Pontifícia Universidade Católica
do Rio de Janeiro

Thiago Duarte Naves

Comparação dos Modelos ReactiveX e Programação Reativa Estruturada em Aplicações Soft Real Time

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática, do Departamento de Informática da PUC-Rio .

Orientador: Profª. Noemi de La Rocque Rodriguez

Rio de Janeiro
Março de 2021

PUC-Rio - Certificação Digital Nº 1913163/CA

Pontifícia Universidade Católica
do Rio de Janeiro

Thiago Duarte Naves

Comparação dos Modelos ReactiveX e Programação Reativa Estruturada em Aplicações Soft Real Time

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática da PUC-Rio . Aprovada pela Comissão Examinadora abaixo:

Profª. Noemi de La Rocque Rodriguez
Orientador
Departamento de Informática – PUC-Rio

Profª. Ana Lúcia de Moura
Pesquisadora Independente

Prof. Francisco Figueiredo Goytacaz Sant'Anna
UERJ

Prof. Roberto Ierusalimschy
PUC-Rio

Rio de Janeiro, 18 de Março de 2021

PUC-Rio - Certificação Digital Nº 1913163/CA

Todos os direitos reservados. A reprodução, total ou parcial do trabalho, é proibida sem a autorização da universidade, do autor e do orientador.

Thiago Duarte Naves

Graduado em engenharia de computação pela Pontifícia Universidade Católica do Rio de Janeiro.

PUC-Rio - Certificação Digital Nº 1913163/CA

Ficha Catalográfica

Duarte Naves, Thiago

Comparação dos Modelos ReactiveX e Programação Reativa Estruturada em Aplicações Soft Real Time / Thiago Duarte Naves; orientador: Noemi de La Rocque Rodriguez. – 2021.

87 f: il. color. ; 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2021.

Inclui bibliografia

1. Informática – Teses. 2. Soft real-time. 3. Programação concorrente. 4. Programação reativa. 5. Orientação a eventos. I. Rodriguez, Noemi de La Rocque. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

PUC-Rio - Certificação Digital Nº 1913163/CA

Aos meus pais, irmãs e família
pelo apoio e encorajamento.

Agradecimentos

À minha orientadora Professora Noemi Rodriguez pelo estímulo e parceria na realização deste trabalho.

Aos meus pais, pela educação, carinho e apoio.

Aos meus amigos e familiares pelos muitos encontros divertidos.

Aos professores que participaram da Comissão examinadora.

Aos professores e funcionários do Departamento pelos ensinamentos e pela ajuda.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

PUC-Rio - Certificação Digital Nº 1913163/CA

Resumo

Duarte Naves, Thiago; Rodriguez, Noemi de La Rocque. **Comparação dos Modelos ReactiveX e Programação Reativa Estruturada em Aplicações Soft Real Time**. Rio de Janeiro, 2021. 87p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Nesse trabalho comparamos o uso da programação reativa estruturada com o uso do ReactiveX no desenvolvimento de aplicações reativas *soft real time*. Apresentamos implementações de aplicações em Lua que demonstram o uso desses modelos em diferentes situações, destacando as vantagens de cada um. Consideramos também o seu uso combinado em uma mesma aplicação. Além disso, implementamos um módulo que permite utilizar a programação reativa estruturada em Lua e utilizamos o módulo RxLua que implementa o modelo ReactiveX.

PUC-Rio - Certificação Digital Nº 1913163/CA

Palavras-chave

Soft real-time; Programação concorrente; Programação reativa; Orientação a eventos.

Abstract

Duarte Naves, Thiago; Rodriguez, Noemi de La Rocque (Advisor). **A Comparison of the Structured Reactive Programming and ReactiveX Models in Soft Real Time Applications**. Rio de Janeiro, 2021. 87p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

In this work we compare the use of structured reactive programming and ReactiveX in the development of reactive soft real time applications. We present application implementations using Lua that demonstrate the use of these models in multiple situations, pointing the advantages of using each one. Another consideration is combining both models in a single application. We also developed a module that allows the use of structured reactive programming in Lua and used the RxLua module which implements the ReactiveX model.

PUC-Rio - Certificação Digital Nº 1913163/CA

Keywords

Soft real-time; Concurrent programming; Reactive programming; Event driven.

Sumário

PUC-Rio - Certificação Digital Nº 1913163/CA

Lista de figuras

PUC-Rio - Certificação Digital Nº 1913163/CA

Lista de Códigos

PUC-Rio - Certificação Digital Nº 1913163/CA

Lista de Abreviaturas

ABS – *Antiblockier-Bremssystem*

HTTP – *Hypertext Transfer Protocol*

LED – *Light-Emitting Diode*

URL – *Uniform Resource Locator*

VoIP – *Voice over Internet Protocol*

PUC-Rio - Certificação Digital Nº 1913163/CA

PUC-Rio - Certificação Digital Nº 1913163/CA

A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools.

Douglas Adams, *Mostly Harmless.*

1
Introdução

Diversos tipos de aplicação precisam processar múltiplas entradas independentes e reagir dentro de intervalos de tempo predeterminados. Aplicações como jogos, sensoriamento e automação industrial se encaixam nessa categoria. Essas aplicações são ditas *reativas* (HAREL; PNUELI, 1985), pois devem reagir continuamente a estímulos externos em um ritmo ditado por eles. Para algumas dessas aplicações, o tempo de resposta é crítico, ou seja, a aplicação deve garantidamente responder dentro do prazo estipulado. Para outras, esse prazo é maleável e atrasos são tolerados.

Existem diversas abordagens e ferramentas para implementar esse tipo de sistema, envolvendo diferentes modelos para tratamento de concorrência. Um modelo que vem recebendo muita atenção é o ReactiveX (REACTIVEX,). Esse modelo foca no fluxo de dados e propõe um conjunto de operações prontas para transformá-los. Outro modelo que visa dar facilidades para o programador é o que chamaremos de blocos concorrentes, presente em linguagens síncronas reativas (SANT'ANNA et al., 2017; BERRY; GONTHIER, 1992). Esse modelo apresenta mecanismos de controle de fluxo especializados que ajudam a implementar reações a eventos e controlar a concorrência entre diferentes partes do código.

PUC-Rio - Certificação Digital Nº 1913163/CA

1.1
Motivação

No modelo ReactiveX, eventos são propagados em cadeias formadas por um objeto observável, uma sequência de operadores e observadores. Os operadores são objetos que aplicam alguma transformação no evento e são ao mesmo tempo observadores e observáveis. Dessa forma, podemos criar uma sequência de operadores para aplicar as transformações necessárias antes de entregar o evento a um observador na aplicação. Alguns desses operadores interagem com mais de uma cadeia simultaneamente, permitindo agir em função de múltiplos observáveis e implementar controle de fluxo. O ReactiveX especifica diversos operadores que implementam funcionalidades comuns visando ter um ferramental disponível para diferentes tipos de cenário. Esses operadores também são pensados como blocos de construção para lógicas mais complexas, uma vez que são tipicamente usados em conjunto em uma cadeia. Operadores, portanto, atuam como uma camada de abstração para o processamento de eventos, permitindo ao programador focar nas transformações necessárias sem

se preocupar com a sua implementação.

O modelo de blocos concorrentes foca no controle de fluxo em função de eventos. Nesse modelo, descrevemos a relação entre blocos de código definindo quais desses blocos podem executar concorrentemente. Ao mesmo tempo, indicamos se blocos devem esperar o término de outros ou se causam o encerramento de outros ao terminar. Dessa forma, criamos uma hierarquia de execução entre diferentes partes do código. A execução do código é controlada pela emissão de eventos. Blocos podem bloquear à espera de um evento e podem emitir eventos para outros blocos executarem. O bloqueio de um bloco congela o seu estado enquanto permite que outros blocos da aplicação executem, de forma que futuramente possa retomar a sua execução do ponto em que parou.

Os dois modelos trazem vantagens para o programador na implementação de reações a eventos e na manutenção de estado associada. Nas comunidades de cada modelo, vemos bons exemplos do seu uso e situações que se beneficiam dessas formas de programar. Entretanto, ao tentar desenvolver aplicações mais complexas, nos deparamos com cenários que fogem aos casos de uso típicos e requerem maior esforço ou implementações parcialmente fora do modelo. Assim, nem sempre é simples fazer uma implementação que siga perfeitamente a filosofia de qualquer dos modelos ou mesmo identificar qual é o mais adequado a um dado problema.

PUC-Rio - Certificação Digital Nº 1913163/CA

1.2 Objetivos do trabalho

Comumente, sistemas desse tipo são implementados utilizando-se *threads* em sistemas operacionais de tempo real. O uso de *threads* implica no uso de mecanismos de sincronização e no potencial para *bugs* decorrentes de condições de corrida. Entretanto, em muitos casos essas aplicações podem ser desenvolvidas sem o emprego de preempção. Neste trabalho, iremos estudar o desenvolvimento de aplicações *single-threaded*, sem preempção ou paralelismo no nível de aplicação. Buscamos, portanto, compreender as vantagens do uso de blocos concorrentes e do ReactiveX ao utilizar apenas concorrência na implementação de sistemas soft real-time. Também exploraremos a combinação desses modelos em uma aplicação buscando determinar possíveis vantagens de uma implementação híbrida.

Afim de demonstrar as diferentes características dos modelos, implementamos uma série de exemplos de diferentes níveis de complexidade em ambos. Ao observar a solução de um mesmo problema nos dois modelos, podemos perceber padrões de código e avaliar o tamanho e a complexidade da solução.

Finalmente, uma aplicação pequena, mas realista, demonstra o uso combinado dos modelos e o compara com implementações utilizando apenas um deles.

Demonstramos o uso de ambos os modelos em Lua para evitar diferenças entre linguagens ao compará-los e para que pudéssemos combiná-los em uma mesma aplicação mais facilmente. Para tal, utilizamos dois módulos Lua que implementam esses modelos: RxLua (RXLUA,) para o ReactiveX e Lua Tasks, desenvolvido por nós, para os blocos concorrentes. Essa abordagem nos permite não só combinar e comparar esses modelos, mas também usá-los em apenas parte de uma aplicação. Nenhum dos módulos possui um *loop* principal de execução. Dessa forma, utilizamos a *engine* de jogos Löve 2D (LÖVE2D,) para criar interfaces com o usuário e alimentar os módulos com eventos. Com essa arquitetura, o *loop* principal fica oculto pelo Löve 2D e a aplicação pode utilizar a interface dos módulos para reagir a estímulos.

Esse trabalho tem como objetivo secundário discutir o ReactiveX, que possui pouca literatura acadêmica publicada. Explicamos a estruturação de aplicações nesse modelo, destacando as ferramentas para controle de concorrência e manutenção de estado. Apresentamos programas de diferentes complexidades, partindo de um exemplo simples até uma aplicação realista que demonstrem situações em que cada modelo melhor se adéque.

PUC-Rio - Certificação Digital Nº 1913163/CA

1.3 Sistemas de tempo real

Nessa seção introduziremos brevemente os diferentes tipos de aplicações de tempo real, destacando os diferentes requisitos de tempo e garantias de cada um. Também citaremos exemplos de aplicações em cada modelo que ilustram a necessidade de tais garantias.

Aplicações de tempo real são programas que reagem continuamente a um fluxo de eventos externos, tendo que cumprir prazos determinados para executar. Dessa forma, esses programas executam em um ritmo determinado externamente, uma vez que não têm controle sobre o momento e a quantidade de eventos recebidos. A importância do cumprimento de prazos depende de cada tarefa a ser realizada pela aplicação em questão.

Podemos dividir esses sistemas em 3 grupos: *Hard real time*, *Firm real time* e *Soft real time*. No primeiro caso, não há utilidade da reação logo após o término do prazo. A falha no cumprimento de um prazo significa uma falha total do sistema. Tipicamente esse tipo de sistema é utilizado em situações em que a falha em atuar a tempo resulta em risco financeiro ou de perda de vidas. Por exemplo, uma linha de produção pode ser paralisada ou ter produtos danificados devido a uma falha de sincronização das máquinas. Da

mesma forma, a falha em atuar o mecanismo de um freio ABS em um carro pode resultar na perda do controle, colocando os ocupantes em risco. (SHIN; RAMANATHAN, 1994)

No caso dos sistemas *Firm real time*, o não cumprimento de um prazo também acarreta na reação não ter utilidade. Contudo, diferentemente do caso *hard real time*, a perda de um prazo não implica na falha do sistema, mas apenas na degradação da qualidade do serviço. Esse tipo de sistema tolera falhas eventuais das tarefas. Alguns jogos se encaixam nessa categoria. Em algumas implementações de jogos *online*, o servidor transmite o estado como um todo para o cliente periodicamente. Se uma dessas mensagens atrasar muito, é melhor descartá-la e considerar a mensagem seguinte, com um estado mais atualizado. Perdas de mensagens causam dessincronizações temporárias entre o servidor e o cliente, atrapalhando o jogador, mas sem necessariamente impedir o funcionamento do jogo.

Sistemas *soft real time*, por outro lado, toleram o não cumprimento de prazos. Nesse caso, a utilidade da reação decai com o tempo após passado o prazo de reação. Evidentemente, a perda de um prazo não implica em uma falha do sistema, mas apenas na degradação da qualidade do serviço. Esse tipo de sistema também tolera falhas eventuais das tarefas. Em sistemas VoIP, por exemplo, pequenos atrasos no envio da voz em uma conversa atrapalham a comunicação, mas não a impedem. Conforme o tempo de atraso aumenta, fica progressivamente mais difícil para os interlocutores manterem a comunicação. Perdas de pacotes de rede resultam em cortes no áudio, novamente degradando a qualidade da comunicação, mas não impedem a conversa de continuar em seguida.

É preciso notar que, mesmo nos casos em que falhas são toleradas, esse não deve ser o comportamento normal do sistema, apenas uma exceção no seu funcionamento. Os prazos das tarefas e os requisitos das mesmas devem ser determinados em tempo de projeto, afim de garantir que a aplicação consiga executar dentro dos limites de tempo esperados. Vale notar que tarefas podem depender da execução de outras tarefas e portanto o tempo de uma reação deve levar em consideração os prazos das várias tarefas envolvidas para determinar o pior caso combinado.

1.4 Estrutura da dissertação

Essa dissertação está dividida da seguinte forma: No capítulo 2 mostramos diferentes modelos de concorrência e explicamos as principais características de cada um. No capítulo 3 apresentamos o módulo Lua Tasks, desenvolvido

PUC-Rio - Certificação Digital Nº 1913163/CA

para permitir o uso do modelo de blocos concorrentes em Lua. No capítulo 4 comparamos o uso de cada modelo em diferentes aplicações e apontamos as vantagens de cada um. No capítulo 5 abordamos as dificuldades no *debug* de aplicações reativas e falamos de algumas ferramentas para auxiliar nesse processo. No capítulo 6 apresentamos nossas conclusões e propomos trabalhos futuros.

PUC-Rio - Certificação Digital Nº 1913163/CA

2
Modelos de Concorrência

Este capítulo discute as dificuldades do tratamento concorrente de múltiplas requisições em aplicações de tempo real. Apresentamos vários modelos de programação utilizados no tratamento de processamento concorrente e destacamos as vantagens e complexidades do uso de cada um.

Em sistemas de tempo real, a aplicação deve ser capaz de reagir a múltiplos estímulos externos dentro de prazos preestabelecidos. Devido à natureza assíncrona e à ausência de correlação entre esses estímulos, a aplicação deve gerenciar a execução concorrente das reações. Além de concorrer pelo tempo de CPU, essas reações podem utilizar recursos compartilhados, como objetos em memória e periféricos de hardware, que devem ser acessados de forma coordenada. Para que a aplicação se mantenha responsiva e seja capaz de atender a múltiplas requisições simultâneas em tempo hábil, também é necessário que aquelas que estiverem esperando algum recurso ou evento não impeçam as demais de executar.

As mudanças no mundo externo são comumente modeladas como eventos na aplicação. A necessidade de reação a esses eventos de forma determinística pode levar a implementações complexas e propensas a erros. Modelos como a hipótese síncrona surgem para fornecer garantias de determinismo baseadas em fundamentos matemáticos e técnicas de verificação formais (POTOP-BUTUCARU; SIMONE; TALPIN, 2005). Esse modelo divide as reações em uma sequência de intervalos de tempo discretos que não se sobrepõem. Em cada intervalo, as entradas do programa são lidas e as computações sobre elas são executadas até que se chegue em um novo estado da aplicação e valores de saída. A hipótese síncrona também garante que dependências entre eventos e variáveis sejam respeitadas. Ou seja, o valor de uma variável e a presença e valor de um evento já estarão determinados quando forem lidos. Essas garantias tornam mais fácil compreender o programa e raciocinar sobre o seu comportamento.

2.1
Modelos clássicos

2.1.1
Threads e corrotinas

Um modelo comumente utilizado em sistemas de tempo real é o de *threads* preemptivas. Nesse modelo, as reações podem ser interrompidas em

PUC-Rio - Certificação Digital Nº 1913163/CA

qualquer ponto e a execução de outra retomada. Essa característica faz com que reações que não interajam possam ser pensadas como se fossem únicas no sistema. Entretanto, reações que compartilhem estado necessitam de mecanismos de sincronização, tipicamente de exclusão mútua, para que o acesso concorrente a recursos compartilhados seja feito de forma consistente. Essa coordenação entre as *threads* é frequentemente complexa e propensa a erros. Ao introduzir interações entre as *threads*, violamos a hipótese síncrona, pois não há garantia da ordem em que acessarão variáveis compartilhadas. Também é preciso considerar que duas *threads* esperando por eventos diferentes não necessariamente executarão na ordem em que esses eventos ocorrerem, gerando um comportamento não determinístico.

Por cada *thread* ter uma pilha de execução própria e sua execução poder ser suspensa, a manutenção do estado é simples e automática. Podemos utilizar chamadas bloqueantes e procedimentos computacionalmente intensivos sem impedir que as demais reações do sistema executem. Essa característica melhora a legibilidade do código e simplifica a implementação.

Poder interromper a execução de uma *thread* permite implementar prioridades diferentes para cada uma. Dessa forma, o escalonador pode interromper a execução de uma *thread* menos prioritária quando uma de maior prioridade ficar pronta para executar. Essa funcionalidade é comumente encontrada em sistemas operacionais de tempo real e fornece um maior controle sobre o tempo de resposta de reações importantes.

Corrotinas (MOURA; IERUSALIMSCHY, 2009), ou *threads* cooperativas, são muito semelhantes às *threads* preemptivas. Corrotinas também podem dedicar um contexto de execução para cada reação e utilizar chamadas aparentemente bloqueantes. Contudo, nesse caso não há troca automática de contexto, de forma que cada corrotina executa até que passe explicitamente o controle para outra. Assim, a troca de contexto ocorre em pontos bem definidos, facilitando entender em que momentos pode haver concorrência e evitando que mecanismos de exclusão mutua sejam necessários. Por não haver troca de contexto automática, condições de corrida e *deadlocks* são muito menos comuns.

Diferente das *threads*, contudo, a ausência de troca de contexto automática faz com que processamentos lentos impeçam as demais reações de executar, o que pode levar à necessidade de dividir o processamento em partes ou inserir pontos em que a corrotina suspenda sua execução para que o sistema continue reativo. Da mesma forma, o escalonador não tem como interromper uma reação menos prioritária quando uma mais prioritária precisar executar.

0Em ambos os casos, APIs síncronas devem realizar trocas de contexto

PUC-Rio - Certificação Digital Nº 1913163/CA

internamente. No caso de *threads*, o sistema operacional passará o controle para outra *thread*. Utilizando corrotinas, a aplicação simula uma chamada síncrona utilizando internamente mecanismos não bloqueantes e trocando de corrotina apropriadamente. Dessa forma, tanto para a *thread* como para a corrotina, a chamada parece síncrona mas internamente causa uma troca de contexto. Apesar da semelhança na forma de programar com *threads* e corrotinas, há a diferença fundamental de que o escalonamento de *threads* é assíncrono e de corrotinas é síncrono. Em aplicações sem paralelismo, essa distinção afeta principalmente o acesso a variáveis compartilhadas, já que tipicamente apenas as *threads* necessitam de mecanismos de exclusão mutua.

Tanto no caso de *threads* como no de corrotinas, o controle do estado fica implícito, mantido na pilha de execução como aconteceria se o programa não precisasse tratar múltiplos eventos simultaneamente. Apesar de simplificar o controle do estado em muitos casos, interfaces síncronas podem dificultar a interação entre diferentes eventos, visto que para tal pode ser necessário expor parte do estado para outra parte da aplicação. Por exemplo, uma variável de conexão poderia ser compartilhada para implementar um *timeout* de uma chamada ou para atender a uma requisição de monitoramento de progresso. Isso dificulta o entendimento da aplicação, por não ser imediatamente claro quais outras partes da aplicação afetam uma reação, além de poder gerar acesso concorrente ao estado.

PUC-Rio - Certificação Digital Nº 1913163/CA

2.1.2 APIs assíncronas, eventos e callbacks

Outro caminho comum é a utilização de APIs assíncronas para acessar recursos cujo tempo de resposta seja grande. Funções que não podem bloquear tipicamente utilizam esse tipo de chamada em situações nas quais uma chamada síncrona demoraria excessivamente. Essas APIs iniciam o acesso em paralelo e tipicamente recebem como parâmetro uma *callback* que será executada quando o acesso for concluído. Dessa forma, a chamada pode retornar imediatamente e o processamento continuará quando a *callback* executar. Essa quebra em duas etapas gera algumas dificuldades na implementação. A função que faz a chamada teve que ser dividida em duas, pois parte executará antes da chamada assíncrona e parte na *callback* ao final dessa execução. Com isso, por vezes é necessário encapsular o contexto de execução para que a função de *callback* tenha acesso às variáveis locais da reação que precisam ser compartilhadas entre as duas funções. Como a primeira função retorna antes que a *callback* execute, essas variáveis devem ser alocadas na *heap* para que não sejam desalocadas antes do término da reação, podendo acarretar em um gerenciamento

manual de memória que de outra forma seria desnecessário. Vale notar que reações que envolvam múltiplas chamadas assíncronas devem implementar uma *callback* para cada chamada, tipicamente iniciando a próxima quando a atual é concluída. Em aplicações em que não há processamento paralelo, esse tipo de API deixa os pontos em que pode haver concorrência explícitos, facilitando entender em quais pontos de uma reação ela deixa de ser atômica. Assim, fica claro em quais momentos pode ser necessário reavaliar o estado global (ADYA et al., 2002). Por ter uma assinatura diferente de uma função síncrona equivalente, ao alterar uma interface de síncrona para assíncrona é necessário alterar também todas as funções que a utilizam. Se por um lado isso acarreta mais mudanças no código, por outro impede que erros surjam de maneira silenciosa pela introdução de concorrência em uma função que outrora era atômica.

Em aplicações orientadas a eventos, módulos enviam mensagens (comumente chamadas de eventos) em canais de comunicação para sinalizar mudanças de estado ou acontecimentos notáveis. Outros módulos podem observar esses canais para receber as mensagens e reagir apropriadamente. Uma mesma aplicação pode utilizar múltiplos canais para melhor organizar os eventos e para que os módulos tenham um controle mais granular sobre quais eventos receberão. Tipicamente os módulos observadores registram uma função de *callback* no canal para serem notificados sobre novos eventos. Em implementações síncronas, as *callbacks* registradas são chamadas em sequência quando um novo evento é enviado no canal.

Como são baseadas em *callbacks*, as reações aos eventos apresentam dificuldades de manutenção de estado semelhantes as do uso de interfaces assíncronas. Em ambos os casos o processamento não é todo feito em uma só função, sendo preciso armazenar as variáveis de estado em um contexto global para ser retomado em uma outra chamada. Contudo, o tratamento de eventos se diferencia por iniciar em decorrência de um acontecimento externo ao módulo, ou mesmo à aplicação. Isso requer que a aplicação possa lidar com múltiplos eventos do mesmo tipo simultaneamente.

Para ilustrar essa situação, suponha um controlador em uma fábrica que monitora sensores e aciona atuadores em função dessas leituras. Suponha também que alguns sensores se comunicam com protocolos que necessitam de algumas trocas de mensagens, enquanto outros enviam suas leituras periodicamente. Como as trocas de mensagem são relativamente lentas e o controlador precisa monitorar vários sensores simultaneamente, este se comunica concorrentemente com todos os sensores. Além disso, o controlador deve poder tratar as mensagens dos sensores que enviam periodicamente a qualquer momento. Assim, o controlador deve instanciar um contexto para cada sensor para guar-

PUC-Rio - Certificação Digital Nº 1913163/CA

dar o estado da comunicação com cada um, atualizando-o em função das mensagens.

Programas que precisam tratar múltiplos eventos concorrentes de um mesmo cliente são ainda mais complexos. Diferente do tratamento de múltiplos clientes simultaneamente, os eventos podem ter interações e atuam sobre um mesmo estado compartilhado. Mesmo com a ausência de preempção, é preciso cuidado no acesso concorrente às variáveis compartilhadas devido ao acesso por múltiplas chamadas assíncronas.

Reações ao tempo, da mesma forma que eventos externos, são tratadas com *callbacks*. Tipicamente essas reações controlam ações periódicas ou *timeouts*. Como não podemos bloquear a execução dentro de *callbacks*, é comum que essas reações tenham que compartilhar estado com outras reações. Em alguns casos, no entanto, podemos apenas ler o relógio global para determinar se um evento deve ser ignorado ou aferir o intervalo de tempo entre dois eventos.

Por exemplo, um jogo precisa reagir rapidamente aos comandos do jogador ao mesmo tempo em que coordena múltiplas reações resultantes desses comandos. Assim, uma ação de salto leva a uma animação que depende de uma simulação física da gravidade que dura um certo período de tempo. Outra ação, seja do mesmo jogador ou não, pode influenciar essa reação, como a colisão com o cenário ou com outro jogador. Todas essas interações levam a representações complexas de estado e, consequentemente, à dificuldade de atualização do mesmo.

2.2 Discussão

2.2.1 Reações ao tempo

Além de reagir a estímulos externos, aplicações reativas frequentemente precisam reagir à passagem do tempo, seja limitando o tempo de espera ou execução, seja executando procedimentos periodicamente. Para isso, as aplicações podem utilizar diferentes APIs. Um relógio global (*timestamp*) oferece uma contagem absoluta do tempo decorrido desde um momento de referência. Temporizadores executam funções periodicamente com um intervalo definido pela aplicação. Funções como *sleep* ou *delay* suspendem a execução de uma *thread* ou corrotina por um determinado período. Eventos de relógio podem ser emitidos em intervalos regulares, entre outras interfaces possíveis. Tipicamente, bibliotecas de concorrência oferecem um subconjunto dessas interfaces compatível com seu modelo de programação.

PUC-Rio - Certificação Digital Nº 1913163/CA

2.2.2 Callback hell

Aplicações que precisam implementar sequências relativamente longas de chamadas assíncronas por vezes o fazem aninhado ou encadeando chamadas em *callbacks*, de forma que cada chamada ao concluir inicie a seguinte. Contudo, isso faz com que um conjunto de operações que semanticamente formam uma única função seja dividido em múltiplas, dificultando seu entendimento. Esse problema é agravado em linguagens sem funções lambda por aumentar a distância entre a implementação das *callbacks* e o ponto em que são referenciadas. Além disso, é necessário referenciá-las pelo nome, que tende a ser pouco descritivo uma vez que elas são múltiplas partes de um mesmo procedimento. Em linguagens sem *closures* há ainda o agravante de um maior gerenciamento manual do contexto compartilhado entre essas chamadas, que deve ser descrito em alguma estrutura que possa ser passada por parâmetro para as *callbacks*. Essa estrutura deve ser alocada na *heap*, uma vez que a função que iniciou a cadeia de chamadas já terá retornado quando as *callbacks* executarem. Consequentemente, é preciso desalocar essa estrutura ao final da execução ou em casos de erro, caso não haja coleta de lixo. Essa situação é conhecida como *callback hell* (EDWARDS, 2009), pela dificuldade em compreender o código decorrente das múltiplas *callbacks*, bem como pela quantidade de gerenciamento manual de estado envolvida.

Tanto a manutenção manual de estado como a organização do código são questões comuns no desenvolvimento de aplicações reativas. Na próxima seção, apresentaremos os modelos que estudaremos neste trabalho, que apresentam abordagens diferentes para o tratamento dessas questões.

PUC-Rio - Certificação Digital Nº 1913163/CA

2.3 Modelos reativos

2.3.1 Programação síncrona estruturada

Neste modelo, descrevemos as operações a serem executadas ao receber um evento de forma imperativa, como uma série de chamadas de funções aparentemente síncronas. Para que o programa continue responsivo apesar das chamadas bloqueantes, é necessário algum mecanismo que permita que a aplicação possa atender a outras reações enquanto uma reação está bloqueada em uma chamada. *Threads* e corrotinas são exemplos desses mecanismos, visto que em ambos os casos há implicitamente uma troca de contexto.

Blocos concorrentes, encontrados em linguagens reativas síncronas, como Céu (SANT'ANNA et al., 2017) e Esterel (BERRY; GONTHIER, 1992), são outra forma de utilizar chamadas síncronas e continuar atendendo a eventos simultâneos. Esses blocos facilitam coordenar a interação entre diferentes eventos e explicitar quais partes do código podem executar concorrentemente, bem como a relação entre elas. A comunicação entre os blocos pode ser feita por envio de mensagens para evitar o compartilhamento de variáveis e gerar um evento que notifica a chegada da mensagem. Assim, essa abordagem provê uma forma estruturada de criar relacionamentos entre eventos. Nesta seção, nos deteremos na descrição de Céu, por ser um dos objetos de estudo do trabalho.

Em Céu, as principais ferramentas para expressar a relação entre os blocos concorrentes (chamados de *trilhas*) são o `par/and` e o `par/or`. Esses operadores agrupam um conjunto de trilhas e as iniciam concorrentemente. Blocos `par/or` terminam quando qualquer uma das suas trilhas termina, encerrando as demais. São geralmente usados para abortar execuções, por exemplo implementando um *timeout*. Para tal, criamos um bloco com duas trilhas no qual uma irá realizar o trabalho e outra apenas esperar por um tempo determinado. Dessa forma, aquela que terminar primeiro causará o encerramento da outra. Independentemente de qual das trilhas terminar primeiro, o bloco como um todo termina nesse momento. Outro uso é uma função estilo `select`, onde queremos aguardar qualquer um entre múltiplos recursos estar disponível. Blocos `par/and`, por outro lado, só terminam quando todas as suas tarefas terminam, podendo ser usados como barreiras para esperar múltiplas tarefas ou como uma forma de agrupar múltiplas tarefas independentes em uma só.

Aninhando esses blocos podemos expressar relações mais complexas, por exemplo, se uma trilha de um `par/or` contiver um `par/and` e essa trilha for encerrada, todas as trilhas do `par/and` serão terminadas. Quando uma trilha inicia um `par/and` ou `par/or`, esta fica bloqueada até que todo o conjunto termine. O código 1 ilustra essa construção. Neste exemplo piscamos 2 LEDs, um deles 10 vezes e o outro 5 vezes. O `par/and` faz com que os LEDs pisquem ao mesmo tempo, mas sem que o término de um dos *loops* termine o outro. Se em qualquer momento o evento FIM for emitido, o `par/or` mais externo termina e encerra todas as trilhas, parando ambos os LEDs.

Um diferencial de Céu é ter o conceito de tempo como parte da linguagem. Como podemos observar nesse exemplo, pode-se utilizar o mesmo operador `await` para esperar um evento ou um período de tempo, inclusive utilizando diferentes unidades de tempo. Esse operador bloqueia a trilha até que o evento ocorra ou até que o tempo determinado passe. Para situações em que algo deve

PUC-Rio - Certificação Digital Nº 1913163/CA

executar periodicamente, podemos utilizar o operador `every` para repetir um bloco de código em intervalos regulares de tempo.

Código 1: Pisca LEDs em Céu

```
1  par/or do
2    par/and do
3      var int i;
4      loop i in [1 -> 10] do
5        led_off(1);
6        await 500ms;
7        led_on(1);
8        await 500ms;
9      end
10   with
11     var int i;
12     loop i in [1 -> 5] do
13       led_off(2);
14       await 500ms;
15       led_on(2);
16       await 500ms;
17     end
18   end
19 with
20   await FIM;
21 end
```

PUC-Rio - Certificação Digital Nº 1913163/CA

Por ter o envio de eventos e a relação de concorrência entre trilhas descritos com estruturas da linguagem, Céu consegue oferecer algumas garantias de execução que impedem ou alertam para problemas comuns. Em geral, essas garantias mantêm a aplicação funcionando de forma reativa e prevenindo condições de corrida. Por exemplo, *loops* infinitos devem esperar eventos a cada iteração utilizando uma chamada `await`. Isso faz com que outras trilhas possam executar, reagindo aos eventos recebidos enquanto a outra trilha executava. Para prevenir condições de corrida e reagir de forma determinística, trilhas acordadas por um mesmo evento executam na ordem em que aparecem no código. Uma vez que não há preempção, o controle passa para outra trilha apenas quando a trilha que está executando bloqueia em uma chamada `await`. O comportamento das trilhas é semelhante ao de corrotinas, que devem passar explicitamente o controle para que outra possa executar, ao mesmo tempo em que mantêm um contexto privado de execução.

Essas linguagens tendem a agrupar código de diferentes reações em um mesmo lugar, uma vez que a estrutura da linguagem descreve parte da lógica de interação delas. Pela mesma razão, pode ser necessário aninhar múltiplos blocos `par`. Assim, pode ser difícil compreender reações mais complexas que dependam de mais de um evento, além de dificultar a organização do código. Para minimizar esse problema, Céu oferece blocos `code/await`, equivalentes a

funções, que podem reagir a eventos e utilizar as estruturas de concorrência da linguagem. Isso permite reaproveitar o código (como com qualquer função) e encapsular as reações a eventos pertinentes para que não poluam o código mais externo. O código 2 mostra o mesmo programa de piscar LEDs utilizando `code/await` para abstrair o *loop* de piscar o LED, dado maior legibilidade ao código.

Código 2: Pisca LEDs em Céu com abstração

```
1  code/await Blink(var int count, var int led) -> none do
2    var int i;
3    loop i in [1 -> count] do
4      led_off(led);
5      await 500ms;
6      led_on(led);
7      await 500ms;
8    end
9  end
10 par/or do
11   par/and do
12     await Blink(10, 1);
13   with
14     await Blink(5, 2);
15   end
16 with
17   await FIM;
18 end
```

A combinação do operador `spawn` e um objeto `pool` oferece uma outra forma de iniciar trilhas. Ao utilizar esses dois recursos para iniciar um bloco `code/await`, o bloco é iniciado concorrentemente com o resto da aplicação, sem bloquear a trilha que fez a chamada. Um objeto `pool` é utilizado para armazenar os contextos dos blocos `code/await` iniciados. O código 3 mostra o uso de um `pool` para piscar 10 LEDs. Nesse exemplo, instanciamos um `pool` com capacidade para 10 contextos `Blink`. Isso significa que podemos ter até 10 instâncias do bloco `code/await Blink` executando. Se tentássemos executar um `spawn` em um `pool` que já está executando o máximo de blocos permitidos, a chamada `spawn` falharia e não teria efeito. `Pools` não precisam ter um limite predefinido, bastando para tal omiti-lo. Nesse caso, novos contextos são alocados conforme necessário.

Essa construção, além de fornecer uma forma mais organizada de iniciar trilhas que não dependem do contexto em que foram iniciadas, permite criar novas trilhas dinamicamente. Assim, podemos tratar casos em que precisamos criar contextos novos em função dos eventos recebidos, sem saber a priori quantas trilhas serão necessárias. Por exemplo, em um servidor HTTP poderíamos utilizar o `spawn` para criar um novo contexto para cada nova

PUC-Rio - Certificação Digital Nº 1913163/CA

conexão. Conforme as conexões terminassem, os contextos correspondentes terminariam e seus recursos voltariam ao `pool` para serem reutilizados por novas conexões ou liberados da memória automaticamente.

Código 3: *Pool* de LEDs

```
code/await Blink(var int led) -> none do
  loop do
    led_off(led);
    await 500ms;
    led_on(led);
    await 500ms;
  end
end
pool[10] Blink blinkers;
var int i;
loop i in [1 -> 10] do
  spawn Blink(i) in blinkers;
end
```

PUC-Rio - Certificação Digital Nº 1913163/CA

2.3.2 ReactiveX

O ReactiveX (ou Reactive Extensions) (REACTIVEX,), é um modelo de programação desenvolvido pela Microsoft que estende o padrão observador e define uma série de interfaces que permitem uma forma de programar com o foco no fluxo de dados. Nesse modelo, descrevemos uma cadeia de operações que devem ser aplicadas de forma assíncrona aos eventos ou dados recebidos de objetos observáveis. Assim, para cada observável, definimos as operações em uma etapa de inicialização e a cada vez que um evento for gerado essas operações serão aplicadas sobre ele. Em geral, deseja-se que essas operações não tenham efeitos colaterais para que cadeias diferentes não interajam (a menos que isso esteja explicitamente descrito) e todas as execuções de uma mesma cadeia tenham o mesmo comportamento. Dessa forma, podemos concentrar todas as transformações de uma cadeia em um só lugar e fornecer um novo observável mais adequado ao resto da aplicação, potencialmente melhorando a legibilidade do código. Além disso, se as operações puderem ser aplicadas de forma independente a múltiplos eventos, em processadores com múltiplos núcleos esses eventos podem ser tratados em paralelo sem que a aplicação precise coordenar esse paralelismo. Assim, é possível melhor utilizar os recursos de hardware sem custo para o programador.

As cadeias são formadas por um objeto observável seguido de uma sequência de operadores. Operadores são observadores que aplicam uma determinada transformação no dado e retornam o resultado como um novo ob-

servável. A especificação do ReactiveX define muitos operadores para executar operações comuns, como `map`, `filter` e `average`. Como são ao mesmo tempo observadores e observáveis, podem ser encadeados para formar uma sequência de transformações a serem aplicadas nos dados recebidos do primeiro observável. Por essa razão, a aplicação pode observar qualquer ponto dessa cadeia, ou mesmo bifurcá-la.

Em sua maioria, os operadores agem como funções puras, fazendo com que a simples observação dos operadores aplicados permita entender o código, sem ter que considerar efeitos de outras partes da aplicação. Alguns operadores recebem dados de mais de um observável, permitindo combinar dados de diferentes fontes ou interromper o fluxo de dados para os observadores. Dessa forma é possível minimizar o estado armazenado na lógica de aplicação, ficando este oculto nos operadores e observáveis.

O ReactiveX apresenta características semelhantes àquelas vistas no modelo *functional reactive programming* (FRP) (ELLIOTT; HUDAK, 1997). Nesse modelo valores são armazenados em variáveis de tipos especiais tais que expressões construídas com essas variáveis sejam reavaliadas quando o valor de alguma delas mudar. Da mesma forma, funções podem receber e retornar valores com esses tipos e são automaticamente executadas quando algum dos seus parâmetros for alterado. Esses tipos especiais se dividem em duas categorias: *Behaviors*, que representam valores que variam continuamente no tempo e *events*, que representam valores que variam de forma discreta no tempo. Os operadores em ReactiveX recalculam seus valores de saída da mesma forma que as funções do FRP quando suas entradas mudam. Assim, o modelo ReactiveX pode ser classificado como "primo" do modelo FRP (BAINOMUGISHA et al., 2013).

A figura 2.1 e o código 4 apresentam um exemplo no qual desenhamos um quadrado interpolando as cores de seus vértices implementado utilizando a *engine* de jogos Löve 2D (LÖVE2D,) e a biblioteca de ReactiveX RxLua (RXLUA,). Nesse exemplo, ao passar o mouse sobre o quadrado, um texto contendo as componentes R, G e B da cor do pixel sob o mouse é mostrado abaixo dele. Na linha 2, instanciamos um `Subject` (tipo de observável) que emite as coordenadas do mouse quando este se movimenta, alimentado pela *callback* `love.mousemoved` do Löve. Na função `love.load`, executada ao iniciar o programa, criamos uma cadeia de operadores para atuar sobre os itens emitidos por esse observável e gerar uma *string* com a cor do pixel sob o cursor. Para tanto, utilizamos o operador `filter` para descartar movimentos do mouse fora do quadrado. Em seguida, mapeamos as coordenadas do mouse para a cor correspondente e a transformamos em uma *string*. Com o método

PUC-Rio - Certificação Digital Nº 1913163/CA

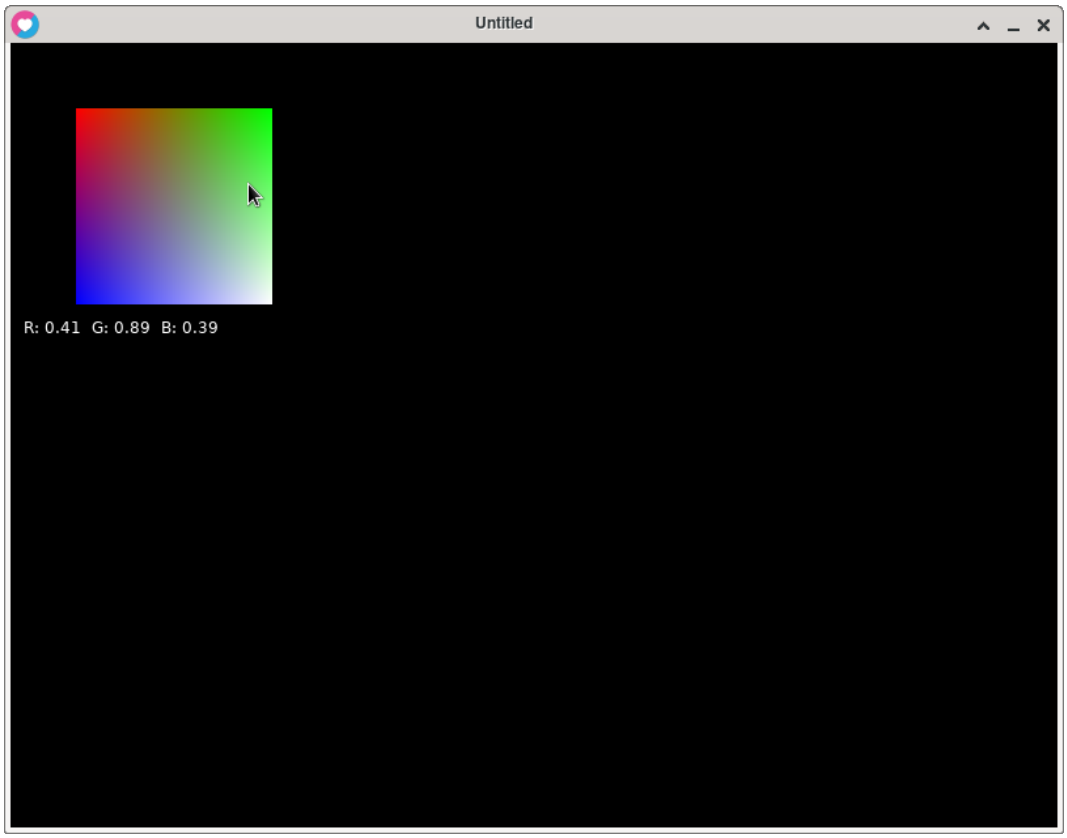

Figura 2.1: Exemplo de obter cor

`subscribe`, inserimos um observador que atualiza o texto na tela.

Uma característica marcante desse modelo é a utilização de operadores predefinidos. Por implementarem operações comuns, o uso desses operadores economiza código e permite expressar transformações de um dado ou evento de forma sucinta. Como esses operadores são padronizados pela especificação do ReactiveX, programadores que conheçam essa biblioteca sabem o que esperar de cada passo em uma cadeia. Apesar disso, o nome dos operadores e quais operadores estão disponíveis pode variar entre linguagens.

Código 4: Exemplo de obter cor

```
1  local rx = require'rx'
2  love.mousemoved = rx.Subject.create()
3  local colors = {{{1, 0, 0}, {0, 1, 0}}, -- Cores dos Vértices
4                  {{0, 0, 1}, {1, 1, 1}}}
5  local dimX, dimY = 150, 150 -- Tamanho do desenho
6  local posX, posY = 50, 50   -- Posição do desenho na janela
7  local imgData = love.image.newImageData(dimX,dimY) -- Cores dos pixeis
8  local img -- Imagem contendo imgData
9  local text = '' -- Texto mostrado abaixo do desenho
10
11 -- Retorna a cor que um pixel deve ter, interpolando linearmente a cor dos vértices
12 function getColor(x, y)
```

PUC-Rio - Certificação Digital Nº 1913163/CA

```
13   x = x / (dimX - 1)
14   y = y / (dimY - 1)
15   local out = {0, 0, 0}
16   for i = 1, 3 do
17     out[i] = (colors[1][1][i] * (1 - x) + colors[1][2][i] * x) * (1 - y) +
18              (colors[2][1][i] * (1 - x) + colors[2][2][i] * x) * y
19   end
20   return out
21 end
22
23 -- Desenha o quadrado em no buffer img
24 function drawSquare()
25   for y = 0, dimY - 1 do
26     for x = 0, dimX - 1 do
27       local r, g, b = unpack(getColor(x, y))
28       imgData:setPixel(x, y, r, g, b, 1)
29     end
30   end
31   img = love.graphics.newImage(imgData)
32 end
33
34 -- Callback de inicialização
35 function love.load()
36   drawSquare()
37   love.mousemoved -- A cada vez que o mouse é movido
38     :filter(function(x, y) -- Apenas considera movimentos dentro do quadrado
39       return x >= posX and x < posX + dimX and
40              y >= posY and y < posY + dimY
41     end)
42     :map(function(x, y) -- Converte a posição do mouse em uma string com a cor do pixel
43       local color = getColor(x - posX, y - posY)
44       return string.format('R: %0.2f  G: %0.2f  B: %0.2f', unpack(color))
45     end)
46     :subscribe(function(s) text = s end) -- Mostra a string gerada
47 end
48
49 -- Callback de desenho
50 function love.draw()
51   love.graphics.draw(img, posX, posY)
52   love.graphics.print(text, 10, posY + dimY + 10)
53 end
```

Observáveis têm 3 estados possíveis, aos quais daremos nomes para maior clareza no texto: **Ativo**, em que dados serão fornecidos para os observadores, **Erro**, quando uma função de erro será executada e o fluxo de dados é interrompido e **Completo** quando o fluxo de dados termina. O observável notifica o observador ao mudar de estado, propagando a mudança de estado pela cadeia de operadores. Observáveis começam no estado **Ativo**. Após mudarem para um dos outros estados, não mais mudam de estado. Isto é,

PUC-Rio - Certificação Digital Nº 1913163/CA

não é possível reiniciar um observável depois dele completar e tampouco é possível para um observável se recuperar de um erro.

Os observáveis se dividem em duas categorias: *Hot* e *cold*. Observáveis *hot* podem emitir dados independentemente de ter algum observador inscrito. Observáveis *cold* garantem que todos os observadores receberão todos os dados emitidos. Observadores *hot* podem, por exemplo, ser utilizados para capturar movimentos do mouse. Como não se tem controle sobre como o usuário irá mover o mouse, não é razoável garantir que um observador que se inscreva no futuro veja todos os movimentos já feitos. Observadores *cold* são utilizados, por exemplo, para iterar sobre estruturas de dados. Para cada novo observador podemos criar um novo iterador para percorrer os dados, garantindo que o novo observador veja todos os dados independentemente dos demais observadores.

Algumas implementações suportam executar partes das cadeias em *threads* diferentes (tanto cooperativas como preemptivas), utilizando `schedulers`. Em ReactiveX, `schedulers` podem representar *threads* específicas, *thread pools*, corrotinas ou outros contextos. Utilizando o operador `subscribeOn`, podemos indicar em qual `scheduler` a cadeia deve iniciar. Com o operador `observeOn`, indicamos uma troca de `scheduler` a partir desse ponto na cadeia.

Alguns *schedulers* e operadores introduzem comportamentos dependentes de tempo nas cadeias. O construtor `Observable.Interval` utiliza um *scheduler* para criar um observável que emite números inteiros em ordem crescente periodicamente. Esse observável é equivalente a um temporizador, visto que os observadores da cadeia serão chamados periodicamente. Como no caso das *callbacks*, não podemos bloquear uma cadeia pois as reações são síncronas. No entanto, o operador `delay` oferece uma funcionalidade semelhante ao bloqueio temporário de uma *thread*, atrasando a propagação de eventos por um determinado período de tempo. O ReactiveX também especifica operadores que têm comportamentos temporais de mais alto nível, como `debounce`, que só propaga um evento se não for emitido outro dentro de um determinado período de tempo. Outro exemplo é o operador `sample`, que divide o tempo em intervalos iguais e só propaga o evento mais recente dentro de cada intervalo.

Independentemente do uso de *threads*, cada observador enxerga a cadeia toda de forma independente. Para tanto, é como se fosse criada uma cópia de toda a cadeia de operadores e feita uma nova inscrição no observável para cada observador que observa o final da cadeia. Essa característica evita que haja interferência entre observadores de uma mesma cadeia, especialmente em operadores que mantêm um contexto interno, como o `average`. Em contrapartida, esse isolamento faz com que seja preciso processar toda a cadeia indivi-

PUC-Rio - Certificação Digital Nº 1913163/CA

dualmente para cada observador. Isso pode levar desnecessariamente ao reprocessamento dos eventos. Esse comportamento pode ser evitado utilizando-se o operador `share`, que faz com que os observadores seguintes a ele compartilhem a cadeia que o antecede. Dessa forma não há reprocessamento dos operadores anteriores na cadeia nem mais de uma inscrição no observável na ponta dela. No caso de cadeias iniciadas em observáveis do tipo *cold*, o reprocessamento da cadeia pode ser benéfico. Para preservar a propriedade desses observáveis de que todos os seus observadores receberão todos os eventos emitidos, impedir o reprocessamento implicaria em armazenar todos os eventos já processados. Como podemos inserir observadores em qualquer ponto da cadeia, esse comportamento teria de ser replicado para cada operador, provavelmente gerando um grande custo de memória. Vale notar que o operador `share` torna a cadeia *hot* a partir dele, enquanto a maioria dos operadores repete o comportamento do anterior.

Em geral, os operadores não se inscrevem no observável sobre o qual atuarão quando são instanciados, apenas o fazendo quando algum observador se inscreve neles. Essa característica é importante não só por evitar processamento desnecessário no caso de ninguém observar o operador, mas principalmente para evitar que observáveis *cold* comecem a gerar eventos sem que haja um observador para recebe-los. Considere o exemplo

```
1 rx.Observable.fromTable(t):map(f):subscribe(print)
```

onde o construtor `fromTable` retorna um observável *cold* que itera sobre a tabela `t`. Por ser do tipo *cold*, este começa a iterar quando algum observador se inscreve e itera sobre toda a tabela imediatamente. Se operadores se inscrevessem ao serem instanciados, o `map` se inscreveria antes que o `subscribe` executasse e consumiria todos o eventos. Consequentemente, o `subscribe` não observaria nenhum evento e nada seria impresso no terminal. O mesmo ocorre quando um observador remove a inscrição do operador: O operador também remove a sua inscrição do observável anterior para deixar de executar e indiretamente sinalizar ao resto da cadeia para que pare de enviar eventos.

Pelo fato dos operadores só se inscreverem no observável anterior na cadeia quando alguém os observa e por criarem uma inscrição para cada um de seus observadores, podemos criar cadeias predefinidas para utilizar em outros pontos da aplicação. Dessa forma, podemos definir uma sequência de operações em algum ponto do programa que exponha os dados de uma forma mais conveniente como um novo observável. Podemos então utilizar o

PUC-Rio - Certificação Digital Nº 1913163/CA

novo observável sem que outros módulos precisem conhecer as transformações aplicadas, melhorando o isolamento entre eles.

Esse modelo funciona bem para transformar *streams* em que não se sabe a priori a quantidade de dados a ser processada e deseja-se aplicar um tratamento uniforme. Todavia, controle de fluxo e cadeias que dependem de múltiplos observáveis implicam em construções um pouco mais complexas. Operadores que atuam sobre mais de uma cadeia permitem criar controle de fluxo e atuar em função de uma combinação de eventos. Por exemplo, o operador `concat` propaga os eventos que forem emitidos pelo seu primeiro observável de entrada até que esse complete. Em seguida, se inscreve no próximo observável e propaga todos os eventos emitidos por ele até que complete, e assim sucessivamente, completando quando o último observável completar. No código 5 utilizamos esse operador para concatenar os vetores "a" e "b".

Código 5: Concatenando vetores

```
1 local rx = require'rx'
2 a = {1, 2, 3}
3 b = {4, 5, 6}
4 oa = rx.Observable.fromTable(a, ipairs)
5 ob = rx.Observable.fromTable(b, ipairs)
6 oa:concat(ob):subscribe(print)
```

Outros operadores permitem completar a cadeia em função de outro observável. Por exemplo, o `takeUntil` completa a cadeia ao receber um evento do observável passado como parâmetro. Assim, podemos utilizar esses operadores para implementar controle de fluxo em uma cadeia.

PUC-Rio - Certificação Digital Nº 1913163/CA

3
Módulo Lua Tasks

Neste capítulo apresentaremos o módulo **Lua Tasks**[1], utilizado neste trabalho para implementar aplicações reativas no modelo de blocos concorrentes em Lua. Dedicamos sub-seções para explicar cada funcionalidade implementada no módulo e para apresentar as suas APIs.

Implementamos o módulo **Lua Tasks** para auxiliar no desenvolvimento de aplicações concorrentes e reativas, fornecendo APIs para descrever blocos concorrentes, como em Céu, *callbacks*, eventos, corrotinas e futuros. O módulo também implementa temporizadores virtuais, facilitando descrever reações em função do tempo. Apesar do módulo implementar todas essas interfaces, neste trabalho vamos nos concentrar apenas nos blocos concorrentes.

Pensamos o módulo para ser integrado com um escalonador externo, que fica encarregado de informar regularmente sobre a passagem do tempo e enviar eventos para que a aplicação possa reagir apropriadamente. Todas as APIs implementadas interagem com as demais, dessa forma cada parte da aplicação pode utilizar aquela que for mais apropriada para enviar e reagir a eventos. As sessões seguintes descrevem os principais mecanismos definidos no módulo **Lua Tasks**.

PUC-Rio - Certificação Digital Nº 1913163/CA

3.1
Tarefas

Tarefas são uma abstração implementada sobre corrotinas para permitir a execução de código de forma concorrente. Elas permitem parar, retomar e encerrar a execução de um trecho de código em função de eventos. Podem ser aninhadas de forma hierárquica, de forma que o término da tarefa mais externa encerra a execução das tarefas mais internas, denominadas sub-tarefas.

3.2
Eventos

Os eventos formam a base da comunicação entre tarefas e do controle da execução das mesmas. Eles permitem o envio de mensagens para um conjunto de tarefas e *callbacks*, podendo ser utilizados em conjunto com as tarefas ou de forma independente. Eventos permitem a troca de mensagens entre tarefas bem como entre tarefas e outras partes da aplicação. Em particular, podem ser utilizados para notificar tarefas de eventos externos, como interação com o

[1]Código disponível em <https://github.com/naves-thiago/lua_tasks>

usuário e passagem de tempo. Uma tarefa pode bloquear aguardando o envio de um evento utilizando a função `await()` e enviar eventos com a função `emit()`, sendo esse o principal mecanismo para cooperação entre elas.

3.3 Blocos concorrentes

Os blocos concorrentes foram inspirados pelos blocos `par/and` e `par/or` de Céu e permitem descrever grupos de tarefas que devem executar concorrentemente até o término de todas (`par/and`) as tarefas ou de qualquer uma delas (`par/or`). Eles expressam relações entre tarefas sem explicitamente passar mensagens entre elas ou utilizar máquinas de estados. Como em Céu, os blocos podem ser aninhados para descrever comportamentos mais complexos.

Neste trabalho iremos focar apenas no aspecto de estruturação do código e troca de mensagens, sem considerar as verificações de determinismo presentes em Céu. Tao pouco iremos garantir que não haja compartilhamento de memória entre blocos concorrentes.

Cada bloco concorrente é implementado como uma tarefa que inicia uma sub-tarefa para cada trilha de execução. Dessa forma, o término do bloco implica no término de todas as trilhas contidas nele. Como qualquer tarefa, esses blocos podem ser terminados externamente e são encerrados automaticamente se a tarefa que os iniciou terminar. Como um bloco também é uma tarefa, ao aninhar blocos é indiferente para um bloco se uma trilha é uma tarefa da aplicação ou outro bloco.

3.4 Futuros

Os futuros permitem aguardar a execução de eventos e obter os dados enviados, fornecendo métodos para cancelar a execução associada e a espera pelo evento. Podem ser utilizados juntamente com tarefas ou de forma independente. Quando utilizados por uma tarefa, permitem que essa aguarde o término da execução sem bloquear imediatamente. Se uma tarefa tentar obter os dados de um futuro ainda pendente, essa será bloqueada até que o evento execute.

3.5 Implementação

Para implementar a suspensão e retomada da execução nas tarefas, utilizamos o módulo de corrotinas de Lua. Esse módulo implementa corrotinas assimétricas, assim a troca de corrotina é feita através de dois métodos: `resume`

PUC-Rio - Certificação Digital Nº 1913163/CA

e `yield`. O método `resume` suspende a corrotina atual e inicia a execução de outra corrotina ou retoma a sua execução do ponto em que foi suspensa. O método `yield` suspende a execução da corrotina que está executando e retorna retorna o controle para aquela que executou `resume`. Ambos os métodos podem receber parâmetros que serão retornados pela chamada oposta. Assim, parâmetros passados para o `resume` são retornados pelo `yield` da corrotina que passou a executar. Caso seja o início da execução desta corrotina, em vez disso a sua função é executada com esses parâmetros. Da mesma forma, parâmetros passados para o `yield` são retornados pela chamada `resume` correspondente. Se a função da corrotina retornar, o controle é retornado para a chamada `resume`, que retorna os valores retornados pela função.

O módulo Lua Tasks implementa a seguintes interfaces:

- **event__t**: Classe cujos objetos representam eventos. Quando um objeto é executado (através do meta-método `__call`), este executa todas as funções em sua lista de *listeners*. Ao inserir um *listener* pode-se escolher se este será executado sempre que esse evento executar ou apenas uma vez, na execução seguinte à sua inserção. A inserção / remoção de *listeners* durante a execução do evento não tem efeito até o término da execução de todos os *listeners*. Isso impede que um *listener* inserido a partir de outro seja executado imediatamente (mas sem garantia de que isso aconteça), causando comportamentos inesperados, especialmente pela ordem de execução dos *listeners* ser indeterminada.
- **task__t**: Classe que define uma tarefa: Abstração implementada sobre corrotinas que permite suspender e retomar uma execução em função de reações a eventos, bem como criar uma hierarquia de tarefas. Essa hierarquia permite encerrar sub-tarefas (tarefas iniciadas a partir de outra) automaticamente junto com a tarefa que as iniciou. O encerramento de sub-tarefas permite a implementação de uma API semelhante aos blocos `par/and` e `par/or` de Céu.

  Diferentemente das corrotinas, quando uma tarefa inicia a execução de uma sub-tarefa, esta se comporta (por padrão) como uma chamada de função bloqueante. Dessa forma, a tarefa mais externa só retoma a sua execução quando a mais interna terminar, mesmo se a sub-tarefa bloquear esperando um evento ou futuro (internamente implementados com um `yield`).

  Cada tarefa executa um evento (membro `done`) ao terminar. Esse evento permite encerrar as sub-tarefas quando a tarefa termina (sub-tarefas

PUC-Rio - Certificação Digital Nº 1913163/CA

registram um *listener* de suicídio ao iniciar). Também pode ser utilizado pela aplicação para detectar o fim de uma tarefa.

- **future_t**: Objetos dessa classe representam processos que estejam ocorrendo em paralelo ou concorrentemente. Permitem que uma tarefa consulte o estado, aborte ou obtenha o resultado do processo. Antes do término de tal processo, se uma tarefa tentar obter seu resultado (utilizando o método `get`), esta será bloqueada até que o resultado esteja disponível.
- **timer_t**: Classe responsável por criar temporizadores que permitem executar *callbacks* em função da passagem do tempo. Cada objeto armazena o *timestamp* da sua próxima execução. Uma *heap* mantém referências para os temporizadores ativos para permitir determinar eficientemente quais temporizadores devem ser executados ao atualizar o relógio global. Todos os temporizadores seguem um único relógio interno ao módulo que deve ser atualizado pelo escalonador.
- **par_and**: Equivalente ao bloco `par/and` de Céu, essa função instancia uma tarefa que ao ser executada inicia as tarefas passadas como parâmetro como sub-tarefas concorrentes. Essa tarefa bloqueia até que todas as sub-tarefas terminem. Os parâmetros podem ser funções em vez de tarefas. Nesse caso, é criada uma sub-tarefa para cada função passada.
- **par_or**: Equivalente ao bloco `par/or` de Céu, essa função instancia uma tarefa que, ao ser executada, inicia as tarefas passadas como parâmetro como sub-tarefas concorrentes. Essa tarefa bloqueia até que qualquer uma das sub-tarefas termine, propagando os valores retornados pela mesma e encerrando as demais.
- **await**: Bloqueia a tarefa até que o evento indicado execute, retornando os dados passados para ele. Se necessário, instancia um objeto `event_t` na tabela global de eventos (`_scheduler.waiting`). O *listener* é removido do evento após a sua execução.
- **emit**: Executa o evento indicado, passando os demais parâmetros como dados do evento. Se o evento não existir na tabela `_scheduler.waiting`, não faz nada. Após executar o evento, verifica se o mesmo ainda possui *listeners*. Caso não tenha, destrói o objeto.
- **await_ms**: Bloqueia a tarefa por um determinado número de milissegundos. Cria um objeto `timer_t` que será escalonado e ao executar retomará a execução da tarefa.
- **in_ms**: Instancia um objeto `timer_t` que executará uma *callback* após o intervalo definido, iniciando a contagem imediatamente. O objeto

PUC-Rio - Certificação Digital Nº 1913163/CA

retornado pode ser usado para cancelar a contagem antes da execução da *callback*.

- **every_ms**: Semelhante à função `in_ms`, mas executa a *callback* periodicamente a cada intervalo, em vez de apenas uma vez.
- **update_time**: Atualiza o relógio global. Verifica a *heap* de temporizadores e executa aqueles cujo *timestamp* for menor ou igual ao novo.
- **now_ms**: Retorna o *timestamp* atual.
- **listen**: Registra uma *callback* como *listener* do evento identificado. O parâmetro, `once`, indica se a *callback* deve executar apenas na próxima vez que o evento executar (sendo removida automaticamente da lista de *listeners*) ou todas as vezes.
- **stop_listening**: Remove a *callback* da lista de *listeners* do evento. Se o evento não existir ou a *callback* não estiver em sua lista de *listeners*, não faz nada.
- **Tratamento de erro**: Para auxiliar no *debug* das aplicações, propagamos o erro Lua para fora das corrotinas. Estendemos a mensagem de erro para incluir a sequência de tarefas que participaram da reação, além da pilha de execução da tarefa que gerou o erro. Dessa forma, é possível observar uma sequência de eventos que passa por múltiplas tarefas como uma única pilha de execução.

PUC-Rio - Certificação Digital Nº 1913163/CA

4
ReactiveX e blocos concorrentes: comparando e combinando os modelos

Nesse capítulo iremos apresentar as principais características da programação utilizando *callbacks*, blocos concorrentes e o ReactiveX, focando principalmente nos dois últimos. Utilizaremos exemplos para mostrar a implementação do mesmo programa nos diferentes modelos. Concluiremos o capítulo discutindo cenários em que o uso de cada modelo é mais vantajoso.

4.1
Manutenção de estado

Uma das principais diferenças na forma de programar entre os modelos é a manutenção de estado quando há interação entre dois ou mais eventos. Em um modelo em que temos apenas *callbacks* representando eventos para reagir, precisamos manter o estado do programa em variáveis globais ou em um contexto compartilhado para que possa ser modificado pelas *callbacks*.

As linguagens síncronas reativas fornecem estruturas para expressar a relação entre eventos sem necessariamente utilizar variáveis para representar estado e *ifs* para determinar como agir ao receber um novo evento. Para tanto, utilizam múltiplas *trilhas* de execução que podem ser suspensas enquanto aguardam a ocorrência de eventos e que mantêm o seu estado na pilha de execução, minimizando o controle manual do mesmo. Além disso, relações entre as trilhas são expressadas por estruturas da linguagem, permitindo aguardar múltiplas trilhas completarem ou abortar a execução de uma ou mais trilhas em função de outras.

O modelo adotado pelo ReactiveX também é baseado em *callbacks*. Contudo, o uso de operadores permite ocultar ao menos parte da representação do estado e fornecer *callbacks* que representem eventos de mais alto nível. Além disso, operadores que atuam sobre mais de um observável permitem expressar relações entre eles fornecendo uma *callback* que é executada em função da lógica da aplicação, não mais somente em função de um evento externo. Esse modelo compartilha características tanto com a programação puramente em *callbacks* quanto com as linguagens síncronas reativas.

Como no padrão observador, inserimos um observador na forma de *callback* em um objeto observável para reagir a um evento. Essas *callbacks* têm livre acesso ao contexto global da aplicação, podendo inclusive gerar novos

PUC-Rio - Certificação Digital Nº 1913163/CA

eventos em outras cadeias. Como as reações são síncronas, as *callbacks* não podem bloquear, pois impediriam todo o programa de executar.

Assim como Céu, o ReactiveX oferece mecanismos para lidar com combinações de eventos sem a necessidade de gerenciar manualmente as mudanças de estado correspondentes. Operadores que atuam sobre mais de um observável ocultam os controles necessários para aguardar múltiplos eventos e combinar dados de diferentes fontes assíncronas. Por exemplo, o operador `takeUntil` completa uma cadeia ao receber um evento de outra, permitindo implementar lógicas de encerramento semelhantes a um bloco `par/or` de Céu encerrando uma *trilha* quando outra termina.

4.2 Estado e níveis de abstração

Blocos concorrentes e ReactiveX apresentam níveis de abstração diferentes. O uso dos blocos concorrentes impacta no controle de fluxo e, pontualmente, na representação do estado. Esse modelo complementa as estruturas de controle de fluxo da linguagem ao introduzir uma forma simples de tratamento de eventos com chamadas síncronas. O ReactiveX também apresenta abstrações para o controle de fluxo mas, diferente dos blocos concorrentes, apresenta uma alternativa às estruturas imperativas da linguagem. Essa mudança implica em uma representação de estado substancialmente diferente. O uso de operadores para processar os eventos faz com que uma maior parte do estado da aplicação fique oculta dentro das instâncias dos operadores que precisam manter um contexto entre eventos ou que armazenem referências para outras cadeias.

No ReactiveX, o uso de operadores para processar eventos e dados associados permite ao programador focar no processamento de dados de forma separada do fluxo de execução. Com funcionalidades comuns já implementadas em operadores, é possível descrever de forma sucinta as etapas de processamento que devem ser aplicadas ao eventos. O modelo de blocos concorrentes não fornece uma funcionalidade equivalente para processamento de dados. Aplicações nesse modelo seguem a mesma linha de implementação de uma aplicação procedural típica. Mesmo utilizando bibliotecas que forneçam algoritmos equivalentes, a aplicação continua responsável por receber os eventos e executar as chamadas apropriadas. Além disso, pode haver diferenças e incompatibilidades nas interfaces dessas bibliotecas, ficando a cargo da aplicação adaptá-las. Assim, observamos que, pelo menos para os casos comuns, o ReactiveX oferece interfaces de mais alto nível do que os blocos concorrentes.

Outra característica comum aos modelos é o uso de abstrações para

PUC-Rio - Certificação Digital Nº 1913163/CA

controlar o encerramento de tratadores de eventos. Nos blocos concorrentes, o uso de uma chamada `par_or` permite que o término de uma tarefa faça com que outras tarefas sejam encerradas. Nessa situação, um conjunto de tarefas está bloqueado a espera de eventos enquanto uma tarefa do grupo executa. O término da tarefa em execução causa o encerramento das demais. A mecânica equivalente no ReactiveX é mudar uma cadeia para o estado "completo" em função de um evento emitido em outra cadeia. O operador `takeUntil`, aplicado em uma cadeia **A**, recebe como parâmetro uma cadeia **B**. Esse operador faz com que a cadeia **A** passe para o estado "completo" quando um evento for emitido em **B**. Outros operadores também podem completar uma cadeia. Por exemplo, o `takeWhile` executa uma função recebida por parâmetro a cada evento e completa a cadeia caso a função retorne `false`.

4.3 Estudo de casos

4.3.1 Utilizando os modelos em Lua

Os módulos Lua RxLua e Lua Tasks implementam os modelos ReactiveX e blocos concorrentes respectivamente. Ambos os módulos foram implementados de forma a fazer parte de uma aplicação maior. Dessa forma, não há um *loop* principal dentro deles que controle a execução. Assim, a aplicação deve prover esse *loop*, bem como fazer chamadas aos módulos para sinalizar a ocorrência de eventos externos. As reações dos módulos têm início nessas chamadas, executando de forma síncrona até que o processamento do evento termine.

Utilizamos a *engine* de jogos Löve 2D para implementar os exemplos apresentados nesse trabalho. Aplicações que utilizam essa *engine* não implementam um *loop* principal. Em vez disso, devem implementar *callbacks* que tratam o início do programa, indicam a passagem do tempo, interações de mouse e teclado, entre outras. Essas *callbacks* devem ser registradas na tabela global `love` como o valor de chaves predefinidas. Assim, `love.load` é o ponto de entrada do programa e tipicamente contém inicializações. Operações gráficas devem ser implementadas em `love.draw`, que é executada a cada vez que um quadro é renderizado. `love.update` é chamada periodicamente e recebe por parâmetro o intervalo de tempo decorrido desde a sua última execução. Para utilizar os módulos nesse contexto, utilizamos essas *callbacks* para fazer chamadas de disparo de eventos. Em `love.load`, podemos criar tarefas do Lua Tasks ou cadeias do RxLua que vão executar as reações aos eventos gerados nas *callbacks* de mouse, teclado e tempo.

PUC-Rio - Certificação Digital Nº 1913163/CA

Utilizar ambos os modelos de forma independente em uma mesma aplicação é simples, uma vez que basta que as *callbacks* do Löve disparem eventos em ambos os módulos. Da mesma forma, podemos construir cadeias e tarefas livremente em `love.load`. Em ambos os casos a reação ao evento é síncrona, portanto, mesmo que ambos os módulos reajam a um mesmo evento, eles não executarão simultaneamente.

4.3.2 Reação a múltiplos eventos

Eventos no mundo acontecem de forma contínua e independente da aplicação. Para reagir a esses eventos, precisamos amostrá-los em intervalos de tempo discretos, seja em nível de hardware, como tratadores de interrupção, seja na aplicação, via *polling*. Para que a aplicação possa agir de forma determinística, a reação aos eventos deve acontecer na ordem em que ocorrem. Além disso, como dois eventos podem ocorrer dentro de um mesmo intervalo de amostragem, também é necessário que haja ordem de tratamento definida para os diferentes tipos de evento.

Seguindo a hipótese de sincronismo, podemos pensar nos eventos como executando em um intervalo de tempo desprezível quando comparado ao intervalo de tempo entre eventos. Dessa forma, também não há tratamento simultâneo de eventos. Ou seja, cada reação é atômica, uma vez que executa sem interrupções até terminar. Essa hipótese é válida para a maioria das situações, entretanto, algumas reações podem desencadear processamentos longos, como processamento de imagem ou criptografia, que não se encaixam nessa hipótese. Nessas situações, pode ser necessário dividir artificialmente o processo em várias etapas ou implementá-lo em um outro modelo de programação.

Suponha um jogo em que 2 jogadores têm um botão cada e ganha aquele que apertar primeiro. Um terceiro botão inicia ou reinicia o jogo. O código 6 mostra uma implementação desse jogo utilizando RxLua e Löve 2D. A linha 1 transforma a *callback* `love.keypressed`, chamada quando uma tecla é pressionada, em um observável do tipo `subject` e envia eventos utilizando o seu meta-método `__call`. As linhas 2 a 4 criam observáveis separados para cada tecla de interesse aplicando um filtro no `subject`. A construção com o operador `exhaustMap` (explicado mais à frente) permite recriar a cadeia na linha 7 para que o jogo possa ser reiniciado. Essa cadeia combina os observáveis dos botões "a" e "b" utilizando o operador `amb`, que propaga apenas os eventos do primeiro observável que emitir um. Assim, se o botão "a" for apertado primeiro, os eventos do botão "b" serão ignorados e vice-versa. Em seguida, o operador

PUC-Rio - Certificação Digital Nº 1913163/CA

`take` propaga apenas o primeiro evento e completa a cadeia, evitando que a mensagem de vitória seja impressa mais de uma vez. Finalmente, o operador `map` transforma o evento contendo a tecla pressionada em um novo evento contendo a *string* de vitória.

Representar processos que precisem ser reiniciados em ReactiveX requer uma abordagem indireta, visto que uma vez que um observável passa para o estado "completo" este não pode mais emitir eventos. Dessa forma, para reiniciar uma cadeia, é necessária uma nova inscrição, por exemplo, em um observável derivado de um iterador, ou mesmo instanciar um novo observável, como em uma chamada de procedimento remoto. Além disso, também é preciso que os observadores da cadeia que completou passem a observar a nova que se inicia. Para solucionar cenários desse tipo, podemos utilizar o operador `exhaustMap`, como no código 6.

Código 6: Exemplo de pressionar primeiro com RxLua

```
love.keypressed = rx.Subject.create()
input_a = love.keypressed:filter(function(key) return key == 'a' end)
input_b = love.keypressed:filter(function(key) return key == 'b' end)
reset   = love.keypressed:filter(function(key) return key == 'i' end)
output  = reset:exhaustMap(function() -- Aguarda a tecla 'i'
  print('Início do jogo')
  return input_a
    :amb(input_b) -- Propaga apenas a primeira tecla apertada ('a' ou 'b')
    :take(1) -- Completa depois de receber 1 evento
    :map(function(key) -- Monta mensagem de vitória
      return key .. ' venceu'
    end)
end)
output:subscribe(print)
```

O operador `exhaustMap` recebe como parâmetro uma função que mapeia cada evento recebido para um observável. O operador então se inscreve nesse observável e propaga todos os eventos recebidos dessa inscrição até que o observável complete. Enquanto o observável retornado pela função não completa, o operador ignora os eventos emitidos pelo observável sobre o qual foi aplicado. Considere o código

```
a:exhaustMap(function() return b end):subscribe(print)
```

em que **b** é um observável tipo *cold*. Quando o observável **a** emite um evento, a função é executada e retorna o observável **b**. O `exhaustMap` se inscreve em **b** e emite seus eventos para **print**. Novos eventos emitidos por **a** são ignorados

PUC-Rio - Certificação Digital Nº 1913163/CA

(e a função não é executada) até que **b** complete. A cadeia toda completa apenas quando **a** e **b** completarem. Com essa construção, do ponto de vista dos observadores após o operador `exhaustMap`, o observável **b** é reiniciado pelo observável **a**. Vale lembrar que, em observáveis do tipo *cold*, cada nova inscrição observa todos os eventos já emitidos pelo observável.

Voltando ao exemplo no código 6, pressionar a tecla "i" com o jogo parado faz com que a linha 7 execute e instancie a cadeia que observa as teclas dos jogadores. Quando uma das teclas "a" e "b" for pressionada, essa cadeia reage emitindo um evento e completando em seguida. O observador na linha 9 recebe a *string* gerada e o jogo para novamente. Como a cadeia interna completou, `exhaustMap` volta a tratar eventos de **reset**.

O mesmo jogo foi implementado no código 7 utilizando apenas *callbacks*. Como na implementação com ReactiveX, utilizamos a *callback* `love.keypressed` para receber os eventos de teclas pressionadas. Logo na primeira linha, notamos que nesse modelo precisamos guardar explicitamente o estado do jogo em uma variável. Em seguida, vemos que há uma *callback* única para todas as teclas. Nela temos que testar qual tecla foi pressionada, verificar em qual estado o programa está e atualizar o estado de acordo. Para uma melhor organização em aplicações maiores, poderíamos ter funções para tratar cada tecla e testar na *callback* do Löve qual tecla foi pressionada, chamando a função correspondente.

Código 7: Exemplo de pressionar primeiro com *callbacks*

```
1  started = false
2  function love.keypressed(key)
3    if started then
4      if key == 'a' or key == 'b' then
5        print(key .. ' venceu')
6        started = false
7      end
8    else
9      if key == 'i' then
10       print('Início do jogo')
11       started = true
12     end
13   end
14 end
```

Nesse modelo, podemos observar claramente que o programa é separado em dois estados: Jogo iniciado e jogo parado, sinalizados pela variável `started`. Também podemos observar as teclas associadas a cada estado do programa. Contudo, é preciso considerar que em um programa maior a separação entre

PUC-Rio - Certificação Digital Nº 1913163/CA

os estados e os eventos pertinentes a cada um deles provavelmente não serão tão claros, especialmente quando houver múltiplas *callbacks* envolvidas.

No código 8 vemos a implementação do jogo com blocos concorrentes. Esse exemplo foi implementado utilizando o módulo Lua Tasks, em conjunto com o Löve 2D, e segue a mesma estrutura que teria uma implementação em Céu. O programa inicia utilizando a função `tasks.emit` para emitir eventos quando a *callback* `love.keypressed` executar. A função `love.keypressed` recebe como primeiro parâmetro uma *string* representando a tecla pressionada e o primeiro parâmetro da função `tasks.emit` é o identificador do evento. Assim, fazendo `love.keypressed = tasks.emit`, transformamos cada tecla em um evento separado. Em seguida, criamos uma nova tarefa, que é uma trilha de execução que pode esperar eventos. Cada iteração do *while* é uma partida inteira do jogo. A chamada `tasks.await('i')` bloqueia a tarefa até que a tecla "i" seja pressionada. Em seguida, o programa imprime a mensagem de início e cria duas novas tarefas concorrentes utilizando a chamada `tasks.par_or`. Como em um bloco `par/or` de Céu, as duas novas tarefas são iniciadas concorrentemente e cada uma aguardará a tecla de um dos jogadores utilizando uma chamada `tasks.await`. Quando uma dessas teclas for pressionada, a tarefa correspondente imprimirá a mensagem de vitória e a chamada `tasks.par_or` retornará, encerrando a outra tarefa no processo. O *loop* reinicia o processo todo para que uma nova partida possa ser iniciada.

Código 8: Exemplo de pressionar primeiro com blocos concorrentes

```
1  tasks = require'tasks'
2  love.keypressed = tasks.emit
3  tasks.task_t:new(function()
4    while true do
5      tasks.await('i')
6      print('Início')
7      tasks.par_or(
8        function()
9          tasks.await('a')
10         print('a venceu')
11       end,
12       function()
13         tasks.await('b')
14         print('b venceu')
15       end
16     )()
17   end
18 end)()
```

Como no caso da implementação com RxLua, o estado não é representado explicitamente na aplicação. Na construção com blocos concorrentes, o estado é representado primariamente por quais eventos estão sendo aguardados,

PUC-Rio - Certificação Digital Nº 1913163/CA

qual tarefa está executando e quais tarefas estão aguardando outras tarefas terminarem. O aninhamento de tarefas, a semântica do `par_or` e `par_and` e a reação às chamadas `await` definem como esse estado evolui em função dos estímulos que a aplicação recebe.

Nessa implementação vemos que o programa repete dois passos: Esperar a tecla "i" e esperar uma das teclas "a" ou "b". O uso da chamada `par_or` deixa claro que o programa deve esperar as teclas dos jogadores simultaneamente mas individualmente, dedicando uma tarefa para cada. Esses dois passos correspondem aos dois estados do programa que são indicados pela variável `started` na implementação com *callbacks*. No entanto, aqui esse estado é mantido na pilha de execução da tarefa mais externa e não em uma variável do programa. O controle é feito pelas trocas de contexto causadas pelas chamadas `await` e `par_or` e pelo *loop* que retorna o programa ao estado inicial.

4.3.3 Aplicação de referência

Para analisar a utilização do modelo ReactiveX em uma aplicação mais realista, adaptamos a implementação de um leitor de notícias apresentada por Ben Lesh (LESH, 2018), líder do projeto RxJS[1], na conferência JSFoo 2018. A aplicação, implementada em formato para celular, carrega uma lista de notícias por HTTP e as mostra na tela como uma sequência de cartões, cada um contendo o texto de uma notícia. A lista é atualizada automaticamente a cada 30 segundos através de uma nova requisição HTTP, que retorna a lista completa de notícias. O usuário pode atualizar manualmente arrastando a página para baixo por uma determinada distância, como é comum em aplicativos móveis. Ao arrastar a página para atualizar, um ícone é arrastado para cima da página. Se o usuário arrastar até o ponto de atualizar, o ícone permanece na tela e fica girando até a chegada da resposta da requisição HTTP. Em seguida o ícone é animado de volta para fora da tela. Se o usuário soltar antes desse ponto, o ícone também é animado para fora da tela. O código apresentado por Lesh foi escrito em JavaScript e utiliza o *framework* Angular. Implementamos a mesma lógica utilizando RxLua, Löve 2D e *mockups* para as requisições HTTP.

Essa aplicação implementa soluções para algumas situações comuns, como clicar e arrastar, animação, reações periódicas e chamada de procedimento remoto. A aplicação também demonstra a combinação dessas ferramentas, com transições de estado e retorno a estados anteriores. Ao arrastar a página para atualizar, temos uma sequência de transições de estado: Ocioso,

[1] RxJS é uma implementação em JavaScript do modelo ReactiveX

PUC-Rio - Certificação Digital Nº 1913163/CA

arrastando, aguardando resposta do servidor e animando o ícone de carregamento, animando o ícone de volta para fora da tela e novamente ocioso.

Por ter todos esses elementos e ser um código escrito por um desenvolvedor experiente no uso do ReactiveX, essa aplicação serviu de referência para a implementação dos exemplos nesse trabalho. Contudo, é um código mais complexo do que o desejável para introduzir esses conceitos ou mesmo como um exemplo de aplicação mais realista. Dessa forma, optamos por apenas comentar sobre suas características e apresentar o código da nossa versão no apêndice C. A título de comparação, também implementamos essa aplicação no modelo de blocos concorrentes, utilizando Lua Tasks e Löve 2D. Essa implementação é apresentada no apêndice D, mas não será discutida.

Vamos explorar esses elementos em maiores detalhes no jogo Pong. Nesse exemplo, podemos observar as construções de interesse da aplicação de referência em um código mais simples. Nesse jogo, o movimento dos rebatedores é semelhante ao arrastar e soltar presente no leitor de notícias. Da mesma forma, há a animação da bola que reage à passagem do tempo, assim como o ícone de carregamento o faz. O reinício de processos também é um padrão importante que está presente nas duas aplicações. Ao atualizar a lista de notícias, a aplicação deve fazer uma nova requisição para o servidor e reagir à sua resposta atualizando a tela. No Pong, reposicionamos a bola e reiniciamos o seu movimento quando um dos jogadores pontua. Assim, apesar das duas aplicações terem temas distintos, ambas exploram as mesmas ferramentas e construções presentes nos modelos estudados.

PUC-Rio - Certificação Digital Nº 1913163/CA

4.3.4 O jogo Pong - ReactiveX

Como uma aplicação mais completa, em vez do leitor de notícias vamos analisar duas implementações do clássico jogo Pong, uma utilizando RxLua e outra utilizando Lua Tasks no modelo de blocos concorrentes. Para simplificar a implementação, não há contagem de pontos, há apenas uma mensagem no terminal quando um dos jogadores pontua e apenas uma partida pode ser jogada por execução do programa. O jogo inicia automaticamente quando o programa começa e a bola é reposicionada automaticamente quando um dos jogadores marca um ponto. Os rebatedores são controlados pelas teclas “w” e “s” para o jogador da esquerda e pelas teclas de seta para cima e para baixo para o jogador da direita. A direção em que a bola é rebatida é dada em função da posição em que colidiu com o rebatedor. Tendo colidido com a metade superior do rebatedor, a bola será rebatida em direção ao topo da tela. Da mesma forma, uma colisão com a metade inferior a rebaterá em direção

à parte inferior da tela. O ângulo desse movimento depende da distância da colisão para o centro do rebatedor. Quanto mais longe do centro, mais a bola será direcionada para uma das bordas da janela. A velocidade horizontal (eixo X) da bola é constante (fora mudanças de sentido) e a velocidade vertical (eixo Y) é controlada pela colisão com os rebatedores, que a variam para mudar a direção da bola. A bola é rebatida pelas bordas superior e inferior da janela apenas trocando o sentido da velocidade no eixo vertical.

Na implementação com RxLua, começamos transformando as *callbacks* `love.keypressed`, `love.keyreleased` e `love.update` em observáveis `subject` para que possam ser utilizadas em cadeias. Nessa implementação parte do estado do jogo é mantido nas cadeias e parte em variáveis globais. Isso facilita algumas interações entre cadeias, como testes de colisão, e acessar variáveis de estado em diferentes pontos de uma mesma cadeia, como no movimento da bola.

O movimento dos batedores é controlado pela cadeia retornada por `bumperPosFactory`, função apresentada no código 9. A cadeia começa filtrando apenas as teclas de interesse. Em seguida, utilizamos a construção com o `exhaustMap` já vista para permitir reiniciar o movimento a cada vez que uma das teclas for pressionada. Quando uma das teclas de movimento é pressionada, a função do `exhaustMap` verifica a tecla para determinar o sentido do movimento e instancia uma cadeia que irá retornar a nova posição do rebatedor a cada evento de `update`. Essa cadeia mapeia o intervalo de tempo decorrido **dt** para um deslocamento em pixeis e utiliza o operador `scan` para acumular os deslocamentos e obter uma posição absoluta. A cada evento, esse operador executa uma função que recebe o valor atual do acumulador e o novo deslocamento e retorna o novo valor do acumulador. O parâmetro `currPos` define o valor inicial do acumulador. A cadeia completa quando o rebatedor encosta em uma das bordas da janela, devido ao operador `takeWhile`, ou quando o jogador solta a tecla, devido ao operador `takeUntil`.

O suporte a *closures* em Lua permite guardar parâmetros de forma privada para cada rebatedor e acessá-los nas funções chamadas por operadores como `map` e `tap`. A função do `exhaustMap` captura os parâmetros da função *factory* para ter acesso às teclas de movimento e posição inicial do rebatedor. O operador `tap`, que funciona como um observador da cadeia, utiliza a variável `currPos` para armazenar a posição do rebatedor. Esse valor é usado como posição inicial na próxima vez que o jogador apertar uma tecla de movimento e o `exhaustMap` recriar a cadeia interna. O uso de *closures* também permite que os operadores da cadeia interna utilizem as variáveis da função do `exhaustMap`. Por exemplo, no operador `takeUntil` a tecla pressionada é comparada com a

PUC-Rio - Certificação Digital Nº 1913163/CA

Código 9: Pong - ReactiveX - bumperPosFactory

```
function bumperPosFactory(keyUp, keyDown, currPos)
  return love.keypressed
    -- Filtra as teclas de interesse
    :filter(function(key) return key == keyUp or key == keyDown end)
    :exhaustMap(function(key)
      -- Define a direção do movimento (constante enquanto a
      -- tecla estiver pressionada)
      local speed = key == keyUp and -bumperSpeed or bumperSpeed
      return love.update:map(function(dt) return dt * speed end)
        -- Acumula a posição Y do rebatedor
        :scan(function(acc, new) return acc + new end, currPos)
        :takeWhile(function(pos)
          return pos >= 0 and pos <= screenHeight - bumperHeight
        end)
        -- Para ao soltar a tecla apenas se for a tecla que
        -- iniciou o movimento
        :takeUntil(love.keyreleased:filter(
            function(k) return key == k end))
        -- Guarda a posição atual para a próxima vez que o rebatedor
        -- for movido
        :tap(function(pos) currPos = pos end)
    end)
end
```

tecla em um evento `keyreleased` para concluir a cadeia apenas se a tecla que foi solta for a que iniciou o movimento.

O código 10 apresenta a implementação do movimento da bola. Como na implementação dos rebatedores, o movimento da bola é dado em função dos eventos de `update`. O operador `map` multiplica o tempo decorrido pelas velocidades em cada eixo para calcular o deslocamento da bola. Novamente, o operador `scan` acumula os deslocamentos para obter a posição absoluta da bola no tempo. Pela flexibilidade desse operador de aplicar uma função definida pela aplicação para gerar o novo valor acumulado, é possível com uma só chamada calcular a posição independentemente nos dois eixos. Uma limitação do `scan`, contudo, é não poder alterar diretamente o acumulador, o que nos obriga a concluir a cadeia e instanciar uma nova quando um dos jogadores pontua para reposicionar a bola no centro da tela. Por essa razão, a posição da bola é calculada em uma cadeia dentro de um `exhaustMap`. Quando detectamos a colisão da bola com a borda esquerda ou direita da janela, emitimos um evento no `subject didScoreS`. A cadeia que calcula a posição da bola observa esse `subject` utilizando o operador `takeUntil` para que seja encerrada quando um dos jogadores pontuar. Não é preciso reiniciar a cadeia quando a bola colide com um dos rebatedores, bastando alterar a sua velocidade e direção de movimento. Dessa forma, basta testar a colisão e alterar as variáveis de velocidade apropriadamente no final da cadeia utilizando

PUC-Rio - Certificação Digital Nº 1913163/CA

Código 10: Pong - ReactiveX - Movimento da bola

```
local ballPosS = startMovingS:exhaustMap(function()
  ballYSpeed = 0
  return updateS
    :map(function(dt) return dt * ballXSpeed, dt * ballYSpeed end)
    :scan(function(acc, newX, newY)
      -- O valor acumulado é uma tabela com 2 entradas:
      -- posições X e Y
      acc = acc or {0, 0}
      acc[1] = acc[1] + newX
      acc[2] = acc[2] + newY
      return acc
    end, {ballXInitial, ballYInitial})
    :takeUntil(didScoreS) -- Completa a cadeia quando alguém pontuar
    :takeUntil(matchEndedS) -- Para a bola quando acabar o tempo da partida
    :tap(function()
      if didColideX() then
        -- Colisão com rebatedor -> calcula nova velocidade em Y
        ballXSpeed = -ballXSpeed
        setBallDirection()
      end
      if didColideY() then
        -- Colisão com a borda
        ballYSpeed = -ballYSpeed
      end
    end)
end)
```

o operador `tap`. As variáveis de velocidade são globais, o que permite que sejam alteradas diretamente, simplificando o tratamento de colisão. Para que a bola pare quando acabar o tempo da partida, utilizamos uma outra instância do operador `takeUntil` para observar o `subject matchEndedS`, fazendo a ligação entre o código que controla o tempo e o que movimenta a bola.

Um observador da posição da bola testa a colisão com as bordas da janela para detectar pontos. Ao detectar, esse observador imprime uma mensagem no terminal e emite um evento no `subject didScoreS` para encerrar o movimento da bola. Em seguida, troca a sua velocidade Y para 0 (a velocidade em X tem módulo constante e não é alterada ao pontuar) e emite um evento no `subject startMovingS` para reiniciar o movimento. A cadeia de movimento da bola utiliza o centro da tela como posição inicial, dessa forma reiniciá-la faz com que a bola seja reposicionada no centro da tela.

Para limitar o tempo da partida, utilizamos um observável to tipo `Interval`. Esse observável emite uma sequência infinita de inteiros em intervalos regulares, começando de 0 e incrementando de 1 a cada intervalo. Esse observável utiliza um `scheduler` definido pela aplicação para agendar o envio dos eventos e para que esta possa indicar a passagem do tempo.

O código 11 apresenta a cadeia que controla o tempo de duração da

PUC-Rio - Certificação Digital Nº 1913163/CA

Código 11: Pong - ReactiveX - Contagem do tempo

```
rx.Observable.Interval(1, clockScheduler)
  :map(function(elapsed) return elapsed + 1 end)
  :takeWhile(function(elapsed)
    -- Completa a cadeia quando acabar o tempo da partida
    return elapsed < matchLen
  end)
  :startWith(0)
  :subscribe(function(elapsed) -- onNext
    -- Atualiza o relógio na tela uma vez por segundo
    local time = math.floor(matchLen - elapsed)
    local min = time / 60
    local sec = time % 60
    clockText = string.format('%02d:%02d', min, sec)
  end,
  nil, -- onError
  function() -- onCompleted
    matchEndedS()
    clockText = '00:00'
  end)
```

PUC-Rio - Certificação Digital Nº 1913163/CA

partida, atualizando o mostrador na tela (variável `clockText`) a cada segundo e emitindo um evento ao final. O mostrador conta de forma regressiva, começando no tempo definido pela variável `matchLen` e emitindo um evento no `subject matchEndedS` ao chegar em 0. A cadeia começa com o observável emitindo uma sequência de inteiros que, por serem emitidos a cada segundo, indicariam o tempo decorrido da partida. Contudo, essa sequência começa em 0 mas o primeiro evento é emitido depois de 1 segundo. Portanto, utilizamos um operador `map` para somar 1 e fazer a sequência indicar o tempo decorrido. O mostrador de tempo na tela só é atualizado a cada evento dessa cadeia, começando em branco. O operador `startWith` insere o inteiro 0 no início da sequência mas, diferente do `Interval`, ele emite o evento imediatamente. Isso faz com que haja um evento para desenhar o mostrador assim que a partida inicia. Com esses três elementos, construímos uma sequência de eventos que indica o tempo decorrido começando no tempo 0. O operador `takeWhile` completa a cadeia quando o tempo decorrido for igual à duração da partida. Um observador atualiza o mostrador a cada evento (*callback* `onNext`) e envia um evento no `subject matchEndedS` quando a cadeia completa (*callback* `onCompleted`).

4.3.5 O jogo Pong - Blocos concorrentes

A implementação com blocos concorrentes também começa criando eventos correspondentes às *callbacks* do Löve 2D. Quando uma tecla é pressionada

um evento com o nome da tecla seguido de "_down" é emitido, por exemplo "a_down" para a tecla "a". Seguindo o mesmo formato, um evento com o sufixo "_up" é emitido quando a tecla é solta. As *callbacks* `love.keypressed` e `love.keyreleased` emitem os evento correspondentes a apertar e soltar uma tecla. A *callback* `love.update` emite o evento "update" contendo o intervalo de tempo decorrido desde a última vez que esse evento foi emitido.

A lógica do jogo é dividida entre três tarefas independentes, uma para movimentar cada rebatedor e uma para movimentar a bola. A *callback* `love.load`, executada no início do programa, inicia as três tarefas.

O código 12 apresenta a tarefa que move o rebatedor esquerdo, controlado pelas teclas "w" e "s". A tarefa cria duas novas tarefas utilizando a função `par_or`. Cada uma dessas tarefas controla o movimento em uma direção, observando as teclas correspondentes. A primeira tarefa inicia esperando a tecla "w" ser pressionada com uma chamada `await`. Em seguida, utiliza uma chamada `par_or` para mover o rebatedor para cima a cada vez que o evento "update" for emitido enquanto a tecla estiver pressionada. Quando a tecla for solta, o evento "w_up" será emitido e fará com que a outra tarefa do `par_or` (linhas 13 à 16) termine e encerre o bloco todo, parando o movimento.

Código 12: Pong - Blocos concorrentes - Movimento do rebatedor

```
1  tasks.task_t:new(function()
2    while true do
3      tasks.par_or( -- Espera 'w_down' e 's_down' simultaneamente
4        function()
5          tasks.await('w_down')
6          tasks.par_or(
7            function()
8              while true do
9                local dt = tasks.await('update')
10               bumper1Y = math.max(0, bumper1Y - bumperSpeed * dt)
11             end
12           end,
13           function()
14             tasks.await('w_up')
15             -- Volta a esperar o início do movimento
16             tasks.emit('bumper1_done')
17           end
18         )(true, true) -- Inicia esse par_or sem bloquear a tarefa e não
19                       -- termina quando ela terminar
20         -- A tarefa termina imediatamente após iniciar o par_or,
21         -- terminando o par_or mais externo (mas não o interno)
22       end,
23       function()
24         tasks.await('s_down')
25         tasks.par_or(
```

PUC-Rio - Certificação Digital Nº 1913163/CA

```
26          function()
27            while true do
28              local dt = tasks.await('update')
29              bumper1Y = math.min(screenHeight - bumperHeight,
30                                  bumper1Y + bumperSpeed * dt)
31            end
32          end,
33          function()
34            tasks.await('s_up')
35            -- Volta a esperar o início do movimento
36            tasks.emit('bumper1_done')
37          end
38        )(true, true) -- Inicia esse par_or sem bloquear a tarefa e não
39                      -- termina quando ela terminar
40        -- A tarefa termina imediatamente após iniciar o par_or,
41        -- terminando o par_or mais externo (mas não o interno)
42      end
43    )()
44    -- A atualização da posição executa como uma tarefa independente
45    -- Espera ela terminar antes de permitir iniciar o movimento novamente
46    tasks.await('bumper1_done')
47  end
48 end)(true, true)
```

Enquanto a tecla de uma direção está pressionada, a tecla da direção oposta não tem efeito. Para isso, é preciso esperar o evento de pressionar de ambas as teclas ao mesmo tempo, mas apenas enquanto o rebatedor estiver parado. Uma vez que o movimento foi iniciado, apenas os eventos "update" e de soltura da tecla que iniciou o movimento devem ser considerados. Assim, a tarefa mais externa inicia uma nova tarefa para cada direção de movimento e cada tarefa dessas bloqueia esperando que a tecla correspondente seja pressionada. Quando uma das teclas for pressionada, a tarefa correspondente inicia o movimento do rebatedor em uma nova tarefa independente[2], retornada pelo `par_or` (linhas 6 e 21). Dessa forma, a tarefa que esperou a tecla ser pressionada termina imediatamente e, por ter sido iniciada por um `par_or` (linha 3), termina a tarefa que estava esperando a tecla da direção oposta ser pressionada. Dessa forma, as únicas tarefas executando são as do `par_or` que movimenta o rebatedor e espera a tecla correspondente ser solta. Uma vez terminado o movimento, devemos voltar a reagir às teclas de ambas as direções. Para isso, quando a tecla de movimento for solta, a tarefa emite o evento "bumper1_done", que desbloqueia a tarefa mais externa e reinicia todo o processo na próxima iteração do `while`.

[2]Os parâmetros `(true, true)` nas linhas 17 e 33 fazem as tarefas iniciarem sem bloquear a tarefa que as iniciou e sem terminar quando ela termina.

PUC-Rio - Certificação Digital Nº 1913163/CA

Código 13: Pong - Blocos concorrentes - Movimento da bola

```
local function update_ball_pos()
  while true do
    local dt = tasks.await('update')
    -- Calcula o deslocamento da bola a cada dt segundos
    ballX = ballX + ballXSpeed * dt
    ballY = ballY + ballYSpeed * dt
    if didColideX() then
      -- Colisão com rebatedor -> calcula nova velocidade em Y
      ballXSpeed = -ballXSpeed
      setBallDirection()
    end
    if didColideY() then
      -- Colisão com a borda superior / inferior
      ballYSpeed = -ballYSpeed
    end
    if ballX > screenWidth - ballSize or
      ballX < 0 then
      -- Colisão com a borda direita / esquerda -> ponto
      print('score')
      ballX = ballXInitial
      ballY = ballYInitial
      ballYSpeed = 0
    end
  end
end
tasks.par_or(
  update_ball_pos,
  function()
    -- Para a bola quando acabar o tempo da partida
    tasks.await_ms(matchLen * 1000)
    clockText = '00:00'
  end,
  function()
    -- Atualiza o relógio na tela uma vez por segundo
    local time = math.ceil(matchLen - tasks.now_ms() / 1000)
    clockText = string.format('%02d:%02d', time / 60, time % 60)
    while true do
      tasks.await_ms(1000)
      local time = math.ceil(matchLen - tasks.now_ms() / 1000)
      clockText = string.format('%02d:%02d', time / 60, time % 60)
    end
  end
)(true, true)
```

A posição da bola é atualizada pela tarefa no código 13. Essa tarefa é dividida em três sub-tarefas que são iniciadas por uma chamada `par_or`: A primeira movimenta a bola, a segunda testa se o tempo da partida acabou e a terceira atualiza o mostrador de tempo (variável `clockText`). Para movimentar a bola, a tarefa aguarda o evento `update` e atualiza as variáveis globais de posição em função da velocidade atual e do tempo decorrido. Em seguida, testa os três casos de colisão: Com os rebatedores, com as bordas superior e inferior e com as bordas direita e esquerda. A tarefa que atualiza o mostrador

PUC-Rio - Certificação Digital Nº 1913163/CA

de tempo fica em um *loop* que a cada iteração suspende a tarefa por um segundo e em seguida atualiza o mostrador. A tarefa que encerra a partida apenas aguarda suspensa o tempo total da partida, em seguida atualizando o mostrador e encerrando. Quando essa tarefa termina, as outras duas são encerradas por fazerem parte de um `par_or`, interrompendo o movimento da bola e a contagem do tempo da partida.

4.4 Combinando os modelos

Observando as duas implementações do pong, notamos que foi mais fácil descrever o movimento da bola usando blocos concorrentes e o movimento dos rebatedores usando RxLua. Ao implementar o movimento de cada rebatedor, precisamos considerar duas situações muito semelhantes: O movimento para cima e para baixo. Cada uma delas executa praticamente o mesmo código, mudando apenas a tecla que inicia o movimento e a direção do deslocamento. Da mesma forma, o código que movimenta os dois rebatedores contém a mesma lógica, mudando apenas as teclas observadas e a variável de posição. Por essa razão, idealmente teríamos uma única implementação para o movimento que pudesse ser parametrizada.

Na implementação com RxLua, fizemos essa implementação parametrizável. Uma função *factory* retorna um observável que captura os parâmetros (teclas a observar e posição inicial do rebatedor) e emite as atualizações da posição em função da tecla pressionada e do evento de passagem do tempo (`update`). Como todas as teclas emitem eventos em um mesmo observável, podemos utilizar o operador `filter` para observar simultaneamente as duas teclas de interesse. Como o evento contém a identificação da tecla pressionada, podemos esperar pelo evento correspondente de soltura da tecla e capturar esse valor. Assim, podemos usar esse valor em um outro filtro mais a frente na cadeia para interromper o movimento. Com exceção da posição inicial, todo o estado necessário para o movimento fica armazenado de forma privada na cadeia, simplificando o uso de uma função *factory*. Apesar da posição de cada rebatedor estar armazenada em uma variável global, as cadeias que atualizam essas posições não precisam acessar diretamente essas variáveis. Essa associação é feita adicionando-se um observador em cada cadeia para atualizar a variável correspondente.

A implementação do movimento dos rebatedores com blocos concorrentes segue uma abordagem totalmente diferente. Nessa implementação, dedicamos uma tarefa para controlar o movimento de cada rebatedor. Diferente da implementação com RxLua, cada tecla tem um evento próprio emitido quando

PUC-Rio - Certificação Digital Nº 1913163/CA

é pressionada e outro quando a tecla é solta. Isso é equivalente a ter em RxLua duas cadeias para cada tecla, uma para pressionamento e outra para soltura. Essa característica permite bloquear a tarefa até que uma das teclas de interesse seja pressionada. Para tal utilizamos uma chamada `par_or` que irá esperar qualquer uma das teclas ser pressionada, iniciar o movimento de forma assíncrona e parar de esperar a outra tecla.

Para que a lógica seguisse o formato que teria uma implementação em Céu, optamos por não passar identificadores de eventos por parâmetro ou em variáveis nem gerar novos identificadores em tempo de execução. Essa escolha impede o uso de uma função tipo *factory* para gerar uma tarefa para cada rebatedor, pois não seria possível parametrizar as teclas de movimento. Da mesma forma, é preciso utilizar duas tarefas para o movimento de cada rebatedor, cada uma reagindo à tecla de uma direção. Isso é necessário para que possamos esperar o evento correspondente à mesma tecla sendo solta. Se removermos essas restrições, podemos simplificar o código substancialmente.

O código 14 mostra a implementação de uma função *factory* que gera tarefas para o movimento de cada rebatedor com os identificadores dos eventos sendo gerados em função dos parâmetros. Nessa versão utilizamos um `par_or` para esperar qualquer uma das teclas ser pressionada e em seguida definir o evento de soltura da tecla e a direção do movimento. Com isso, unificamos os dois blocos muito semelhantes que tratavam separadamente cada tecla na implementação anterior. Além disso, o parâmetro `bumper` identifica qual dos rebatedores devemos considerar, unificando a implementação dos dois rebatedores.

Nesse exemplo, os eventos gerados ao pressionar e soltar teclas e o evento de atualização do relógio (`update`) são semanticamente externos, já que sinalizam para a aplicação de mudanças externas a ela. Contudo, a própria aplicação emite esses eventos quando as *callbacks* do Löve 2D executam. Isso oferece alguma flexibilidade na geração dos eventos. Por exemplo, para um evento de pressionamento de tecla, podemos optar por gerar eventos separados para cada tecla ou um só evento com a identificação da tecla como dado. Essa flexibilidade pode simplificar a lógica da aplicação que utiliza esses eventos. Se utilizássemos um evento para todas as teclas, teríamos uma implementação do movimento dos rebatedores muito semelhante a do código 14, mesmo mantendo a restrição de não poder usar variáveis para identificar eventos. É preciso notar, entretanto, que nem sempre a aplicação terá controle sobre o formato dos eventos, tendo que lidar diretamente com eventos externos com um formato predefinido. Essa limitação pode impor uma implementação mais complexa do que o ideal, como exemplificado no movimento dos rebatedores no código 12.

PUC-Rio - Certificação Digital Nº 1913163/CA

Esse problema pode ser mitigado utilizando-se eventos internos para adaptar o formado dos eventos externos para um mais apropriado. Poderíamos utilizar duas tarefas para transformar os eventos separados de duas teclas em um só com a tecla como dado. Cada tarefa ficaria em um *loop* esperando o evento de uma tecla e em seguida emitindo um evento comum com a identificação da tecla como dado.

Código 14: Pong - Blocos concorrentes - Movimento do rebatedor - 2

```
function bumperTaskFactory(key_up, key_down, bumper)
  tasks.task_t:new(function()
    while true do
      local up_event -- Evento de soltura da tecla
      local speed -- Velovidade e direção do movimento
      tasks.par_or( -- Aguarda uma tecla ser pressionada
        function()
          tasks.await(key_up .. '_down')
          up_event = key_up .. '_up'
          speed = -bumperSpeed
        end,
        function()
          tasks.await(key_down .. '_down')
          up_event = key_down .. '_up'
          speed = bumperSpeed
        end
      )()

      tasks.par_or( -- Movimenta o rebatedor
        function()
          tasks.await(up_event)
        end,
        function()
          while true do
            local dt = tasks.await('update')
            if bumper == 1 then
              bumper1Y = math.max(0, math.min(screenHeight - bumperHeight,
                                              bumper1Y + speed * dt))
            else
              bumper2Y = math.max(0, math.min(screenHeight - bumperHeight,
                                              bumper2Y + speed * dt))
            end
          end
        end
      )()
    end
  end)
end
```

Diferente do movimento dos rebatedores, o movimento da bola só depende da passagem do tempo. É preciso, no entanto, considerar duas escalas de tempo: Atualizações rápidas e pequenas para movimentar a bola e o tempo total da partida para pará-la. Outra diferença é que o movimento dos rebatedores é sempre em função da sua posição anterior, enquanto a bola é reposicionada no centro da janela quando um jogador pontua. Essa característica dificulta a

PUC-Rio - Certificação Digital Nº 1913163/CA

implementação com RxLua, mas não tem impacto na versão com blocos concorrentes. Assim, observamos que a implementação com blocos concorrentes é mais simples do que a que usa RxLua, situação oposta ao movimento dos rebatedores.

O movimento da bola na implementação com RxLua é controlado pelo observável `ballPosS` que, dentro de um `exhaustMap`, utiliza os eventos de passagem do tempo para calcular os deslocamentos. Um operador `scan` acumula esses deslocamentos em uma posição absoluta independentemente para os eixos X e Y. Para definir a posição da bola diretamente, por esta estar encapsulada no operador `scan`, precisamos concluir e recriar a cadeia. Utilizamos um operador `takeUntil` para concluir a cadeia quando um jogador pontua, o que faz com que a bola volte à posição inicial no centro da janela. Isso ocorre quando um evento é emitido no observável `didScoreS`. Para parar a bola ao final da partida, um segundo operador `takeUntil` encerra a cadeia quando um evento for emitido no observável `matchEndedS`. Para detectar que um jogador pontuou, um observador inscrito na cadeia `ballPosS` testa a colisão com as bordas direta e esquerda. Ao detectar a colisão, emite eventos em duas cadeias: Em `didScoreS` para parar a bola e permitir que seja reposicionada e em `startMovingS` para reposicionar e reiniciar o movimento da bola. Um segundo observador atualiza as variáveis globais de posição da bola para que possam ser usadas pela *callback* de desenho do Löve 2D. Uma segunda cadeia é responsável pela contagem do tempo da partida. Essa cadeia emite o tempo decorrido a cada um segundo e completa quando o tempo da partida termina. Um observador atualiza o mostrador na tela e emite um evento na cadeia `matchEndedS` quando a cadeia completa, parando a bola.

Na implementação com blocos concorrentes, utilizamos uma chamada `par_or` que inicia uma tarefa para controlar o movimento da bola, uma para atualizar o mostrador de tempo e uma para encerrar a partida. A tarefa que movimenta a bola só depende do evento `update`, uma vez que será encerrada pelo `par_or` quando o tempo acabar. Essa tarefa movimenta a bola, trata a colisão com os rebatedores e com as bordas da janela, inclusive detectando pontos e reposicionando a bola. O seu código é bastante simples de entender por depender de apenas um evento e por ter pouco controle de fluxo. Da mesma forma, o controle do mostrador de tempo depende apenas de passagem do tempo, ficando em um *loop* que suspende a tarefa por um segundo e atualiza o mostrador em seguida. O encerramento da partida é feito pela terceira tarefa nesse `par_or`, que apenas fica suspensa pela duração total da partida, atualiza o mostrador para “00:00” e termina, encerrando as outras duas.

PUC-Rio - Certificação Digital Nº 1913163/CA

4.4.1 Implementação híbrida

Dadas as complexidades apresentadas de cada modelo, propomos uma implementação híbrida utilizando ao mesmo tempo ambos os modelos. O movimento dos rebatedores é quase independente do movimento da bola, interagindo apenas no teste de colisão da bola com os rebatedores através de variáveis globais. Por isso, é bastante simples utilizar o código que movimenta os rebatedores da versão RxLua e o código de movimento da bola e controle de tempo da versão com blocos concorrentes em uma mesma aplicação. Vale notar que em ambas as versões o teste de colisão já era feito utilizando-se essas mesmas variáveis.

Na integração com as *callbacks* do Löve 2D, apenas o código que movimenta os rebatedores precisa receber os eventos de apertar e soltar das teclas. Dessa forma, essas *callbacks* podem ser transformadas diretamente em `Subjects` (observáveis Rx) para que gerem eventos que possam ser utilizados nas cadeias do RxLua. Por outro lado, tanto o movimento dos rebatedores quanto o da bola precisam de atualizações de tempo. Uma *callback* envia eventos em um `Subject` e no Lua Tasks para que ambos os modelos possam reagir à passagem do tempo.

Com os dois modelos na mesma aplicação obtivemos um código menor e mais simples do que usando apenas um deles. Em contrapartida, essa união requer que o programador conheça ambos os modelos para compreender o código, o que pode aumentar a barreira de entrada para colaboração. Em qualquer projeto de programação é necessário conhecer as ferramentas disponíveis para que se possa fazer o uso mais eficiente delas. Dessa forma, não vemos o aumento de complexidade gerado como uma desvantagem, apenas uma consequência de um aumento de opções.

Observamos que partes da aplicação com controle de fluxo mais complexo, principalmente quando é preciso retornar a estados anteriores, tendem a ser mais difíceis de implementar com ReactiveX. Isso é consequência da semântica dos observáveis e de alguns operadores. Um observável que passou para um estado de erro ou de "concluído" não pode mais sair desse estado. Operadores que mantêm estado interno, como o `scan`, não têm uma interface para modificar diretamente esse estado. Assim, é comum ter que destruir uma cadeia e recriá-la para mudar de estado. No modelo de blocos concorrentes, há menos estado oculto, sendo mais fácil modificá-lo. O retorno a um estado anterior muitas vezes se resume a colocar a lógica dentro de um *loop*. Por outro lado, encapsular o estado ajuda a implementar código que possar ser instanciado múltiplas vezes, como no caso do movimento dos rebatedores.

PUC-Rio - Certificação Digital Nº 1913163/CA

Para reações em que múltiplos eventos interagem ou que envolvem muitas trocas de estado, pode não ser fácil determinar qual dos modelos resultará na implementação mais simples. Com blocos concorrentes é possível que a reação fique fragmentada em vários blocos pelo aninhamento de chamadas `par_and` e `par_or`. Essa fragmentação dificulta o entendimento da sequência de operações, especialmente se houver 3 ou mais dessas chamadas aninhadas. Em ReactiveX, por outro lado, essas as interações serão representadas como uma sequencia de operadores que atuam sobre mais de um observável. Pela dificuldade de percorrer uma sequência cíclica de estados, pode ser necessário representar a reação utilizando mais de uma cadeia, gerando fragmentação da sequência de operações.

Para cenários em que é aplicado um processamento uniforme sobre uma massa de dados ou sequência de eventos, o ReactiveX pode ser o modelo mais adequado. Uma vantagem desse modelo é a disponibilidade de diversos operadores que implementam operações comuns e que podem ser encadeados para formar processamentos mais complexos. Isso permite expressar as transformações de forma compacta e evita que operações comuns precisem ser reimplementadas. No entanto, essa facilidade depende muito dos operadores existentes serem adequados à solução do problema em questão, uma vez que o desenvolvimento de novos operadores pode ser mais complexo do que implementar a reação em outro modelo. O modelo de blocos concorrentes não oferece nada equivalente a esses operadores e portanto essas funcionalidades deve ser fornecidas pela aplicação ou por outras bibliotecas.

Em ReactiveX[3], a implementação de reações que envolvam processamentos caros em cadeias que utilizem operadores como `takeUntil` e `takeWhile` pode levar a desperdício de processamento. Como esses operadores só afetam os operadores e observadores seguintes a eles, parte da cadeia pode continuar executando desnecessariamente. A solução é utilizar o objeto `Subscription`, retornado ao inserir um observador na cadeia, para remover a inscrição quando a cadeia completar. Ao remover uma inscrição, toda a cadeia é notificada e para de gerar dados para aquele observador. Contudo, essa solução não funciona em todos os cenários. Alguns observáveis, como o `fromTable`, produzem todos os seus eventos assim que um observador se inscreve, antes da chamada de inscrição retornar. Assim, só se tem acesso ao objeto `Subscription` após processar todos os eventos, impedindo o uso da solução acima. No modelo de blocos concorrentes, temos mais controle sobre o fluxo do programa e do uso de recursos. Uma tarefa pode simplesmente retornar para parar de executar. Alternati-

[3]Aqui nos referimos particularmente à implementação do RxLua v0.0.3, utilizada nesse trabalho.

PUC-Rio - Certificação Digital Nº 1913163/CA

vamente, podemos utilizar uma chamada `par_or` para definir um método de cancelamento de uma tarefa antes que ela comece a executar. De dentro da tarefa podemos emitir um evento que cause o encerramento de outra tarefa desse `par_or`, terminando a primeira. Assim, o modelo de blocos concorrentes pode ser mais adequado quando for preciso garantir que procedimentos não executarão desnecessariamente.

Blocos concorrentes também oferecem maior facilidade na reação ao tempo. Tarefas podem bloquear esperando por eventos ou por tempo apenas fazendo uma chamada de função. Dessa forma, é simples descrever sequências de ações e esperas. Com ReactiveX, como não se pode bloquear a execução de uma cadeia[4], é difícil descrever sequências que intercalem a espera por tempo com espera por eventos. O `CooperativeScheduler` permite agendar a execução de funções para um momento futuro. Esse `scheduler` executa as funções dentro de corrotinas que podem suspender por intervalos de tempo arbitrários utilizando uma chamada `yield`. Entretanto, essas execuções não fazem parte de uma cadeia e consequentemente não podem utilizar operadores. Não há uma interface para esperar um evento dentro dessas corrotinas, de forma que ficam restritas a reagir apenas ao tempo. Em algumas situações, contudo, o uso de operadores que atuam em função do tempo, como `debounce` e `delay`, pode simplificar a aplicação. Diferente do `CooperativeScheduler`, esses operadores atuam em cadeias, podendo portanto ser utilizados em conjunto com os demais operadores.

PUC-Rio - Certificação Digital Nº 1913163/CA

[4]Pode ser possível utilizando `schedulers`, dependendo da implementação. Não há suporte para suspender a execução durante uma reação em RxLua.

5
Debugging

Nesse capítulo discutiremos as dificuldades no *debug* de aplicações reativas desenvolvidas nos modelos apresentados. Em seguida discutiremos ferramentas e métodos para auxiliar na solução de tais dificuldades.

Ao tentar seguir o fluxo de execução de um programa escrito com ReactiveX utilizando um *debugger* tradicional nos deparamos com o problema de que grande parte do controle de fluxo acontece dentro da biblioteca. Essa característica dificulta a colocação de pontos de parada pois muitos dos pontos de interesse são exatamente esses pontos de controle de fluxo ocultos. Pelo mesmo motivo, não há uma forma fácil de executar cada operador em uma cadeia de forma individual, observando o resultado do seu processamento. Ao seguir a execução de uma reação utilizando um *debugger*, nos vemos obrigados a entrar e sair do código da biblioteca conforme a execução segue de um operador para outro. Isso dificulta consideravelmente o entendimento do programa, visto que agora temos de entender também a lógica interna da biblioteca.

Seguir a execução de um evento se torna ainda mais complicado quando há mais de um observador em algum ponto da cadeia. Como ocorre ao colocar um ponto de parada em uma função chamada em mais de um ponto de um programa, o programa para nesse ponto múltiplas vezes. No caso de uma função, podemos identificar facilmente a chamada de interesse, seja observando a pilha de execução, seja pelo programa ter parado em um outro ponto de parada anterior. Entretanto, o mesmo não acontece ao utilizar o ReactiveX pois tipicamente a parte do programa anterior à chamada de um operador é comum e a parte que diferencia as chamadas está no final da cadeia como observador. Dessa forma, o ponto do programa que diferencia essas chamadas ainda não executou e, consequentemente, analisar a pilha não o identifica.

Dadas essas dificuldades, a utilidade de um *debugger* tradicional é limitada. Assim, instrumentar o código para imprimir mensagens quando ocorrem eventos é não só comum, mas sugerido em livros populares (BANKEN; MEIJER; GOUSIOS, 2018). Modelos reativos necessitam de ferramentas especializadas para que o programador possa observar e compreender o comportamento do programa de forma eficiente (BANKEN; MEIJER; GOUSIOS, 2018; SALVANESCHI; MEZINI, 2016). Essas ferramentas devem mostrar o grafo de operadores das cadeias para que se tenha uma visão unificada da sequência de operadores em cada cadeia e seus observadores. É preciso notar que a aplicação pode modificar dinamicamente as cadeias, além de inserir e remover

PUC-Rio - Certificação Digital Nº 1913163/CA

observadores. Dessa forma, o grafo apresentado deve reagir a essas mudanças. Também é preciso permitir que os nós sejam inspecionados, para que se possa verificar o estado armazenado em cada operador sem a necessidade de entender o código da biblioteca.

Por se tratar de uma aplicação reativa, parar a execução pode alterar o comportamento do programa. Dessa forma, além de pontos de parada, ferramentas de *tracing* que armazenem o fluxo de execução e mudanças de estado no tempo seriam de grande ajuda.

Os *debuggers* tradicionais se mostram mais úteis no modelo de blocos concorrentes. Diferente do modelo ReactiveX, esse modelo tem um formato próximo de uma programação procedural. Apesar de parte do controle de fluxo ser feito pela biblioteca ao trocar de tarefa, a maior parte do estado continua representado em variáveis do programa. Dessa forma, podemos colocar pontos de parada nas tarefas e inspecionar ou modificar variáveis.

Devido aos eventos serem globais, no modelo de blocos concorrentes pode ser mais difícil do que em ReactiveX determinar quais partes do código são afetadas por um evento. Isso acontece pois não há uma referência ou objeto representando o evento sendo passado explicitamente para as funções que o utilizam. Com ReactiveX, todo observável é um objeto que a aplicação instanciou e precisa passar para as funções que desejarem o observar.

Casos de erro em ReactiveX são tratados como apenas mais uma *callback* em uma cadeia. Em RxLua, os operadores executam chamadas de código da aplicação dentro de chamadas protegidas, capturando erros Lua e os passando para a *callback* de erro do observador. Enquanto isso permite que a aplicação trate erros previstos, como uma falha de conexão de rede ou uma tentativa de ler um arquivo inexistente, dificulta o rastreamento de outros erros. O observador recebe apensa a mensagem de erro, sem pilha de execução nem informações de qual operador estava executando, o que não deixa claro o ponto da cadeia que originou o erro.

No modelo de blocos concorrentes pode ser difícil descobrir a origem da execução que causou um erro. Isso se dá pois cada reação pode emitir eventos que causem novas reações. Para que seja possível saber qual evento iniciou uma cadeia de execuções que terminou em erro, é preciso conhecer a sequência de eventos internos executada e quais tarefas os emitiram. Na implementação do módulo Lua Tasks, instrumentamos as tarefas para que, ao emitir um erro, a pilha de execução de todas as tarefas que participaram sejam capturadas, juntamente com a sequência de execução dessas tarefas. Toda essa informação é propagada em um novo erro para a aplicação. A pesquisa apresentada por Banken e Meijer em (BANKEN; MEIJER; GOUSIOS, 2018) indica uma

PUC-Rio - Certificação Digital Nº 1913163/CA

dificuldade em compreender implementações com ReactiveX quando o escopo de uma cadeia é maior do que apenas uma classe ou método. Acreditamos que o mesmo possa ocorrer com blocos concorrentes devido à natureza global dos eventos.

Ao combinar os modelos em uma aplicação, somamos as dificuldades observadas em ambos. Podemos combinar os modelos em reações distintas ou em uma mesma reação. No primeiro caso, a situação não é muito diferente de só ter um dos modelos, visto que uma vez identificada a reação de interesse ela estará totalmente contida em um dos modelos. A partir desse ponto, pode-se usar as ferramentas adequadas a esse modelo para o *debug*. Quando há interação entre os modelos em uma mesma reação, nenhuma das ferramentas individualmente seria adequada. Uma ferramenta que compreenda múltiplos modelos certamente seria muito mais complexa e especializada. Dada a pluralidade de ferramentas e implementações existentes, é improvável a existência de uma solução pronta que suporte a combinação dos modelos escolhidos. Nessa situação, o mais provável é ter de recorrer a um *debugger* tradicional e à impressão de mensagens para determinar o fluxo do código e inspecionar variáveis.

PUC-Rio - Certificação Digital Nº 1913163/CA

6
Conclusões e trabalhos futuros

A programação concorrente é uma área complexa em que muitos modelos propõem soluções para melhor expressar reações a eventos e minimizar erros decorrentes da concorrência. Nesse trabalho, focamos no uso de concorrência no desenvolvimentos de aplicações reativas *soft real time* utilizando blocos concorrentes e ReactiveX. Utilizando duas ferramentas que implementam esses dois modelos, apontamos as diferenças, vantagens e dificuldades de cada um em diferentes situações. Concluímos que a combinação de mais de um modelo em uma mesma aplicação pode trazer vantagens ao permitir implementar cada parte do programa com a ferramenta mais adequada.

Os modelos apresentados diferem na integração com o restante da aplicação devido às diferenças no estilo de programação e nível de abstração de cada um. Os blocos concorrentes, por oferecerem menos abstrações, se integram mais facilmente a partes do código que fogem ao modelo. De fato, o seu uso estende e complementa a programação procedural tradicional, operando sobre variáveis locais e globais e executando chamadas de função. O ReactiveX, por outro lado, tenta ocultar a maior parte do estado e oferecer alternativas à programação procedural que foquem no fluxo de dados e na transformação dos mesmos. Em muitos casos, permite expressar operações grandes ou complexas de forma simples e direta ao utilizar operadores prontos. Contudo, esse maior nível de abstração dificulta a sua integração com partes do código que não sigam o modelo. O desenvolvimento de novos operadores, para que a aplicação possa utilizar as cadeias em situações não suportadas, pode não ser simples por fugir do uso típico do modelo. Além disso, enquanto o comportamento dos operadores é especificado pelo modelo, sua implementação não é. Assim, a forma de implementar novos operadores é totalmente dependente de cada implementação do modelo e pode nem ser documentada, como é o caso do RxLua.

Observamos que algumas situações não podem ser expressadas adequadamente nos modelos estudados. No Löve 2D, as chamadas gráficas para gerar um quadro devem ser feitas dentro da *callback* `love.draw`. Diferente das outras *callbacks*, nesta não estamos modificando o estado da aplicação, mas sim gerando uma representação gráfica desse estado. Por determinar o momento em que o estado é apresentado ao usuário, essa *callback* difere das demais ao necessitar acessar um contexto compartilhado com a aplicação como um todo. Do ponto de vista do usuário, a restrição de só executar chamadas gráficas

PUC-Rio - Certificação Digital Nº 1913163/CA

nessa reação impede as demais reações de agir imediatamente, uma vez que só haverá mudança da tela quando `love.draw` executar. Observe que o estado do programa é atualizado de forma independente dos momentos em que esse estado é apresentado, podendo inclusive ser atualizado muito mais rapidamente do que é apresentado ao usuário. Essencialmente, `love.draw` amostra o estado gerado pelas demais *callbacks* da *engine*.

Essa assimetria entre as *callbacks* que alteram o estado e aquela que o observa (`love.draw`) é difícil de representar tanto com blocos concorrentes como com ReactiveX. O formato estilo *pipeline* das cadeias do ReactiveX é mais adequado a reações imediatas aos eventos e isolamento das reações. A cadeia mantém o estado oculto dentro dos operadores e, após processar um evento, chama a *callback* do observador. Além disso, as cadeias só executam quando há um observador inscrito, não possibilitando que estas processem um evento que não será consumido imediatamente. Essas características fazem com que uma das formas mais simples de implementar a `love.draw` é fazer com que os observadores das cadeias escrevam em um estado compartilhado que será acessado para desenhar o quadro. Assim, observamos que em cenários como esse a utilização de apenas cadeias e operadores é uma solução mais complexa do que fugir do modelo nesse ponto. A implementação com blocos concorrentes é mais simples nesse caso. Por ter maior compartilhamento de contexto do que o ReactiveX, nesse modelo é mais natural ter um estado global compartilhado. Diferente do ReactiveX, nos blocos concorrentes podemos suspender a execução de uma tarefa, o que permitiria sincronizar a execução das trilhas que reagem aos comandos do usuário com um evento gerado pela `love.draw`. Assim, as trilhas poderiam esperar um evento que indicasse o momento em que é seguro executar chamadas gráficas. Essa solução, entretanto, tem dois problemas: Ao esperar o evento de *draw*, as trilhas deixariam de responder a outros eventos e não há garantia da ordem em que as trilhas executariam quando esse evento fosse emitido. Dessa forma, vemos que é preciso combinar os efeitos das várias trilhas em um mesmo contexto compartilhado que possa ser acessado de forma completa no momento de gerar o quadro.

Ao integrar com bibliotecas de terceiros, nem sempre existe a opção de APIs assíncronas ou baseadas em eventos que permitam o uso desses modelos. Essas integrações podem forçar o uso de chamadas bloqueantes, o que faria com que a aplicação parasse de responder a outros eventos durante a chamada. Da mesma forma, podem haver requisitos da aplicação que necessitem de operações computacionalmente caras. Mesmo que seja possível alterar o código nesses casos para limitar o tempo em quem bloqueiem a aplicação, isso pode não ser prático e dificultar a compreensão do código. Observe que esse tipo

PUC-Rio - Certificação Digital Nº 1913163/CA

de construção equivale à situação de *callback hell* ao encadear continuações. Essas duas situações violam a hipótese síncrona ao fazer com que a resposta a um evento demore um tempo não desprezível.

Na mesma linha da combinação de modelos que apresentamos, a LuaGravity (SANT'ANNA; IERUSALIMSCHY,) é uma linguagem para programação de aplicações reativas que combina a programação imperativa com o foco no fluxo de dados. Nessa linguagem podemos definir variáveis cujo valor é dado por uma expressão envolvendo outras variáveis. Ao alterar o valor de uma variável, este é propagado de forma a atualizar automaticamente todas as variáveis que dependem desta, reavaliando as expressões correspondentes. Internamente, essas variáveis são implementadas utilizando *Reactors*, tratadores de evento que podem ser relacionados para encadear suas execuções. *Reactors* também podem suspender a sua execução mantendo um estado local, por exemplo para esperar o término da execução de outro *reactor*. Essas características são semelhantes a algumas presentes nos blocos concorrentes e no ReactiveX. Há a orientação a fluxo de dados e encadeamento de tratadores de evento como no ReactiveX e a programação imperativa e suspensão de execução com contexto local presentes nos blocos concorrentes.

Nossa análise dos modelos foi baseada em implementações particulares e considerou apenas o desenvolvimento de aplicações em Lua. Uma análise mais ampla desses modelos poderia considerar o seu uso em outras linguagens com diferentes características ou mesmo outras implementações em Lua.

Nesse trabalho não comparamos uso de memória, tempo de execução, geração de lixo ou outras características de desempenho dessas implementações. Assim, propomos como trabalhos futuros testes de uso de recursos dos modelos e a comparação deles em outras linguagens de programação.

O código do módulo Lua Tasks está disponível em <https://github.com/naves-thiago/lua_tasks>. O código dos exemplos apresentados está disponível em <https://github.com/naves-thiago/dissertation>.

PUC-Rio - Certificação Digital Nº 1913163/CA

7
Referências bibliográficas

ADYA, A. et al. Cooperative task management without manual stack management. In: **Proceedings of the 2002 USENIX Annual Technical Conference**. [S.l.]: ACM, 2002. p. 289–302. Citado na página 21.

BAINOMUGISHA, E. et al. A survey on reactive programming. **ACM Computing Surveys**, v. 45, p. 1–34, 2013. Citado na página 28.

BANKEN, H.; MEIJER, E.; GOUSIOS, G. Debugging data flows in reactive programs. In: ACM. **ICSE '18: Proceedings of the 40th International Conference on Software Engineering**. [S.l.], 2018. p. 752–763. Citado 2 vezes nas páginas 62 e 63.

BERRY, G.; GONTHIER, G. The esterel synchronous programming language: design, semantics, implementation. **Science of Computer Programming**, Elsevir, v. 19, n. 2, p. 87–152, 1992. Citado 2 vezes nas páginas 13 e 24.

EDWARDS, J. Coherent reaction. In: ACM. **ACM Transactions on Object Oriented Programming Systems Languages and Applications**. [S.l.], 2009. p. 925–932. Citado na página 23.

ELLIOTT, C.; HUDAK, P. Functional reactive animation. **SIGPLAN Not.**, Association for Computing Machinery, v. 32, n. 8, p. 263–273, ago. 1997. Citado na página 28.

HAREL, D.; PNUELI, A. On the development of reactive systems. **Apt K.R. (eds) Logics and Models of Concurrent Systems. NATO ASI Series (Series F: Computer and Systems Sciences)**, Springer, v. 13, p. 477–498, 1985. Citado na página 13.

LESH, B. **Complex features made easy with RxJS**. 2018. <https://www.youtube.com/watch?v=B-nFj2o03i8>. Acessado: 2020-06-05. Citado na página 46.

LÖVE2D. <https://love2d.org>. Acessado: 2020-06-05. Citado 2 vezes nas páginas 15 e 28.

MOURA, A. L. D.; IERUSALIMSCHY, R. Revisiting coroutines. **ACM Transactions on Programming Languages and Systems**, ACM, v. 31, n. 2, 2009. Citado na página 19.

POTOP-BUTUCARU, D.; SIMONE, R.; TALPIN, J.-P. The synchronous hypothesis and synchronous languages. **Embedded Systems: Handbook**, 01 2005. Citado na página 18.

REACTIVEX. <http://reactivex.io>. Acessado: 2020-10-25. Citado 2 vezes nas páginas 13 e 27.

PUC-Rio - Certificação Digital Nº 1913163/CA

RXLUA. <https://github.com/bjornbytes/RxLua>. Acessado: 25.10.2020. Citado 2 vezes nas páginas 15 e 28.

SALVANESCHI, G.; MEZINI, M. Debugging reactive programming with reactive inspector. In: ACM. **ICSE '16: Proceedings of the 38th International Conference on Software Engineering**. [S.l.], 2016. p. 728–730. Citado na página 62.

SANT'ANNA, F.; IERUSALIMSCHY, R. Luagravity, a reactive language based on implicit invocation. Citado na página 67.

SANT'ANNA, F. et al. The design and implementation of the synchronous language céu. **ACM Transactions on Embedded Computing Systems**, ACM, n. 98, 2017. Citado 2 vezes nas páginas 13 e 24.

SHIN, K. G.; RAMANATHAN, P. Real-time computing: A new discipline of computer science and engineering. In: **Proceedings of the IEEE**. [S.l.]: IEEE, 1994. v. 82, n. 1, p. 6–24. Citado na página 16.

PUC-Rio - Certificação Digital Nº 1913163/CA

A
Pong - ReactiveX

Código 15: Pong com RxLua

```
rx = require'rx'
require'exhaustMap'
require'interval'

love.keypressed = rx.Subject.create()
love.keyreleased = rx.Subject.create()

local screenWidth = 640
local screenHeight = 480

local bumperSpeed = 400
local bumperWidth = 15
local bumperHeight = 80
local bumper1X = bumperWidth
local bumper1Y = (screenHeight - bumperHeight) / 2
local bumper2X = screenWidth - 2 * bumperWidth
local bumper2Y = bumper1Y

local ballSpeed = 250
local ballSize = 20
local ballXInitial = (screenWidth - ballSize) / 2
local ballYInitial = (screenHeight - ballSize) / 2
local ballX = ballXInitial
local ballY = ballYInitial
local ballXSpeed = ballSpeed
local ballYSpeed = 0

local matchLen = 180 -- Seconds
local clockScheduler = rx.CooperativeScheduler.create(0)
local clockText = ''

local updateS = rx.Subject.create()
function love.update(dt)
  updateS(dt)
  clockScheduler:update(dt)
end

function bumperPosFactory(keyUp, keyDown, currPos)
  return love.keypressed
    -- Filtra as teclas de interesse
    :filter(function(key) return key == keyUp or key == keyDown end)
    :exhaustMap(function(key)
      -- Define a direção do movimento (constante enquanto a
      -- tecla estiver pressionada)
```

PUC-Rio - Certificação Digital Nº 1913163/CA

```
45      local speed = key == keyUp and -bumperSpeed or bumperSpeed
46      return updateS:map(function(dt) return dt * speed end)
47        -- Acumula a posição Y do rebatedor
48        :scan(function(acc, new) return acc + new end, currPos)
49        :takeWhile(function(pos)
50          return pos >= 0 and pos <= screenHeight - bumperHeight
51        end)
52        -- Para ao soltar a tecla apenas se for a tecla que
53        -- iniciou o movimento
54        :takeUntil(love.keyreleased:filter(function(k) return key == k end))
55        -- Guarda a posição atual para a próxima vez que o rebatedor
56        -- for movido
57        :tap(function(pos) currPos = pos end)
58    end)
59 end
60
61 function didColideX()
62   if ballY + ballSize>= bumper1Y and ballY <= bumper1Y + bumperHeight and
63     ballX <= bumper1X + bumperWidth and ballXSpeed < 0 then
64     return true
65   end
66
67   if ballY + ballSize >= bumper2Y and ballY <= bumper2Y + bumperHeight and
68     ballX + ballSize >= bumper2X and ballXSpeed > 0 then
69     return true
70   end
71
72   return false
73 end
74
75 function didColideY()
76   return (ballY <= 0 and ballYSpeed < 0) or
77     (ballY + ballSize >= screenHeight and ballYSpeed > 0)
78 end
79
80 function setBallDirection()
81   local bumperY = ballX < screenWidth / 2 and bumper1Y or bumper2Y
82   local top = math.max(ballY, bumperY)
83   local distCenter = 2 * math.abs(bumperY + bumperHeight / 2 - top) / bumperHeight
84   local dirY = top < bumperY + bumperHeight / 2 and -1 or 1
85   ballYSpeed = distCenter * ballSpeed * dirY
86 end
87
88 function love.load()
89   love.window.setMode(screenWidth, screenHeight)
90   love.graphics.setFont(love.graphics.newFont(18))
91   love.graphics.setColor(1, 1, 1)
92
93   local matchEndedS = rx.Subject.create()
94
95   local bumper1YS = bumperPosFactory('w', 's', bumper1Y)
96   local bumper2YS = bumperPosFactory('up', 'down', bumper2Y)
```

PUC-Rio - Certificação Digital Nº 1913163/CA

```
97  bumper1YS:subscribe(function(pos) bumper1Y = pos end)
98  bumper2YS:subscribe(function(pos) bumper2Y = pos end)
99
100 local didScoreS = rx.Subject.create()
101 local startMovingS = rx.BehaviorSubject.create(1)
102 local ballPosS = startMovingS:exhaustMap(function()
103   ballYSpeed = 0
104   return updateS
105     :map(function(dt) return dt * ballXSpeed, dt * ballYSpeed end)
106     :scan(function(acc, newX, newY)
107       -- O valor acumulado é uma tabela com 2 entradas:
108       -- posições X e Y
109       acc = acc or {0, 0}
110       acc[1] = acc[1] + newX
111       acc[2] = acc[2] + newY
112       return acc
113     end, {ballXInitial, ballYInitial})
114     :takeUntil(didScoreS) -- Completa a cadeia quando alguém pontuar
115     :takeUntil(matchEndedS) -- Para a bola quando acabar o tempo da partida
116     :tap(function()
117       if didColideX() then
118         -- Colisão com rebatedor -> calcula nova velocidade em Y
119         ballXSpeed = -ballXSpeed
120         setBallDirection()
121       end
122       if didColideY() then
123         -- Colisão com a borda
124         ballYSpeed = -ballYSpeed
125       end
126     end)
127 end)
128
129 rx.Observable.Interval(1, clockScheduler)
130   :map(function(elapsed) return elapsed + 1 end)
131   :takeWhile(function(elapsed)
132     -- Completa a cadeia quando acabar o tempo da partida
133     return elapsed < matchLen
134   end)
135   :startWith(0)
136   :subscribe(function(elapsed) -- onNext
137     -- Atualiza o relógio na tela uma vez por segundo
138     local time = math.floor(matchLen - elapsed)
139     local min = time / 60
140     local sec = time % 60
141     clockText = string.format('%02d:%02d', min, sec)
142   end,
143   nil, -- onError
144   function() -- onCompleted
145     matchEndedS()
146     clockText = '00:00'
147   end)
148
```

PUC-Rio - Certificação Digital Nº 1913163/CA

```
149   ballPosS:subscribe(function(pos)
150     ballX = pos[1]
151     ballY = pos[2]
152   end)
153
154   ballPosS:subscribe(function()
155     if ballX > screenWidth - ballSize or
156       ballX < 0 then
157       print('score')
158       didScoreS(1)
159       startMovingS(1)
160     end
161   end)
162 end
163
164 function love.draw()
165   love.graphics.print(clockText, 280, 10)
166   love.graphics.rectangle('fill', ballX, ballY, ballSize, ballSize)
167   love.graphics.rectangle('fill', bumper1X, bumper1Y, bumperWidth, bumperHeight)
168   love.graphics.rectangle('fill', bumper2X, bumper2Y, bumperWidth, bumperHeight)
169 end
```

PUC-Rio - Certificação Digital Nº 1913163/CA

B
Pong - Blocos concorrentes

Código 16: Pong com blocos concorrentes

```
local tasks = require'tasks'

local screenWidth = 640
local screenHeight = 480

local bumperSpeed = 400
local bumperWidth = 15
local bumperHeight = 80
local bumper1X = bumperWidth
local bumper1Y = (screenHeight - bumperHeight) / 2
local bumper2X = screenWidth - 2 * bumperWidth
local bumper2Y = bumper1Y

local ballSpeed = 250
local ballSize = 20
local ballXInitial = (screenWidth - ballSize) / 2
local ballYInitial = (screenHeight - ballSize) / 2
local ballX = ballXInitial
local ballY = ballYInitial
local ballXSpeed = ballSpeed
local ballYSpeed = 0

local matchLen = 180 -- Seconds
local clockText = ''

function didColideX()
  if ballY + ballSize>= bumper1Y and ballY <= bumper1Y + bumperHeight and
    ballX <= bumper1X + bumperWidth and ballXSpeed < 0 then
    return true
  end

  if ballY + ballSize >= bumper2Y and ballY <= bumper2Y + bumperHeight and
    ballX + ballSize >= bumper2X and ballXSpeed > 0 then
    return true
  end

  return false
end

function didColideY()
  return (ballY <= 0 and ballYSpeed < 0) or
    (ballY + ballSize >= screenHeight and ballYSpeed > 0)
end

```

PUC-Rio - Certificação Digital Nº 1913163/CA

```
45 function setBallDirection()
46   local bumperY = ballX < screenWidth / 2 and bumper1Y or bumper2Y
47   local top = math.max(ballY, bumperY)
48   local distCenter = 2 * math.abs(bumperY + bumperHeight / 2 - top) / bumperHeight
49   local dirY = top < bumperY + bumperHeight / 2 and -1 or 1
50   ballYSpeed = distCenter * ballSpeed * dirY
51 end
52
53 function startBumper1Task()
54   tasks.task_t:new(function()
55     while true do
56       tasks.par_or( -- Espera 'w_down' e 's_down' simultaneamente
57         function()
58           tasks.await('w_down')
59           tasks.par_or(
60             function()
61               while true do
62                 local dt = tasks.await('update')
63                 bumper1Y = math.max(0, bumper1Y - bumperSpeed * dt)
64               end
65             end,
66             function()
67               tasks.await('w_up')
68               -- Volta a esperar o início do movimento
69               tasks.emit('bumper1_done')
70             end
71           )(true, true) -- Inicia esse par_or sem bloquear a tarefa e não
72                         -- termina quando ela terminar
73           -- A tarefa termina imediatamente após iniciar o par_or,
74           -- terminando o par_or mais externo (mas não o interno)
75         end,
76         function()
77           tasks.await('s_down')
78           tasks.par_or(
79             function()
80               while true do
81                 local dt = tasks.await('update')
82                 bumper1Y = math.min(screenHeight - bumperHeight,
83                                     bumper1Y + bumperSpeed * dt)
84               end
85             end,
86             function()
87               tasks.await('s_up')
88               tasks.emit('bumper1_done')
89             end
90           )(true, true) -- Inicia esse par_or sem bloquear a tarefa e não
91                         -- termina quando ela terminar
92           -- A tarefa termina imediatamente após iniciar o par_or,
93           -- terminando o par_or mais externo (mas não o interno)
94         end
95       )()
96       -- A atualização da posição executa como uma tarefa independente
```

PUC-Rio - Certificação Digital Nº 1913163/CA

```
97        -- Espera ela terminar antes de permitir iniciar o movimento novamente
98        tasks.await('bumper1_done')
99      end
100   end)(true, true)
101 end
102
103 function startBumper2Task()
104   tasks.task_t:new(function()
105     while true do
106       tasks.par_or( -- Espera 'w_down' e 's_down' simultaneamente
107         function()
108           tasks.await('up_down')
109           tasks.par_or(
110             function()
111               while true do
112                 local dt = tasks.await('update')
113                 bumper2Y = math.max(0, bumper2Y - bumperSpeed * dt)
114               end
115             end,
116             function()
117               tasks.await('up_up')
118               -- Volta a esperar o início do movimento
119               tasks.emit('bumper2_done')
120             end
121           )(true, true) -- Inicia esse par_or sem bloquear a tarefa e não
122                         -- termina quando ela terminar
123           -- A tarefa termina imediatamente após iniciar o par_or,
124           -- terminando o par_or mais externo (mas não o interno)
125         end,
126         function()
127           tasks.await('down_down')
128           tasks.par_or(
129             function()
130               while true do
131                 local dt = tasks.await('update')
132                 bumper2Y = math.min(screenHeight - bumperHeight,
133                                     bumper2Y + bumperSpeed * dt)
134               end
135             end,
136             function()
137               tasks.await('down_up')
138               tasks.emit('bumper2_done')
139             end
140           )(true, true) -- Inicia esse par_or sem bloquear a tarefa e não
141                         -- termina quando ela terminar
142           -- A tarefa termina imediatamente após iniciar o par_or,
143           -- terminando o par_or mais externo (mas não o interno)
144         end
145       )()
146       -- A atualização da posição executa como uma tarefa independente
147       -- Espera ela terminar antes de permitir iniciar o movimento novamente
148       tasks.await('bumper2_done')
```

PUC-Rio - Certificação Digital Nº 1913163/CA

```
149     end
150   end)(true, true)
151 end
152
153 function startBallTask()
154   local function update_ball_pos()
155     while true do
156       local dt = tasks.await('update')
157       -- Calcula o deslocamento da bola a cada dt segundos
158       ballX = ballX + ballXSpeed * dt
159       ballY = ballY + ballYSpeed * dt
160       if didColideX() then
161         -- Colisão com rebatedor -> calcula nova velocidade em Y
162         ballXSpeed = -ballXSpeed
163         setBallDirection()
164       end
165       if didColideY() then
166         -- Colisão com a borda superior / inferior
167         ballYSpeed = -ballYSpeed
168       end
169       if ballX > screenWidth - ballSize or
170         ballX < 0 then
171         -- Colisão com a borda direita / esquerda -> ponto
172         print('score')
173         ballX = ballXInitial
174         ballY = ballYInitial
175         ballYSpeed = 0
176       end
177     end
178   end
179   tasks.par_or(
180     update_ball_pos,
181     function()
182       -- Para a bola quando acabar o tempo da partida
183       tasks.await_ms(matchLen * 1000)
184       clockText = '00:00'
185     end,
186     function()
187       -- Atualiza o relógio na tela uma vez por segundo
188       local time = math.ceil(matchLen - tasks.now_ms() / 1000)
189       clockText = string.format('%02d:%02d', time / 60, time % 60)
190       while true do
191         tasks.await_ms(1000)
192         time = math.ceil(matchLen - tasks.now_ms() / 1000)
193         clockText = string.format('%02d:%02d', time / 60, time % 60)
194       end
195     end
196   )(true, true)
197 end
198
199 function love.load()
200   love.window.setMode(screenWidth, screenHeight)
```

PUC-Rio - Certificação Digital Nº 1913163/CA

```
201   love.graphics.setFont(love.graphics.newFont(18))
202   love.graphics.setColor(1, 1, 1)
203   startBumper1Task()
204   startBumper2Task()
205   startBallTask()
206 end
207
208 function love.keypressed(key, scancode, isrepeat)
209   tasks.emit(key .. '_down')
210 end
211
212 function love.keyreleased(key)
213   tasks.emit(key .. '_up')
214 end
215
216 function love.update(dt)
217   tasks.emit('update', dt)
218   tasks.update_time(dt * 1000)
219 end
220
221 function love.draw()
222   love.graphics.print(clockText, 280, 10)
223   love.graphics.rectangle('fill', ballX, ballY, ballSize, ballSize)
224   love.graphics.rectangle('fill', bumper1X, bumper1Y, bumperWidth, bumperHeight)
225   love.graphics.rectangle('fill', bumper2X, bumper2Y, bumperWidth, bumperHeight)
226 end
```

PUC-Rio - Certificação Digital Nº 1913163/CA

C
Leitor de notícias - ReactiveX

Código 17: Leitor de notícias com RxLua

```
local tasks = require'tasks'
local cards = require'cards'
local loading_icon_t = require('loadingIcon').loading_icon_t
local animations = require'animations'
local rx = require'rx'
require'exhaustMap'
require'catchError'
require'share'
require'resub'
require'endWith'

love.mousemoved = rx.Subject.create()
love.mousepressed = rx.Subject.create()
love.mousereleased = rx.Subject.create()

local news_cards
local load_ico

function love.load()
  love.window.setMode(410, 600)
  love.window.setTitle("News")

  local loadNews = http_get('/newsfeed')
    :catchError(function(e)
      print('[ERROR] error loading news feed')
      print('[ERROR] ' .. e)
    end)

  -- Subject that forces a news refresh
  local refresh = rx.BehaviorSubject.create(1)

  -- News feed observable
  local news = refresh:exhaustMap(function()
    return loadNews
  end):share()

  -- Reload news periodically
  timer(0, 30000):subscribe(refresh)

  -- Card list to display the news
  news_cards = cards.card_list_t:new(5, 5, 400, window_height() - 5)
  news:subscribe(function(n)
    news_cards:clear()
    for _, str in ipairs(n) do
```

PUC-Rio - Certificação Digital Nº 1913163/CA

```
45      local c = cards.card_t:new(str)
46      news_cards:add_card(c)
47    end
48  end)
49
50  -- Loading icon object
51  load_ico = loading_icon_t:new(195, 0, 20)
52
53  -- Loading icon spring back animation
54  local load_ico_move_home = rx.Observable.defer(function()
55      return animations.tween(load_ico.y, 0, 200)
56    end)
57
58  -- Report mouse Y movement while the mouse is down relative to
59  -- the mouse down position
60  local mouse_drag = love.mousepressed:exhaustMap(function(start_x, start_y)
61    return rx.Observable.concat(
62        love.mousemoved
63          -- extract Y and offset by the start Y
64          :map(function(x, y) return y - start_y end)
65          -- stop tracking when the mouse is released
66          :takeUntil(love.mousereleased),
67        load_ico_move_home
68        )
69          -- Trigger a news reload when we go past half window
70          :tap(function(y)
71            if y > window_height() / 2 then
72              refresh:onNext(1)
73            end
74          end)
75          -- Stop when we get bellow half screen
76          :takeWhile(function(y) return y <= window_height() / 2 end)
77  end):share()
78
79  -- Animate the icon back home after loading the news
80  local load_ico_position_update = mouse_drag:exhaustMap(function()
81    return rx.Observable.concat(
82      mouse_drag:takeUntil(news),
83      load_ico_move_home)
84  end)
85
86  -- Emits the positions for the loading icon
87  local load_ico_position = load_ico_position_update
88    -- Start outside the screen (account for diagonal size due to rotation)
89    :startWith(-load_ico.size / 2)
90    -- Offset by the square size
91    :map(function(y) return y - load_ico.size end)
92
93  -- Loading icon rotation observable
94  local load_ico_rotate = refresh:exhaustMap(function()
95    return animations.tween(0, 360, 500)
96        :resub()
```

PUC-Rio - Certificação Digital Nº 1913163/CA

```
97          :takeUntil(news)
98          :endWith(0)
99    end)
100
101   load_ico_position:subscribe(function(p) load_ico.y = p end)
102   load_ico_rotate:subscribe(function(r) load_ico.rotation = r end)
103 end
104
105 function love.update(dt)
106   tasks.update_time(dt * 1000)
107 end
108
109 function love.draw()
110   news_cards:draw()
111   load_ico:draw()
112 end
113
114 function window_height()
115   local _, h = love.window.getMode()
116   return h
117 end
118
119 ------------------------------------------------
120 -- HTTP request mock
121 local mock_content = {
122   'Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed nunc nisl, '..
123     'volutpat id aliquet eu, semper in nisi.',
124   'Maecenas nec ornare libero. Class aptent taciti sociosqu ad litora '..
125     'torquent per conubia nostra, per inceptos himenaeos. '..
126     'Praesent mattis ex eget dolor sagittis ornare.',
127   'Morbi imperdiet pharetra arcu.',
128   'Curabitur rhoncus, lectus ac elementum lacinia, ligula elit mollis velit, '..
129     'egestas porttitor nisi dui in eros.',
130   'Nam turpis tellus, malesuada at augue ac, mollis dictum lorem. Morbi mi mi, '..
131     'laoreet ut erat sed, faucibus egestas lorem. '..
132     'Nam sodales lacus nec viverra sagittis.',
133   'Mauris in sodales lorem, non blandit nulla. Vestibulum ante ipsum primis '..
134     'in faucibus orci luctus et ultrices posuere cubilia curae; '..
135     'Aenean mollis metus eget venenatis venenatis. '..
136     'Mauris elementum cursus rhoncus. Duis at nisl eu dolor congue aliquam.',
137   'Aliquam eu magna vel odio malesuada lacinia et sit amet justo. '..
138     'Curabitur in posuere quam.',
139   'Pellentesque ultricies bibendum sapien ac lobortis. Mauris eget ex augue.',
140   'Phasellus dignissim vitae urna id hendrerit. '..
141     'Maecenas malesuada vulputate arcu a accumsan. '..
142     'Quisque dictum blandit risus, ac consequat lectus scelerisque vitae.',
143   'Duis ac gravida velit. Nulla lectus ipsum, ullamcorper a nulla sed, '..
144     'volutpat blandit ipsum. Donec cursus tellus ut vestibulum posuere.',
145   'Vestibulum nec odio sed magna venenatis porttitor ac ut metus. '..
146     'Vivamus eu tortor eget est venenatis '..
147     'lacinia. Mauris aliquet nunc ut velit sollicitudin luctus. '..
148     'Curabitur iaculis commodo enim, nec volutpat libero sollicitudin id. '..
```

PUC-Rio - Certificação Digital Nº 1913163/CA

```
149     'Phasellus nec cursus tortor. Donec ultrices, justo at pharetra laoreet, '..
150     'lacus dui blandit risus, quis vehicula augue justo nec lacus.',
151   'Phasellus varius pulvinar tristique. Fusce mi arcu, venenatis eu nulla at, '..
152     'fringilla porta turpis. Praesent commodo condimentum risus, '..
153     'id lobortis ex. Ut eget nisl ligula.',
154 }
155
156 local mock_content_steps = {3, 5, 3} -- how many posts to add on each reply
157 local mock_current_step = 1 -- next content step to use
158 local mock_sent = 0 -- sent posts
159 local http_task = tasks.task_t:new(function()
160   while true do
161     tasks.await('get news')
162     tasks.await_ms(2000)
163     local count = mock_sent + mock_content_steps[mock_current_step]
164     count = math.min(count, #mock_content)
165     tasks.emit('news', {unpack(mock_content, 1, count)})
166     if mock_current_step < #mock_content_steps then
167       mock_sent = mock_sent + mock_content_steps[mock_current_step]
168       mock_current_step = mock_current_step + 1
169     end
170     tasks.emit('news done')
171   end
172 end)
173 http_task()
174
175 function http_get(path)
176   return rx.Observable.create(function(observer)
177     local function onNext(_, n)
178       observer:onNext(n)
179     end
180
181     local function onCompleted()
182       observer:onCompleted()
183     end
184
185     tasks.listen('news', onNext)
186     tasks.listen('news done', onCompleted)
187     tasks.emit('get news')
188     return rx.Subscription.create(function()
189       tasks.stop_listening('news', onNext)
190       tasks.stop_listening('news done', onCompleted)
191     end)
192   end)
193 end
194
195 ----------------------------------------------
196 -- Timer interface
197 function timer(initial, interval)
198   return rx.Observable.create(function(observer)
199     local count = initial
200     local function onNext()
```

PUC-Rio - Certificação Digital Nº 1913163/CA

```
201       observer:onNext(count)
202       count = count + 1
203     end
204     local timer = tasks.every_ms(interval, onNext)
205     return rx.Subscription.create(function()
206       timer:stop()
207     end)
208   end)
209 end
```

PUC-Rio - Certificação Digital Nº 1913163/CA

D
Leitor de notícias - Blocos concorrentes

Código 18: Leitor de notícias com blocos concorrentes

```
local tasks = require'tasks'
local cards = require'cards'
local loading_icon_t = require('loadingIcon').loading_icon_t
local tween = require('animations').tween

local reload_task
local news_cards
local load_ico

-- Request the news and fill the news_cards
function update_news()
  tasks.task_t:new(function()
    local success, n = http_get('/newsfeed')
    if not success then
      print('[ERROR] error loading news feed')
      print('[ERROR] ' .. n)
      return
    end

    news_cards:clear()
    for _, str in ipairs(n) do
      local c = cards.card_t:new(str)
      news_cards:add_card(c)
    end
  end)()
end

-- Animate the loading icon back home
function load_ico_move_home()
  tween(load_ico.y, -load_ico.size, 200, function(y)
    load_ico.y = y
  end)()
end

-- Reload task function
function manual_reload_f()
  while true do
    local _, start_y = tasks.await('mousepressed')
    local reload = tasks.par_or(
      function()
        local y
        repeat
          _, y = tasks.await('mousemoved')
          load_ico.y = y - start_y - load_ico.size
```

PUC-Rio - Certificação Digital Nº 1913163/CA

```
45          until y - start_y >= window_height() / 2
46          return true
47        end,
48        function()
49          tasks.await('mousereleased')
50        end
51      )
52
53      if reload() then
54        local spin_task = tween(0, 360, 500, function(r)
55          load_ico.rotation = r
56        end, true)
57        tasks.par_or(spin_task, update_news)()
58        load_ico.rotation = 0
59      end
60      load_ico_move_home()
61    end
62  end
63
64  function love.load()
65    love.window.setMode(410, 600)
66    love.window.setTitle("News")
67
68    load_ico = loading_icon_t:new(195, -30, 20)
69    news_cards = cards.card_list_t:new(5, 5, 400, window_height() - 5)
70
71    -- Load news when the app start
72    update_news()
73
74    -- Reload news periodically
75    tasks.task_t:new(function()
76      while true do
77        tasks.await_ms(30000)
78        update_news()
79      end
80    end)()
81    reload_task = tasks.task_t:new(manual_reload_f)
82    reload_task()
83  end
84
85  function love.update(dt)
86    tasks.update_time(dt * 1000)
87  end
88
89  function love.draw()
90    news_cards:draw()
91    load_ico:draw()
92  end
93
94  function window_height()
95    local _, h = love.window.getMode()
96    return h
```

PUC-Rio - Certificação Digital Nº 1913163/CA

```lua
end

function love.mousemoved(...)
  tasks.emit('mousemoved', ...)
end

function love.mousepressed(...)
  tasks.emit('mousepressed', ...)
end

function love.mousereleased(...)
  tasks.emit('mousereleased', ...)
end

-------------------------------------------------
-- HTTP request mock
local mock_content = {
  'Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed nunc nisl, '..
    'volutpat id aliquet eu, semper in nisi.',
  'Maecenas nec ornare libero. Class aptent taciti sociosqu ad litora '..
    'torquent per conubia nostra, per inceptos himenaeos. '..
    'Praesent mattis ex eget dolor sagittis ornare.',
  'Morbi imperdiet pharetra arcu.',
  'Curabitur rhoncus, lectus ac elementum lacinia, ligula elit mollis velit, '..
    'egestas porttitor nisi dui in eros.',
  'Nam turpis tellus, malesuada at augue ac, mollis dictum lorem. Morbi mi mi, '..
    'laoreet ut erat sed, faucibus egestas lorem. '..
    'Nam sodales lacus nec viverra sagittis.',
  'Mauris in sodales lorem, non blandit nulla. Vestibulum ante ipsum primis '..
    'in faucibus orci luctus et ultrices posuere cubilia curae; '..
    'Aenean mollis metus eget venenatis venenatis. '..
    'Mauris elementum cursus rhoncus. Duis at nisl eu dolor congue aliquam.',
  'Aliquam eu magna vel odio malesuada lacinia et sit amet justo. '..
    'Curabitur in posuere quam.',
  'Pellentesque ultricies bibendum sapien ac lobortis. Mauris eget ex augue.',
  'Phasellus dignissim vitae urna id hendrerit. '..
    'Maecenas malesuada vulputate arcu a accumsan. '..
    'Quisque dictum blandit risus, ac consequat lectus scelerisque vitae.',
  'Duis ac gravida velit. Nulla lectus ipsum, ullamcorper a nulla sed, '..
    'volutpat blandit ipsum. Donec cursus tellus ut vestibulum posuere.',
  'Vestibulum nec odio sed magna venenatis porttitor ac ut metus. '..
    'Vivamus eu tortor eget est venenatis '..
    'lacinia. Mauris aliquet nunc ut velit sollicitudin luctus. '..
    'Curabitur iaculis commodo enim, nec volutpat libero sollicitudin id. '..
    'Phasellus nec cursus tortor. Donec ultrices, justo at pharetra laoreet, '..
    'lacus dui blandit risus, quis vehicula augue justo nec lacus.',
  'Phasellus varius pulvinar tristique. Fusce mi arcu, venenatis eu nulla at, '..
    'fringilla porta turpis. Praesent commodo condimentum risus, '..
    'id lobortis ex. Ut eget nisl ligula.',
}

local mock_content_steps = {3, 5, 3} -- how many posts to add on each reply
```

PUC-Rio - Certificação Digital Nº 1913163/CA

```
149 local mock_current_step = 1 -- next content step to use
150 local mock_sent = 0 -- sent posts
151 local http_task = tasks.task_t:new(function()
152   while true do
153     tasks.await('get news')
154     tasks.await_ms(2000)
155     local count = mock_sent + mock_content_steps[mock_current_step]
156     count = math.min(count, #mock_content)
157     tasks.emit('news', {unpack(mock_content, 1, count)})
158     if mock_current_step < #mock_content_steps then
159       mock_sent = mock_sent + mock_content_steps[mock_current_step]
160       mock_current_step = mock_current_step + 1
161     end
162   end
163 end)
164 http_task()
165
166 function http_get(path)
167   tasks.emit('get news')
168   return tasks.par_or(
169     function()
170       return true, tasks.await('news')
171     end,
172     function()
173       return false, tasks.await('news error')
174     end)()
175 end
```

PUC-Rio - Certificação Digital Nº 1913163/CA