UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

DIEGO BORGES

Rivers - API para Processamento de *Stream* para linguagem Go

Projeto de Diplomação

Orientador: Prof. Dr. Marcelo Pimenta

Porto Alegre
2015

“Do not communicate by sharing memory; instead, share memory by communicating”

— Rob Pike

AGRADECIMENTOS

Agradeço a todas as pessoas que me estimularam o bastante para realizar este trabalho: Aos meus pais e a toda minha família, que me apoiou, cobrou, deu toda a estrutura necessária que viabilizou meus estudos durante todos esses anos e, principalmente, serviu de inspiração e modelo para a formação de meu caráter e val. A Bianca, que muitas vezes, sempre com muita paciência evitou com que eu fizesse coisas menos importantes em detrimento deste trabalho. Ao Prof. Pimenta, meu orientador que me ajudou não somente durante o desenvolvimento deste trabalho mas também ao longo dos anos de curso compartilhando seus conhecimentos e ideias com uma didática impecável.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS **6**
LISTA DE FIGURAS **7**
LISTA DE TABELAS **8**
RESUMO **9**
ABSTRACT **10**
1 INTRODUÇÃO **11**
1.1 Objetivos **11**
1.2 Estrutura do Texto **12**
2 FUNDAMENTOS **14**
2.1 Streams **14**
2.2 A Linguagem Go **15**
2.2.1 Interfaces 15
2.2.2 Goroutines 16
2.2.3 Canais de Comunicação 17
3 TRABALHOS RELACIONADOS **19**
3.1 Apache Spark **19**
3.2 Java 8 Streams **19**
3.3 Reactive eXtensions **20**
4 RIVERS **21**
4.1 Arquitetura **21**
4.2 Building Blocks **24**
4.2.1 Streams 24
4.2.2 Attachables 25
4.2.3 Producers 25
4.2.4 Transformers 29
4.2.5 Consumers 32
4.2.6 Combiners 34
4.2.7 Dispatchers 35
4.2.8 O Contexto Global 36
4.3 Suporte a Paralelização **37**
5 PROCESSO DE DESENVOLVIMENTO ÁGIL DE RIVERS **39**
5.1 Coleta de Requisitos e Roadmap **39**
5.2 Tests & Benchmarks **41**
6 RIVERS: APPLICAÇÕES REAIS **42**
6.1 Appx - Appengine eXtensions **42**
6.2 Web Scrapping **44**
7 CONCLUSÃO **47**
7.1 Análise dos Resultados **47**
7.2 Trabalhos Futuros **48**
REFERÊNCIAS **49**

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
CPU	Central Processing Unit
FIFO	First In First Out
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
HTTP	Hypertext Transfer Protocol
NoSQL	Non SQL
MVP	Minimum Viable Product
TDD	Test-Driven Development
HTML	HyperText Markup Language
URL	Uniform Resource Locator

LISTA DE FIGURAS

Figura 2.1 Exemplo de stream pipeline em Unix. 15
Figura 2.2 Exemplo de implementação de uma interface em Java. 16
Figura 2.3 Exemplo de implementação de uma interface em Go. 16
Figura 2.4 Exemplo de uso de Goroutines. 17
Figura 2.5 Exemplo de uso de canais de comunicação. 18

Figura 3.1 Exemplo de uso da API scala de Spark. 19
Figura 3.2 Exemplo de uso da API de streams em Java8 20

Figura 4.1 Componentes Básicos de um Pipeline Rivers 22
Figura 4.2 Exemplo de Uso de Combiner. 23
Figura 4.3 Exemplo de Uso de Dispatcher. 24
Figura 4.4 Rivers Stream. 25
Figura 4.5 Attachable Interface. 25
Figura 4.6 Producer Interface. 26
Figura 4.7 Implementação de Producer que gera números inteiros entre um intervalo pré-definido. 27
Figura 4.8 Implementação de RangeProducer em termos de producers.Observable. 28
Figura 4.9 Transformer Interface. 29
Figura 4.10 Filter Transformer. 30
Figura 4.11 Uso do Filter Transformer em um pipeline Rivers. 30
Figura 4.12 Filter Transformer em termos de um Observer 31
Figura 4.13 Uso do EvensOnly Filter em um pipeline Rivers. 31
Figura 4.14 Exemplos de uso de operações de filtros. 32
Figura 4.15 Consumer Interface. 32
Figura 4.16 Aplicação de consumers em Rivers. 33
Figura 4.17 Implementação do Consumer Count. 33
Figura 4.18 Implementação do Consumer utilizando o tipo Sink como base. 34
Figura 4.19 Combiner Interface. 34
Figura 4.20 Zip Combiner Interface 35
Figura 4.21 Dispatcher Interface. 35
Figura 4.22 Exemplo de particionamento de um stream de números inteiros. 35
Figura 4.23 Context Interface. 36
Figura 4.24 Fluxo de um Pipeline com Paralelismo ativado. 37
Figura 4.25 Exemplo de Paralelismo em Rivers. 38

Figura 5.1 Roadmap visualizado em um dashboard Waffle. 40
Figura 5.2 Ciclo de desenvolvimento aplicando a técnica TDD. 41

Figura 6.1 Exemplo de uso da API nativa do Datastore. 43
Figura 6.2 Exemplo de uso da API de streamming do framework Appx utilizando Rivers com paralelismo ativado. 44
Figura 6.3 Exemplo de um Crawler que implementa a interface stream.Producer. 45
Figura 6.4 Exemplo de um Scrapper que implementa a interface stream.Transformer. 46
Figura 6.5 Exemplo de uso do Crawler e Scrapper em um pipeline Rivers 46

LISTA DE TABELAS

Tabela 5.1 Rivers Roadmap: MVP vs. V2.0 40
Tabela 5.2 Comparação entre duas versões de um programa: Sequencial sem o uso de Rivers e Paralelo com o uso de Rivers 41

RESUMO

Nos últimos anos o poder computacional evoluiu drasticamente e os sistemas computacionais atuais podem beneficiar-se de máquinas com múltiplas unidades de processamento para realizar concorrentemente tarefas de maneira mais eficiente.

Go tira proveito do poder computacional de hardwares modernos implementando um modelo de concorrência gerenciado pelo runtime da linguagem conhecido como goroutines e sincronização via troca de mensagens através do uso de canais de comunicação.

Este trabalho tem como objetivo a criação de um framework para processamento de streams de dados utilizando o modelo de concorrência da linguagem Go como fundação e padrões bem conhecidos como Produtor-Consumidor e Go Pipeline Pattern provendo uma API fluente e extensível para criação e composição de pipelines complexos de processamento de dados aplicando conceitos de programação funcional.

Palavras-chave: API. rivers. golang. stream. pipeline. concurrency.

Rivers - Stream Processing API for Golang

ABSTRACT

During the past few years hardware power has evolved drastically and today's systems can leverage multi-core CPUs in order to perform concurrent tasks more effectively.
Go takes advantage of this hardware power by implementing a simple though extremely powerful concurrency model built on top of concepts such as message passing via channels and a more lightweight form of thread managed by the runtime known as goroutines.
The goal of this work is to provide a framework for data stream processing built on top of Go's concurrency model along with well known patterns such as the Producer-Consumer pattern and the Go pipeline pattern in order to provide a fluent and extensible API for building and composing complex data processing pipelines through functional programming concepts.

Keywords: API, rivers, golang, stream, pipeline, concurrency.

1 INTRODUÇÃO

Nos últimos anos a necessidade de se processar grandes volumes de dados de maneira eficiente e flexível tem aumentado drasticamente. Isso se deve ao fato de que sistemas computacionais estão cada vez mais complexos e frequentemente envolvem a colaboração entre diversos outros sistemas que atuam como fonte ou consumidores de dados. O fato de que as pessoas estão cada vez mais conectadas aos seus dispositivos móveis coletando e produzindo dados a todo momento é um dos indícios dessa crescente complexidade que tende a continuar à medida que novas tecnologias são introduzidas como por exemplo *Internet of Things* (COUNCIL, 2015), aumentando ainda mais a quantidade de dados produzidos e com isso a necessidade de se processar e comunicar estes dados entre os diferentes sistemas que compõem o que conhecemos como Internet.

Dados são gerados assincronamente por diversas fontes como por exemplo aplicativos móveis, sensores, sistemas web e fluem de um sistema a outro através de mecanismos de comunicação tais como APIs e *Push Notifications* (APPLE, 2015) passando por transformações, filtros e agregações para se adequar ao contexto e as necessidades de cada sistema. Todo este processamento quando aplicado a grandes quantidades de dados em um contexto de sistemas distribuídos torna-se surpreendentemente desafiador. Várias soluções foram propostas nos últimos anos para abordar o processamento assíncrono de grandes volume de dados dentre elas o processamento de *streams*, solução explorada por este trabalho no contexto da linguagem de programação Go da Google.

A decisão pelo uso da linguagem de programação Go se deve a alguns fatores. Go introduz um modelo de concorrência poderoso e simples baseado em troca de mensagens através da colaboração entre canais de comunicação e *goroutines* melhores explicados nas seções 2.2.3 e 2.2.2 permitindo um único programa executar centenas de milhares de fluxos concorrentes de execução com o mínimo de overhead, o que a torna uma linguagem extremamente atraente no contexto de processamento de *streams*. Go tem ganhado notável adoção na comunidade de desenvolvedores tanto pela simplicidade da linguagem que permite com que soluções elegantes sejam empregadas à problemas complexos como no desenvolvimento de web servers, aplicações que fazem intenso uso de mecanismos de comunicação de rede como TCP e UDP, *parsers* e etc. No entanto, a linguagem não provê abstrações nativas para o processamento de *streams* tornando o trabalhado de um desenvolvedor longe de ser trivial.

1.1 Objetivos

Este trabalho tem como principal objetivo a proposta e desenvolvimento de Rivers uma API simples e flexível para processamento de streams através da abstração de pipelines para a linguagem de programação Go da Google. A solução proposta tem como base teórica conceitos

do modelo Produtor-Consumidor (KOCHER, 2005) juntamente com conceitos de programação funcional tais como *Collection Pipeline* definido em (FOWLER, 2015a). Em sua fundação prática, a solução explora o modelo de concorrência da linguagem Go a fim de usufruir ao máximo dos recursos de *hardware* visando atingir níveis extremos de concorrência e paralelismo.

Visando o reuso de código, a API proposta disponibiliza um conjunto mínimo de abstrações para cada um dos conceitos envolvidos na criação de um pipeline e seus estágios de processamento de *stream* que servirão como base para a implementação de componentes mais complexos e especializados, tais como mecanismos básicos para a produção de dados de diversas fontes como por exemplo arquivos, *sockets* e listas assim como componentes especializados em operações de transformações de dados como por exemplo *Map*, *Reduce*, *Filter* encontradas em outras soluções similares discutidas no capítulo 3. A fim de adequar-se a cenários não inicialmente previstos Rivers disponibiliza mecanismos para que abstrações nativas da API possam ser estendidas com novas implementações permitindo a criação de pipelines para o processamento de qualquer tipo de dados independente de sua origem.

Finalmente Rivers, seguindo conceitos de programação funcional deve permitir que estágios do pipeline possam ser combinados de maneira simples tal que dados produzidos por um estágio possam ser consumidos por um próximo estágio de maneira assíncrona e concorrente, sendo possível (porém opcional) a paralelização de estágios específicos a fim de aumentar a eficiência de processamento de dados de um estágio em particular.

1.2 Estrutura do Texto

Este trabalho está organizado em sete capítulos os quais abordam temas relevantes para o entendimento das decisões tomadas no processo de desenvolvimento da solução e suas motivações assim como definição de conceitos básicos utilizados como fundação do trabalho. A seguir cada capítulo é apresentado de maneira mais detalhada.

Capítulo 1 Apresenta o contexto em que o trabalho foi desenvolvido as motivações que justificam a solução proposta assim como seus objetivos.

Capítulo 2 Descreve os termos e conceitos básicos utilizados como fundação do trabalho dando suporte teórico e prático para a solução proposta.

Capítulo 3 Discute alguns trabalhos relacionados que serviram de base e inspiração para muitas das decisões técnicas tomadas ao longo do processo de desenvolvimento deste trabalho.

Capítulo 4 Apresenta a solução proposta em detalhes assim como a arquitetura desenvolvida e seus componentes internos

Capítulo 5 Descreve o processo de desenvolvimento ágil empregado assim como as práticas utilizadas para a validação gradual de requisitos assim como a integração contínua e constante da implementação empregada.

Capítulo 6 Apresenta alguns usos detectados ao longo do desenvolvimento assim como aplicações reais da solução em ambientes de produção juntamente como a visão de desenvolvedores que fizeram para da validação deste trabalho.

Capítulo 7 Por fim, são apresentados as conclusões, resultados e algumas considerações finais com relação a possíveis melhorias e avanços futuros.

2 FUNDAMENTOS

Neste capítulo serão discutidos alguns conceitos básicos utilizados como fundação no desenvolvimento da solução assim como alguns detalhes técnicos relacionados a linguagem de programação Go necessários para o entendimento dos mecanismos implementados em Rivers e discutidos em capítulos seguintes.

2.1 Streams

Streams são definidos como uma sequência de dados que são produzidos assincronamente ao longo do tempo fluindo de sua fonte (conhecida como *upstream*) a um destino (chamado de *downstream*) (CASWELL, 2013). Frequentemente *streams* são comparados a coleções de dados como por exemplo *arrays* e listas. No entanto *streams*, diferentemente de coleções, não definem necessariamente um tamanho fixo, podendo produzir elementos indefinidamente ao longo do ciclo de vida de um programa. Apesar desta diferença, as APIs de *streams* são muitas vezes modeladas de maneira que se assemelham a APIs de manipulação de coleções compartilhando muitas das operações como *filter*, *map*, *reduce* etc.

Streams são amplamente aplicados no âmbito computacional no entanto em muitos casos seus conceitos não são explicitamente aparentes. Um exemplo clássico são os *Unix Pipes*, mecanismos utilizados para realizar comunicação entre processos (TANENBAUM, 2014) via canais de comunicação conhecidos como *standard input* e *output* ou simplesmente *stdin* e *stdout* respectivamente. Uma operação de *pipe* conecta o canal *stdout* de um programa ao *stdin* de outro permitindo com que dados sejam transmitidos de um lado ao outro passando por diferentes estágios de processamento específicos a cada programa. A figura 2.1 mostra os comandos Unix *find*, *grep* e *wc* sendo combinados via operações de *pipe* formando um *pipeline* com três estágios de processamento.

As aplicações de *streams* são tantas que muitas linguagens de programação adotam completamente os seus conceitos e disponibilizam APIs robustas que permitem a criação de *pipelines* complexos de processamento de stream de dados, alguns exemplos são as APIs de NodeJS (NODEJS, 2015), o pacote IO de da linguagem de programação Haskell (HASNKELL, 2015) e a API de streams de Java (JAVA8, 2015).

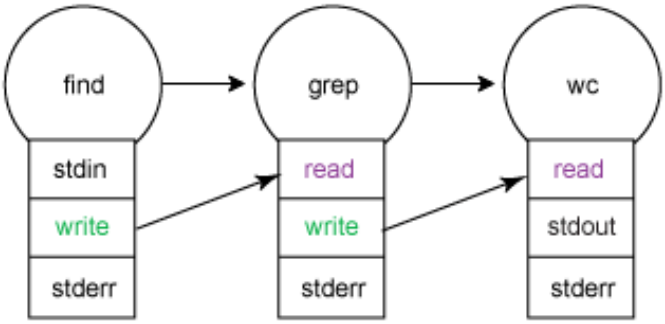


Figura 2.1 – Exemplo de stream pipeline em Unix.

2.2 A Linguagem Go

Go é uma linguagem de programação *open source*, estaticamente tipada (MEIJER, 2015) com suporte a *garbage collector* (MICROSYSTEMS, 2006) criada pela Google com foco em simplicidade, produtividade e concorrência. O sistema de tipos da linguagem aborda de maneira inovadora alguns conceitos como por exemplo interfaces (ORACLE, 2015) que difere do conceito tradicional implementado por linguagens como Java. Outro aspecto interessante da linguagem é seu modelo de concorrência baseado em *Goroutines* (GOLANG, 2015a) conceito similar ao de *Coroutines* (BERKELEY, 2015) e troca de mensagens (YAVATKAR, 2015) através do uso de canais de comunicação. A seguir serão apresentados alguns dos conceitos da linguagem que foram essenciais na implementação de Rivers.

2.2.1 Interfaces

Interface é o mecanismo através do qual sistemas podem ser modelados visando reuso e extensibilidade. Contratos abstratos são especificados descrevendo determinadas funcionalidades de um sistema independente de possíveis implementações, isso permite com que qualquer componente satisfazendo uma determinada interface possa ser utilizado como provedor da funcionalidade em questão. Em muitas linguagens de programação como Java, componentes precisam explicitamente declarar sua intenção de implementar uma interface e com isso sendo necessário definir as interfaces necessárias de um sistema como parte da modelagem da solução e consequentemente tornando a solução menos suscetível a futuras alterações de design. O código 2.2 mostra um exemplo de uma classe Java implementando uma interface específica.

```
1  public interface Movable {
2          public void moveTo(Point position);
3  }
4
5  public class Car implements Movable {
6          public void moveTo(Point position) {
7                  // Omitted code
8          }
9  }
```

Figura 2.2 – Exemplo de implementação de uma interface em Java.

Em Go, interfaces são satisfeitas implicitamente sem a necessidade de que componentes declarem explicitamente sua intenção de implementar uma interface eliminando assim hierarquias de dependências presentes em linguagens orientadas a objetos (PECINOVSKý, 2015). Para que um componente A satisfaça a interface B, A deve simplesmente implementar todos os métodos declarados em B. Desta maneira decisões arquiteturais de sistema podem ser tomadas em momentos futuros a medida em que novos casos de uso são introduzidos e padrões de código detectados evoluindo assim a arquitetura gradativamente. O código a seguir mostra como o exemplo Java em 2.2 pode ser reescrito em Go. Note que o tipo *Car* não possui qualquer dependência com a interface *Movable* ao contrário da versão Java, porém pode ser utilizado como um tipo *Movable* por implementar o método *MoveTo* na linha 7.

```
1  type Movable interface {
2          MoveTo(point Point)
3  }
4
5  type Car struct {}
6
7  func (car *Car) MoveTo(point Point) {
8          // Omitted code
9  }
```

Figura 2.3 – Exemplo de implementação de uma interface em Go.

Na seção 4.1 será discutido como Rivers faz o uso de interfaces para definir os building blocks do framework permitindo que novas funcionalidades sejam introduzidas à arquitetura de maneira transparente e não intrusiva.

2.2.2 Goroutines

Goroutine é o mecanismo que a linguagem Go utiliza para executar código de maneira concorrente e potencialmente em paralelo de maneira similar a outros mecanismos como por exemplo *coroutines* (BERKELEY, 2015) e *threads* (JOSEPH, 2003) porém com algumas diferenças que fazem com que seu papel no modelo de concorrência da linguagem seja crucial.

Goroutines são basicamente funções que executam assincronamente e concorrentemente utilizando o comando go e são gerenciadas pelo *runtime* da linguagem. Ao contrário de *threads* que exigem pelo menos 1MB de memória inicial para inicialização de sua pilha de execução, *Goroutines* necessitam apenas de 2KB de memória para sua inicialização podendo ajustar este valor sob demanda alocando e desalocando espaço na *Heap* (BOOTCAMP, 2012), permitindo com que centenas de milhares de *Goroutines* possam ser executadas concorrentemente. Além disso o custo de troca de contexto entre *Goroutines* é extremamente baixo comparado com *threads* uma vez que *Goroutines* são gerenciadas pelo *runtime* da linguagem Go e ao contrário de *threads* não necessitam acessar recursos do Sistema Operacional para realizar a troca de contexto. (KRISHNA, 2014) faz uma excelente análise do funcionamento de *Goroutines* em comparação com o funcionamento de OS *threads*. A figura 2.4 mostra uma *Goroutine* sendo criada na linha 16 utilizando o comando *go* para executar assincronamente a função *say* com o parâmetro "world".

```
1  package main
2
3  import (
4          "fmt"
5          "time"
6  )
7
8  func say(s string) {
9          for i := 0; i < 5; i++ {
10                 time.Sleep(100 * time.Millisecond)
11                 fmt.Println(s)
12         }
13 }
14
15 func main() {
16         go say("world")
17         say("hello")
18 }
```

Figura 2.4 – Exemplo de uso de Goroutines.

2.2.3 Canais de Comunicação

Um canal de comunicação é o mecanismo básico utilizado para realizar a comunicação entre *Goroutines* via troca de mensagem e a sincronização de suas execuções permitindo que dados sejam enviados de uma *Goroutine* à outra de maneira segura sem a necessidade de compartilhar memória. Canais são a base para o modelo *Produtor-Consumidor* (KOCHER, 2005) utilizado como fundação na implementação de Rivers.

Dados são escritos e lidos de um canal de maneira síncrona sendo desnecessário o uso de

outras primitivas de concorrência como por exemplo *semáforos* ou *mutex* (SCHMIDT, 2015). Por padrão uma *Goroutine* ao escrever um dado em um canal bloqueia sua execução até que outra *Goroutine* leia este dado, liberando espaço para que outro dado seja escrito. No entanto um canal pode ser criado com um *Buffer* permitindo com que vários dados possam ser escritos sem que sejam consumidos, bloqueando então apenas quando não houver mais espaço no *Buffer*. Canais podem ser de somente escrita, somente leitura ou ambos permitindo implementar alguns padrões de design interessantes como por exemplo *Pipeline Pattern* (GOLANG, 2015b). A figura 2.5 mostra um exemplo de criação de um canal com um *Buffer* de capacidade 2 (linha 2) e duas mensagens enviadas nas linhas 3 e 4 e logo em seguida consumidas nas linhas 6 e 7.

```
1  func main() {
2    messages := make(chan string, 2)
3    messages <- "Hello "
4    messages <- "World!"
5
6    msg1 := <- messages
7    msg2 := <- messages
8
9    print(msg1)
10   print(msg2)
11 }
```

Figura 2.5 – Exemplo de uso de canais de comunicação.

3 TRABALHOS RELACIONADOS

A seguir serão descritos brevemente alguns projetos que inspiraram a criação de Rivers não necessariamente relacionados à linguagem Go mas que aplicam conceitos similares para processamento de *streams*.

3.1 Apache Spark

Apache Spark (APACHE, 2015) é uma solução para computação de tempo real, tolerante a falhas e com suporte a *clustering* (BUYTAERT, 2004), com APIs em diferentes linguagens de programação como por exemplo *scala*, *java* e *python*. *Spark* tornou-se muito popular em áreas de *Machine Learning* (NG, 2015) e *Analytics* (ROUSE, 2008) devido ao seu grande poder computacional que permite que grandes volumes de dados possam ser processados e distribuídos em diferentes máquinas de um *cluster* sendo uma solução muito atrativa também para sistemas de processamento de eventos (STREAMING, 2015).

```
val textFile = spark.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

Figura 3.1 – Exemplo de uso da API scala de Spark.

A API de Rivers é fortemente inspirada na fluência da API *Spark*, aplicando *Method Chaining* (FOWLER, 2015b) sempre que possível aumentando a legibilidade do código resultante. Ambas APIs aplicam ao máximo os conceitos de programação funcional disponibilizando operações de filtros, mapeadores e agregadores de dados provendo um considerável nível de composição entre operações permitindo a criação de pipelines complexos de processamento de *streams* fortemente suscetíveis ao reuso. Rivers ao contrário de *Spark* propõe uma solução mais simples a nível de linguagem não sendo necessário lidar com a complexidade de se manter um *cluster* de máquinas para criação de um pipeline de processamento. Suporte a *clustering* em Rivers é proposto como parte de trabalhos futuros.

3.2 Java 8 Streams

A versão 8 da linguagem de programação Java introduz a abstração de *streams* (JAVA8, 2015) como parte de suas bibliotecas padrão e assim como outras linguagens e plataformas como *Spark*, disponibiliza uma API fluente baseada em conceitos de programação funcional, permitindo a criação de *pipelines* para processamento de *streams* com suporte a paralelização.

Assim como *Spark*, a API de *streams* de Java contribuíram para o design da API de Rivers a fim de prover uma API similar para a linguagem Go que não fosse somente simples porém familiar a desenvolvedores com conhecimentos de programação funcional e extensível permitindo que a solução fosse aplicada a novos casos de uso introduzindo implementações diferentes dos componentes básicos do *framework* 4.2.

```
List<Integer> transactionsIds =
    transactions.parallelStream()
                .filter(t -> t.getType() == Transaction.GROCERY)
                .sorted(comparing(Transaction::getValue).reversed())
                .map(Transaction::getId)
                .collect(toList());
```

Figura 3.2 – Exemplo de uso da API de streams em Java8.

3.3 Reactive eXtensions

Reactive eXtensions (RX) (REACTIVEX, 2015b) é um movimento que tomou tração nos últimos anos liderado por empresas como (NETFLIX, 2015) e (MICROSOFT, 2015) dentre outras que procuram tornar *mainstream* conceitos de programação reativa (ODERSKY, 2013) empregando vários conceitos de programação funcional e conhecidos padrões de design como *Observable Pattern* (REACTIVEX, 2015a) na criação de APIs assíncronas para tratamento de fluxo de dados. Estes conceitos quando combinados permitem com que APIs poderosas sejam implementadas abstraindo muito dos desafios envolvidos na programação assíncrona como por exemplo sincronização de *threads*, estruturas de dados concorrentes e operações de IO não bloqueantes. Existem várias implementações da especificação dentre elas estão a API Java (JAVARX, 2015) e Javascript (RXJS, 2015).

Apesar de Rivers não seguir necessariamente a especificação de *Reactive eXtensions*, muitas decisões de design em Rivers foram baseados em conceitos similares aos utilizados na especificação de RX, como por exemplo realização de *back-pressure* (CASWELL, 2013) do *pipeline* através da utilização de Go *buffered channels* (SEGUIN, 2013). *Reactive eXtensions* e outros projetos explorados neste capítulo mostram que processamento de *streams* é uma solução interessante para muitos problemas relacionados a diversas áreas tecnológicas como por exemplo *Big Data*, *Analytics*, *Event Processing* e muitas linguagens de programação adotaram estes conceitos provendo APIs nativas que permitem com que pipeline de processamento de *streams* sejam criados de maneira simples abstraindo muito dos desafios envolvidos com relação a concorrência e paralelização de um *pipeline* por exemplo, motivando a criação de Rivers.

4 RIVERS

Neste capítulo é apresentada a API Rivers, sua arquitetura juntamente com seus principais elementos e características que compõem a solução.

4.1 Arquitetura

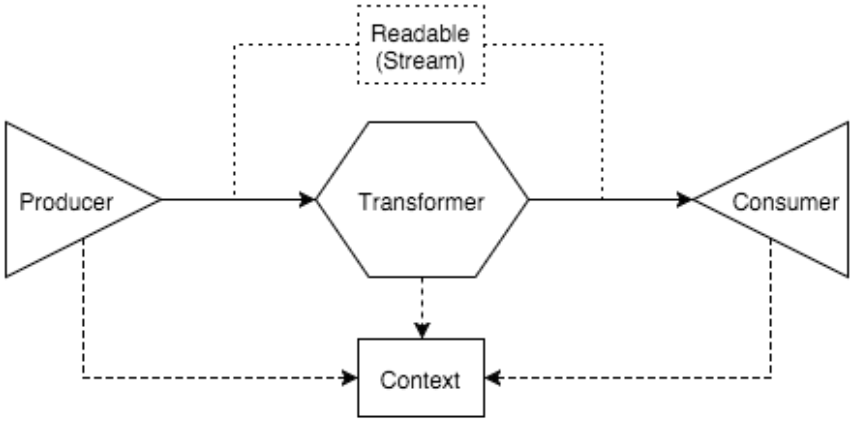
A arquitetura Rivers foi modelada com o intuito de prover uma API simples, flexível e extensível para a criação de *pipelines* complexos de processamento de *stream* de dados, tendo como fundação princípios do modelo produtor-consumidor aplicados ao modelo de concorrência da linguagem Go juntamente com o *design pattern Pipeline* e conceitos de programação funcional. Rivers faz uso de *Goroutines* para o processamento concorrente e assíncrono de cada estágio do *pipeline* e canais para realizar a comunicação entre os mesmos. A fim de promover reuso e extensibilidade a API é definida em termos de Go interfaces disponibilizadas no pacote *stream* juntamente com componentes pré-definidos que podem ser combinados para criação de *pipelines* desde os mais simples até os mais complexos com mecanismos de *fork* e *join* por exemplo.

Pipelines em Rivers frequentemente são compostos por um estágio inicial conhecido como *Producer* (ver seção 4.2.3) responsável por gerar assincronamente os dados a serem processados. Opcionalmente, um *pipeline* pode ter um ou mais estágios intermediários conhecidos como *Transformers* (ver seção 4.2.4) responsáveis por transformar os dados que passam por eles aplicando funções tais como *filter* e *flatmap* disponibilizando o resultado em um *stream* de leitura para que o próximo estágio possa consumi-lo de maneira assíncrona. O último estágio de um *pipeline* é representado por um *Consumer* (ver seção 4.2.5) que bloqueia a execução do programa até que todos os dados sejam consumidos e o *pipeline* encerrado. Este último estágio ao fim da execução coleta e retorna qualquer eventual erro durante a execução para que o mesmo possa ser tratá-lo apropriadamente. Cada estágio do *pipeline* é conectado sequencialmente a um estágio seguinte utilizando *streams* de escrita e leitura conhecidos respectivamente por *Writable* e *Readable*, discutidos na seção 4.2.1. Estágios produzindo dados como *producers* e *transformers* escrevem estes valores em um *stream* de escrita, disponibilizando a versão de leitura deste mesmo *stream* a um próximo estágio para que este possa consumi-lo de maneira assíncrona.

Estágios do *pipeline* compartilham um mesmo contexto de execução utilizado para coordenar o fluxo de dados e sinalizar o término prematuro do *pipeline* devido a uma operação de *short-circuit* (CS-132-WATERLOO, 2005) ou devido a falhas ocorridas em algum ponto da execução. Este mecanismo permite com que cada estágio verifique o estado atual do *pipeline* antes de iniciar qualquer processamento podendo finalizar sua execução caso o *pipeline* tenha sido encerrado. Em situações aonde o contexto de execução é finalizado devido a erros, cada es-

tágio do *pipeline* deve garantir que qualquer recurso sendo utilizado seja liberado corretamente como descritores de arquivos, canais de comunicação, conexões com base de dados, etc. Um estágio pode requisitar o término do contexto de execução devido a uma falha ou pelo término prematuro como mencionado nos casos de operações de *short-circuit*. Este mecanismo permite que *dowstreams* do pipeline possam notificar *upstreams* com relação ao término da execução finalizando a produção de dados.

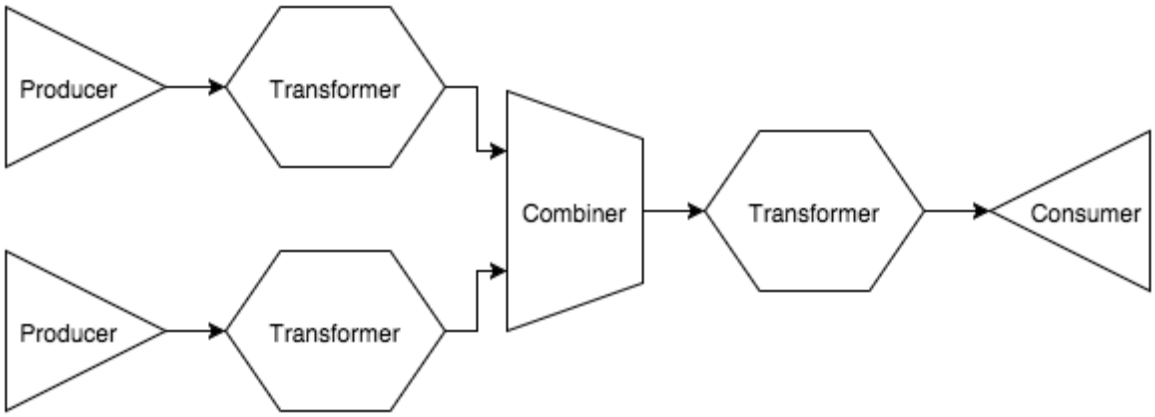
O diagrama 4.1 mostra a relação entre estes componentes de um *pipeline* e sua representação equivalente em código Go. Em um *pipeline* Rivers o fluxo de dados sempre parte de um *Producer* a um *Consumer*, podendo existir entre eles um ou mais *Transformers*, estágios intermediários de transformação de dados. Cada um destes componentes do *pipeline* são representados no diagrama pelos triângulos da esquerda, direita e hexágono respectivamente. Estágios do pipeline são conectados entre si através de *streams* de leitura, representados pelas setas sólidas no diagrama estabelecendo assim o fluxo de dados direcional da esquerda para a direita. Como representado no diagrama, uma única instância do contexto de execução é compartilhada entre os estágios do *pipeline* responsável por supervisionar e monitorar a execução.




```
rivers.From(producer).Apply(transformer).Then(consumer)
```

Figura 4.1 – Componentes Básicos de um Pipeline Rivers.

Pipelines podem assumir estruturas bem complexas como mencionado anteriormente. Rivers provê mecanismos para combinar múltiplos *streams* assim como duplicar um *stream* em vários outros. Estes mecanismos são conhecidos como *Combiners* e *Dispatchers* respectivamente e permitem a construção de *pipelines* mais complexos com vários fluxos concorrentes de processamento. *Combiners* são muito convenientes em situações aonde dados são produzidos por fontes de dados diferentes porém devem ser processados por uma mesma sequência de operações. O diagrama 4.2 representa dois *pipelines* sendo combinados em um único utilizando um *Combiner* e seu resultado sendo processado por um *Transformer* e em seguida consumido por um *Consumer*. Diferentes algoritmos podem ser aplicados ao se combinar *streams* os quais são discutidas em maiores detalhes na seção 4.2.6.



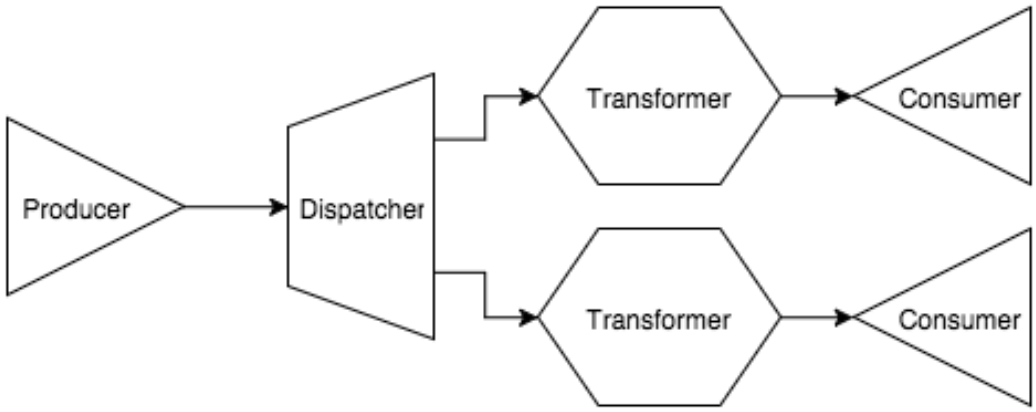

```
employees := rivers.FromFile("/tmp/employees.csv").ByLine().Map(toEmployee)
employers := rivers.FromFile("/tmp/employers.csv").ByLine().Map(toEmployer)

err := rivers.Merge(employees, employers).Each(saveEntities(database)).Then(consumer)
```

Figura 4.2 – Exemplo de Uso de Combiner.

Um *Dispatcher* por sua vez permite com que dados de um *stream* possam ser condicionalmente classificados e processados separadamente por uma sequência de operações distintas no pipeline. Rivers fornece algumas implementações de *Dispatchers* como por exemplo as operações *Partition* e *Split*. A primeira particiona o *stream* em dois outros baseado em um predicado que é aplicado a cada elemento do *stream* retornando dois novos *streams*. Elementos que atendem o predicados são redirecionados ao primeiro *stream* e o restante redirecionados ao segundo. A operação de *Split* simplesmente duplica o *stream* em dois outros idênticos permitindo que diferentes sequências de operações possam ser aplicadas a um mesmo elemento concorrentemente.

A figura 4.3 representa um *pipeline* utilizando um componente *Dispatcher* que particiona o *stream* de números inteiros criado pelo *Producer* aplicando o predicado *ByEvensAndOdds* a cada elemento. Como resultado dois novos *streams* são criados, o primeiro contendo apenas números pares e o segundos apenas números ímpares, cada *stream* é então concorrentemente processado por um *Transformer* e consumido por um *Consumer*.




```
evens, odds := rivers.FromRange(1, 100).Partition(ByEvensAndOdds)

evens.Each(processEvens).Then(consumer)
odds.Each(processOdds).Then(consumer)
```

Figura 4.3 – Exemplo de Uso de Dispatcher.

O design aplicado na implementação de Rivers visa disponibilizar uma API fluente (FOWLER, 2005) similar a soluções encontradas em outras plataformas como *Spark* 3.1 e na API de *streams* de Java 8 3.2. Este estilo de API incentiva a composição de pequenos componentes com responsabilidades bem específicas a fim de resolver um problema maior. O conceito de composição é bem comum no contexto de programação funcional e se encaixa de maneira muito conveniente no contexto de processamento de *streams*. Operações tais como *filter*, *take*, *flatmap*, *reduce*, etc permitem com que *pipelines* complexos sejam criados sem a necessidade de estender a API diretamente. Nos casos em que as operações nativas não sejam suficientes é possível implementar interfaces específicas da API para introduzir novos componentes que se comportam como por exemplo *Producers*, *Transformers*, *Consumers* ou qualquer outra interface disponível na API.

4.2 Building Blocks

Nesta seção são introduzidos os componentes básicos da arquitetura Rivers utilizados na criação de *pipeline* de processamento de *streams*.

4.2.1 Streams

Streams em Rivers comportam-se como *Unix Pipes* (YU, 2015). Eles possuem uma extremidade inicial aonde dados podem ser escritos por um *Producer* ou *Transformer* por exemplo e uma extremidade final de onde dados podem ser lidos, geralmente por um *Transformer* ou *Consumer*. *Streams* são os conectores que permitem com que diferentes componentes do

sistema sejam combinados na criação de *pipelines* e é por onde os dados são transmitidos de um estágio a outro podendo haver um *buffer* entre eles. No contexto de *streams* estes estágios são conhecidos como *upstream* e *downstream* respectivamente.

Em situações em que o *buffer* de um *stream* esteja cheio, assim como no modelo Produtor-Consumidor o componente produzindo dados é bloqueado até que pelo menos um item seja consumido do *buffer*. De maneira análoga se o *buffer* estiver vazio o consumidor é bloqueado até que um novo item seja produzido e adicionado ao *buffer*. No contexto de *streams* este mecanismo bloqueante é conhecido como *back-pressure* (CASWELL, 2013) e é todo ele gerenciado pelo *runtime* da linguagem Go que também detecta situações de *deadlock* (INGALLS, 2004) encerrando a execução do programa quando necessário. A figura 4.4 ilustra a criação de um *stream* com um *buffer* de capacidade 10 e um único item é escrito e em seguida lido utilizando as componentes *writable* e *readable* do *stream*:

```
readable, writable := stream.New(10)
writable <- 1
data := <-readable // data == 1
```

Figura 4.4 – Rivers Stream.

4.2.2 Attachables

Componentes de um pipeline em Rivers precisam satisfazer a interface *Attachable* 4.5 o que permite com que o contexto de execução atual do componente em questão seja devidamente configurado no momento em que o componente é conectado ao *pipeline*. É importante que os componentes de um *pipeline* performem sob um mesmo contexto de execução uma vez que este contexto é usado para sinalizar o término da execução devido a erros ou operações de *short-circuit* permitindo com que cada componente finalize sua execução apropriadamente.

```
type Attachable interface {
        Attach(Context)
}
```

Figura 4.5 – Attachable Interface.

4.2.3 Producers

Producers são responsáveis pela produção de dados assincronamente disponibilizando-os em um *stream* de leitura que é consumido pelo estágio seguinte do *pipeline*, aplicando o *Pipeline Pattern* (GOLANG, 2015b). Para utilizar um tipo como produtor de dados em um *pipeline* Rivers este deve implementar a interface *Producer* 4.6 do pacote *stream*, note que um *Producer* deve satisfazer também a interface *Attachable*:

```
type Producer interface {
        Attachable
        Produce() (out Readable)
}
```

Figura 4.6 – Producer Interface.

Para que uma implementação de *Producer* possa ser utilizada em um *pipeline* Rivers de maneira efetiva, é necessário que o seguinte contrato seja satisfeito:

1. Implementar a interface *stream.Producer*;
2. Implementar o *Pipeline Pattern* como parte do método *Produce*;
3. Como parte da *Goroutine* produzindo dados: executar em modo *defer* a função *Recover* do contexto;
4. Fechar a componente *writable* do *stream* uma vez que a produção de dados é encerrada;
5. Finalizar a *Goroutine* no caso em que o canal *Done* ou *Failure* do contexto forem fechados.

A figura 4.7 mostra uma implementação de *Producer* que contempla o contrato anterior, produzindo números inteiros entre um intervalo pré-definido:

```
1  type RangeProducer struct {
2          context  stream.Context
3          Capacity int
4          from, to int
5  }
6
7  func (producer *RangeProducer) Attach(context stream.Context) {
8          producer.context = context
9  }
10
11 func (producer *RangeProducer) Produce() stream.Readable {
12         readable, writable := stream.New(producer.to - producer.from)
13
14         go func() {
15                 defer producer.context.Recover()
16                 defer close(writable)
17
18                 for i := producer.from; i < producer.to; i++ {
19                         select {
20                         case <-producer.context.Done():
21                                 return
22                         case <-producer.context.Failure():
23                                 return
24                         default:
25                                 writable <- i
26                         }
27                 }
28         }()
29
30         return readable
31 }
```

Figura 4.7 – Implementação de Producer que gera números inteiros entre um intervalo pré-definido.

Analisando a implementação de *RangeProducer* é possível verificar que o contrato anterior é implementado corretamente. Cada item do contrato é discutido a seguir:

1. Implementando os métodos *Attach* e *Produce* a interface *stream.Producer* é satisfeita;
2. O *Pipeline Pattern* é aplicado da seguinte maneira: o método *Produce* cria um *stream* na linha 12 com capacidade igual ao quantidade de dados gerados, disparando uma *Goroutine* na linha 14 que assincronamente escreve dados no *stream* através da componente *writable* e por fim retorna a componente *readable* do *stream* na linha 30 permitindo com que o próximo estágio do *pipeline* possa eventualmente consumir os dados produzidos.
3. Na linha 15 a função *Recover* do contexto é executada em modo *defer*. Isto permite com que o contexto capture e trate qualquer possível erro fatal na execução da *Goroutine* antes que a mesma seja encerrada. Como parte da lógica de *Recover*, o contexto fecha o canal emphFailure sinalizando a falha a todos os componentes do *pipeline* que verificam

o estado de falha ao checar o fechamento deste canal em um bloco *select*, linha 22.

4. Na linha 16 o fechamento da componente *writable* do *stream* é executado em modo *defer*. Esse passo garante que o *stream* será fechado mesmo em situações de erro fatal ocorridos como parte da execução da *Goroutine*.
5. A execução da *Goroutine* é encerrada caso algum dos canais *Done* linha 21 ou *Failure* linha 23 forem fechados. Este mecanismo de parada faz uso de uma propriedade de canais muito importante que diz que todo canal fechado está pronto para receber (ver 2.2.3) o que faz com que o caso em particular seja selecionado no bloco *select*. O fechamento do canal *Done* indica que algum estágio do *pipeline* requisitou o término da execução por ter finalizado corretamente seu processamento. Esse comportamento pode ser verificado em operações de *short-circuit* como *Find*, *Any* que podem requisitar o término da execução por terem encontrado o resultado o final mesmo que ainda existam dados a serem produzidos. O fechamento do canal *Failure* por sua vez indica que algum estágio do *pipeline* encerrou prematuramente devido a uma falha fatal que foi capturada e tratada pela função de *Recover* do contexto.

Este simples contrato permite com que diversos tipos de *Producers* sejam criados, simples como o exemplo anterior assim como mais complexos como por exemplo *producers* que geram dados de um *socket*, linhas de um arquivo ou até mesmo dados de uma API fornecidos através de um *stream* HTTP.

A fim de reduzir o *boilerplate* necessário na implementação de um *Producer*, Rivers provê o componente *Observable* do pacote *producers* que satisfaz por completo o contrato anterior reduzindo consideravelmente o número de linhas necessárias na implementação de um *Producer*. A implementação de *RangeProducer* pode ser reescrita em termos de um *producers.Observable* como mostrado na figura 4.8:

```
1  func FromRange(from, to int) stream.Producer {
2          return &producers.Observable{
3                  Capacity: to - from,
4                  Emit: func(emitter stream.Emitter) {
5                          for i := from; i <= to; i++ {
6                                  emitter.Emit(i)
7                          }
8                  },
9          }
10 }
```

Figura 4.8 – Implementação de RangeProducer em termos de producers.Observable.

Todos os detalhes do contrato são encapsulados e abstraídos pelo componente *Observable*. A função *Emit* linha 4 é executada pelo próprio *Observable* para dar início o processo de produção dos dados e o componente *emitter* passado como parâmetro da função é utilizado

para emitir os dados produzidos no *stream*, disponibilizando-os ao próximo estágio do *pipeline*. Rivers também disponibiliza algumas implementações pré-definidas de *producers* para processamento de listas, objetos, arquivos e *sockets*.

4.2.4 Transformers

Transformers são estágios intermediários de um *pipeline* especializados em transformação de dados. A figura 4.9 mostra a interface implementada por um *Transformer*.

```
type Transformer interface {
        Attachable
        Transform(in Readable) (out Readable)
}
```

Figura 4.9 – Transformer Interface.

Uma vez conectados a um *pipeline*, *Transformers* aplicam sua função de transformação aos dados a medida que passam pelo estágio e seu resultado enviado ao próximo estágio do pipeline. Este processamento é assíncrono e implementado aplicando mais uma vez o *Pipeline Pattern*.

Em geral *transformers* são especializados e responsáveis por exercer uma função em específica no contexto de um *pipeline*. Vários *transformers* podem ser combinados para realizar lógicas de processamento mais complexas, como por exemplo uma série de filtros seguidos por *transformers* que acessam dados de uma API, ou banco de dados. Rivers disponibiliza várias funções de transformação pré-definidas como por exemplo *Filter*, *Map*, *Each* e outros.

Implementações de *Transformers* assim como *Producers* devem satisfazer um contrato que descreve os pontos necessários que uma implementação deve seguir. Estes pontos são descritos a seguir:

1. Implementar a interface *stream.Transformer*;
2. Implementar o *Pipeline Pattern* como parte do método *Transform*;
3. Como parte da *Goroutine* transformando dados: executar em modo *defer* a função *Recover* do contexto;
4. Fechar a componente *writable* do *stream* uma vez que a transformação de dados é encerrada;
5. Finalizar a *Goroutine* no caso em que o canal *Done* ou *Failure* do contexto forem fechados;
6. Finalizar a *Gouroutine* caso não haja mais dados a serem consumidos do *upstream*.

As motivações para cada item do contrato são similares as descritas com relação a implementação de um *Producer*. A figura 4.10 implementa um *Transformer* que permite com que apenas números pares sejam enviados ao estágio seguinte do *pipeline*, e a figura 4.11:

```
1  type FilterTransformer struct {
2          context stream.Context
3  }
4
5  func (t *FilterTransformer) Attach(context stream.Context) {
6          t.context = context
7  }
8
9  func (t *FilterTransformer) Transform(in stream.Readable) stream.Readable {
10         readable, writable := stream.New(in.Capacity())
11         even := func(data stream.T) bool { return data.(int) % 2 == 0 }
12
13         go func() {
14                 defer t.context.Recover()
15                 defer close(writable)
16
17                 for {
18                         select {
19                         case <-t.context.Failure():
20                                 return
21                         case <-t.context.Done():
22                                 return
23                         default:
24                                 data, more := <-in
25                                 if !more {
26                                         return
27                                 }
28
29                                 if even(data) {
30                                         writable <- data
31                                 }
32                         }
33                 }
34         }()
35
36         return readable
37 }
```

Figura 4.10 – Filter Transformer.

```
rivers.FromRange(1, 10).Apply(&FilterTransformer{}).Then(consumer)
```

Figura 4.11 – Uso do Filter Transformer em um pipeline Rivers.

Implementações de *transformers* e *producers* são relativamente similares, com algumas restrições:

1. Na linha 25 da implementação, a execução do *transformer* é finalizada caso não existam mais dados a serem consumidos do *stream* sendo transformado, neste caso o parâmetro in da função *Transform*;

2. A operação de filtro é aplicada na linha 29 e dados que satisfazem a condição do filtro, neste caso números pares, são enviados ao estágio seguinte do *pipeline*.

Rivers disponibiliza o tipo *Observer* do pacote *transformers* que encapsula e abstrai cada ponto do contrato mencionado anteriormente e pode ser usado como base da implementação de novos *transformers*, a figura seguinte reescreve o filtro anterior em termos de um *Observer*, e seu uso em um *pipeline* Rivers é mostrado na figura 4.13:

```
1  func EvensOnly() stream.Transformer {
2          even := func(data stream.T) bool {
3                  return data.(int) % 2 == 0
4          }
5
6          return &transformers.Observer{
7                  OnNext: func(data stream.T, emitter stream.Emitter) error {
8                          if even(data) {
9                                  emitter.Emit(data)
10                         }
11                         return nil
12                 },
13         }
14 }
```

Figura 4.12 – Filter Transformer em termos de um Observer

```
rivers.FromRange(1, 10).Apply(EvensOnly()).Then(consumer)
```

Figura 4.13 – Uso do EvensOnly Filter em um pipeline Rivers.

Cada dado que chega ao *Transformer* causa com que o método *OnNext* do *Observer* seja invocado com dois parâmetros, o dado em si e uma instância de *stream.Emitter* que pode ser usado para enviar dados que satisfazem o filtro ao estágio seguinte do *pipeline*. Caso o método *OnNext* retorne um erro o *pipeline* é encerrado e o contexto sinaliza o erro a todos os estágios fechando o canal *Failure*.

Filtros são operações muito úteis no processamento de streams e são essencialmente especializações de um Transformer. Por este motivo Rivers provê como parte da API mecanismos para se aplicar filtros em um stream de maneira extremamente simples, sem a necessidade de se implementar um novo Transformer. A figura 4.14 mostra a aplicação de algumas implementações de filtros existentes na API.

```
even := func(data stream.T) bool {
  return data.(int) % 2 == 0
}

rivers.FromRange(1, 10).Filter(evensOnly).Then(consumer)
rivers.FromRange(1, 10).Take(evensOnly).Then(consumer)
rivers.FromRange(1, 10).TakeFirst(evensOnly).Then(consumer)
rivers.FromRange(1, 10).Drop(evensOnly).Then(consumer)
rivers.FromRange(1, 10).DropFirst(evensOnly).Then(consumer)
```

Figura 4.14 – Exemplos de uso de operações de filtros.

4.2.5 Consumers

Consumers são componentes responsáveis por consumir de maneira síncrona dados de um *Readable stream* e representam o estágio final do *pipeline*, eles garantem com que o programa termine apenas ao final da execução do *pipeline* e reportam qualquer eventual erro de execução do mesmo. Assim como qualquer componente de um *pipeline* em Rivers, *consumers* são *Attachables* podendo ser conectados ao contexto de um *pipeline* em tempo de execução. A figura 4.15 mostra a interface implementada por um *Consumer*.

```
type Consumer interface {
        Attachable
        Consume(in Readable)
}
```

Figura 4.15 – Consumer Interface.

Consumers podem ser utilizados para realizar operações de agregamento de valores como por exemplo *Count*, que calcula o número total de dados que passam pelo *Consumer* até o final da execução do *pipeline*. Outra implementação útil de *Consumer* são os conhecidos *Collectors*, componentes responsáveis por coletar os dados que passam pelo *consumer* para serem utilizados ao final da execução do *pipeline*. A figura 4.16 mostra algumas aplicações de *consumers* em Rivers.

```go
var numbers []int
var first int
var last int

items, err := rivers.FromRange(1, 100).Collect()
first, err := rivers.FromRange(1, 100).CollectFirst()
last, err := rivers.FromRange(1, 100).CollectLast()

err := rivers.FromRange(1, 100).CollectAs(&numbers)
err := rivers.FromRange(1, 100).CollectFirstAs(&first)
err := rivers.FromRange(1, 100).CollectLastAs(&last)
err := rivers.FromRange(1, 100).CollectBy(&collectFunc)

err := rivers.FromRange(1, 100).SortBy(sortingFunc)
err := rivers.FromRange(1, 100).GroupBy(groupingFunc)
err := rivers.FromRange(1, 100).Drain()
```

Figura 4.16 – Aplicação de consumers em Rivers.

Consumers ao contrário dos outros componentes do *pipeline* não implementam o *Pipeline Pattern* uma vez que estes representam estágios síncronos do *pipeline* e não produzem *stream* de dados como resultado. A figura seguir representa uma implementação simples de *Consumer* que realiza a operação de *Count* mencionada anteriormente:

```go
type Counter struct {
        context stream.Context
        Count   int
}

func (counter *Counter) Attach(context stream.Context) {
        counter.context = context
}

func (counter *Counter) Consume(in stream.Readable) {
        defer counter.context.Recover()

        for {
                select {
                case <-counter.context.Failure():
                        return
                case data, more := <-in:
                        if !more {
                                return
                        }
                        counter.Count++
                }
        }
}
```

Figura 4.17 – Implementação do Consumer Count.

O contrato que uma implementação de *Consumer* deve seguir é relativamente simples,

composto por basicamente três passos:

1. Executar em modo *defer* a função *Recover* do contexto como parte da implementação do método *Consume*;
2. Finalizar a execução caso o canal *Failure* do contexto foi fechado;
3. Consumir dados do *Readable stream* até que o *stream* seja encerrado.

Apesar de simples este contrato acaba por se repetir em diversas implementações de *consumers*, por isso Rivers disponibiliza o tipo *Sink* do pacote *consumers* que pode ser utilizado como base na implementação de novos *consumers*. A implementação do *Consumer Count* pode ser reescrita em termos de um tipo *Sink* como mostrado a seguir:

```
1 func NewCounter(count *int) stream.Consumer {
2         return &consumers.Sink{
3                 OnNext: func(data stream.T) {
4                         (*count)++
5                 },
6         }
7 }
```

Figura 4.18 – Implementação do Consumer utilizando o tipo Sink como base.

4.2.6 Combiners

Combiners são mecanismos utilizados para combinar assincronamente diferentes *streams* de dados em um único *stream* que pode então ser processado por estágios seguintes do *pipeline*. Um caso de uso seria combinar diferentes fontes de dados em um único *stream*, como por exemplo o resultado de uma requisição à uma API *restful* e uma busca a um banco de dados. Os dados produzidos por cada uma dessas fontes após serem combinados em um único *Readable stream*, podem ser processados pela mesma sequência de estágios. A figura a seguir representa a interface implementada por um *Cobiner* em Rivers:

```
type Combiner interface {
        Attachable
        Combine(in ...Readable) (out Readable)
}
```

Figura 4.19 – Combiner Interface.

Rivers API além de permitir com que qualquer tipo que satisfaça a interface acima possa ser utilizado como um Combiner em um pipeline, algumas implementações úteis de combiners são nativas da API e prontas para serem utilizadas como por exemplo Merge que combina dados de fontes diferentes em uma política FIFO aonde os primeiros dados que chegam de qualquer uma das fontes são imediatamente enviados para o stream final. Outra política de junção de

streams é a de Zip que coleta um item de cada fonte alternadamente até que todos os dados sejam combinados, dentre outras implementações.

```
ages := rivers.FromData(28, 29, 30)
names := rivers.FromData("Joe", "Alice", "Bob")

agesAndNames, err := ages.Zip(names).Collect()
// agesAndNames == []stream.T{28, "Joe", 29, "Alice", 30, "Bob"}
```

Figura 4.20 – Zip Combiner Interface.

4.2.7 Dispatchers

Dispatchers são utilizados para assincronamente redirecionar o fluxo de dados de um *Readable stream* a um ou mais *Writable streams* particionando o *pipeline* em vários ramos. Em casos mais complexos, *Dispatchers* podem implementar diferentes lógicas de redirecionamento utilizando funções de classificação conhecidas como predicados, nestes casos os dados que não satisfazerem o predicado são redirecionados a um *Readable stream* resultante da operação. A figura 4.21 representa a interface implementada por um *Dispatcher*.

```
type Dispatcher interface {
        Attachable
        Dispatch(from Readable, to ...Writable) (out Readable)
}
```

Figura 4.21 – Dispatcher Interface.

Em particular *dispatchers* são úteis para classificar e agrupar dados em *streams* diferentes podendo processar cada *stream* utilizando lógicas específicas. Operações de partição são exemplos clássicos de *Dispatchers* em Rivers. A figura a seguir mostra um *stream* de números inteiros sendo particionado em dois outros *streams*, o primeiro contendo apenas números pares e o segundo números ímpares.

```
evensAndOdds := func(data stream.T) bool {
        return data.(int) % 2 == 0
}

evens, odds := rivers.FromRange(1, 10).Partition(evensAndOdds)
```

Figura 4.22 – Exemplo de particionamento de um stream de números inteiros.

Rivers API disponibiliza algumas implementações de *dispatchers* pré-definidas além da mencionanda anteriormente, como por exmplo *Split* que duplica um *stream* em dois outros idênticos útil para aplicar diferentes sequências de transformações concorrentes aos dados do

stream como por exemplo salvar entidades na base de dados e concorrentemente indexar a informação em uma *engine* de busca ou salvar a informação em uma *cache*.

4.2.8 O Contexto Global

Coordenar vários estágios concorrentes de um *pipeline* e garantir o término da execução mesmo na presença de erros são algumas das funções do contexto o qual é compartilhado por cada componente conectado ao *pipeline*. Um contexto implementa alguns mecanismos utilizados por cada componente descrito até o momento que permitem com que cada um deles possam sinalizar o término da execução devido a uma falha por exemplo, permitindo com que qualquer *Goroutine* em execução possa suspender seu processamento verificando o canal *Failure* do contexto. Em casos em que um estágio do *pipeline* falhe causando uma situação de *panic*, o programa não finaliza inesperadamente uma vez que como descrito nos contratos de cada componente, a função de *Recover* do contexto é executada em modo *defer* como parte do processamento permitindo que o contexto capture o erro e sinalize a falha fechando o canal *Failure* causando com que cada componente suspenda sua execução liberando qualquer recurso de máquina utilizando até o momento, como conexões com base de dados, descritores de arquivos, etc.

```
type Context interface {
        Close(err error)
        Recover()
        Err() error
        Failure() <-chan struct{}
        Done() <-chan struct{}
}
```

Figura 4.23 – Context Interface.

A figura acima representa a interface implementada por um contexto em Rivers. Qualquer componente conectado ao *pipeline* pode verificar o estado atual do *pipeline* através dos canais de comunicação *Failure* e *Done* do contexto ou requisitar o término da execução através do método *Close* de um contexto, provendo uma informação de erro opcional. Caso nenhum erro seja fornecido, a execução do *pipeline* é finalizada com sucesso mesmo que ainda haja elementos a serem produzidos. Este cenário é usual em operações de *short-circuit* como por exemplo *Find* que sinaliza o término do *pipeline* assim que o primeiro elemento satisfazendo a condição de busca passe pelo estágio.

Em casos em que o canal *Done* do contexto é fechado qualquer *Producer* ou *Transformer* conectados ao *pipeline* de acordo com seus contratos devem encerrar seus processamentos liberando qualquer recurso alocado durante sua execução. Este mecanismo aonde cada componente é responsável por verificar o estado atual do contexto antes de realizar qualquer processamento

e suspender a execução em caso de término devido a erros (canal *Failure*) ou devido ao término prematuro (canal *Done*) permite com que centenas de milhares de estágios concorrentes sejam conectados ao *pipeline* sem impactar na complexidade de gerência e coordenamento do *pipeline*, uma vez que sinalizar o término da execução de cada um destes estágios se resume no fechamento de um dos canais mencionados. Isso é possível devido ao excelente modelo de concorrência da linguagem Go que baseia-se na filosofia de compartilhamento de memória através da comunicação em vez de comunicar através do compartilhamento de memória.

4.3 Suporte a Paralelização

Concorrência e paralelismo são conceitos similares porém diferentes, assim como abordado de maneira brilhante por Rob Pike em sua talk (PIKE,) *Concurrency is not Parallelism*. Estágios de um *pipelines* em Rivers são processados concorrentemente e cada estágio pode ainda ser paralelizado utilizando os recursos de *hardware* como *multi-cores* para atingir níveis extremos de paralelismo aumentado a eficiência do *pipeline*.

Rivers permite que certos tipos de *Transformers* possam ser paralelizados, como por exemplo um *Map* ou *Each transformers*. Para atingir níveis aceitáveis de paralelismo Rivers replica o *Transformer* em questão em vários outros, tantos quanto for a capacidade do *stream* sendo consumido, por exemplo se um *producer* tem a capacidade de produção de 3 elementos o *Transdormer* seria replicado obtendo 3 instâncias paralelas. Cada novo *Trnasformer* consome dados do estágio anterior concorrentemente, produzindo seus resultados em um mesmo *stream* resultante que é consumido pelo estágio senguinte do *pipeline*. A figura 4.24 mostra como seria o fluxo de um *pipeline* com paralelismo ativado.

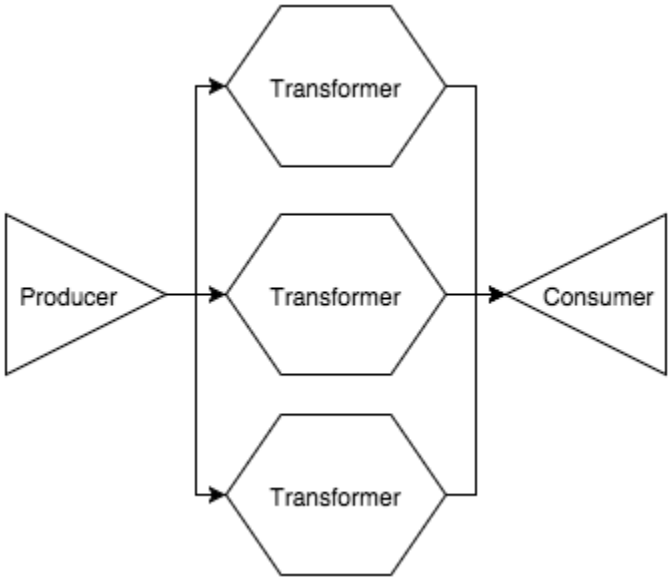


Figura 4.24 – Fluxo de um Pipeline com Paralelismo ativado.

Essa solução permite aliviar *bottlenecks* relacionados a estes tipos de *transformers*. O

código 4.25 mostra um exemplo de *pipeline* sendo executado sequencialmente e outro com paralelismo ativado:

```
1  // ./employees.csv contains 10000 employee entries
2  employees1 := rivers.FromFile("./employees.csv")
3  employees2 := rivers.FromFile("./employees.csv").Parallel()
4
5  // UpdateOrSave takes about 500 ms
6  err := employees1.Each(UpdateOrSave).Drain()
7  err := employees2.Each(UpdateOrSave).Drain()
```

Figura 4.25 – Exemplo de Paralelismo em Rivers.

No exemplo acima, dois *pipelines* são criados para processar um arquivo CSV contendo 10000 entradas com informações de empregados de uma determinada empresa. Para cada empregado deve-se verificar sua existência na base de dados atualizar determinada informação caso exista, ou criar uma nova entrada para o empregado correspondente. Esse procedimento leva em torno de 500 ms por empregado. O primeiro *pipeline* por não utilizar paralelismo levaria um tempo total de aproximadamente 1.38 horas para finalizar o processamento. Já o segundo *pipeline*, por ativar o uso de paralelismo, Rivers replica o estágio *Each* para cada entrada de *Employee* extraída do arquivo CSV executando cada estágio concorrentemente distribuindo a carga de trabalho em diferentes *threads* fazendo o uso de todos os *cores* disponíveis na máquina. O tempo total de execução do segundo *pipeline* medido em uma máguina com 8 *cores* foi de aproximadamente 2.58 segundos.

5 PROCESSO DE DESENVOLVIMENTO ÁGIL DE RIVERS

Rivers surgiu da necessidade de se implementar soluções simples porém eficientes para o processamento de grandes volumes de dados. Na empresa Bearch Inc esse processo é recorrente e pela falta de uma solução nativa no contexto da linguagem de programação Go utilizada amplamente nos projetos internos da empresa, desenvolvedores acabavam por implementar lógicas de processamento redundantes que se repetiam ao longo do desenvolvimento de vários projetos. Devido aos custos deste retrabalho e aos padrões encontrados no processo de desenvolvimento de vários projetos, foi proposto uma solução para ajudar a reduzir a quantidade de código necessário para se implementar essas rotinas de processamento assim como possibilitar o reuso de lógicas existentes de soluções anteriores.

Este capítulo discute brevemente o processo de desenvolvimento empregado assim como as práticas utilizadas para guiar a evolução da solução minimizando a quantidade de retrabalho necessário ao longo das iterações.

5.1 Coleta de Requisitos e Roadmap

Requisitos foram levantados a fim de se ter um conjunto mínimo de funcionalidades que pudesse formar o MVP inicial para que se iniciasse o desenvolvimento. A tabela 5.1 lista as features consideranas no Roadmap de Rivers, algumas delas selecionadas para compor o MVP e outras implementadas somente na versão seguinte. O MVP é composto pelo menor conjunto de features necessário para se ter o mínimo de funcionalidades disponíveis para se implementar um pipeline de processamento de streams que possa ser utilizado em diferentes contextos já identificados em projetos anteriores da empresa, como por exemplo no processamento de entidades da base de dados assim como no processamento de resultados de chamadas de APIs de outros sistemas.

A fim de atender as necessidades dos projetos utilizados como casos de uso, foi decidido então que uma solução útil e viável teria que prover pelo menos uma implementação genérica de cada um dos estágios que compõem um pipeline. Um Producer deveria permitir com que diferentes fontes de dados sejam utilizadas no pipeline como geradores de dados como por exemplo base de dados, APIs Restful e arquivos. Uma implementação de Transformer deve permitir que uma lógica específica de processamento possa ser aplicada aos dados sendo transmitidos pelo stream e seu resultado passado ao próximo estágio. Um Consumer por sua vez deve permitir que dados possam ser coletados de maneira síncrona ao final do pipeline assim como possíveis erros de execução. Por fim, a implementação inicial deveria prover um mecanismo simples que pudesse detectar falhas em tempo de execução notificando-as aos estágios do pipeline para que os mesmos possam suspender sua execução. Implementações mais especializadas de cada um

Backlog	MVP	V2.0
Contexto Global	X	
Producer Interface	X	
Producers Especializados (List, Socket, File...)		X
Transformer Interface	X	
Transformers Especializados (Map, Each, Filter...)		X
Consumer Interface	X	
Consumers Especializados (Count, Reduce...)		X
Dispatchers		X
Combiners		X
Panic Recovering	X	
Failure Retries		X
Suporte a Paralelização		X
Pipelines Distribuídos		X

Tabela 5.1 – Rivers Roadmap: MVP vs. V2.0

destes estágios seriam o foco em versões futuras da API como por exemplo producers especializados em leitura de arquivos, drivers de base de dados específicas, transformers especializados em operações como Map, Reduce, Filter, etc, mecanismos para se executar o pipeline de maneira distribuída em um cluster de máquinas.

O plano de desenvolvimento foi mantido e disponibilizado como GitHub (GITHUB, 2015) issues apresentado através de um dashboard Kanban (BLOG, 2015) disponível em Waffle.io (WAFFLE.IO, 2015) para facilitar a visualização do progresso mantendo a informação e comunicação centralizada, como mostrado na figura 5.1. Ao longo do desenvolvimento, cada feature implementada era prontamente testada em casos de uso reais por outros desenvolvedores e feedbacks coletados a fim de aprimorar a solução de acordo com as necessidades reais do time.

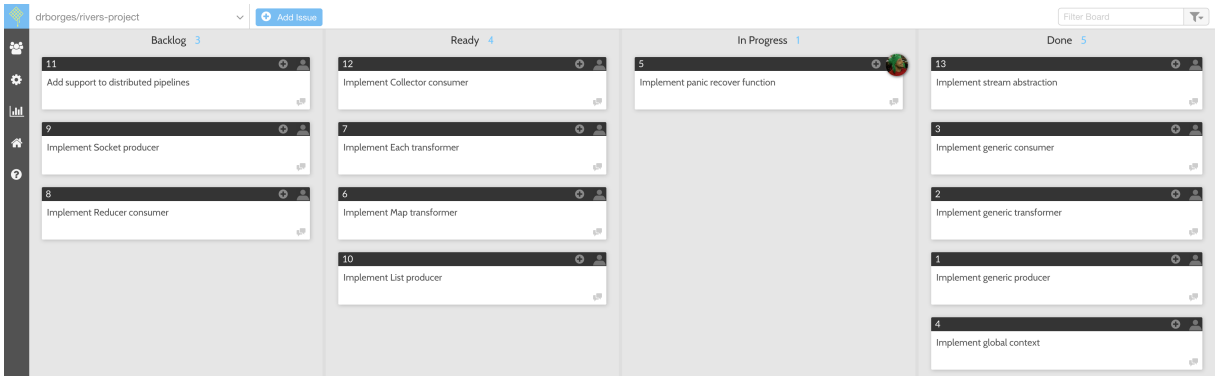


Figura 5.1 – Roadmap visualizado em um dashboard Waffle.

5.2 Tests & Benchmarks

Foi utilizada a técnica Test Driven Development (BECK, 2001) para toda nova funcionalidade implementada. Escrevendo-se os casos de teste antes mesmo de se ter a funcionalidade ajudou a guiar o design da API gradativamente, uma vez que toda complexidade envolvida na implementação da nova funcionalidade era colocado inicialmente de lado, permitindo o desenvolvedor focar no design da API se colocando como usuário da mesma em um primeiro momento. Através do uso desta técnica, foi possível alcançar uma cobertura de testes razoável criando um conjunto de testes de regressão muito útil na detecção de quebra de funcionalidades já existentes devido a alterações de certas áreas do codebase. Visando tirar o máximo de proveito da técnica, foi utilizado a ferramenta fsnotify (YOUNGMAN, 2015) para execução dos testes de regressão sempre que um arquivo no codebase é alterado, disponibilizando um relatório dos resultados de cada teste executado com isso um feedback instantâneo com relação as últimas alterações no codebase. A figura 5.2 representa o ciclo de desenvolvimento seguido aplicando a técnica TDD.

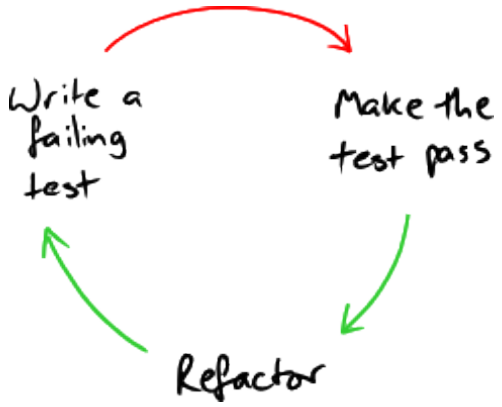


Figura 5.2 – Ciclo de desenvolvimento aplicando a técnica TDD.

Alguns benchmarks foram implementados para medir a eficiência de um pipeline segundo a solução proposta em comparação com soluções utilizada previamente em outros projetos. A tabela a seguir mostra os resultados medidos entre duas versões de um programa que processa 1000 entidades *datastore* aonde o processamento leva em média 1 seg por entidade. Na primeira versão (ver figura 6.2) foi implementado um pipeline Rivers com paralelização ativada e na segunda versão (ver figura 6.1) não foi utilizado Rivers ou qualquer tipo de concorrência e as entidades *datastore* são processadas sequencialmente.

Jobs	Número de Goroutines	Tempo Médio de Execução (seg)
Pipeline Rivers	1000	1.16
Solução Sequencial	1	>1000

Tabela 5.2 – Comparação entre duas versões de um programa: Sequencial sem o uso de Rivers e Paralelo com o uso de Rivers

6 RIVERS: APPLICAÇÕES REAIS

Neste capítulo serão descritas algumas aplicações reais de Rivers utilizadas ao longo do desenvolvimento para validar os conceitos e soluções propostas por este trabalho.

6.1 Appx - Appengine eXtensions

Appx (BORGES, 2015) é um conjunto de extensões da plataforma *Appengine* da Google (GOOGLE, 2015a) para o runtime da linguagem Go que oferece uma solução similar à Object Relational Mapping (AMBLER, 2015) para modelagem da camada de domínio de aplicações que utilizam a solução de *NoSQL* (FOWLER, 2015c) Datastore (GOOGLE, 2015b) da plataforma. Rivers foi utilizado para dar suporte a *Continuous Querying Over Data Stream* (BABU, 2001), permitindo que grandes conjuntos de dados possam ser continuamente buscados da base de dados em batches via paginação e processados assincronamente como um stream de entidades Datastore em poucas linhas de código.

Processar um grande conjunto de dados aplicando filtros, mapeamentos e realizando a paginação dos resultados é relativamente complexo quando se utilizando apenas as APIs nativas do Datastore. Desenvolvedores precisam explicitamente implementar a paginação contínua dos dados através da manipulação de cursores de busca, realizar tratamento de possíveis erros de execução além de implementar a lógica de processamento dos dados, sendo necessário várias linhas de código para se obter o resultado final levando a uma solução difícil de manter e muita duplicação de código quando esse processamento deve ser aplicado a diferentes entidades da base de dados. A figura 6.1 mostra a implementação de um agendador de reuniões que envia email, sms e atualiza o calendário de cada empregado de uma empresa cujo título seja *Manager* ou *Director* com as informações da reunião. Devido a uma limitação do Datastore, é possível processar no máximo 1000 entidades por vez, sendo necessário realizar a paginação contínua dos dados até que todos eles sejam processados. Grande parte desta implementação não possui relação direta com a lógica necessária para o agendamento da reunião mas sim com a paginação dos dados.

```go
type Employee struct {
        Name string
        Area string
        Role string
}

func ScheduleManagersOrDirectorsUSMeeting(context appengine.Context, pageCursor string) {
        DatastoreMaxPageSize := 1000
        query := datastore.NewQuery("Employee").Filter("Area =", "US").Limit(DatastoreMaxPageSize)

        if cursor, err := datastore.DecodeCursor(pageCursor); err == nil {
                query = query.Start(cursor)
        }

        done := false
        iterator := query.Run(context)

        for !done {
                employee := new(Employee)
                if employeeKey, state := iterator.Next(employee); state != datastore.Done {
                        if isManagerOrDirector(employee) {
                                if err := sendEmailInvite(employee); err != nil {
                                        handleSendEmailInvideError(err)
                                }

                                if err := sendSMSNote(employee); err != nil {
                                        handleSendSMSNoteError(err)
                                }

                                if err := updateCalendarEvents(employee); err != nil {
                                        handleUpdateCalendarEventsError(err)
                                }
                        }
                } else {
                        done = true
                }
        }

        if nextCursor, err := iterator.Cursor(); err == nil {
                ScheduleManagersOrDirectorsUSMeeting(context, nextCursor.String())
        }
}
```

Figura 6.1 – Exemplo de uso da API nativa do Datastore.

Appx utiliza Rivers para reduzir grande parte desta complexidade sendo necessárias poucas linhas de código para se obter o mesmo resultado do exemplo anterior. A figura 6.2 mostra como podemos reescrever este mesmo caso de uso utilizando a API de streamming de Appx. Devido a arquitetura extensível de Rivers sua integração com Appx é relativamente simples, sendo necessário que Appx apenas implemente a interface *stream.Producer* de Rivers que encapsula a execução da query, tratamento de erros e a paginação contínua dos resultados emitindo cada entidade datastore que satisfaz a query em um *stream.Readable* ao qual operações como *Filter*, *Each*, *Map* e *Reduce* podem ser aplicadas formando um pipeline de processamento de entidades *Datastore*. Desta maneira o resultado final não é somente mais legível mas como também fácil de manter e alterações na lógica de processamento é tão simples como adicionar ou remover filters, mappers, etc.

```
type  Employee struct {
        appx.Model
        Name string
        Area string
        Role string
}

func ScheduleManagersOrDirectorsUSMeeting(context appengine.Context) {
        employeesByArea := datastore.NewQuery("Employee").Filter("Area =", "US")
        err :=appx.NewDatastore(context).Query(employeesByArea).
                StreamOf(Employee{}).Parallel().
                Filter(isManagerOrDirector).
                Each(sendEmailInvite).
                Each(sendSMSNote).
                Each(updateCalendarEvents).
                Drain()

        if err != nil {
                handlePipelineError(err)
        }
}
```

Figura 6.2 – Exemplo de uso da API de streamming do framework Appx utilizando Rivers com paralelismo ativado.

6.2 Web Scrapping

Web Scraping (WEBHARVY, 2015) é uma solução simples e eficaz para coleta e extração de dados na web e são frequentemente utilizados em conjunto com *Web Crawlers* (WORKS, 2015), sistemas responsáveis por automatizar a busca e descoberta da informação, basicamente acessando documentos web através de *URLs*, extraindo e visitando cada URL no documento retornado aplicando este processo recursivamente até que uma determinada condição de parada seja satisfeita, a fim de extrair e agregar determinados tipos de informação presentes em cada documento.

Este processo de *crawling* por envolver altos níveis de tráfego de rede pode vir a ser muito custoso e a possibilidade de paralelizar partes do processo pode ajudar a reduzir este custo. Rivers pode ser utilizado na implementação de pipelines com foco em *Web Crawling* e *Scrapping*, aonde cada *crawler* implementa a interface *stream.Producer* com uma lógica específica responsável por "caminhar"a web seguindo determinadas regras específicas do crawler passando cada documento encontrado ao próximo estágio do pipeline que implementa a interface *stream.Transformer* responsável por extrair as informações necessárias do documento implementando a função de scrapping. A imagen 6.3 mostra um exemplo de *Crawler* utilizado como *stream.Producer* em um pipeline Rivers responsável por extrair URLs de documentos HTML com a listagem de gastos públicos das cidades do Rio Grande do Sul disponíveis no

Portal da Transparência (TRANSPARêNCIA, 2015), passando cada URL ao estágio seguinte do pipeline que implementa um scrapper como mostrado na figura 6.4 que dado uma URL, o documento correspondente é recuperado via HTTP e as informações de gastos extraídas e coletadas em um dicionário chave-valor aonde a chave representa a área correspondente ao gasto como por exemplo educação ou saúde e seu valor representando o total investido naquela área. A figura 6.5 mostra o uso do *Crawler* e *Scrapper* em um pipeline Rivers com paralelização ativa, podendo processar até 510 URLs em paralelo que é o número total de cidades a serem processadas e representa a capacidade máxima de produção do pipeline especificado pelo *Crawler* na linha 25.

```
23  func CityExpensesCrawler(year int) stream.Producer {
24          return &producers.Observable{
25                  Capacity: 510,
26                  Emit: func(emitter stream.Emitter) {
27                          lastPage := 34
28                          for i := 0; i < lastPage; i++ {
29                                  resp, _ := http.Get(PaginatedCitiesList(year, i))
30
31                                  doc, _ := goquery.NewDocumentFromReader(resp.Body)
32                                  links := doc.Find("#listagem tbody tr td a")
33                                  links.Each(func(_ int, sel *goquery.Selection) {
34                                          path, _ := sel.Attr("href")
35                                          emitter.Emit(CityExpensesURL(path))
36                                  })
37                          }
38                  },
39          }
40  }
```

Figura 6.3 – Exemplo de um Crawler que implementa a interface stream.Producer.

```
42 func CityExpensesPerAreaScrapper(cityExpensesURL stream.T) stream.T {
43         resp, _ := http.Get(cityExpensesURL.(string))
44         doc, _ := goquery.NewDocumentFromReader(resp.Body)
45 
46         expenses := doc.Find("#listagem tr")
47         areas := expenses.Find("td.firstChild").
48                 Map(func(_ int, sel *goquery.Selection) string {
49                         return strings.TrimSpace(sel.Text())
50                 })
51 
52         amounts := expenses.Find("td.colunaValor").
53                 Map(func(_ int, sel *goquery.Selection) string {
54                         return strings.TrimSpace(sel.Text())
55                 })
56 
57         expensesPerArea := make(map[string]string)
58         for i, area := range areas {
59                 expensesPerArea[area] = amounts[i]
60         }
61 
62         return expensesPerArea
63 }
```

Figura 6.4 – Exemplo de um Scrapper que implementa a interface stream.Transformer.

```
72 func main() {
73         targetYear := 2015
74         rivers.From(CityExpensesCrawler(targetYear)).Parallel().
75                 Map(CityExpensesPerAreaScrapper).
76                 Each(printExpenses)
77 }
```

Figura 6.5 – Exemplo de uso do Crawler e Scrapper em um pipeline Rivers.

7 CONCLUSÃO

Rivers propões uma API simples, extensível e intuitiva para processamento de streams de dados para a linguagem Go, utilizando de construções e conceitos de programação funcional familiares a maioria dos desenvolvedores abstraindo as complexidades e detalhes de implementação relacionados ao processamento concorrente e paralelo dos dados, de maneira que o desenvolvedor passa se concentrar apenas na lógica de negócio intrínseca ao pipeline.

O modelo de concorrência da linguagem Go provou-se não somente muito conveniente mas também extremamente eficiente e foi crucial para o sucesso da implementação da solução uma vez que muito da complexidade dos mecanismos de detecção de falhas, de comunicação entre os componentes do sistema via canais de comunicação e gerência de recursos alocados durante a execução de grandes quantidades de fluxos de processamento concorrentes não afetaram de maneira considerável a complexidade final da solução.

Por se tratar de um experimento e guiado inicialmente pelas necessidades e casos de uso da empresa Bearch Inc (INC, 2015) a API sofreu várias alterações ao longo do desenvolvimento a medida em que novos casos de uso foram descobertos assim como devido a refatorações afim de remover complexidades visando a simplicidade da API e ao mesmo tempo sem prejudicar a flexibilidade e extensibilidade da solução.

7.1 Análise dos Resultados

Ao longo do desenvolvimento muitas informações foram coletadas desde benchmarks até feedbacks de desenvolvedores utilizando o framework em outros projetos da empresa afim de avaliar a solução em termos de performance, simplicidade e flexibilidade. Os resultados dos benchmarks foram muito satisfatórios e mostraram um ganho considerável em performance ao se paralelizar estágios do pipeline além da execução concorrente de cada estágio.

O design funcional da API agradou desenvolvedores pela simplicidade da solução e o fato de se poder estender a API com novos componentes implementando as interfaces definidas pelo framework possibilitou o uso de Rivers em contextos bem variados. Algumas das decisões técnicas quanto ao design da API foram guiados por feedbacks de usuários que alongo de várias conversas e experimentos ajudaram a moldar a API final desde a nomenclatura das operações até mesmo com relação a melhor solução para se paralelizar pipelines sem expor qualquer tipo de complexidade ao usuário do framework. Um ponto negativo levantado por desenvolvedores é que devido ao fato da sintaxe da linguagem Go em alguns aspectos ser um pouco extensa comparado com outras linguagens como scala e python, é necessário escrever um pouco mais de código, porém isso pode ser contornado fornecendo implementações específicas de operações recorrentes que possam ser reusadas evitando a necessidade de se implementar a mesma funcionalidade em diferentes contextos.

7.2 Trabalhos Futuros

Ao longo do desenvolvimento da solução, alguns casos de uso interessantes foram detectados, propostos e discutidos como por exemplo a possibilidade de se implementar pipelines distribuídos, permitindo a execução de diferentes estágios do pipeline em um cluster de máquinas seguindo um modelo similar ao modelo MapReduce proposto por Google (ZHAO; PJESIVAC-GRBOVIC, 2009). Apesar de ser um caso de uso muito interessante, a complexidade de implementação não justificou sua necessidade momentânea e portanto não foi considerado prioridade no desenvolvimento. Porém essa possibilidade não foi completamente descartada e fica proposta como trabalho futuro uma vez que existem necessidades reais que justificam o investimento em tal solução, especialmente nos domínios de Big Data aonde o volume de dados a serem processados é incrivelmente grande e a possibilidade de se distribuir e processar conjuntos menores destes dados em diferentes máquinas agregando seus resultados ao final é muito atraente.

REFERÊNCIAS

AMBLER, S. W. **ORM**. 2015. Disponível em: <http://www.agiledata.org/essays/mappingObjects.html>.

APACHE. **Spark Framework**. 2015. Disponível em <http://spark.apache.org>. Acesso em Outubro 2015.

APPLE. Push notification service. 2015. Disponível em: <http://apple.co/1iP08v6>.

BABU, J. W. S. **Continuous Queries over Data Streams**. 2001. Disponível em: <http://ilpubs.stanford.edu:8090/527/1/2001-9.pdf>.

BECK, K. **Test Driven Development: By Example**. [S.l.]: Addison-Wesley Professional, 2001.

BERKELEY, c. C. **Chapter 5: Sequences and Coroutines**. 2015. Disponível em: <http://wla.berkeley.edu/~cs61a/fa11/lectures/streams.html#coroutines>.

BLOG, K. **Kanban**. 2015. Disponível em: <http://kanbanblog.com/explained>.

BOOTCAMP, C. **Memory: Stack vs Heap**. 2012. Disponível em: <http://gribblelab.org/CBootcamp/7_Memory_Stack_vs_Heap.html>.

BORGES, D. **Appx**. 2015. Disponível em: <https://github.com/drborges/appx>.

BUYTAERT, K. **Clustering**. 2004. Disponível em: <http://tldp.org/HOWTO/openMosix-HOWTO/x135.html>.

CASWELL, T. **Daddy What's a Stream?** 2013. Disponível em: <http://howtonode.org/streams-explained>.

COUNCIL, I. **Internet of Things**. 2015. Disponível em: <http://www.theinternetofthings.eu>.

CS-132-WATERLOO. **Short-Circuit Boolean Expressions**. 2005. Disponível em: <https://www.student.cs.uwaterloo.ca/~cs132/Weekly/W02/SCBooleans.html>.

FOWLER, M. **Fluent Interfaces**. 2005. Disponível em <http://bit.ly/1PwJAtd>. Acesso em Outubro 2015.

FOWLER, M. **Collection Pipeline**. 2015. Disponível em <http://bit.ly/UKJipi>. Acesso em Outubro 2015.

FOWLER, M. **Method Chaining**. 2015. Disponível em: <http://www.sitepoint.com/a-guide-to-method-chaining>.

FOWLER, M. **NoSQL**. 2015. Disponível em: <http://martinfowler.com/nosql.html>.

GITHUB. **Github**. 2015. Disponível em: <https://github.com>.

GOLANG. **Goroutines**. 2015. Disponível em: <https://golang.org/doc/effective_go.html/goroutines>.

GOLANG. **Pipeline Pattern**. 2015. Disponível em: <https://blog.golang.org/pipelines>.

GOOGLE. **Appengine**. 2015. Disponível em: <https://cloud.google.com/appengine/docs>.

GOOGLE. **Datastore**. 2015. Disponível em: <https://cloud.google.com/datastore>.

HASNKELL. **Streams**. 2015. Disponível em: <https://hackage.haskell.org/package/io-streams>.

INC, B. **Bearch Web Site**. 2015. Disponível em: <https://getbearch.com>.

INGALLS, R. P. **CSCI.4210: Operating Systems Deadlock**. 2004. Disponível em: <http://www.cs.rpi.edu/academics/courses/fall04/os/c10/>.

JAVA8. **Streams**. 2015. Disponível em: <http://bit.ly/1ustKrM>.

JAVARX. **JavaRX**. 2015. Disponível em: <https://github.com/ReactiveX/RxJava>.

JOSEPH, A. **Operating Systems and Systems Programming**. 2003. Disponível em: <https://inst.eecs.berkeley.edu/~cs162/sp03/Lectures/L03.pdf>.

KOCHER, D. **Concurrency Patterns**. 2005. Disponível em <http://bit.ly/1WbldXl>.

KRISHNA. **How Goroutines Work**. 2014. Disponível em: <http://blog.nindalf.com/how-goroutines-work>.

MEIJER, P. D. E. **Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages**. 2015. Disponível em: <https://www.ics.uci.edu/~lopes/teaching/inf212W12/readings/rdl04meijer.pdf>.

MICROSOFT. **Microsoft**. 2015. Disponível em: <http://http://www.microsoft.com>.

MICROSYSTEMS, S. **Memory Management in the Java HotSpot™ Virtual Machine**. [S.l.], 2006. Disponível em: <http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf>.

NETFLIX. **Netflix**. 2015. Disponível em: <http://www.netflix.com>.

NG, A. **Machine Learning**. 2015. Disponível em: <https://www.coursera.org/learn/machine-learning>.

NODEJS. **Streams**. 2015. Disponível em: <https://nodejs.org/api/stream.html>.

ODERSKY, M. **Principles of Reactive Programming**. 2013. Disponível em: <https://www.coursera.org/course/reactive>.

ORACLE. **Java Interfaces**. 2015. Disponível em: <https://docs.oracle.com/javase/tutorial/java/concepts/interface.html>.

PECINOVSKý, R. Learn object oriented thinking and programming. In: ____. [s.n.], 2015. cap. 13 The Inheritance of Interface Types. Disponível em: <http://www.cs.unc.edu/~dewan/242/s04/notes/ipc.PDF>.

PIKE, R. **Concurrency is not Parallelism**. Disponível em <http://bit.ly/1DcWhAW>. Acesso em: Outubro 2015.

REACTIVEX. **Observable Design Pattern**. 2015. Disponível em: <http://reactivex.io/documentation/observable.html>.

REACTIVEX. **Reactive Streams**. 2015. Disponível em: <http://reactivex.io>.

ROUSE, M. **Analytics**. 2008. Disponível em: <http://searchdatamanagement.techtarget.com/definition/data-analytics>.

RXJS. **RxJs**. 2015. Disponível em: <https://github.com/Reactive-Extensions/RxJS>.

SCHMIDT, D. C. **An OO Encapsulation of Lightweight OS Concurrency Mechanisms in the ACE Toolkit**. 2015. Disponível em: <https://www.dre.vanderbilt.edu/~schmidt/PDF/ACE-concurrency.pdf>.

SEGUIN, B. K. **Introduction to Golang: Buffered Channels**. 2013. Disponível em: <http://openmymind.net/Introduction-To-Go-Buffered-Channels>.

STREAMING, S. **Apache Spark Streaming**. 2015. Disponível em: <http://spark.apache.org/streaming>.

TANENBAUM, A. S. **INTERPROCESS COMMUNICATION**. [S.l.]: Pearson; 4 edition, 2014.

TRANSPARêNCIA, P. da. **TRANSFERÊNCIA DE RECURSOS POR ESTADO/MUNICÍPIO UF: RIO GRANDE DO SUL, EXERCÍCIO: 2015**. 2015. Disponível em: <http://www.portaldatransparencia.gov.br/>.

WAFFLE.IO. **Waffle.io**. 2015. Disponível em: <http://waffle.io>.

WEBHARVY. **What Is Web Scrapping?** 2015. Disponível em: <https://www.webharvy.com/articles/what-is-web-scraping.html>.

WORKS, H. S. **How Internet Search Engines Work**. 2015. Disponível em: <http://computer.howstuffworks.com/internet/basics/search-engine1.htm>.

YAVATKAR, R. Interprocess communication. In: ____. [s.n.], 2015. cap. 1.3 Message Passing. Disponível em: <http://www.cs.unc.edu/~dewan/242/s04/notes/ipc.PDF>.

YOUNGMAN, N. **FSNotify**. 2015. Disponível em <https://github.com/go-fsnotify/fsnotify>. Acesso em Outubro 2015.

YU, K. **Unix Bootcamp**. 2015. Disponível em: <http://bit.ly/1VYYg9I>.

ZHAO, J.; PJESIVAC-GRBOVIC, J. **MapReduce: The programming model and practice**. 2009. Disponível em <http://bit.ly/1LRy3id>.