradas essas informações, o configurador atualiza automaticamente as consultas com base no contexto dos dispositivos. Essa nova configuração inicia o processo de migração de componentes entre servidores *Fog*, fazendo com que componentes próximos ao dispositivo móvel acompanhem seu deslocamento. A execução do processamento CEP acontece na camada 2. Já (LUTHRA; KOLDEHOFE; STEINMETZ, 2018), apresenta um estudo parcial para transição e reconfiguração de CEP distribuído envolvendo também a *Cloud*, como um modelo tolerante à falhas executado em paralelo à orquestração na *Fog*.

Apesar de todos os trabalhos estarem relacionados à IoT, as *things* são visualizadas apenas como fornecedores de dados, e não como agentes de processamento.

2.6.3 *Computação na Borda da Rede*

A *Fog computing* apresenta enorme potencial, contudo, mostra também, limitações. Atualmente, os dispositivos localizados na *Fog* são servidores *edge*, roteadores inteligentes, estação base, entre outros. Dependendo do número de dispositivos próximos em uma mesma região, eles podem não apresentar bom desempenho e replicar os problemas encontrados na nuvem. De maneira paralela, e também complementar, as pesquisas apresentadas a seguir mostram que o processamento nos próprio dispositivos voltam a fazer sentido.

A Figura 13 ilustra um exemplo da computação na borda, onde o *offloading* horizontal é realizado em busca de maior poder computacional entre os dispositivos da camada 1.

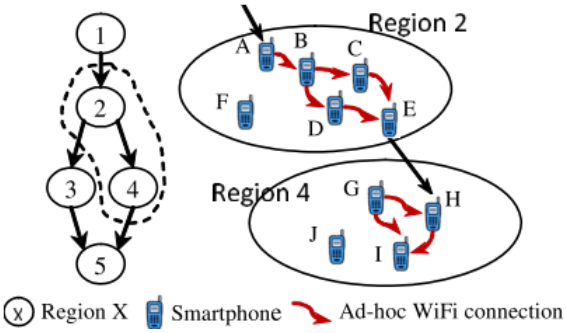


Figura 13 – *Offloading* horizontal na camada 1 - Fonte: (WANG; PEH, 2014)

(SOTO et al., 2016) apresentam *CEML*, um *framework* para processamento de eventos complexos integrado à técnicas de *machine learning*, que é executado em dispositivos na *Mist*. De maneira similar, (CARDOSO et al., 2018) realizam um estudo para, através de CEP, detectar ataques de negação de serviço na IoT. As pesquisas, apesar de não mencionarem a possibilidade de *offloading*, mostram que é possível realizar CEP em dispositivos com pouco poder de processamento.

(WANG; PEH, 2014) apresentam o *Mobistreams*, um DSPS formado por *clusters* em uma *Mobile Adhoc Network* (MANET). A ideia principal é trazer o processamento dos servidores remotos para os *smartphones* com objetivo de aliviar o *overhead* na rede de telefonia. O *Mobistreams* enfatiza a tolerância a falhas no contexto de *smartphones*, considerando o ambiente limitado em bateria e processamento. Por fim, o trabalho aponta que o processamento em servidores remotos pode ser otimizado através do processamento em redes locais, mesmo quando realizadas em dispositivos de menor poder computacional. O *Mobistreams* é voltado para *smartphones*, não realiza processamento nos produtores de dados e não processa eventos complexos.

(CHOOCHOTKAEW et al., 2017) propõem o *EdgeCEP*, uma estratégia de distribuição de processamento de eventos complexos entre dispositivos da IoT com base em *brokers* distribuídos. O *EdgeCEP* define um plano de execução com base nas distâncias entre os nós e as consultas são especificadas em uma linguagem própria. O trabalho não traz o uso de políticas nem menciona a utilização de contexto.

(DAUTOV; DISTEFANO, 2017) sugerem uma arquitetura em camadas para fusão de dados através de processamento de eventos. O trabalho explora apenas filtros, e não avalia outros potenciais do engenho *Drools* utilizado. A pesquisa, apesar de trabalhar com as três camadas, e a possibilidade de *offloading*, não propõe um processamento sensível à contexto.

(KAMISIńSKI; GOEBEL; PLAGEMANN, 2013) e (STARKS; PLAGEMANN, 2015) propõem uma arquitetura genérica para cenários de mobilidade baseada na divisão de consultas CEP entre os agentes de processamento. Através de uma abordagem *Top-Down*, cada nó da rede é responsável por executar uma parte da consulta, formando a ideia de árvores de processamento. Cada folha da árvore fica encarregada de processar eventos simples, enquanto seus ancestrais consolidam os dados gerando eventos complexos. A arquitetura considera estratégias de distribuição e divisão das consultas a partir de uma classificação fixa e pré-definida em relação à mobilidade dos provedores de dados em redes estruturadas. As pesquisas não tratam a possibilidade do *offloading* operacional.

2.6.4 *Web of Things*

Recentemente, alguns trabalhos considerando a análise de dados com CEP no contexto de WoT foram publicados. (MORENO et al., 2018) utilizam processamento de eventos complexos para detectar incertezas, como falha na seleção e identificação dos eventos

(*matches*), aumentando a precisão e corretude das notificações. Porém, a pesquisa, apesar de mencionar a WoT, não explora seu universo. A execução ocorre em computadores pessoais, e nenhum protocolo compatível com a *Web* é mencionado. Já em (KAMILARIS et al., 2017), um cenário de cidades inteligentes é explorado, com coleta dos dados nos dispositivos da *WoT* baseado em serviços de descoberta. Os dados são enviados ao servidor CEP, que é localizado na *Web*.

2.6.5 Comparação dos trabalhos relacionados

Assim como os estudos apresentados, esta tese também propõe uma solução para processamento de fluxo de dados na IoT, porém considera a possibilidade de análise dos dados também nas *things*, diferente das soluções voltadas para computação na *Fog*.

Numa abordagem similar à (CARDELLINI et al., 2018), este trabalho usa o conceito de políticas e regras que analisam o consumo de recursos de um nó, mas os componentes arquiteturais serão projetados para a IoT, podendo também ser executado em dispositivo da camada 1.

Diferente de (DAUTOV; DISTEFANO, 2017), o objetivo é trazer o potencial da técnica de CEP para a borda da rede, e não apenas utilizá-la como filtro de dados, mas também executar consultas complexas. E numa estratégia análoga à (CHOOCHOTKAEW et al., 2017), utiliza uma espécie de *broker* para a distribuição das tarefas, porém, esse pode realizar o *offloading*, caso dispositivos da *Mist* não estejam disponíveis.

A estratégia apresentada por (WANG; PEH, 2014), apesar de considerar a mobilidade dos *smartphones*, não realiza processamento nos próprio produtores de dados e não executa processamento de eventos complexos. Já as arquiteturas propostas por (KAMISIńSKI; GOEBEL; PLAGEMANN, 2013) e (STARKS; PLAGEMANN, 2015), apesar de utilizarem CEP, consideram mobilidade apenas dos agentes de processamento, mas assumem produtor e consumidor fixos. Ambos não processam dados nos produtores, e possuem como premissa agentes de igual poder computacional. Em suas arquiteturas, as sub-árvores de processamento não apresentam dinamismo compatível com a mobilidade da IoT.

A arquitetura *GiTo* apresenta uma abordagem onde as *things* são também agentes de processamento, e a análise dos dados é realizada de maneira colaborativa nas três camadas. Utilizando uma estrutura sensível a contexto, os dispositivos são integrados através de protocolos já conhecidos na Web, viabilizando a troca de informação na WoT e permitindo a execução do CEP colaborativo.

3 ARQUITETURA GITO

Com a evolução dos dispositivos da IoT, que além de conectividade podem possuir recursos computacionais de modo a permitir um processamento eficiente, é possível efetuar análise de dados localmente, trazendo a computação para a verdadeira borda da rede. Esse movimento de usar o poder computacional do próprio dispositivo enfatiza o uso de um novo paradigma chamado de *Mist computing* (YOGI; CHANDRASEKHAR; KUMAR, 2017) (PREDEN et al., 2015). Porém, a IoT ainda apresenta vários pontos de melhoria, principalmente em relação à interoperabilidade dentro de um ecossistema heterogêneo de tecnologias (NEUMANN et al., 2016). Nesse contexto, a WoT é considerada um refinamento da IoT através da integração de coisas inteligentes não apenas à camada da Internet (infraestrutura de rede), mas também à Web (camada de aplicação) (GUINARD; TRIFA; WILDE, 2010). A WoT propõe que padrões já utilizados na Web, como HTTP e REST sejam reusados, adaptados e integrados a IoT, facilitando assim a troca de informações entre dispositivos e servidores remotos (TRAN et al., 2017).

Considerando as abordagens na *Cloud*, na *Fog* e na *Mist*, este capítulo apresenta a arquitetura *GiTo*: uma uma estrutura orientada à políticas para coordenação de processamento de eventos complexos na WoT. A *GiTo* foi projetada para, em tempo de execução, tomar a decisão de onde processar os dados com base em um conjunto de políticas.

GiTo é um codinome para *Global IoT*. Daqui em diante, o termo *sistema* representa qualquer aplicação que implementa essa arquitetura.

3.1 Metodologia e Planejamento

Com objetivo de entender melhor o universo dos IFPS e propor um arquitetura genérica que seja realmente aplicável à sistemas reais que tratam fluxos contínuo de dados, este trabalho foi guiado por um planejamento baseado em uma metodologia para projeto de arquitetura e *frameworks* já consolidada.

As principais metodologias nessa área apresentam as seguintes abordagens:

- *Bottom-Up*
  A partir de aplicações já existentes é feita uma análise com objetivo de obter um entendimento detalhado das características internas e identificar pontos em comum. Uma vez essa análise realizada, trabalha-se na construção da base do modelo gerando classes a partir de exemplos concretos (JOHNSON, 1997)(CRESPI; GALSTYAN; LERMAN, 2008).
- *Top-Down*
  É formulada uma visão global do modelo sem detalhamento de seus componentes

internos, gerando uma macro-arquitetura. No processo de refinamento, seus componentes são detalhados até atingir um nível em que se possa validar o modelo original (BARROS, 2007)(BRUIN; VLIET, 2002).

A abordagem *bottom-up* evita que seja construída uma solução genérica demais (problema causado pelo método *top-down*), enquanto que a estratégia *top-down* reduz a chance do modelo ficar bastante especializado no conjunto de aplicações que é utilizado na sua construção (problema inerente à abordagem *bottom-up*) (BARROS, 2007). Buscando um equilíbrio que reduz os problemas inerentes à cada das estratégias citadas, este trabalho utilizará uma abordagem híbrida.

A metodologia utilizada se baseia no modelo apresentado por (PARSONS et al., 1999) e é composta por 6 fases que se assemelham às etapas de Análise e Projeto encontradas em um ciclo de desenvolvimento de software (SOMMERVILLE; ARAKAKI; MELNIKOFF, 2008). Uma vez compreendendo o domínio que a arquitetura está inserida e suas características, projetam-se os principais componentes e suas conexões para posterior desenvolvimento de um sistema. Nesse último, componentes adicionais podem ser criados, atualizando assim o projeto inicial.

Diante do exposto, tem-se a definição das atividades executadas na busca da arquitetura final, como mostrado na Figura 14.

Figura 14 – Definição da arquitetura: Planejamento das atividades

A lista a seguir detalha as atividades, descrevendo o que será contemplado em cada uma delas.

1. A fase de análise do domínio busca identificar características na execução do processamento de fluxo de dados e será realizada em 3 etapas:

   - Estudar sobre a área de processamento de fluxos de dados para obter o entendimento teórico dos principais conceitos e sua aplicabilidade;
   - Buscar em ferramentas e aplicações desenvolvidas na indústria o que de fato é aplicável;

- Desenvolver aplicações utilizando as ferramentas identificadas no passo anterior com objetivo de aplicar os conceitos na prática, visualizando-os de maneira concreta e operacional.

2. Com base nas características identificadas e no entendimento construído na fase 1, especificar os requisitos funcionais e não funcionais que irão delinear a arquitetura.
3. Projetar a arquitetura *GiTo*, contemplando os requisitos especificados.
4. Implementar:
   - Um sistema genérico para processamento de fluxo de dados com base na definição da arquitetura;
   - Uma aplicação utilizando como base o sistema desenvolvido.
5. Avaliar a arquitetura através de um experimento que mostre as vantagens e desvantagens de sua utilização.
6. Analisar e apresentar os resultados obtidos.

3.2 Análise do Domínio

Como mencionado, essa fase é composto de 3 sub-etapas. Após os estudos realizados e o entendimento conceitual formado, os trabalhos relacionados foram identificados (ver Seção 2.6). Além desses, na sub-etapa 2, foram identificadas e analisadas ferramentas utilizadas na indústria,e percebeu-se que um sub-conjunto delas serve de base para a grande parte das aplicações. Com objetivo de entender melhor as ferramentas, aplicações que processam fluxos de dados foram desenvolvidas.

As principais ferramentas foram criadas por empresas do mercado, como *Linkedin*, *IBM*, *Twitter*, *JBoss* e *Red Hat*. Em particular, a plataforma *Storm*, criada pelo *Twitter*, apresenta uma comunidade bastante ativa (IQBAL; SOOMRO, 2015), e vem ganhando cada vez mais atenção depois que se tornou um projeto *Apache*. *Benchmarks* mostram que o *Apache Storm* é uma excelente escolha quando se deseja uma baixa latência do processamento (WANG, 2016). Porém, o *Storm* não realiza CEP. Nesse contexto, a ferramenta *Esper* é considerada como o principal engenho *open-source* do mercado (CUGOLA; MARGARA, 2012). *Esper* possui uma linguagem para especificação de regras baseada em SQL, permitindo declarações de consultas complexas de maneira simplificada. Por esses motivo, este trabalho optou por analisar e desenvolver aplicações utilizando essas duas plataformas.

Na segunda sub-etapa foram desenvolvidas 4 aplicações utilizando o *Storm* e o *Esper*:

1. Verificação de fraudes em promoções na *black friday* brasileira;
2. Detecção de ataques em redes de computadores;

3. Simulação do *Trend Topics* do Twitter;
4. Monitoramento de lâmpadas em um ambiente residencial simulado.

Nesta análise de domínio, foi observado um conjunto de características durante o levantamento dos trabalhos relacionados e a construção das aplicações. Essas peculiaridades se destacaram e servirão como entrada para a especificação dos requisitos.

- A utilização de componentes arquiteturais para a coleta de dados;
- O uso de balanceamento de carga entre servidores distribuídos;
- A falta de contexto computacional na divisão e de distribuição do processamento;
- A *bufferização* e agregação de dados;
- A ausência de transparência de localização em alguns casos;
- Os componentes flexíveis, os quais podem ser integrados à solução em tempo de execução.

Essas informações ajudaram na elicitação dos requisitos, influenciando diretamente a proposta de arquitetura.

3.3 Requisitos

Uma vez realizada a Análise do Domínio, os requisitos funcionais e não funcionais foram especificados. A arquitetura *GiTo* deve ser flexível, uma vez que sistemas compatíveis com ela serão executados na *Mist*, a *Fog* e a *Cloud*.

O objetivo geral da proposta apresentada é processar fluxos contínuos de dados em tempo real.

3.3.1 Requisitos Funcionais

1. **Contexto e Regras**

   - **RF01** - Capturar informações contextuais
     Deve-se coletar informações do ambiente onde está ocorrendo o processamento. Por exemplo: consumo de CPU e memória do dispositivo, métricas do processamento dos dados, como taxa de frequência, número de eventos recebidos e informações sobre estado das conexões e consumo energético.
   - **RF02** - Monitorar o uso dos recursos disponíveis
     Limites relacionados ao uso dos recursos e métricas do processamento devem ser especificados como regras de controle a serem respeitadas durante todo a execução. O monitoramento dessas regras é essencial.

2. **Coletar e Processar Dados**

   - **RF03** - Coletar Dados
     Deve-se capturar os dados proveniente dos sensores conectados ao dispositivo. Idealmente, o processo de coleta foi otimizado para apenas encaminhar o registro para o análise, não realizando outras operações, como filtragem ou qualquer funcionalidade de negócios. O mesmo deve acontecer para dados recebidos da rede durante o processo de *offloading*.
   - **RF04** - Processar os dados
     Deve-se processar fluxos contínuo de dados em tempo real provenientes de sensores acoplados aos dispositivos, assim como aqueles recebidos pela rede. Deve-se ao máximo otimizar a análise dos dados em busca de uma redução no tempo de resposta das notificações encontradas. Caso o processamento em um único dispositivo esteja carregado, este deve ser distribuído.
   - **RF05** - Controlar o fluxo de dados
     Após a coleta dos dados, caso o dispositivo não tenha poder computacional para realizar o processamento e nem consiga fazer o *offloading*, deve-se parar ou reduzir o fluxo de dados, diminuindo, por exemplo, a frequência de coleta.

3. **Offloading**

   - **RF06** - Realizar *offloading* computacional
     Quando um dispositivo não pode mais processar os dados localmente, este deve tentar realizar a transferência parcial ou total dos dados e do processamento para um outro dispositivo presente na *Mist*, na *Fog* ou na *Cloud*.
   - **RF07** - Descobrir Dispositivo para *Offloading*
     Antes de realizar o *offloading* dos dados, deve-se definir para qual dispositivo os dados serão enviados. Para isso, é necessário realizar uma busca para encontrar dispositivos próximos habilitados à colaborar no processamento.

     O dispositivo atual deve informar as características dos dados e do processamento para que os outros dispositivos possam verificar se podem ou não realizar a análise dos dados. Deve-se primeiro tentar realizar o *offloading* na *Mist*, depois na *Fog* e posteriormente na *Cloud*.

     A busca por novos dispositivos deve suportar transparência de localização.

4. **Mobilidade**

   - **RF08** - Suportar Mobilidade
     A arquitetura deve prever a mobilidade dos dispositivos, garantindo a correta execução quando um dispositivo trocar de área, e sair, por exemplo, do domínio de um determinado servidor.

3.3.2 Requisitos Não-Funcionais

1. **RNF01** - Flexibilidade
   A arquitetura deve possuir componentes com fraco acoplamento entre si. Esses poderão ser ativados ou desativados, dependendo do poder computacional de cada dispositivo.

2. **RNF02** - Desempenho
   O tempo de resposta das notificações no processamento dos dados deve ser menor quando comparado aos cenários executados de maneira isolada por dispositivos que não tinham poder computacional no momento do processamento.

3. **RNF03** - Capacidade de trabalhar *off-line*
   Na ausência de conectividade, os dispositivos devem agir de maneira isolada, garantindo seu funcionamento. Porém, respeitando as regras definidas.

3.4 Arquitetura

Com o objetivo de atender os requisitos definidos, a arquitetura *GiTo* sugere a criação de uma infraestrutura de coordenação com base em políticas no processamento de eventos complexos. Assim, os dispositivos poderão decidir qual o melhor local para processar os dados por eles produzidos.

Numa visão macro e conceitual, a Figura 15 ilustra uma ideia geral da solução derivada deste trabalho. A camada mais externa contém os dispositivos da IoT, que são produtores de dados e representam os nós mais na extremidade da arquitetura. Eles são capazes de tomar algum tipo de decisão e possuem a autonomia para se conectar a outro nó, caracterizando a chamada *Mist computing* (PREDEN et al., 2015). Caso esses dispositivos apresentem restrição para analisar seus próprios dados, eles podem realizar o *offloading* computacional para nós na mesma camada ou de camadas mais ao centro.

A camada intermediária contém os *edge servers*, que fornecem suporte aos dispositivos da borda com intuito de adicionar recursos extras, criando uma opção complementar para o processamento. Essa camada é conhecida como *Fog*, e nela, os dispositivos presentes se comunicam entre si, criando um barramento para compartilhar informações. Por fim, a camada ao centro é representada pelo servidor na nuvem, oferecendo capacidade computacional em larga escala ao sistema como um todo.

Na *Mist*, os dispositivos vizinhos criam um *cluster* podendo trocar informações. Quando o dispositivo atual precisa delegar seu processamento, ele realiza uma busca por outros dispositivos no seu próprio *cluster*. Caso não tenha sucesso, ele pode se comunicar com os servidores na *Fog* ou *Cloud*. Assim, tem-se o *offloading* horizontal (na mesma camada) ou vertical (entre camadas), onde mais de um dispositivo colabora no processamento. É recomendado que, mesmo quando um dispositivo da *Mist* processa seus dados, ele

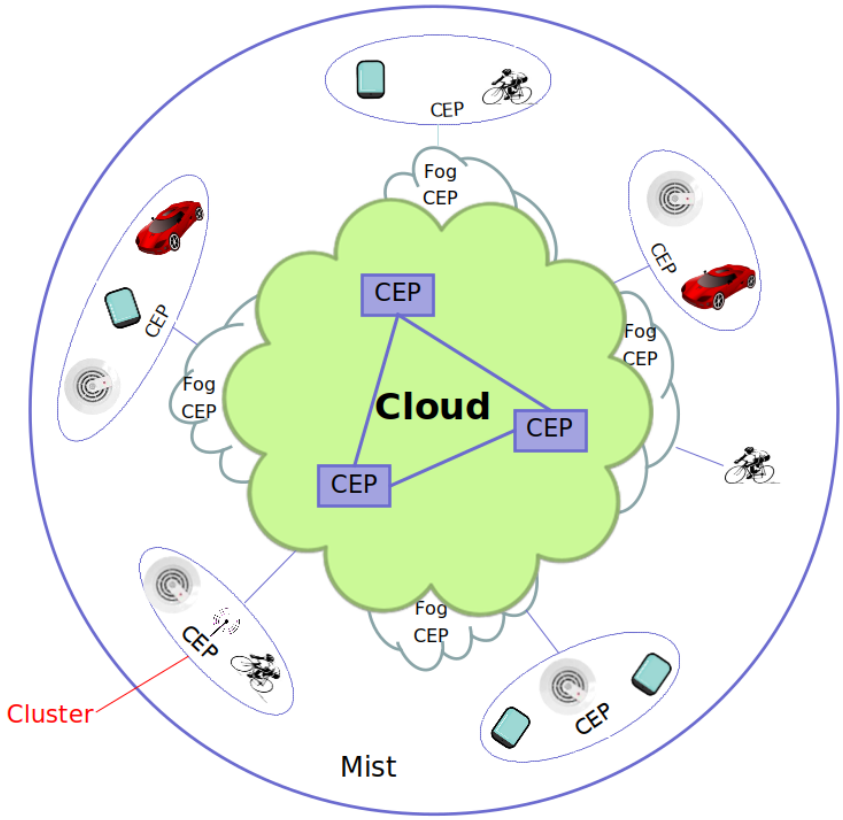


Figura 15 – Visão geral da Arquitetura *GiTo*

compartilhe o resultado do processamento com as camadas mais ao centro, fornecendo informações que podem enriquecer futuras tomadas de decisão por parte do sistema como um todo.

O engenho CEP é executado em todas as camadas, por todos os dispositivos, formando assim um CEP distribuído.

3.4.1 Agentes de Processamento

Trazendo a visão conceitual da Figura 15 para uma visão clássica em camadas, chega-se à Figura 16(a), onde, a camada inferior representa a *Mist*, a intermediária ilustra a *Fog* e a camada superior associada à *Cloud*. É possível ter, através dessa visão, uma noção da hierarquia em camadas, de modo que, os dispositivos das camadas superiores tendem a processar um conjunto maior de dados.

Visualizando os dispositivos da IoT como nós de uma rede de computadores, tem-se a arquitetura *GiTo*, como ilustrado na Figura 16(b). Nesta perspectiva, os dispositivos são visualizados de forma independente e se comunicam para realizar tarefas em comum (TANENBAUM; STEEN, 2002). Na arquitetura aqui proposta, esses nós são chamados de agentes de processamento *GiTo*, pois são capazes de processar tanto os dados produzidos por eles próprios quanto os enviados por outro agente, respeitando os requisitos definidos.

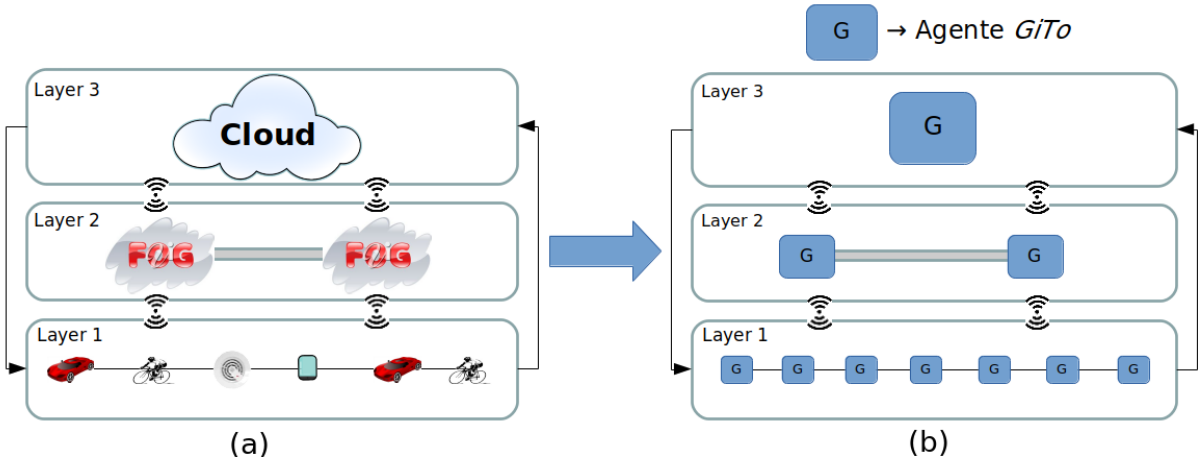


Figura 16 – Abstraindo cenário como agentes de processamento

Um agente *GiTo* representa um nó que pode estar inserido em qualquer camada. Ele é representado por um computador, que independente da camada a qual pertence, apresenta algum tipo de restrição, como poder de processamento, memória, energia entre outros. De acordo com o IETF, esses agentes são conhecidos como *Constrained Nodes* (BORMANN M. ERSUE, 2018), e são classificados em dois grupos: (M) microcontroladores e (J) propósito geral. No grupo J, os dispositivos tendem a apresentar memória RAM e Flash em chips separados (ex. *Raspberry Pi*).

A arquitetura *GiTo* é voltada para processamento de dados em nós que possuem conectividade e que podem se comunicar com servidores remotos localizados na Internet. Por essa razão, consideram-se como agentes *GiTo* os dispositivos do grupo J na classificação do IETF. Esses agentes possuem capacidade computacional para suportar protocolos como HTTP, *Constrained Application Protocol* (CoAP) e *Websocket*, viabilizando suas integrações com servidores na Web, e contextualizando-os na WoT. A Figura 17 ilustra o agente.

O agente *GiTo* localizado na *Mist* coleta dados dos sensores acoplados a ele, como temperatura, umidade, localização. Ele pode receber outras informações através das interfaces de rede, ou realizar o *offloading* parcial ou total dos seus dados para agentes próximos ou para os servidores na *Fog* ou *Cloud*.

No restante deste trabalho, o termo dispositivo se refere ao agente *GiTo*.

3.4.2 Componentes

Apresentada a visão geral da arquitetura, pode-se detalhar seus componentes internos. Esses componentes estão presentes em todo agente *GiTo*, e dependendo da configuração e poder computacional do dispositivo, podem ser ativados ou desativados. A Figura 18 ilustra os componentes e seus relacionamentos. Todo agente compatível com essa arquitetura

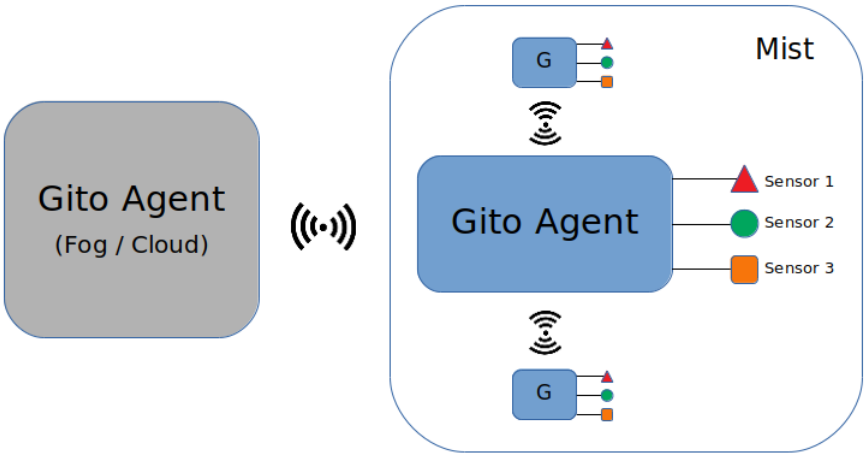


Figura 17 – Agente *GiTo*: sensores conectados, comunicação com outros agentes na *Mist*, *Fog* e *Cloud*

está habilitado a participar do processamento de dados colaborativo.

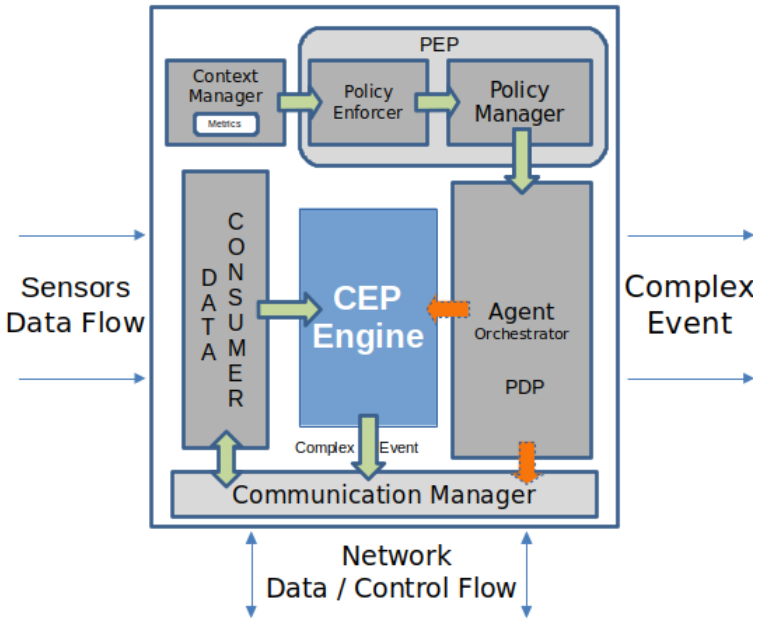


Figura 18 – Componentes da arquitetura *GiTo*

O componente *CEP engine* influencia diretamente no consumo dos recursos do dispositivo, pois é nele que os dados são analisados. Logo, quando um agente realiza o *offloading*, o engenho CEP é desativado e os dados são migrados para outro nó. Por esse motivo, na Figura 18, o engenho CEP é apresentado em destaque. As setas contínuas indicam que existem troca de mensagens entre os componentes. Já as setas tracejadas são uma informação lógica mostrando que o Agente Orquestrador direciona o processamento para o engenho local ou distribui pela rede.

A decisão de um dispositivo ativar ou não determinado componente pode ser tomada em dois instantes: em sua configuração e durante sua execução. Na configuração, é preciso

classificar os dispositivos. A partir desta classificação, é possível determinar previamente quais componentes serão executados em quais agentes. Durante a execução, um agente que está com todos os componentes ativos pode precisar interrompê-los devido às políticas estabelecidas.

A seguir, uma breve descrição de cada componente, informando suas responsabilidades.

- **Data Consumer**
  Responsável por coletar os dados provenientes dos sensores e repassar para o engenho CEP.

- **CEP Engine**
  O engenho CEP é responsável por processar os dados e detectar os eventos complexos. Os dados que serão processados podem chegar a partir do *Data Consumer* ou do *Communication Manager*.

- **Agent Orchestrator**
  Componente responsável por definir os agentes envolvidos durante o processamento dos dados. Representa o PDP juntamente com o *Policy Manager*.

- **Context Manager**
  Gerencia as informações de contexto definindo aquelas que serão utilizadas pelo agente em questão, e repassa seus estados ao *Policy Enforcer*.

- **Policy Enforcer**
  Garante o monitoramento das regras de políticas. Sempre que uma regra for violada, uma notificação é enviada ao *Policy Manager*. Faz parte do PEP.

- **Policy Manager**
  Esse componente faz parte tanto do PEP como do PDP. É responsável pela execução das políticas. Quando uma regra é violada, avalia-se as políticas que são influenciadas pela regra em questão. Caso alguma política seja violada, orquestrador de agentes é notificado.

- **Communication Manager**
  Gerencia as conexões de entrada e saída e valida o protocolo de aplicação. Os dados coletados por outros agentes que chegam da rede através do processo de *offloading* são fornecidos para o componente CEP.

3.4.3 Consumidor de Dados

O *Data Consumer* deve ter o mínimo de processamento necessário para coletar os dados dos sensores. Ou seja, deve-se apenas ler/capturar a informação e repassar para o dado

recebido ao engenho CEP tão logo ele coleta. Caso o *offloading* esteja sendo realizado, o dado é redirecionado através do *Communication Manager*.

É recomendado que nenhuma operação ou tratamento seja feito da informação, e que outros componentes assumam o processamento do dado recém-chegado. Caso seja necessária alguma transformação, um *wrapper* realizado por outro componente (abstraído na arquitetura) pode ser feito.

A otimização na execução do *Data Consumer* é necessária para que o mesmo não se torne um gargalo na análise. Porém, caso o *offloading* para outro agente não seja possível, o orquestrador de agentes pode solicitar uma redução na frequência de leitura dos dados, gerando assim uma "descanso" no processamento.

3.4.4 Engenho CEP

Como visto na Seção 2.3.1, um engenho CEP é preparado para analisar os dados provenientes de múltiplos fluxos em tempo real. O componente em si, deve ser leve para que possa ser executado em um agente *GiTo* e deve suportar consultas de diferentes complexidades, como filtros, agregação, padrões, janela de tempo entre outros.

O engenho se comunica com o *DataConsumer* para receber os dados, e notifica a aplicação que está sendo executada localmente e a rede, através de uma *interface*. Assim, ele deve suportar mais de uma consulta ao mesmo tempo.

Este componente pode ser ativado e desativado através de um arquivo de configuração ou dependendo do estado do dispositivo. Neste último, o orquestrador de agentes é responsável pelo seu ciclo de vida, iniciando e finalizando de acordo com as políticas especificadas.

3.4.5 Orquestrador de Agentes e o Fluxo de Dados

Quando um dispositivo detecta que não está habilitado a analisar os dados produzidos por ele próprio, o processo de *offloading* é realizado. Essa tomada de decisão leva à necessidade de um componente específico chamado de *Agent Orchestrator*, que gerencia o fluxo de dados e especifica qual novo nó deve continuar o processamento que está sendo realizado pelo dispositivo atual. A Figura 19 apresenta o orquestrador, mostrando seus *inputs* e *output*. Este componente define o novo agente com base no contexto dos nós, nas regras para o processamento dos dados e nas diretrizes definidas para a execução. Estas diretrizes são definidas como políticas, que quando violadas, disparam ações visando manter a estabilidade do sistema como um todo.

O orquestrador é executado de forma distribuída em todos os nós. Quando um agente não pode mais continuar o processamento, ele estabelece uma conexão com outro agente, e seus orquestradores trocam informações para determinar se o *offloading* é possível. Esse

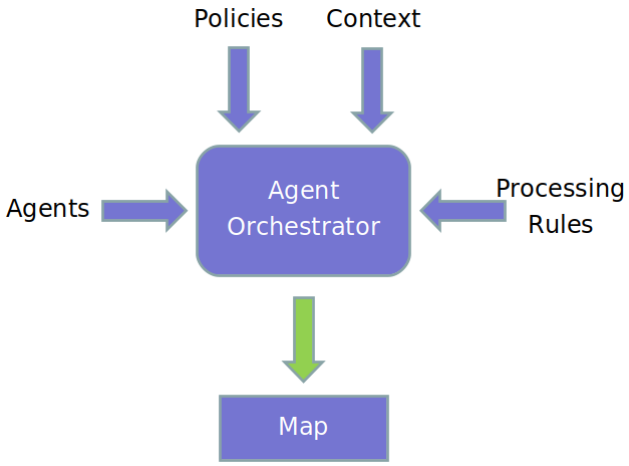


Figura 19 – Orquestrador de agentes

componente produz como saída um mapa de distribuição que indica qual ou quais agentes participarão do processamento dos dados a partir daquele determinado nó.

É responsabilidade do orquestrador definir em tempo de execução o fluxo percorrido pelos dados através dos agentes *GiTo*, determinando quais dispositivos irão colaborar no processamento.

3.4.5.1 *Função para Tomada de Decisão*

Dada a existência de um orquestrador de agentes, que recebe como entrada um conjunto de políticas e informações contextuais do ambiente onde o sistema distribuído está sendo executado, pode-se definir que cada agente decida se ele próprio é capaz de colaborar no processamento dos dados. Para tal, cada um deve implementar uma função de tomada (Ftd) de decisão que recebe 3 parâmetros e retorna um valor verdadeiro ou falso. Essa função é dependente de plataforma e é definida na fase de implementação, conforme a seguinte assinatura:

$$\textbf{\textit{boolean}}\ Ftd(W, RT, P)$$

Onde, W representa o *workload* atual do agente que deseja realizar o *offloading*, contendo características dos dados e da complexidade da análise que está sendo realizada. RT é o seu tempo de resposta máximo (caso essa política esteja declarada), e P é o conjunto de políticas relacionado ao agente que receberá os dados. A função calcula o custo computacional e verifica se o agente está habilitado a efetuar o processamento. A função de tomada decisão retorna V, um valor indicando se é possível ou não realizar o *offloading*. Essas informações são trocadas no *handshaking* associado ao processo de *offloading*.

Por exemplo, um dispositivo X está processando um conjunto de dados, e detecta que sua execução está lenta (política violada) para a necessidade de uma determinada aplicação. Este, envia para outros agentes, informações relacionadas ao conjunto de dados, como frequência de coleta, características da análise efetuada e o tempo de resposta.

Os agentes, ao receberem essas informações, verificam se tem condições de atender a demanda, considerando suas próprias restrições (políticas) e tarefas atuais.

3.4.6 Políticas e Contexto

Como detalhado na seção anterior, o orquestrador de agentes é responsável por definir por qual conjunto de dispositivos o fluxo de processamento irá seguir. Para tal, é preciso compreender as políticas que influenciarão nessa tomada de decisão, assim como quais as informações de contexto serão consideradas nessa seleção.

A estratégia adotada na arquitetura *GiTo* é baseada na aplicação do *loop* de controle MAPE-K (Seção 2.5), onde o *ContextManager* monitora o agente, o *PolicyEnforcer* e o *PolicyManager* analisam o contexto, e o orquestrador de agentes fica responsável por planejar e executar as ações.

3.4.6.1 *Contexto*

As pessoas e os objetos tomam decisões com base no contexto que estão inseridos. As informações contextuais auxiliam na interpretação do ambiente onde os dispositivos estão envolvidos, viabilizando a escolha de uma (re)ação apropriada, que irá agir sobre os próprios dispositivos, alterando assim o estado do contexto (Seção 2.4).

As atividades básicas de um componente sensível ao contexto são: aquisição, interpretação, disseminação e (re)ação. Na IoT, a aquisição pode acontecer a partir de diversas fontes, cada uma possuindo sua característica, precisão e custo/esforço associados. Por exemplo, a localização de um determinado dispositivo pode ser obtida a partir do seu GPS, ou inferida a partir do servidor *Fog* da região onde ele se encontra ou mesmo por outro dispositivo vizinho.

Na arquitetura *GiTo*, o orquestrador de agentes recebe um conjunto de informações utilizadas no processo de tomada de decisão. Todavia, nem todos os sensores podem prover dados sem interrupção, devido a problemas já conhecidos, como baixa bateria, perda de conectividade entre outros. A Tabela 1 apresenta uma lista (não exaustiva) de itens contextuais, classificando-os quanto à etapa de aquisição em primária ou secundária (ver Seção 2.4).

O *Context Manager* captura os valores dos itens de contexto e faz uma pré-análise transformando o dado em informação. Nessa etapa o dado é convertido para uma representação comum para posterior compartilhamento com o *Policy Enforcer*. Por fim, a etapa de reação, também chamada de fase de tomada de decisão, é definida como parte da execução do orquestrador de agentes. Ou seja, uma variação no contexto leva o sistema a se reconfigurar de acordo com as políticas definidas.

Tabela 1 – Informações contextuais

Item de Contexto	**Descrição**	**Classificação**
Localização	Localização do agente	Primária Secundária
Identidade	Identificador do agente	Primária
Poder Computacional	Características de SW/HW do agente (CPU, memória etc.)	Primária
Tipo do Dado	Semântica associada ao dado	Secundária
Taxa de transferência dos dados	Quantidade de dados produzidos por segundo pelo dispositivo da WoT	Primária
Mobilidade	Nível de mobilidade baseado no tipo do dispositivo	Secundária
Energia	Percentual de bateria	Primária
Conectividade	Qualidade da conexão estabelecida com outro agente	Primária Secundária
Tempo de Retorno	Tempo de envio do evento, mais processamento, mais recebimento da notificação	Secundária

3.4.6.2 *Políticas*

A arquitetura *GiTo*, através do orquestrador, flexibiliza o processamento de dados em tempo real, ora executando no próprio agente, ora realizando o *offloading* para um outro dispositivo. Contudo, para que o orquestrador tenha ciência de quando tomar a decisão, é preciso que restrições comportamentais da execução sejam especificadas. Essa definição é realizada através da declaração de políticas e visa manter a execução consistente e fiel às suas especificações.

Como mostrado na Seção 2.5.1, uma política é definida com base em um alvo ou escopo de atuação. As políticas exercem restrições (R) sobre os atributos do sistema alvo (S), que, quando violadas, podem mudar seu estado através da execução de uma determinada (re)ação. Ou seja, quando o estado atual do sistema recebe algum estímulo (E), um conjunto de regras é verificado, impedindo que o sistema alcance um estado não esperado. Assim, pode-se definir uma política como uma função P, onde:

$$P : (E, S, R) \implies A$$

A função P tem como retorno um conjunto de ações A. Para cada item contextual presente na Tabela 1, pode-se ter um conjunto de políticas implementadas. Considere

a situação exemplo onde um sistema (S) está processando um conjunto de dados e a política *tempo de resposta máximo* da análise desses dados é 2s (restrição - R). Durante a execução, a latência de rede aumenta (novo estímulo - E) e o tempo de resposta alcança 3s. O sistema detecta essa variação e consequente violação da política e notifica o orquestrador que dispara uma ação (A) garantindo que o tempo de resposta volte para um valor menor ou igual à 2s.

3.4.6.3 *Classificação das Políticas*

Visando flexibilizar a definição de regras e fornecer possibilidades de combinar um conjunto de restrições, a arquitetura *GiTo* classifica suas políticas como *simples* ou *composta*.

Uma política simples é definida como uma regra contendo uma única restrição do sistema, formada por um nome e um valor, como no exemplo anterior, onde *tempo de resposta máximo*=2s. Independente de outras políticas existentes, quando uma política simples é violada, o orquestrador de agentes é notificado. Já a política composta permite a composição de regras através da combinação de políticas simples.

A definição de uma políticas é detalhada na Seção 3.4.6.5.

3.4.6.4 *Instanciando o Padrão IETF/DMTF*

A *GiTo* utiliza a arquitetura de políticas recomendada pelo IETF/DMTF (ver Seção 2.5.3) com objetivo de garantir o correto uso das regras e restrições declaradas.

O padrão do IETF/DMTF é utilizado pelos principais sistemas de gerenciamento baseados em políticas (*Policy-based Management System - PBMS*), pois captura os princípios por trás desse tipo de sistema (AGRAWAL SERAPHIN CALO, 2008). Logo, a arquitetura aqui proposta também utiliza esse padrão como base para a estratégia de gerenciamento de políticas.

Como mostrado na Figura 11, o padrão *IETF* consiste em quatro componentes, onde dois deles foram integrados à arquitetura *GiTo*: o *PolicyManager* e o *Policy Enforcer*. Eles determinam como as políticas são monitoradas e instanciados conforme a Figura 20.

Todo o ciclo é iniciado através da PMT. Uma vez a política criada e definida, ela será armazenada no repositório de políticas (PR) (fig. 20(a)). Esse repositório pode ser um banco de dados, um arquivo de configuração ou mesmo informações recebidas pela rede e salvas em memória. A forma ideal de armazenar as políticas é dependente de plataforma e é definida na fase de implementação. Sempre que for necessário analisar um conjunto de políticas contra determinada situação, é preciso filtrá-lo de acordo com o estado do ambiente da execução (fig. 20(b)) (ex. não faz sentido monitorar latência de rede se o processo é local). Uma vez definido o escopo de políticas, este é encaminhado ao *Policy Manager* (fig. 20(c)). Esse gerenciador analisa o conjunto de políticas resultante e o envia ao *Policy Enforcer* (fig. 20(d)). Quando uma regra é violada, o *Policy Manager*

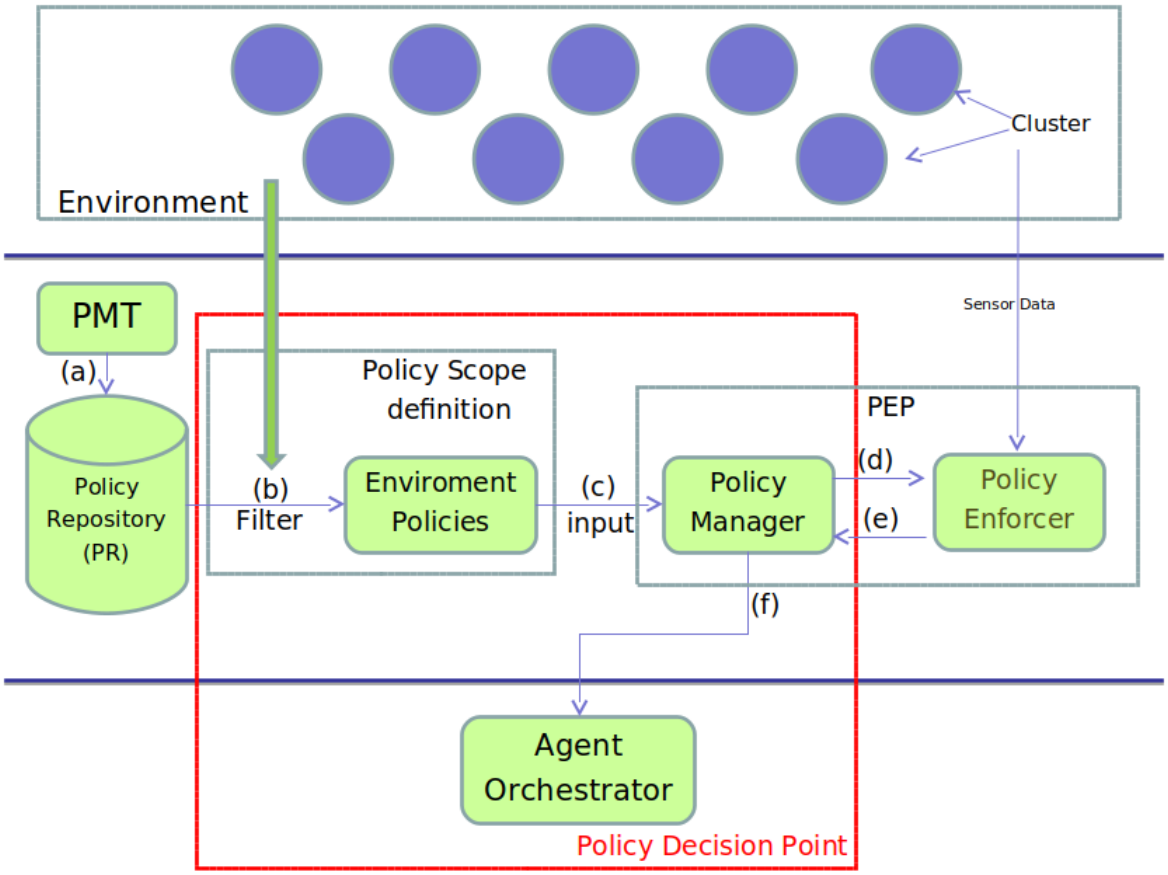


Figura 20 – Gerenciamento e estratégia de uso das Políticas

é notificado 20(e)). Caso a política seja do tipo simples, ele dispara um evento para o orquestrador de agentes para que uma ação seja tomada (fig. 20(f)). Se a informação recebida estiver associada à uma política composta, o *manager* analisa todas as regras da política, checando o momento certo para notificar o orquestrador. A análise dos dados continuará após execução do Orquestrador, que determinará onde eles serão processados. Nessa estratégia de políticas, o componente PDP é formado pelos filtro, *policy manager* e orquestrador de agentes.

Com intuito de facilitar o entendimento do gerenciamento de políticas, o diagrama de sequência da Figura 21 ilustra os passos realizados até que uma violação seja detectada. Inicialmente acontece o filtro das políticas, para posterior registro no *PolicyEnforcer*. Nota-se a chamada assíncrona ao orquestrador de agentes. Essa, permite que uma nova *thread* seja iniciada para tratar a violação sem interrupção no gerenciamento.

3.4.6.5 *Definição de Políticas*

Considerando a classificação de políticas e o padrão IETF apresentados, e sabendo que uma política é definida por um conjunto de regras que são agrupadas de forma coerente, a *GiTo* propõe o uso das formas normais disjuntivas e conjuntivas para especificação das

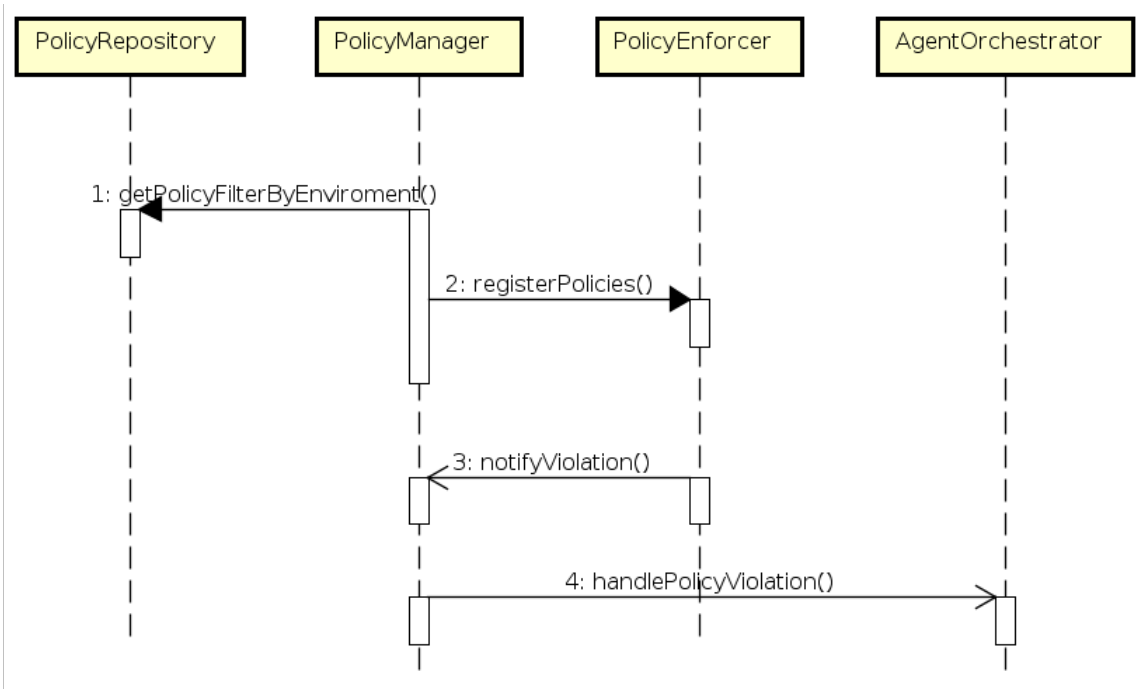


Figura 21 – Gerenciamento das Políticas - Diagrama de Sequência

políticas, onde:

- DNF (Disjunctive Normal Form) – as políticas simples são compostas através da operação lógica "E" e a união de seus grupos é realizada através da operação lógica "OU" (MOORE; ELLESSON, 2001).
- CNF (Conjunctive Normal Form) – as políticas simples são compostas através da operação lógica "OU" e a união de seus grupos é realizada através da operação lógica "E" (MOORE; ELLESSON, 2001).

A normalização é importante para facilitar integração com outras plataformas existentes, reduzindo assim possíveis problemas de interoperabilidade. Cada política deve conter conjunto de condições e um conjunto de ações, que estabelecem a semântica se <condição> então <ação>. Desta maneira, se as condições de uma regra forem satisfeitas, então as ações relacionadas a esta regra serão executadas. Para resolução de conflitos entre regras em uma política composta, se faz necessário o uso de prioridades.

3.4.7 Communication Manager

Quando um agente é iniciado, um serviço de descoberta de dispositivos é configurado. Para tal, o *Communication Manager* cria instâncias de servidores *multicast* que permitem que os agentes sejam encontrados quando outros realizarem uma busca. Quando um nó não pode processar mais os dados localmente, uma ação de descoberta na mesma camada do agente é ativada. Tão logo um novo dispositivo habilitado a receber os dados é encontrado,

um conexão é estabelecida e o *offloading* é iniciado. Caso contrário, o agente tentará realizar o *offloading* vertical.

Para viabilizar a operação de *offloading*, esse gerenciador possui dois subcomponentes: o *Agent Discovery*, responsável por descobrir quais outros agentes podem colaborar no processamento, e o *Offloading Server*. Esse segundo pode ser instanciando conforme demanda, para responder o *Agente Discovery* ou para receber os dados produzidos por outros agentes.

O *Communication Manager* ainda possui um terceiro subcomponente para suportar a mobilidade dos dispositivos diante de outros agentes e de servidores remotos, o *Handover Register*. Esse mantém o estado das conexões visando a continuidade do processamento quando, por exemplo, um agente sai da área de um servidor *Fog*, porém, se conecta com outra instância posteriormente.

3.5 Arquitetura vs. Requisitos

A Tabela 2 mostra a relação dos requisitos com os componentes apresentados na arquitetura.

	RF01	RF02	RF03	RF04	RF05	RF06	RF07	RF08
Data Consumer			X		X			
Context Manager	X							
Policy Manager		X						
Policy Enforcer		X						
Agent Orchestrator					X	X	X	
Network Manager						X	X	X
CEP Engine				X				

Tabela 2 – Mapeamento: Arquitetura x Requisitos

Os requisitos não-funcionais atuam de maneira ortogonal. Logo, todo sistema que implementa a arquitetura terá que garantir a conformidade com eles.

3.6 *GiTo*: Dinâmica de Execução

Refinando a arquitetura, analisando seus componentes e iniciando uma associação de operações e relacionamentos, tem-se a dinâmica de execução dos sistemas compatíveis com a arquitetura *GiTo*. Com base nos requisitos apresentados, e com intuito de facilitar o entendimento do relacionamento entre os componentes, esta seção apresenta dois casos de uso executados pelos agentes: (a) Monitorar uso das políticas e (b) Executar *offloading*.

No caso de uso (a), mostrado pelo diagrama de atividades da Figura 22, tem-se o monitoramento das políticas.

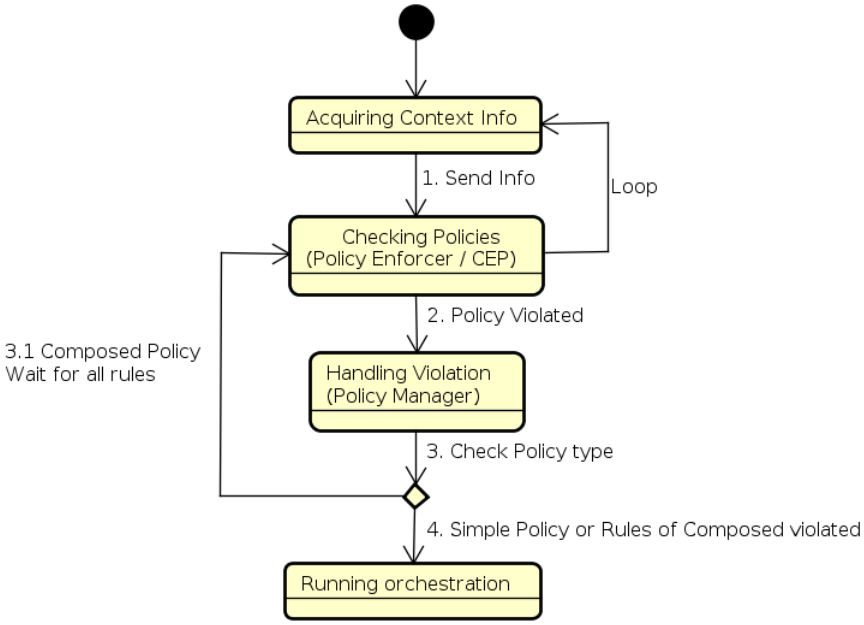


Figura 22 – Diagrama de Atividades - monitoramento das políticas

1. O sistema constantemente captura as informações de contexto, como consumo da Unidade Central de Processamento (CPU) e memória, latência de rede, tempo de resposta das notificações, etc. Essas informação são enviadas ao *Policy Enforcer*;
2. O *Policy Enforcer* recebe as informações de contexto e verifica as políticas definidas. Caso alguma regra de seja violada, o *Policy Manager* é notificado;
3. Quando uma regra é violada, o *manager* checa o tipo da política a qual a regra pertence. Uma notificação é enviada ao orquestrador de agentes quando a política é simples. Se composta, o *Manager* só enviará a notificação caso todas as regras sejam violadas;
4. O orquestrador de agentes inicia o processo de *offloading*.

Já o caso de uso (b), ilustrado pelo diagrama de estados da Figura 23, representa a continuidade do diagrama anterior. O orquestrador de agentes inicia o *handshaking* de *offloading*, executando a função de tomada de decisão nos agentes candidatos a colaborar no processamento.

O dispositivo A deseja realizar o *offloading* dos dados e processamento. O seu orquestrador verifica se existe um agente disponível na *Mist*. Existindo um novo agente, ele executa a função de tomada de decisão, e se estiver habilitado a receber os dados, o *offloading* é iniciado. Quando não há dispositivos, o servidor na *Fog* é conectado e uma análise do custo computacional do processamento (função de tomada de decisão) é realizada. Caso a

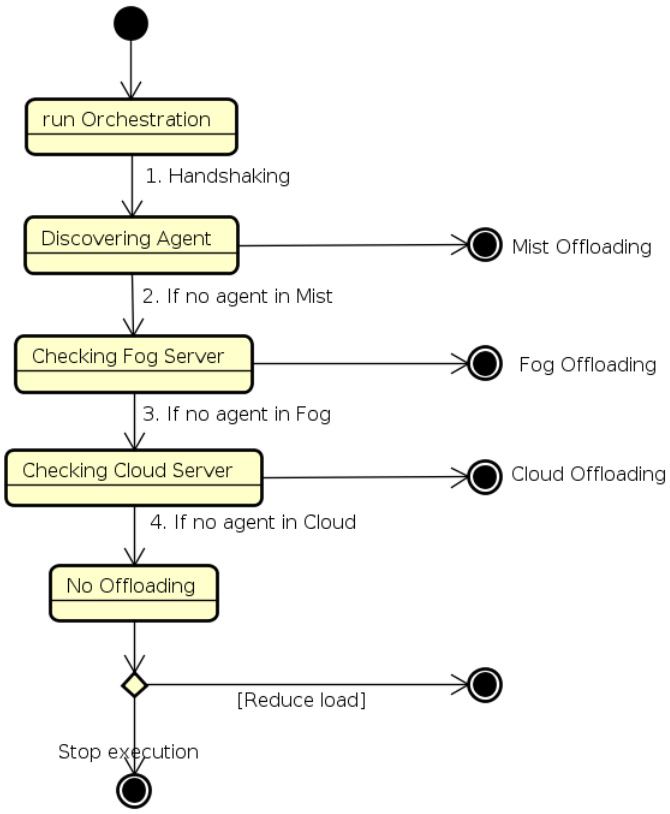


Figura 23 – Diagrama de Estados - Orquestração de agentes

Fog não tenha condições, o mesmo procedimento é realizado com um servidor na *Cloud*. Se este último também não estiver apto a colaborar, é recomendado que o dispositivo original pare o processamento ou reduza a frequência de coleta dos dados.

Quando o *offloading* é realizado, o agente da camada 1 (*mist*) envia os dados por ele produzido para um outro dispositivo. Porém, depois de um tempo, esse dispositivo pode ficar sobrecarregado e apresentar consumos de recursos elevados, prejudicando assim o tempo de resposta da análise dos dados. Nesse caso, ele sinaliza que não poderá mais continuar, e retorna o comando do fluxo para o agente gerador dos dados. Idealmente, uma nova busca por um agente apto à processar os seus dados é realizada, desconsiderando o dispositivo do primeiro *offloading*. Isso evita que o processamento fique sendo enviado e retornado entre duas mesmas máquinas continuamente.

3.7 Considerações Finais

A IoT e a WoT vem de fato causando uma revolução na TIC e nas vidas das pessoas. Com a quantidade de dispositivos conectados produzindo dados continuamente, cada vez mais é necessário utilizar contexto para orientar as tomadas de decisões e otimizar o processamento desses dados. Porém, para que o resultado do processamento seja considerado correto ela precisa chegar no tempo certo, considerando as necessidades das aplicações

e expectativa dos usuários que esperam a informação. A arquitetura *GiTo* propõe uma solução para melhorar o processamento desses dados através do uso de políticas. Todo sistema que implementa essa arquitetura é dito compatível com a *GiTo* e está habilitado a colaborar no processamento. A *GiTo* foi projetada para ser executada tanto na *Mist*, como na *Fog* e na *Cloud*. Para isso, um conjunto mínimo de componentes foram especificados, e eles podem ser ativados dinamicamente de acordo com o contexto que estão inseridos.

Os próximos passos seguem o planejamento apresentado neste capítulo. Uma vez definida a solução, é preciso validá-la através de uma implementação.

4 IMPLEMENTAÇÃO

Com a arquitetura especificada e refinada no capítulo anterior, a atividade seguinte prevista no planejamento consiste na sua implementação através de um Prova de Conceito. A *GiTo* considera diversos itens contextuais e dados de diferentes fontes. Porém, nesta Prova de Conceito, foram considerados dados de consumo de CPU e memória, tempo de resposta do processamento dos eventos complexos e latência de rede entre os agentes na *Mist* e os servidores remotos. Itens contextuais relacionados à eficiência energética (ex. consumo de bateria) e mobilidade não foram implementados neste protótipo.

4.1 Estratégia de Desenvolvimento e Linguagem de Programação

Neste etapa, foi desenvolvido um protótipo com base nos requisitos e nos componentes arquiteturais, com intuito de suportar diferentes fluxos de dados. Posteriormente, na fase de avaliação, uma aplicação específica será construída com base nesta PoC.

Com conhecimento prévio sobre as plataformas de processamento de dados em tempo real adquirido na fase de Análise do Domínio (Seção 3.2), a estratégia inicial para a implementação do *GiTo* era estender o *Apache Storm*, para posterior integração com o engenho CEP *Esper*. Porém, em testes preliminares utilizando um computador *Raspberry Pi*[1], detectou-se que o *Storm* requer um alto poder computacional, não sendo recomendado para classes mais baixas do grupo J na classificação do IETF (BORMANN M. ERSUE, 2018). De fato, o *Storm* utiliza módulos específicos do *JavaEE* (ORACLE, 2018), tornado-o dependente desta plataforma e exigindo mais recursos.

O engenho *Esper* também foi testado através da execução de aplicativos e apresentou baixo consumo dos recursos de memória e CPU. Assim, optou-se por implementar os componentes da arquitetura como um novo sistema, reutilizando apenas *Esper* e integrando-o como o engenho CEP na arquitetura *GiTo*. Por esse motivo, *Java* foi a linguagem escolhida para a criação do protótipo.

4.2 Análise dos Dados

Como mencionado na seção anterior, o engenho CEP escolhido para análise dos dados foi o *Esper*. Este é bastante difundido e é um componente *open-source*, de fácil integração e disponível sob a licença GNU GPL. *Esper* suporta uma linguagem declarativa para processamento de eventos (*EPL*) similar ao *SQL* padrão. Através da *EPL*, é possível definir regras para detectar padrões em um fluxo contínuo de dados.

[1]

Quando iniciado, o *Esper* é configurado com os tipos dos dados que serão analisados e o número de *threads* de processamento, que neste *PoC* sempre é igual ao número de processadores (*cores*) existentes no dispositivo *GiTo*. O *Esper* foi integrado como uma classe de nome *CEP* com funções de inicialização e finalização externalizadas, facilitando assim o seu uso por outros componentes. Também é possível adicionar um novo tipo de dado durante a execução do engenho, o que permite a realização do *offloading* para agentes *GiTo* sem a necessidade prévia do conhecimento semântico do evento recebido.

4.3 Políticas e Informações de Contexto

Dos quatro elementos de política do IETF, a arquitetura *GiTo* apresenta os mais importantes para a execução: PEP e PDP. Nesta implementação, os componentes *PMT* e *PR* são representados por um arquivo *XML*, onde são especificadas as políticas e suas regras. As regras de políticas suportadas pelo protótipo são apresentadas na Tabela 3 e um exemplo de declaração contendo políticas simples e composta é mostrado na Figura 24.

Descrição	Unidade
Consumo máximo de CPU	%
Consumo máximo de memória	% e MB
Tempo de resposta do CEP médio e máximo	milissegundos
Round Trip Time médio e máximo (da aplicação)	milissegundos
Latência de rede média e máxima	milissegundos

Tabela 3 – Políticas suportadas no PoC

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <policies type="DNF">
4      <policy type="simple" name="max_memory" value="0.8" action="offloading"/>
5      <policy type="simple" name="max_cep_response_time" value="5" action="offloading"/
6
7      <policy type="composed" action="offloading">
8        <rule name="max_cpu" value="0.6"/>
9        <rule name="max_memory" value="0.6"/>
10     </policy>
11 </policies>
```

Figura 24 – Políticas declaradas como *Extensible Markup LanguageXML* (XML)

Nesse exemplo é declarado um conjunto de políticas na forma normal disjuntiva (linha 3), onde duas políticas simples são especificadas, uma relacionada ao percentual do consumo máximo de memória e a outra ao tempo de retorno do engenho CEP. Em seguida, na linha 7, uma política composta contendo regras do consumo máximo de CPU e memória é especificada. É possível combinar qualquer regra da Tabela 3 na declaração de uma

política composta. Se a condição de qualquer política do exemplo for alcançada, uma ação de *offloading* é realizada, uma vez que todas elas declaram essa operação no campo *action*.

O PEP poderia ser representado pelo componente *CEP engine*. Porém, como este trabalho propõe analisar a execução do CEP, o monitoramento das políticas não será implementado com a utilização do engenho, com intuito de não influenciar nos resultados obtidos. Assim, para representar o PEP, foi desenvolvida a classe *Policy Enforcer*. Essa recebe informações de contexto do componente *Context Manager* e verifica as políticas em tempo de execução. A Figura 25 mostra o trecho de código que notifica os componentes internos, que nesta PoC é o Orquestrador de agentes. Todas as notificações são enviadas de maneira assíncrona, ou seja, uma nova *thread* é disparada para realizar a notificação, enquanto o monitoramento continua ativo.

```
105  private void checkIfPolicyIsViolated(Policy p, float policyValue, float systemValue) {
106      if (p != null) {
107          if (systemValue > policyValue) {
108              if (p.isSimple()) {
109                  notifyListeners(null, ContextListener.STATUS_POLICY_ELAPSED);
110              }
111          }
112      }
113  }
```

Figura 25 – *Policy Enforcer*: checagem de políticas

Para a aquisição de informações dos itens de *hardware* do ambiente de execução, foi utilizada a *Application Programming Interface* (API) *Sigar*[2], que coleta informações do consumo de CPU e memória. Já as informações relacionadas ao processamento dos dados, foi preciso coletá-las a partir das ações executadas pelos outros componentes da arquitetura, como número de dados coletados pelo agente (*DataReader*) e número de eventos processados pelo *CEP Engine*. A latência de comunicação e de rede são fornecidas pelo *Communication Manager*. Para viabilizar a execução do *Sigar* nos dispositivos *Raspberry PI*, foi preciso compilar a biblioteca e gerar um *.so* para a plataforma *ARM*.

O *Esper* contém um módulo para publicação de métricas relacionadas ao seu processamento. Porém, assim como no monitoramento das políticas, optou-se por não utilizar esse recurso por considerá-lo intrusivo, podendo impactar nos resultados. Nesse caso, para ter acesso ao número real de eventos analisados, o código do *Esper* foi alterado, introduzindo um contador (inteiro) e o incrementando sempre que um evento simples fosse retirado do *buffer* interno do engenho para ser processado.

Todas as métricas são consolidadas no *Context Manager-Metrics* e direcionadas para o *Policy Enforcer*, que as repassa para o *PolicyManager*. Quando este último detecta uma violação, o orquestrador de agentes é ativado.

[2] https://github.com/hyperic/sigar

4.4 Coleta de Dados

O *Data Consumer* foi desenvolvido de forma que seja possível parametrizar sua execução. A classe implementada foi chamada de *DataReader*, e pode ser iniciada para trabalhar com diferentes quantidade de *threads*, o que permite uma coleta paralela dos dados. O intervalo de leitura dos dados também pode ser parametrizado, viabilizando alterar a taxa de transferência dos eventos em tempo de execução.

O dado é enviado para uma *thread* despachante, o que permite que o procedimento de coleta possa ler outro dado imediatamente. O *Data Reader* suporta leitura de dados de diferentes fontes (ex. sensores e rede) em paralelo. Para cada *source* estabelecido, existe uma *thread* despachante separada. Esse paralelismo é essencial para os sistemas CEP.

4.5 Orquestrador de Agentes e *Offloading*

O orquestrador de agente é fundamental na arquitetura *GiTo*. Através dele, os agentes se comunicam e colaboram entre si por meio do *offloading* computacional. O orquestrador funciona de forma distribuída, e é responsável por definir um plano de execução para o processamento com base nas políticas definidas.

Apesar da arquitetura *GiTo* suportar o *offloading* horizontal e vertical, nesta PoC, apenas a segunda abordagem foi implementada, uma vez que a ideia era analisar a sua viabilidade em um dos cenários e este precisava de uma configuração mais simples. Porém, uma vez a conexão estabelecida, a estratégia é similar, já que todos os agentes utilizam a função de tomada de decisão para decidir se deve ou não aceitar a solicitação de *offloading*.

O orquestrador é implementado como uma máquina de estado seguindo o padrão de projeto *State* (GAMMA, 1995). Sempre que um alerta de política violada é recebido, uma ação é realizada com o objetivo de manter o sistema consistente. A Figura 26 apresenta os estados internos utilizados. Quando um agente inicia a busca por outros para realização do *offloading*, o sistema entra no estado *STATE_REMOTE_AS_CLIENT_REQUEST*. Quando o *offloading* é confirmado, o estado *STATE_REMOTE_AS_CLIENT_OK* é alcançado.

```
3  public interface GitoState {
4
5      public static final int STATE_IDLE                    = 0;
6      public static final int STATE_LOCAL                   = 1;
7      public static final int STATE_REMOTE_AS_CLIENT_REQUEST = 2;
8      public static final int STATE_REMOTE_AS_CLIENT_OK      = 3;
9      public static final int STATE_REMOTE_AS_SERVER         = 4;
10     public static final int STATE_END_PROCESSING           = 6;
11     public static final int STATE_RESET                    = 7;
12 }
```

Figura 26 – Estados Internos do *GiTo*

Um determinado agente pode processar dados que ele produz, ou proveniente de um outro dispositivo. Quando o orquestrador de agentes recebe um evento indicando que uma política foi violada, ele pode executar duas possíveis ações: (1) realizar o *offloading* do processamento para outro nó ou (2) interromper o processo de *offloading* proveniente de outro agente.

Na ação (1), o agente produtor dos dados estabelece uma conexão com um outro agente, de acordo com os passos abaixo:

1. Envia uma mensagem *User Datagram Protocol* (UDP) em *multicast* para a camada da Mist; Caso algum outro agente esteja habilitado a colaborar, a conexão é estabelecida (esse passo não foi implementado nessa PoC);
2. Se a conexão não foi criada no passo anterior, o agente atual estabelece uma conexão TCP com o servidor da *Fog* através de um canal de controle (diferente do canal de dados). Caso o servidor possa atender à demanda, o *offloading* é realizado criando uma nova conexão de dados;
3. Se o *offlloading* não tiver sido realizado nem na *Mist* nem na *Fog*, o procedimento do passo 2 é executado em direção à *Cloud*.

Em todos os passos, o agente atual envia ao nó candidato as informações sobre a frequência dos dados, o tipo de consulta, a janela de tempo utilizada e o tempo de resposta máximo. Ao receber as informações, o agente candidato executa a função de tomada de decisão e retorna a informação se pode ou não colaborar no processamento. Quando o *offloading* é confirmado, o orquestrador de agentes do dispositivo produtor dos dados executa a reconfiguração do sistema, através das mudanças dos estados internos e carregamento das novas informações. A Figura 27 ilustra o trecho de código desta parte, onde o *DataConsumer* tem seu modo de execução alterado e adaptado. Os demais componentes tem suas execuções com base nos estados do sistema.

```
59
60      Metrics.EXECUTION_TYPE = Constants.TYPE_EXECUTION_CLIENT;
61      Metrics.EXECUTION_REMOTE_IP = remoteIp;
62
63
64      GitoProperties.getInstance().reloadProperties();
65      CEP.getInstance().stopCep();
66      GeneralDataReader[] readers = GitoProcess.getInstance().getDataReaderArray();
67      for (GeneralDataReader r : readers) {
68          r.reConfigure(remoteIp);
69      }
70
71      StateMachine.getInstance().changeState(GitoState.STATE_REMOTE_AS_CLIENT_OK, null);
72      control.setCurrentOffLoadingType(OffLoadingListener.MSG_TYPE_PROCESSING_REMOTE);
73
```

Figura 27 – Orquestrador de Agentes: reconfiguração dos componentes

No caso da ação (2), o *offloading* já está em execução, e o agente detecta que não pode continuar o procedimento. Neste momento, um comando de controle é enviado ao

dispositivo produtor dos dados alertando sobre a interrupção, e as conexões de controle e de dados são finalizadas. O orquestrador do agente produtor dos dados analisa o seu contexto, e caso ainda esteja sobrecarregado, a ação (1) é realizada novamente.

4.5.1 Função de Tomada de decisão

A arquitetura *GiTo* define uma função que é executada sempre que um agente recebe a solicitação de *offloading*. Essa função deve responder ao agente que deseja compartilhar seus dados se o dispositivo atual deve ou não colaborar no processamento. A implementação da função de tomada de decisão é dependente de plataforma.

Para a PoC, esta função foi simplificada e desenvolvida conforme o algoritmo abaixo.

```
1
2  dataThroughput = (numberOfEvents / numberOfConn)
3  contextDevice = dataThroughput != HIGH &&  queryComplexity != HIGH &&
       CPU <= 50
4  responseTimeOk = (currentResponseTime <= remoteResponseTime)
5
6  IF (contextDevice && responseTimeOk)
7          return true;
8  ELSE
9          return false;
10 ENDIF
```

Listing 4.1 – Função de Tomada de Decisão

A função verifica se o *data throughput* por conexão e a complexidade da consulta possuem valores diferentes de *High*. Em caso positivo, analisa se o consumo de CPU é menor que 50% e a política de tempo de resposta do agente solicitante é menor que o seu valor atual. Se todas as condições forem aceitas, o *offloading* é permitido.

4.6 Conectividade

O gerenciador de comunicação é responsável por tratar as conexões de rede e controlar os servidores ativos.

4.6.1 Protocolos de Comunicação

A WoT é considerada como um refinamento da IoT através da integração de coisas inteligentes não apenas à camada da Internet (infraestrutura de rede), mas também à Web (camada de aplicação) (GUINARD; TRIFA; WILDE, 2010). Assim, para colocar os agentes

de processamento *GiTo* na WoT, e permitir que esses possam se integrar com servidores remotos através de caminhos já conhecidos (TRAN et al., 2017), o protótipo desenvolvido utiliza os protocolos *WebSocket* e *HTTP* para o *offloading* do processamento.

Quando um dispositivo produz dados numa alta frequência, com taxa de produção menor que 50 eventos por segundo, é recomendado utilizar um canal bi-direcional orientado à conexão, como *WebSocket*, reduzindo assim o custo operacional com criação de novas conexões constantemente (PIMENTEL; NICKERSON, 2012)(KARAGIANNIS et al., 2015). Quando a frequência de produção de dados é baixa, o protocolo *HTTP* pode ser usado.

A latência de comunicação é calculada através dos *frames* de controle *ping-pong* implementados pelo *Websocket* (FETTE; MELNIKOV, 2016), como apresentado na Figura 28. Já a latência de rede, foi coletada através da ferramenta *Ping*, que utiliza o protocolo *Internet Control Message Protocol* (ICMP)[3].

```
70@Override
71 public void onWebsocketPong( WebSocket conn, Framedata f ) {
72     if(f.getPayloadData().array().length > 0) {
73         long pingTimeStamp = f.getPayloadData().getLong();
74         long currentTime = System.currentTimeMillis();
75
76         Log.log(true, "Ping-Pong - Diff_Time = " + (currentTime - pingTimeStamp));
77     }
78 }
```

Figura 28 – Utilização dos *Frames Ping-pong* do *Websocket*

4.6.2 Servidores

Para suportar o *offloading* vertical, foram implementados nos agentes *GiTo* em três servidores distintos:

1. Um servidor *HTTP* para recebimento dos dados;
2. Um servidor *WebSocket*, também para troca de dados;
3. Um servidor TCP como canal de controle, sendo responsável por receber e enviar comandos relacionados ao processo de *offloading*.

Como mencionado, o *offloading* horizontal não foi desenvolvido nesta PoC. Mas, idealmente, todo agente *GiTo* que esteja participando da colaboração na camada 1, deve ter pelo menos um servidor *Multicast* iniciado. Este permite que os agentes sejam encontrados quando o *offloading* horizontal é realizado.

O servidor *HTTP* apenas recebe dados e os direcionada para o componente CEP, enquanto que o servidor *WebSocket* se comporta como um canal bi-direcional. Logo, neste segundo, existem dois tipos de conexão: *publish* e *subscribe*.

[3] https://tools.ietf.org/html/rfc792

Os servidores de dados são ativados ou desativados através de um arquivo de configuração, e o canal de controle criado quando o PoC é iniciado no modo servidor.

4.6.3 Publish / Subscribe

Os dispositivos da IoT possuem mobilidade que tornam suas conexões intermitentes em diversas situações, ora por instabilidade de rede, ora por tentativas de economizar seus recursos. Além disso, possibilitam um fraco acoplamento entre as entidades comunicantes (GOMES et al., 2018). Logo, seguindo a arquitetura genérica proposta por *Cugola* e descrita na Seção 8, a qual enfatiza a técnica de CEP como uma extensão do modelo *Publish / Subscribe*, esta implementação possui três modos de conexão para os clientes. Um dispositivo ao estabelecer uma conexão com um servidor de dados, ele pode se identificar de três maneiras:

1. *Publisher*: agente que apenas produz dados e os envia ao CEP. Não recebe notificações;
2. *Subscriber*: agente se registra para receber notificações do processamento;
3. *Publisher_Subscriber*: o agente tanto produz e compartilha seus dados, como também recebe notificações dos eventos complexos gerados.

Quando uma conexão em modo *Subscriber* e *Publisher_Subscriber* é estabelecida, o dispositivo deve informar seu identificador para que possa receber notificações direcionadas à ele. Quando o *id* não é especificado, um agente *subscriber* recebe notificações de todos os agentes.

5 EXPERIMENTAÇÃO

Este capítulo apresenta a metodologia utilizada no experimento que avaliou a arquitetura *GiTo*, conforme planejamento apresentado nos capítulos anteriores. O intuito é usar o protótipo implementado e exercitar os componentes quando fluxos de diferentes tipos de dados, e também construir uma aplicação com base em dados reais.

A estratégia da experimentação consiste em duas etapas. Na primeira, foi realizada uma análise de desempenho da Prova de Conceito através da aplicação cenário nas camadas da *Mist*, *Fog* e *Cloud*, executando-a separadamente. O objetivo desta etapa é testar a implementação da arquitetura e avaliar a execução dos componentes da arquitetura assim como seus impactos no consumo dos recursos. Na segunda etapa, a aplicação foi executada e analisada integrando as três camadas.

Neste experimento, a *Cloud* será utilizada como um servidor remoto, e suas funcionalidades de escalabilidade, elasticidade entre outras não serão exploradas, uma vez que se deseja observar os limites de cada dispositivos, para na etapa 2 otimizar o processamento sem recursos adicionais.

5.1 Etapa 1: Avaliação de Desempenho

Esta seção detalha a análise de desempenho realizada, descrevendo a metodologia, ferramentas de avaliação adotadas e seu resultado.

De acordo com (JAIN, 1990), uma análise de desempenho nunca é igual a outra. As métricas, técnicas de avaliação e outras características empregadas em um estudo dificilmente serão reutilizadas em outra análise. Contudo, existem etapas em comum à todas elas. Os passos sugeridos por Jain serviram de guia para esse trabalho. A lista abaixo enumera os passos adotados.

1. Definir objetivos e escopo da análise
2. Identificar os serviços oferecidos pelo componente a ser avaliado e seus possíveis resultados
3. Selecionar métricas
4. Selecionar os *workloads*
5. Selecionar os fatores
6. Projetar o experimento
7. Analisar e interpretar dos dados
8. Apresentar os resultados

Cada passo foi instanciado para o experimento deste trabalho conforme seções a seguir.

5.1.1 Objetivo do Experimento

O objetivo deste experimento é analisar o custo de desempenho do Processamento de Eventos Complexos a partir dos dados produzidos pelos agente *GiTo*, considerando três modos de execução diferentes: processamento local (camada *Mist*), *offloading* na *Fog* e *offloading* na *Cloud* separadamente. A Figura 29 ilustra essas três possibilidades. Nesta etapa, o componente textitAgent Orquestration não é exercitado, sendo ele desativado durante todas as repetições. Com isso, é possível verificar os limites de processamentos de cada camada, com intuito de melhor definir futuras políticas de coordenação.

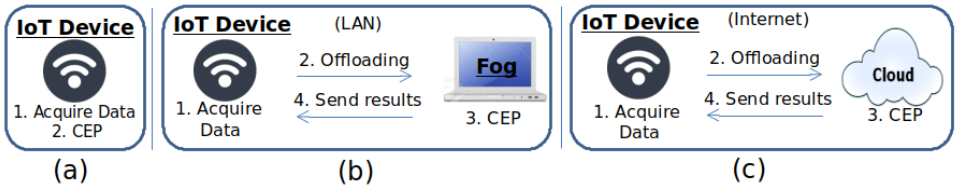


Figura 29 – CEP no (a) próprio dispositivo, (b) *Fog* e (c) *Cloud*

As agentes produzem dados criando um fluxo contínuo de informação que chegará até o engenho CEP. No modo de execução (a), esse fluxo é analisado localmente e os eventos complexos detectados são notificados à uma aplicação ativa no próprio agente. Em (b) e (c), os dados são enviados para um servidor na *Fog* e na *Cloud*, respectivamente, que os processa e retorna os eventos encontrados. No modo (b), o dispositivo e o servidor na *Fog* estão em uma mesma LAN, enquanto no (c) o servidor está localizado na Internet.

Entender o custo operacional do agente *GiTo*, independente de sua localização e em diferentes cenários, é importante para identificar os limites dos dispositivos em cada camada separadamente, o que permite uma melhor definição dos valores das políticas na arquitetura *GiTo*. Como visto anteriormente (Seção 4), toda a comunicação entre os dispositivos e os servidores é realizada através de protocolos já difundidos na Web, contextualizando os agentes envolvidos na WoT.

No restante deste artigo, o termo *modo de execução* está relacionado às execuções na *Mist*, *Fog* e *Cloud*.

5.1.2 Aplicação Cenário e seus Serviços

Para a análise de desempenho foi desenvolvida uma aplicação com base em um cenário que distribui promoções aos passageiros de táxis. A aplicação recebe dados contendo as localizações atuais dos táxis, que nesse contexto, são representados pelos agentes *GiTo*, e periodicamente as compara com promoções geo-localizadas emitidas por empresas par-

ceiras. Caso a promoção seja direcionada para uma localização próxima de um táxi, o seu passageiro receberá um desconto na corrida e o motorista uma bonificação.

Como serviço principal, esse cenário entrega promoções. Porém, o que deseja avaliar é quando e como essas promoções chegam aos passageiros, analisando o processamento dos dados em tempo real. Por exemplo, uma promoção só será eficaz se o evento complexo for recebido quando o passageiro ainda estiver dentro do táxi, caso contrário ele não será beneficiado.

Esse cenário utilizará dados provenientes de sensores de localização (GPS), e também dados de negócios criados para o experimentos (as promoções). Apesar de não trabalhar diretamente com fluxos provenientes das *things* da *Internet of Things*, pode-se através deste cenário exercitar a arquitetura da mesma maneira, através de fluxos provenientes de operações de entrada e saída, ora local, ora da rede. O cenário permite testar a arquitetura nas três camadas. No caso da *Mist*, promoções são enviadas ao táxi que irá processá-las em conjunto com seus próprios dados. Nos casos do *offloading* para a *Fog* e a *Cloud*, os táxis enviam suas localizações para os servidores remotos processarem juntamente com as promoções recebidas dos parceiros.

5.1.3 Métricas Avaliadas

Quando um sistema executa seu serviço corretamente, seu desempenho é calculado pelo tempo que levou para efetuar as operações, a frequência a qual o serviço é realizado e o consumo dos recursos associados às funcionalidades desempenhadas. Essas três métricas referentes à *tempo-frequência-recurso* são também chamadas de capacidade de resposta, produtividade e utilização. Além dessas relacionadas ao sucesso do serviço, existe também aquelas de erro e outras quando o serviço não é executado ou é interrompido inesperadamente após seu início. Essas três categorias de métricas são conhecidas como velocidade, confiabilidade e disponibilidade, respectivamente.

Quando a execução de um engenho CEP leva a um erro, esse geralmente está relacionado à inconsistência na sua configuração inicial, como regras mal formadas, tipo de dado inexistente ou a não detecção de um evento quando ele deveria ser notificado (ou o contrário). Como neste experimento as regras e os dados são especificados antes da execução, e o objetivo não é analisar a corretude do engenho, mas sim a sua capacidade de resposta em condições diversas, métricas de erro não serão especificadas. Uma premissa da análise é que o engenho *Esper* é um componente caixa preta. Assim, para avaliar a execução do processamento de dados nos agentes *GiTo* independente de qual camada ele estará presente, definiram-se as seguintes métricas:

- Consumo médio de CPU (em % e MHz);
- Consumo médio de memória RAM (em MBytes);
- Número de dados (eventos simples) enviados ao engenho CEP;

- Número de dados (eventos simples) processados pelo engenho CEP;
- Número de eventos compostos notificados pelo engenho CEP (*matches*);
- Tempo de resposta (TR) médio e máximo entre a publicação de um evento e o recebimento da notificação disparada por este mesmo evento;
- Latência de rede média - *Round Trip Time* (RTT) associado à camada de rede;
- Número de execuções iniciadas e não finalizadas com sucesso (serviço interrompido).

Com essas métricas coletadas é possível analisar em que condições o CEP está sendo realizado, considerando os agentes e o ambiente que ele estão inseridos.

5.1.4 Carga de Trabalho

O termo *carga de trabalho de teste* é genérico e corresponde a qualquer carga utilizada em estudos relacionados à performance. De acordo com (JAIN, 1990), uma carga de teste pode ser real ou sintética. A primeira observada quando tem-se um sistema executando em operações normais ou em produção, não pode ser reproduzida, e geralmente, não é adequada para testes. Já a carga de trabalho sintética, cujas características são similares às reais, podem ser aplicadas repetidas vezes em um ambiente controlado, e por isso é mais indicada para experimentos. A principal razão é que a sintética é uma representação da carga real, além de ser facilmente alterada (sem impactos operacionais) para ter características específicas incorporadas que facilitem sua medição.

Com intuito de atender a aplicação cenário e se aproximar de um experimento real, a análise de desempenho deste trabalho utilizou uma carga de trabalho sintética, a qual usa um conjunto de dados derivado de um projeto realizado pela *Microsoft Research*[1].

5.1.4.1 *Massa de Dados*

O experimento coletou dados de táxis na cidade de Pequim (China) durante um período de 1 semana (7 dias). A amostra de dados resultante, conhecida como *T-Drive trajectory data sample*, contém trajetórias de 10537 táxis compartilhados nesse período. O número total de pontos de localização é em torno de 15 milhões e a distância total percorrida pelos táxis nesse período alcança 9 milhões de quilômetros (YUAN et al., 2011) (YUAN et al., 2010). O intervalo de amostragem médio é em torno de 177 segundos, com uma distância de 623 metros. O registro abaixo é um exemplo dos dados publicados por um táxi.

1,2008-02-02 20:30:34,116.49625,39.9146

[1] https://www.microsoft.com/en-us/research/publication/t-drive-trajectory-data-sample/

Um registro possui quatro campos separados por vírgula, cada um contendo o identificador do táxi, data e hora (*timestamp*) do momento em que o dado foi reportado, assim como a sua localização representada pelas coordenadas geográficas de longitude e latitude.

A partir dessa amostra, foi criado um fluxo contínuo com informações dos táxis e deste foi gerado um novo conjunto de informações originando as promoções geo-localizadas. Cada promoção possui as coordenadas de longitude e latitude extraída aleatoriamente a partir dos registros original da *Microsoft*. Logo, o experimento considera dois tipos de dados, que serão injetados no engenho CEP para serem processados. Nesse contexto, os táxis representam os agentes *GiTo*, consequentemente, as coisas da *Web of Things*.

5.1.4.2 *Consulta CEP*

Os dados alcançam o engenho CEP através de diferentes fluxos de dados criados pelos táxis e pelo conjunto de promoções. Porém, o custo operacional da análise destes dados é influenciado diretamente pelas regras CEP que especificam as relações verificadas entre os eventos simples. Quando uma expressão analisada pelo engenho detecta um evento complexo, a aplicação é notificada. Neste experimento, as consultas foram especificadas para atender a aplicação cenário.

O engenho *Esper*, utilizado na implementação da arquitetura (Seção 4.1), permite a declaração de vários tipos de regras com diferentes complexidades.

Este experimento considera dois tipos de consulta que foram analisadas separadamente. A primeira, de baixa complexidade, envolve operações de filtragem através de comparação simples de valores. Já a segunda regra, de complexidade alta, utiliza o conceito de padrões (*pattern*) e uma função definida pelo usuário. Assim, pode-se exercitar o engenho em situações de carga distintas. Importante mencionar que, se um dispositivo consegue processar dados através de um regra que utiliza padrões, é possível executar regras do tipo filtro, agregação entre outras mais simples (SCHILLING et al., 2010).

Considerando os dois fluxos de dados (táxi e promoção) pode-se especificar as regras CEP.

- Consulta de baixa complexidade

  A consulta CEP contendo a operação de filtro verifica se, para toda promoção geo-referenciada recebida, existem táxis próximos de sua localização. A distância especificada é de 500 metros, logo, se um táxi estiver nessa distância ou mais perto da coordenada geográfica associada à promoção, um *match* é identificado, e um evento complexo gerado. Em seguida, a aplicação receberá uma notificação. A regra EPL para essa consulta é mostrada na Listagem 5.1. A função *distance* é dita como *user-defined function*, e é definida fora do engenho como uma classe registrada no *Esper*, no momento da sua configuração.

```
1 select * from
```

```
2     Taxi.std:lastevent() as t,
3     Promotion.std:lastevent() as p
4 where
5     distance(t.latitude, t.longitude, p.latitude, p.longitude) < 500
```

Listing 5.1 – Regra EPL - utilização de filtro

Como o número de eventos de promoção é menor que o número de dados compartilhados pelos táxis, essa regra inicia uma expressão sempre que uma promoção é recebida, e finaliza a atual, deixando apenas uma única ativa. Toda vez que um evento de táxi é enviado ao CEP, a expressão é analisada comparando as informações dos dois fluxos.

- Consulta de Alto complexidade
  A segunda regra utiliza padrões, janela de tempo e também a função *distance*. A consulta seleciona os táxis que estejam a 500 metros ou menos da coordenada associada à promoção num intervalo de X segundos a partir da publicação do táxi, onde X representa o tamanho da janela de tempo da consulta. A consulta é injetada no engenho CEP para que seja possível avaliar seu comportamento à medida que os dados são recebidos. A regra EPL com alta complexidade é apresentada na listagem 5.2.

```
1 select * from pattern
2     [every t=Taxi -> (every p=Promotion
3     (distance(latitude, longitude, t.latitude, t.longitude)<500))
4 where
5     timer:within(X seconds)]
```

Listing 5.2 – Regra EPL - utilização de Padrões

Nesta regra, o atributo *every* é utilizado e determina que para todo evento de táxi recebido, uma expressão é instanciada e permanece ativa até que uma promoção validada como da mesma área seja encontrada, ou quando ela for removida pela janela de tempo. Ou seja, durante a execução, várias expressões ficam ativas ao mesmo tempo, e são avaliadas à medida que os dados são recebidos pelo engenho CEP.

5.1.4.3 *Fatores e Níveis*

Dentre os parâmetros que influenciam o desempenho do sistema, alguns são fixos, e outros são variáveis. Parâmetros de *hardware*, por exemplo, uma vez definidos, não têm sua configuração alterada (discutido na Seção 5.1.5.6). Já aqueles parâmetros que variam durante o experimento são chamados de *fatores*, e seus possíveis valores de *níveis*. A Tabela 4 mostra os fatores e níveis considerados nesta avaliação.

O modo de execução representa a camada onde cada execução é realizada. Como visto na seção anterior, a janela de tempo representa o intervalo em segundos que o engenho CEP vai manter cada evento recebido até um *match* acontecer. Caso o tempo expire, o

Tabela 4 – Fatores e níveis do experimento

Fatores	**Níveis**
Modo de Execução	*Mist*, *Fog* e *Cloud*
Complexidade da Consulta	Filtro e Padrão
Intervalo de Coleta dos dados (ms)	1, 2, 5, 10, 20, 50 e *timestamp* do registro
Tamanho da Janela de Tempo (em segundos)	120, 300, e 600
Número de fluxos/dispositivos (táxis separados)	1, 4, e 8

evento é descartado (ESPER, 2016). Os níveis escolhidos para esse fator foram baseados em valores reais para a aplicação cenário.

A taxa de transferência representa o intervalo de coleta dos dados que cada agente *GiTo* irá utilizar. Os cenários utilizando o *timestamp* original do registro são chamados de execuções referências (seção 5.1.5) que, mesmo sendo executado em um ambiente controlado, tende a demonstrar um comportamento próximo do real. Os resultados gerados por esses cenários servirão de base comparativa para a análise final. Quando o *timestamp* do registro não é utilizado, a leitura dos dados usará os intervalos especificados para o fator *Taxa de transferência dos dados*. O valor de *1ms* é o menor valor possível para o intervalo de coleta, e os demais foram especificados respeitando uma variação de tempo que possibilite observar mudanças incrementais nas métricas.

O número de fluxo de dados está associado à quantidade de dispositivos que irão participar da execução. Cada um destes dispositivos representa um táxi que propagará um fluxo de dados separado dos demais, contendo apenas os dados por ele produzido. Além destes fluxos independentes, existirá um outro publicando dados de todos os demais táxis da região. Com diferentes taxas de transferência e variando o número de fluxos, é possível simular diferentes rajadas de dados. A escolha dos níveis para esse fator levou em consideração a limitação de *hardware* disponível para o experimento. A complexidade da consulta já foi detalhada (Seção 5.1.4.2), e também é apresenta como um fator.

Utilizou-se a notação *(Modo_execução, complexidade_consulta, taxa_transferência, Tamanho_Janela, num_fluxos)* para representar os cenários de testes. Por exemplo, *(Mist, Pattern, 1ms, 600s, 4)* se refere ao cenário executado no próprio dispositivo, executando a consulta do tipo *Pattern* com janela de tempo de 600s, com intervalo de coleta de dados de 1ms e tendo 4 fluxos publicando dados.

Importante registrar que, o tipo e a classificação do tráfego de rede durante a execução do experimento, é um item transparente para esta pesquisa. Os dados são enviados por um canal *socket* à medida que eles são produzidos, através de *streams* de rede padrões na linguagem de programação utilizada. Os fluxos das mensagens são definidas na Seção 5.1.5.

5.1.5 Design do Experimento

Uma vez as métricas selecionadas e a carga de trabalho definida, pode-se projetar a execução do experimento. A Tabela 4 apresenta os fatores e níveis, que combinados levam a um conjunto de 252 cenários de testes (excluindo os cenários referência), onde cada um foi repetido 30 vezes, tendo cada execução uma duração de 30 minutos.

As execuções referência (*timestamp* original do registro) foram configuradas com os níveis que, teoricamente, mais sobrecarregam o processamento: complexidade padrão, 8 fluxos de dados de táxis separados e janela de tempo de 600 segundos, variando apenas o modo de execução. Assim, obteve-se um resultado simulando um cenário real, que posteriormente foi comparado aos demais. Cada execução referência teve uma duração de 12 horas e foi repetida 10 vezes. Com essa duração é possível observar possíveis falhas no sistema ao longo de horas de processamento, como por exemplo, erro de *memory leak*.

Além dos cenários de testes mencionados, foram executados outros dois: um processando os dados no engenho CEP sem a utilização do *GiTo*, apenas coletando os dados e enviando-os para análise do CEP; o outro exercitando os componentes da arquitetura sem de fato processar os dados, apenas simulando o envio ao engenho que neste cenário foi desativado. A adição destes cenários auxilia a análise dos experimentos, mostrando a influência dos componentes arquiteturais nos resultados obtidos.

5.1.5.1 *Clusterização dos dados*

Com intuito de selecionar os registros para cada modo de execução planejado, a amostra de dados passou por um processo de *clusterização*.

Como mencionado na Seção 5.1.4, os dados possuem registros de 10537 táxis por um período de 7 dias. Em uma análise detalhada dos arquivos, percebeu-se que o intervalo entre duas publicações para um mesmo táxi não é regular. Com intuito de melhor exercitar o processamento na *Mist*, *Fog* e *Cloud*, foi realizada uma classificação dos dados agrupando-os por dia/hora e geo-localização. Esse agrupamento identificou os táxis com maior número de registros e as áreas com maior densidade de publicações. O processo seguiu as seguintes etapas:

- Filtrar os horários em sequência com maior número de registros;
- Selecionar a área de Pequim com maior número de táxis que publicaram os dados;
- Identificar os táxis que mais compartilharam suas localizações.

Para a clusterização dos dados de acordo com as coordenadas geográficas, foi utilizado o sistema de *geohash* (NIEMEYER, 2008)(MOUSSALLI; SRIVATSA; ASAAD, 2015). Essa técnica visualiza o globo terrestre como uma estrutura espacial hierárquica, onde cada sub-espaço pode ser representado por um código de letras e números, que são facilmente convertidos a partir de pontos com latitude e longitude. A Figura 30 ilustra esse

sistema para áreas próximas à capital chinesa. Neste exemplo, temos duas áreas com código com três digitos (*wx4* e *wx5*) e suas sub-divisões. Esse sistema trabalha com até doze dígitos e pode converter coordenadas em código de área, mesmo quando o agente estiver *offline*.

Figura 30 – Sistema *Geohash* - exemplo para região de *Beijing*

O processo de clusterização identificou o intervalo entre 7h e 19h do dia 2008-02-04 como o período com maior número de publicações, sendo a região de *Beijing* com código *geohash wx4g* a que apresentou uma concentração maior de táxis alimentando a aplicação. O táxi com id 6275 foi identificado como aquele com maior número de registros compartilhados, sendo esse o táxi monitorado durante esta avaliação.

5.1.5.2 *Protocolos de comunicação*

Como visto na Seção 4.6.1, os protocolos implementados no gerenciador de comunicação permitem que os agentes *GiTo* se comuniquem com aplicações Web. Com isso, esses dispositivos publicam seus dados para servidores remotos através de requisições HTTP ou de conexões *Websocket*.

Nas execuções referência, o HTTP-*Get* foi utilizado para publicação dos dados, uma vez que o intervalo médio de compartilhamentos é de 215 segundos para a base dos táxis, e 15 segundos para as promoções. O uso do HTTP reduz a necessidade de manter a conexão *TCP* aberta para um fluxo com baixa frequência de dados.

Nos cenários de testes com alta frequência de dados (taxa de transferência em *ms*), a comunicação entre os dispositivos e servidores remotos foi realizada através de *WebSockets*. Essa abordagem provê uma redução no tráfico de rede e na latência de comunicação,

quando comparada ao modelo de *polling*, melhorando os cenários que simulam rajadas com alto volume de dados (PIMENTEL; NICKERSON, 2012).

5.1.5.3 *Execuções nas Camadas GiTo*

Essa seção descreve os modos de execução, detalhando quantidade de nós, fluxos de dados e especificando o que representa cada métrica coletada.

Em todos os cenários de teste e suas repetições:

- As métricas dos dispositivos da WoT são relacionadas ao agente *GiTo* com id 6275;
- O número de *threads* alocadas para o engenho CEP corresponde ao número de *cores* da máquina onde o CEP está sendo executado;
- As promoções foram publicadas a cada 15 segundos, enquanto que registros dos táxis foram compartilhados continuamente, respeitando a Tabela 4.

- Execução Local - Camada *Mist*

  Na execução local, o engenho CEP foi configurado no próprio agente *GiTo*. O cenário era composto por 2 nós, cada um emitindo um fluxo de dados.

  – Nó 1: Agente representando o táxi com id 6275. Este dispositivo ficou responsável por executar o engenho CEP processando os dados produzidos por ele próprio;
  – Nó 2: Uma máquina auxiliar criando 1 fluxo de dados com promoções que são publicadas para o agente com id 6275 através de requisições HTTP-*Get*.

  Os consumos de CPU e memória são coletados a partir do agente que representa o táxi 6275; o número de eventos enviados ao engenho CEP é a soma dos dados dos táxis e das promoções; o tempo de resposta se refere ao intervalo entre a publicação de um registro (promoção ou táxi) e a notificação do evento composto gerado por ele. As latências de comunicação e de rede foram consideradas zero, uma vez que os dados são processados no próprio dispositivo.

  Visualizando o cenário real da aplicação e as limitações dos dispositivos, o agente no modo de execução *Mist* processou apenas 1 fluxo de dados de táxi: o que o ele próprio produzia. Assim, foram executados 24 cenários de testes combinados de acordo com os fatores e níveis da Tabela 4, respeitando essa restrição dos fluxos.

- *Offloading* para a *Fog*

  Na execução na Fog, o engenho CEP foi instalado em um servidor localizado na mesma rede/intranet dos agentes *GiTo*. O cenário foi composto por 3 nós, com um número de fluxos na máquina auxiliar variando de acordo com a Tabela 4.

  - Nó 1: Um dispositivo representando o táxi com id 6275 compartilhando dados com o servidor na *fog*;
  - Nó 2: Uma máquina auxiliar publicando dados com:
    1. Um número de fluxos de acordo com a Tabela 4, onde juntos representam os táxis da região *wx4g* que mais publicaram suas localizações (não considerando o táxi 6275) através de conexões *WebSocket* distintas. Ou seja, cada agente desse conjunto compartilha seus dados de forma separada dos demais da região;
    2. 1 fluxo com dados de todos os outros táxis da região *wx4g* através de uma conexão *WebSocket*;
    3. 1 fluxo com dados das promoções via HTTP-*Get*.
  - Nó 3: Um servidor executando o engenho CEP localizado na mesma rede dos agentes, porém em uma *virtual LAN* separada. Este servidor envia para os nós 1 e 2 os eventos complexos detectados pelo CEP de acordo com o id de cada táxi.

  Foram executados 72 cenários de testes combinados de acordo com os fatores apresentados na Tabela 4. Neste modo de execução, os consumos de CPU e memória medidos são coletados do dispositivo associado ao táxi 6275 e do servidor; o número de eventos enviados ao engenho CEP é soma dos dados dos táxis de todos os fluxos e das promoções; o tempo de resposta se refere ao intervalo de quando a registro (promoção ou táxi) foi publicado pela nó 2 e enviado ao servidor, e eventos complexos enviados de volta ao agente com id 6275 (nó 1). Como mencionado no capítulo de implementaçao, a latência de comunicação é calculada a partir dos *frames* de controle *ping-pong* implementados pelo *Websocket* (FETTE; MELNIKOV, 2016), enquanto a de rede foi coletada através do protocolo ICMP.

- *Offloading* para a *Cloud*
  Nas execuções envolvendo a nuvem, uma composição similar à apresentada para a *Fog* foi especificada. As principais diferenças neste modo são: o nó 2 publica os dados de todos os táxis de todas as regiões, ou seja, os táxis compartilham também localizações fora da região *wxg4*, enquanto o nó 3 representa um servidor localizado na Internet. O nó 1 continua sendo o táxi com id 6275 e representa o agente *GiTo* monitorado na camada *Mist*.

5.1.5.4 *Execuções complementares*

Com intuito de verificar o impacto que a arquitetura pode causar nos resultados obtidos, dois outros cenários de testes foram adicionados na avaliação. Estes foram avaliados apenas na *Mist*, processando os dados localmente, uma vez que nesta camada os dispositivos tendem a apresentar menor poder computacional.

- Arquitetura *GiTo* com engenho CEP desativado
  Este cenário de teste permite medir o custo da arquitetura implementada independente do consumo gerado pelo CEP, pois os dados não são processados. Todo o fluxo entre os componentes é realizado, com exceção do engenho CEP. Neste caso, complexidade da consulta e tamanho da janela são ignorados. Como na *mist* as execuções processam apenas 1 fluxo de dados de táxi (ver Seção 5.1.5.3), apenas o fator taxa de transferência dos dados foi variado.

- CEP sem utilização da arquitetura *GiTo*
  Ao se executar o engenho CEP separado da arquitetura, tem-se o processamento dos dados independente dos componentes *GiTo*. Esse cenário de teste funciona como um *double-check* ou uma contraprova para a situação anterior. Neste caso, para ter uma consistência e viabilizar a comparação com o cenário da arquitetura sem o engenho CEP, mais uma vez apenas o fator taxa de transferência dos dados foi variado.

5.1.5.5 *Coleta das Métricas*

Com intuito de reduzir a influência da coleta dos dados nas execuções do sistema *GiTo*, optou-se por uma estratégia de "dividir para conquistar" (NIEUWPOORT; KIELMANN; BAL, 2001). Nessa abordagem, cada componente armazena seus valores separadamente, e a cada intervalo de 1 minuto os dados coletados são consolidados no componente *Metrics* presente no *ContextManager*, que assincronamente salva os valores em arquivo.

Todas as métricas são coletadas através de *logs* com *tokens* específicos, com exceção dos consumos de CPU e memória. Essas medições foram realizadas pelo programa Nmon[2], que captura as informações diretamente do sistema operacional *linux*.

Posteriormente, com todas as métricas coletadas, um *script* é executado consolidando os resultados de todas as execuções para que seja realizada a análise dos resultados obtidos.

5.1.5.6 *Infraestrutura de Hardware e Rede*

Atualmente, vários (micro)computadores e plataformas facilitam o desenvolvimento de soluções para IoT. Dentre eles, um computador que vem ganhando destaque pelo seu pequeno tamanho acoplado ao poder de processamento é o *Raspberry Pi* (RASPBERRYPI, 2016). Como já mencionado na Seção 4.1, esse computador é capaz de executar o *Esper*. Por esses motivos, este experimento utilizou a seguinte infraestrutura de *hardware*:

- *Raspberry Pi* 3 Modelo B como agente *GiTo* da WoT da camada *Mist*. Esse representou o táxi 6275 monitorado durante todos os modos de execução. Esse dispositivo será encontra instalado no Centro de Informática da UFPE;

[2] http://nmon.sourceforge.net/

- Um computador com 4 vcpus e 8GB de memória RAM. Esta máquina está localizada no Centro de Informática da UFPE em uma *vLan* diferente da *Raspberry Pi*, e foi utilizada como o servidor *Fog*;
- Um computador com 8 vcpus e 32GB de memória RAM localizado no Instituto Metrópole Digital (IMD)-UFRN e foi utilizado como servidor *cloud* na Internet.

Apesar do experimento não utilizar funcionalidades da computação na nuvem, como elasticidade, a utilização do servidor no IMD-UFRN é necessário para simularmos uma rede diferente da intranet do CIn-UFPE.

A escolha do *Raspberry Pi* como dispositivo da WoT se deu pelo fato do experimento trabalhar com grande volume de dados, necessitando assim de um *hardware* capaz de trabalhar com rajadas de dados à altas taxas de transferência. Assim, foi possível observar o comportamento do engenho em diferentes situações.

5.2 Etapa 2: Políticas em Uso

Essa etapa visa integrar as três camadas em uma única execução, e de fato experimentar a arquitetura *GiTo* e o componente orquestrador de agentes. Uma vez realizada a análise de desempenho, e compreendido os limites da execução, pode-se definir os valores para as métricas e utilizá-los como regras das políticas. O objetivo não é testar a arquitetura com diferentes volumes de dados, como anteriormente, mas sim, exercitar a utilização do monitoramento das políticas e a distribuição dos dados através da realização do *offloading* computacional.

Dos testes especificados anteriormente, espera-se que o cenário que trabalha com a consulta *pattern*, janela de tempo de 600s e taxa de transferência de 1ms, seja aquele que irá apresentar maior consumo dos recursos do dispositivo. Logo, esse será exercitado de forma integrada, e com o uso de políticas garantir que comportamentos não esperados aconteçam.

5.2.1 Políticas

Como mostrado no Capítulo 4, as políticas são definidas através de um arquivo XML e utilizadas durante toda a execução do sistema com objetivo de garantir o estado correto da execução. Nesses testes, as políticas relacionadas às métricas de CPU, memória, tempo de resposta e latência de comunicação foram escolhidas, uma vez que é possível alterá-las através do volume de dados configurados para cada máquina utilizada no experimento, diferentemente da métrica latência de rede.

Os valores foram definidos após a análise dos resultados da Etapa 1, e referem-se ao momento atual da execução. Por exemplo, o valor do consumo de CPU especificado na regra de política é o valor máximo permitido, e não a média dos valores coletados.

Após a Etapa 1 e análise dos resultados, foi verificada a necessidade de uma melhoria na especificação das regras de políticas, e os campos *repetitions* e *order* foram adicionados. O primeiro campo permite definir uma tolerância na regra, por exemplo, uma ação só será disparada caso o consumo de memória ultrapasse duas medições consecutivas. Isso reduz o *overhead* inicial desnecessário do processo de *offloading* quando existem picos de consumo isolados na execução (os *outliers*). Já o segundo campo, permite a definição de relação de ordem e sequência entre as regras de uma política composta. Com isso, podem-se definir políticas mais sofisticadas, como mostrado na Figura 31.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <policies type="DNF">
4     <policy type="simple" name="max_memory" value="0.8" action="offloading" repetitions="3"/>
5     <policy type="simple" name="max_cep_response_time" value="5" action="offloading" repetitions="3"/>
6
7     <policy type="composed" action="offloading" repetitions="3">
8       <rule name="max_cpu" value="0.6" order="1"/>
9       <rule name="max_memory" value="0.6" order="2"/>
10    </policy>
11 </policies>
```

Figura 31 – Políticas declaradas como XML - campos *repetitions* e *order*

Apenas o campo *repetitions* foi inserido na implementação da PoC, uma vez que só esse tipo de política foi implementado. A política da linha 6 foi utilizada nas execuções desta etapa.

5.2.2 Cenários de Teste

A infraestrutura utilizará a mesma configuração da análise de desempenho, e dois cenários de testes serão exercitados. Os dispositivos das camadas serão iniciados ao mesmo tempo, com políticas similares definidas, alterando apenas os seus valores.

- **Cenário 1: Servidor *Fog* disponível**
  Nesta execução, o processamento dos dados é iniciado no agente *GiTo* com identificador 6275. Quando uma política for violada, o *offloading* para o servidor na *Fog*, que estará habilita à processar os dados.

  Durante a execução, novos agentes iniciarão também a publicação de dados para o Servidor *Fog*, sobrecarregando-o. Esse volume de dados irá aumentar o tempo de resposta do agente 6275, que irá encerrar o *offloading* com esse servidor, e iniciar com a *Cloud*.

  É esperada a realização de duas operações de *offlloading*: (*Agente* → *Fog*) e (*Agente* → *Cloud*).

- **Cenário 2: Servidor *Fog* sobrecarregado**
  Similar ao cenário 1, o processamento é iniciado no agente. Porém, o servidor na

Fog já esta processando dados de outros dispositivos, enquanto o servidor na *Cloud* está disponível.

O agente iniciará o *handshaking* de *offloading* com o servidor na *Fog*, e irá negar o *offloading*. Assim, o agente realizará o *offloading* com o a *Cloud*.

É esperada a realização de 1 operação de *offlloading*: (*Agente* $\rightarrow$ *Cloud*).

Com as execuções da etapa 2, é possível perceber o uso das políticas atuando no sistema, verificar o balanceamento do volume de carga e analisar o tempo de resposta final.

6 RESULTADOS

Este capítulo apresenta os resultados das execuções dos cenários de testes conforme metodologia definida como avaliação da arquitetura *GiTo*. Inicialmente, são mostrados os números relacionados à execução referência. Depois, os demais cenários terão seus resultados detalhados com foco no tempo de resposta, pois essa métrica é impactada por todas as outras. Por fim, os valores da execução da arquitetura com as políticas em uso e uma discussão sobre elas são apresentados.

6.1 Execução de Referência

Como apresentado na Seção 5.1.5, nestas execuções foram utilizados os *timestamps* originais dos registros. A Tabela 4 especifica os níveis para os fatores selecionados e utilizados. Nesta execução foram escolhidos aqueles que, teoricamente, mais sobrecarregam o processamento: consulta *pattern* com janela de tempo de 600s e 8 agentes *GiTo* compartilhando seus dados com os servidores da *Fog* e da *Cloud*. Assim, foi possível exercitar o cenários com rajadas similares às obtidas no experimento da *Microsoft Research*, uma vez que o envio dos dados foi controlado pelo *timestamp* de cada registro.

Foram monitorados o táxi com id 6275 representando o agente da camada *Mist*, e os servidores remotos na *Fog* e na *Cloud*, que receberam os dados dos demais táxis, conforme detalhado no projeto do experimento.

Na Tabela 5 são apresentados os resultados obtidos nas execuções referências, mostrando os valores para as métricas especificadas. O agente *GiTo* 6275 processa apenas os seus dados quando o CEP é executado localmente, independente da região geográfica de sua localização. Quando o *offloading* é realizado para a nuvem, todos os registros de todos os táxis são enviados ao servidor remoto.

Tabela 5 – Execuções Referência

	Mist	Fog	Cloud
CPU Rasp/Server (%)	0.45	0.36/1.43	0.37/1.81
Memória Rasp/Server (MB)	44.71	25.30/239.64	26.53/442.49
Eventos Enviados pelo Agente 6275	29808	17714	29814
Eventos Detectados pelo CEP	12896	112923	174395
Eventos Processados	32688	671972	1723020
T.R. Médio/Máximo (ms)	129/270	181/1184	127/704
Latência Médio/Máxima de Rede (ms)	NA	0.87/5.27	6.79/77.54
Latência Médio/Máxima de Comunicação (ms)	NA	4.54/1217.27	18/1979.75

Na *Fog*, o número de dados enviados pelo agente 6275 é menor comparado aos outros modos, pois o servidor localizado na região *wx4g* recebe apenas os dados publicados pelos táxis quando eles estão dentro dessa área. Na nuvem, os dados de todos os táxis são processados, logo, mais eventos complexos são detectados. Na *mist*, o CEP processa um número maior de eventos (32688) do que os recebidos (29808), pois, além dos dados dos táxis, eventos internos do temporizador do próprio engenho são analisados para que seja possível controlar a janela de tempo. Os resultados das execuções mostram baixo uso dos recursos para todos os modos. Os consumos de CPU e memória do agente na camada *Mist* para publicar os seus dados para *Fog* e *Cloud* são similares aos observados quando ele próprio realiza o CEP.

A *Fog* apresentou tempos de resposta médio e máximo maiores que os números da *Cloud*. Provavelmente, isso aconteceu devido à densidade da região *wx4g* para o servidor utilizado. Apesar dessa diferença, o tempo de resposta médio das notificações foi baixo em todos os cenários, em torno de 205 milissegundos para todas as repetições, o que torna os eventos complexos recebidos no dispositivo aptos a serem utilizados pela aplicação. Caso tivéssemos um tempo de resposta alto (por exemplo, acima de 1 minuto), a promoção poderia não fazer sentido para o passageiro daquele táxi, pois o mesmo poderia já ter desembarcado. Neste caso, a qualidade da informação recebida poderia não seria aceitável.

Para os cenários de testes propostos, não foram detectados problemas em nenhum dos modos de execução no processamento de eventos complexos. É importante registrar que, para uma solução real, considerando a configuração das execuções referências, uma infraestrutura de menor custo seria uma opção viável, uma vez que os percentuais de uso de CPU e memória foram baixos. De fato, com os parâmetros analisados e com os tempos de resposta como os apresentados na Tabela 5, é possível realizar o processamento de dados no próprio agente *GiTo* presente na *Mist*, mesmo que esse tenha um poder computacional menor que o dispositivo utilizado no experimento. Dessa forma, dispensa-se a necessidade de servidores remotos e reduz-se o investimento financeiro da aplicação como um todo. Essa abordagem é válida, inclusive, para soluções onde o tempo de resposta é crítico, como as aplicações na área de saúde (XU et al., 2013)(CRACIUNESCU et al., 2015). Para a aplicação de táxi proposta, os resultados observados indicam a *Mist* como uma opção mais apropriada.

O resultado da Tabela 5 fica como uma referência para as demais execuções, uma vez que os outros fatores/níveis buscam medir o comportamento do engenho CEP em situações variadas, principalmente quando o *throughput* dos dados é maior. A latência de rede pouco influenciou no resultado, uma vez que valores baixos foram apresentados.

6.2 Testes em Camadas

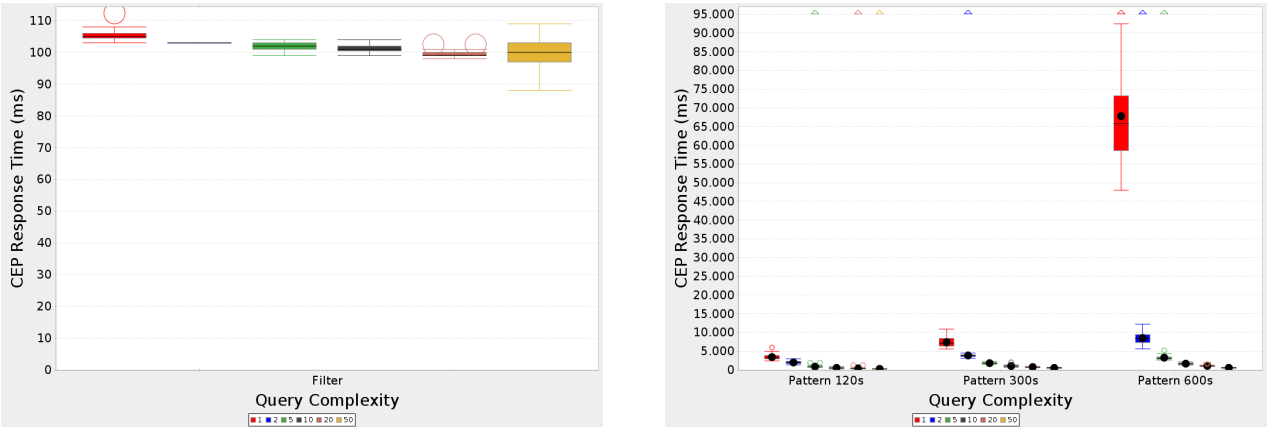
Esta seção detalha os resultados das execuções que utilizaram tempos pré-definidos como taxa de transferência de dados (coleta das informações dos sensores) e exercitaram a arquitetura *GiTo* através de um volume de carga maior do que os cenários referências.

Os gráficos apresentados buscam uma melhor compreensão dos valores obtidos. Para simplificar a referência aos cenários de testes, utilizou-se a notação descrita no Capítulo 5: *(Modo_execução, complexidade_consulta, taxa_transferência, Tamanho_Janela, num_fluxos)* com intuito de identificar um determinado teste.

6.2.1 Resultados na *Mist*

Neste modo de execução, o fator número de fluxos foi sempre 1, pois, apenas os dados produzidos pelo próprio agente da camada *Mist* foram analisados.

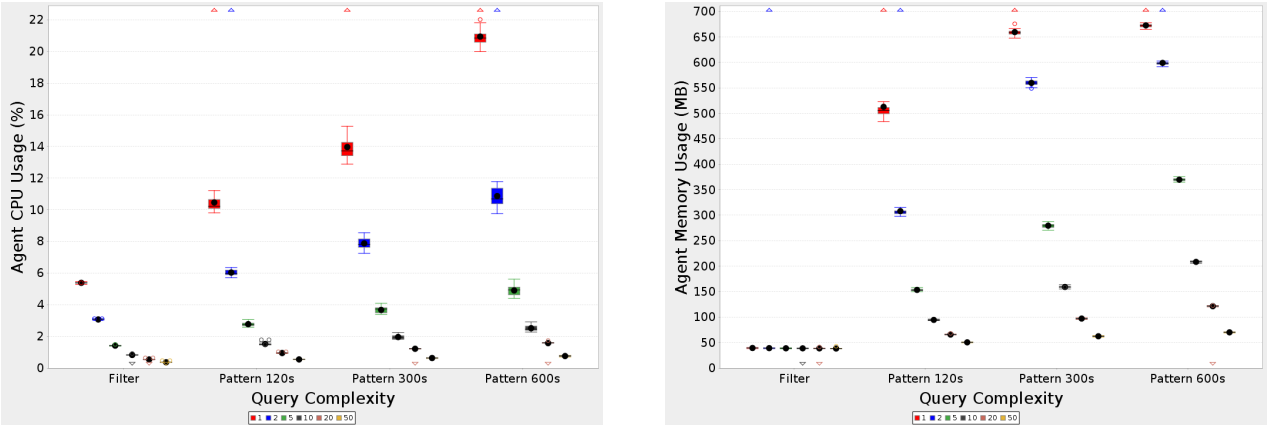
O gráfico da Figura 32(a) mostra o tempo de resposta médio para as execuções *filter*, enquanto a Figura 32(b) ilustra essa mesma métrica para a consulta *pattern*. Para as consultas mais simples, o tempo de resposta é praticamente constante e baixo, em torno de 102ms, enquanto que nas consultas complexas ele aumenta, chegando perto de 5s para janela 120s, ultrapassando 10s para 300s e medindo valores perto de 93s para o pior caso.



(a) *Filter* (b) *Pattern*

Figura 32 – Mist: Tempo de Resposta

Para entender essa diferença de comportamento, foram analisados também os consumos de CPU e memória, cujo valores obtidos são apresentados pelos gráficos *bloxplot* das Figuras 33(a) e a 33(b), respectivamente, variando a taxa de transferência dos dados, a complexidade da consulta e a janela de tempo. Os gráficos mostram a pequena oscilação nos consumos para todas as repetições, em todos os cenários, chegando a variar no máximo em 2% para CPU e 6% para memória.



(a) *CPU* (b) *Memória*

Figura 33 – Mist: CPU e Memória variando taxa de transferência e complexidade da consulta

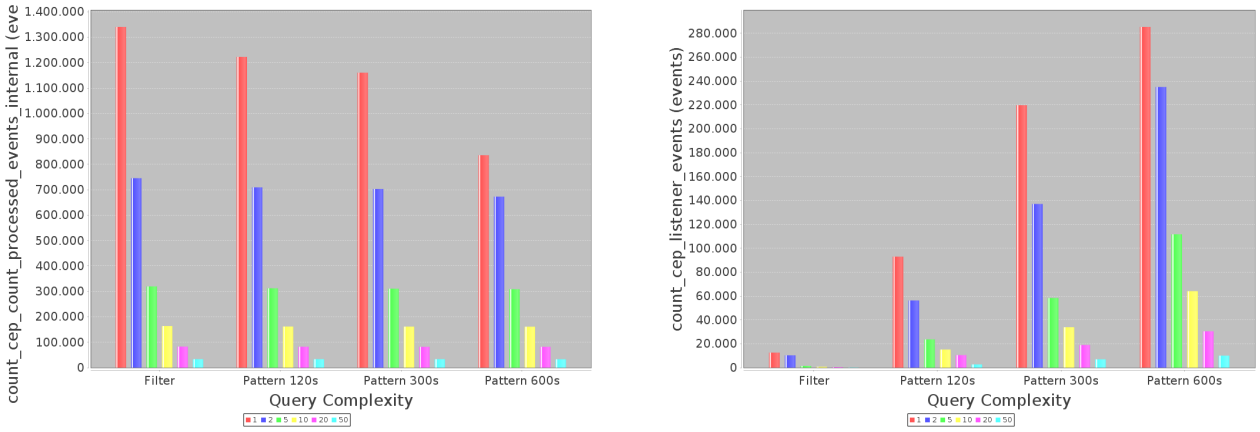
Os consumos de CPU e memória são diretamente proporcionais à taxa de transferência dos dados. Quanto maior a quantidade de eventos em um determinado período de tempo, maior a carga no engenho, aumentando assim o custo do processamento. O mesmo comportamento foi o observado em relação à complexidade da consulta e a janela de tempo. Consultas simples, como filtros, não tiveram consumo médio de CPU maior que 5,5% e apresentaram utilização de memória praticamente constante, em torno 39,30MB (5,32%). Já nas execuções para *Pattern*, a CPU chegou, em média, a 21%, com pico de 48,5% (não mostrado no gráfico), enquanto que a alocação de memória ultrapassou 500MB (69%) para a janela de 120 segundos, chegando próximo à 660MB (88%) e 670MB (91%) nas janelas de 300 e 600 segundos, respectivamente. Nestes casos, foi verificado que o método *sendEvent()* do engenho *Esper* aloca os dados em uma fila interna, para posterior processamento. Esse enfileiramento pode ter relação com o significativo aumento de memória à medida que a taxa de transferência dos dados aumenta. O fato do consumo de memória alcançar valores próximos ao limite disponível no cenário (1ms, *Pattern*, 600s) justifica o elevado tempo de resposta quando comparado aos demais casos, pois, com a memória sobrecarregada toda a execução é comprometida, impactando também no tempo de processamento das operações.

O processo de análise dos dados tem três principais linhas de execução: a coleta dos dados, o monitoramento das políticas e o CEP. Pegando como exemplo o cenário (*pattern*, 1ms, 600s), onde o consumo médio de CPU ficou em torno de 21%, foi verificado que 3.8% estava relacionado ao processo de captura dos dados, menos de 0.1% para os componentes relacionados à política e 17.1% para o engenho CEP. Para taxa de transferência menores, e consultas de baixa complexidade, o engenho CEP foi pouco exigido. Nota-se esse comportamento na gráfico 33(a), onde a consulta *Filter* apresentou baixo consumo de CPU, com média de 5.4% para o pior caso.

O gráfico da Figura 34(a) mostra a quantidade de eventos processados pelo engenho

CEP. Os cenários com (*Pattern*, 600s) tiveram um número menor de dados processados para o intervalo 1ms. Foi verificado que o número de dados coletados também foi menor (não mostrado no gráfico). Provavelmente, o consumo elevado de CPU e memória impactaram no desempenho da execução como um todo. Esse cenário foi considerado como o pior caso. Mesmo assim, por manter uma quantidade maior de eventos ativos na janela de tempo, conseguiu detectar mais eventos compostos Figura 34(b).

Como o engenho CEP foi configurado para trabalhar com um *Buffer* interno para recebimento dos dados, o número de eventos coletados da fonte de dados externa foi sempre igual ao número de eventos enviados ao CEP. Este, por sua vez, foi o mesmo valor observado para o número de eventos processados. Ou seja, todos os eventos enviados foram sempre processados.



(a) Eventos Simples Processados

(b) Eventos Compostos Detectado

Figura 34 – Mist: Eventos simples e Eventos Compostos

Neste modo de execução, menos de 0,43% das repetições apresentaram erros, como promoção não recebida no agente *GiTo*, tempo de execução diferente do esperado e exceções inesperadas.

O resultados obtidos mostram que é possível executar CEP localmente, porém, para consultas de complexidade alta com *throughput* elevado dos dados, o consumo elevado de memória se apresenta como um gargalo.

6.2.2 Resultados na *Fog*

No modo de execução *Fog*, todo o processamento dos dados é realizado em um servidor localizado na mesma *Local Area Network* (LAN) do agente *GiTo* da camada *Mist*, porém em diferentes *vLANs*. O dispositivo da IoT coleta os dados e os envia por um canal *websocket*, reduzindo o consumo dos recursos localmente e realizando o *offloading*.

Inicialmente, uma comparação com a execução local do cenário (*pattern*, 1ms, 600s), considerado o pior caso, foi realizada, sendo publicado apenas o fluxo de dados do agente

6275. Esse paralelo entre os cenários local e na *fog* serve para entender melhor o *offloading* para de um único dispositivo. O resultado obtido é mostrado na Tabela 6, onde houve uma migração dos dados e do processamento. Os valores obtidos mostram a queda no consumo de memória e CPU do agente 6275 na camada 1, que se mantém constante para um mesmo intervalo de coleta, uma vez que o engenho CEP não é processado nele. Houve também uma redução no tempo de resposta das notificações, que mesmo baixo em comparação ao modo de execução local, pode ser alto para algumas aplicações. O consumo de memória no servidor *Fog* apresentou valores acima do esperado, quando comparado à execução *Local*. Suspeita-se de uma relação com a taxa de transferência de 1ms, o qual o engenho CEP não conseguiu processar todos os eventos recebidos, fazendo com que o *buffer* interno ou de rede acumulasse eventos simples recebidos.

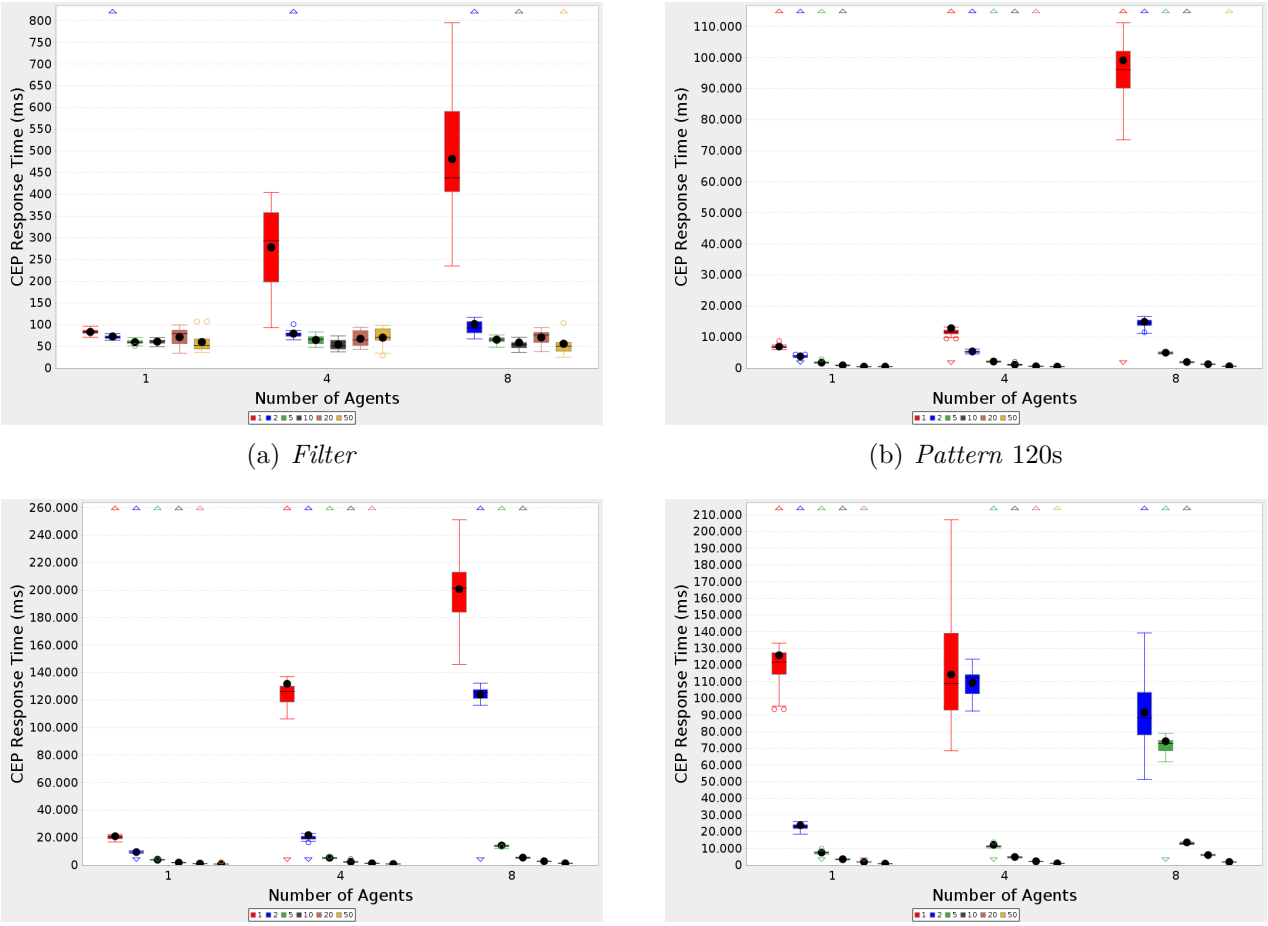
Apesar de válido o cenário, esse não é realista, uma vez que a *Fog* e a *Cloud* possuem (teoricamente) servidores capazes de processar dados de vários dispositivos ao mesmo tempo. Assim, diferente da execução local, os demais testes na *Fog* consideram, além do dispositivo monitorado (táxi 6275), o *offloading* realizado por N agentes ao mesmo tempo, com N variando entre 1, 4 e 8, além de um fluxo contendo todos os demais táxis da região onde o servidor está localizado.

Tabela 6 – *Mist* vs. *Fog* - *Pattern*, 1ms, janela de 600s e 1 fluxo de dados

	Mist	Fog (IoT Device / Server)
Consumo médio CPU (%)	21	6,1 / 12
Consumo médio Memória (%)	90,6 (680MB)	3,73 (27,97MB) / 23,71 (1699MB)
Eventos Processados	845000	1.340.000
T.R. Médio (segundos)	66s	9,1s

Os gráficos da Figura 35 ilustram os resultados observados para a métrica tempo de resposta. É possível notar sua variação, principalmente quando o volume de carga é alto (vários fluxos com baixos intervalos de coleta). Para operação *filter*, os valores obtidos não ultrapassam *800ms*, com valores entre 50ms e 100ms para grande parte dos cenários. Variações significativas foram apresentadas para as execuções *pattern* com janela de tempo maiores processando 4 e 8 fluxos adicionais. Por exemplo, no cenário *(Fog, 1ms, pattern, 600s, 4 fluxos)* o tempo de resposta foi maior que 200s. Para outros intervalos de coleta de dados, como 2ms e 5ms, os valores obtidos se aproximam de 5s para os casos com menor carga de dados, duplicando nos cenários intermediários e ultrapassando 80s nos piores casos. Os gráficos 35(b), 35(c) e 35(d) apresentam os resultados coletados. Esse aumento no tempo de resposta não tem relação com a latência de rede que, durante as repetições, não ultrapassou 0,9ms.

Buscando um melhor entendimento dos valores já mostrados para a métrica tempo de resposta, apresenta-se os resultados dos consumos de CPU e memória, tanto do agente



(a) *Filter* (b) *Pattern* 120s

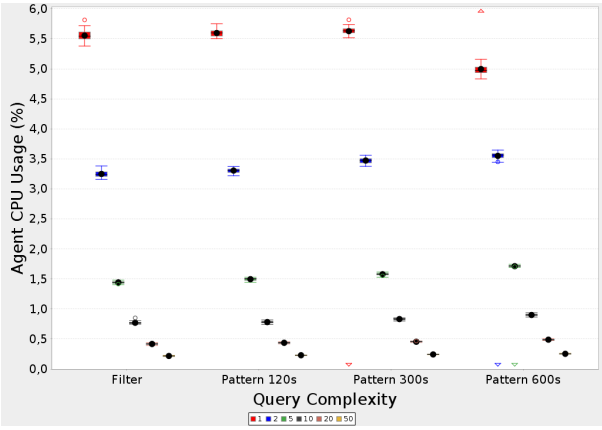
(c) *Pattern* 300s (d) *Pattern* 600s

Figura 35 – Fog: Tempo de Resposta

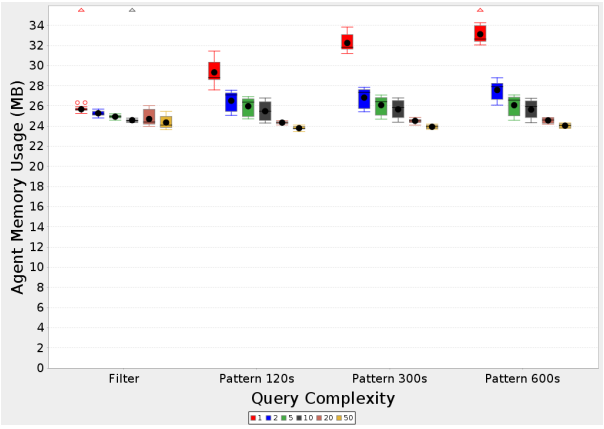
na *Mist* como do servidor na *Fog*. Os gráficos das Figuras 36(a) e 36(b) refere-se ao dispositivo na *Mist*. Nota-se a similaridade nos valores quando comparados aos obtidos na Tabela 6. Os dados são migrados para o servidor e todo o processamento é realizado fora da camada *Mist*. O intervalo de coleta de *1ms* apresenta consumo de CPU maior que os demais porque o sistema de fato executa mais operações na captura dos dados (mais operações de *I/O*).

A coleta de dados no agente localizado na *Mist* não apresentou consumo elevado dos recursos do dispositivo. Diferentemente do servidor da *Fog*, que nestes cenários ficou responsável pelo processamento dos eventos complexos. As Figuras 36(c), 36(d), 36(e) e 36(e) mostram os valores obtidos para esse último. Assim, como na execução local, a consulta *filter* mostrou resultados baixos, apesar da variação quando o intervalo de coleta é alterado. Porém, o servidor mostrou elevado consumo de CPU e memória para taxas mais altas (1ms e 2ms). Essa fato provavelmente aconteceu porque a área *wx4g* possui muitos táxis que foram beneficiados com as promoções, aumentado o processamento e exigindo mais do engenho CEP, além do possível acúmulo de eventos no *buffer* do engenho.

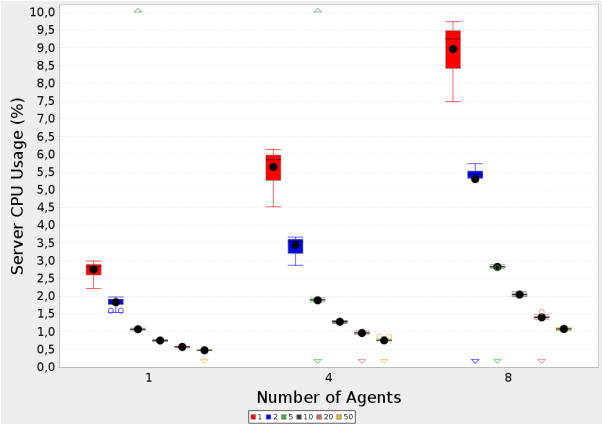
Neste cenário de testes, utilizando a base selecionada, as taxas de transferência men-



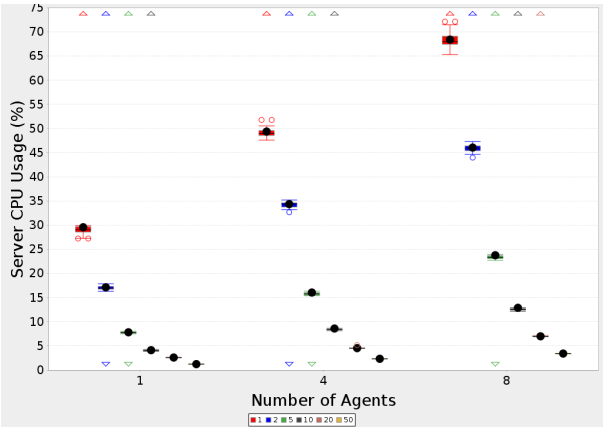
(a) CPU - Agente na Mist



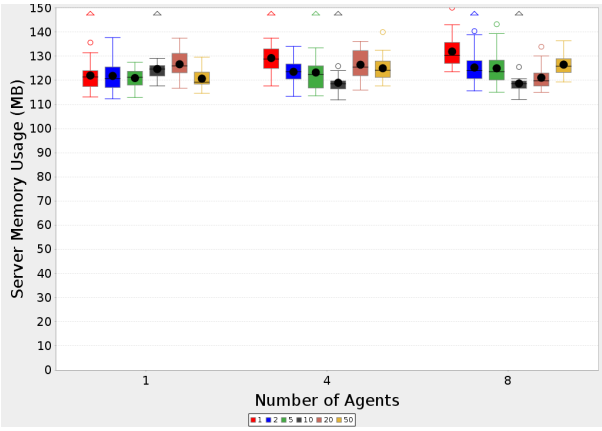
(b) Memória - Agente na Mist



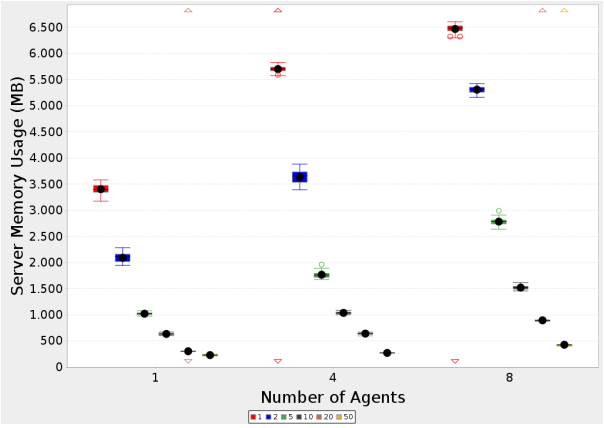
(c) CPU - Servidor Fog - Consulta *FILTER*



(d) CPU - Servidor Fog - Consulta *PATTERN*



(e) Memória - Servidor Fog - Consulta *FILTER*

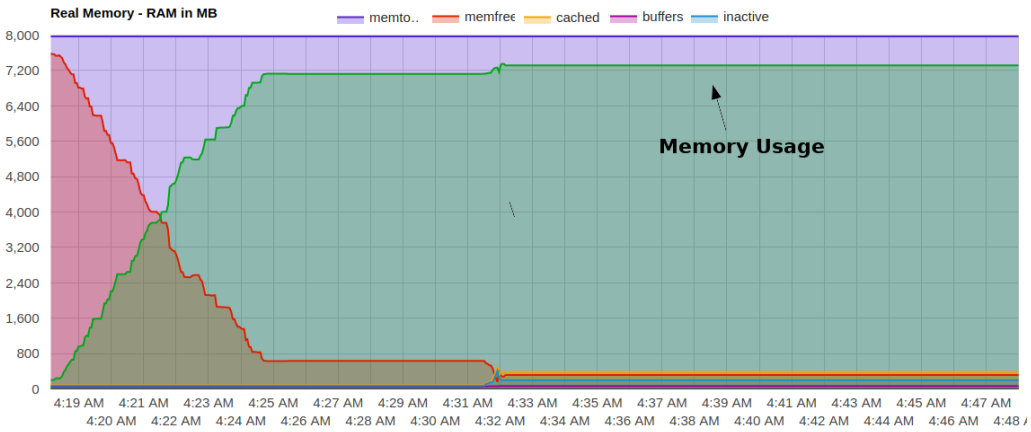


(f) Memória - Servidor Fog - Consulta *PATTERN*

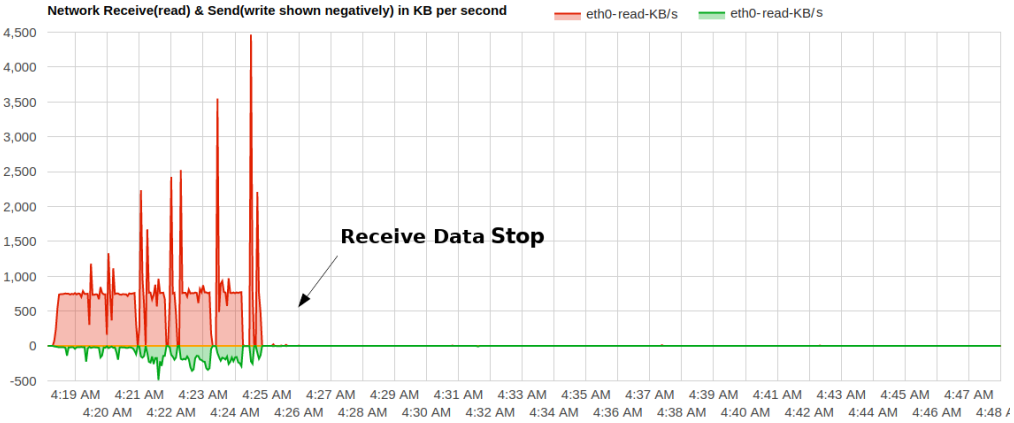
Figura 36 – Fog: Consumo de CPU e Memória do agente *GiTo* 6275 e Servidor

cionadas não são adequadas. Tomando como exemplo os cenários de teste com 1ms, janela de tempo 600s e 8 dispositivos adicionais (não mostrados nos gráficos), o servidor na *Fog* apresentou erro de *OutOfMemory* para todas as repetições. A Figura 37 mostra que após 6 minutos (a) a memória é preenchida, (b) os dados deixam de ser recebidos, mas, (c) o uso da CPU continua alto, mesmo o agente 6275 na camada *Mist* recebe uma notificação

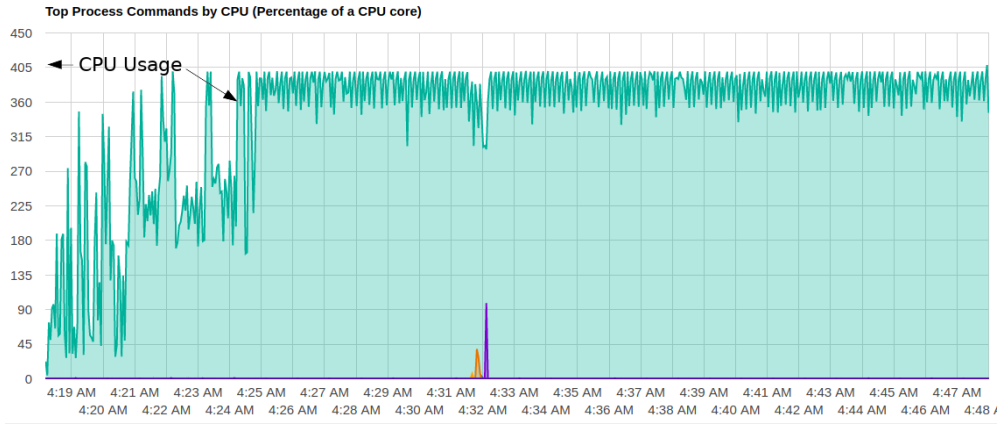
de conexão *websocket* encerrada. Ou seja, o servidor apresenta um mal funcionamento devido à sobrecarga no sistema.



(a) *Memória*



(b) Rede



(c) CPU

Figura 37 – Fog: *Out of Memory - Pattern* com janela de 600s e 8 fluxos adicionais

Neste modo de execução, 6,16% das repetições apresentaram erros, sendo 98 execuções finalizadas com tempo de duração diferente do esperado, geralmente ocorrido por falha nas conexões.

6.2.3 Resultados na *Cloud*

No modo de execução *Cloud*, todo o processamento dos dados é realizado em um servidor localizado no IMD-UFRN, que como o Centro de Informática-UFPE, também está ligado à Rede Nacional de Pesquisa (RNP). Similar as execuções anteriores, o dispositivo na camada da *Mist* coleta os dados e realiza o *offloading* através de uma conexão *websocket*.

O objetivo desta seção é apresentar os resultados quando vários fluxos de dados são processados pelo engenho CEP, tentando se aproximar de um cenário real. Porém, assim como na seção que detalhou o processamento na *Fog*, as métricas coletadas para o servidor remoto quando apenas o agente 6275 realiza o *offloading* também são apresentadas. A Tabela 7 mostra os valores obtidos, agregando os três modos de execução, para o cenário de teste que tem se apresentado como pior caso.

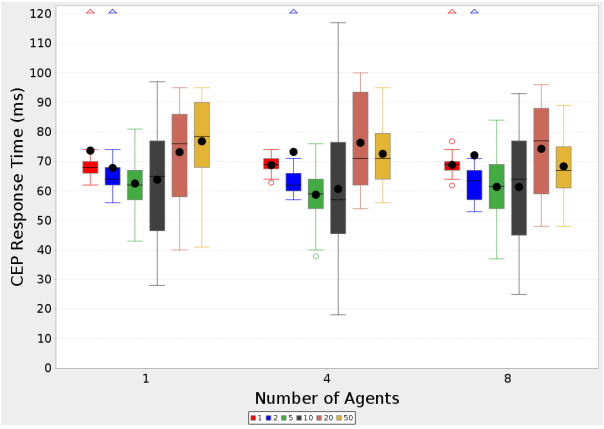
Tabela 7 – *Mist* vs. *Fog* vs. *Cloud* - *Pattern*, 1ms, janela de 600s e 1 fluxo de dados

	Mist	Fog (IoT Device / Server)	Cloud (IoT Device / Server)
Consumo médio CPU (%)	21	6,1 / 12	5,9 / 3,1
Consumo médio Memória (%)	68	3,73 / 23,71	3,76 / 6,56
Eventos Processados	845000	1.340.000	1.340.000
T.R. Médio (segundos)	66s	9,1s	1,7s

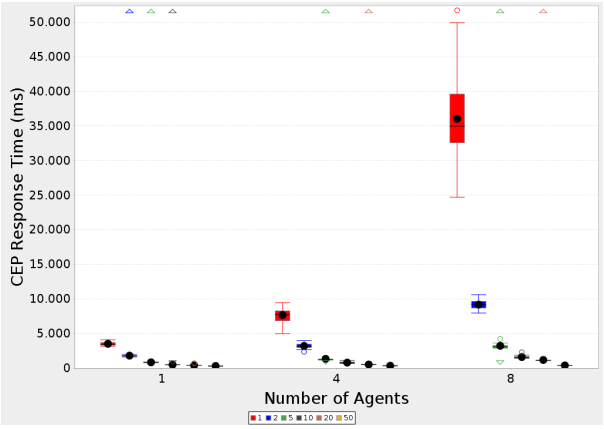
Os valores da Tabela 6 mostram que para processar apenas 1 fluxo no cenário (1ms, *pattern*, 600s) a nuvem consegue obter melhor tempo de resposta, porém, boa parte do seu poder computacional fica ocioso. Quando N agentes realizam o *offloading* ao mesmo, o *throughput* dos dados é multiplicado por N, exigindo maior desempenho do servidor. O gráfico da Figura 38 mostra os tempos de resposta obtidos quando com vários fluxos de dados provenientes dos agentes. Os resultados alcançam média de 36s na janela de 300s e 140s para 600s com 8 fluxos (esse último não apresentado no gráfico).

Os valores dos tempos de resposta são justificados pelos consumos de memória e CPU. A Figura 40 ilustra os resultados coletados para essas duas métricas na nuvem variando o número de fluxos adicionais para as complexidades *Filter* e *Pattern*, essa segunda fixando a janela de tempo da consulta em 300s. Os valores observados para o agente *GiTo* na *mist* não serão mostrado, uma vez seu comportamento é similar ao da execução na *Fog* e das Tabelas 6 e 7.

O consumo médio de CPU para operações *filter* foi de 2,8% para o cenário com 8 fluxos extras, com memória não ultrapassando 240MB. Diferente do processamento *pattern*, onde o consumo de CPU atingiu 42% para o cenário *(1ms, 300s, 8 fluxos)* com memória chegando à valores próximos de 15GB. Para janela de tempo e intervalo de coleta de dados menores, os resultados para o custo operacional foram reduzidos. Os gráficos ainda mostram a pequena variação nos valores obtidos nas 30 repetições. O alto consumo de

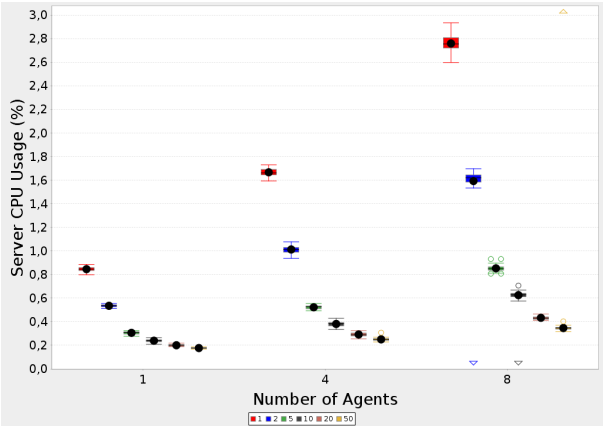


(a) *Filter*

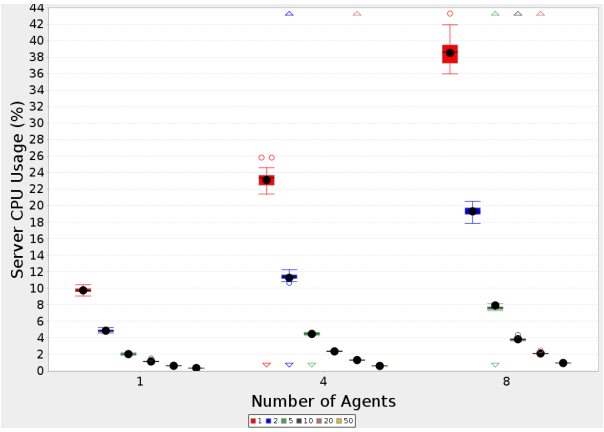


(b) *Pattern* 300s

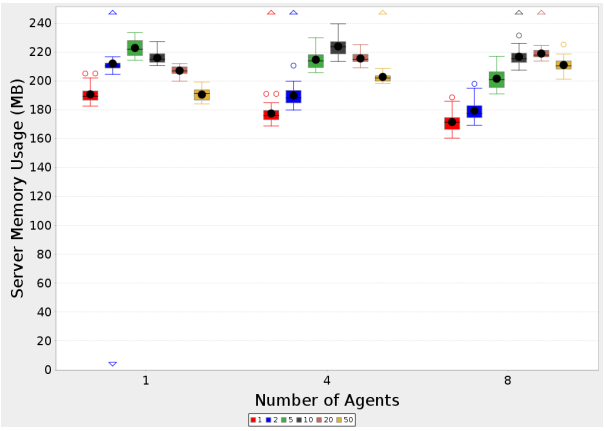
Figura 38 – Cloud: Tempo de Resposta médio



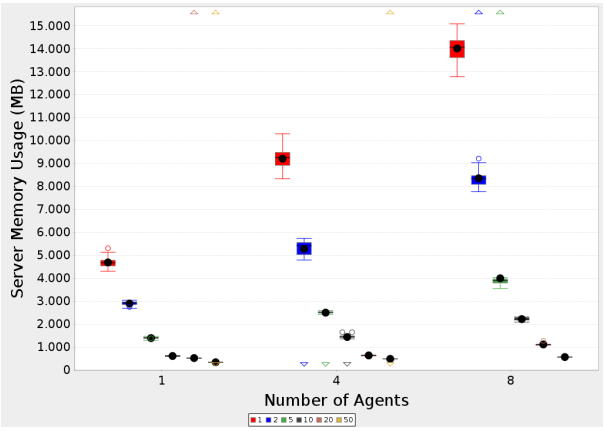
(a) CPU - Consulta Filter



(b) CPU - Consulta Pattern 300s



(c) Memória - Consulta Filter



(d) Memória - Consulta Pattern 300s

Figura 39 – Cloud: Consumo de CPU e Memória do servidor

memória para o caso com intervalo de 1ms se deve ao fato que neste modo de execução, todos os dados, de todas as áreas são enviados ao servidor.

Assim como na *Fog*, a latência de rede não foi um problema, pois os valores obtidos

através do *ping-icmp* não ultrapassaram 8ms e o *ping-pong* do *websocket* com valor médio máximo de 3,2s para a janela 300s e 4,2s para 600s.

Neste modo de execução, 11,64% das repetições apresentaram erros, sendo 247 execuções finalizadas com tempo de duração diferente do esperado, geralmente ocorrido por falha nas conexões.

6.3 Discussão

A implementação da arquitetura *GiTo*, quando executada de maneira pré-definida (sem a utilização do agente orquestrador), conseguiu integrar os componentes desenvolvidos ao engenho CEP. O processamento local apresentou uma boa estabilidade, com baixa taxa de erro (0,5%). Nas execuções na *Fog* e na *Cloud*, os agentes conseguiram realizar o *offloading*, enviando e recebendo as notificações através do *Communication Manager*. Os componentes *Policy Manager*, *Policy Enforcer* e o *Context Manager* apresentaram valores corretos para o monitoramento das políticas. Para checá-los, foi executada uma ferramenta externa ao PoC, chamada *nmon*[1].

6.3.1 Volume de Dados

Ao variar o número de dispositivos/fluxos e a taxa de transferência de dados para os dispositivos, pode-se avaliar o comportamento do sistema em situações com diferentes volumes de dados, verificando inclusive o retorno das informações. O intervalo de coleta e o *throughput* de dados influenciam diretamente no consumo de recursos, tanto do agente *GiTo* na camada *Mist*, como nos servidores remotos quando o *offloading* está sendo realizado.

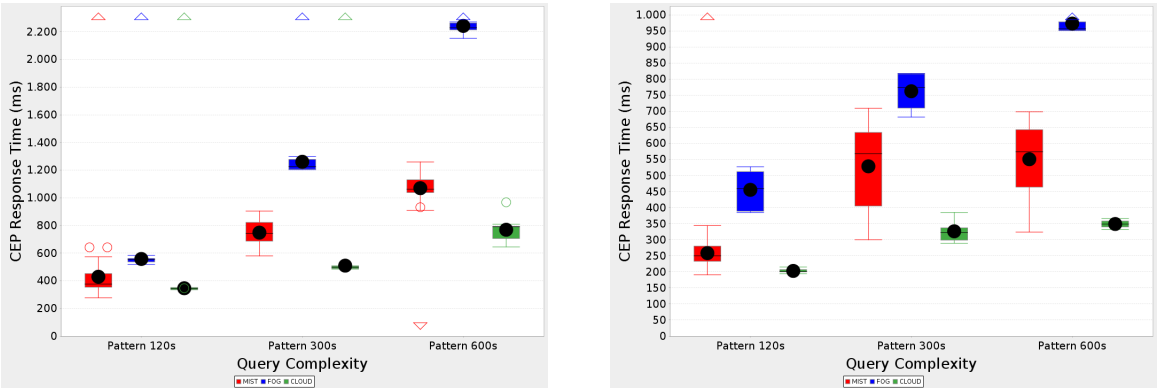
Os cenários referências (*timestamp* original)mostram que para um volume de dados pequeno, com *throughput* em torno de 215 segundos, uma infraestrutura simples consegue processar os dados em um tempo hábil para as aplicações. Nesse, a execução local se apresentou como uma boa solução para o processamento em tempo real. Sabendo que outras aplicações reais utilizam intervalo de publicação dos dados na casa de segundos, essa abordagem pode ser interessante. Os ônibus do grande Recife compartilham suas localizações à cada 30 segundos; e estudos mostram o monitoramento de temperatura ambiente em uma casa inteligente coletando os valores em um intervalo de 10 segundos (KELLY; SURYADEVARA; MUKHOPADHYAY, 2013).

Em relação aos testes em camadas, com diferente *throughputs*, a configuração da infraestrutura utilizada não se mostrou adequada para os volumes apresentados com 1ms, 2ms, 5ms e 10ms. Apesar dos servidores terem processado os dados recebidos, o tempo de resposta aumentou consideravelmente para intervalos de coleta pequenos, principalmente

[1] http://nmon.sourceforge.net

pelo elevado consumo de memória. Todas as métricas observadas tiveram incremento que degradam o desempenho da máquina, gerando um gargalo e comprometendo o tempo de resposta das notificações, reduzindo assim a utilidade do evento recebido por parte do dispositivo.

A partir de 20ms, os valores coletados apresentaram melhora no tempo de resposta do CEP. Pegando como exemplo o cenário de testes (*pattern*, 4 fluxos), e variando o modo de execução e a janela de tempo, obteve-se os valores para tempo de resposta para 20ms e 50ms, como mostrado nas Figuras 40(a) e 40(b), respectivamente. Nestes gráficos, a execução local sempre apresenta 1 fluxo. Ou seja, quando comparamos *Mist* com *Fog*, estamos comparando *Mist* com 1 fluxo, e *Fog* com 4 fluxos. Testes locais sempre tiveram apenas 1 fluxo, com intuito de mostrar a possibilidade do processamento no próprio dispositivo.



(a) 20ms (b) 50ms

Figura 40 – Mist vs. Fog vs. CLoud: Tempo de Resposta - *Pattern* e 4 fluxos

6.3.2 Complexidade da Regra EPL e Janela de Tempo

Para consultas simples, do tipo *Filter*, todos os cenários apresentaram tempo de resposta baixos, mesmo aqueles com alto *throughput* de dados. Isso viabiliza o uso deste tipo de consulta na *Mist*, uma vez que os recursos foram poucos consumidos. Para consultas *pattern* combinadas com o atributo *every*, o consumo foi maior. Nessas, quase sempre o uso de janela de tempo é necessária, uma vez que se busca relação entre eventos que estão chegando, e aqueles que já foram recebidos. Os resultados mostram que o tamanho da janela tem sua influência direta no desempenho do engenho, sendo de fato necessário a existência do componente para melhor organizar o processamento. Janelas grandes consomem muita memória, sendo o gargalo da execução, e quando combinadas com alto *throughput* de dados, podem inviabilizar a aplicação do CEP. Para janela de tempo maiores que 120s, a execução na *Cloud* mostrou-se a melhor opção, uma vez que a latência de rede teve pequena influência no tempo de resposta. Já para janelas de tempos pequenas, a *mist* é uma alternativa de menor custo.

Os valores obtidos mostram que quando dobramos o tamanho da janela de tempo, o consumo de memória é elevado em mais de 30% para *fog* e *cloud*, e 40% para a *Mist*, pois um número maior de eventos é retido em *cache*. Logo, quando um novo evento é recebido, um maior número de expressões regulares é executado com objetivo em encontrar dados associados à consulta CEP. Esse número de comparações justifica o aumento no consumo de CPU.

As execuções na *Mist* apresentaram um alto incremento no tempo de resposta médio quando a janela de tempo é duplicada. Esse comportamento pode não ser aceitável para algumas aplicações de tempo real. Mesmo as execuções configuradas para trabalhar com 1ms, nota-se uma redução na taxa de coleta dos dados como consequência do aumento no processamento e no consumo de memória. Esse comportamento não foi observado nem na *fog* nem na *cloud*. Provavelmente, na *mist* foi preciso executar operações de *swap*, reduzindo a performance do engenho.

De fato, o tamanho da janela está diretamente ligado ao consumo de memória, que por sua vez aumenta o número de comparações realizadas pelo engenho, influenciando também no consumo de CPU. Mesmo tendo ciência que a consulta CEP é definida pela camada de aplicação, e tem papel crítico nos resultados nos *matches* e notificações do engenho, os valores observados mostram que uma variação da janela de tempo dinâmica e automática pode trazer bons resultados, reduzindo a carga do processamento e viabilizando a continuidade da análise dos dados na camada atual.

Independente do tamanho da janela, o consumo de CPU observado apresentou um padrão, como mostrado na Figura 41. Para uma determinada janela de tempo, não importa os dados contidos na memória do CEP, o processamento apresenta picos de consumo de CPU. Esse incremento ocorre devido a execução das expressões para detectar os padrões, principalmente aquelas que resultam valores que serão enviados aos dispositivos.

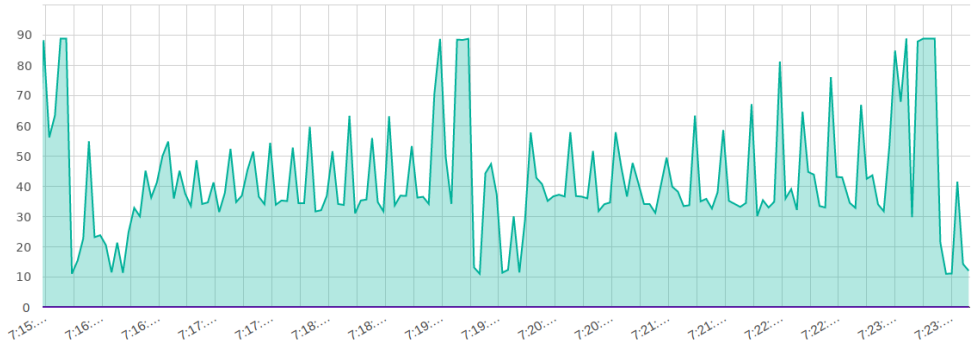
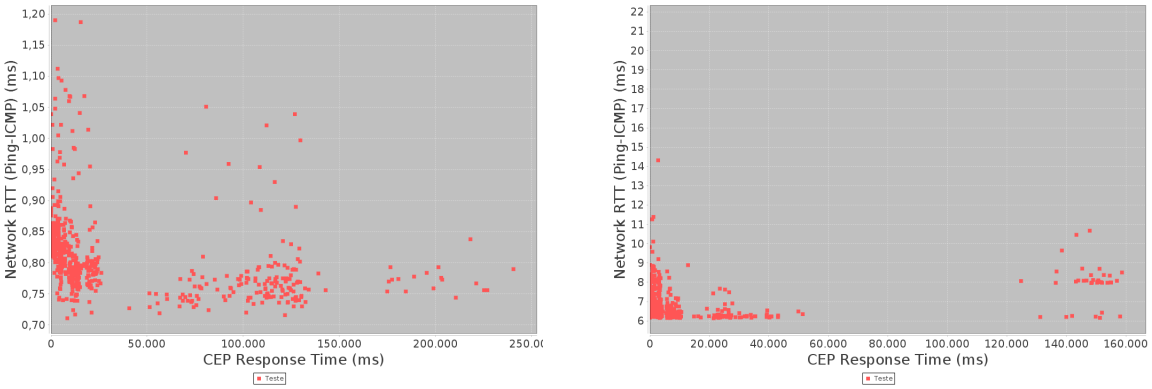


Figura 41 – Consumo de CPU - amostra de execução

6.3.3 Latência de Rede

Os resultados obtidos para as latências de rede foram baixos, tanto para a *Fog*, como para a *Cloud*, como mostradas na Figura 42.



(a) Fog (b) Cloud

Figura 42 – Latência de Rede para comunicação servidores *Fog* e *Cloud*

Os valores obtidos mostram o baixo impacto desta métrica no processamento, não influenciado no aumento do tempo de resposta, apesar de ser a principal justificativa para estudos que levam a análise dos dados para a borda do sistema. Em testes realizados em outras redes, especificamente em acessos à máquinas na *Amazon Web Services* (AWS)[2], a latência de rede média foi em torno de 200ms em redes ADSL e de 360ms na rede 4G (Vivo e OI). Esses valores podem impactar algumas consultas que já tem baixo tempo de respostas, mas continuam desprezíveis para os valores apresentados para altas taxas de dados.

Por fim, os resultados obtidos nos testes mostram que, apesar de existir três camadas bem definidas, as fronteiras entre elas são tênues. Um conjunto de parâmetros foram analisados, e eles, quando combinados, influenciam diretamente na decisão de onde executar o processamento para ter um melhor resultado. Por isso, o experimento a seguir apresenta o uso dos componentes de gerenciamento de políticas definido na arquitetura *GiTo*.

6.4 Etapa 2: Políticas em Uso

Esta etapa do experimento teve como objetivo integrar as três camadas, executando o cenário de teste (*Pattern*, 1ms, 600s). O objetivo foi manter o tempo de resposta abaixo de 5s. Os gráficos abaixo mostram percentual maior que 100%, pois a ferramenta *nmon* utilizada é baseado no comando *top* do *linux* com modo *IRIX* ativado. Logo, caso um dispositivo tenha 4 *cores*, seu consumo máximo de CPU é 400%.

6.4.1 Cenário 01: *Fog* Disponível

No primeiro cenário, o agente realiza dois *offloading*, como mostrado na Figura 43, onde o consumo de CPU nas três camadas é apresentado durante 23 minutos contínuos. No

[2] https://aws.amazon.com/

momento (1) da figura (em 1:21h), o agente 6275 detecta a violação da política, notifica o orquestrador de agentes. Esse, conecta-se ao servidor *Fog* através do canal de controle, que confirma o execução do *offloading*. Neste momento, o consumo de CPU do agente reduz e tempo de resposta diminui para próximo dos 100ms (Figura 44). No momento (2), próximo à 1:28h, outros agentes se conectam à *Fog*. Esse novo volume de dados aumenta o consumo de CPU do servidor, e prejudica o tempo de resposta para o agente 6275. No momento (3), em 1:30h, novamente o agente detecta outra violação da política de tempo, encerra a conexão atual, e realiza novo *offloading* com o servidor da *Cloud*, garantindo assim, o tempo de resposta baixo durante toda a execução. Esse mesmo cenário, quando executado apenas localmente, teve tempo de resposta médio em 68s. A arquitetura *GiTo* conseguiu manter o tempo baixo, mesmo quando comparado com apenas o fluxo do agente 6275 sendo processado na *Fog* (Tabela 6).

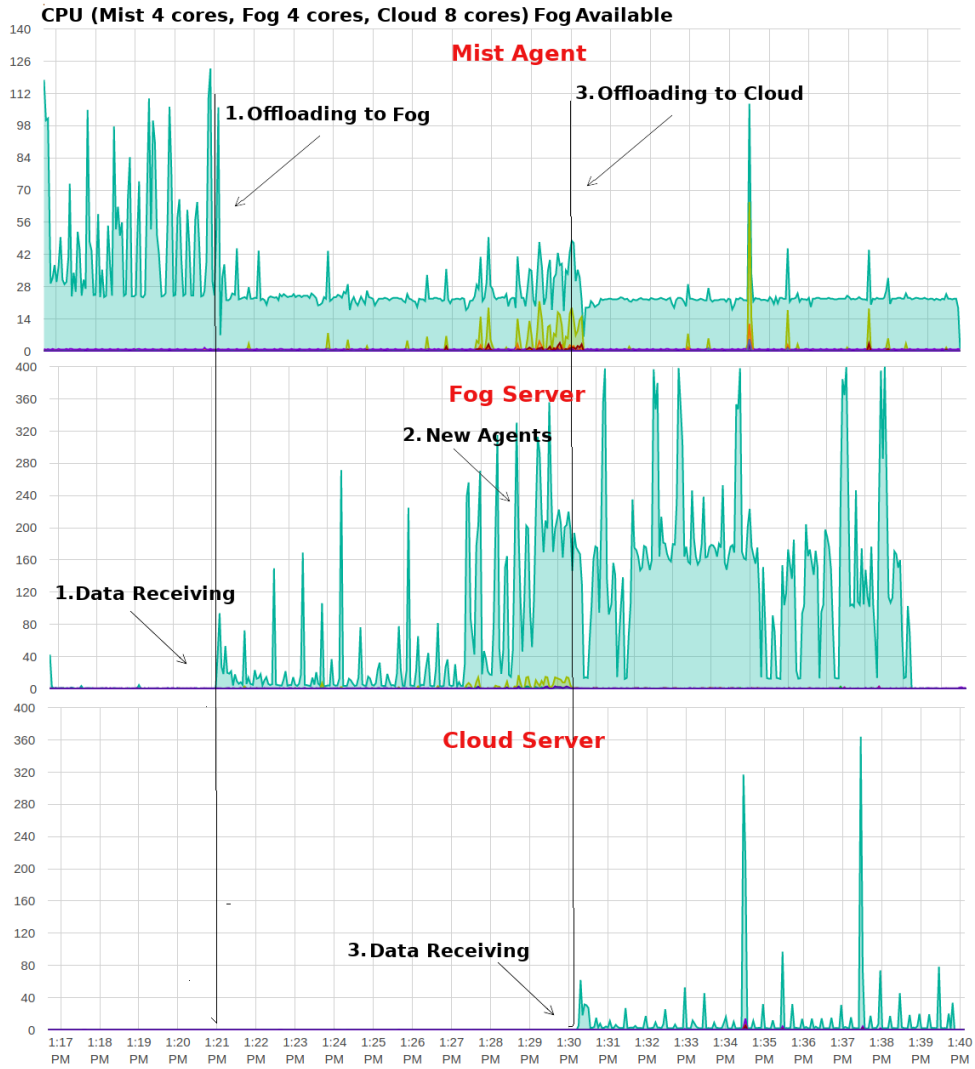


Figura 43 – Consumo de CPU - Camadas integradas - Fog Disponível

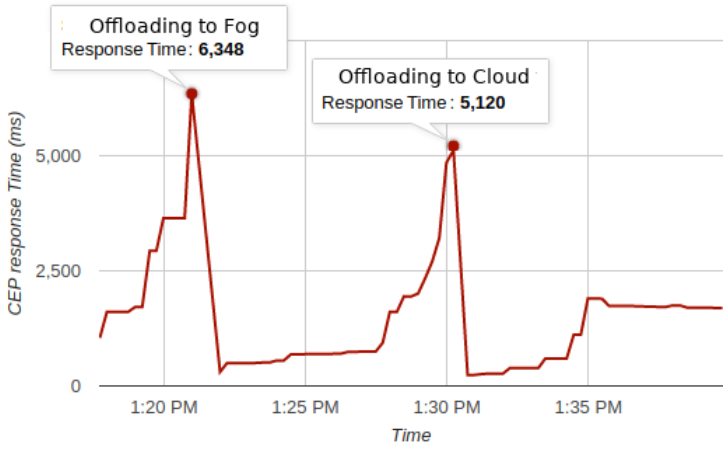


Figura 44 – Tempo de Resposta - Camadas integradas - Fog Disponível

6.4.2 Cenário 02: *Fog* Não Disponível

No cenário 2, o agente realiza apenas um *offloading*. Pois, na tentativa de se comunicar com a *Fog*, o servidor não aceitou os dados, forçando o agente a realizar a comunicação com a *Cloud* com intuito de manter o tempo de resposta abaixo dos 5s.

Os consumos de CPU das três camadas para um trecho da execução são apresentados na Figura 45, e os valores observados para o tempo de resposta na Figura 46. No momento 2:44:30h, o agente 6275 detecta a violação da política e estabelece uma conexão com o servidor *Fog* (canal de controle). Porém, durante o *handshaking*, o servidor nega a solicitação, uma vez que o seu tempo de resposta já está acima dos 5s e o seu consumo de CPU maior que 50% (função de tomada de decisão). Esse negação força o agente da *mist* a realizar o *offloading* para a *Cloud*. Nesse momento, o consumo de CPU do agente reduz, assim como seu tempo de resposta e o processamento é continuado no servidor remoto na nuvem.

Os resultados dessa etapa comprovam a necessidade de componentes que realizem a tomada de decisão de onde executar o processamento. A PoC, seguindo as definições da arquitetura *GiTo*, conseguiu detectar o aumento do tempo de resposta, e consequente violação da política. E através do *Agent Orquestration*, ações de *offloading* foram executadas.

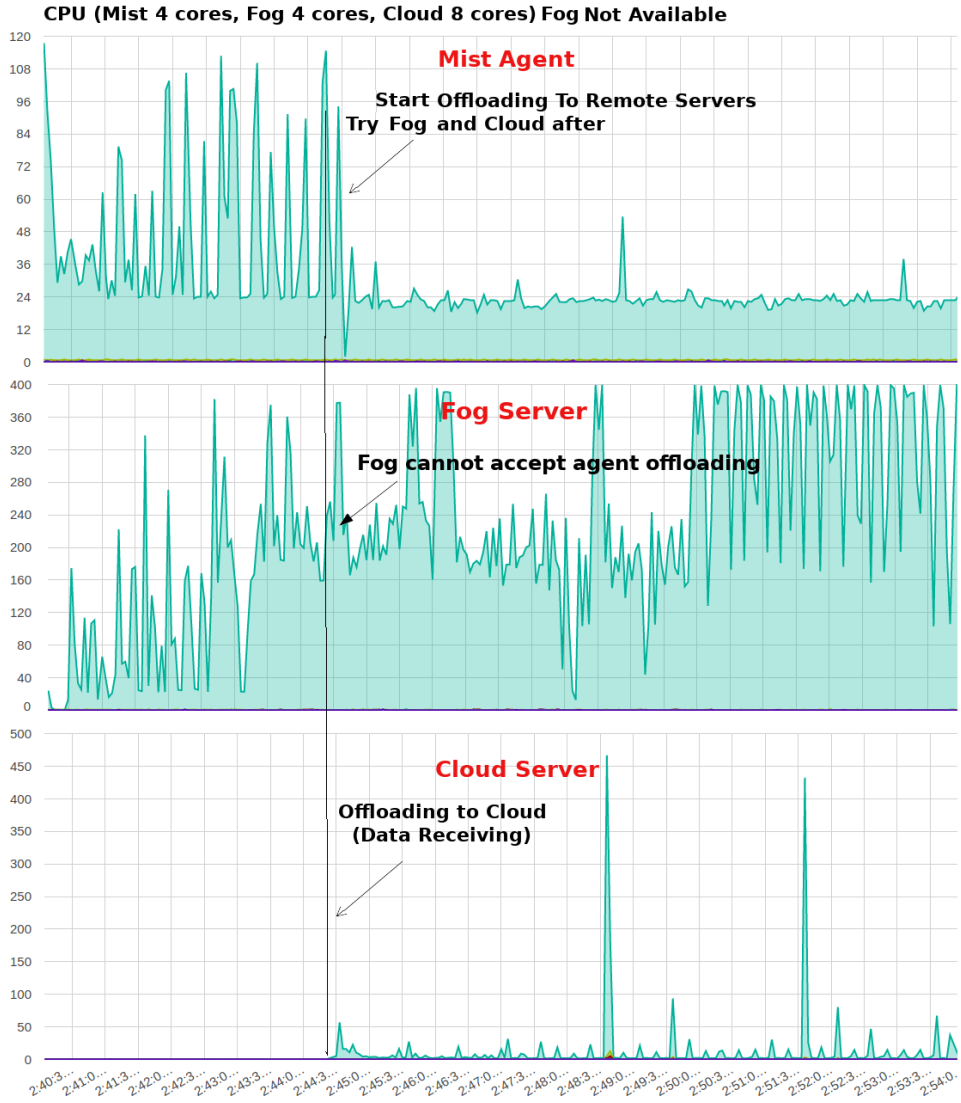


Figura 45 – Consumo de CPU - Camadas integradas - Fog não Disponível

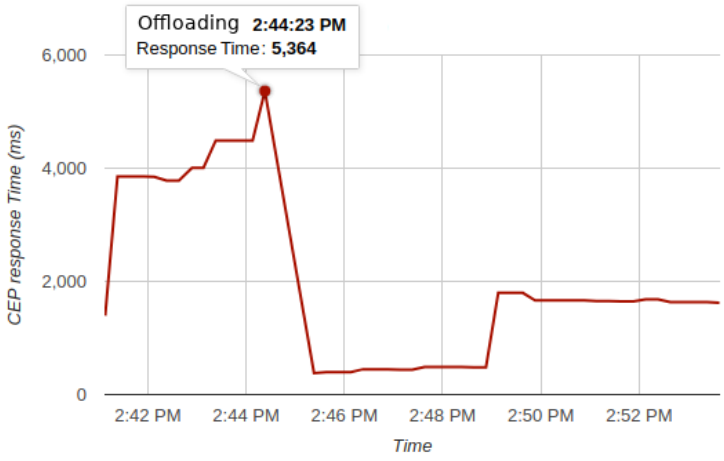


Figura 46 – Tempo de Resposta - Camadas integradas - Fog não Disponível

6.5 Impacto da arquitetura *GiTo* no Processamento dos Dados

Para analisar o impacto da arquitetura *GiTo* no CEP, foram realizados testes com a PoC sem o engenho CEP e componentes de políticas, e também com uma outra aplicação (*CEP App*) executando o mesmo processamento realizado no cenário de teste *(Mist, Pattern, 600s, 1ms, 1 fluxo)*, porém sem os componentes arquiteturais da *GiTo*.

A Tabela 8 mostra as médias dos valores obtidos, onde *CEP App* representa a aplicação independente da *GiTo*.

Tabela 8 – Execuções com e sem a arquitetura *GiTo*: *(Mist, Pattern, 600s, 1ms, 1 fluxo)*

	GiTo	CEP App	*GiTo* (sem CEP)	*GiTo* (sem CEP e Políticas)
Consumo CPU (%)	20,75	20,75	3,8	3,8
Consumo Memória (MB)	670	666,97	27,5	26

Quando a arquitetura foi executada com o componente CEP desativado, o consumo de CPU foi de 3,8% e de memória 27,5%. Esse mesmo valor foi observado quando os componentes de políticas e CEP foram desligados juntos. Para as execuções do CEP com e sem a arquitetura *GiTo*, os números obtidos são (praticamente) os mesmos, tanto para CPU como memória. Os resultados da tabela mostram a pequena influência da arquitetura *GiTo* no processamento dos eventos.

6.6 Considerações Finais

O experimento realizado valida a necessidade da proposta da arquitetura *GiTo*. Os resultados enfatizam que o melhor local para realizar CEP não depende apenas de um único fator; em outras palavras, a decisão de onde executar o processamento está relacionada com as características e necessidades da aplicação que estiver sendo executada no dispositivo da IoT, assim como o ambiente no qual este está inserido. Ou seja, do ponto de vista da análise dos dados, as fronteiras entre as camadas não são tão claras e objetivas. Por exemplo, um mesmo dispositivo pode atuar de diferentes maneiras. No exemplo realizado, um táxi aparece na camada 1 (*mist*), provendo dados que serão analisados. Esse mesmo táxi, em um outro contexto, numa aplicação distinta, pode ser definido como um servidor *Fog*, capaz de processar dados dos seus diversos sensores associados aos componentes eletrônicos presentes nos carros atuais. Independente da aplicação, os valores observados mostram que um dispositivo que recebe dados com um *throughput* baixo, poderá processá-los e gerar uma resposta sem a necessidade de conexão com servidores externos, reduzindo assim o tempo de resposta. Caso contrário, será preciso compartilhar os dados com um servidor remoto.

A *Fog*, que geralmente é formada por *edge servers* ou mesmo roteadores, mostrou-se limitada para regiões densas de dispositivos IoT, pois, como muitos dados são compartilhados, o consumo de memória tende a ser elevado. Porém, pode ser utilizada como um *gateway*, consolidando dados e repassando para servidores mais robustos. Já a *Mist*, pode ser uma solução interessante quando o dispositivo da IoT precisa executar consultas mais simples. Já com uma combinação de altas taxas de transferência de dados com complexidade de consulta alta, incluindo janela de tempo e o atributo *every*, o *offloading* para a *Cloud* será preciso, assim como um bom investimento em infraestrutura, viabilizando o processamento com baixo tempo de resposta. Por fim, o experimento mostrou que, para a carga de dados analisada na aplicação cenário, a memória se apresentou como gargalo principal do processamento.

A arquitetura *GiTo* apresentou resultado satisfatório para o contexto da WoT. O consumo inserido pelos seus componentes não influenciarão o resultado, tornando-a viável para dispositivos limitados. A utilização de protocolos já difundidos mostraram que é possível rápida integração com outras camadas e entidades, uma vez que as principais linguagens de programação já dão suporte à padrões da Web. Os resultados sugerem um novo componente na arquitetura, responsável por balancear a carga recebida através de agregação, filtro ou mesmo redução do *throughput* de dados produzidos pela *thing*. Por fim, fica evidente a necessidade de ter um processamento dinâmico que permita decidir em tempo de execução o melhor local para realizar a análise dos dados, enfatizando a utilidade e aplicação da arquitetura *GiTo*.

7 CONCLUSÃO

Este capítulo traz um resumo do trabalho apresentado, mostrando como os objetivos propostos foram atingidos e quais as contribuições desta pesquisa para a área de CEP e IoT. Em seguida, são apontados os principais problemas encontrados ao longo do desenvolvimento do trabalho. Finalmente, as possibilidades de melhoria, assim como novas linhas de investigação dentro do mesmo tema são apresentadas.

Como visto nos capítulos iniciais, a IoT vem criando novas oportunidades nas mais diversas áreas do mercado, contudo, traz consigo diversos desafios. Dentre eles, a necessidade de análise dos dados produzidos numa velocidade cada vez maior e a comunicação entre os dispositivos se apresentam como barreiras que ainda precisam ser superadas. Estudos mostram a migração da computação centralizada do paradigma *Cloud Computing* para a borda da rede, em busca da pulverização do processamento cada vez mais distribuído. Esses já alcançam servidores intermediários da computação na *Fog*, e agora chegam à *Mist*, onde os próprios dispositivos que produzem os dados na IoT utilizam seu poder computacional para executar funcionalidades em cima dos dados, visando, entre outros, uma redução nos tempos de resposta das operações.

7.1 Contribuições

Este trabalho mostrou que é possível executar processamento de eventos complexos nos dispositivos da IoT, validando a hipótese inicialmente estabelecida. Porém, nem todos os tipos de análise dos dados são possíveis, devido ao consumo elevado de recursos dos dispositivos em algumas situações. Para isso, foi apresentada a arquitetura *GiTo*, que tem como objetivo coordenar o CEP através de um conjunto de políticas, ora executando o processamento na camada *Mist*, ora realizando o *offloading* computacional para servidores nas camadas *Fog* e *Cloud*. Essa arquitetura é direcionado aos dispositivos de IoT do grupo J da classificação do IETF.

Inicialmente, a arquitetura foi exercitada através de uma análise de desempenho, onde vários cenários de testes foram executados separadamente nas camadas. O intuito foi avaliar os limites do dispositivo para diferentes situações. Observaram-se elevados tempo de resposta do processamento quando o volume de dados é alto. Posteriormente, as três camadas foram integradas através da arquitetura *GiTo*, onde esta ficou responsável por monitorar a execução, evitando assim tempo de resposta das operações acima do esperado.

Os resultados mostram que não é recomendado realizar CEP em uma única camada, mesmo que seja a *Cloud*. E que os próprios dispositivos podem colaborar no processamento. Ou seja, existe a necessidade do processamento dinâmico na IoT, onde os dispositivos precisam de autonomia e infraestrutura para tomar decisões com base nas car-

acterísticas coletadas em tempo real do ambiente que estão inseridos. O gerenciamento de políticas consome poucos recursos da máquina, e os valores observados mostram seu baixo custo. Logo, o uso de políticas para coordenação de CEP se mostrou eficiente no dinamismo do processamento, reduzindo o consumo dos recursos quando comparado às execuções separadas. Por fim, a principal contribuição desta tese é arquitetura *GiTo*, capaz de realizar o processamento de forma distribuída e dinâmica. Além disso, o experimento realizado apresentou resultados importantes, mostrando que é possível executar CEP, porém com limites em determinadas situações.

7.2 Limitações

O trabalho apresenta o potencial da solução adotada, contudo, mostra também pontos de melhorias e limitações. A lista abaixo enumera o conjunto delas.

- O estado interno do CEP e os dados presentes no *buffer* interno não são transferidos quando o *offloading* é realizado. Para isso, é necessário entender a construção dos engenhos CEP (no caso o *Esper*) para extrair os dados de suas estruturas internas e depois enviar pela rede;
- Apesar da arquitetura *GiTo* contemplar o *offloading* horizontal, execuções testes não foram realizadas na camada *Mist*. É válido mencionar que, embora esta seja apresentada como uma limitação, uma vez o sub-componente *Agente Discovery* do *Communication Manager* esteja funcional, a colaboração horizontal muito se assemelha à vertical, pois protocolos da *Web* foram utilizados justamente para minimizar o problema de interoperabilidade.
- As comparações das execução nas camadas não consideram questões envolvendo virtualização dos servidores remotos, nem detalhes do conjunto de instruções de cada dispositivo.

7.3 Trabalhos Futuros

De acordo com os resultados obtidos, pode-se esperar que a continuação deste trabalho seja de grande contribuição para a pesquisa na área de processamento de dados em tempo real. Abaixo seguem algumas possíveis extensões da pesquisa realizada e da arquitetura *GiTo*.

- Realização de testes e análises considerando descoberta de agentes na *Mist*, assim como colaboração de processamento entre eles;
- Classificação de políticas de coordenação após realização de análises envolvendo também outras aplicações cenários;

- Recomendação de políticas de forma automática, através do aprendizado adquirido nos processamentos já realizados;
- Analisar a eficiência da arquitetura *GiTo* com diferentes aplicações ao mesmo tempo;
- Distribuição dos dados e do processamento com base na divisão semântica das consultas CEP.
- Realizar experimentos coletando e analisando outras informações contextuais, como consumo de energia e mobilidade dos dispositivos da IoT.

REFERÊNCIAS

ABOWD, G. D.; DEY, A. K.; BROWN, P. J.; DAVIES, N.; SMITH, M.; STEGGLES, P. Towards a better understanding of context and context-awareness. In: *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing*. London, UK, UK: Springer-Verlag, 1999. (HUC '99), p. 304–307. ISBN 3-540-66550-1. Disponível em: <http://dl.acm.org/citation.cfm?id=647985.743843>.

AGRAWAL SERAPHIN CALO, K. w. L. J. L. D. V. D. *Policy Definition and Usage Scenarios for Self-Managing Systems*. [S.l.]: IBM Press, 2008. v. 1.

ANHEMBI. *Metodologia de Pesquisa Científica - O Método Hipotético-Dedutivo*. 2015. Disponível em: <http://www2.anhembi.br/html/ead01/metodologia_pesq_cientifica_80/lu04/lo4/index.htm>.

ANICIC, D.; RUDOLPH, S.; FODOR, P.; STOJANOVIC, N. Stream reasoning and complex event processing in etalis. *Semantic Web - On linked spatiotemporal data and geo-ontologies*, IOS Press, Amsterdam, The Netherlands, The Netherlands, v. 3, n. 4, p. 397–407, out. 2012. ISSN 1570-0844. Disponível em: <http://dl.acm.org/citation.cfm?id=2590208.2590214>.

ARKIAN, H. R.; DIYANAT, A.; POURKHALILI, A. Mist: Fog-based data analytics scheme with cost-efficient resource provisioning for iot crowdsensing applications. *Journal of Network and Computer Applications*, Elsevier, v. 82, p. 152 – 165, 2017. ISSN 1084-8045.

ATZORI, L.; IERA, A.; MORABITO, G. The Internet of Things: A survey. *Computer Networks: The International Journal of Computer and Telecommunications*, Elsevier B.V., v. 54, n. 15, p. 2787–2805, 2010. ISSN 13891286. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/S1389128610001568>.

BANDARA, M. N.; RANASINGHE, R. M.; ARACHCHI, R. W. M.; SOMATHILAKA, C. G.; PERERA, S.; WIMALASURIYA, D. C. A complex event processing toolkit for detecting technical chart patterns. In: *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. [S.l.]: IEEE, 2015. p. 547–556.

BAPTISTA, G. L. B.; RORIZ, M.; VASCONCELOS, R.; OLIVIERI, B.; VASCONCELOS, I.; ENDLER, M. On-line Detection of Collective Mobility Patterns through Distributed Complex Event Processing. *Monografias em Ciência da Computação*, PUC-Rio de Janeiro, v. 13, n. DECEMBER, 2013.

BARROS, T. *CMF: um framework multi-plataforma para desenvolvimento de aplicações para dispositivos móveis*. Dissertação (Mestrado), UFPE, 2007.

BENNACEUR, A.; FRANCE, R.; TAMBURRELLI, G.; VOGEL, T.; MOSTERMAN, P. J.; CAZZOLA, W.; COSTA, F. M.; PIERANTONIO, A.; TICHY, M.; AKŞIT, M.; EMMANUELSON, P.; GANG, H.; GEORGANTAS, N.; REDLICH, D. Mechanisms for leveraging models at runtime in self-adaptive software. In: BENCOMO, N.; FRANCE, R.; CHENG, B. H. C.; ASSMANN, U. (Ed.). *Models@run.time: Foundations, Applications, and Roadmaps*. Cham: Springer International Publishing, 2014. p. 19–46. ISBN 978-3-319-08915-7. Disponível em: <https://doi.org/10.1007/978-3-319-08915-7_2>.

BLACKSTOCK, M.; LEA, R. Toward interoperability in a web of things. In: *Proceedings of the 2013 ACM Conference on Pervasive and Ubiquitous Computing Adjunct Publication*. New York, NY, USA: ACM, 2013. (UbiComp '13 Adjunct), p. 1565–1574. ISBN 978-1-4503-2215-7. Disponível em: <http://doi.acm.org/10.1145/2494091.2497591>.

BONOMI, F.; MILITO, R.; NATARAJAN, P.; ZHU, J. Fog Computing: A Platform for Internet of Things and Analytics. *Studies in Computational Intelligence*, Springer, v. 546, p. 169–186, 2014. ISSN 1860949X. Disponível em: <http://link.springer.com/10.1007/978-3-319-05029-4>.

BONOMI, F.; MILITO, R.; ZHU, J.; ADDEPALLI, S. Fog Computing and Its Role in the Internet of Things. *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, p. 13–16, 2012. ISSN 978-1-4503-1519-7. Disponível em: <http://doi.acm.org/10.1145/2342509.2342513$\protect\delimiter"026E30F$npapers2://publication/doi/10.1145/2342509.2342513>.

BORGIA, E. The internet of things vision: Key features, applications and open issues. *Computer Communications*, v. 54, p. 1 – 31, 2014. ISSN 0140-3664. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0140366414003168>.

BORMANN M. ERSUE, A. K. C. G. C. *Terminology for Constrained-Node Networks*. [S.l.], 2018.

BREZILLON, P. Context in artificial intelligence: I. a survey of the literature. *Computers and artificial intelligence*, TRANSLIBRIS, v. 18, p. 321–340, 1999.

BRUIN, H. de; VLIET, H. van. Top-down composition of software architectures. In: *Proceedings Ninth Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*. [S.l.]: IEEE, 2002. p. 147–156.

BRUN, Y.; SERUGENDO, G. M.; GACEK, C.; GIESE, H.; KIENLE, H.; LITOIU, M.; MüLLER, H.; PEZZè, M.; SHAW, M. *Software Engineering for Self-Adaptive Systems*. Berlin, Heidelberg: Springer-Verlag, 2009. 48–70 p. Disponível em: <http://dx.doi.org/10.1007/978-3-642-02161-9_3>.

BRöRING, A.; SCHMID, S.; SCHINDHELM, C.; KHELIL, A.; KäBISCH, S.; KRAMER, D.; PHUOC, D. L.; MITIC, J.; ANICIC, D.; TENIENTE, E. Enabling iot ecosystems through platform interoperability. *IEEE Software*, v. 34, n. 1, p. 54–61, Jan 2017. ISSN 0740-7459.

CARDELLINI, V.; PRESTI, F. L.; NARDELLI, M.; RUSSO, G. R. Decentralized self-adaptation for elastic data stream processing. *Future Generation Computer Systems*, v. 87, p. 171 – 185, 2018. ISSN 0167-739X. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0167739X17326821>.

CARDOSO, A. M. da S.; LOPES, R. F.; TELES, A. S.; MAGALHãES, F. B. V. Poster abstract: Real-time ddos detection based on complex event processing for iot. In: *2018 IEEE/ACM Third International Conference on Internet-of-Things Design and Implementation (IoTDI)*. [S.l.]: IEEE, 2018. p. 273–274.

CASTRO, M. *Internet das coisas impulsionará o setor de logística*. 2015. Teste. Disponível em: <http://www.painellogistico.com.br/internet-das-coisas-impulsionara-o-setor-de-logistica/>.

CHAKRAVARTHY, S.; JIANG, Q. *Stream data processing: a quality of service perspective: modeling, scheduling, load shedding, and complex event processing*. [S.l.]: Springer Science & Business Media, 2009. v. 36.

CHOOCHOTKAEW, S.; YAMAGUCHI, H.; HIGASHINO, T.; SHIBUYA, M.; HASEGAWA, T. Edgecep: Fully-distributed complex event processing on iot edges. In: *2017 13th International Conference on Distributed Computing in Sensor Systems (DCOSS)*. [S.l.]: IEEE, 2017. p. 121–129. ISSN 2325-2944.

CISCO. *The Internet of Things*. 2014. Disponível em: <http://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/iot-aag.pdf>.

CQL, O. *Oracle CEP CQL Language Reference*. 2016. Disponível em: <https://docs.oracle.com/cd/E16764_01/doc.1111/e12048/intro.htm>.

CRACIUNESCU, R.; MIHOVSKA, A.; MIHAYLOV, M.; KYRIAZAKOS, S.; PRASAD, R.; HALUNGA, S. Implementation of fog computing for reliable e-health applications. In: *2015 49th Asilomar Conference on Signals, Systems and Computers*. [S.l.]: IEEE, 2015. p. 459–463. ISSN 1058-6393.

CRESPI, V.; GALSTYAN, A.; LERMAN, K. Top-down vs bottom-up methodologies in multi-agent system design. *Autonomous Robots*, v. 24, n. 3, p. 303–313, Apr 2008. ISSN 1573-7527. Disponível em: <https://doi.org/10.1007/s10514-007-9080-5>.

CUGOLA, G.; MARGARA, A. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 44, n. 3, p. 15:1–15:62, jun. 2012. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/2187671.2187677>.

CUGOLA, G.; MARGARA, A.; PEZZè, M.; PRADELLA, M. Efficient analysis of event processing applications. In: *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. New York, NY, USA: ACM, 2015. (DEBS '15), p. 10–21. ISBN 978-1-4503-3286-6. Disponível em: <http://doi.acm.org/10.1145/2675743.2771834>.

DAUTOV, R.; DISTEFANO, S. Three-level hierarchical data fusion through the iot, edge, and cloud computing. In: *Proceedings of the 1st International Conference on Internet of Things and Machine Learning*. New York, NY, USA: ACM, 2017. (IML '17), p. 1:1–1:5. ISBN 978-1-4503-5243-7. Disponível em: <http://doi.acm.org/10.1145/3109761.3158388>.

DEY, A. K. Understanding and using context. *Personal Ubiquitous Comput.*, Springer-Verlag, London, UK, UK, v. 5, n. 1, p. 4–7, jan. 2001. ISSN 1617-4909. Disponível em: <http://dx.doi.org/10.1007/s007790170019>.

ESPER. *Esper EPL - Event Processing Language*. 2016. Disponível em: <http://www.espertech.com/esper/release-5.2.0/esper-reference/html/epl_clauses.html>.

ETZION, O.; NIBLETT, P. *Event processing in action*. [S.l.]: Manning Publications Co., 2010. ISBN 978-1935182214.

EUGSTER, P. T.; FELBER, P. A.; GUERRAOUI, R.; KERMARREC, A.-M. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, ACM, New York, NY, USA, v. 35, n. 2, p. 114–131, jun. 2003. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/857076.857078>.

FETTE, I.; MELNIKOV, A. Rfc 6455-the websocket protocol.(2011). *URL https://tools. ietf. org/html/rfc6455*, 2016.

FRANCIS, T.; MADHIAJAGAN, M. A comparison of cloud execution mechanisms: Fog, edge and clone cloud computing. *Proceeding of the Electrical Engineering Computer Science and Informatics*, v. 4, p. 446–450, 2017.

GAMMA, E. *Design patterns: elements of reusable object-oriented software*. [S.l.]: Pearson Education India, 1995.

GARLAN, D.; SCHMERL, B.; STEENKISTE, P. An architecture for coordinating multiple self-management systems. *Proceedings. Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004)*, p. 243–252, 2004. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1310707>.

GARTNER. *Gartner Hype Cycle - 2015*. 2015. Disponível em: <http://www.gartner.com/newsroom/id/3114217>.

GOMES, R.; BOULOUKAKIS, G.; COSTA, F.; GEORGANTAS, N.; ROCHA, R. D. QoS-Aware Resource Allocation for Mobile IoT Pub/Sub Systems. In: *2018 International Conference on Internet of Things (ICIOT)*. Seattle, United States: Springer, 2018. Disponível em: <https://hal.inria.fr/hal-01797933>.

GROSS, T.; PRINZ, W. Awareness in context. In: KUUTTI, K.; KARSTEN, E. H.; FITZPATRICK, G.; DOURISH, P.; SCHMIDT, K. (Ed.). *ECSCW 2003*. Dordrecht: Springer Netherlands, 2003. p. 295–314. ISBN 978-94-010-0068-0.

GUAN, G. Qoe-aware edge computing for complex iot event processing. 2017.

GUBBI, J.; BUYYA, R.; MARUSIC, S.; PALANISWAMI, M. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, Elsevier B.V., v. 29, n. 7, p. 1645–1660, 2013. ISSN 0167739X. Disponível em: <http://dx.doi.org/10.1016/j.future.2013.01.010>.

GUINARD, D.; TRIFA, V.; WILDE, E. A resource oriented architecture for the web of things. p. 1–8, Nov 2010.

HAJIBABA, M.; GORGIN, S. A Review on Modern Distributed Computing Paradigms : Cloud Computing , Jungle Computing. *Journal Of Computing & Information Technology*, v. 22, n. 2, p. 69–84, 2014. ISSN 1330-1136.

HONG, K.; LILLETHUN, D. Mobile fog: a programming model for large-scale applications on the internet of things. *Proceedings of the second ACM SIGCOMM Workshop on Mobile Cloud Computing*, p. 15–20, 2013. Disponível em: <http://dl.acm.org/citation.cfm?id=2491270>.

HUNT, P.; KONAR, M.; JUNQUEIRA, F.; REED, B. ZooKeeper: Wait-free Coordination for Internet-scale Systems. *USENIX Annual Technical ...*, v. 8, p. 11–11, 2010. ISSN 0210945x. Disponível em: <http://portal.acm.org/citation.cfm?id=1855851$\protect\delimiter"026E30F$nhttps://www.usenix.org/event/usenix10/tech/full{_}papers/Hunt.>

INTEL. *A Guide to the Internet of Things*. 2018. Disponível em: <https://www.intel.com/content/www/us/en/internet-of-things/infographics/guide-to-iot.html/>.

IQBAL, M. H.; SOOMRO, T. R. Big data analysis: Apache storm perspective. *International journal of computer trends and technology*, v. 19, n. 1, p. 9–14, 2015.

JAIN, R. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. [S.l.]: John Wiley & Sons, 1990.

JOHNSON, R. E. Frameworks = (components + patterns). *Commun. ACM*, ACM, New York, NY, USA, v. 40, n. 10, p. 39–42, out. 1997. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/262793.262799>.

JOVANOVIC, Z.; BACEVIC, R.; MARKOVIC, R.; RANDJIC, S. Android application for observing data streams from built-in sensors using rxjava. In: *Telecommunications Forum Telfor (TELFOR), 2015 23rd*. [S.l.]: IEEE, 2015. p. 918–921.

JUN, C.; CHI, C. Design of complex event-processing ids in internet of things. In: *2014 Sixth International Conference on Measuring Technology and Mechatronics Automation*. [S.l.]: IEEE, 2014. p. 226–229. ISSN 2157-1473.

JUNG, H. S.; YOON, C. S.; LEE, Y. W.; PARK, J. W.; YUN, C. H. Processing iot data with cloud computing for smart cities. *International Journal of Web Applications*, v. 9, n. 3, 2017.

KAMILARIS, A.; PITSILLIDES, A.; PRENAFETA-BOLD, F. X.; ALI, M. I. A web of things based eco-system for urban computing - towards smarter cities. In: *2017 24th International Conference on Telecommunications (ICT)*. [S.l.]: IEEE, 2017. p. 1–7.

KAMISIńSKI, P.; GOEBEL, V.; PLAGEMANN, T. A reconfigurable distributed cep middleware for diverse mobility scenarios. IEEE, p. 615–620, March 2013.

KARAGIANNIS, V.; CHATZIMISIOS, P.; VÁZQUEZ-GALLEGO, F.; ALONSO-ZARATE, J. A Survey on Application Layer Protocols for the Internet of Things. *Transaction on IoT and Cloud Computing (TICC), 2015*, v. 1, n. 1, jan. 2015. Disponível em: <http://icas-pub.org/ojs/index.php/ticc/article/view/47>.

KELLY, S. D. T.; SURYADEVARA, N. K.; MUKHOPADHYAY, S. C. Towards the implementation of iot for environmental condition monitoring in homes. *IEEE Sensors Journal*, v. 13, n. 10, p. 3846–3853, Oct 2013. ISSN 1530-437X.

LIU, X.; DASTJERDI, A. V.; CALHEIROS, R. N.; QU, C.; BUYYA, R. A stepwise auto-profiling method for performance optimization of streaming applications. *ACM Trans. Auton. Adapt. Syst.*, ACM, New York, NY, USA, v. 12, n. 4, p. 24:1–24:33, nov. 2017. ISSN 1556-4665. Disponível em: <http://doi.acm.org/10.1145/3132618>.

LOGWEB. *Novo centro de inovação da DHL Americas projeta futuro da logística*. 2018. Disponível em: <http://www.logweb.com.br/novo-centro-de-inovacao-da-dhl-americas-projeta-futuro-da-logistica/>.

LUCKHAM, D. *The power of events: An introduction to complex event processing in distributed enterprise systems*. [S.l.]: Springer, 2008.

LUETH, K. L. *State of the IoT 2018: Number of IoT devices now at 7B – Market accelerating*. 2018. Disponível em: <https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/>.

LUTHRA, M.; KOLDEHOFE, B.; STEINMETZ, R. Adaptive complex event processing over fog-cloud infrastructure supporting transitions. *KuVS-Fachgespräch Fog Computing 2018*, p. 17, 2018.

MADSEN, K. G. S.; ZHOU, Y.; CAO, J. Integrative dynamic reconfiguration in a parallel stream processing engine. In: *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. [S.l.]: IEEE, 2017. p. 227–230. ISSN 2375-026X.

MCCARTHY, D.; DAYAL, U. The architecture of an active database management system. *SIGMOD Rec*, ACM, v. 18, n. 2, p. 215–224, jun. 1989. ISSN 0163-5808. Disponível em: <http://doi.acm.org/10.1145/66926.66946>.

MCNEELY, C.; HAHM, J.-o. The big (data) bang: Policy, prospects, and challenges. *Review of Policy Research*, v. 31, n. 4, p. 304–310, 2014. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1111/ropr.12082>.

MENDES, A.; LOSS, S.; CAVALCANTE, E.; LOPES, F.; BATISTA, T. Mandala: An agent-based platform to support interoperability in systems-of-systems. In: *Proceedings of the 6th International Workshop on Software Engineering for Systems-of-Systems*. New York, NY, USA: ACM, 2018. (SESoS '18), p. 21–28. ISBN 978-1-4503-5747-0. Disponível em: <http://doi.acm.org/10.1145/3194754.3194757>.

MIORANDI, D.; SICARI, S.; De Pellegrini, F.; CHLAMTAC, I. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, Elsevier B.V., v. 10, n. 7, p. 1497–1516, 2012. ISSN 15708705. Disponível em: <http://dx.doi.org/10.1016/j.adhoc.2012.02.016>.

MIRANDA, J.; M??KITALO, N.; GARCIA-ALONSO, J.; BERROCAL, J.; MIKKONEN, T.; CANAL, C.; MURILLO, J. M. From the Internet of Things to the Internet of People. *IEEE Internet Computing*, v. 19, n. 2, p. 40–47, 2015. ISSN 10897801.

MOORE, B.; ELLESSON, E. Rfc 3060- policy core information model – version 1 specification. *https://tools.ietf.org/html/rfc3060*, 2001.

MORENO, N.; BERTOA, M. F.; BARQUERO, G.; BURGUEÑO, L.; TROYA, J.; GARCÍA-LÓPEZ, A.; VALLECILLO, A. Managing uncertain complex events in web of things applications. Springer International Publishing, Cham, p. 349–357, 2018.

MOUSSALLI, R.; SRIVATSA, M.; ASAAD, S. Fast and flexible conversion of geohash codes to and from latitude/longitude coordinates. In: *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. [S.l.]: IEEE, 2015. p. 179–186.

MUNIR, A.; KANSAKAR, P.; KHAN, S. U. Ifciot: Integrated fog cloud iot: A novel architectural paradigm for the future internet of things. *IEEE Consumer Electronics Magazine*, v. 6, n. 3, p. 74–82, July 2017. ISSN 2162-2248.

NALLAPERUMA, D.; SILVA, D. D.; ALAHAKOON, D.; YU, X. A cognitive data stream mining technique for context-aware iot systems. In: *IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society*. [S.l.]: IEEE, 2017. p. 4777–4782.

NEUMANN, M. A.; BACH, C. T.; MICLAUS, A.; RIEDEL, T.; BEIGL, M. Always-on web of things infrastructure using dynamic software updating. In: *Proceedings of the Seventh International Workshop on the Web of Things*. New York, NY, USA: ACM, 2016. (WoT '16), p. 5–10. ISBN 978-1-4503-4874-4. Disponível em: <http://doi.acm.org/10.1145/3017995.3017997>.

NEWMAN, I.; BENZ, C. R. *Qualitative-quantitative research methodology: Exploring the interactive continuum*. [S.l.]: SIU Press, 1998.

NIEMEYER, G. *Geohash*. 2008.

NIEUWPOORT, R. V. van; KIELMANN, T.; BAL, H. E. Efficient load balancing for wide-area divide-and-conquer applications. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 36, n. 7, p. 34–43, jun. 2001. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/568014.379563>.

ORACLE. *Java™ EE at a Glance*. 2018. Disponível em: <https://www.oracle.com/technetwork/java/javaee/overview/index.html>.

OSANAIYE, O.; CHEN, S.; YAN, Z.; LU, R.; CHOO, K. R.; DLODLO, M. From cloud to fog computing: A review and a conceptual live vm migration framework. *IEEE Access*, v. 5, p. 8284–8300, 2017. ISSN 2169-3536.

OTTENWäLDER, B.; KOLDEHOFE, B.; ROTHERMEL, K.; HONG, K.; LILLETHUN, D.; RAMACHANDRAN, U. Mcep: A mobility-aware complex event processing system. *ACM Trans. Internet Technol.*, ACM, New York, NY, USA, v. 14, n. 1, p. 6:1–6:24, ago. 2014. ISSN 1533-5399. Disponível em: <http://doi.acm.org/10.1145/2633688>.

PARSONS, D.; RASHID, A.; SPECK, A.; TELEA, A. A "framework" for object oriented frameworks design. In: *Proceedings Technology of Object-Oriented Languages and Systems. TOOLS 29 (Cat. No.PR00275)*. [S.l.]: IEEE, 1999. p. 141–151.

PEDIADITAKIS, D.; GOPALAN, A.; DULAY, N.; SLOMAN, M.; LODGE, T. Home network management policies: Putting the user in the loop. In: *2012 IEEE International Symposium on Policies for Distributed Systems and Networks*. [S.l.]: IEEE, 2012. p. 9–16.

PIMENTEL, V.; NICKERSON, B. G. Communicating and displaying real-time data with websocket. *IEEE Internet Computing*, v. 16, n. 4, p. 45–53, July 2012. ISSN 1089-7801.

PIRES, P. F.; DELICATO, F. C.; BATISTA, T.; BARROS, T.; CAVALCANTE, E.; PITANGA, M. Plataformas para a Internet das Coisas. *Anais do Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, p. 110–169, 2015.

PREDEN, J. S.; TAMMEMäE, K.; JANTSCH, A.; LEIER, M.; RIID, A.; CALIS, E. The benefits of self-awareness and attention in fog and mist computing. *Computer*, v. 48, n. 7, p. 37–45, July 2015. ISSN 0018-9162.

PUSCHMANN, D.; BARNAGHI, P.; TAFAZOLLI, R. Adaptive clustering for dynamic iot data streams. *IEEE Internet of Things Journal*, v. 4, n. 1, p. 64–74, Feb 2017. ISSN 2327-4662.

RASPBERRYPI. *Raspberry Pi - Teach, Learn, and Make with Raspberry Pi*. 2016. Disponível em: <https://www.raspberrypi.org/>.

SCHILLING, B.; KOLDEHOFE, B.; PLETAT, U.; ROTHERMEL, K. Distributed heterogeneous event processing: Enhancing scalability and interoperability of cep in an industrial context. ACM, New York, NY, USA, p. 150–159, 2010. Disponível em: <http://doi.acm.org/10.1145/1827418.1827453>.

SHIH, P.-J. From the cloud to the edge: The technical characteristics and application scenarios of fog computing. *International Journal of Automation and Smart Technology*, v. 8, p. 61–64, 06 2018.

SKARLAT, O.; BACHMANN, K.; SCHULTE, S. Fogframe: Iot service deployment and execution in the fog. *KuVS-Fachgespräch Fog Computing 2018*, p. 5, 2018.

SOMMERVILLE, I.; ARAKAKI, R.; MELNIKOFF, S. S. S. *Engenharia de software*. [S.l.]: Pearson Prentice Hall, 2008.

SOTO, J. A. C.; JENTSCH, M.; PREUVENEERS, D.; ILIE-ZUDOR, E. Ceml: Mixing and moving complex event processing and machine learning to the edge of the network for iot applications. In: *Proceedings of the 6th International Conference on the Internet of Things*. New York, NY, USA: ACM, 2016. (IoT'16), p. 103–110. ISBN 978-1-4503-4814-0. Disponível em: <http://doi.acm.org/10.1145/2991561.2991575>.

STARKS, F.; PLAGEMANN, T. P. Operator placement for efficient distributed complex event processing in manets. In: *2015 IEEE 11th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. [S.l.]: IEEE, 2015. p. 83–90.

STOJANOVIC, N.; STOJANOVIC, L.; XU, Y.; STAJIC, B. Mobile cep in real-time big data processing: Challenges and opportunities. In: *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. New York, NY, USA: ACM, 2014. (DEBS '14), p. 256–265. ISBN 978-1-4503-2737-4. Disponível em: <http://doi.acm.org/10.1145/2611286.2611311>.

TANENBAUM, A. S.; STEEN, M. V. *Distributed systems: principles and paradigms*. [S.l.]: Prentice hall Englewood Cliffs, 2002. v. 2.

TRAN, N. K.; SHENG, Q. Z.; BABAR, M. A.; YAO, L. Searching the web of things: State of the art, challenges, and solutions. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 50, n. 4, p. 55:1–55:34, ago. 2017. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/3092695>.

VIEIRA, V.; TEDESCO, P.; SALGADO, A. C. Modelos e processos para o desenvolvimento de sistemas sensíveis ao contexto. *Andre Ponce de Leon F. de Carvalho, Tomasz Kowaltowski.(Org.). Jornadas de Atualização em Informatica*, p. 381–431, 2009.

WANG, H.; PEH, L. Mobistreams: A reliable distributed stream processing system for mobile devices. In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. [S.l.]: IEEE, 2014. p. 51–60. ISSN 1530-2075.

WANG, Y. *Stream Processing Systems Benchmark: StreamBench*. [S.l.]: LAP LAMBERT Academic Publishing, 2016. ISBN 978-3330326576.

WEISER, M. The computer for the 21st century. *SIGMOBILE Mob. Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 3, n. 3, p. 3–11, jul. 1999. ISSN 1559-1662. Disponível em: <http://doi.acm.org/10.1145/329124.329126>.

WEN, Z.; YANG, R.; GARRAGHAN, P.; LIN, T.; XU, J.; ROVATSOS, M. Fog orchestration for internet of things services. *IEEE Internet Computing*, v. 21, n. 2, p. 16–24, Mar 2017. ISSN 1089-7801.

WIRED. *Hackers Remotely Kill a Jeep on the Highway*. 2015. Disponível em: <https://www.youtube.com/watch?v=MK0SrxBC1xs>.

XU, J.; CHEN, Z.; TANG, J.; SU, S. T-storm: Traffic-aware online scheduling in storm. In: *2014 IEEE 34th International Conference on Distributed Computing Systems*. [S.l.]: IEEE, 2014. p. 535–544. ISSN 1063-6927.

XU, Y.; STOJANOVIC, N.; STOJANOVIC, L.; KOSTIC, D. An approach for dynamic personal monitoring based on mobile complex event processing. In: *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*. New York, NY, USA: ACM, 2013. (MoMM '13), p. 464:464–464:473. ISBN 978-1-4503-2106-8. Disponível em: <http://doi.acm.org/10.1145/2536853.2536866>.

YAN, L.; ZHANG, Y.; YANG, L. T.; NING, H. *The Internet of things: from RFID to the next-generation pervasive networked systems*. [S.l.]: CRC Press, 2008.

YI, S.; LI, C.; LI, Q. A Survey of Fog Computing: Concepts, Applications and Issues. *Proceedings of the 2015 Workshop on Mobile Big Data - Mobidata '15*, p. 37–42, 2015. Disponível em: <http://0-dl.acm.org.cataleg.uoc.edu/citation.cfm?id=2757384.2757397$\protect\delimiter"026E30F$nhttp://dl.acm.org/citation.cfm?doid=2757384.2757397>.

YI, S.; LI, C.; LI, Q. A survey of fog computing: Concepts, applications and issues. In: *Proceedings of the 2015 Workshop on Mobile Big Data*. New York, NY, USA: ACM, 2015. (Mobidata '15), p. 37–42. ISBN 978-1-4503-3524-9. Disponível em: <http://doi.acm.org/10.1145/2757384.2757397>.

YOGI, M. K.; CHANDRASEKHAR, K.; KUMAR, G. V. Mist computing: Principles, trends and future direction. *CoRR*, abs/1709.06927, 2017. Disponível em: <http://arxiv.org/abs/1709.06927>.

YUAN, J.; ZHENG, Y.; XIE, X.; SUN, G. Driving with knowledge from the physical world. In: *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. NY, USA: IEEE, 2011. (KDD '11), p. 316–324. ISBN 978-1-4503-0813-7. Disponível em: <http://doi.acm.org/10.1145/2020408.2020462>.

YUAN, J.; ZHENG, Y.; ZHANG, C.; XIE, W.; XIE, X.; SUN, G.; HUANG, Y. T-drive: Driving directions based on taxi trajectories. In: *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*. NY, USA: ACM, 2010. (GIS '10), p. 99–108. ISBN 978-1-4503-0428-3. Disponível em: <http://doi.acm.org/10.1145/1869790.1869807>.

ZORZI, M.; GLUHAK, A.; LANGE, S.; BASSI, A. From today's intranet of things to a future internet of things: A wireless- and mobility-related view. *Wireless Commun.*, IEEE Press, Piscataway, NJ, USA, v. 17, n. 6, p. 44–51, dez. 2010. ISSN 1536-1284. Disponível em: <http://dx.doi.org/10.1109/MWC.2010.5675777>.