ARTSIA: Escalabilidade de Aplicações para Sistemas Inteligentes de Transportes em um Ambiente de Computação em Neblina

Thiago Correia Medeiros, Carlos Alberto V. Campos
Programa de Pós-Graduação em Informática (PPGI)
Universidade Federal do Estado do Rio de Janeiro (UNIRIO)
Rio de Janeiro, Brasil
{thiago.correia, beto}@uniriotec.br

***Resumo*—Aumentar a eficiência em escalabilidade de aplicações é uma forma de melhorar a entrega de serviços para os consumidores finais. Este artigo propõe uma plataforma chamada ARTSIA (Architecture for Response Time Sensitive ITS Applications), que possui uma arquitetura para apoiar aplicações para Sistemas Inteligentes de Transportes com requisitos mais rígidos de latência e perdas de requisições utilizando computação em neblina. Visando garantir o mínimo de perdas possíveis em requisições oriundas da camada terminal, ARTSIA conta com a proposta de um algoritmo baseado nas redes neurais autorregressivas e em limiares adaptativos de escalabilidade, para analisar os recursos dos servidores em uma solução de computação em neblina e indicar o melhor momento para aumentar ou diminuir a capacidade da infraestrutura que suporta a aplicação de acordo com a demanda gerada em função das variações em sua carga de trabalho. Além disso, fazendo uso de técnicas e políticas de escalabilidade, este trabalho apresenta uma avaliação experimental de diferentes cenários para gerenciar a escalabilidade. Assim, uma comparação foi realizada entre o algoritmo proposto e algumas técnicas da literatura e os resultados obtidos mostraram uma menor perda de requisições quando se usa o algoritmo proposto.**

***Index Terms*—computação em neblina, computação na nuvem, escalabilidade de aplicações, sistema de transporte inteligente, limiares adaptativos**

I. INTRODUÇÃO

Estima-se que já somos mais de 7,7 bilhões de pessoas no mundo, e este número tende a crescer segundo [1]. Com o aumento populacional há o aumento de cidades e pessoas que vivem nessas cidades. Algumas cidades são planejadas e é possível suportar o aumento demográfico, porém, em sua maioria, as cidades não foram planejadas para suportar todo o aumento que vem sendo observado. Uma das consequências do elevado número de pessoas nas cidades, é o maior volume de trânsito. Com o acúmulo de veículos transitando pelas cidades, despende-se de mais tempo para que uma pessoa chegue ao seu destino. Por exemplo, pessoas que moram em áreas mais afastadas dos grandes centros urbanos utilizam uma grande proporção das horas do seu dia no trânsito para chegar ao seu local de trabalho, quando em comparação às horas trabalhadas. Com relação aos serviços públicos, o trânsito ruim pode prejudicar policiais que demoram a chegar na cena do crime, bombeiros que precisam de urgência para salvar pessoas e até mesmo ambulâncias que carregam pessoas em situação de vida ou morte. As pessoas também são prejudicadas financeiramente uma vez que táxis e veículos de aplicativos aumentam suas taxas por conta do aumento do tempo das corridas, fazendo com o que o consumidor pague mais.

Cidades Inteligentes (Smart Cities) é uma grande área de estudos onde pesquisadores utilizam o método científico para oferecer propostas de melhorias baseadas em dados coletados de vários pontos da cidade e integrações de serviços de forma que os cidadãos tenham algumas melhorias de vida. Especificamente com relação a movimentação das pessoas na cidade, Sistemas Inteligentes de Transporte (Intelligent Transportation Systems - ITS) é uma área de estudo onde se visa coletar, analisar e integrar dados com outros sistemas para poder refinar as informações e oferecer uma melhora na qualidade de vida em relação a transporte para as pessoas nas cidades [2]. Existem algumas soluções de ITS que utilizam a tecnologia chamada Computação na Nuvem (Cloud Computing) para permitir uma interação entre todos os dispositivos dentro da cidade. Em [3] é investigado como diminuir o tempo dos serviços computacionais em VCC (Vehicular Cloud Computing). Já em [4] é introduzido um paradigma de VCC que propõem um modelo de coleta de dados que beneficia as aplicações de ITS. Em [5] é apresentada uma arquitetura de Computação na Nuvem para suportar o gerenciamento de veículos em grandes cidades.

Oferecer alguns tipos de serviços para ITS através de computação na nuvem, por vezes, podem se tornar ineficiente em algumas cidades, por conta do longo tempo de resposta aos serviços. Computação em Neblina (Fog Computing) é uma tecnologia que trabalha com uma camada intermediária entre a nuvem e os consumidores. Duas características que contribuem para adoção de computação em neblina são: alta largura de banda e baixa latência. Computação em neblina trabalha na borda da rede, podendo acessar a nuvem ou não. Nem todas as aplicações precisam enviar dados à nuvem para realizar suas tarefas. Uma tomada de decisão dentro da própria rede e uma resposta mais rápida podem ser a solução para algumas aplicações de ITS que visam melhorar o transporte. O artigo [6] propõe um sistema de coleta de dados através

de sensores em ônibus urbanos. Em [7] é apresentada uma arquitetura cooperativa baseada em computação em neblina para redes de veículos inteligentes. Já em [8] é apresentada a plataforma chamada FogFly, baseada em computação em neblina para resolver problemas de otimização de semáforos.

Antes de desenvolver qualquer solução com computação em neblina é preciso considerar suas limitações. Dentre elas podemos citar: a capacidade restrita de processamento e de armazenamento. Sistemas baseados em neblina possuem recursos limitados, logo, precisam ser bem planejados. Além de planejados, seus recursos precisam ser muito bem gerenciados. Uma infraestrutura de neblina pode suportar várias aplicações, logo, algumas aplicações podem ter mais utilização que outras em determinados momentos do dia. Sendo assim, torna-se necessário alocar mais recursos para uma aplicação quando aumentada sua demanda de uso, assim como é preciso liberar recursos dessa aplicação quando sua demanda diminui, para que outras aplicações possam fazer uso desses mesmos recursos. A literatura denomina essa capacidade de aumentar ou liberar recursos através do termo escalabilidade e, em [9], os autores dizem que a escalabilidade em computação em neblina ainda é um tema a ser trabalhado. Assim, [10] faz uso de computação em neblina para estender a escalabilidade de plataformas IoT/M2M em ambientes industriais. Ainda na área industrial, [11] apresenta uma plataforma de virtualização integrada, onde também é apresentada uma proposta de escalabilidade de aplicações baseada em lógica fuzzy. Com recursos divididos em uma universidade na Itália e outra na Suécia, [12] realiza um experimento onde exibe uma arquitetura que realiza escalabilidade utilizando uma algoritmo geométrico.

Com base em nossa revisão da literatura, consideramos que este é o primeiro trabalho a propor uma plataforma baseada em Computação em Neblina, que gerencia suas aplicações usando técnicas de escalabilidade para atender aplicações de ITS. Como aplicação ITS utilizamos um serviço que busca os pontos de interesse de uma determinada região de acordo com a posição do veículo. Algumas aplicações de ITS, como a aplicação utilizada, possuem serviços em que o tempo de resposta influencia diretamente o valor da informação, de forma a satisfazer as requisições de veículos, para uma melhor tomada de decisão. Dado isto, apresentamos a ARTSIA, uma plataforma para gerenciar recursos de Computação em Neblina fazendo uso de escalabilidade de aplicações ITS que realizam a entrega de informações sensíveis ao tempo.

As principais contribuições deste trabalho são as seguintes:

- Uma arquitetura de computação em neblina proposta para o suporte às aplicações de ITS;
- A proposta de um algoritmo de escalabilidade para Computação em Neblina onde seus limiares de escalabilidade variam com o tempo;
- Avaliação experimental da arquitetura e algoritmo propostos, em que são comparadas algumas abordagens de escalabilidade e a abordagem da ARTSIA através da análise da perda de requisições.

II. Conceitos preliminares

Esta seção apresentará conceitos a respeito de Computação em Neblina e de Escalabilidade de aplicações que serão importantes para o entendimento da nossa proposta.

A. *Computação em Neblina*

Computação em Neblina é um paradigma de computação que tem por objetivo aproximar os serviços oferecidos na computação na nuvem para perto da borda da rede. Serviços esses que podem ser de processamento, armazenamento, gerenciamento de tarefas, entre outros. Os autores em [9] fazem uma alusão de que neblina é a nuvem mais próxima ao chão. Os trabalhos [9], [13] afirmam que soluções de Computação em Neblina devem seguir a arquitetura mostrada na Figura 1. Chamamos de camada na nuvem (camada cloud) a parte que se encontra a maior capacidade computacional. Esta camada pode ser construída por vários datacenters com todo tipo de serviço de computação. Além disso, essa camada permite que sistemas de neblina estendam parte de trabalho para a nuvem. A camada de neblina (camada fog) é uma camada intermediária onde se encontram as aplicações do sistema de neblina. Aplicações de neblina podem dar uma resposta mais rápida ao consumidor por estarem na mesma rede, em uma parte chamada de borda da rede. Assim, essa camada possui recursos limitados de processamento, armazenamento e serviços de rede em relação a camada da nuvem. Porém, consegue oferecer uma menor latência. A camada terminal é onde estão os consumidores das aplicações de neblina. Assim, essa camada é composta de sensores, dispositivos IoT, dispositivos móveis como celulares, tablets, veículos, etc.

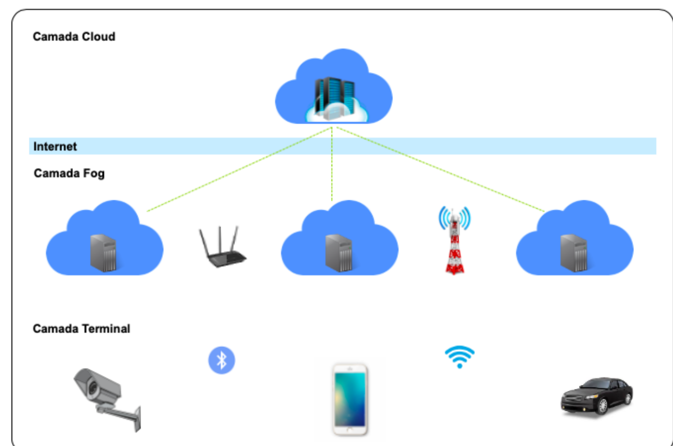

Figura 1. Arquitetura hierárquica em camadas da Computação em Neblina.

B. *Escalabilidade de Aplicações*

Escalabilidade e elasticidade são termos utilizados na literatura para representarem a capacidade de uma arquitetura redimensionar seus recursos para que suas aplicações possam atender uma carga de trabalho variável no tempo oriunda da camada terminal. Esse redimensionamento se dá com o aumento de recursos para utilização da aplicação quando a

carga de trabalho aumenta, o que chamamos de *scale-up*. Da mesma forma, *scale-down* é a liberação dos recursos para que outras aplicações utilizem-os, quando a carga diminuiu.

Segundo [14], os métodos de escalabilidade de serviços podem ser classificados em vertical ou horizontal. Escalar verticalmente um serviço, seria aumentar ou diminuir sua capacidade sem criar instâncias do mesmo. Escalar horizontalmente serviços seria duplicar sua instância e alocá-la para a mesma aplicação. Com relação as políticas que as plataformas usam para escalarem recursos [14] as classificam em: reativa ou proativa. Uma política reativa é quando a plataforma, uma vez definido os limiares [1] para realizar um *scale-up* ou *scale-down*, espera-se que o uso de medidas chegue a esse limiar para que a plataforma tome uma ação. Já na política proativa é quando a plataforma junto a utilização de algum método tenta prever a aproximação dos limiares e age com antecedência.

III. Trabalhos Relacionados

Existem trabalhos que tratam a questão da escalabilidade em Computação em Neblina. Cada trabalho, de acordo com o objetivo da aplicação trata a escalabilidade utilizando métodos e políticas diferentes.

O artigo [10] propõe o uso da Computação em Neblina como uma alternativa à Computação em Nuvem para estender a escalabilidade de plataformas IoT/M2M. Para atender os dispositivos M2M, a plataforma oneM2M [2] possui dois tipos de nó: um em que reside o servidor oneM2M e fica na nuvem; e outro que fica na neblina. A virtualização nos nós da nuvem foi feita utilizando Docker[3] e foi utilizado HAProxy[4] como balanceador de carga. Docker Swarm[5] foi utilizado para trabalhar os Fog Workers como uma Fog Cluster. Foram escolhidas duas abordagens para os testes: estática onde cada Fog Worker possuía 3 instâncias de serviço e dinâmica onde as instâncias do Fog Worker começavam com 1 unidade e poderia escalar até 4 unidades, considerando o limiar de *scale-up* como 80% de utilização do processamento. Para comparação foram utilizadas as medidas: consumo de energia em cada Fog Worker; uso de processamento em cada Fog Worker e o tempo de resposta por requisição.

FRAS é uma plataforma em neblina para virtualização integrada de aplicações industriais apresentada em [11]. A plataforma é composta por um orquestrador, pela rede docker, um gerenciamento de agentes e um balanceador de carga. Os componentes são baseados em contêiner como VNFs (Virtual Network Functions). Cada nó neblina possui uma Docker Engine[6]. O docker engine recebe instruções do orquestrador sobre como gerenciar os contêineres, monitora os recursos do host e envia para o orquestrador e gerencia a rede com todos os contêineres no nó. Com as informações recebidas pelo orquestrador ele decide como alocar os recursos na plataforma.A proposta de escalabilidade de recursos no FRAS é baseada em teoria fuzzy e leva em consideração a carga de processamento, uso de memória e o uso de redes. A solução oferece uma escala de serviço com menor atraso médio e taxa de erro em comparação aos esquemas da Amazon AWS, DC/OS[7], EWMA e C-Scale.

Em [12] é proposta uma arquitetura em neblina para dispositivos IoT. A arquitetura da proposta segue a arquitetura hierárquica de computação em neblina com três elementos: camada IoT, onde estão os sensores e acionadores; camada neblina, onde estão os nós que realizam os cálculos estatísticos das informações dos sensores e realiza o monitoramento dos seus próprios recursos e; camada nuvem, que coordena e orquestra a distribuição de recursos na camada neblina. Os nós de neblina são virtualizados em contêineres utilizando docker e cada nó possui ao menos 3 instâncias: uma com Broker MQTT[8], uma com MongoDB[9] e a outra com a aplicação de negócio. O dispositivo físico dos nós de neblina são RaspberryPi[10] e as imagens utilizadas pelo Docker são buscadas no serviço de registro do Docker Hub[11]. A proposta prevê que caso um nó em neblina esteja próximo de não atender a demanda de requisições ele deve informar a nuvem para que ela decida se irá atender essa demanda ou irá repassar para outro nó na camada neblina. Para isso, os autores propuseram um monitoramento geométrico proativo considerando a utilização dos recursos de processamento, memória e disco.

IV. Arquitetura do ARTSIA

Para apresentar nossa proposta de escalabilidade em computação em neblina, especificamente para aplicações ITS criamos uma arquitetura especializada que segue a arquitetura de Computação em Neblina sendo dividida em três camadas: nuvem, neblina e terminal, sendo ilustrada na Figura 2.

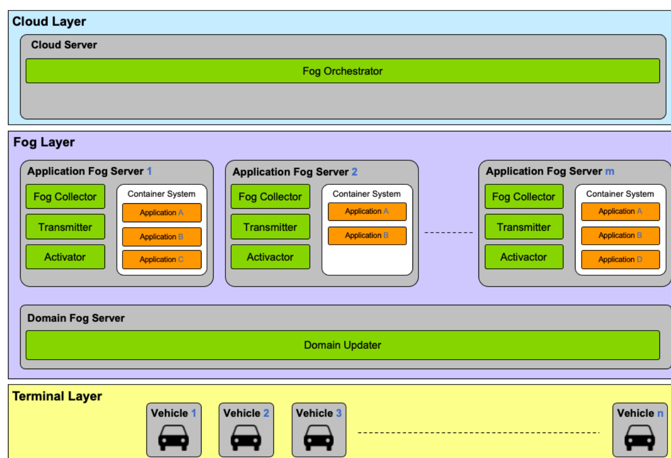

Figura 2. Representação da arquitetura em camadas do ARTSIA.

[1] Neste trabalho usaremos o termo limiar para referenciar o termo *threshold*.
[2] https://www.onem2m.org/
[3] https://www.docker.com/
[4] http://www.haproxy.org/
[5] https://docs.docker.com/engine/swarm/
[6] https://docs.docker.com/engine/
[7] https://dcos.io/
[8] https://mosquitto.org/
[9] https://www.mongodb.com/
[10] https://www.raspberrypi.org/
[11] https://hub.docker.com/

A camada terminal é onde ficam os clientes de aplicações que consomem e fornecem informações as aplicações ITS. Essas interações com as aplicações ITS se dão através da comunicação de duas camadas: camada terminal e camada de neblina. Na nossa proposta consideramos que esses clientes em sua maioria serão os veículos.

A camada de neblina é a principal camada da arquitetura. Nesta camada ocorre a maior parte do processamento. Os veículos (clientes) se comunicam com as aplicações diretamente após serem endereçadas com o auxílio do Atualizador de Domínios. Ainda na camada de neblina, existem os servidores de aplicação. Nesses servidores estão publicadas as aplicações ITS (representados na Figura 2 pelos Serviços de Negócio). Cada servidor de neblina possui um sistema de contêiner onde é possível fazer virtualização de várias aplicações ITS no mesmo servidor de neblina. O sistema de contêiner permite que os recursos do servidor (processamento, memória, armazenamento, etc) sejam compartilhados entre as aplicações. Com esse compartilhamento o sistema de contêiner automaticamente readequa a utilização de recursos de cada aplicação de acordo com sua carga de trabalho. Porém, como esses recursos de um servidor de neblina são limitados, a arquitetura prevê que hajam *m* servidores de neblina, para que uma aplicação possa ser escalada entres os mesmos.

A arquitetura do ARTSIA divide o sistema de neblina em 5 módulos: Atualizador de Domínios, Transmissor, Coletor Fog, Orquestrador Fog e Ativador. Cada módulo é independente e toda a comunicação entre os módulos é feita de forma assíncrona. Desta forma, se algum módulo ou servidor ficar sem funcionar por um tempo, o funcionamento do sistema não será interrompido. Todas as variáveis utilizadas tanto na arquitetura quanto no algoritmo proposto estão descritas na Tabela I.

Tabela I
VARIÁVEIS UTILIZADAS NA ARQUITETURA E NO ALGORITMO PROPOSTO.

Variável	Descrição
Tad	intervalo de execução do Atualizador de Domínios
Tesa	tempo de inicialização de uma instância
Tcf	intervalo de execução no Coletor Fog
Tcc	intervalo de execução do Transmissor
To	intervalo de execução do Orquestrador Fog
Ta	intervalo de execução do Ativador
Pa	pontos de base do algoritmo
Pp	ponto de previsão do algoritmo
Qtd	quantidade de medidas utilizadas
Tmd	tempo mínimo de disponibilidade de uma instância

A. Atualizador de Domínios

O módulo Atualizador de Domínios tem a responsabilidade de manter atualizado quais são os servidores disponíveis para o consumo de aplicações pela camada terminal. Cada aplicação ativa pode estar em um ou mais servidores. A inclusão ou exclusão de um servidor na lista do Atualizador de Domínios se dá exatamente pelo efeito da escalabilidade de aplicações. Como observado na Figura 2, é exemplificado o Serviço B nos Servidores Fog de Aplicação 1 e 2, logo o Atualizador de Domínio deve trabalhar para que haja apontamentos para o Serviço A, tanto no Servidor Fog de Aplicação 1, quanto no Servidor Fog de Aplicação 2. Existe somente um Servidor Fog de Domínios, no qual está o módulo Atualizador de Domínios. A cada tempo que chamaremos de *Tad* é realizado uma coleta de informações do Servidor Cloud sobre quais servidores e aplicações estão disponíveis para consumo. Após identificar que deve disponibilizar um domínio de aplicação para consumo, este módulo aplica um tempo de espera *Tesa* antes de disponibilizar a aplicação. Este tempo se deve ao fato de que uma aplicação leva um tempo de carregamento para estar disponível. Se forem impostas requisições a ela antes deste tempo, as mesmas serão perdidas.

Alguns trabalhos, como em [10], [11], seguem o modelo de nuvem e utilizam balanceadores de carga. Essa solução tem se mostrado eficiente quando trabalhada com recursos como o uso de processamento e/ou memória como insumo para algoritmos de escalabilidade. Como dito na introdução, o ARTSIA é para aplicações ITS onde o tempo de resposta precisa ser curto devido à importância da informação no momento. Esse requisito torna a vazão (throughput) um recurso principal para os cálculos de escalabilidade. Logo, o balanceador neste caso sofre um gargalo, tornando a solução ineficiente. Sendo assim, o módulo de Atualizador de Domínios torna-se uma boa opção, por que ele não recebe a carga de trabalho, mas mantém a rede sempre atualizada de quais aplicações estão disponíveis e onde estão.

B. Coletor Fog

O módulo Coletor Fog tem a responsabilidade de coletar medidas dos recursos físicos do Servidor Fog de Aplicação em que ele se encontra para que seja utilizado como insumo na tomada de decisão do Orquestrador Fog. Exemplo de medidas que podem ser coletadas são percentual de processamento, percentual de memória, taxa de uso de disco rígido, etc. O Orquestrador Fog é quem decide sobre as ações de escalabilidade dentro do sistema de neblina. Cada Servidor Fog de Aplicação possui um módulo do Coletor Fog.

Cada recurso é representado por *Ra*, *Rb*, *Rc*, ... , *Rz*. Cada evento de coleta *EC* é coletado as medidas *MRa*, *MRb*, *MRc*, ... , *MRz*. A cada tempo *Tcf* ocorre um evento de coleta que é representado por *EC1*, *EC2*, *EC3*, ... , *ECn*.

C. Transmissor

O módulo Transmissor tem a responsabilidade de enviar as medidas de recursos coletadas pelo Coletor Fog para o Servidor Cloud. Cada Servidor de Aplicação Fog possui um módulo do Transmissor. Existe uma relação de confiança entre cada Servidor Fog de Aplicação e o Servidor Cloud. A cada tempo *Tcc* é realizado o envio com as medidas coletadas dos recursos para o Servidor Cloud.

D. Orquestrador Fog

O Orquestrador Fog tem a responsabilidade de tomar a decisão de qual servidor Servidor Fog de Aplicação precisa

habilitar/desabilitar a utilização de uma aplicação de negócio (*scale-up*/*scale-down*), baseado nas medidas coletadas nos Servidor Fog de Aplicação e previamente armazenadas no Servidor Cloud,.

Só existe um Servidor Cloud e dentro dele um módulo Orquestrador Fog. A cada tempo *To* o Orquestrador Fog realiza a leitura das medidas dos Servidores Fog de Aplicação que já estão armazenadas no Servidor Cloud, pois foram enviadas por cada Servidor Fog de Aplicação através do módulo Transmissor. A seguir, ele aplica um algoritmo de escalabilidade que com base das medidas lidas, fornece instruções de escalabilidade para cada servidor Servidor Fog de Aplicação e armazena no próprio Servidor Cloud. Tais instruções são utilizadas por cada módulo Ativador.

Em [10] é apresentado um orquestrador de neblina na camada de neblina. Já em [12], [15] e [16] colocam este módulo na camada da nuvem. No ARTSIA este módulo fica na nuvem por uma questão de manutenibilidade. A perda deste serviço no sistema, estando ele na nuvem, pode se dar por um problema no servidor ou uma quebra de link de internet. Ambas situações são improváveis de ocorrer, e caso ocorra pode ser corrigida rapidamente e de forma remota. Se este serviço estivesse na camada de neblina, precisaria do deslocamento de um técnico até o ponto de localização do servidor para seu restabelecimento.

E. Ativador

O módulo Ativador tem a responsabilidade de habilitar/desabilitar o uso de uma aplicação de negócio (*scale-up*/*scale-down*). Cada Servidor Fog de Aplicação possui um módulo Ativador. A cada instante *Ta* o Ativador busca no Servidor Cloud quais instruções o Orquestrador Fog definiu que o Servidor Fog de Aplicação em que o Ativador está precisa fazer no momento com relação a escalabilidade de aplicações. Partindo do princípio que o ativador que consulta a nuvem e não o inverso contribui para independência da camada de neblina. Essa independência é importante no contexto de ITS e é um diferencial em relação aos trabalhos [12] e [16].

F. Algoritmo Proposto

Para tentar obter melhores resultados propomos um algoritmo baseado em uma política proativa de escalabilidade, a qual chamamos de SABANN (Scalability Algorithm Based on Neural Networks). Nesse algoritmo utilizamos uma técnica de predição para séries temporais baseada em redes neurais autorregressivas (ARNN). Em [14] é apresentada uma revisão de algumas técnicas de séries temporais para trabalhar com escalabilidade de recursos para computação na nuvem, porém não cita as redes neurais. Na pesquisa [17] foi feito uma revisão sistemática sobre metodologias de redes neurais artificiais para séries temporais. Nessa revisão foi feito um critério de pontuação para as técnicas encontradas na pesquisa. Dentre as selecionadas, a ARNN foi uma das mais bem avaliadas. Para utilizar a ARNN é preciso informar a quantidade de medidas anteriores *Pa* como entrada e qual o ponto futuro de saída *Pp*. Ainda assim, consideramos o fato de que servidores de neblina podem ter configurações de hardware diferentes e estar sofrendo cargas de trabalho diferentes por estarem aptos a trabalhar com mais de uma aplicação. Por essa razão, aplicamos o que chamamos de limiares adaptativos. Limiares adaptativos analisam as últimas medidas *Qtd*, e com base nisso definem novos valores de limiares para *scale-up* e *scale-down*, a cada execução do algoritmo. Também foi considerado o tempo que mudou o estado da aplicação para disponível, o qual chamamos de *Tesa*. Por diversas vezes verificamos que logo após um *scale-down* os algoritmos realizaram um *scale-up*. Isso causava um consumo de processamento desnecessário, por vezes, permitindo perdas de requisições da camada terminal. Por essa razão, utilizamos um tempo mínimo em que uma instância precisa ficar disponível para que não impacte a execução, o que chamamos de *Tmd*. O Algoritmo 1 representa os passos que foram utilizado pelo SABANN utilizando os procedimentos descrito nesta seção.

Algorithm 1 Algoritmo SABANN

```
Require: Qtd > 0 ∨ Tesa > 0 ∨ Tmd > 0 ∨ Pa < Qtd ∨ Pp < Pa
  servidores ← lerServidoresAtivos()
  limiares ← lerLimiares()
  ultimoMovimento ← lerUltimoMovimento()
  for all servidor ∈ servidores do
      medidas ← filtrarMedidas(servidor)
      if size(medidas) > Qtd then
          medidas ← filtrarUltimasMedidas(Qtd)
          limiares ← calcularLimiares(medidas)
      end if
  end for
  escreverLimiares(limiares)
  ultimoMovimento ← lerUltimoMovimento()
  acao ← pegarAcao(ultimoMovimento)
  tempo ← pegarTempo(ultimoMovimento)
  if isDown(acao)or(isUp(acao)andtempo > (Tesa + Tmd)) then
      scaleUp ← FALSE
      scaleDown ← FALSE
      for all servidor ∈ servidores do
          medidas ← filtrarMedidas(servidor)
          limUp ← pegarLimiarUp(servidor, limiares)
          limDown ← pegarLimiarDown(servidor, limiares)
          pontoFuturo ← ARNN(medidas, Pa, Pp)
          if previsao > limUp then
              scaleUp ← TRUE
          end if
          if previsao < limDown then
              scaleDown ← TRUE;
          end if
      end for
      if scaleUp then
          escalarServidorFog()
      end if
      if scaleDown then
          retirarServidorFog()
      end if
  end if
```

V. Experimentos e Resultados

Nesta avaliação experimental buscamos validar as seguintes hipóteses:

- H1 Quando aplicadas políticas de escalabilidade proativas, a taxa de perda de requisições não menores que quando se utilizam políticas reativas.
- H2 Políticas de escalabilidade proativas apresentam uma variabilidade dos resultados obtidos durante o experimento menor que políticas reativas.

H3 O algoritmo ARTSIA deve apresentar a menor taxa de perda de requisição em comparação com as outras propostas.

A avaliação experimental foi dividida em duas fases: identificação de recursos e execução completa. Para emular os veículos da camada terminal foi utilizada uma ferramenta chamada Apache JMetter[13] na versão 5.3. Com essa ferramenta foi possível emular vários veículos emitindo uma grande quantidade de requisições. Para rodar essa ferramenta foi utilizado um computador na plataforma Windows com 1,7 GHz Intel Core i5 com 6 GB de memória RAM.

A Figura 3 mostra a carga de requisições gerada nos experimentos. A carga dura 30 minutos na forma de uma "pirâmide" de threads a cada 10 minutos. A geração de carga começa com 300 threads e tem seu pico em 7.300 threads. O objetivo da criação das "pirâmides" é emular uma carga onde pudessem ocorrer necessidades de *scale-up* e *scale-down* para de fato testarmos a escalabilidade nos cenários avaliados.

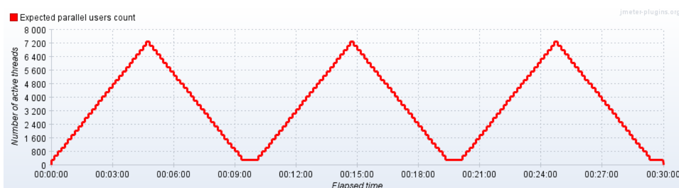

Figura 3. Carga de Requisições no JMetter

A seguir demonstraremos a implementação da plataforma ARTSIA que foi utilizada nos experimentos. Na Seção V-B foram também criados quatro cenários para comparar as técnicas já utilizadas em trabalhos anteriores e o algoritmo SABANN dentro da plataforma ARTSIA.

A. *Implementação*

A comunicação entre a camada terminal e camada de neblina se dá pelo protocolo HTTP. A aplicação desenvolvida para o experimento fornece serviços através do estilo arquitetural REST. Já a comunicação entre a camada de neblina e a camada da nuvem utiliza o protocolo SSH (Secure Shell).

O Coletor Fog realiza a coleta de recursos e armazena em um arquivo CSV (Comma Separated Values). Os recursos utilizados foram vazão *Ra* e processamento *Rb*. Para cada evento de coleta *EC* foi definido o *Tcf* de 5 segundos e são coletadas a vazão em Mbits/s *MRa* e o processamento em percentual de utilização *MRb*. O valor de 5 segundos foi escolhido após verificarmos que o tempo de coleta levou aproximadamente 5 segundos, sendo assim, em nossa implementação, é o tempo mínimo. O script de execução foi desenvolvido em Shell Script (SS) e ele executa uma aplicação de coleta desenvolvida em Java. Para a coleta de processamento foi utilizada uma biblioteca chamada Java Sigar como no artigo [18]. Para coletar a vazão foi utilizada a ferramenta linux Dstat[12].

[13]https://jmeter.apache.org/
[12]https://linux.die.net/man/1/dstat

Foi definido para o módulo Transmissor o *Tcc* de 10 segundos. O tempo de 10 segundos foi escolhido pois após testes, verificamos que é o tempo otimizado, por não impactar nos outros módulos. A cada *Tcc* o módulo pega os dados obtidos pelo Coletor Fog previamente armazenado em CSV, envia esses dados ao Servidor Cloud, e limpa o arquivo CSV de forma que nunca se tenha dados repetidos. O script de execução foi desenvolvido em SS e o envio é feito utilizando uma ferramenta linux SCP (Secure Copy). A relação de confiança entre os servidores da neblina e o servidor da nuvem é feita utilizando uma chave privada na comunicação entre eles.

No Orquestrador Fog foi definido um *To* de 5 segundos. O tempo de 5 segundos foi escolhido pois após testes, verificamos que é o tempo ótimo considerando pegar sempre dados recentes. O script de execução foi desenvolvido em SS e o algoritmo foi feito em linguagem de programação R. Foi utilizado um algoritmo para cada cenário V-B.

Para o Ativador foi definido um *Ta* de 10 segundos. O tempo de 10 segundos foi escolhido pois verificamos que é o tempo otimizado que não causa impacto nos outros módulos. Utilizando SS o Ativador busca as instruções da nuvem com SCP e utiliza uma aplicação Java que verifica a necessidade de realizar um *scale-up* ou *scale-down*.

Foi definido para o Atualizador de Domínio o *Tad* de 5 segundos. O atualizador de módulos é reflexo do resultado do Orquestrador Fog, sendo assim não faz sentido ter um tempo diferente do utilizado no Orquestrador Fog. Também foi configurado *Tesa* como 15 segundos. Já este valor foi escolhido após várias observações da aplicação iniciando. Utilizando SS e uma aplicação Java que desenvolvemos ele busca as instruções do Orquestrador Fog com SCP e atualiza a lista de servidores disponíveis no Servidor DNS.

No Sevidor Fog de Domínio foi configurado um servidor DNS que é consultado pelo roteador para resolver os domínios das requisições. Os sistema de contêiner utilizado foi o Docker. Cada aplicação é um contêiner instanciado dentro do Docker Engine. Os servidores de neblina são computadores Raspberry pi 3 com 1,2 GHz de 64-bit quad-core ARMv8, 1 GB de RAM e com sistema operacional Raspbian 10 instalado. O serviço da nuvem utilizado foi da AWS Services. O servidor da nuvem foi uma instância t2.micro com sistema operacional Ubuntu 18.04.4 LTS.

No algoritmo SABANN definimos *Qtd* como 120. Isso representa que utilizaremos os últimos 10 minutos do experimento como base para renovação de limiares. Definimos *Tesa* com 15 segundos, por que é o tempo médio que a nossa aplicação leva para ficar disponível. Já o tempo mínimo de disponibilidade após iniciar *Tmd*, colocamos 30 segundos. Por último, com o objetivo de capturar uma tendência, seja de alta ou de baixa, definimos *Pa* e *Pp* como 12 e 3, respectivamente.

B. *Cenários*

Para testar arquitetura foram propostos alguns cenários baseados em abordagens de trabalhos já existentes na literatura. Foi implementado e utilizado um algoritmo de escalabilidade

para cada cenário, que será executado pelo Orquestrador Fog. Assim, serão adotados quatro cenários e que serão descritos a seguir.

O primeiro cenário é adotar uma abordagem reativa baseada em um recurso computacional. Essa abordagem é uma abordagem clássica e foi adotada por [10] e [16]. O recurso adotado será a vazão *Ra* e chamaremos este cenário de Clássico.

Semelhante ao primeiro, segundo cenário também é adotar uma abordagem reativa, porém com uma combinação de recursos utilizando lógica fuzzy como feito por [11]. Para este cenário utilizamos a vazão *Ra* e processamento *Rb* e chamaremos este cenário de Fuzzy.

O terceiro cenário irá adotar uma abordagem proativa utilizando um modelo de série temporal como utilizado em [12]. Para este algoritmo utilizamos ARMA, uma vez que já foi utilizado também em soluções na nuvem como em [19]. Chamaremos este cenário de ARMA.

O último cenário irá utilizar o algoritmo proposto, que também faz uso de políticas proativas. Logo, chamaremos este cenário de SABANN.

C. Fase 1: Identificação de Recursos

A primeira fase consistiu em identificar quais recursos computacionais eram utilizados pela aplicação de forma que justificasse seu uso em um algoritmo de escalabilidade de aplicações. Dentro do plano de carga de requisições já citado foram observados alguns recursos que tiveram suas medidas coletadas de um servidor de neblina. Dentre os recursos observados os que tiveram significativa variação foram processamento e vazão.

Como visto da Figura 4, existe uma variação do processamento ao longo do plano, porém observamos após dezenas de rodadas que sua média gira em torno de 31% de uso de processamento. Fato esse que confirma que a característica da aplicação não é uso excessivo de processamento.

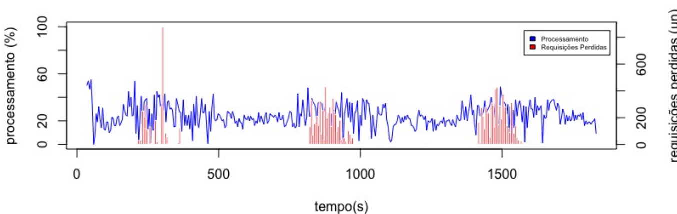

Figura 4. Comportamento do processamento do servidor de neblina durante a execução do plano de carga de requisições em uma rodada.

A Figura 5 mostra que a vazão tem uma leve tendência de alta nos picos de carga, e algumas vezes elas também atingem o pico nesses momentos. Por vezes, observamos que a taxa de vazão obtida pelo Coletor Fog ficava em zero quando começava a ocorrer perda de requisições, sendo um comportamento similar ao travamento do acesso da rede ao servidor de neblina. Também observamos que após dezenas de rodadas sua média ficava por volta de 5,5 Mbits/s.

Depois de vários testes definimos os limiares de escalabilidade para a fase de execução completa. Com relação a

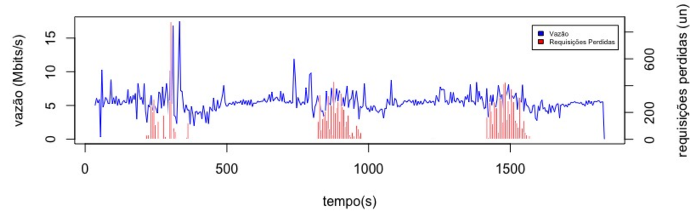

Figura 5. Comportamento da vazão do servidor de neblina durante a execução do plano de carga de requisições em uma rodada.

vazão, para *scale-up* definimos 6,6 Mbits/s e para *scale-down* 4,4 Mbits/s. Especificamente no Cenário Fuzzy, onde também utilizamos processamento, definimos 35% como limiar de *scale-up* e 20% para *scale-down*. Munido dessas informações passamos a segunda fase que foi a execução completa do experimento.

D. Fase 2: Execução Completa

Esta fase se concentra na execução completa do experimento envolvendo os servidores de neblina e o servidor da nuvem. Dentro dos cenários apresentados V-B foram realizadas dez rodadas, a média e o nível de confiança de 95% cada cenário.

A principal métrica de desempenho utilizada para a comparação dos cenários foi a taxa de requisições perdidas diante da carga de requisições geradas.

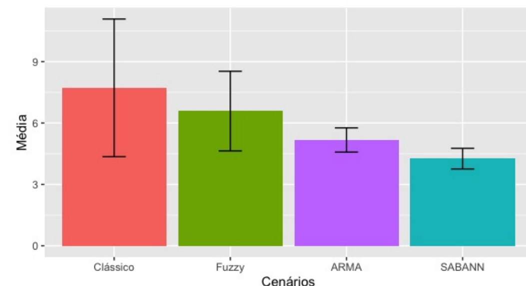

Figura 6. Média da taxa de requisições perdidas no experimento.

A Figura 6 apresenta uma análise a respeito do número médio de requisições perdidas em cada um dos cenários. Podemos observar que o cenário Clássico é o cenário onde existe a maior variação de taxa de requisições perdidas por rodada com um desvio-padrão de 3,36. Durante as rodadas deste cenário, observamos que ações de *scale-up* e *scale-down* foram executadas com tempo muito próximo uma da outra por diversas vezes, fazendo com que o benefício de escalar outra aplicação nem sempre fosse utilizado.

Durante as rodadas do cenário Fuzzy observamos que o fato de termos a combinação de duas variáveis como ponto a ser alcançado para a tomada de decisão de qual ação de escalabilidade tomar, algumas vezes fez com que a quantidade de requisições perdidas aumenta-se por não terem alcançado seu ponto de *scale-up*. Em outros momentos, ficou-se mais tempo que o necessário com mais de uma instância da aplicação ativa sem necessidade por não alcançarem o ponto de *scale-down*.

No cenário ARMA, no nosso experimento observamos que, com a carga de requisições projetada para que pudéssemos

ter ações tanto de *scale-up* quanto de *scale-down*, os pontos futuros para tomada de decisão giravam em torno da média geral da rodada. Observamos também que por adotar uma política proativa, este cenário por vezes conseguiu prever o momento de escalar uma nova instância da aplicação, fazendo com que a taxa de requisições perdidas obtida ao longo de 10 rodadas fossem melhor que os cenários Clássico e Fuzzy.

A característica de adaptabilidade dos limiares por vezes fez com que os pontos de escalabilidade do cenário SABANN sofressem alterações. Isso fez com que as ações de *scale-up* e *scale-down* acompanhassem a taxa de vazão que era utilizada nos servidores de neblina. O algoritmo SABANN também possui uma abordagem proativa, tentando prever o próximo ponto de escalabilidade da plataforma. Observamos que a combinação da abordagem proativa com os limiares adaptativos trouxe uma maior precisão do momento de escalar novas instâncias permitindo que a plataforma se antecipasse ao ponto de saturamento do servidor onde há perdas de requisições. Assim, o algoritmo SABANN se mostrou melhor que o cenário ARMA apresentando uma taxa média de requisições perdidas de 4,25% contra 5,17% do ARMA.

Comprovando a hipótese H1 vimos que os cenários ARMA e SABANN, que foram utilizados uma política proativa, tiveram médias de 5,17% e 4,25% respectivamente, enquanto os cenários Clássico e Fuzzy, que utilizaram uma política reativa tiveram médias de 7,72% e 6,58% respectivamente. Ainda assim, validando a hipótese H2, os cenários ARMA e SABANN tiveram uma variabilidade bem pequena, ambos com desvio-padrão abaixo de 0,6 enquanto Clássico e Fuzzy os desvio são acima de 1,9. Por fim, verificamos que o algoritmo SABANN teve o melhor resultado na proposta de diminuir a quantidade de requisições perdidas oriundas da camada terminal, satisfazendo a hipótese H3.

VI. Conclusão

Este artigo apresentou uma proposta de arquitetura e algoritmo de escalabilidade de aplicações na Computação em Neblina para aplicações ITS que possuem necessidade de respostas rápidas devido à importância do tempo da informação. A abordagem proposta combina conceito de Computação em Neblina e técnicas de escalabilidade de aplicações para criar sistemas que mitiguem a quantidade de requisições perdidas oriundas da camada terminal. Demonstramos através de uma experimentação real quatro cenários de escalabilidade utilizando a mesma aplicação ITS, três delas com métodos e políticas de trabalhos já existentes e a quarta baseada em nossa proposta. Propusemos uma solução diferenciada baseada em ARNN e uma técnica de limiares adaptativos de escalabilidade. Através dos experimentos realizados, verificamos uma redução na quantidade de requisições perdidas com o uso do algoritmo proposto em relação aos outros algoritmos comparados, evidenciando a importância do presente trabalho para a área de Computação em Neblina com suporte as aplicações ITS.

Como trabalhos futuros, consideramos avaliar mais cenários e tipos de algoritmos baseados em aprendizado de máquina buscando encontrar soluções que diminuam ainda mais a quantidade de requisições perdidas.

Referências

[1] "População mundial deve chegar a 9,7 bilhões de pessoas em 2050, diz relatório da onu," https://nacoesunidas.org/populacao-mundial-deve-chegar-a-97-bilhoes-de-pessoas-em-2050-diz-relatorio-da-onu/amp/, accessed: 2020-07-13.

[2] A. M. Nagy and V. Simon, "Survey on traffic prediction in smart cities," *Pervasive and Mobile Computing*, vol. 50, pp. 148–163, 2018.

[3] M. Chaqfeh, N. Mohamed, I. Jawhar, and J. Wu, "Vehicular cloud data collection for intelligent transportation systems," in *2016 3rd Smart Cloud Networks & Systems (SCNS)*. IEEE, 2016, pp. 1–6.

[4] Z. Jiang, S. Zhou, X. Guo, and Z. Niu, "Task replication for deadline-constrained vehicular cloud computing: Optimal policy, performance analysis, and implications on road traffic," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 93–107, 2017.

[5] R. I. Meneguette, "A vehicular cloud-based framework for the intelligent transport management of big cities," *International Journal of Distributed Sensor Networks*, vol. 12, no. 5, p. 8198597, 2016.

[6] P. Cruz, F. F. da Silva, R. G. Pacheco, R. D. S. Couto, P. B. Velloso, M. E. M. Campista, L. H. M. K. Costa *et al.*, "Sensingbus: Using bus lines and fog computing for smart sensing the city." *IEEE Cloud Comput.*, vol. 5, no. 5, pp. 58–69, 2018.

[7] W. Zhang, Z. Zhang, and H.-C. Chao, "Cooperative fog computing for dealing with big data in the internet of vehicles: Architecture and hierarchical resource management," *IEEE Communications Magazine*, vol. 55, no. 12, pp. 60–67, 2017.

[8] Q. T. Minh, C. M. Tran, T. A. Le, B. T. Nguyen, T. M. Tran, and R. K. Balan, "Fogfly: A traffic light optimization solution based on fog computing," in *Proceedings of the 2018 ACM International Joint Conference and 2018 International Symposium on Pervasive and Ubiquitous Computing and Wearable Computers*, 2018, pp. 1130–1139.

[9] C. Mouradian, D. Naboulsi, S. Yangui, R. H. Glitho, M. J. Morrow, and P. A. Polakos, "A comprehensive survey on fog computing: State-of-the-art and research challenges," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 1, pp. 416–464, 2017.

[10] C.-L. Tseng and F. J. Lin, "Extending scalability of iot/m2m platforms with fog computing," in *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*. IEEE, 2018, pp. 825–830.

[11] F.-H. Tseng, M.-S. Tsai, C.-W. Tseng, Y.-T. Yang, C.-C. Liu, and L.-D. Chou, "A lightweight autoscaling mechanism for fog computing in industrial applications," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 10, pp. 4529–4537, 2018.

[12] A. Zanni, S. Forsstrom, U. Jennehag, and P. Bellavista, "Elastic provisioning of internet of things services using fog computing: An experience report," in *2018 6th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*. IEEE, 2018, pp. 17–22.

[13] M. Aazam, S. Zeadally, and K. A. Harras, "Fog computing architecture, evaluation, and future research directions," *IEEE Communications Magazine*, vol. 56, no. 5, pp. 46–52, 2018.

[14] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Elasticity in cloud computing: state of the art and research challenges," *IEEE Transactions on Services Computing*, vol. 11, no. 2, pp. 430–447, 2017.

[15] C. C. Byers, "Architectural imperatives for fog computing: Use cases, requirements, and architectural techniques for fog-enabled iot networks," *IEEE Communications Magazine*, vol. 55, no. 8, pp. 14–20, 2017.

[16] S. Taherizadeh, V. Stankovski, and M. Grobelnik, "A capillary computing architecture for dynamic internet of things: Orchestration of microservices from edge devices to fog and cloud providers," *Sensors*, vol. 18, no. 9, p. 2938, 2018.

[17] A. Tealab, "Time series forecasting using artificial neural networks methodologies: A systematic review," *Future Computing and Informatics Journal*, vol. 3, no. 2, pp. 334–340, 2018.

[18] A. Souza, N. Cacho, A. Noor, P. P. Jayaraman, A. Romanovsky, and R. Ranjan, "Osmotic monitoring of microservices between the edge and cloud," in *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2018, pp. 758–765.

[19] K. Li, "Quantitative modeling and analytical calculation of elasticity in cloud computing," *IEEE Transactions on Cloud Computing*, 2017.