UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

CARLOS EDUARDO ZIMMERLE DE LIMA

USO DE DATA FLOWS PARA PROCESSAMENTO DE EVENTOS COMPLEXOS NA INTERNET DAS COISAS

RECIFE
2017

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

CARLOS EDUARDO ZIMMERLE DE LIMA

USO DE DATA FLOWS PARA PROCESSAMENTO DE EVENTOS COMPLEXOS NA INTERNET DAS COISAS

Monografia apresentada ao Centro de Informática (CIN) da Universidade Federal de Pernambuco (UFPE), como requisito parcial para conclusão do Curso de Sistemas de Informação, orientada pelo professor Kiev Santos da Gama.

RECIFE
2017

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

CARLOS EDUARDO ZIMMERLE DE LIMA

USO DE DATA FLOWS PARA PROCESSAMENTO DE EVENTOS COMPLEXOS NA INTERNET DAS COISAS

Monografia submetida ao corpo docente da Universidade Federal de Pernambuco, defendida e aprovada em 13 de julho de 2017.

Banca Examinadora:

Kiev Santos da Gama

Orientador

Carlos André Guimarães Ferraz

Examinador

Dedico o presente trabalho aos meus pais,
Katia Cristina e José Carlos.

AGRADECIMENTOS

Agradeço primeiramente à minha família pelo apoio. Em especial, à minha mãe, Katia Cristina, que sempre batalhou para que tivesse uma boa formação e ao meu pai, José Carlos, que sempre me incentivou e me apoiou a dar meu melhor.

Agradeço também aos meus colegas pelo companheirismo e dedicação ao longo do curso, assim como, aos meus professores, que se empenharam em repassar seus conhecimentos. Em especial, a professora Carla Silva pelo apoio dado ao longo dos semestres e ao professor Kiev Gama por ter me ajudado na elaboração do trabalho.

“I do not think that there is any other quality so essential to success of any kind as the quality of perseverance. It overcomes almost everything, even nature.”
John D. Rockefeller

RESUMO

A Internet das coisas vem crescendo num ritmo alucinante nos últimos tempos. De fato, predições apontam que logo-logo bilhões de objetos estarão conectados à Internet. No meio tempo, esforços vem sendo feitos para que tais objetos possam ser acessados via web também. Graças a esses esforços, uma nova classe de ferramentas gráficas baseadas nos conceitos de fluxos de dados, que visam conectar componentes da web e dispositivos da Internet das coisas, agora passam a permitir a construção de aplicações que misturam o mundo físico ao mundo virtual. No meio tempo, sistemas, que empregam conceitos de processamento de eventos complexos a fim de detectar padrões a partir da correlação de eventos, vem sendo largamente empregados nas mais diversas áreas. Porém, mesmo que existam interesses envolvendo Internet das coisas e processamento de eventos complexos, conceitos de CEP ainda vem sendo pouco endereçados nessas ferramentas.

Palavras-chave: Internet das coisas. Web das coisas. Mashups físicos. Processamento de Eventos Complexos.

ABSTRACT

The Internet of things has been growing at an amazing pace in the last years. In fact, predictions have pointed out that billions of objects will be connected to the Internet. Meanwhile, efforts are being carried out to enable those objects to be accessible via web as well. Thanks to those efforts, a new class of graphical tools based on the concepts of data flow, that aim to connect web components and Internet of things' devices, can now be used to build applications that mix the physical world with the virtual one. Meanwhile, systems, that use concepts of complex event processing willing to detect event patterns through the correlation of events, are being employed throughout many different areas. However, even though there are interests around Internet of things and complex event processing, CEP concepts are still being poorly addressed on those tools.

Keywords: Internet of things. Web of things. Physical mashups. Complex event processing.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
CEP	Complex Event Processing
HAT	Hardware Attached on Top
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
JS	JavaScript
JSON	JavaScript Object Notation
LED	Light-emitting Diode
MQTT	Message Queue Telemetry Transport
NPM	Node Package Manager
PIR	Passive Infrared
REST	Representational State Transfer
RX	ReactiveX
SQL	Structured Query Language
URI	Uniform Resource Identifier
WoT	Web of Things
XML	Extensible Markup Language

1 INTRODUÇÃO

Diversas ferramentas voltadas para construção de aplicações para Internet das coisas têm surgido nos últimos anos. Entre outras coisas, elas permitem que recursos da web e dispositivos físicos se comuniquem e troquem dados de forma rápida e direta, o que vem permitindo a criação de aplicações cada vez mais elaboradas que mesclam o mundo físico e virtual. Tais ferramentas em sua maioria são construídas utilizando os conceitos de programação orientadas a fluxo de dados, que entre outras coisas permite um rápido desenvolvimento e prototipação de aplicações de IoT.

O conceito de processamento de dados complexos (CEP) foi criado como forma de atender a alta demanda de aplicações que fossem capazes de processar dados em tempo (quase) real, onde métodos tradicionais como sistemas de gerenciamento de banco de dados falharam. CEP permite correlacionar eventos de níveis mais baixos e produzir eventos de alto nível [1] que entre outras coisas podem ser usados como entrada para outras aplicações, disparos de operações, criação de informações relevantes, etc.

Apesar disso, pouco esforço ainda tem sido utilizado para mesclar estes dois mecanismos.

Nesta seção é apresentado a motivação, o objetivo e estrutura do documento.

1.1 MOTIVAÇÃO

O número de objetos conectados na Internet superou o número de seres humanos na terra somente em 2010 [2]. Por volta de 2013, esse número era de 9 bilhões e especialistas apontam que, dado o ritmo atual, em 2050, esse número deve chegar a casa dos 24 bilhões [3]. Todos os esses dados demonstram a importância e o investimento que está sendo feito na Internet das coisas. De fato, espera-se que IoT revolucione o modo em que vivemos, trabalhamos, cada aspecto da nossa vida [2] [4]. Com ele, bilhões de novos tipos de dados e eventos serão gerados e explorá-los será a chave fundamental para construção de um mundo mais inteligente [4].

Conceitos de web das coisas (WoT) tem colaborado para a disseminação de dispositivos na web. Graças a isso, ferramentas, com intuito de facilitar o desenvolvimento de aplicação para IoT, têm cada vez mais surgido. Seu modelo de programação permite que qualquer programador, que nunca tenham programado para IoT, possa criar aplicações simplesmente conectando caixas que representam dados,

funcionalidades e dispositivos, e fazer com que eles interajam com recursos presentes na web. Por outro lado, tecnologias como processamento de eventos complexos que estão disponíveis no mercado a um bom tempo e tem sido utilizadas nos mais diversos cenários, incluindo no próprio processamento de eventos de sensores para criar eventos compostos, possuem o potencial de ajudar a criar aplicações de WoT mais elaboradas, cooperando assim para a evolução do universo de IoT e permitindo que informações extraídas a partir de eventos do mundo real possam ser geradas em tempo hábil.

1.2 OBJETIVO

Este trabalho tem como objetivo principal implementar funcionalidades de processamento de eventos complexos em ferramentas baseadas em programação orientada a fluxos para Internet das coisas, também chamada de mashups físicos. Ao empoderar usuários com estas funcionalidades, eles poderão utilizar uma abordagem visual para efetuar processamento de eventos complexos em fluxos de dados advindos da IoT. Ao final do trabalho, todas as funcionalidades implementadas serão disponibilizas de forma aberta através de repositórios online de modo que qualquer poderá obtê-los e usá-los.

1.3 ESTRUTURA DO DOCUMENTO

Os capítulos que seguem estão estruturados da seguinte maneira:

Capítulo 2: Apresenta a contextualização do trabalho, mostrando o cenário atual na Internet das coisas e uma visão geral em web das coisas, mashups físicos e processamento de eventos complexos.

Capítulo 3: Apresenta as ferramentas utilizados no trabalho.

Capítulo 4: Apresenta as funcionalidades implementadas.

Capítulo 5: Apresenta a conclusão, incluindo dificuldades encontradas durante o projeto e trabalhos futuros.

2 CONTEXTUALIZAÇÃO

Neste capítulo serão abordados os assuntos necessários para o entendimento do presente trabalho. São considerados assuntos relacionados à Internet das coisas. Devido a granularidade do assunto e o foco do trabalho, Internet das coisas não será abordado em grande profundidade. Em vez disso, a primeira parte focará mais em web das coisas e, principalmente, mashups físicos, ferramentas baseadas na web que fazem uso de data flows (fluxo de dados) com o objetivo de facilitar a criação de aplicações misturando smart things (coisas inteligentes) e recursos da web. Em seguida, é abordado os conceitos de processamentos de eventos complexos (CEP – complex event processing), conceito esse que já vem sendo largamente usado em diversas áreas, mas que vem sendo cada vez mais requisitada na Internet das coisas, descrevendo assim principais conceitos relacionados, funcionalidades e usos.

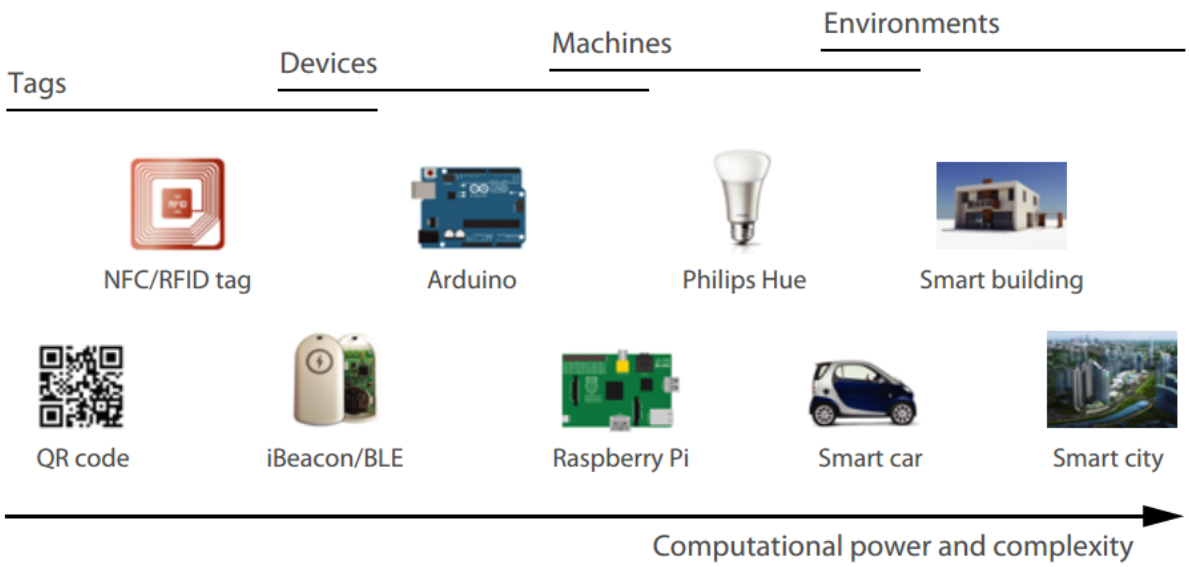
Este capítulo está organizado do seguinte modo: A seção 2.1 apresenta uma visão geral sobre Internet das coisas, tendo com subseções web das coisas e mashups físicos. A seção 2.2, por sua vez, descreve o conceito de processamento de eventos complexos. Finalmente, a seção 2.3 correlaciona Internet das coisas e processamento de eventos complexos.

2.1 INTERNET DAS COISAS

Nos últimos anos, inúmeros novos dispositivos têm sido conectados à Internet. Só em 2010, o número de objetos conectados na rede superou o número de seres humanos na terra [2]. Por volta de 2013, esse número era de 9 bilhões e especialistas apontam que dado o ritmo atual, em 2020, esse número deve chegar a casa dos 24 bilhões [3]. A esse fenômeno foi dado o nome de Internet das coisas (Internet of Things ou IoT, no inglês), que trata-se da interconexão de “coisas” na Internet. Coisas, por sua vez, nada mais são que objetos do dia-a-dia que possuem algum dispositivo de computação e transmissão [5]. Uma melhor definição é dada por Swan [6], onde é considerado a ideia de Internet das coisas o fato de coisas, que através da Internet (seja por RFID, rede sem fio LAN ou rede de longa distância), podem ser lidas, reconhecidas, localizadas, endereçadas e controladas. As coisas são das mais variadas naturezas, podendo ser termostatos, sistemas de monitoramento e controle HVAC (aquecimento, ventilação e ar-condicionado) ou coisas mais corriqueiras como uma xícara de café ou uma cadeira [2] [5]. A Figura 1, retirada de [4], exemplifica os diversos tipos de coisas em IoT, variando de Auto-ID tags em geral (bar codes, QR

codes, e NFC/RFID tags) que podem ser utilizadas em produtos com propósitos de identificação, até o nível de edificações e cidades inteligentes.

Figura 1 - Exemplos de coisas inteligentes na Internet das coisas



Fonte: GUINARD e TRIFA, 2016, p. 5.

É esperado que essa nova revolução no mundo da Internet possa melhorar a qualidade de vida das pessoas, impactar lares e negócios, além de impulsionar a economia mundial [2]. Dentre as áreas que IoT espera-se impactar, pode-se citar: transporte, saúde, indústria, situações emergenciais, etc [2]. Um exemplo interessante é citado em [1] em que descrito um lar inteligente (smart home, no inglês) onde diversas coisas seriam automatizadas, tal como a auto regulação do ar condicionado. Conforme mencionado em [7], ao fazer uso de algum mecanismo de processamento de streams de dados, o ar condicionado poderia ser regulado de tal forma que poderia proporcionar um ambiente mais agradável ao residentes, assim como poderia impactar também na medição de energia, levando-se em conta as medições de sensores externos (ou dados providos de serviços externos através da Internet) que entre outras coisas permitissem acesso a condições externas, tais como temperatura, tempo, etc. De fato, um dos principais objetivos do conceito de IoT, não só conectar as coisas na Internet mas permitir que as próprias coisas possam se conectar (machine-to-machine ou M2M) a fim de criar ambientes inteligentes (smart environments) [3].

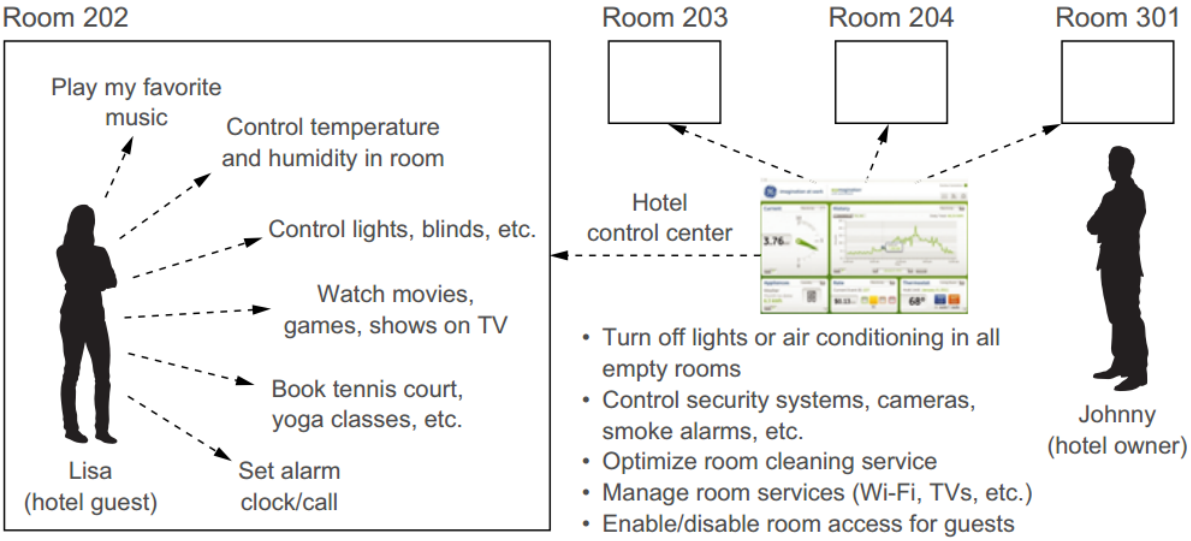
Contudo, desafios ainda permeiam o universo da Internet das coisas, como o fato que é praticamente impossível criar um ecossistema de coisas que possam comunicar-se de forma direta [4]. O que se tem na atualidade são ilhas de coisas com

falta de interoperabilidade na camada de aplicação [5] [8]. Guinarde e Trifa [4] vão mais além, chamando a atual Internet das coisas de Intranet das coisas. Diversos fatores contribuem para isso, como a quantidade de tecnologias proprietárias que implementam e utilizam diferentes protocolos, assim como o fato da Internet das coisas focar mais nas camadas inferiores do que na de aplicação no modelo de referência OSI [8] [4]. Assim, não existe um único protocolo na camada de aplicação, mas vários. Esse cenário assim demanda soluções que permitam ao mesmo a integração dessas ilhas formadas por redes de dispositivos físicos de forma prática e universal, assim como, a possibilidade de criação de aplicações desacopladas e escaláveis.

2.1.1 WEB DAS COISAS

Conforme mencionado anteriormente, a Internet das coisas tem aplicado mais esforços ao longo dos últimos anos em camadas focadas na conectividade e pouco na camada de aplicação. Este fato culminou no uso de diferentes protocolos proprietários que pouco cooperam para a criação de uma rede mundial que interconecte facialmente serviços e coisas. A fim de ilustrar a dificuldade que usuários que gostariam de investir em tal proeminente tecnologia encontram, Guinarde e Trifa [4] nos apresentam um exemplo de um usuário, mais especificamente um dono de uma rede de hotéis, cujo nome fictício seria Johnny B., que gostaria de maximizar o gerenciamento de sua hotelaria, através da conexão dos aparelhos e dispositivos dos quartos de hotel a um sistema de controle. Ao mesmo tempo, este sistema poderia cooperar para o melhoramento da experiência dos clientes. A Figura 2 ilustra a situação pretendida por Johnny.

Figura 2 - Exemplo de uso de dispositivos e interoperabilidade em IoT



Fonte: GUINARD e TRIFA, 2016, p. 7.

Assim como em diversos casos em IoT, Johnny possivelmente teria que obter cada dispositivo, sensores, equipamentos advindos de diferentes fornecedores. Fazer com que cada dispositivo consiga trabalhar de forma conjunta, exigiria de Johnny um investimento pesado na criação de um sistema para integrar todos os gadgets com seu sistema de hotel inteligente desejado. Mesmo que ele conseguisse, chances de problemas ocorrem no sistema seriam altas, o que incluem problemas pós implementação em termos de extensibilidade e manutenibilidade. Na hipótese de que Johnny decidisse construir ele mesmo o sistema e procurasse fazer as compras dos dispositivos de um único produtor, novamente ele encontraria problemas visto que é quase improvável que ele consiga todos eles advindos de um mesmo fornecedor. Se encontrar, existe uma grande probabilidade de que o sistema provido por esse produtor não corresponderia ao que Johnny estava esperando, um sistema de fácil uso e configuração. Assim ele acabaria optando pela construção de um novo sistema que por sua vez lhe tomaria tempo não só para construí-lo mas para levar em consideração fatores essenciais para um bom sistema, como escalabilidade, confiabilidade e segurança [4].

Como uma forma de sanar os problemas encarados por muitos Johnnys da vida e para que a Internet das coisas se torne real [4], a conceito de web das coisas (web of things ou WoT, no inglês) vem ganhando cada vez mais destaque. O que a web das coisas propõe é fazer com que a Internet das coisas seja incorporada a web (camada de aplicação), tornando coisas inteligentes como cidadãos de primeira classe [9]. Kenda et. al. [5] define WoT como a conexão das ilhas de coisas (cenário atual de

IoT) utilizando padrões da web. Muitos defensores do conceito argumentam que não existe motivo de continuar criando novos protocolos (consequentemente, sistemas proprietários e acoplados) a fim de possibilitar a comunicação das coisas [4] [9] [10]. Invés disso, eles defendem a reutilização de algo que é adotado e utilizando no mundo para criar sistemas escaláveis e iterativos, a web [4] [9]. Desta forma, qualquer pessoa (desenvolvedor) com conhecimentos básicos nas tecnologias web (HTTP, HTML, JavaScript) poderia fazer parte desta revolução de dispositivos, sendo que para isso ele apenas precisaria de um editor de textos para começar a interagir com as coisas inteligentes [4] [9].

A arquitetura proposta para web das coisas faz uso de padrões e tecnologias usualmente utilizadas na web [10]. De uma forma geral, servidores da web são embutidos nas coisas inteligentes e recursos do mundo físico passam a serem manipulados através do uso da arquitetura REST [9] [10]. A escolha dos proponentes em utilizar a arquitetura REST se dá pelo fato de que ele tem como uma das principais essências a criação de serviços pouco acoplados que por sua vez estimula o reuso. Além disso, todos os recursos são facialmente endereçados e acessados utilizando URIs [9]. Existem muitas vantagens de se utilizar essa metodologia. Uma delas é o fato de que as coisas inteligentes passam a serem identificadas através de URIs, que por consequência permite que elas sejam enviadas, referenciadas em web sites e salvas nos favoritos [9]. Além disso, coisas passam a poderem ser encontradas ou descobertas através da navegação na Internet, além do fato que a maior parte de sua interação passa ser em um navegador da web, ferramenta essa largamente disponível e intensamente utilizada por todos os tipos de usuários [9]; sem mencionar o fato de poder contar com mecanismos que são utilizados na web e que colaboraram para a escalabilidade e sucesso da web, tais como o conceito de caching, balanceamento de carga, indexação e busca [9].

Assim como qualquer serviço que utiliza REST, os métodos HTTP também estão disponíveis e são utilizados para realizar as interações com as coisas. GET é utilizado para obter a representação de um recurso [9]; por exemplo, obter a medição de temperatura ambiente. PUT é usado para atualizar o estado de um recurso ou criar um recurso [9]; por exemplo, pode ser usado para controlar um LED [9]. DELETE é utilizado visando a deleção ou desativação de um recurso [9]; por exemplo, ele poderia ser utilizado para desativar um dispositivo. Por último, POST cria um novo recurso; por exemplo, cria um novo feed que permitiria o rastreio de uma coisa.

A Figura 3, retirada de [10], exemplifica o uso de URIs para acessar recurso em WoT. No contexto do artigo, esses dois URIs são utilizados para obter representações de recursos em um wireless sensor network (no caso Sun SPOT). Wireless sensor network é uma tecnologia que ganhou destaque no começo dos anos 2000 graças aos avanços na miniaturização de computadores embarcados e chips de rádio (radio networking chips) [4]. Ele consiste em um computador pequeno single-board, que devido ao seu custo e bateria de longa duração, eles são muitas vezes colocados em áreas para o monitoramento de espaços físicos utilizando para isso um conjunto de sensores [4]. Cada spot, por sua vez, teria alguns sensores (luz, temperatura, acelerômetro, etc.), alguns atuadores (saídas digitais, LEDs, etc.) e alguns componentes internos (radio, bateria) [10]. No caso da primeira URI, ao acessá-la utilizando um browser por exemplo, seria feito uma requisição ao recurso light (luz) do recurso sensors (sensores), do spot1 [10]. Na segunda URI, ao requisitá-la, seria retornado links par todos os tipos de sensores providos por spot1 [10].

Figura 3 - Exemplos de URIs

```
http://.../sunspots/spot1/sensors/light
http://.../sunspots/spot1/sensors/
```

Fonte: GUINARD et. al., 2011, p. 5.

Em termos de representação dos recursos, dois tipos de formatos são comumente empregados, um para cada tipo de interação com o dispositivo [10]. Para interação humano-máquina, a forma representacional preferível seria o HTML [10]. Já no caso de máquina-máquina, a representação sugerida é o JSON no lugar de XML, visto que JSON é mais leve que o XML, o que torna-o mais indicado para sistemas com capacidades limitadas [10]. A Figura 4 mostra um exemplo de um JSON obtido ao consultar o recurso temperatura de um Sun SPOT.

Figura 4 - JSON representando temperatura em um Sun SPOT

```
1     {"resource":
2     {"methods":["GET"],
3     "name":"Temperature",
4     "children":[],
5     "content":
6     [{"description":"Current Temperature",
7     "name":"Current Ambient Temperature",
8     "value":"27.75"}]}}
```

Fonte: GUINARD et. al., 2011, p. 7.

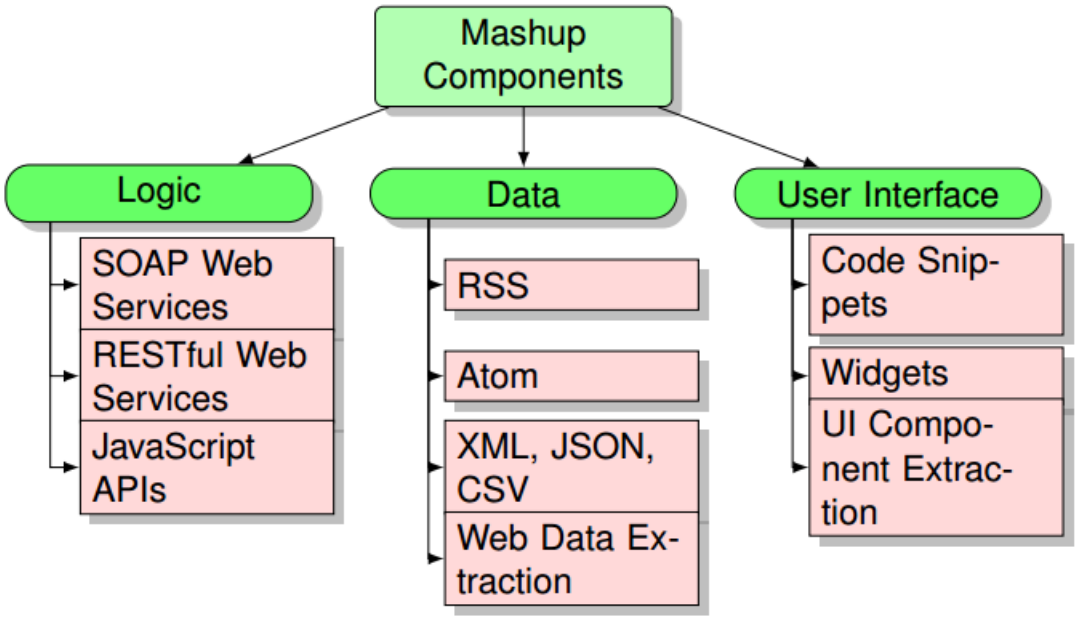
Em consequência da integração à web, as coisas passam a serem cidadãos de primeira classe da web [4]. Desse modo, um leque de possibilidades se abre ao mundo das coisas, como o fato de que agora elas podem ser acessadas e utilizadas nas mais diversas aplicações da web sem problemas ou esforços [4]. Seguindo este princípio, as coisas vêm sendo integradas em ferramentas que dentre outras coisas, facilitam a criação de aplicações híbridas envolvendo IoT e vários serviços disponíveis na web [4].

2.1.2 MASHUPS FÍSICOS

Mashup é um conceito advindo da Web 2.0. Mashup nada mais é que uma aplicação que agrega dois ou mais componentes da web [11] e usa-os para criar novas aplicações [10]. Os componentes por ele agregados podem ser dados, lógica de aplicação ou interfaces do usuário [11]. Individualmente, o termo utilizado no inglês para se referir aos componentes é mashup components [11]. Por outro lado, o mecanismo que descreve como o mashup é operado e seus componentes orquestrados é chamado de mashup logic. Ele define os componentes selecionados, o fluxo de controle e dados, mediação de dados e transformação dos mesmos entre componentes distintos [11].

Os componentes do mashup (mashup components) são as peças que formam um mashup [11]. Eles são constituídos por diversas tecnologias e padrões [11]. A Figura 5 sumariza os diferentes tipos de tecnologias empregadas nos componentes.

Figura 5 - Componentes de um mashup



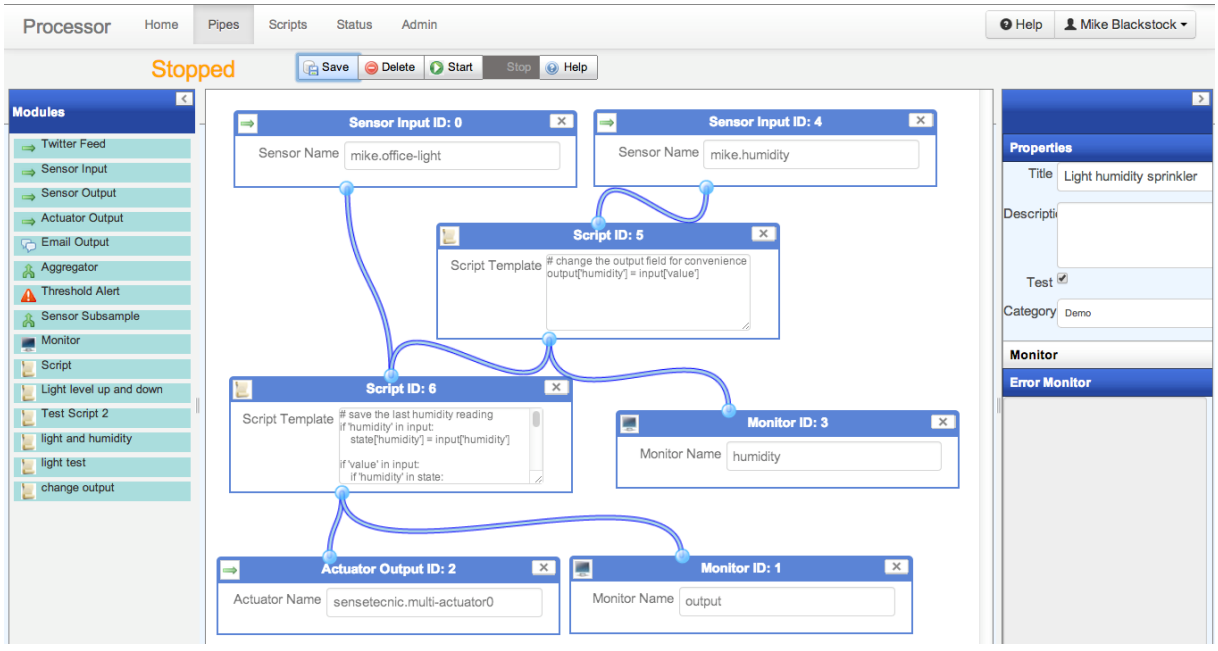
Fonte: Sheng et. al., 2017, p. 76.

Componentes lógicos (logic components) oferecem o uso de funcionalidades através de algoritmos reusáveis visando determinadas funções [11]. Dados (data) proveem acesso aos dados [11]; estes podem ser mais estáticos tal como feeds (RSS) ou mais dinâmicos como web services [11]. Por fim, interface do usuário (user interface) agrega tecnologias de componentes padronizados com o intuito de prover fácil integração entre interfaces do usuário advindos de terceiros e interfaces do usuário na aplicação de mashup [11].

Com o objetivo de criar uma maneira simples de criar mashups, ferramentas, construídas em torno de protocolos de comunicação uniforme e APIs baseadas em REST, foram propostas [11]. Microsoft Popfly e Yahoo Pipes são exemplos de ferramentas que foram desenvolvidas nos primórdios do segmento [11]. Nos dias atuais, graças aos estudos em WoT, uma nova classe de mashups tem surgido que estende os conceitos de mashups para o mundo físico dos objetos [11] [10]. Para essa classe dá-se o nome de mashups físicos, que permite com que serviços de coisas físicas sejam combinados com serviços providos por recursos da web, possibilitando assim a prototipação e criação de forma rápida e fácil de sistemas de WoT [4] [11] [12]. Mashups físicos, assim como ferramentas de mashups predecessoras a eles, são construídos em torno de interfaces REST e de forma geral são baseados nos conceitos de modelagem (programação) de fluxo de dados (data flow) [11] [12]. As pesquisas em torno de programação de fluxo de dados remontam meados da década

de 70 onde era aplicada na exploração de paralelismo dos processadores [12]. Ela tem sido aplicada em diversas áreas aos longos dos anos, tais como alta performance em computação paralela tirando proveito das arquiteturas de multiprocessadores, músicas, brinquedos assim como aplicações industriais [12]. Programação de fluxo de dados baseia-se na representação de recursos de forma visual em forma de grafos direcionados, isto é entrada de dados, saídas e funções são representados através de nós, onde a interconexão deles através de arcos dita o fluxo de dados entre os mesmos [12]. No contexto de mashups físicos, em geral, os nós (também referidos como caixas por alguns autores [4]) abstraem operações já implementadas, como obter dados de uma API REST, enquanto que arcos (ou fios [4]) representam o fluxo de dados e controles [4]; as trocas de mensagens entre as caixas são feitas de forma assíncrona, por exemplo utilizando estilo de comunicação REST [11]. Para compor a aplicação, editores gráficos são normalmente oferecidos pelos WoT mashups [11]. Exemplos de WoT mashups incluem: IBM Node-RED, glue.things (extende Node-RED [8]), WoTKit, e Clickscript [11]. A Figura 6 demonstra um exemplo de um editor gráfico, especificamente do WoTKit Processor.

Figura 6 - Editor gráfico do WoTKit



Fonte: Who is FRED?[1]

Existem vários fatores que levam a adoção e uso de tais ferramentas. Primeiro, mashups físicos nada mais faz que reunir todo o conceito abordado no

[1] Disponível em: <http://sensetecnic.com/who-is-fred/>. Acesso em jun. de 2017.

desenvolvimento de WoT e entregá-lo numa forma pronta para uso [4]. Assim, permite que os usuários foquem mais no desenvolvimento de aplicações complexas, em especial na conexão de componentes e lógica da aplicação [4]. Segundo, facilita a integração com os mais variados recursos da web. Por exemplo, um usuário pode facialmente utilizar dados providos das coisas e criar uma aplicação que se utiliza desses dados para postar em redes sociais (o exemplo em [4], que envia um alerta de intrusão para o Twitter[1] quando a atualização no estado do sensor PIR). Terceiro, o uso de programação baseada em fluxo de dados facilita não só a criação e entendimento por parte de desenvolvedores, mas também por não programadores [12]. Finalmente, estudos vem mostrando os benefícios da aplicação de programação orientada a fluxo no desenvolvimento de aplicações dirigidas a dados (data-driven applications) em ambientes inteligentes [13]. Resultados apontam para um claro mapeamento partindo do design da arquitetura para implementação, aumento na velocidade de desenvolvimento de software, além de promover reusabilidade e manutenibilidade [13].

Conforme evidenciado no início do capítulo, tendências apontam para uma grande onda de novas tecnologias de IoT a serem geradas nos próximos anos. Avanços em WoT, em especial mashups, evidenciam a proliferação dos dispositivos físicos nos ambientes virtuais que estão desencadeando/desencadearão inúmeras aplicabilidades diferentes do que estamos acostumados a lhe dar nesse mundo de conteúdo atual da web. Ao alinhar os dois fatos apresentados anteriormente, faz-se necessário o uso de alguma ferramenta na qual possa tirar proveito dos inúmeros eventos que essa maré de tecnologias trará com o modelo de fácil desenvolvimento e integração das coisas na web providos pelos mashups físicos.

2.2 PROCESSAMENTO DE EVENTOS COMPLEXOS

O conceito de processamento de eventos complexos (complex event processing ou CEP) surgiu em resposta ao crescente aumento de geração de dados e a necessidade cada vez maior de processá-los em (quase) tempo real, no qual o modelo empregado em sistema de gerenciamento de banco de dados (database management system ou DBMS) não era capaz de atender [5] [14]. Os sistemas de banco de dados tradicionais requerem que dados seja armazenas e indexados para posterior uso, o que contrastava com os requerimentos de um motor para

[1] https://twitter.com

processamento de fluxos de informação (information processing flow engine ou IFP engine) no qual CEP se enquadra [14].

Antes de adentrar o universo de CEP, se faz necessário entender e definir o que seria um evento. De acordo com Kenda et. al. [5], eventos podem ser vistos como qualquer coisa que tenha ocorrido ou que possa ser percebido como tendo acontecido. Por exemplo, o recebimento de uma mensagem, alterações na umidade do ar, leituras geradas por um sensor, etc. Eventos, por sua vez, podem se relacionar através de tempo, causalidade e agregação [1]. Ao agregar eventos, novos eventos podem ser formados. A estes eventos gerados a partir de um conjunto de eventos dá-se o nome de eventos complexos [1].

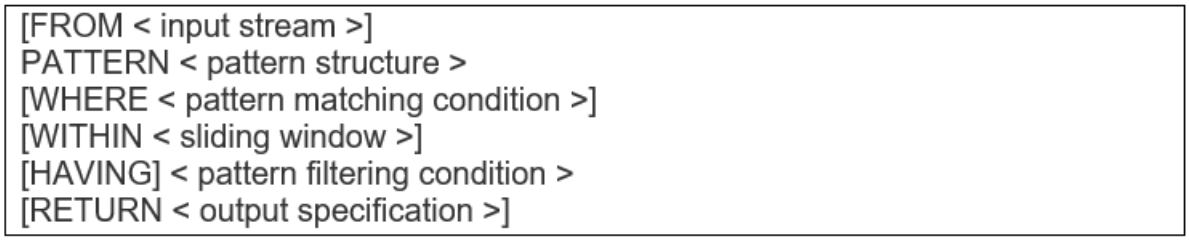
De um modo geral, CEP pode ser visto como um sistema que recebe e correlaciona um conjunto de eventos de baixo nível e assim gera eventos de alto nível [14]. Ele é diretamente relacionado a detecção de padrões de eventos utilizando técnicas e procedimentos de filtragem, correlação, contextualização e análise em dados advindos de diferentes fontes [15]. Adicionalmente, ele põe grande ênfase na detecção de padrões complexos envolvendo relacionamentos de ordem e sequência [14]. Seu objetivo principal é identificar oportunidades e ameaças através da detecção e geração dos eventos de alto nível, e responde-los o mais rápido possível [15]. Por exemplo, um alerta de incêndio poderia ser gerado quando três diferentes sensores detectassem a existência de fogo, que estivessem localizados numa área menor que 100m², com uma temperatura acima dos 60°C e com 10 segundos de diferença um do outro [14].

2.2.1 OPERAÇÕES E PADRÕES

Para modelos que se enquadram no sistema de processamento de fluxo de informações (IFP), como CEP, existem atualmente dois tipos de linguagens predominantes, linguagens de transformação e linguagens de detecção, também conhecidas como baseadas em padrões [14]. O segundo tipo é normalmente mais empregado em CEP, sendo que ambas compartilham diversos tipos de operações [14]. Seus operadores podem ser divididos em dois tipos: bloqueantes, os quais só podem ser executadas depois que todo ou uma parte do fluxo de entrada é lido e não bloqueantes, no qual retornam resultados à medida que novos dados são lidos [14]. Para dar suporte às operações bloqueantes e até mesmo como uma forma de impor um limitante da quantidade de eventos que operações não bloqueantes devem

considerar, um elemento estrutural destas linguagens é comumente utilizado em diversas operações, chamado janelas (windows) [14]. Essas janelas representam o primeiro passo para aplicar padrões [15] e normalmente diversos tipos de windows são empregados [14]. De uma forma geral, elas podem ser classificadas em dois tipos: lógicas (baseadas em tempo) e físicas (baseadas em quantidade) [14]. Para o primeiro caso, janelas lógicas, limites de elementos na janela são determinados em termos de tempo, como por exemplo, considerar os eventos recebidos nos últimos três minutos. O segundo limita a quantidade em torno de um valor definido, como por exemplo, a janela deve armazenar apenas três eventos para que as determinadas operações sejam executadas. Existem ainda outro tipo de classificação de janela no qual considera o modo como os limites da mesma se movem [14]. A Figura 7 mostra um exemplo de uma query escrita na linguagem de CEP chamada SASE+.

Figura 7 - Exemplo de query em CEP



Fonte: SASE[1]

Um fato importante a se observar na Figura 7 é a semelhança com o a linguagem SQL para banco de dados. Em geral, linguagens CEP baseadas em queries, seguem a mesma lógica da apresentada na Figura 7, permitindo o uso de grande parte das operações listadas de forma resumida no Quadro 1 apresentadas em Cugola e Margara [14].

Quadro 1 - Resumo das operações de CEP

Operador	Descrição
Single-Item Operators	Eles podem ser de dois tipos: Seleção e elaboração. O primeiro seleciona informações baseadas no conteúdo das mesmas. Elaboração, por outro lado, transforma conteúdos relativos ao evento.
Logic Operators	Usados na detecção de itens de informação. Conjunção, disjunção, repetição e negação se enquadra nesta categoria.
Sequences	Parecido com logic operators, mas leva em consideração a ordem de chegada dos itens.
Iterations	Expressa sequências de informação que satisfazem uma condição de iteração.
Join Operator	Correlaciona dois fluxos de informação tal como em DBMS.
Bag Operators	Combina fluxos de informações distintos considerando-os como sacos (bags) de itens.

[1] Disponível em: <http://sase.cs.umass.edu/>. Acesso em jun. 2017.

(conclusão)

Operador	Descrição
Duplicate Operator	Permite que um fluxo seja duplicado.
Group-by Operator	Particiona o fluxo em partições diferentes, a partir de uma propriedade.
Order-by Operator	Ordena itens.
Parameterization	Filtra fluxos baseadas em informações advindas de outros fluxos.
Flow Creation	Permite criar novo fluxo a partir de um conjunto de itens
Aggregates	Agrega conteúdo de itens para produzir nova informação, tal como calcular média ou o valor máximo.

Fonte: Cugola e Margara, 2012, p. 19-28.

De um modo geral, como pode ser visto, alguns dos operadores apresentados realmente possuem certo relacionamento com os utilizados em SQL. Outros nem tanto, sendo voltados exclusivamente para manipulação e detecção em streams de eventos. É importante observar que todos eles são acompanhados de exemplos em [14], ambos em linguagem de transformação e detecção. O Quadro 2 com informações retiradas de [16] sumariza um conjunto de padrões que correspondem a fundação das aplicações de CEP e que inclusive incluem alguns dos operadores também listados no Quadro 1.

Quadro 2 - Resumo dos padrões de CEP

CEP Pattern	Descrição
Filtering	Avalia atributos de eventos utilizando uma condição lógica.
In-Memory Caching	Também conhecido como window (janela), armazena eventos em memória ambos recentes ou advindos de um banco de dados.
Aggregation over Windows	Usado para computar operações estatísticas em eventos na memória.
Database Lookups	Uso de dados históricos para enriquecer eventos de entrada.
Database Writes	Em determinada situações, o armazenamento de eventos, processados ou não, em banco de dados pode ser utilizado.
Correlation (Joins)	Correlaciona eventos advindos de diferentes streams.
Event Pattern Matching	Checa a ocorrência de eventos relacionados. Utiliza-se do tempo na composição de relacionamentos.
State Machines	Usado nos casos em que comportamento e processos complexos devem ser modelados e rastreados. Eventos são usados para sinalizar transições de um estado para outro.
Hierarchical Events	Usado para criar eventos complexos que possuem uma relação hierárquica.
Dynamic Queries	Pode aparecer de 3 formas diferentes: • Registro de queries dinamicamente: Permite inserção de queries dinamicamente sem necessidade de reinicialização do servidor; • Queries de requisição/resposta: Realiza as análises em streams normalmente, mas só retorna os resultados quando explicitamente solicitado. • Queries de subscrição: Registra interesse em determinados eventos. Resposta é retornada para os subscritos.

Fonte: Coral8, 2006, p. 1-17.

Os padrões apresentados em [16] estende as funcionalidades contidas em [14] que, por sua vez, são sumarizadas no Quadro 1 dado que ele adiciona outros fatores, tais como iterações com banco de dados para lidar com dados históricos ou maquinas de estado. Diversos exemplos e ilustrações são apresentados nas seções que descrevem cada um desses padrões em [16]. Conforme é visto nos capítulos mais à frente, ambos padrões e funcionalidades descritos no Quadro 1 e 2 foram levados em consideração para a elaboração do projeto, especialmente os do Quadro 2 dada a descrições detalhadas encontradas no artigo.

2.3 CEP E IOT

Conforme mencionado no início do capítulo, existem um grande indicativo que 20 a 50 bilhões de dispositivos estarão conectados Internet daqui a alguns anos [4]. Com eles, uma enorme rajada de eventos irá certamente ser gerado, além do fato de que cada vez mais aplicações de IoT passarão a demandar extração de informações úteis advindas desses eventos. CEP tem se provado eficaz na produção de informação de valor elaboradas a partir de um conjunto de eventos e ao longo dos anos tem sido empregado nas mais diversas áreas, variando de mercado de capitais até sensores [15] [16]. Além disso, diversos estudos vêm vinculo CEP à IoT, inclusive relacionando o uso de CEP em mashups de WoT, como pode ser visto em [5] [7]. Deste modo, CEP pode ser apontado como uma das ferramentas mais proeminentes para a geração de eventos complexos em IoT.

Olhando para o exemplo de Johnny B. apresentado na seção 2.1.1, não é difícil pensar em meios de estender a sistema de controle da hotelaria dele através do uso de operações de CEP. Por exemplo, Chen et. al. [7] apresenta um exemplo onde ao cruzar leituras de temperatura de dentro e fora de um prédio, o sistema de ar condicionado pode ser autorregulado de forma a proporcionar um ambiente mais agradável que acabaria implicando também por consequência na redução do consumo de energia. Isto se encaixa perfeitamente no que Johnny está a procura, ao buscar um hotel inteligente. Outro exemplo é apresentado por Cugola e Margara [14], onde um sistema de proteção de incêndio de um prédio estaria interessado em disparar um alerta quando a temperatura ultrapasse a faixa dos 40°C, mas apenas se um evento advindo do detector de fumaça localizado no mesmo ambiente emitisse um alerta de fumaça também. Isto não só promoveria um ambiente mais seguro para os hóspedes do hotel, como também permitiria que possíveis incêndios nele fossem

evitados antes mesmo de acontecerem. Enfim, são inúmeras as possibilidades que a junção de IoT e CEP pode e poderão proporcionar, principalmente agora que coisas passaram a integrar a web, o que abre a porta para aplicações mais complexas e elaboradas que, com toda certeza, impactarão o cenário atual.

3 FERRAMENTAS

Este capítulo trata das ferramentas e bibliotecas utilizadas para criação dos componentes implementados. Ele apresenta o Node-RED, mashup físico escolhido para servir como base para a implementação. Também são apresentados os módulos (equivalente a bibliotecas em outras linguagens) do Node.js que foram utilizados no desenvolvimento.

Ele está organizado do seguinte modo: Seção 3.1 mostra uma visão geral do Node-RED, cobrindo em seguida seus nós (blocos na qual o modelo de programação do Node-RED se baseia) assim como passos para criação de um nó customizado na subseção que segue. Partindo da seção 3.2, o foco muda do Node-RED para os módulos disponibilizados no NPM (biblioteca de módulos do Node.js) que foram utilizados na implementação. Para os módulos na seção 3.2, eles são apresentados em ordem de importância na implementação, partindo dos principais, que orquestram as principais funcionalidades, para os mais básicos, que são atribuídos tarefas de menor granularidade.

3.1 NODE-RED

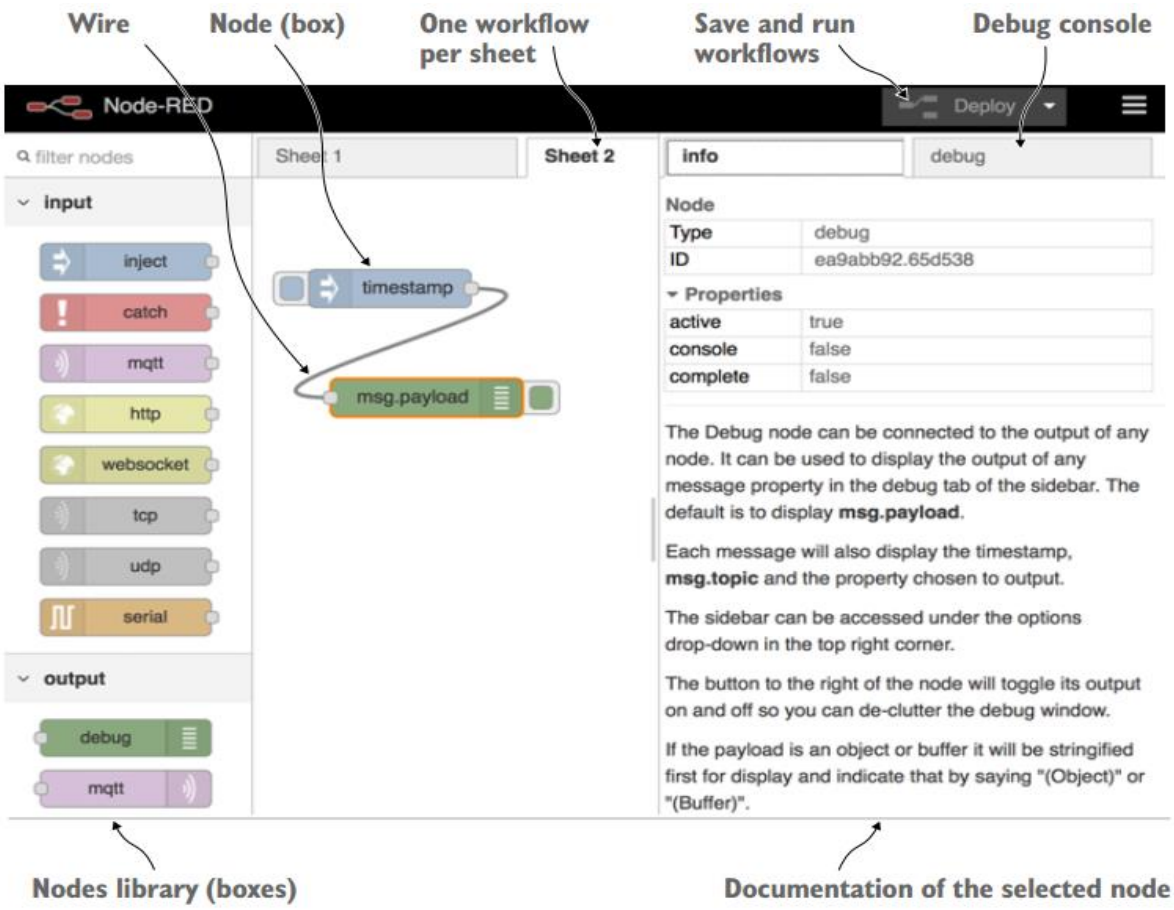
Node-RED[1] é uma ferramenta (mashup físico) voltada para criação de mashups para IoT. De código aberto (licença Apache 2.0), foi criado por uma equipe da IBM e recentemente integrou-se a JS Foundation[2]. Sua construção é baseada em JavaScript, mais especificamente utiliza o Node.js como o seu principal framework. Desta forma, Node-RED consegue tirar proveito do modelo de I/O dirigido a eventos (event-driven) não bloqueante (non-blocking) do Node.js adequado para aplicações que lidam com manipulação intensa de dados em tempo real [11]. Ele pode ser executado em diferentes plataformas [17], tais como os sistemas operacionais comumente utilizados em máquinas locais (Windows, Mac, Linux), em microcomputadores como o Raspberry PI (bastante utilizados na construção de aplicações de IoT), além de soluções em cloud. Seu modelo de programação baseia-se na programação baseada em fluxos direcionada Internet das coisas [18]. Sendo assim, toda sua programação é realizada com base na construção de workflows entre nós (também chamados de caixas ou blocos) e conexões (também chamados de fios

[1] https://nodered.org/
[2] https://js.foundation/

ou arcos) seguindo o mesmo modelo já discutido no capítulo 2 na seção sobre mashups físicos. Dentre os motivos, ele foi escolhido para ser utilizado na implementação pois ele é especialmente orientado para construção mashups para IoT [4] e possui uma grande comunidade ativa que diariamente lança novas funcionalidades.

Seu ambiente de desenvolvimento consiste em uma interface web subdivida em três seções. A primeira mais à esquerda agrupa todos os nós disponíveis [4], seja ele provido de antemão pela instalação do Node-RED ou instalado a parte posteriormente. Um canvas (editor visual) na parte central, onde os workflows são criados [4] [12]. Para criar um fluxo, o usuário só necessita arrastar e soltar os nós contidos na seção a esquerda para dentro do canvas e conectar as respectivas portas dos nós [11] [12]. Com o nó no canvas, as configurações dele podem ser acessadas clicando duas vezes no mesmo. A terceira e última parte mostra a documentação dos nós atualizada ao selecionar diferentes nós [4]. Também, ela contém uma aba que corresponde ao console de debug [4]. A Figura 8 demostra o visual do node-RED destacando as diferentes partes encontradas nele.

Figura 8 - Visão geral do visual do Node-RED



Fonte: GUINARD e TRIFA, 2016, p. 290.

Dentre as funcionalidades do Node-RED, merece destaque a forma fácil de exportação dos fluxos (flows). Eles são exportados em formato JSON, que permite dentre outras coisas a fácil distribuição e até mesmo a importação de fluxos de terceiros. Inclusive, ambos nós e fluxos customizados podem ser encontrados no seguinte repositório [https://flows.nodered.org/], com ressalva de que os nós lá são um mero espelhamento dos nós publicados no repositório NPM[1].

3.1.1 NÓS

Por padrão, diversos nós já vem pré-instalado no Node-RED. Eles basicamente podem ser divididos em três tipos: nós de entrada, nós de processamento e nós de saída [19]. Toda a entrada da aplicação em Node-RED é provida através dos nós de entrada. Eles normalmente possuem uma porta representada por uma pequena caixa cinza no lado direito do nó. Os nós de realizam algum processamento em partes do fluxo. Eles normalmente possuem uma porta do lado esquerdo e uma do lado direito. Os de saída proveem um meio de saída para a aplicação, seja na seção de debug ou serviço externo. Toda a comunicação entre os nós é baseada na troca de mensagens, mais especificamente através do objeto chamado msg. Muitas das vezes, msg é acompanhado de uma propriedade chamada payload que como o nome já diz representa a carga útil da mensagem (msg pode conter inúmeros parâmetros) [19].

3.1.2 CRIAÇÃO DE NÓS

Os nós em Node-RED são criados a partir de dois arquivos: um HTML representando a lógica e visual do lado cliente e um arquivo JavaScript implementando a manipulação do lado servidor [19]. Os trechos de código contidos na Figura 9 e 10 foram retirados da documentação online do Node-RED e é apresentado na documentação como um exemplo para a implementação de nós customizados. Ele nada mais faz que criar um nó de processamento para transformar toda mensagem contida no atributo payload do objeto msg e transformá-la em maiúscula. Para que os nós em Node-RED interajam no fluxo, todos eles herdam de duas classes: a classe base, Node e a classe EventEmitter [12]. A classe Node permite que qualquer nó envie mensagem para o fluxo através da função send() [12]. Já a classe EventEmitter, permite que o nó escute o de entrada input e registre uma função callback para manipulação de mensagens advindas do fluxo [12]. A Figura 9

[1] https://www.npmjs.com/

representa o a lógica do lado servidor, no qual demostra como o nó interage com o evento input e manda a mensagem de volta ao fluxo através do método send().

Figura 9 - Exemplo de criação de nó arquivo JS

```
module.exports = function(RED) {
    function LowerCaseNode(config) {
        RED.nodes.createNode(this,config);
        var node = this;
        node.on('input', function(msg) {
            msg.payload = msg.payload.toLowerCase();
            node.send(msg);
        });
    }
    RED.nodes.registerType("lower-case",LowerCaseNode);
}
```

Fonte: Criando seu primeiro nó[1]

A Figura 10 representa a lógica do lado cliente do Node-RED. Ela é separada em três blocos de script, cada um representando diferentes partes do nó. A primeira parte corresponde a configurações mais gerais, tais como a label do nó, cor, ícone, quantos nós de entrada e saída, etc. Um ponto a ser ressaltado é que um nó pode ter apenas zero ou uma entrada. Em contrapartida, ele pode ter de zero ou mais saídas [17]. O segundo script no HTML corresponde as configurações/opções que são oferecidas pelo nó, o que basicamente corresponde a um formulário HTML. A última conterá a parte descritiva do nó que será renderizada na parte mais à direita da interface do Node-RED quando o nó é clicado uma vez.

[1] Disponível em: <https://nodered.org/docs/creating-nodes/first-node>. Acesso em jul. 2017.

Figura 10 - Exemplo de criação de nó arquivo HTML

```
<script type="text/javascript">
    RED.nodes.registerType('lower-case',{
        category: 'function',
        color: '#a6bbcf',
        defaults: {
            name: {value:""}
        },
        inputs:1,
        outputs:1,
        icon: "file.png",
        label: function() {
            return this.name||"lower-case";
        }
    });
</script>

<script type="text/x-red" data-template-name="lower-case">
    <div class="form-row">
        <label for="node-input-name"><i class="icon-tag"></i> Name</label>
        <input type="text" id="node-input-name" placeholder="Name">
    </div>
</script>

<script type="text/x-red" data-help-name="lower-case">
    <p>A simple node that converts the message payloads into all lower-case
characters</p>
</script>
```

Fonte: Criando seu primeiro nó[1]

Para que vinculação das partes funcione (os três scripts no HTML com o script no servidor), todas devem referenciar um mesmo tipo registrado (register type). No caso do exemplo o tipo registrado escolhido foi o “lower-case”.

3.2 NODE.JS MÓDULOS

Uma das grandes vantagens do Node-RED ser implementado em Node.js, é a possibilidade do uso do repositório NPM para importação e uso das mais variadas bibliotecas implementadas em JS. Para que um nó tenha acesso a qualquer dos pacotes disponíveis, é necessário apenas que seja incluso como dependência do projeto no arquivo package.json, o nome e a versão da biblioteca. Assim, ao ser instalado o nó desejado em qualquer plataforma que possua Node-RED (e Node.js consequentemente), será baixado automaticamente as dependências declaradas.

[1] Disponível em: <https://nodered.org/docs/creating-nodes/first-node>. Acesso em jul. 2017.

Não diferente de outros projetos em Node.js e com o intuito de fazer reuso de funcionalidades já criadas, todas as minhas implementações fizeram uso de módulos disponíveis no NPM. Nas subseções que seguem, é apresentada uma visão geral dos módulos utilizados.

3.2.1 RXJS

RxJS [https://www.npmjs.com/package/rxjs] é a versão do ReactiveX[1] para JavaScript. ReactiveX ou RX é uma API focada na construção de programas baseados em eventos assíncronos que utiliza stream observáveis (observable) [20] [21]. O projeto, de código aberto, é mantido pela Microsoft e possuem diversos adeptos de grande porte além da Microsoft, tais como Netflix, GitHub, Trello, airbnb, etc [20]. Na implementação dos nós, RxJS é utilizado como um dos principais orquestradores na lógica por trás dos nós. Sendo direcionada a manipulação de streams de eventos, a biblioteca oferece alguns dos padrões utilizados por motores de CEP (conforme mostrado no Quadro 1 e 2).

3.2.2 ALASQL

AlaSQL [https://www.npmjs.com/package/alasql] é uma biblioteca JavaScript que permite que desenvolvedores JS possam utilizar SQL em dados em memória. Entre suas diversas funcionalidades, ele permite a criação e manipulação de tabelas em memória, manipulação de arrays através de queries SQL e manipulação de arquivos (JSON, CSV, XLS, etc.) [22]. Estende as funcionalidades oferecidas pela RxJS, no sentido que ele contém algumas operações especialmente voltadas para SQL que não são encontradas em RxJS, como por exemplo todos os tipos de joins utilizados em banco relacionais. Além disso, ele oferece um conjunto abrangente de funções de agregação comumente encontrados em SQL tais como média, soma, máximo, mínimo, etc., além de outras estatísticas, tais como desvio padrão e variância.

3.2.3 BOOLEAN-PARSER

Boolean-parser [https://www.npmjs.com/package/boolean-parser] é outra biblioteca codificada em JS de código aberto que permite converter uma query, que utiliza as palavras chaves/operadores booleanos AND e OR, em um array

[1] http://reactivex.io/

bidimensional que contém todas as possibilidades possíveis de um ponto de vista booleano [23]. Ela foi utilizada em conjunto com o módulo pamatcher para a checagem de padrões em eventos.

3.2.4 PAMATCHER

Pamatcher [https://www.npmjs.com/package/pamatcher] é uma biblioteca implementada em JS de código aberto cuja funcionalidade é bem parecida a de expressões regulares, mas, invés de fazer checagens em strings, ele foca em fazer checagens de padrões em cima de tipo de dados iteráveis do JavaScript (arrays, maps, sets, etc.) [24]. Ela utiliza-se da saída gerada pelo boolean-parser para checar padrões em eventos

3.2.5 LODASH

Lodash [https://www.npmjs.com/package/lodash] é uma biblioteca JS que reuni um conjunto de operadores auxiliares visando facilitar a manipulação de arrays, objetos, variáveis, etc [25]. Lodash foi utilizado no projeto de forma a facilitar a codificação em algumas partes do projeto. Seu uso é especialmente útil no módulo implementado de padrão de eventos, no qual a funcionalidade order-by é utilizada para ordenar dois eventos e assim poder proceder com a checagem de padrão nos eventos.

3.2.6 SAFE-EVAL

Safe-eval [https://www.npmjs.com/package/safe-eval] é outra biblioteca JS de código aberto cujo objetivo é permitir que códigos em strings sejam avaliados de forma segura. Seu uso me assegura que apenas funcionalidades inerentes ao JavaScript sejam executadas, excluindo assim funcionalidades especificas do Node.js que, entre outras coisas, possuem funcionalidades para interagir com o sistema operacional.

4 IMPLEMENTAÇÃO

Este capítulo mostra uma visão geral das funcionalidades implementadas durante a criação deste trabalho. São descritas as funcionalidades usadas, fatores que influenciaram na escolha das mesmas e uma visão geral sobre os nós implementados (cep aggr, cep join e cep pattern). É importante observar que todo o código fonte das implementações está presente no final do trabalho na seção de apêndice.

Ele está organizado do seguinte modo: Seção 4.1 mostra as funcionalidades implementadas, a seção 4.2 fala sobre os fatores de escolha, a seção 4.3 e suas subseções tratam dos nós (componentes) implementados e, finalmente, a seção 4.4 apresenta um exemplo criado, que utiliza os componentes implementados.

4.1 FUNCIONALIDADES IMPLEMENTADAS

As funcionalidades escolhidas foram baseadas nas funcionalidades apresentadas em [14] e especialmente os padrões apresentados em [16], mencionados no capitulo 2. Antes de um estudo mais aprofundado em CEP, um único nó, no qual o usuário poderia através de opções selecionar que tipos de padrões ele gostaria de utilizar, era tida como objetivo. Porém, dada complexidade e levando em conta a essência do Node-RED em criar componentes reusáveis, esta ideia foi abandonada. Em [14], na seção sobre funcionalidades de CEP, é mencionada uma ferramenta, que adota uma representação gráfica que encadeia as operações de CEP, chamada Aurora. Nela, funcionalidades em formato de caixas distintas são conectadas formando assim um fluxo. Já em [16], foi possível verificar, através dos exemplos dados, a existência de certos padrões na construção de queries, que nem sempre utilizam todas as opções, mas apenas algumas especificas à funcionalidade pretendida. Baseado nesses argumentos e evidências, a implementação de diferentes padrões de CEP foi considerada, cada um correspondendo a um nó diferente.

As funcionalidades implementadas foram: agregação, correlações (joins) e casamento de padrões de eventos. Todas elas podem ser encontradas no Quadro 1 e 2. Para composição das opções presentes nos nós, foram utilizados extensivamente os padrões apresentados em [16], devido ao fato de serem bem detalhados através de ilustrações e exemplos escritos numa linguagem de consulta baseada em SQL para motores de CEP. Outros dois padrões, windows e filtros, acabaram por virem juntos, visto seus vínculos na construção dos padrões escolhidos.

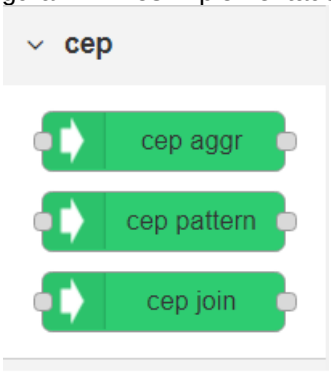
4.2 FATORES DE ESCOLHA

A decisão de que padrões ou funcionalidades seriam implementadas envolveu algumas variantes, entre elas, tempo e complexidade. Após uma análise das funcionalidades e padrões apresentados no Quadro 1 e 2 e leituras adicionais como [26], e levando em conta os dois limitantes mencionados anteriormente, foi analisado, dentre as funcionalidades de CEP, aquelas que seriam mais familiares e que fossem advindas de outras tecnologias comumente utilizadas. Levando em consideração que CEP está de certa forma relacionado com SQL, as primeiras funcionalidades a serem consideradas foram agregação e join. Para o casamento de padrões (pattern matching), foi considerado o fato dela ter uma forte relação com a operação de join [16] e o fato de que CEP está intimamente ligado a detecção de padrões [16]. Além disso, seu uso é bastante interessante levando em conta suas áreas de aplicabilidade (não necessariamente vinculadas diretamente com IoT), tais como detecção de fraudes, monitoramento de processos de negócio e segurança de redes [16].

4.3 NÓS IMPLEMENTADOS

Todo o código está disponível ambos no GitHub [https://github.com/CeZL/node-red-contrib-cep] e no NPM [https://www.npmjs.com/package/node-red-contrib-cep] sob a licença de código aberto MIT. Além disso, o Node-RED possui uma página própria onde é possível pesquisar também por nós que estão disponíveis no NPM e que foram publicados contendo a palavra chave “node-red”. O código assim também pode ser descoberto e acessado através de [https://flows.nodered.org/node/node-red-contrib-cep]. Neles, o usuário que pretender utilizar os nós conta com uma descrição detalhada das opções oferecidas pelos nós assim como o passo de instalação. A instalação em si é simples, requerendo do usuário a execução de uma linha de comando dentro de uma pasta que possui vínculo com o Node-RED (normalmente ~/.node-red). Mesmo que tenham sido três nós diferentes implementados, graças ao sistema de módulos do Node.js, eles puderam ser agrupados em um único pacote. Logo após a instalação, uma nova categoria na seção a esquerda do Node-RED, que engloba todos os nós disponíveis, é criada. O nome para cada um segue o padrão de ter CEP em minúsculo (seguindo o padrão dos nós disponíveis na plataforma) seguido do nome ou abreviação do nome do padrão utilizado. A Figura 11 demonstra como eles aparecem no Node-RED após a instalação.

Figura 11 - Nós implementados



Fonte: Elaborada pelo autor.

Para diminuir a complexidade do projeto, cada nó processa uma quantidade mínima de eventos inerente ao tipo de operação. O Quadro 3 (que está também disponível online em todos os repositórios) sumariza o conjunto de opções disponíveis para cada nó, assim como, a quantidade de eventos suportados por cada um deles.

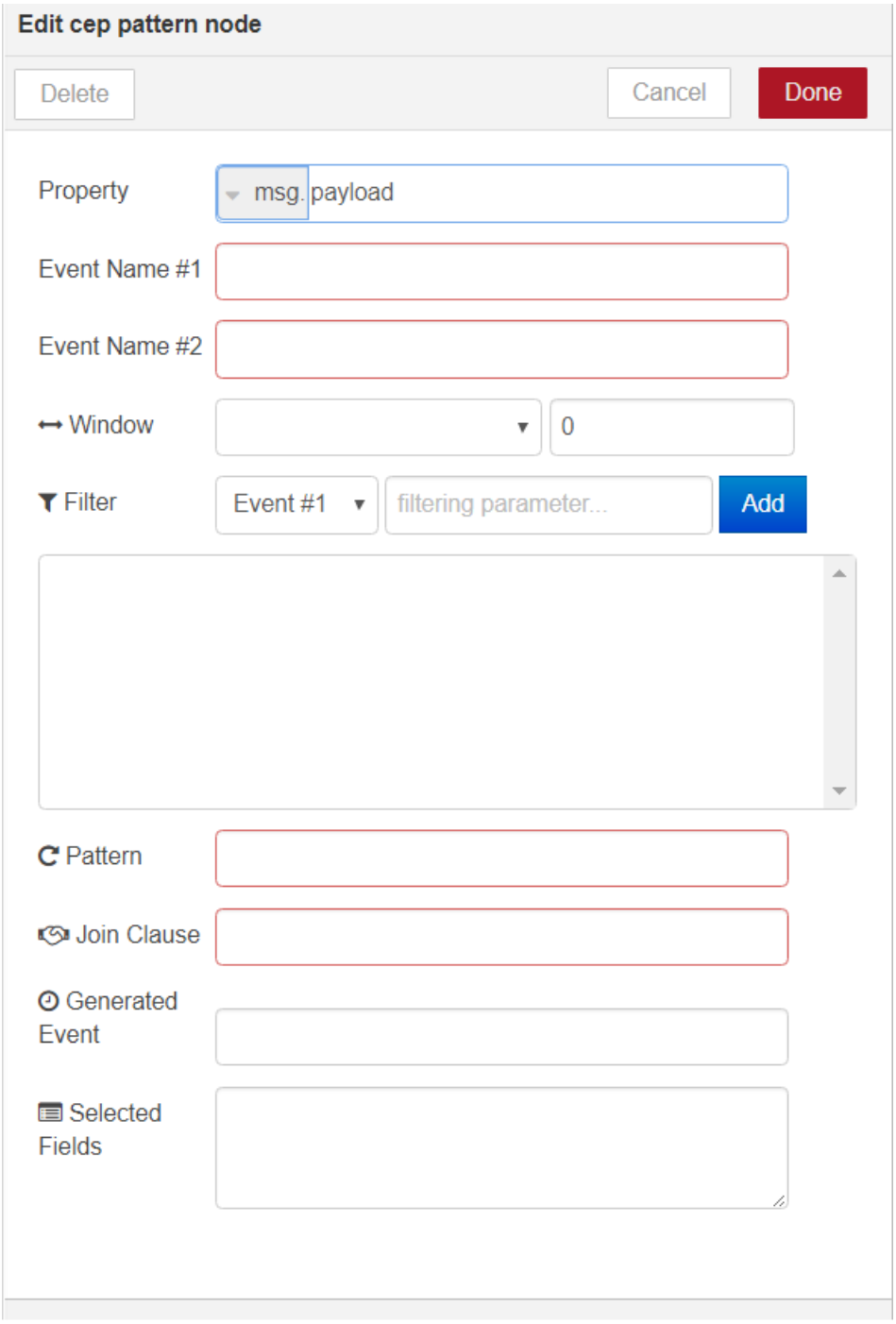
Quadro 3 - Resumo dos nós implementados

Nó	Opções Disponíveis	Número de Eventos
cep aggr	Property, Event Name, Window, Window Parameter, Group By, Having, Filter, AVG, COUNT, MAX, MEDIAN, MIN, STDEV, SUM, VAR, Generated Event, Selected Fields	1
cep pattern	Property, Event Name #1, Event Name #2, Window, Window Parameter, filter, Pattern, Join Clause, Generated Event, Selected Fields	2
cep join	Property, Event Name #1, Window, Window Parameter, Event Name #2, Window, Window Parameter, Filter, Join Clause, Generated Event, Selected Fields	2

Fonte: Elaborada pelo autor.

As opções disponíveis, mostradas no Quadro 3, podem ser acessadas através de uma aba escondida que, por sua vez, pode ser acessada ao arrastar um nó para o canvas e clicar duas vezes nele. A Figura 12 mostra as opções disponíveis para nó de pattern, ressaltando que todos os outros seguem o mesmo estilo.

Figura 12 - Tela de opções disponíveis para o nó pattern



Fonte: Elaborada pelo autor.

Ao clicar uma vez no nó, é possível visualizar a descrição dele na seção mais à direita do Node-RED, o que inclui descrições sobre cada uma das opções disponíveis. Um ponto a ser observado é que as descrições contidas nos repositórios são mais detalhas, visto que, demonstrou-se ser impraticável colocar muitos detalhes na aba de informações do nó devido ao pequeno espaço disponível para o mesmo.

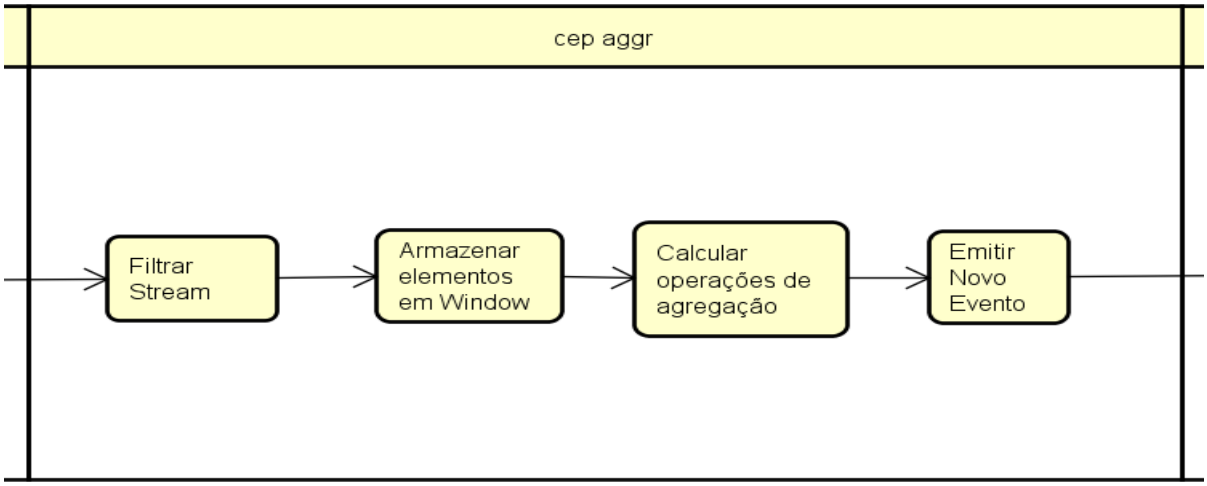
A interação com os campos do nó se dá através de uma propriedade, que é definida nos eventos pela propriedade eventName e que deve estar presente em todos os eventos. Ele funciona de forma semelhante a um alias em tabela SQL, permitindo que as propriedades sejam corretamente selecionadas.

4.3.1 CEP AGGR

O nó de agregação permite o uso de funções de agregação em eventos. Sua funcionalidade pode ser encontrada no Quadro 1 e 2, sob os nomes aggregates e aggregation over windows, respectivamente. Nele é possível aplicar filtragem, selecionar o tipo de janela (window) a ser utilizada, o conjunto de operações de agregação a serem computadas e campos que devem ser replicados nos eventos gerados. As janelas implementadas variam entre window count e window time. Para a primeira, é possível especificar uma quantidade exata de eventos que devem ser armazenados antes do processamento. Já a segunda, recebe como parâmetro uma quantidade que equivalerá a quantidade de tempo, em milissegundos, que a janela deve armazenar eventos. As funções de agregação disponíveis seguem muitas apontadas em [15], que incluem: avg (média), count (contagem), max (máximo), min (mínimo), sum (soma), stddev (desvio padrão), median (mediana) e variance (variância).

A Figura 13 ilustra em alto nível os passos seguidos dentro do nó de agregação para o processamento dos eventos.

Figura 13 - Funcionamento interno do nó de agregação



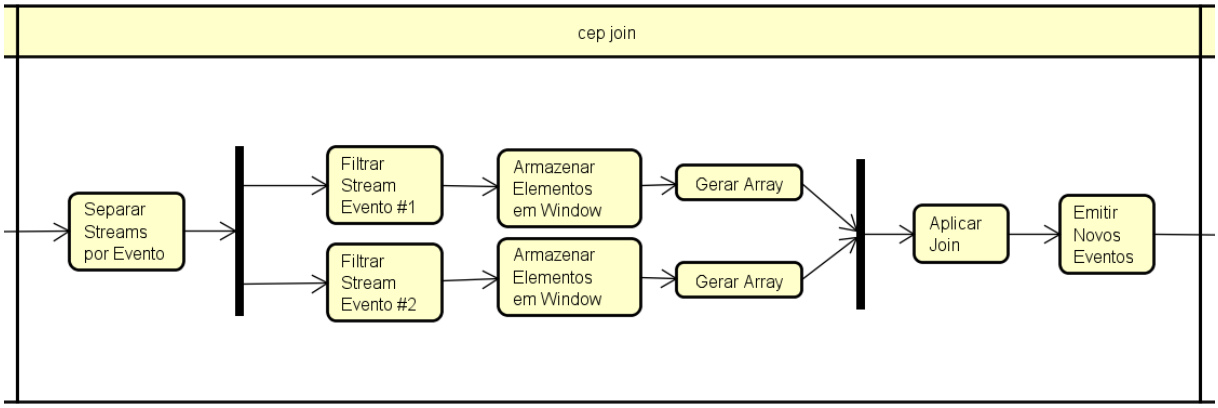
Fonte: Elaborada pelo autor.

4.3.2 CEP JOIN

O nó de join correlaciona eventos baseando-se em atributos relacionados, informados através das opções na GUI. Ele está presente em ambos Quadro 1 e 2. Assim como o de agregação, nele a opção de filtragem também encontra-se disponível com a diferença que o usuário pode especificar para qual evento a filtragem deve ser aplicada, tendo em vista que este nó trabalha com dois eventos. Sabendo disso, para cada tipo de evento também terá um tipo de window diferente, sendo estes tipos os mesmos disponíveis para o nó de agregação. Assim como no de agregação, existe um campo que permite informar que atributos deverão ser inclusos no(s) evento(s) de saída (equivalente a uma operação de select em SQL).

A Figura 14 contém uma lane do diagrama de atividades que exemplifica a sequência de tarefas que são executadas a fim de gerar os novos eventos correlatados.

Figura 14 - Funcionamento interno do nó de join



Fonte: Elaborada pelo autor.

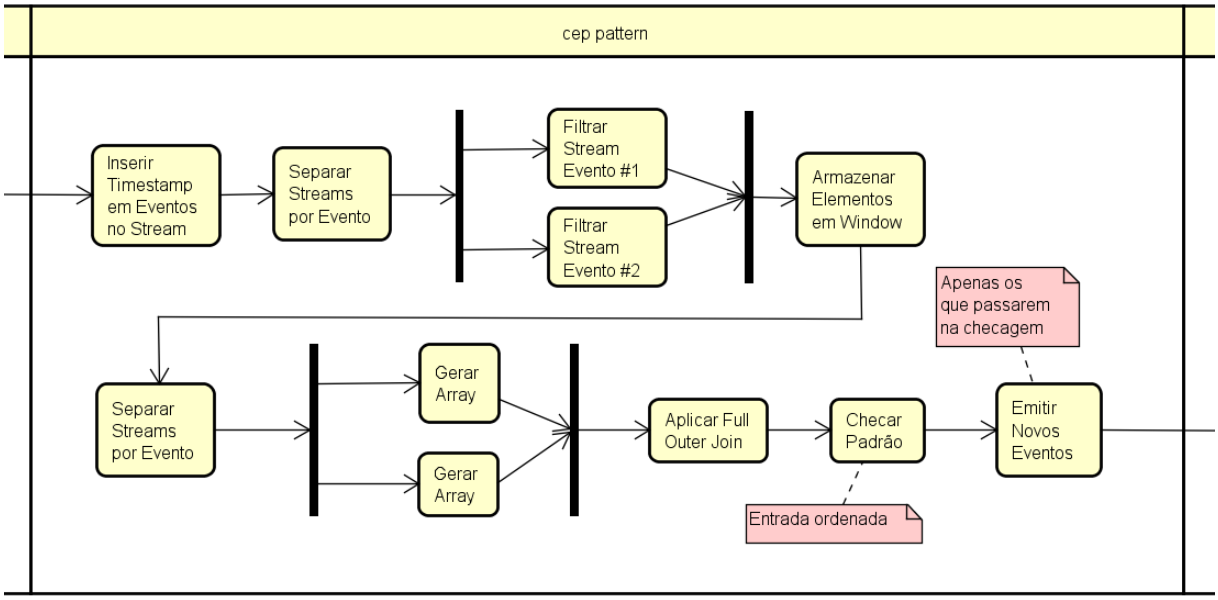
4.3.3 CEP PATTERN

O nó de pattern (padrão) tem como princípio checar a ocorrência de eventos relacionados, levando em conta o tempo como fator de ordem. Tempo se faz necessário tendo em vista a possibilidade expressar o acontecimento de um evento seguido de outro. As opções disponíveis para este nó seguem muitas das opções disponíveis para o nó de join, inclusive o campo de cláusula join. A diferença advém de dois fatores. O primeiro, neste nó só existe a opção de seleção de uma window, seguindo o padrão event pattern matching presente em [16]. O segundo ponto, é que existe um campo para expressar o padrão de ocorrência de eventos. As funcionalidades de ocorrência de eventos seguem os mesmos apontados por “event pattern matching” presentes em [16] e algumas das operações de “logic operators”

[14], com a diferença que por hora não existe ainda a opção de negação (Not) ou repetição (repetition). Desta forma, estão disponíveis para uso os operandos AND, OR e FOLLOWED BY (neste caso implementado como uma vírgula).

A Figura 15 demonstra em alto nível o funcionamento interno do nó de pattern.

Figura 15 - Funcionamento interno do nó de pattern



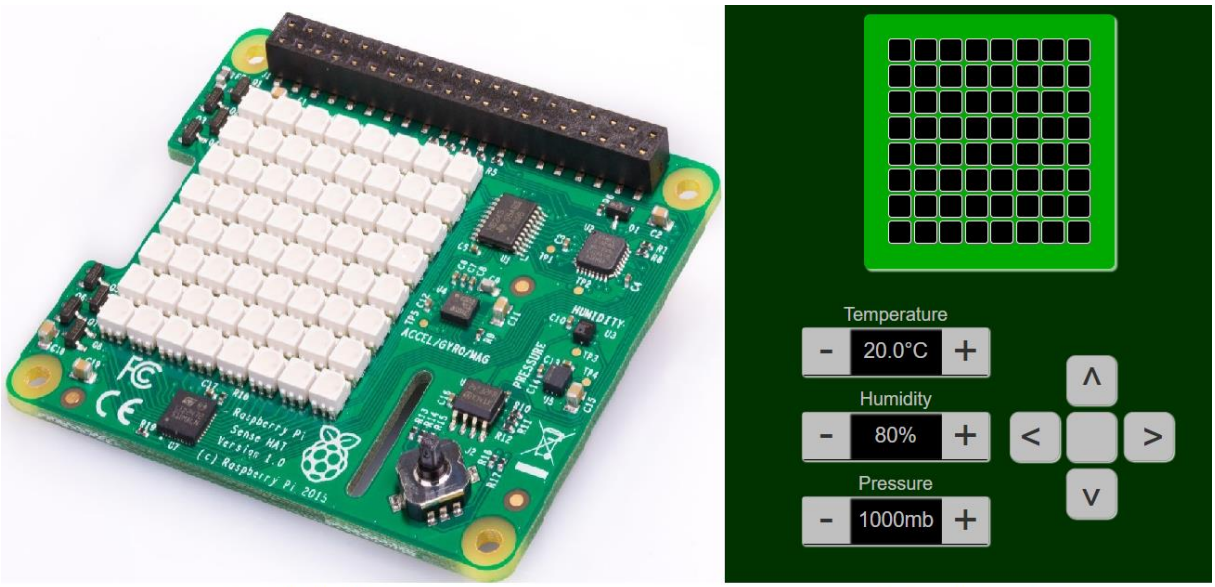
Fonte: Elaborada pelo autor.

4.4 EXEMPLO SENSE HAT

Como forma de demonstrar o uso dos nós, foi desenvolvido um pequeno exemplo que utiliza um simulador desenvolvido pela própria equipe do Node-RED chamado node-red-node-pi-sense-hat-simulator [https://www.npmjs.com/package/node-red-node-pi-sense-hat-simulator]. Ele simula um tipo de HAT (hardware attached on top) chamado Sense HAT que foi desenvolvido especialmente para a missão Astro PI [27]. Na prática, HATs são placas que se encaixam no topo do Rapberry PI e que seguem uma especificação criada pela Raspberry PI Foundation como forma de facilitar o uso de boards no Raspberry PI [28]. Neles podem ser incluídos sensores, atuadores, controladores, etc. de diferentes tipos. No caso específico do Sense HAT, ele possui um conjunto de sensores e atuadores disponíveis, tais como: acelerômetro, giroscópio, LEDs, medidores de temperatura, pressão, etc [29]. O simulador, por sua vez, ainda não implementa todas as funcionalidades disponíveis na placa real. Ao instalá-lo, ele dispõe de pseudo leituras de pressão, temperatura e umidade. Também consegue-se interagir com os atuadores dele, neste caso um conjunto de LEDs, que se encontrariam no topo da

placa real, disponíveis numa interface web. Todas as opções disponíveis podem ser encontradas em seu repositório online. A Figura 16 mostra tanto a placa real à esquerda como a interface WEB na qual o usuário tem acesso ao instalar o simulador no Node-RED. É importante notar que a interface web simula o conjunto de LEDs do Sense HAT e disponibiliza um conjunto de botões, que permitem alterar dados das pseudo leituras, assim como interagir com um joystick.

Figura 16 - Sense HAT



(a) Sense HAT Real (b) Sense HAT Simulador

Fonte: (a) Sense HAT[1] (b) Elaborada pelo autor

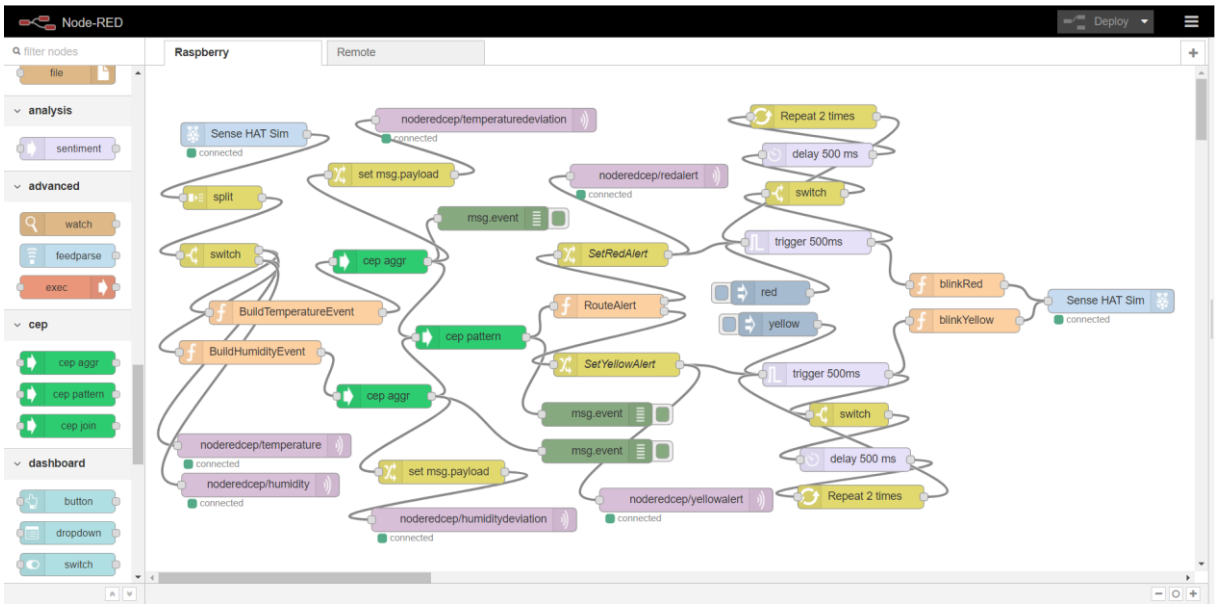
O exemplo basicamente faz uso das leituras de temperatura e umidade, assim como interage com o conjunto de LEDs visíveis na interface web. A ideia dele gira em torno da manipulação dessas leituras e geração de alertas caso uma anomalia, oscilação acima de um determinado valor x das leituras, ocorra. Para isso, a aplicação precisa acumular ambas as leituras, medir a variabilidade delas e só assim tomar alguma decisão.

Na aplicação, o primeiro nó utilizado é o de agregação. Sua escolha se fez necessária graças aos cálculos de estatística que ele possui. A partir do uso do desvio padrão, que mede o quanto dados estão dispersos em relação à média [30], é possível verificar o nível de oscilação das leituras, tanto para mais e menos da média. Se elas não oscilam, este valor será igual a zero. Como as leituras em si não mudam sozinhas, neste cenário é necessário que se altere as mesmas manualmente pela interface web oferecida pelo simulador. Tendo os desvios padrões das mesmas geradas a partir do

[1] Disponível em: <https://www.raspberrypi.org/products/sense-hat/>. Acesso em jun. 2017.

acumulo de dados advindos do simulador em um determinado espaço de tempo, esse resultado é jogado para o próximo nó, o de pattern. O nó de pattern é utilizado com o intuito de checar se ocorreram as oscilações de ambos os tipos, temperatura e umidade, acima de uma determinada tolerância. As combinações de nós podem ser vistas na Figura 17, frisando que os nós em verde mais claro são os nós de cep utilizados no exemplo (no caso, dois de agregação para cada tipo de leitura e um de pattern para checar o padrão nos desvios advindos dos nós de agregação).

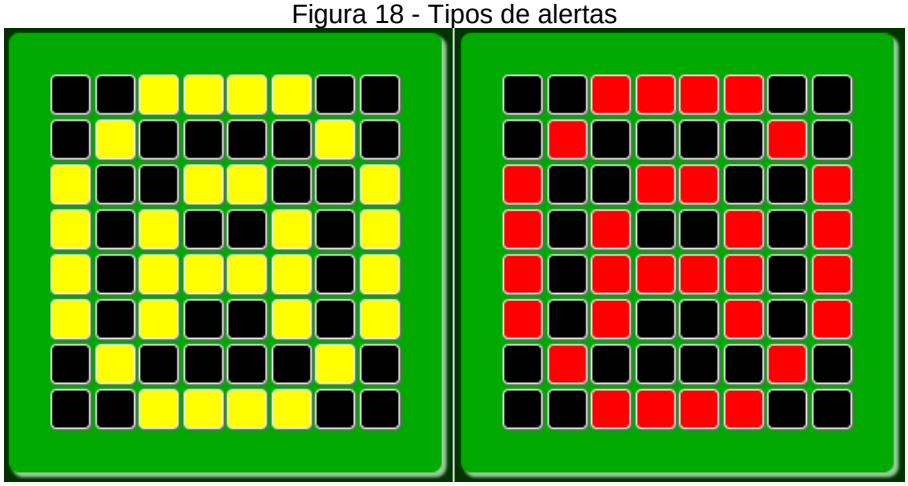
Figura 17 - Exemplo sense HAT implementação



Fonte: Elaborada pelo autor.

Assim que o nó de pattern identifica o padrão em um conjunto de desvios padrões, um novo evento é jogado no fluxo que indicará um alerta a ser gerado nos LEDs do HAT. Como forma de distinguir se houveram anomalias em ambos os desvios ou apenas em um, uma verificação é feita antes de mandar o alerta para os LEDs do HAT. Se houverem anomalias em ambas as leituras, o fluxo gera um alerta em forma de A na cor vermelha que piscam três vezes seguidas com um certo atraso de um para o outro. Caso contrário, apenas uma anomalia, o mesmo alerta é piscado, mas na cor amarela. A Figura 18 mostra os tipos de alertas gerados.

Figura 18 - Tipos de alertas



Fonte: Elaborada pelo autor.

Contudo, esses dados e eventos de certa forma ainda estão presos ao Raspberry PI e ao HAT, quando, num contexto de web das coisas, eles seriam também úteis se fosse possível consumi-los ou visualizá-los através de um serviço ou uma página na web. Através do uso do nó node-red-dashboard [https://www.npmjs.com/package/node-red-dashboard] que permite facialmente a criação de dashboards, um fluxo paralelo foi criado para simular a execução do Node-RED em um dispositivo remoto no qual geraria esse dashboard a partir dos dados coletados da aplicação principal. O envio desses dados, por sua vez, foi feito através de um protocolo de aplicação específico para comunicação M2M (machine-to-machine) baseados no conceito de publish-subscribe [4], que também encontra-se disponível como nó pré-instalado no Node-RED, chamado MQTT. Assim, no fluxo da aplicação principal, foram enxertados nós de saída MQTT nas diversas partes do fluxo, assim como, no pseudo fluxo remoto foram inseridos nós de entradas MQTT responsáveis por receber os dados e alimentar o dashboard.

Os dados enviados para o fluxo remoto são as leituras de temperatura e pressão, cálculos de desvio padrão, assim como alertas que viessem a ocorrer. Com exceção dos últimos tipos de dados, alertas, cada um deles são exibidos em abas diferentes e são constantemente atualizados. Para o alerta, foi configurada sua aparição nas telas em forma de popup com bordas ou amarelas ou vermelhas (indicam o tipo de alerta) que é posicionado no canto superior direito da tela. A medida que alertas são gerados no fluxo principal, é gerado tanto um alerta no dashboard como uma mensagem de áudio informando sobre o alerta. Todas estas informações encontram-se disponíveis nos repositórios da aplicação. Foi compartilhado também

uma pasta do Google Drive[1] [https://drive.google.com/drive/folders/0BzfGWK0OB5yhWXloRnhaOFBhNmc?usp=sharing] que contém capturas de tela detalhando cada ponto específico do exemplo, assim como os fluxos em formato JSON que permitem que qualquer pessoa possa importa-los e executá-los em suas ferramentas Node-RED.

Existem diversas situações na qual este exemplo poderia ser utilizado. Por exemplo, ele poderia ser aplicado em testes com os sensores para testar a precisão nas leituras. Em outros cenários, os alertas gerados poderiam ser usados como indicadores para um problema na atmosfera do ambiente que os sensores se encontram ou até mesmo para um problema com os próprios sensores. Enfim, devem existir outros cenários que poderiam ser explorados, não mencionando o fato que não foram utilizados todos os sensores disponíveis, o que dá margem a criação de exemplos mais elaborados.

[1] https://www.google.com/drive/

5 CONCLUSÃO

Este trabalho teve como objetivo a implementação de operações de processamento de eventos complexos em plataformas que utilizam-se da programação orientada a fluxo de dados para criação de aplicações para Internet das coisas. Nele, foram discutidos os principais conceitos acerca do atual cenário da Internet das coisas, que entre outras coisas incluem web das coisas, que veio para transportar o mundo isolado de IoT para web, e mashups físicos, ferramentas que permitem o rápido desenvolvimento e prototipação de aplicações para IoT e WoT. Assuntos relacionados ao processamento de eventos complexados foram também abordados, focando principalmente em suas funcionalidades, que colaboraram para a construção dos componentes na plataforma de mashups escolhida.

Através de pesquisas, Node-RED mostrou-se ser o candidato mais indicado em termos de mashups físicos encontrados na atualidade. Com seu fácil modelo de programação, componentes podem ser facilmente criados, estendidos e compartilhados, o que vem colaborando para um vasto arsenal de funcionalidades customizadas disponíveis e de uma comunidade altamente ativa que colabora diariamente para o enriquecimento da plataforma.

Por fim, levando em conta alguns limitantes, três funcionalidades de muitas puderam ser implementadas e compartilhadas através de repositórios online. Suas aplicabilidades puderam ser testadas através de um exemplo que misturou leituras de sensores, interação com atuadores e alimentação de gráficos, assim como, geração de alertas através de um dashboard.

5.1 DIFICULDADES ENCONTRADAS

As maiores dificuldades ao longo do projeto foram relacionadas a encontrar a combinação de bibliotecas ideal para satisfazer o propósito do projeto. Por mais que existisse um módulo NPM [https://www.npmjs.com/package/eep] direcionado de certa forma a CEP, ele só empregava parte das funcionalidades, como windows e função de agregação, deixando o restante das manipulações a serem feitas manualmente utilizando o Node.js. Ao optar pelo uso da biblioteca RxJS, voltada para manipulação de eventos, a flexibilidade e o estilo de programação oferecido pelo paradigma reativo funcional foi bastante útil, além de um conjunto rico de métodos prontos para uso.

5.2 TRABALHOS FUTUROS

Conforme discutido, para a construção do trabalho, foram levados em conta apenas algumas funcionalidades de CEP. Dado que muitos dos outros padrões poderão ser também implementados, um dos objetivos para trabalhos futuros será dar continuidade a expansão das funcionalidades disponibilizadas. Para isso, serão levados em conta também melhoramentos das funcionalidades presentes. Posto o fato da biblioteca ser disponibilizada em código aberto, espera-se que a utilização por parte de usuários da comunidade de desenvolvedores possa originar sugestões para melhoria dos componentes. É importante observar que realmente pessoas tem pesquisado sobre o uso de processamento de eventos complexos nas plataformas. Somente no mês de junho do ano de 2017, época na qual a ferramenta foi publicada, noventa e sete downloads haviam sido realizados conforme registrado no repositório do projeto no NPM[1]. Por fim, até a data de publicação da ferramenta, nenhuma outra biblioteca direcionada a CEP mostrava-se ter sido implementada. Ao ser feita uma pesquisa nas bases do NPM[2] ou na biblioteca do Node-RED[3] por “complex event processing” ou termos relacionados, apenas a biblioteca desenvolvida como parte deste trabalho estava disponível.

[1] https://www.npmjs.com/package/node-red-contrib-cep
[2] https://www.npmjs.com/
[3] https://flows.nodered.org/

6 REFERÊNCIAS

1. ROBINS, D. B.; WA, R. **Complex event processing**. In CSEP 504. [S.l.]: [s.n.]. 2010.
2. AL-FUQAHA, A. et al. Internet of things: A survey on enabling technologies, protocols, and applications. **IEEE Communications Surveys & Tutorials**, v. 17, p. 2347-2376, 2015.
3. GUBBI, J. et al. Internet of Things (IoT): A vision, architectural elements, and future directions. **Future generation computer systems**, v. 29, p. 1645-1660, 2013.
4. GUINARD, D.; TRIFA, V. **Building the web of things:** with examples in node. js and raspberry pi. [S.l.]: Manning Publications Co., 2016.
5. KENDA, K. et al. Mashups for the web of things. In: _____ **Semantic Mashups**. [S.l.]: Springer, 2013. p. 145-169.
6. SWAN, M. Sensor mania! the internet of things, wearable computing, objective metrics, and the quantified self 2.0. **Journal of Sensor and Actuator Networks**, v. 1, p. 217-253, 2012.
7. CHEN, C. Y. et al. **Complex event processing for the internet of things and its applications**. Automation Science and Engineering (CASE), 2014 IEEE International Conference on. [S.l.]: [s.n.]. 2014. p. 1144-1149.
8. KLEINFELD, R. et al. **glue. things:** a Mashup Platform for wiring the Internet of Things with the Internet of Services. Proceedings of the 5th International Workshop on Web of Things. [S.l.]: [s.n.]. 2014. p. 16-21.
9. GUINARD, D.; TRIFA, V.; WILDE, E. **A resource oriented architecture for the web of things**. Internet of Things (IOT), 2010. [S.l.]: [s.n.]. 2010. p. 1-8.
10. GUINARD, D. et al. From the internet of things to the web of things: Resource-oriented architecture and best practices. **Architecting the Internet of things**, p. 97-129, 2011.
11. SHENG, M. et al. **Managing the Web of Things:** Linking the Real World to the Web. [S.l.]: Morgan Kaufmann, 2017.
12. BLACKSTOCK, M.; LEA, R. **Toward a distributed data flow platform for the web of things (distributed node-red)**. Proceedings of the 5th International Workshop on Web of Things. [S.l.]: [s.n.]. 2014. p. 34-39.
13. LOBUNETS, O.; KRYLOVSKIY, A. Applying Flow-based Programming Methodology to Data-driven Applications Development for Smart Environments, 2014.
14. CUGOLA, G.; MARGARA, A. Processing flows of information: From data stream to complex event processing. **ACM Computing Surveys (CSUR)**, v. 44, p. 15, 2012.
15. SABOOR, M.; RENGASAMY, R. Designing and developing Complex Event Processing Applications. **Sapient Global Markets**, 2013.
16. CORAL8, I. **Complex Event Processing: Ten Design Patterns**. Coral8, Inc. [S.l.]. 2006.
17. NODE-RED, Documentação. **Node-RED**. Disponivel em: <https://nodered.org/docs/>. Acesso em: 3 jun. 2017.
18. NODE-RED, Flow-based programming for the Internet of Things. **Node-RED**. Disponivel em: <https://nodered.org/>. Acesso em: 25 jun. 2017.

19. OLSSON, J. A. A. J. **Using Node-Red to Connect Patient, Staff and Medical Equipment**. Linkoping University. [S.l.], p. 57. 2016.
20. REACTIVEX, An API for asynchronous programming. **ReactiveX**. Disponivel em: <http://reactivex.io/>. Acesso em: 3 jun. 2017.
21. RXJS, Overview. **ReactiveX**. Disponivel em: <http://reactivex.io/rxjs/manual/overview.html>. Acesso em: 3 jun. 2017.
22. ALASQL, Documentação. Disponivel em: <https://github.com/agershun/alasql>. Acesso em: 8 jun. 2017.
23. BOOLEAN-PARSER, Documentação. **GitHub**. Disponivel em: <https://github.com/riichard/boolean-parser-js>. Acesso em: 10 jun. 2017.
24. PAMATCHER, Documentação. **Pamatcher**. Disponivel em: <http://pamatcher.js.org/>. Acesso em: 5 jun. 2017.
25. LODASH, Documentação. **Lodash**. Disponivel em: <https://lodash.com/>. Acesso em: 5 jun. 2017.
26. UNDERSTANDING Complex Event Processing (CEP). **WSO2**. Disponivel em: <http://wso2.com/library/blog-post/2013/11/understanding-complex-event-processing-(cep)-operators-with-wso2-cep-(siddhi)/>. Acesso em: 5 jun. 2017.
27. SENSE HAT. **Raspberry PI**. Disponivel em: <https://www.raspberrypi.org/products/sense-hat/>. Acesso em: 20 jun. 2017.
28. ADAMS, J. Introducing Raspberry PI HATs. **Raspberry PI**, 2014. Disponivel em: <https://www.raspberrypi.org/blog/introducing-raspberry-pi-hats/>. Acesso em: 5 jun. 2017.
29. SENSE HAT Simulator, Repositório. Disponivel em: <https://www.npmjs.com/package/node-red-node-pi-sense-hat-simulator>. Acesso em: 15 jun. 2017.
30. CORREIA, M. S. B. B. Probabilidade e Estatística, 2003.

APÊNDICE A – Códigos Fontes

As seções seguintes reúnem os códigos fontes das funcionalidades implementadas no desenvolvimento do projeto.

A.1 CEP AGGR

A.1.1 ARQUIVO JS

```
//dependencies
var Rx = require('rxjs/Rx');
var safeEval = require('safe-eval');
var _ = require('lodash');
var alasql = require("alasql");
var util = require('util');

//custom helper functions
var buildLambda = require("./helperFunctions.js").buildLambda;

//queries
const aggregQuery = "SELECT %s FROM ? %s";
const aggregQueryGroupBy = "SELECT %s FROM ? %s GROUP BY %s";
const aggregQueryGroupByHaving = "SELECT %s FROM ? %s GROUP BY %s HAVING
%s";

module.exports = function(RED) {

    function cepAggr(config) {
      RED.nodes.createNode(this, config);

            var node = this;
      //message that will be probably passed on the flow
      var msg = {};

      //getting the values from html
      node.filters = config.filters || [];
      node.property = config.property || "payload";
      node.eventName = config.eventName || "";
      node.windowType = config.windowType || "counter";
      node.windowParam = config.windowParam || 0;

      node.avgAlias = config.avgAlias || "avgAggr";
      node[node.avgAlias] = config.avg;
      node[`${node.avgAlias}Field`] = config.avgField ?
`AVG(${config.avgField})` : "";

      node.countAlias = config.countAlias || "countAggr";
      node[node.countAlias] = config.count;
      node[`${node.countAlias}Field`] = config.countField ?
`COUNT(${config.countField})` : "COUNT(*)";

      node.maxAlias = config.maxAlias || "maxAggr";
      node[node.maxAlias] = config.max;
      node[`${node.maxAlias}Field`] = config.maxField ?
`MAX(${config.maxField})` : "";

      node.medianAlias = config.medianAlias || "medianAggr";
      node[node.medianAlias] = config.median;
```

```
      node[`${node.medianAlias}Field`] = config.medianField ?
`MEDIAN(${config.medianField})` :  "";

      node.minAlias = config.minAlias || "minAggr";
      node[node.minAlias] = config.min;
      node[`${node.minAlias}Field`] = config.minField ?
`MIN(${config.minField})` :  "";

      node.stdevAlias = config.stdevAlias || "stdevAggr";
      node[node.stdevAlias] = config.stdev;
      node[`${node.stdevAlias}Field`] = config.stdevField ?
`STDEV(${config.stdevField})` : "";

      node.sumAlias = config.sumAlias || "sumAggr";
      node[node.sumAlias] = config.sum;
      node[`${node.sumAlias}Field`] = config.sumField ?
`SUM(${config.sumField})` : "";

      node.varianceAlias = config.varianceAlias || "varAggr";
      node[node.varianceAlias] = config.variance;
      node[`${node.varianceAlias}Field`] = config.varianceField ?
`VAR(${config.varianceField})` : "";

      node.newEvent = config.newEvent || "aggregateEvent";
      node.fields = config.fields;

      node.groupby = config.groupby;
      node.having = config.having;

      if(node.windowParam > 0){

        var aliases = [node.avgAlias, node.countAlias, node.maxAlias,
node.medianAlias, node.minAlias, node.stdevAlias, node.sumAlias,
node.varianceAlias];
        var aggregateSelect = "", separator = "";

        for(let i = 0; i < aliases.length; i++){
          if(node[aliases[i]] && node[`${aliases[i]}Field`]){
            aggregateSelect += (separator + node[`${aliases[i]}Field`] +
" AS " + aliases[i]);
            separator = ", ";
          }
        }

        if(node.fields){
          aggregateSelect+= (separator + node.fields);
        }

        if(aggregateSelect){
          //builds the query
          var query = aggregQuery;
          if(node.groupby){
            if(node.having){
              query = aggregQueryGroupByHaving;
              query = util.format(query, aggregateSelect, node.eventName,
node.groupby, node.having);
            }else{
              query = aggregQueryGroupBy;
              query = util.format(query, aggregateSelect, node.eventName,
node.groupby);
            }
```

```
            }else{
              query = util.format(query, aggregateSelect, node.eventName);
            }

            //creates a mapping function according to property informed by
the user
            var mapping = safeEval(buildLambda(`msg.${node.property};`,
"msg"));

            //creates and evaluates filtering function
            for(let i in node.filters){
              node.filters[i] = safeEval(buildLambda(node.filters[i],
node.eventName));
            }

            //creates an Observable/stream from input event and maps each
emition of msg to msg.[property] (informed by the user)
            var inputEvent = Rx.Observable.fromEvent(node,
'input').map(mapping);

            //filtering operations
            for(let i in node.filters){
              inputEvent = inputEvent.filter(node.filters[i]);
            }

            //selects the window
            switch (node.windowType) {
              case "timer": //Time Window
                inputEvent =
inputEvent.windowTime(node.windowParam).concatMap(x => x.toArray());
                break;
              default: //Count Window
                inputEvent =
inputEvent.windowCount(node.windowParam).concatMap(x => x.toArray());
            }

            inputEvent.subscribe({
              next: emitedWindow =>{
                //making sure that the window has really output values
                if(emitedWindow && emitedWindow.length > 0){
                  var result = alasql(query, [emitedWindow]);
                  _.forEach(result, (obj) => {
                    msg.event = {eventName: node.newEvent};
                    _.forEach(obj, (value, key) => {
                      if(value != undefined){
                        if(_.isNumber(value)){
                          msg.event[key] = _.round(value, 10);
                        }else{
                          msg.event[key] = value;
                        }
                      }
                    });
                    //sends message
                    node.send(msg);
                  });
                }
              },
              error: err => node.error('Error: ' + err),
              complete: () => {}
            });
        }
```

```
      }
    }
    RED.nodes.registerType("cepAggr", cepAggr);
}
```

A.1.2 ARQUIVO HTML

```
<script type="text/javascript">
    RED.nodes.registerType('cepAggr',{
        name: "cep aggr",
                category: 'cep',
        color: '#2ecc71',
        defaults: {
                            filters: {value: [], required: false},
            property: {value:"payload", required:true, validate:
RED.validators.typedInput("msg")},
            eventName: {required: true},
            windowType: {value:"count", required: true},
            windowParam: {value: 0, required: true},
            avg: {required: false},
            avgField: {required: false},
            avgAlias: {required: false},
            count: {required: false},
            countAlias: {required: false},
            max: {required: false},
            maxField: {required: false},
            maxAlias: {required: false},
            median: {required: false},
            medianField: {required: false},
            medianAlias: {required: false},
            min: {required: false},
            minField: {required: false},
            minAlias: {required: false},
            stdev: {required: false},
            stdevField: {required: false},
            stdevAlias: {required: false},
            sum: {required: false},
            sumField: {required: false},
            sumAlias: {required: false},
            variance: {required: false},
            varianceField: {required: false},
            varianceAlias: {required: false},
            newEvent: {required: false},
            fields: {required: false},
            groupby: {required: false},
            having: {required: false}
        },
        inputs: 1,
        outputs: 1,
        icon: "arrow-in.png",
        label: function() {
            return this.name||"cep aggr";
        },
        paletteLabel: "cep aggr",
        labelStyle: "cep aggr",
        inputLabels: "event stream",
        outputLabels: "new event",
        oneditprepare: function() {
          var node = this;
```

```
          //creates an typedInput
          $("#node-input-property").typedInput({default:
'msg',types:['msg']});

          //creates an editableList
          $("#node-input-filters-container").css('min-
height','150px').css('min-width','450px').editableList({
            removable: true,
            addButton: false,
            addItem: function(row, index, data) {
                  $(row).html(`filter(${data})`);
            }
          });

          //adds items to editableList from filters
          if(node.filters && node.filters.length != 0){
            for(let i = 0; i < node.filters.length; i++){
             $("#node-input-filters-container").editableList('addItem',
node.filters[i]);
            }
          }

          //registers callback to add itens to editableList
          $("#addFilter").click(function(){
            let filterParameter = $("#filter").val();
            $("#node-input-filters-container").editableList('addItem',
filterParameter);
          });

          $(".aggregate").change(function(){
            if(this.checked){
              if(this.id == "node-input-avg"){
                $(".averageOptions").prop("disabled", false);
              }else if(this.id == "node-input-count"){
                $(".countOptions").prop("disabled", false);
              }else if(this.id == "node-input-max"){
                $(".maxOptions").prop("disabled", false);
              }else if(this.id == "node-input-median"){
                $(".medianOptions").prop("disabled", false);
              }else if(this.id == "node-input-min"){
                $(".minOptions").prop("disabled", false);
              }else if(this.id == "node-input-stdev"){
                $(".stdevOptions").prop("disabled", false);
              }else if(this.id == "node-input-sum"){
                $(".sumOptions").prop("disabled", false);
              }else{
                $(".varOptions").prop("disabled", false);
              }
            }else{
              if(this.id == "node-input-avg"){
                $(".averageOptions").prop("disabled", true);
              }else if(this.id == "node-input-count"){
                $(".countOptions").prop("disabled", true);
              }else if(this.id == "node-input-max"){
                $(".maxOptions").prop("disabled", true);
              }else if(this.id == "node-input-median"){
                $(".medianOptions").prop("disabled", true);
              }else if(this.id == "node-input-min"){
                $(".minOptions").prop("disabled", true);
              }else if(this.id == "node-input-stdev"){
                $(".stdevOptions").prop("disabled", true);
```

```
                }else if(this.id == "node-input-sum"){
                  $(".sumOptions").prop( "disabled", true);
                }else{
                  $(".varOptions").prop("disabled", true);
                }
              }
            });

            if($("#node-input-fields").val()){
              $("#fieldsText").val($("#node-input-fields").val());
            }

            $("#fieldsText").change(function(){
              $("#node-input-fields").val($(this).val());
            });
                  },
              oneditsave: function() {
                    var node = this;

            //cleans the list and readds all the values (new and old) again
                    node.filters = [];
                    var filters = $("#node-input-filters-
container").editableList('items');

                    filters.each(function(i) {
              let filter = $(this).data('data');
                node.filters.push(filter);
                    });
              }
          });
</script>

<script type="text/x-red" data-template-name="cepAggr">

      <div class="form-row">
            <label for="node-input-property"> Property</label>
            <input type="text" id="node-input-property" style="width:
70%"/>
      </div>

      <div class="form-row">
            <label for="node-input-eventName"><i class="fa fa-clock-
o"></i> Event Name</label>
            <input type="text" id="node-input-eventName" style="width:
70%"/>
      </div>

      <div class="form-row">
            <label for="node-input-windowType"><i class="fa fa-arrows-
h"></i> Window</label>
            <select id="node-input-windowType" style="width: 40%">
                  <option value="counter">Count Window</option>
                  <option value="timer">Time Window</option>
            </select>
            <input type="text" id="node-input-windowParam"
placeholder="Window Parameter" style="width: 30%;">
      </div>

  <div class="form-row">
    <label for="node-input-groupby"><i class="fa fa-users"></i> Group
By</label>
```

```
    <input type="text" id="node-input-groupby" style="width: 35%">
    <input type="text" id="node-input-having" style="width: 35%"
placeholder="Having">
  </div>

      <div class="form-row">
            <label for="filter"><i class="fa fa-filter"></i>
Filter</label>
    <input type="text" id="filter" placeholder="filtering parameter..."
style="width: 60%;">
    <button type="button" id="addFilter" class="btn btn-primary"
    style="border-radius:0; box-shadow: none; padding: 6px
12px;">Add</button>
      </div>

      <div class="form-row node-input-filters-container-row">
        <ol id="node-input-filters-container"></ol>
  </div>

  <div id="average" class="form-row" style="margin: 0 5px;">
    <div style="width: 20%; float: left;">
      <input type="checkbox" id="node-input-avg" value="average"
      style="display: inline; width:auto; margin: 13px 0;"
class="aggregate"> AVG
    </div>
    <div style="width: 80%; float: left;">
      <input type="text" id="node-input-avgField" placeholder="Field"
      style="width: 45%; margin: 5px 0;" disabled="true"
class="averageOptions">
      <input type="text" id="node-input-avgAlias" placeholder="Alias"
      style="width: 45%; margin: 5px 0;" disabled="true"
class="averageOptions">
    </div>
  </div>

  <div id="count" class="form-row" style="clear: both; margin: 0 5px;">
    <div style="width: 20%; float: left;">
      <input type="checkbox" id="node-input-count" value="count"
      style="display: inline; width:auto; margin: 13px 0;"
class="aggregate"> COUNT
    </div>
    <div style="width: 80%; float: left;">
      <input type="text" id="node-input-countField" placeholder="Field"
      style="width: 45%; margin: 5px 0;" disabled="true"
class="countOptions" value="*">
      <input type="text" id="node-input-countAlias" placeholder="Alias"
      style="width: 45%; margin: 5px 0;" disabled="true"
class="countOptions">
    </div>
  </div>

  <div id="max" class="form-row" style="clear: both; margin: 0 5px;">
    <div style="width: 20%; float: left;">
      <input type="checkbox" id="node-input-max" value="max"
      style="display: inline; width:auto; margin: 13px 0;"
class="aggregate"> MAX
    </div>
    <div style="width: 80%; float: left;">
      <input type="text" id="node-input-maxField" placeholder="Field"
      style="width: 45%; margin: 5px 0;" disabled="true"
class="maxOptions">
```

```
      <input type="text" id="node-input-maxAlias" placeholder="Alias"
      style="width: 45%; margin: 5px 0;" disabled="true"
class="maxOptions">
    </div>
  </div>

  <div id="median" class="form-row" style="clear: both; margin: 0 5px;">
    <div style="width: 20%; float: left;">
      <input type="checkbox" id="node-input-median" value="median"
      style="display: inline; width:auto; margin: 13px 0;"
class="aggregate"> MEDIAN
    </div>
    <div style="width: 80%; float: left;">
      <input type="text" id="node-input-medianField" placeholder="Field"
      style="width: 45%; margin: 5px 0;" disabled="true"
class="medianOptions">
      <input type="text" id="node-input-medianAlias" placeholder="Alias"
      style="width: 45%; margin: 5px 0;" disabled="true"
class="medianOptions">
    </div>
  </div>

  <div id="min" class="form-row" style="clear: both; margin: 0 5px;">
    <div style="width: 20%; float: left;">
      <input type="checkbox" id="node-input-min" value="min"
      style="display: inline; width:auto; margin: 13px 0;"
class="aggregate"> MIN
    </div>
    <div style="width: 80%; float: left;">
      <input type="text" id="node-input-minField" placeholder="Field"
      style="width: 45%; margin: 5px 0;" disabled="true"
class="minOptions">
      <input type="text" id="node-input-minAlias" placeholder="Alias"
      style="width: 45%; margin: 5px 0;" disabled="true"
class="minOptions">
    </div>
  </div>

  <div id="stdev" class="form-row" style="clear: both; margin: 0 5px;">
    <div style="width: 20%; float: left;">
      <input type="checkbox" id="node-input-stdev" value="stdev"
      style="display: inline; width:auto; margin: 13px 0;"
class="aggregate"> STDEV
    </div>
    <div style="width: 80%; float: left;">
      <input type="text" id="node-input-stdevField" placeholder="Field"
      style="width: 45%; margin: 5px 0;" disabled="true"
class="stdevOptions">
      <input type="text" id="node-input-stdevAlias" placeholder="Alias"
      style="width: 45%; margin: 5px 0;" disabled="true"
class="stdevOptions">
    </div>
  </div>

  <div id="sum" class="form-row" style="clear: both; margin: 0 5px;">
    <div style="width: 20%; float: left;">
      <input type="checkbox" id="node-input-sum" value="sum"
      style="display: inline; width:auto; margin: 13px 0;"
class="aggregate"> SUM
    </div>
    <div style="width: 80%; float: left;">
```

```
      <input type="text" id="node-input-sumField" placeholder="Field"
      style="width: 45%; margin: 5px 0;" disabled="true"
class="sumOptions">
      <input type="text" id="node-input-sumAlias" placeholder="Alias"
      style="width: 45%; margin: 5px 0;" disabled="true"
class="sumOptions">
    </div>
  </div>

  <div id="varAggr" class="form-row" style="clear: both; margin: 0 5px;">
    <div style="width: 20%; float: left;">
      <input type="checkbox" id="node-input-variance" value="variance"
      style="display: inline; width:auto; margin: 13px 0;"
class="aggregate"> VAR
    </div>
    <div style="width: 80%; float: left;">
      <input type="text" id="node-input-varianceField"
placeholder="Field"
      style="width: 45%; margin: 5px 0;" disabled="true"
class="varOptions">
      <input type="text" id="node-input-varianceAlias"
placeholder="Alias"
      style="width: 45%; margin: 5px 0;" disabled="true"
class="varOptions">
    </div>
  </div>

  <div class="form-row" style="clear: both;">
    <label for="node-input-newEvent"><i class="fa fa-clock-o"></i>
Generated Event</label>
    <input type="text" id="node-input-newEvent" style="width: 70%" />
  </div>

  <div class="form-row" style="clear: both; margin-bottom: 200px;">
    <label for="fieldsText"><i class="fa fa-list-alt"></i> Selected
Fields</label>
    <textarea rows="3" style="width: 70%; resize: vertical;"
id="fieldsText"></textarea>
    <input type="text" id="node-input-fields" style="display: none;" />
  </div>

</script>

<script type="text/x-red" data-help-name="cepAggr">
      <p>Aggregation operations to be carried over an event stream.</p>
      <ul>
            <li><b>Property: </b>Indicates which object on
<code>msg</code> carries the event information.</li>
            <li><b>Event Name: </b>Indicates the name of the event. Used
to refer to some attribute from the object that carries the event
properties.</li>
            <li><b>Window: </b>Shows the types of windows that can be
used on the stream.</li>
            <li><b>Window Parameter[number]: </b>Used to inform how many
events a Window should store (count window) or how long(milliseconds) it
should be collecting events (time window).</li>
    <li><b>Group By: </b>A comma separeted list of properties that events
should be grouped by.</li>
    <li><b>Having: </b>Works with Group By. It imposes a constraint based
on aggregate function.</li>
```

```
    <li><b>Filter: </b>You can write any numbers of filters to be used on
event properties. You are allowed to use logic operations as well
following the javascript syntax, e.g. <code>&&</code>, <code>||</code>,
etc.</li>
    <li><b>Aggregate Operators: </b>To perform an operation, you just
need to check a checkbox, inform the property that the operation should
operate on, and optionally an alias.</li>
            <li><b>Generated Event: </b>Indicates the the name of the new
event generated. It is placed on <code>msg.event.eventName</code>.</li>
            <li><b>Selected Fields: </b>A comma separated list of event's
fields that should be included on the output event. Alias can be used
just like SQL using the AS keyword.</li>
      </ul>
</script>
```

A.2 CEP JOIN

A.2.1 ARQUIVO JS

```
//dependencies
var Rx = require('rxjs/Rx');
var safeEval = require('safe-eval');
var alasql = require("alasql");
var _ = require('lodash');
var util = require('util');

//custom helper function
var buildLambda = require("./helperFunctions.js").buildLambda;

//query
const joinQuery = "SELECT '%s' AS eventName %s FROM ? %s INNER JOIN ? %s
ON %s";

module.exports = function(RED) {

    function cepJoin(config) {
      RED.nodes.createNode(this, config);

            var node = this;

      //message that will be passed on the flow
      var msg = {};

      //getting the values from html
      var filters = config.filters || [];
      node.property = config.property || "payload";
      node.eventName1 = config.eventName1;
      var windowType1 = config.windowType1 || "counter";
      var windowParam1 = config.windowParam1 || 0;
      node.eventName2 = config.eventName2;
      var windowType2 = config.windowType2 || "counter";
      var windowParam2 = config.windowParam2 || 0;
      node.joinClause = config.joinClause;
      node.newEvent = config.newEvent || "joinEvent";
      node.fields = config.fields || "";

      if(windowParam1 > 0 && windowParam2 > 0 && node.eventName1 &&
node.eventName2 && node.joinClause){
        node.windows = [];
        node.windows.push({"windowType": windowType1, "windowParam":
windowParam1});
```

```
        node.windows.push({"windowType": windowType2, "windowParam":
windowParam2});

        //creates a mapping function according to property informed by
the user
        var mapping = safeEval(buildLambda(`msg.${node.property}`,
"msg"));

        //split the filters set by the user to be applied on different
stream/observables
        node.filterEvent1 = filters.filter(filterRule =>
filterRule.filterEvent == "Event#1");
        node.filterEvent2 = filters.filter(filterRule =>
filterRule.filterEvent == "Event#2");

        //creates the lambda/arrow functions for the filters
        for(let i in node.filterEvent1){
          node.filterEvent1[i] =
safeEval(buildLambda(node.filterEvent1[i].filterParameter,
node.eventName1));
        }

        for(let i in node.filterEvent2){
          node.filterEvent2[i] =
safeEval(buildLambda(node.filterEvent2[i].filterParameter,
node.eventName2));
        }

        //preparing the join query
        var query = util.format(joinQuery, node.newEvent, node.fields ?
`, ${node.fields}`: node.fields, node.eventName1, node.eventName2,
node.joinClause);

        //creates the observable/stream from input event and maps each
emition of msg to msg.[property] (informed by the user)
        var inputEvent = Rx.Observable.fromEvent(node,
'input').map(mapping);
        var subject = new Rx.Subject();
        var multicasted = inputEvent.multicast(subject);

        //filters the main stream and separates it based on eventName
        var eventStream1 = multicasted.filter(event1 => event1.eventName
== node.eventName1);

        var eventStream2 = multicasted.filter(event2 => event2.eventName
== node.eventName2);

        //filters each stream
        for(let i in node.filterEvent1){
          eventStream1 = eventStream1.filter(node.filterEvent1[i]);
        }

        for(let i in node.filterEvent2){
          eventStream2 = eventStream2.filter(node.filterEvent2[i]);
        }

        //selects the window
        for(var i = 0; i < node.windows.length; i++){
          switch (node.windows[i].windowType) {
            case "timer": //Time Window
              if(i == 0){
```

```
                eventStream1 =
eventStream1.windowTime(node.windows[i].windowParam);
              }else{
                eventStream2 =
eventStream2.windowTime(node.windows[i].windowParam);
              }
              break;
            default: //Count Window
              if(i == 0){
                eventStream1 =
eventStream1.windowCount(node.windows[i].windowParam);
              }else{
                eventStream2 =
eventStream2.windowCount(node.windows[i].windowParam);
              }
          }
        }
        //subscribes to both stream and creates a new stream that will
output both outputs from the two streams subscribed
        var joinStream = Rx.Observable.zip(
                          eventStream1.concatMap(x => x.toArray()),
                          eventStream2.concatMap(x => x.toArray())
                        );

        //subscribes to the stream that contains the result from the
previous two streams
        joinStream.subscribe({
          next: (combinedStream) => {
            if(combinedStream && combinedStream.length == 2){
              //gets each array output from the joinStream
              var arrayEvent1 = combinedStream[0];
              var arrayEvent2 = combinedStream[1];

              if(arrayEvent1 && arrayEvent2){
                //performs the join operation
                let join = alasql(query, [arrayEvent1, arrayEvent2]);
                //loops through the array and sets each object that
represents a join to the node.event reference
                _.forEach(join, value => {
                  msg.event = value;
                  //sends the message/event to the flow
                  node.send(msg);
                });
              }
            }
          },
          error: err => node.error('Error: ' + err),
          complete: () => {}
        });

        multicasted.connect();
      }
    }
    RED.nodes.registerType("cepJoin", cepJoin);
}
```

A.2.2 ARQUIVO HTML

```
<script type="text/javascript">
    RED.nodes.registerType('cepJoin',{
```

```
        name: "cep join",
        category: 'cep',
        color: '#2ecc71',
        defaults: {
                        filters: {value: [], required: false},
             property: {value:"payload", required:true, validate:
RED.validators.typedInput("msg")},
             eventName1: {required: true},
             eventName2: {required: true},
             windowType1: {value:"count", required: true},
             windowParam1: {value: 0, required: true, validate:
RED.validators.number()},
             windowType2: {value:"count", required: true},
             windowParam2: {value: 0, required: true, validate:
RED.validators.number()},
             joinClause: {required: true},
             newEvent: {required: false},
             fields: {required: false}
        },
        inputs: 1,
        outputs: 1,
        icon: "arrow-in.png",
        label: function() {
            return this.name||"cep join";
        },
        paletteLabel: "cep join",
        labelStyle: "cep join",
        inputLabels: "event stream",
        outputLabels: "new event",
        oneditprepare: function() {
          var node = this;

          //creates an typedInput
          $("#node-input-property").typedInput({default:
'msg',types:['msg']});

          //creates an editableList
          $("#node-input-filters-container").css('min-
height','150px').css('min-width','450px').editableList({
            removable: true,
            addButton: false,
            addItem: function(row, index, data) {
                  $(row).html(`filter(${data.filterParameter}) from
${data.filterEvent}`);
            }
          });

          //adds items to editableList from filters
          if(node.filters && node.filters.length != 0){
            for(let i = 0; i < node.filters.length; i++){
             $("#node-input-filters-container").editableList('addItem',
node.filters[i]);
            }
          }

          //registers callback to add itens to editableList
          $("#addFilter").click(function(){
            let filterEvent = $("#filterEvent").val();
            let filterParameter = $("#filter").val();
            $("#node-input-filters-container").editableList('addItem',
{"filterEvent": filterEvent, "filterParameter": filterParameter});
```

```
            });

            if($("#node-input-fields").val()){
              $("#fieldsText").val($("#node-input-fields").val());
            }

            $("#fieldsText").change(function(){
              $("#node-input-fields").val($(this).val());
            });
                  },
              oneditsave: function() {
                    var node = this;

            //cleans the list and readds all the values (new and old) again
                    node.filters = [];
                    var filters = $("#node-input-filters-
container").editableList('items');

                    filters.each(function(i) {
              let filter = $(this).data('data');
              node.filters.push(filter);
                    });
              }
          });
</script>

<script type="text/x-red" data-template-name="cepJoin">

      <div class="form-row">
            <label for="node-input-property"> Property</label>
            <input type="text" id="node-input-property" style="width:
70%"/>
      </div>

      <div class="form-row">
            <label for="node-input-eventName1">Event Name #1</label>
            <input type="text" id="node-input-eventName1" style="width:
70%"/>
      </div>

  <div class="form-row">
            <label for="node-input-windowType1"><i class="fa fa-arrows-
h"></i> Window</label>
            <select id="node-input-windowType1" style="width: 40%">
                  <option value="counter">Count Window</option>
                  <option value="timer">Time Window</option>
            </select>
            <input type="text" id="node-input-windowParam1"
placeholder="Window Parameter" style="width: 30%;">
      </div>

  <div class="form-row">
            <label for="node-input-eventName2">Event Name #2</label>
            <input type="text" id="node-input-eventName2" style="width:
70%"/>
      </div>

      <div class="form-row">
            <label for="node-input-windowType2"><i class="fa fa-arrows-
h"></i> Window</label>
            <select id="node-input-windowType2" style="width: 40%">
```

```
                    <option value="counter">Count Window</option>
                    <option value="timer">Time Window</option>
                </select>
                <input type="text" id="node-input-windowParam2"
placeholder="Window Parameter" style="width: 30%;">
        </div>

        <div class="form-row">
                <label for="filterEvent"><i class="fa fa-filter"></i>
Filter</label>
      <select id="filterEvent" style="width: 20%;">
        <option value="Event#1">Event #1</option>
        <option value="Event#2">Event #2</option>
      </select>
      <input type="text" id="filter" placeholder="filtering parameter..."
style="width: 40%;">
      <button type="button" id="addFilter" class="btn btn-primary"
      style="border-radius:0; box-shadow: none; padding: 6px
12px;">Add</button>
        </div>

        <div class="form-row node-input-filters-container-row">
          <ol id="node-input-filters-container"></ol>
    </div>

    <div class="form-row">
      <label for="node-input-joinClause"><i class="fa fa-handshake-o"></i>
Join Clause</label>
      <input type="text" id="node-input-joinClause" style="width: 70%" />
    </div>

    <div class="form-row">
      <label for="node-input-newEvent"><i class="fa fa-clock-o"></i>
Generated Event</label>
      <input type="text" id="node-input-newEvent" style="width: 70%" />
    </div>

    <div class="form-row">
      <label for="fieldsText"><i class="fa fa-list-alt"></i> Selected
Fields</label>
      <textarea rows="3" style="width: 70%; resize: vertical;"
id="fieldsText"></textarea>
      <input type="text" id="node-input-fields" style="display: none;" />
    </div>
</script>

<script type="text/x-red" data-help-name="cepJoin">
    <p>A join operation to be carried out on two events.</p>
    <ul>
      <li><b>Property: </b>Indicates which object on <code>msg</code>
carries the event information.</li>
      <li><b>Event Name #1 and #2: </b>Indicate the name of the events.
Used to refer to some attribute from the object that carries the event
properties.</li>
      <li><b>Window: </b>Shows the types of windows that can be used on the
streams.</li>
      <li><b>Window Parameter[number]: </b>Used to inform how many events a
Window should store (count window), or how long(milliseconds) it should
be collecting events (time window).</li>
      <li><b>Filter: </b>You can write any numbers of filters to be used on
event properties. You are allowed to use logic operations as well
```

```
following the javascript syntax, <code>&&</code>, <code>||</code>,
etc.</li>
    <li><b>Join Clause: </b>Used to bind events together using properties
from both events.</li>
    <li><b>Generated Event: </b>Indicates the the name of the new event
generated. It is placed on <code>msg.event.eventName</code>.</li>
    <li><b>Selected Fields: </b>A comma separated list of event's fields
that should be included on the output event. Alias can be used just like
SQL using the AS keyword.</li>
  </ul>
</script>
```

A.3 CEP PATTERN

A.3.1 ARQUIVO JS

```
//dependencies
var Rx = require('rxjs/Rx');
var safeEval = require('safe-eval');
var _ = require('lodash');
var alasql = require("alasql");
var pamatcher = require('pamatcher');
var booleanParser = require('boolean-parser');
var util = require('util');

//custom helper functions
var buildLambda = require("./helperFunctions.js").buildLambda;
var stringIdentifier = require("./helperFunctions.js").stringIdentifier;

//pseudo identifiers used to create event names that will be used on the
processing and to insert timestamps to infer
//the order that the events arrive on the node
const pseudoEventName1 = stringIdentifier();
const pseudoEventName2 = stringIdentifier();

const pseudoTimestamp = stringIdentifier();
const pseudoTimestamp1 = stringIdentifier();
const pseudoTimestamp2 = stringIdentifier();

//query
const queryFullOuterJoin = `SELECT '%s' AS eventName, %s.eventName AS
${pseudoEventName1}, %s.eventName AS ${pseudoEventName2},
%s.${pseudoTimestamp} AS ${pseudoTimestamp1}, %s.${pseudoTimestamp} AS
${pseudoTimestamp2} %s FROM ? %s FULL OUTER JOIN ? %s ON %s`;

//pattern to be used on pamatcher
const pattern = "[{repeat: (x) => x}, %s, {repeat: (x) => x}]";

module.exports = function(RED) {

    function cepPattern(config) {
      RED.nodes.createNode(this, config);

            var node = this;
      //message that will be passed on the flow
      var msg = {};

      //getting the values from html
      var filters = config.filters || [];
      node.property = config.property || "payload";
      node.eventName1 = config.eventName1;
```

```
        node.eventName2 = config.eventName2;
        node.windowType = config.windowType || "counter";
        node.windowParam = config.windowParam || 0;
        node.pattern = config.pattern;
        node.joinClause = config.joinClause;
        node.newEvent = config.newEvent || "patternEvent";
        node.fields = config.fields || "";

        if(node.windowParam > 0 && node.eventName1 && node.eventName2 &&
node.pattern && node.joinClause){

        var matcher, fullQuery, parsedPattern = "", parsedArray,
parsedString, separator = "";

        var mapping = safeEval(buildLambda(`msg.${node.property}`, "msg"));

        //separates each filtering op, transform each one in function and
evaluates them
        node.filterEvent1 = filters.filter(filterRule =>
filterRule.filterEvent == "Event#1");
        node.filterEvent2 = filters.filter(filterRule =>
filterRule.filterEvent == "Event#2");

        for(let i in node.filterEvent1){
          node.filterEvent1[i] =
safeEval(buildLambda(`${node.filterEvent1[i].filterParameter}`,
node.eventName1));
        }

        for(let i in node.filterEvent2){
          node.filterEvent2[i] =
safeEval(buildLambda(`${node.filterEvent2[i].filterParameter}`,
node.eventName2));
        }

        // preparing the query
        fullQuery = util.format(queryFullOuterJoin, node.newEvent,
node.eventName1, node.eventName2, node.eventName1, node.eventName2,
node.fields ? `, ${node.fields}`: node.fields, node.eventName1,
node.eventName2, node.joinClause);

        //builds the pattern
        //each comma will be used as delimiter to build the pattern
        node.pattern = node.pattern.split(",");

        for(let i in node.pattern){
          parsedArray =
booleanParser.parseBooleanQuery(node.pattern[i].trim());
          parsedString = JSON.stringify(parsedArray);

          if(parsedArray.length == 1){
            parsedPattern += (separator + parsedString);
          }else{
            parsedPattern += (separator + `{or: ${parsedString}}`);
          }

          separator = ", ";
        }
        parsedPattern = util.format(pattern, parsedPattern);
        parsedPattern = safeEval(parsedPattern);
        matcher = pamatcher(parsedPattern);
```

```
      //creates the Observable from input event and maps each emition of
msg to msg.[property] (informed by the user)
      var inputEvent = Rx.Observable.fromEvent(node,
'input').map(mapping);

      //inserts a timestamp used to control data's arrival
      inputEvent = inputEvent.map(event => {event[pseudoTimestamp] =
(+new Date()); return event});

      var eventStream1 = inputEvent.filter(event1 => event1.eventName ==
node.eventName1);
      var eventStream2 = inputEvent.filter(event2 => event2.eventName ==
node.eventName2);

      for(let i in node.filterEvent1){
        eventStream1 = eventStream1.filter(node.filterEvent1[i]);
      }

      for(let i in node.filterEvent2){
        eventStream2 = eventStream2.filter(node.filterEvent2[i]);
      }
      //merges both streams
      var combined = eventStream1.merge(eventStream2);

      //selects the window
      if(node.windowType && node.windowParam > 0){
        switch (node.windowType) {
          case "timer": //Time Window
            combined = combined.windowTime(node.windowParam);
            break;
          default: //Count Window
            combined = combined.windowCount(node.windowParam);
        }
      }

      combined.subscribe({
        next: (combinedStream) => {
          let arrayStream1 = combinedStream.filter(event1 =>
event1.eventName == node.eventName1).toArray();
          let arrayStream2 = combinedStream.filter(event2 =>
event2.eventName == node.eventName2).toArray();
          //subscribes to each stream and output both the results
together
          var joinStream = Rx.Observable.zip(
                            arrayStream1,
                            arrayStream2
                          );
          joinStream.subscribe({
            next: (arrayStream) => {
              let array1 = arrayStream[0];
              let array2 = arrayStream[1];

              let joinResult = alasql(fullQuery, [array1, array2]);

              //loops trough each object output
              _.forEach(joinResult, (value) =>{
                //temporary objects to check the pattern
                let eventA = {eventName: value[pseudoEventName1],
pseudoTimestamp: value[pseudoTimestamp1]};
```

```
                let eventB = {eventName: value[pseudoEventName2],
pseudoTimestamp: value[pseudoTimestamp2]};
                let comb = [eventA, eventB];

                comb = _.orderBy(comb, ["pseudoTimestamp"], ["asc"]);
                //hack to prevent null since the pamatcher doesn't work
with null but with undefined is ok
                eventA = comb[0].eventName? comb[0].eventName: undefined;
                eventB = comb[1].eventName? comb[1].eventName: undefined;
                comb = [eventA, eventB];

                let result = matcher.exec(comb);

                if(result.test){
                  //removes temporary values
                  value = _.omit(value, [pseudoEventName1,
pseudoEventName2, pseudoTimestamp1, pseudoTimestamp2]);
                  //creates the event with the fields selected by the
user
                  msg.event = value;
                  //sends the message to the output
                  node.send(msg);
                }
              });
            },
            error: err => node.error('Error: ' + err),
            complete: () => {}
          });
        }
      });

      }
    }
    RED.nodes.registerType("cepPattern", cepPattern);
}
```

A.3.2 ARQUIVO HTML

```
<script type="text/javascript">
    RED.nodes.registerType('cepPattern',{
        name: "cep pattern",
        category: 'cep',
        color: '#2ecc71',
        defaults: {
                        filters: {value: [], required: false},
            property: {value:"payload", required:true, validate:
RED.validators.typedInput("msg")},
            eventName1: {required: true},
            eventName2: {required: true},
            windowType: {value:"count", required: true},
            windowParam: {value: 0, required: true, validate:
RED.validators.number()},
            pattern: {required: true},
            joinClause: {required: true},
            newEvent: {required: false},
            fields: {required: false}
        },
        inputs: 1,
        outputs: 1,
        icon: "arrow-in.png",
        label: function() {
```

```
            return this.name||"cep pattern";
        },
        paletteLabel: "cep pattern",
        labelStyle: "cep pattern",
        inputLabels: "event stream",
        outputLabels: "new event",
        oneditprepare: function() {
          var node = this;

          //creates an typedInput
          $("#node-input-property").typedInput({default:
'msg',types:['msg']});

          //creates an editableList
          $("#node-input-filters-container").css('min-
height','150px').css('min-width','450px').editableList({
            removable: true,
            addButton: false,
            addItem: function(row, index, data) {
                  $(row).html(`filter(${data.filterParameter}) from
${data.filterEvent}`);
            }
          });

          //adds items to editableList from filters
          if(node.filters && node.filters.length != 0){
            for(let i = 0; i < node.filters.length; i++){
              $("#node-input-filters-container").editableList('addItem',
node.filters[i]);
            }
          }

          //registers callback to add itens to editableList
          $("#addFilter").click(function(){
            let filterEvent = $("#filterEvent").val();
            let filterParameter = $("#filter").val();
            $("#node-input-filters-container").editableList('addItem',
{"filterEvent": filterEvent, "filterParameter": filterParameter});
          });

          if($("#node-input-fields").val()){
            $("#fieldsText").val($("#node-input-fields").val());
          }

          $("#fieldsText").change(function(){
            $("#node-input-fields").val($(this).val());
          });
                },
            oneditsave: function() {
                  var node = this;

          //cleans the list and readds all the values (new and old) again
                  node.filters = [];
                  var filters = $("#node-input-filters-
container").editableList('items');

                  filters.each(function(i) {
            let filter = $(this).data('data');
            node.filters.push(filter);
                  });
            }
```

```
        });
</script>

<script type="text/x-red" data-template-name="cepPattern">

      <div class="form-row">
            <label for="node-input-property"> Property</label>
            <input type="text" id="node-input-property" style="width:
70%"/>
      </div>

      <div class="form-row">
            <label for="node-input-eventName1">Event Name #1</label>
            <input type="text" id="node-input-eventName1" style="width:
70%"/>
      </div>

  <div class="form-row">
            <label for="node-input-eventName2">Event Name #2</label>
            <input type="text" id="node-input-eventName2" style="width:
70%"/>
      </div>

  <div class="form-row">
            <label for="node-input-windowType"><i class="fa fa-arrows-
h"></i> Window</label>
            <select id="node-input-windowType" style="width: 40%">
                  <option value="counter">Count Window</option>
                  <option value="timer">Time Window</option>
            </select>
            <input type="text" id="node-input-windowParam"
placeholder="Window Parameter" style="width: 30%;">
      </div>

      <div class="form-row">
            <label for="filterEvent"><i class="fa fa-filter"></i>
Filter</label>
    <select id="filterEvent" style="width: 20%;">
      <option value="Event#1">Event #1</option>
      <option value="Event#2">Event #2</option>
    </select>
    <input type="text" id="filter" placeholder="filtering parameter..."
style="width: 40%;">
    <button type="button" id="addFilter" class="btn btn-primary"
    style="border-radius:0; box-shadow: none; padding: 6px
12px;">Add</button>
      </div>

      <div class="form-row node-input-filters-container-row">
        <ol id="node-input-filters-container"></ol>
  </div>

  <div class="form-row">
    <label for="node-input-pattern"><i class="fa fa-repeat"></i>
Pattern</label>
    <input type="text" id="node-input-pattern" style="width: 70%" />
  </div>

  <div class="form-row">
    <label for="node-input-joinClause"><i class="fa fa-handshake-o"></i>
Join Clause</label>
```

```
    <input type="text" id="node-input-joinClause" style="width: 70%" />
  </div>

  <div class="form-row">
    <label for="node-input-newEvent"><i class="fa fa-clock-o"></i>
Generated Event</label>
    <input type="text" id="node-input-newEvent" style="width: 70%" />
  </div>

  <div class="form-row">
    <label for="fieldsText"><i class="fa fa-list-alt"></i> Selected
Fields</label>
    <textarea rows="3" style="width: 70%; resize: vertical;"
id="fieldsText"></textarea>
    <input type="text" id="node-input-fields" style="display: none;" />
  </div>

</script>

<script type="text/x-red" data-help-name="cepPattern">
  <p>A join operation to be carried out on two events.</p>
  <ul>
    <li><b>Property: </b>Indicates which object on <code>msg</code>
carries the event information.</li>
    <li><b>Event Name #1 and #2: </b>Indicate the name of the events.
Used to refer to some attribute from the object that carries the event
properties.</li>
    <li><b>Window: </b>Shows the types of windows that can be used on the
streams.</li>
    <li><b>Window Parameter[number]: </b>Used to inform how many events a
Window should store (count window), or how long(milliseconds) it should
be collecting events (time window).</li>
    <li><b>Filter: </b>You can write any numbers of filters to be used on
event properties. You are allowed to use logic operations as well
following the javascript syntax, e.g. <code>&&</code>, <code>||</code>,
etc.</li>
    <li><b>Pattern: </b> Enables the creation of some patten to be
checked on the events stored on the window. Operators allowed: AND, OR
and <b>,</b>(comma).</li>
    <li><b>Join Clause: </b>Used to bind events together using properties
from both events.</li>
    <li><b>Generated Event: </b>Indicates the the name of the new event
generated. It is placed on <code>msg.event.eventName</code>.</li>
    <li><b>Selected Fields: </b>A comma separated list of event's fields
that should be included on the output event. Alias can be used just like
SQL using the AS keyword.</li>
  </ul>
</script>
```

A.4 PACKAGE.JSON

```
{
  "name": "node-red-contrib-cep",
  "version": "0.0.3",
  "description": "A set of nodes that can be used to create complex event
processing operations",
  "dependencies": {
    "alasql": "0.4.0",
    "boolean-parser": "0.0.2",
    "lodash": "4.17.4",
    "pamatcher": "0.3.0",
```

```
    "rxjs": "5.4.0",
    "safe-eval": "0.3.0"
  },
  "keywords": [
    "node-red",
    "Complex Event Processing",
    "cep"
  ],
  "node-red": {
    "nodes": {
      "cep aggr": "cep/cepAggr.js",
      "cep pattern": "cep/cepPattern.js",
      "cep join": "cep/cepJoin.js"
    }
  },
  "license": "MIT",
  "repository": {
    "type": "git",
    "url": "https://github.com/CeZL/node-red-contrib-cep"
  }
}
```

A.5 FUNÇÕES AUXILIARES

```
function buildLambda(expression, ...parameters){
  let separator = '';
  let parameter = '';
  let i = 0;

  while(i < parameters.length){
    parameter += (separator + parameters[i]);
    separator = ', ';
    i++;
  }

  return `(${parameter}) =>  ${expression}`;
}

function stringIdentifier() {
  return '_' + Math.random().toString(36).substr(2, 9);
}


module.exports = {
  buildLambda: buildLambda,
  stringIdentifier: stringIdentifier
};
```