Universidade Federal de Pernambuco
Centro de Informática

Graduação em Ciências da Computação

Análise de Performance de Operadores no CEPSwift

Hélmiton Moraes da Silva Cunha Júnior

Trabalho de Graduação

Recife
Junho de 2018

Universidade Federal de Pernambuco
Centro de Informática

Hélmiton Moraes da Silva Cunha Júnior

Análise de Performance de Operadores no CEPSwift

Trabalho apresentado ao Programa de Graduação em Ciências da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: *Kiev Santos da Gama*

Recife
Junho de 2018

À todo aquele que não desistiu.

Agradecimentos

Agradeço a todos os autores que nunca conhecerei, mas que transformaram a mim através de suas aulas e seus livros. Agradeço a todos os cientistas e pesquisadores que vieram antes de mim e me forneceram ferramentas incríveis e expandiram a minha visão de mundo através do seu árduo trabalho. Agradeço porém, acima de tudo, a toda pessoa que entende que compartilhar o conhecimento é uma obrigação que todos temos, não só com nós mesmos e aqueles ao nosso redor, mas para a humanidade como um todo.

Resumo

Este trabalho de graduação toma como referência estudos sobre a simetria e relação entre Processamentos de Eventos Complexos e programação Reativa, acerca das vantagens do uso de álgebra relacional numa aliança entre as duas tecnologias. As referências mais influentes deste trabalho foram estudos realizados pelo Centro de Informática da Universidade Federal de Pernambuco, e o mesmo toma como premissa o argumento por eles proposto: dado que Processamento de Eventos Complexos e Programação Reativa compartilham os mesmos princípios de fluxo de dados, uma vez que ambas são orientadas a eventos, é interessante explorar as vantagens de combinar as duas tecnologias e realizar o processamento de eventos complexos de forma reativa. O objetivo deste relatório é implementar, quantificar e documentar as vantagens obtidas ao usar programação reativa no processamento de eventos complexos, especificamente através da análise de performance de diferentes implementações de operadores de agregação de eventos complexos. Isto também inclui a própria implementação destes operadores, que serão de natureza estatística. O artefato do estudo é a biblioteca open-source CEPSwift, fruto de um dos trabalhos de referência para esta tese.

Palavras-chave: Programação Orientada a Eventos, Processamento de Eventos Complexos, Programação Reativa

Sumário

CAPÍTULO 1

Introdução

Para entender o contexto deste trabalho, é necessário conhecer primeiro alguns conceitos básicos sobre arquitetura de eventos, começando pela definição de evento. Um evento é definido por David Luckham [Luc08], como uma atividade de interesse registrada e propagada em um sistema, seja qual for o contexto. Ao processar eventos de um sistema para tomar decisões, encontramos o conceito de Programação Orientada a Eventos (Event Driven Architecture - EDA), um tipo especial de Programação Orientada a Serviços (Service-Oriented Architecture - SOA). A EDA é embasada em processamento de eventos, dispondo de serviços que são invocados a partir de gatilhos programados que podem ser acionados por tais eventos [MM06]. Um bom exemplo de EDA são os event-driven communication models, frequentemente usados no contexto de IoT, como demonstrado em outro trabalho produzido pelo Centro de Informática Universidade Federal de Pernambuco [SGV+04].

Processamento de Eventos Complexos (Complex Event Processing - CEP) é uma camada construída sobre a EDA, responsável pelo processamento de múltiplas fontes de eventos para identificar padrões e acionar serviços de acordo [RW10]. O CEP também é capaz de fazer uma análise temporal de do conjunto de eventos do sistema, o que lhe permite averiguar o comportamento dos eventos de ao longo do tempo, além de acumular, agregar e compor tais eventos. A combinação desse monitoramento com a manipulação e propagação de eventos torna o CEP fortemente flexível e escalável, capaz de se ajustar à diferentes condições e contextos dinâmicos de uma aplicação. CEP tem se popularizado com aplicações escaláveis e sistemas distribuídos, principalmente no contexto de IoT devido à facilidade de integração com padrões de distribuição orientada à mensagem, comuns em redes de dispositivos embarcados [CYH+17] [QSF+16].

Analogamente, outra tecnologia que toma proveito das vantagens da orientação à eventos é a Programação Reativa (Reactive Language - RL). Assim como CEP, RL é outra camada construída sobre a EDA. Também de maneira similar ao processamento de eventos complexos, uma aplicação reativa responde a mudanças no estado de componentes de interesse, seguindo o mesmo padrão de processamento e propagação de mudanças.

Devido às semelhanças estruturais de orientação à eventos [MS13], a sinergia entre RL e CEP pode apresentar vantagens significativas ainda não exploradas, uma vez que a integração dos dois conceitos ainda é um problema em aberto [BG17]. como detectar padrões de eventos complexos, e propagar suas mudanças de forma reativa a fim de melhorar o tempo de resposta do processamento.

1.1 Motivação

Este trabalho de graduação toma como ponto de partida a tese de graduação de George Belo Guedes [BG17] acerca da implementação baseada em programação reativa do framework CEPSwift, o primeiro e único framework CEP atualmente disponível de forma aberta para a linguagem Swift. Propõe-se que o leque de operadores disponível na ferramenta seja aumentado, além de fazer uma análise comparativa da performance de diferentes implementações destes novos operadores de eventos complexos, baseando-se em métricas referenciadas na academia [LB14] a fim de validar a premissa de melhoria através do uso de programação reativa.

1.2 Objetivos

1.2.1 Implementar Operadores

Este trabalho se propõe a implementar e integrar novos operadores estatísticos de agregação ao repertório de operadores CEP do framework CEPSwift. Os novos operadores a serem implementados são média, esperança, e variância, consistindo em uma contribuição real à ferramenta de código aberto.

1.2.2 Demonstrar Caso de Uso

Este trabalho se propõe a demonstrar um caso de uso dos operadores implementados, de forma a validar a corretude quanto a completude dos novos operadores desenvolvidos para o CEPSwift. Este caso de uso será produzido como uma aplicação teste para iOS.

1.2.3 Analisar Performance

Este trabalho também se propõe a fazer uma análise prática da performance dos operadores listados acima para diferentes implementações da mesma ferramenta. Esta análise tomará o benchmark CEPBen [LB14] e o livro Art of Computer Systems Performance Analysis Techniques For Experimental Design Measurements Simulation And Modeling [Jai91] como referências, e deve servir como argumento a cerca da melhora ou piora de performance, além de servir como teste de conceito para as otimizações inferidas pelo uso de programação reativa. Devem ser comparadas implementações reativas nativas e iterativas dos mesmos operadores.

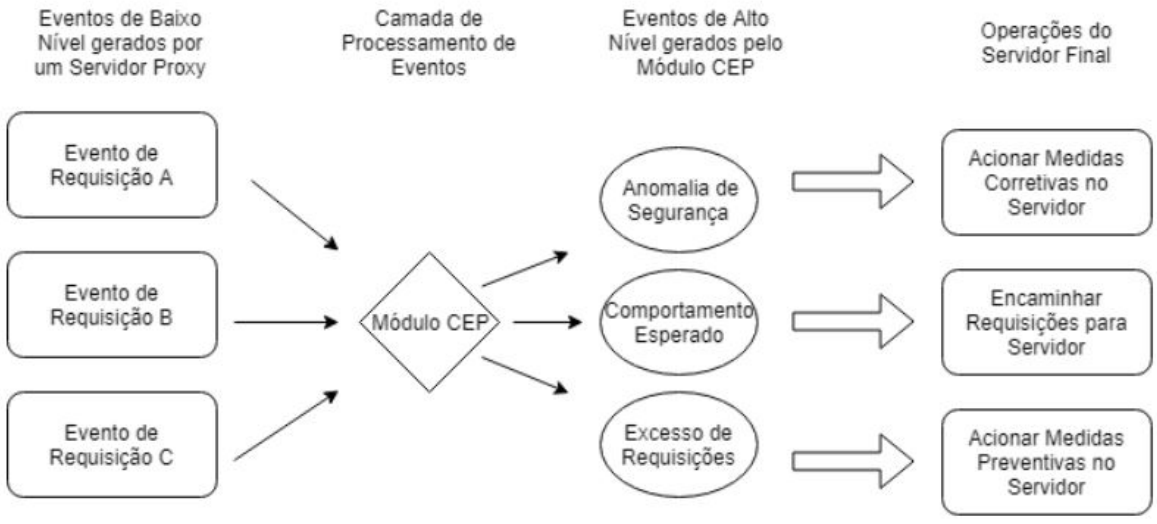
CAPÍTULO 2

Contexto

Esta tese de graduação toma como principal referência e ponto de partida o trabalho de graduação de George Belo Guedes [BG17] acerca da implementação do CEPSwift, framework de processamento de eventos complexos para a linguagem Swift, usufruindo das vantagens da programação reativa. Enquanto o trabalho de graduação referenciado contém argumentos mais aprofundados quanto às vantagens proporcionadas pelo uso da álgebra relacional no CEP, este trabalho atual foca na implementação e validação de novos componentes da ferramenta e do caso de uso para testes.

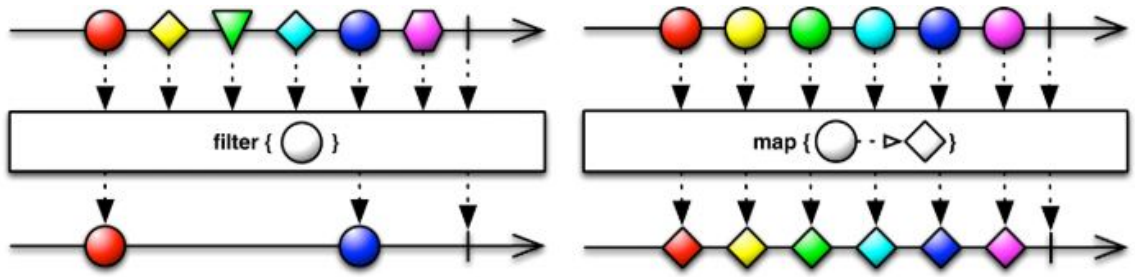
2.1 Processamento de Eventos Complexos

De acordo com [RW10], CEP pode ser visto como um serviço que recebe e identifica eventos de baixo nível e gera, através da agregação destes, eventos de alto nível. Eventos de baixo nível são eventos simples e unitários, como o timeout de uma operação de rede ou uma requisição à um servidor. Eventos de alto nível são gerados a partir da manipulação, combinação e/ou agregação de eventos simples e enviados para outro serviço EDA para que as reações adequadas sejam realizadas.



2.2 Operadores CEP

Para gerar estes eventos complexos, um sistema CEP segue regras bem definidas de manipulação, filtragem e agregação eventos simples para identificar padrões de interesse. Os operadores de um sistema CEP são os responsáveis por aplicar tais operações e verificações sobre fontes de eventos (event streams). É através dos operadores que os sistemas CEP combinam, filtram e manipulam eventos resultantes de streams para tentar identificar as condições de gatilho das regras do sistema. A partir daqui é fácil de identificar semelhanças conceituais entre CEP e RL, uma vez que filtragem, mapeamento e agregação de dados já são métodos de processamento e propagação de mudanças bem difundidos na programação reativa.



O uso da abstração de modelagem dados de frequente variação em série temporal fornecida pelo paradigma reativo para o processamento e propagação de eventos de alto nível é o ponto central da relação RL e CEP observada nesse estudo. Uma excelente referência ao tópico é o trabalho de Magara e Salvaneschi [MS13] que faz um estudo mais aprofundado sobre essa relação.

2.3 Classificação de Operadores

De acordo com a literatura [Cor06], existem 10 padrões de design fundamentais para sistemas CEP, que aparecem repetidamente em suas aplicações, desde à mais simples à mais sofisiticada. Estes seriam como blocos de construção, usados para construir uma aplicação CEP, e são usados para classificar os operadores CEP de acordo com sua natureza, como visto na tabela abaixo.

Design Pattern	Descrição	Aplicações
Filtering	Filtra o stream a partir de uma condição lógica	Capturar leituras fora do escopo esperado de uma rede de sensores
In-memory caching	Guarda múltiplos eventos na memória	Guardar cliques e buscas feitas por um usuário de um sistema web
Aggregation over windows	Computa medidas estatísticas a partir de eventos salvos na memória	Computar o mínimo e máximo uso de CPU e memória de uma máquina em um dado intervalo de tempo
Database lookups	Adquire o histórico ou referência contextual de eventos vindouros de uma base de dados relacional	Verificar o histórico de alertas de um IP suspeito em uma aplicação de segurança de rede
Database Writes	Envia eventos para uma base de dados relacional	Guardar relatórios sobre nós específicos de uma rede
Correlation (Joins)	Une múltiplos event streams em um único event stream	Correlacionar e gerenciar múltiplos sistemas num processo de negócios
Event pattern matching	Identifica padrões de eventos temporais através de múltiplos event streams	Detectar padrões de fraude em serviços financeiros
State machines	Modela comportamentos e processos complexos através de uma máquina de estados	Acompanhar o usuário (série de transações entre estados) em uma página web
Hierarchical Evemts	Processa, analisa e compõe eventos hierárquicos	Gerar um evento com estrutura XML
Dynamic Queries	Submete, requisita e subscreve dinamicamente à queries parametrizadas	Ajustar dinamicamente um servidor através de técnicas de machine learning aplicadas ao ambiente CEP

2.4 Plataforma: CEPSwift

O CEPSwift é um framework de código aberto da linguagem Swift, resultante de um trabalho de pesquisa do Centro de Informática da UFPE. Atualmente ele é o único framework de processamento de eventos complexos de código aberto disponível para a linguagem, gerando a motivação inicial para o estudo. A ferramenta se propõe a facilitar a construção de apps que precisam gerar, manipular ou processar múltiplos streams de dados. O framework CEPSwift dispõe de uma série de operadores CEP que permitem realizar estas tarefas, porém o número destes operadores ainda é limitado, criando uma necessidade de adicionar novas opções de operadores. Outra contribuição ao framework é a criação de uma ferramenta de testes para avaliação de operadores, também desenvolvida por este trabalho, usada para coletar os dados de testes do mesmo.

2.5 Cenário: Internet of Things e Big Data

Um dos usos populares de CEP se encontra na emergente área de Internet das Coisas (Internet of Things - IoT), como bem descrito no trabalho When Things Matter, "*The usage of background knowledge about events and their relations to other concepts in the application domain can improve the expressiveness and flexibility of CEP systems. Huge amounts of domain background knowledge stored in external knowledge bases can be used in combination with event processing in order to achieve more knowledgeable complex event processing*" [QSF+16, p. 147]. Esta citação determina que o conhecimento contextual sobre os eventos e suas relações com outros conceitos da aplicação pode melhorar a expressividade e responsividade de sistemas CEP. A atual popularidade de Internet of Things, tanto no estado da arte como no estado da prática advoga a favor dessa ideia e motiva a adição de operadores CEP relacionados à este contexto ao repertório do CEPSwift.

Muitas pesquisas já combinaram os conceitos de CEP, IoT e fundamentos de Big Data para realizar as mais diferentes tarefas, dada a quantidade de dados em série temporal gerada pelos sistemas que aplicam esta integração. Desde predição de eventos complexos para aplicações de IoT proativas [ACMZ15] à produção de frameworks de Big Data Analytics para cidades inteligentes [SZGA15], há vários exemplos demonstram o grande valor em analisar o montante de dados gerados por estas aplicações, fornecendo a motivação para o escopo dos novos operadores. Partimos então para a implementação de operadores estatísticos, que possam ser usados como base para extrair dados pertinentes sobre o comportamento de eventos ao longo do tempo. Os operadores selecionados foram escolhidos pois representam computações estatísticas básicas, necessárias para a construção de análises estatísticas mais complexas, como a regressão linear. Sendo assim, a implementação destes também visa pavimentar o caminho para futuras contribuições para o framework CEPSwift à longo prazo, já que são os seus primeiros operadores estatísticos.

2.6 Validação: Performance

Além da implementação de operadores estatísticos, este trabalho também concerne a validação da premissa dos trabalhos anteriores: analisar a diferença de performance entre o processamento de eventos complexos usando programação reativa de suas outras possíveis implementações. Para tanto, tomamos como referência e inspiração duas obras de referência: o CEPBen, modelo de benchmark desenhado para avaliar a performance dos comportamentos funcionais de um sistemas CEP [LB14].

É importante ter em mente que o processo avaliação de um sistema depende muito do contexto do próprio sistema e dos fatores/hipóteses a serem observadas, como muito bem descrito em [Jai91], "successful evaluation is a art that cannot be produced mechanically". Mas ainda assim há regras e boas práticas que podemos seguir para realizar uma avaliação de performance confiável, significativa e sem viés. Precisamos antes de um conhecimento básico sobre terminologias e técnicas de avaliação de performance. Para tal fim usaremos os conceitos apresentados no em [Jai91] para definir a técnica de avaliação, métricas, parâmetros, e análises mais adequadas para nosso estudo.

2.3.1 Objetivo da Avaliação

A definição do objetivo de uma avaliação de performance deve ser o ponto inicial, de onde partem todas as outras variáveis e técnicas a serem selecionadas. Também será necessário escolher a técnica de avaliação apropriada, de modo que o resultado da avaliação não seja enviesado pelo "jogo da razão" [Jai91], técnica de análise de dados onde os dados de um mesmo estudo podem advogar contra ou à favor de um sistema simplesmente mudando a razão matemática pela qual a métrica do teste é avaliada. Definimos então, com clareza, que o objetivo do teste de performance a ser realizado é quantificar a diferença de performance entre processamentos de eventos complexos com e sem o uso de programação reativa. Deste modo, não queremos provar que o uso da linguagem reativa é melhor para a performance ou realizar um benchmark minucioso, mas sim aferir se seu uso confere qualquer diferença à performance, e se o fizer, quantificar e avaliar essa mesma diferença.

Medir a performance de um sistema computacional requer ao menos duas ferramentas — uma para carregar o sistema com trabalho (load generator) e outra ferramenta para medir, de alguma forma, os trabalhos realizados [Jai91]. Neste trabalho, as duas ferramentas foram implementadas, de forma que possam ser usadas como ferramentas de testes superficiais de performance do framework.

2.3.2 Load Generator

A ferramenta load generator é responsável por gerar trabalho para o sistema através da produção de cargas de trabalho (workloads), que são essencialmente o input necessário para que o sistema realize as atividades a serem analisadas. Por exemplo, o workload de uma CPU poderia ser a lista de instruções a serem executadas por ela. Enquanto isso, o monitor

2.3.2 Monitor

A ferramenta monitor é é responsável pelo registro das informações de interesse do sistema e é através dele que as medições são realizadas. Este registro ocorre durante o processamento do workload proveniente do load generator. No mesmo exemplo anterior, o monitor teria a tarefa de medir o tempo de resposta da CPU à lista de instruções executada.

2.3.3 Métricas

Há três métricas fundamentais relacionadas à análise de performance na razão tempo por recurso, sendo elas a responsividade, produtividade e utilização. Para clarificar os termos, usaremos o exemplo de um gateway de rede como ilustração. A responsividade de um gateway de rede pode ser medida pelo seu tempo de resposta, intervalo de tempo entre a chegada de um pacote na rede e sua entrega com êxito. A produtividade de um gateway pode ser medida através do throughput, número de pacotes de rede encaminhados por determinada razão de tempo. A utilização de um gateway de rede pode ser medida através da porcentagem de tempo onde os recursos do gateway estão ocupados para dada carga de trabalho.

Em nosso estudo, selecionamos responsividade e produtividade como as métricas de avaliação de operadores estatísticos no CEPSwift:

- **Responsividade**

  Calculada de duas formas: Turnaround Time, relação entre o tempo de entrada de N eventos em um sistema CEP e o tempo de detecção de M eventos complexos esperados. Response Time, relação entre o tempo de requisição e resultado de uma operação. No

nosso caso, o response time será usado para melhorar a precisão de comparação entre diferentes configurações de testes [Jai91]. Tanto para o turnaround time quanto para o response time, quanto menor o valor medido, melhor o resultado do teste.

- **Produtividade**

  Calculada pelo Throughput do sistema, ou a razão entre os N eventos processados por um sistema CEP (eventos consumidos e produzidos) e uma determinada unidade de tempo [LB14]. Em nosso estudo adotaremos a notação de transações por segundo (tps) para esta métrica, onde quanto maior o valor, melhor o resultado do teste.

2.3.4 Metodologia

As métricas acima serão computadas na mesma plataforma, através de simulações parametrizadas para cada variação de implementação de cada operador estatístico proposto neste trabalho. Isto será feito de forma a permitir uma avaliação baseada em testes sobre a performance de um mesmo operador quando implementado de formas diferentes. Analisaremos, finalmente, os dados gerados por um número de simulações para que possamos argumentar contra ou a favor do uso de programação reativa no processamento de eventos complexos.

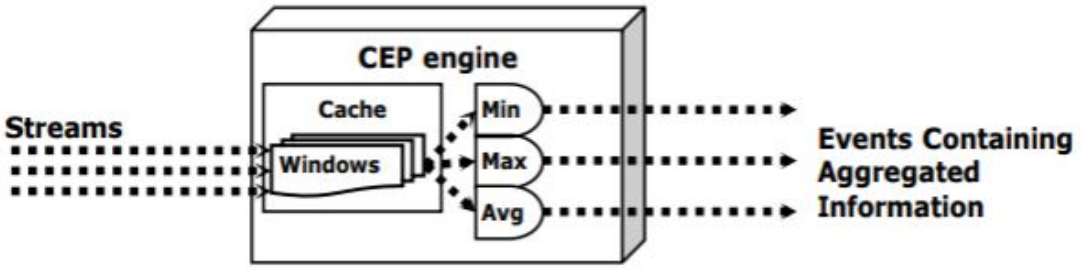
Usaremos os operadores de média e esperança no processamento de eventos gerados por uma simulação de rede de sensores. Esta simulação gera um workload grande o suficiente para nos fornecer dados de performance significativos, necessário para analisar a fim de gerar eventos complexos de natureza corretiva para a rede simulada a partir de eventos de leitura de temperatura e de desconexão (timeout) de sensores da rede. Também faremos uma análise da relação entre alguns parâmetros da rede simulada (como quantidade de sensores na rede e a quantidade de eventos de timeout, por exemplo) para identificar como essas mudanças afetam nossas simulações.

CAPÍTULO 3

Implementação

3.1 Operadores Estatísticos

Para nossa implementação, foram escolhidos operadores estatísticos básicos que, quando combinados fornecem a estrutura de uma análise estatística mais complexa: a regressão linear. Dado que todos estes operadores são de análise estatística, todos eles se classificam como operadores de Aggregation over Windows.



Outros operadores menores também foram desenvolvidos no decorrer do trabalho, como probability, sum e toArray, mas estes não serão avaliados individualmente e sim dentro do contexto dos operadores selecionados. São estes:

- Média

$$\bar{x} = \frac{x_1 + x_2 + \ldots + x_n}{n} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Operador de agregação que realiza a média aritmética de uma variável dentro de um conjunto de eventos de interesse. Este operador é talvez o mais importante dentre os operadores de cunho estatístico uma vez que serve como base para inúmeros outros operadores de análise composta.

Implementação Reativa

```
extension EventStream where T: NumericEvent {
    public func sum() -> Observable<Int> {
        return self.observable
            .map({$0.numericValue})
            .scan(0) { (lastSlice, newValue) in
                return lastSlice + newValue
            }
    }

    public func count() -> Observable<Int> {
        return self.observable
            .map({$0.numericValue})
            .scan(0) { (lastSlice, _) in
                return lastSlice + 1
            }
    }

    public func average(timeWindow: Double, currentDate: Date) -> Observable<Int> {
        let doubled = Observable
            .combineLatest(self
                .filter(predicate:
                    {currentDate.timeIntervalSince($0.timestamp) < timeWindow})
                .sum(),
                self.filter(predicate:
                    {currentDate.timeIntervalSince($0.timestamp) < timeWindow})
                    .count())
            {return $0/$1}
        return doubled.skip(1)
    }
```

Implementação Iterativa

```
let keepaliveUpdateRule2 = timeoutEvents.stream.toArray()
keepaliveUpdateRule2.subscribe(onNext: {
    var sum = 0
    for value in $0 {
        sum += value.numericValue
    }
    let it_average = sum/$0.count
    if (it_average > minimalTimeout) {
        self.kpaUpdate(timeout: it_average)
    }
}).disposed(by: disposeBag)
```

- Esperança

$$E[X] = \sum_{i=1}^{\infty} x_i p(x_i)$$

Operador de agregação que computa o valor esperado de determinada variável à partir de

um conjunto de eventos de interesse. Seu valor é baseado na probabilidade de ocorrência de determinado valor em razão do total de ocorrências.

Implementação Reativa

```
public func expected(val: Int, trials: Int) -> Observable<Double> {
    let probability = self.probability(val: val, trials: trials)
    let doubled = Observable
        .combineLatest(probability, self.count()) { return $0*Double($1)}
    return doubled.skip(1)
}
```

Implementação Iterativa

```
let tempUpdateRule2 = tempEvents.stream.toArray()
tempUpdateRule2.subscribe(onNext: {
    var matches = 0
    for event in $0 {
        if (event.numericValue == toExpect) {
            matches += 1
        }
    }
    let it_expectedValue = ((matches/$0.count) * (noIterations-$0.count))
    if (it_expectedValue > maxTemperature) {
        self.tempUpdate(temp: it_expectedValue)
    }
}).disposed(by: disposeBag)
```

- Variância

$$\mathrm{var}(X) = \mathrm{E}((X - \mu)^2)$$

Também conhecida como Processo Estocástico, variância é um operador que mede a dispersão estatística de uma variável, exprimindo o quão distante os valores dessa variável estão em relação ao seu valor esperado.

Implementação Reativa

```
//variance = prob(x)*trials*1-prob(x)
public func variance(dataSize: Int) -> Observable<Double> {
    // Dataset mean
    let a = self.sum().map {$0/dataSize}
    // Square diferences
    let b = Observable.combineLatest( a, self.observable
        .map({$0.numericValue})) {return ($0 - $1)*($0 - $1)}
    // Averaged squared diferences
    let sumAll = b.scan(0) {(a,b) in return a + b}
    let countAll = b.scan(0) {(a,_) in return a + 1}
    let doubled = Observable.combineLatest(
        sumAll.map({Double($0)}),
        countAll.map({Double($0)})) {($0/$1)}
    return doubled.skip(3)
}
```

Implementação Iterativa

```
let marginUpdateRule2 = tempEvents.stream.toArray()
marginUpdateRule2.subscribe(onNext: {
    var dataset = Array<Int>()
    var mean:Double = 0
    for event in $0 {
        mean += Double(event.numericValue)
        dataset.append(event.numericValue)
    }
    mean = mean/Double($0.count)
    let sumAll:Double = Double(dataset.reduce(0, +))
    let variance:Double = sumAll/Double(dataset.count)
    if (variance > 1000) {
        self.marginUpdate(value: variance)
    }
}).disposed(by: disposeBag)
```

3.2 Load Generator

Para aproximar ainda mais a proposta da tese e sua motivação, foi desenvolvido um gerador de carga para o CEPSwift que simula o comportamento de uma rede de sensores. Esta simulação de rede emite uma variação de eventos de timeout de sensores e de leituras de ambiente, que por sua vez são processados para gerar outros eventos complexos.

3.2.1 Parametrização e Aleatoriedade

A geração de workload de nossa simulação leva em consideração os seguintes parâmetros da rede simulada: Número de Sensores na rede, fator que dita a quantidade de geradores de eventos

de leitura de temperatura e de timeout, nosso workload. Em nossa simulação o número de sensores é o fator multiplicador de geração de carga. Uma rede com o dobro de sensores realiza o dobro de leituras de temperatura e, analogamente, espera-se também o dobro de eventos de timeout.

Qualidade de Serviço da rede, fator que dita a probabilidade da rede gerar um evento de timeout de conexão com um censor. Para cada sensor na rede um evento de timeout pode ser levantado a partir de um intervalo de tempo mínimo da entrada do sensor na rede ou de sua última desconexão por timeout. A probabilidade do timeout ocorrer aumenta quanto menor for a qualidade de serviço da rede.

Frequência de Leitura da rede, fator que dita a frequência de eventos de leitura de temperatura gerados por um censor. É a principal constante de geração de workload da simulação, capturando valores aproximados da temperatura simulada. Para fins de teste, é esperado que as leituras de temperatura tenham uma margem de erro de 3 graus celsius em relação à temperatura simulada.

Número de Iterações, fator que representa o número esperado de leitura de temperatura realizado pelos sensores na simulação. Cada leitura realizada entre intervalos de tempo determinados pelo parâmetro de frequência de leitura. O tempo total de simulação pode ser calculado, aproximadamente, pelo número de iterações de leitura multiplicado pela frequência de leitura dos sensores.

Tempo de keepalive dos Sensores, fator que dita o tempo de tolerância permitido aos sensores da rede antes que a mesma gere um evento de timeout. Corresponde ao tempo mínimo necessário para que um censor gere um evento de timeout, análogo ao tempo de frequência de leitura dos sensores. Este valor deve ser manipulado através do processamento de eventos de timeout.

Temperatura e Margem, fatores que ditam, respectivamente, a temperatura inicial da sala dos sensores e a margem de erro de leitura de temperatura esperada de cada sensor. No decorrer de nossa simulação a temperatura deve ser manipulada através do processamento de eventos de leitura dos sensores.

Produção de Eventos de Timeout

```
// Avoid unwanted synchronized behaviour
Thread.sleep(forTimeInterval: TimeInterval(n+1))
for _ in 1...self.noIterations {
    var newTemp: Int
    // Wait for frequency interval
    Thread.sleep(forTimeInterval: TimeInterval(self.status.readFreq))
    // Randomize sensor reading
    let newMargin = Int(arc4random_uniform(UInt32(self.tempMargin)));
    // Calculate new temperature
    if (self.tempMargin < self.temperature) {
        newTemp = self.temperature - self.tempMargin + newMargin
    } else {
        newTemp = self.tempMargin - self.temperature + newMargin
    }
    // Add new event to given sensor
    let date = Date()
    self.queueTemp.async {
        self.sensorReadEvents
            .addEvent(event: SensorReadEvent(date: Date(), data: newTemp))
    }
    self.queueMonitor.sync {
        self.monitor.append(ticket:
            monitorTicket(type:"TRE",data:newTemp,date:date, id:n))
    }
}
// Count sensor as finished
self.queueMonitor.sync {
    self.sensorsDone += 1
}
if (self.sensorsDone >= self.status.noSensors) {
    //self.getTickets()
    self.simulationStatus.addEvent(event: TimeoutEvent(date: Date(), data: 0))
}
```

Produção de Eventos de Temperatura

```
// Avoid unwanted synchronized behaviour
Thread.sleep(forTimeInterval: TimeInterval(n))
while (self.sensorsDone < self.status.noSensors) {
    // Wait for keepalive timeout interval
    Thread.sleep(forTimeInterval: TimeInterval(self.status.kpaTimeout))
    // Simulate if the sensor responded keepalive or failed
    let timeout = self.status.qos - Int(arc4random_uniform(UInt32(100)))
    let date = Date()
    // If failed, add new timeout event
    if (timeout < 0) {
        self.queueTimeout.async {
            self.timeoutEvents.addEvent(
                event: TimeoutEvent(date: date, data: (timeout * -1)))
        }
        self.queueMonitor.sync {
            self.monitor.append(ticket:
                monitorTicket(type:"KTE",data:(timeout * -1),date:date, id: n))
        }
    }
```

Assim como comumente é visto em uma rede de sensores real, os sensores não possuem qualquer sincronização entre si, e as threads de geração de evento vão, eventualmente, colidir na entrega do evento produzido. Este é um comportamento esperado e também desejado, já que nos poupa o trabalho de implementar um mecanismo de perda de pacotes na rede. De-

sejamos assumir um comportamento simulado próximo do esperado de uma rede real, já que nosso objetivo não é fazer um benchmark preciso de performance, mas sim fazer um quadro comparativo geral das diferenças de performance entre os paradigmas implementados. Nossa simulação usa um valor aleatório para decidir pela ocorrência ou não de um evento de time-out. A seleção e geração desse valor aleatório é feita em conformidade com as propriedades desejadas de um bom gerador de números aleatórios, como visto em [Jai91], incluindo:

1. Computável de forma eficiente, a se evitar overhead durante a simulação.

2. Escopo de valor largo, de forma a evitar a reciclagem de valores anteriores o que poderia gerar um enviesamento.

3. Valores consecutivos gerados de forma independente e uniforme, de modo a haver o mínimo possível de correlação entre os valores aleatórios gerados.

3.3 Monitor

Implementada de forma embarcada na nossa simulação, a camada de monitoramento pode ser classificada como um event-driven monitor, responsável pela medição correta das métricas selecionadas para o estudo. É uma camada presente tanto no gerador de cargas quanto no processamento de eventos, estando distribuída pelos módulos da simulação. Assim como a maioria dos monitores [Jai91], o nosso adiciona um certo overhead aos testes, já que esta camada também consome pode computacional. Entretanto, o mesmo overhead é adicionado igualmente à todos os testes, então este pode ser desconsiderado da análise comparativa pois estamos procurando diferenças performáticas entre operadores de paradigmas diferentes.

```
struct monitorTicket {
    // Event details
    var type: String
    var data: Int
    var date: Date
    var id: Int
    let formatter = DateFormatter()
    func toString() -> String {
        formatter.dateFormat = "HH:mm:ss.SSSS"
        return "\(type),\(id),\(data),\(formatter.string(from: date))"}
}
func printDate(string: String) {
    let date = Date()
    let formatter = DateFormatter()
    formatter.dateFormat = "HH:mm:ss.SSSS"
    print(string + formatter.string(from: date))
}
class Monitor {
    var tickets = Array<monitorTicket>()
    public func getTickets() -> Array<monitorTicket> {
        return tickets
    }
    public func append(ticket: monitorTicket) {
        tickets.append(ticket)
    }
}
```

O monitor é responsável por acumular tickets de todos os eventos gerados na simulação e de pontos de interesse de processamento, como início e fim do cálculo de um operador estatístico ou tempo de resposta de uma carga de eventos. Ao término da simulação, os timestamps acumulados são devolvidos em forma de relatório para que possam ser analisados.

```
//Triggering Functions
func kpaUpdate(timeout: Int) {
    self.simulation.adjustKeepAlive()
    self.monitor.append(ticket: monitorTicket(type:"KTU",data:timeout,date:Date(), id: 9))
}
func tempUpdate(temp: Int) {
    self.simulation.decreaseTemperature()
    self.monitor.append(ticket: monitorTicket(type:"TCU",data:temp,date:Date(), id: 9))
}
func marginUpdate(value: Double) {
    self.simulation.decreaseMargin()
    self.monitor.append(ticket: monitorTicket(type:"MCU",data:Int(value),date:Date(), id: 9))
}
```

3.3.1 Responsividade: Turnaround e Response Time

O turnaround time é calculado pela diferença entre o timestamp de entrada de uma carga de eventos e o timestamp de um evento complexo gerado a partir da carga. O response time é calculado pelo tempo entre a chegada de um evento a ser processado e o resultado desse processamento.

3.3.2 Produtividade: I/O Throughput

A produtividade pode ser calculada pela fator de transação (entrada e saída) de eventos por unidade de tempo da simulação. Usaremos a razão TPS (transactions per second) para sua

notação.

3.4 Eventos

Os nossos eventos de workload seguem um protocolo customizado chamado NumericEvent, para que possam fornecer acesso a operadores que manipulam eventos com valores numéricos.

```
public protocol NumericEvent: Numeric, Event {
    static func / (lhs: Self, rhs: Self) -> Self
    static func * (lhs: Self, rhs: Self) -> Self
    var numericValue: Int { get set }
}
```

NetworkStatus

Evento de status da rede de sensores simulada, contendo os parâmetros atuais da simulação da rede de sensores. Na prática, representa o gateway da nossa rede, e fornece os valores para os parâmetros atuais da nossa simulação.

```
class NetStatusEvent: Event {
    var timestamp: Date
    var noSensors: Int
    var qos: Int
    var readFreq: Int
    var kpaTimeout: Int

    init(date: Date, noSensors: Int, qos: Int, readFreq: Int, kpaTimeout: Int) {
        self.timestamp = date
        self.noSensors = noSensors
        self.qos = qos
        self.readFreq = readFreq
        self.kpaTimeout = kpaTimeout
    }
}
```

SensorRead

Evento de leitura de um sensor da rede de sensores. Carrega o valor da leitura de temperatura de uma sala, sendo medida em diferentes lugares a fim de detectar diferenças de temperatura e permitir uma análise geral do ambiente.

```
final class SensorReadEvent: NumericEvent, Comparable {
    var magnitude = Int()
    typealias Magnitude = Int
    typealias IntegerLiteralType = Int

    var timestamp: Date
    var numericValue: Int

    init(date: Date, data: Int) {
        self.timestamp = date
        self.numericValue = data
    }
```

Temperature Control Update

Evento complexo gerado a partir do processamento de eventos de leitura dos sensores. Em nossa simulação, o Control Update simula o ajuste automático do controle de temperatura de uma sala a partir dos eventos de leitura gerados pelos sensores.

```
class TemperatureUpdateEvent: Event, Comparable {
    var timestamp: Date
    var temp: Int

    init(date: Date, temperature: Int) {
        self.timestamp = date
        self.temp = timeout
    }
```

Há duas regras de atualização de temperatura em nossas simulações: a primeira é de que o sensor deve ser ajustado quando a esperança de temperatura calculada pelos sensores se torne maior do que o termo prefixado de 1500. O ajuste deve diminuir a margem de erro de leitura dos sensores e assim diminuir a variância das leituras de temperatura.

```
// Update room temperature if average temperature is > maxTemperature
let tempUpdateRule = tempEvents.stream
    .average(timeWindow: tempTimeWindow, currentDate: Date())
tempUpdateRule.subscribe(onNext: {
    tempCounter+=1
    if ($0 > maxTemperature) {
        if (self.checkIntegrity()) {
            self.tempUpdate(temp: $0)
        }
    }
}).disposed(by: disposeBag)
```

A segunda regra é que a temperatura deve ser reduzida caso a média de leitura dentro de uma janela de tempo recente ultrapasse um valor máximo definido. O ajuste deve diminuir a temperatura do ambiente como um todo, temporariamente até que ela se eleve de novo na simulação.

```
// Adjust sensor reading margin
let marginUpdateRule = tempEvents
    .stream
    .variance(dataSize: noIterations*noSensors)

marginUpdateRule.subscribe(onNext: {
        if ($0 > 1000) {
            self.marginUpdate(value: $0)
        }
    }).disposed(by: disposeBag)
}
```

Timeout

Algumas redes de sensores precisam verificar constantemente sua conexão com seus dispositivos, processo normalmente feito através da troca de mensagens keepalive com cada um de seus sensores. Um evento de timeout e invocado caso o censor falhe em responder a requisição de keepalive dentro de um tempo de tolerância, o keepalive timeout.

```
final class TimeoutEvent: NumericEvent, Comparable {
    var magnitude = Int()
    typealias Magnitude = Int
    typealias IntegerLiteralType = Int

    var timestamp: Date
    var numericValue: Int

    init(date: Date, data: Int) {
        self.timestamp = date
        self.numericValue = data
    }
```

Keepalive Timeout Update

Evento complexo gerado a partir do processamento de eventos de timeout dos sensores. Em nossa simulação, o Keepalive Timeout Update simula o ajuste automático de keepalive timeout de uma rede de sensores.

```
class KpaUpdateEvent: Event {
    var timestamp: Date
    var timeout: Int

    init(date: Date, timeout: Int) {
        self.timestamp = date
        self.timeout = timeout
    }
}
```

A regra de atualização de keepalive usada em nossas simulações é de que o ajuste deve

ocorrer quando a média de ocorrência de timeout de um sensor ultrapassa a o valor definido dentro de um determinado intervalo de segundos. O valor de n varia de acordo com as simulações, enquanto o valor de ajuste é determinado pela razão entre frequência de keepalives e o número de sensores da rede.

```
// KeepAlive Timeout Rule
let kpaUpdateRule = timeoutEvents.stream
    .average(timeWindow: kpaTimeWindow, currentDate: Date())
kpaUpdateRule.subscribe(onNext: {
    kpaCounter+=1
    if ($0 > maxTimeout) {
        if (self.checkIntegrity()) {
            self.kpaUpdate(timeout: $0)
        }
    }
}).disposed(by: disposeBag)
```

CAPÍTULO 4

Experimentação

Nossa experimentação consiste de uma série de simulações com diferentes parâmetros para que possamos analisar a mudança de performance nas diferentes execuções. Nossas experimentações devem nos fornecer dados suficientes para que possamos fazer uma análise justa e imparcial do impacto performático em estudo. É importante que executemos mais de uma simulação por análise, para que possamos enxergar a tendência da mudança de performance de forma não enviesada por possíveis outliers.

4.1 Simulações

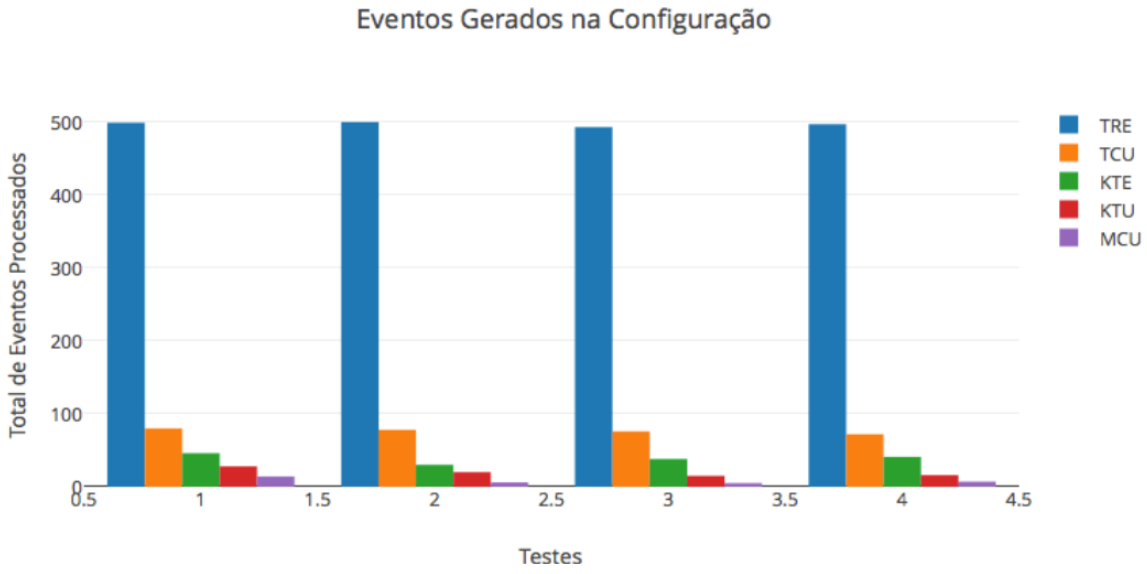
Realizaremos diferentes execuções de 4 configurações básicas de simulação da rede de sensores. Para cada configuração de simulação foram realizados um par de testes referente a cada paradigma de programação. Cada configuração de simulação procura explorar a diferença de performance apresentada pelos operadores, através de mudanças de parâmetros da rede simulada. Tais mudanças devem aumentar ou diminuir a frequência, quantidade e resposta de eventos no nosso sistema CEP. Todas as simulações foram executadas no XCode 9.4.1, com ambos o Swift e rxSwift na versão 4.0. Apresentamos a seguir a distribuição dos eventos gerados em cada um dos testes, sendo os dois primeiros execuções de operadores de paradigma reativos e os dois últimos de paradigma iterativo. É importante lembrar que as regras de frequência de geração de eventos da simulação adiciona um overhead de tempo. Esse overhead pode ser calculado como o número de simulações vezes o tempo de frequência do teste. Uma vez que ambos parâmetros são fixos em todos os nossos testes, as avaliações a seguir já subtraem esse overhead do resultado.

Teste 1: Configuração Base

Nossa configuração inicial se apresenta pelos seguintes parâmetros e resultados de teste:

```
// Parameters
let maxTimeout = 25  // Rule for max timeout value
let idealTemperature = 18 // Rule for expectancy temperature value
let maxTemperature = 20 // Rule for change temperature value
let maxMargin = 10  // Rule for Sensor adjustment
let noIterations = 50 // Number of reading to be performed by each sensor
let roomTemp = 30 // Starting room temperature
let tempGrowth = 2 // Rate at which temperature increases
let sensorErrMargin = 5 // Error margin for censor`s temperature reading
let sensorReadFreq = 2 // Frequency of sensor reading (sec)
let noSensors = 4 // Number of sensors (and threads) genereting events
let networkQoS = 50 // Rule for timeout Event
let kpaTimeout = 3 // Rule for keepAlive
```

Estes parâmetros foram escolhidos para que a primeira simulação tivesse o comportamento esperado de uma rede de sensores comum. Através deles, geramos os resultados:

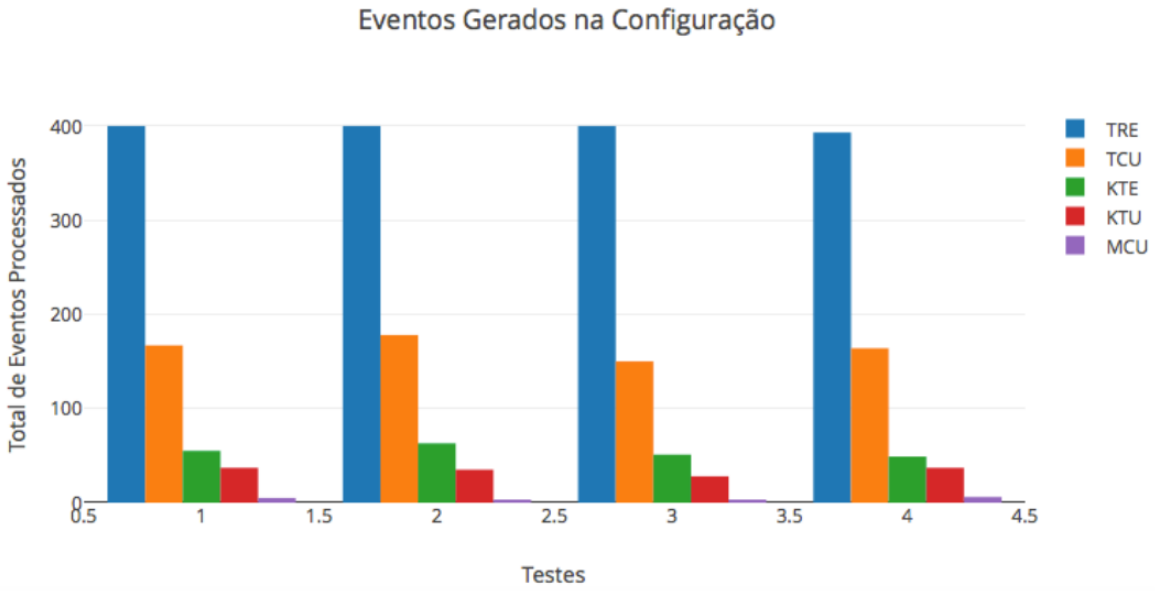


Teste Tipo	React #1	React #2	Iter #1	Iter #2
Total de Eventos	620	610	591	627
Consumidos	498	500	495	498
Produzidos	122	110	96	119
Turnaround	2.1740s	2.1810s	57.8070s	62.2301s
Throughput (t/s)	333,5	333	10,8	10,6

Teste 2: Mais Variância

Configuramos agora nossa simulação para que aumente a variância dos seus valores gerados e da mudança de temperatura na simulação. Assim podemos observar se a mudança do grau de aleatoriedade afeta de imediato o comportamento performático dos operadores.

```
// Parameters
let maxTimeout = 5  // Rule for max timeout value
let idealTemperature = 18 // Rule for expectancy temperature value
let maxTemperature = 20 // Rule for change temperature value
let maxMargin = 10  // Rule for Sensor adjustment
let noIterations = 50 // Number of reading to be performed by each sensor
let roomTemp = 30 // Starting room temperature
let tempGrowth = 4 // Rate at which temperature increases
let sensorErrMargin = 15 // Error margin for censor`s temperature reading
let sensorReadFreq = 2 // Frequency of sensor reading (sec)
let noSensors = 3 // Number of sensors (and threads) genereting events
let networkQoS = 15 // Rule for timeout Event
let kpaTimeout = 2 // Rule for keepAlive
```

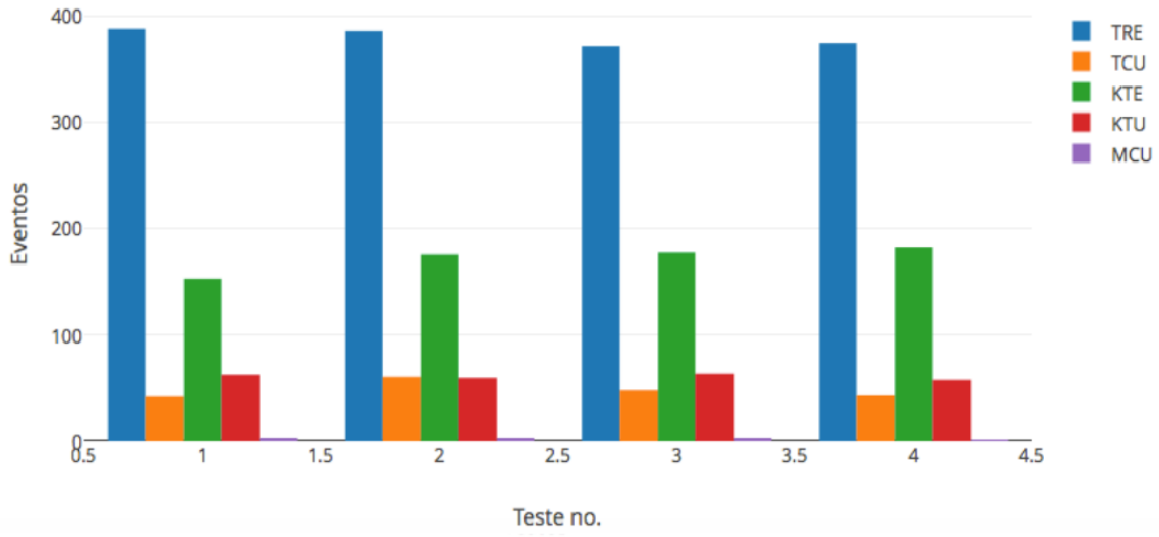


Teste Tipo	React #1	React #2	Iter #1	Iter #2
Total de Eventos	658	669	632	649
Consumidos	455	453	451	442
Produzidos	203	216	181	207
Turnaround	3.1801s	3.7300s	71.2770s	72.6320s
Throughput (t/s)	219,3	217	8,9	9

4.1.3 Rajada de Eventos

Configuramos agora nossa simulação a fim de gerar eventos com maior frequência. Para isso, reduzimos as frequências de atualização e ajustamos o cálculo do tempo de execução de acordo:

```
// Parameters
let maxTimeout = 5  // Rule for max timeout value
let idealTemperature = 18 // Rule for expectancy temperature value
let maxTemperature = 20 // Rule for change temperature value
let maxMargin = 10  // Rule for Sensor adjustment
let noIterations = 50 // Number of reading to be performed by each sensor
let roomTemp = 25 // Starting room temperature
let tempGrowth = 2 // Rate at which temperature increases
let sensorErrMargin = 5 // Error margin for censor`s temperature reading
let sensorReadFreq = 1 // Frequency of sensor reading (sec)
let noSensors = 3 // Number of sensors (and threads) genereting events
let networkQoS = 50 // Rule for timeout Event
let kpaTimeout = 1 // Rule for keepAlive
```

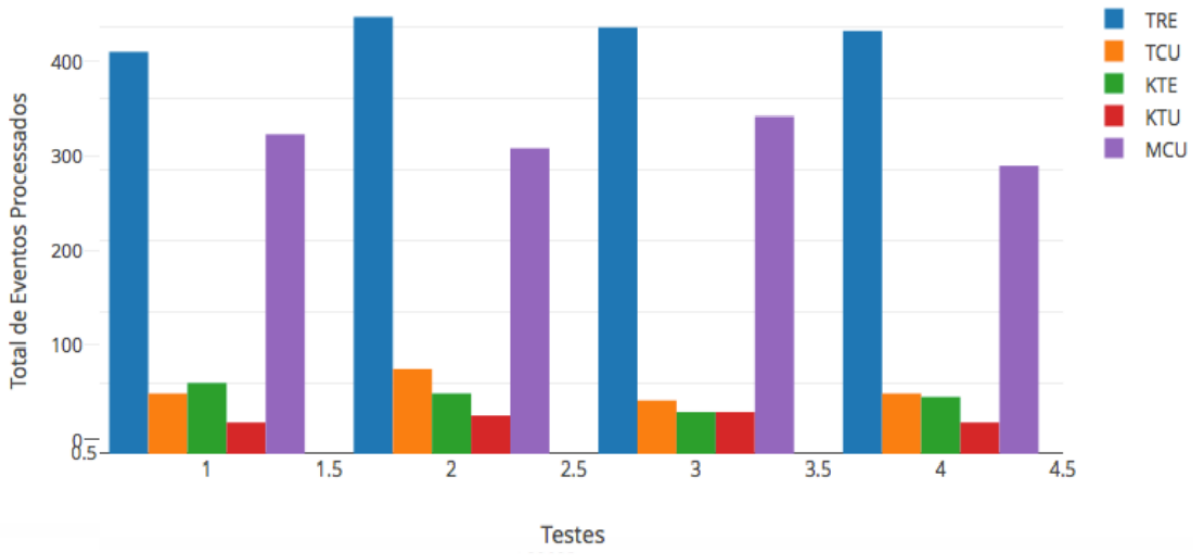


Teste Tipo	React #1	React #2	Iter #1	Iter #2
Total de Eventos	649	685	665	500
Consumidos	541	562	550	438
Produzidos	108	123	115	76
Turnaround	4.0860s	3.3420s	65.0440s	69.5800s
Throughput (t/s)	139,5	142,3	8,6	8,9

4.1.4 Evento Complexo: MCU

Configuramos agora nossa simulação para gerar muitos eventos MCU. Em nosso teste esse evento complexo é gerado através do cálculo da variância, o operador com maior complexidade computacional dentre os implementados já que realiza a operação para todos os eventos gerados por aquele observable. Faremos este teste para observarmos se a mudança de comportamento performático acompanha o aumento de complexidade das transações.

```
// Parameters
let maxTimeout = 25  // Rule for max timeout value
let idealTemperature = 22 // Rule for expectancy temperature value
let maxTemperature = 60 // Rule for change temperature value
let maxMargin = 15  // Rule for Sensor adjustment
let noIterations = 50 // Number of reading to be performed by each sensor
let roomTemp = 60 // Starting room temperature
let tempGrowth = 5 // Rate at which temperature increases
let sensorErrMargin = 30 // Error margin for censor`s temperature reading
let sensorReadFreq = 1 // Frequency of sensor reading (sec)
let noSensors = 3 // Number of sensors (and threads) genereting events
```



Teste Tipo	React #1	React #2	Iter #1	Iter #2
Total de Eventos	861	829	812	795
Consumidos	455	432	441	428
Produzidos	406	397	371	376
Turnaround	7.0060s	9.8160s	127.2560s	130.7450s
Throughput (t/s)	123	92,1	6,3	6,1

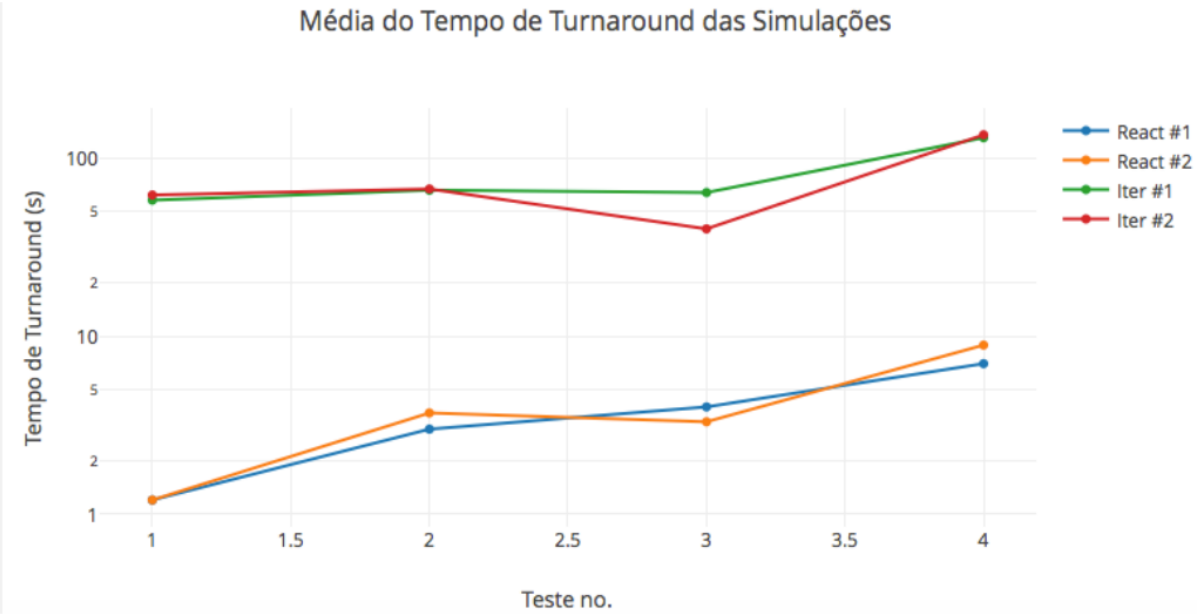
CAPÍTULO 5

Análise

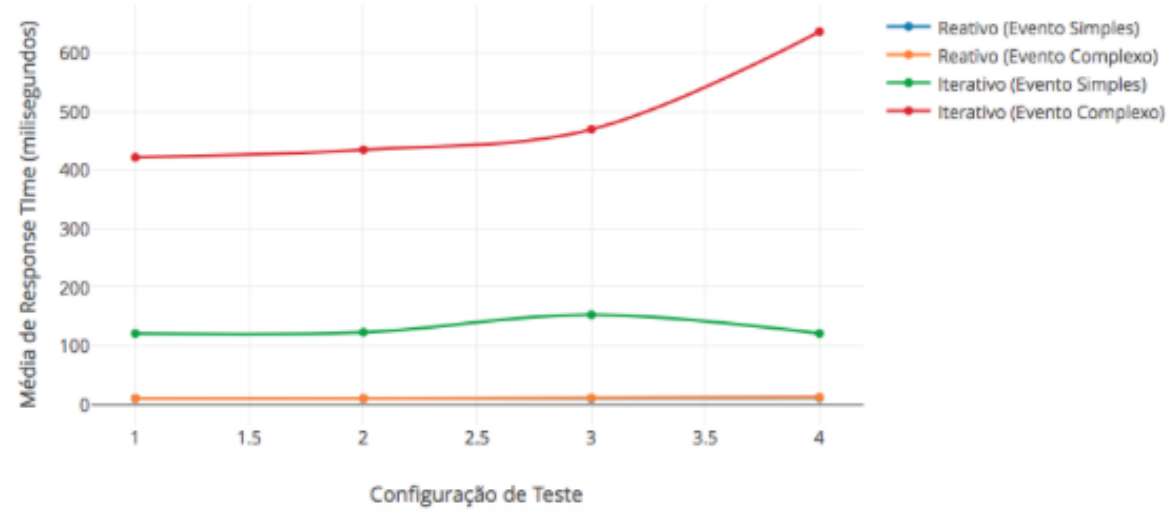
5.1 Quadro Geral

Podemos verificar imediatamente a vantagem da programação reativa em termos de performance já na primeira configuração de testes. Na configuração base o throughput conferido pelos operadores de paradigma reativo chega a ser 30 vezes maior do que o encontrado através de operadores iterativos. De fato, o mesmo pode ser observado na segunda configuração de testes, onde o **throughput** reativo é, em média 24 vezes maior e em todas as configurações restantes. No geral, a capacidade de transações por segundo dos operadores reativos encontrada em nossos testes foi, pelo menos, **20 vezes maior**.

Isso implica diretamente em tempos de **turnaround** do processamento de operadores, onde foi possível identificar que, para todos os testes das configurações apresentadas, o turnaround de processamentos que usavam operadores reativos era, pelo menos, **10 vezes menor** do que o encontrado com o uso de operadores iterativos.



A diferença fica ainda mais evidente quando avaliamos a métrica de response time dos operadores para eventos simples (TRE e KTE) e eventos complexos (TCU, KTU e MCU). Nela podemos verificar que os operadores reativos não somente tem performance melhor, mas também mais estável e confiável. Em média o response time de eventos simples e eventos complexos são quase indistinguíveis quando usamos operadores reativos.



Além disso, o aumento da frequência de eventos consumidos (objetivo da configuração 3) não é refletido nos operadores reativos, enquanto causa um aumento visível em operadores iterativos. A configuração de testes 4, que diz respeito ao aumento da variância dos inputs gerados nos ajuda a evidenciar a diferença de crescimento de custo temporal, principalmente por que ela requer que todo o conjunto de dados seja analisado constantemente. Essa repetição de instruções causa um grande impacto na performance dos operadores iterativos, gargalo que não se apresenta nos operadores reativos por conta da forma como as mudanças são propagadas funcionalmente.

CAPÍTULO 6

Conclusão

Este trabalho apresentou um estudo preliminar sobre as vantagens do uso do paradigma reativo para o processamento de eventos complexos. Espera-se que os experimentos, dados e análises descritos neste documento servem como motivação para a disseminação da programção reativa nas mais diversas áreas de trabalho computacional. O CEPSwift é o um framework pioneiro por aproximar os conceitos de processamento de eventos complexos e programação reativa na linguagem swift, e este documento espera ter provado, na prática, que este é um caminho frutífero.

6.1 Trabalhos Futuros

É esperado que os operadores implementados por este estudo possam servir como fundação para a construção de operadores de análise estatística mais robustos, completos e gratificantes. É importante ressaltar que a avaliação de performance através da simulação, apesar de válida e significativa e ter valor como argumento (principalmente em nosso contexto de comparatividade) não é suficiente por si só como comprovação de análise performática. Espera-se que em trabalhos futuros a mesma avaliação de performance seja feita através de modelos analíticos, para que junto à simulação, possam transmitir uma análise de performance ainda mais concreta e confiável [Jai91]. O material produzido pode ser encontrado neste link.

Referências Bibliográficas

[ACMZ15] A. Akbar, F. Carrez, K. Moessner, and A. Zoha. Predicting complex events for pro-active iot applications. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, pages 327–332, Dec 2015.

[BG17] George Belo Guedes. Cepswift: Complex event processing framework for swift undergraduate thesis george belo guedes. 12 2017.

[Cor06] Inc. Coral8. Complex event processing: Ten design patterns. 2006.

[CYH+17] S. Choochotkaew, H. Yamaguchi, T. Higashino, M. Shibuya, and T. Hasegawa. Edgecep: Fully-distributed complex event processing on iot edges. In *2017 13th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 121–129, June 2017.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques For Experimental Design, Measurement, Simulation, and Modeling, NY: Wiley*. 04 1991.

[LB14] Chunhui Li and Robert Berry. Cepben: A benchmark for complex event processing systems. In Raghunath Nambiar and Meikel Poess, editors, *Performance Characterization and Benchmarking*, pages 125–142, Cham, 2014. Springer International Publishing.

[Luc08] David Luckham. The power of events: An introduction to complex event processing in distributed enterprise systems. In Nick Bassiliades, Guido Governatori, and Adrian Paschke, editors, *Rule Representation, Interchange and Reasoning on the Web*, pages 3–3, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[MM06] Brenda M Michelson. Event-driven architecture overview: Event-driven soa is just part of the eda story. 01 2006.

[MS13] Alessandro Margara and Guido Salvaneschi. Ways to react : Comparing reactive languages and complex event processing. 2013.

[QSF+16] Yongrui Qin, Quan Sheng, Nickolas Falkner, Schahram Dustdar, Hua Wang, and Athanasios Vasilakos. When things matter: A survey on data-centric internet of things. 64, 02 2016.

[Rea18a] ReactiveX Official Documentation. Filter operator diagram, 2018. [Online; accessed June 25, 2018].

[Rea18b] ReactiveX Official Documentation. Map operator diagram, 2018. [Online; accessed June 25, 2018].

[RW10] David B. Robins and Redmond Wa. Complex event processing. In *In CSEP 504*, 2010.

[SGV+04] Eduardo Souto, Germano Guimarães, Glauco Vasconcelos, Mardoqueu Vieira, Nelson Souto Rosa, and Carlos André Guimarães Ferraz. A message-oriented middleware for sensor networks. In *Middleware for Pervasive and Ad-hoc Computing*, 2004.

[SZGA15] Martin Strohbach, Holger Ziekow, Vangelis Gazis, and Navot Akiva. *Towards a Big Data Analytics Framework for IoT and Smart City Applications*, pages 257–282. Springer International Publishing, Cham, 2015.

Este volume foi tipografado em LaTeX na classe UFPEThesis (www.cin.ufpe.br/~paguso/ufpethesis).