MOMP

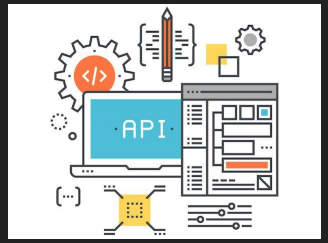
Message Oriented Middleware based on Priority

Douglas Soares, Jônatas de Oliveira, Ullayne Fernandes, Valdemiro Vieira

Objetivo

- Implementar um middleware orientado a mensagens baseado em prioridades de mensagens, que suporta múltiplas conexões, baseado no modelo publish/subscribe. (Tipo 1).

Cenário de uso



Produtor

Eventos

MOMP

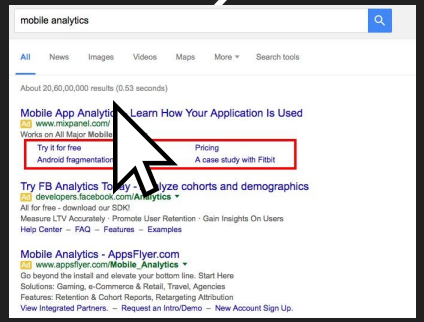
Eventos

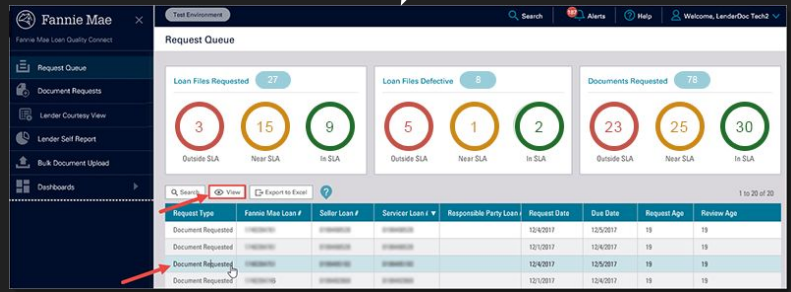
Consumidor

Eventos

Consumidor

Eventos





Requisitos

	Requisitos Funcionais
RF01	Tópicos (filas) podem ser criados e clientes podem produzir/consumir os dados dos tópicos
RF02	Mensagens dos tópicos possuem prioridade, tais que mensagens com maior prioridade são enviadas primeiro.
RF03	Serializar e deserializar mensagem
RF04	Gerenciamento das filas
RF05	Comunicação assíncrona
RF06	Consumidores irão receber dados de forma passiva
RF07	Modelo publish/subscribe
RF08	Controle de concorrência

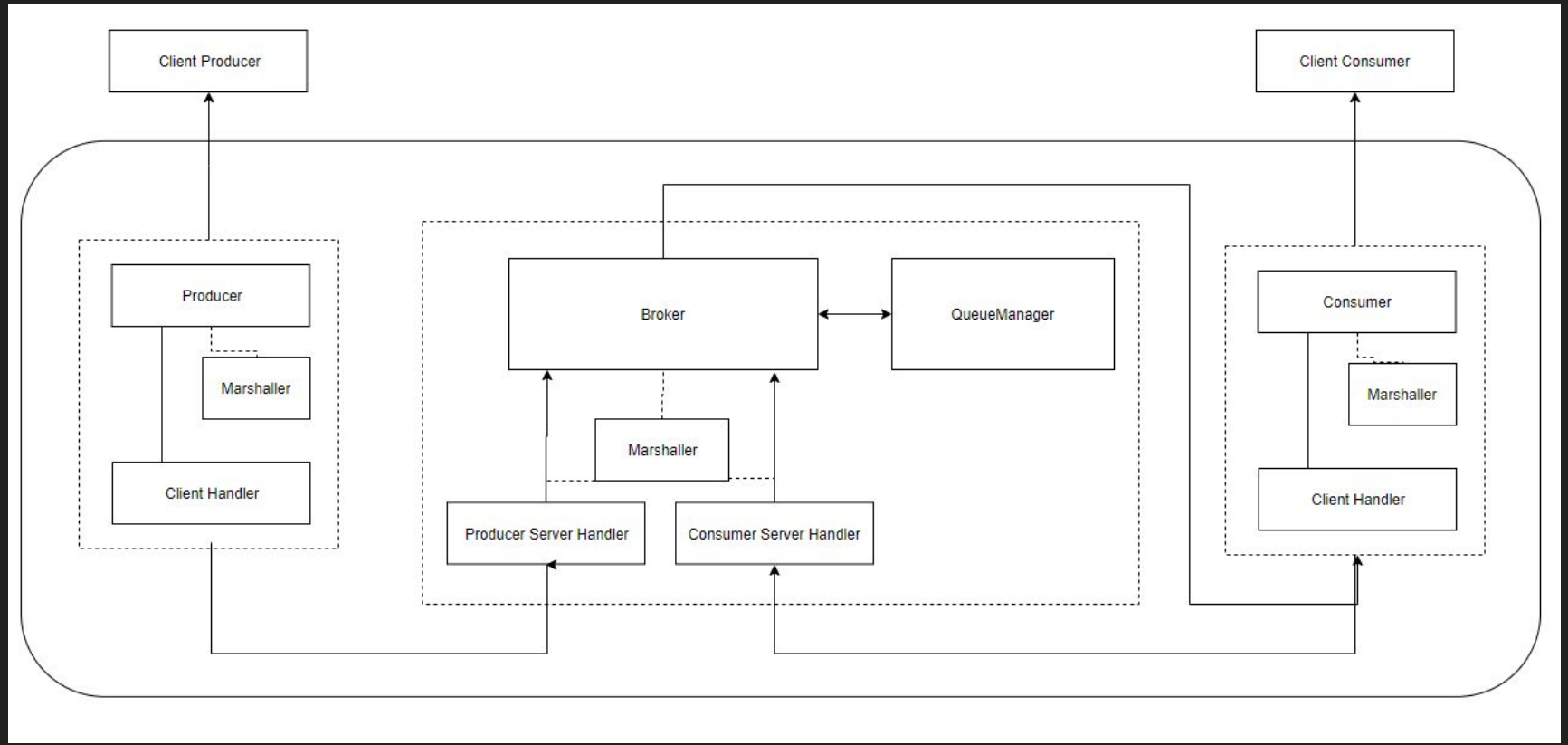
Requisitos

	Requisitos Não-Funcionais
RNF01	Consistência - consumidores não vão receber dados já consumidos (duplicados)

Transparência

- Transparência de concorrência - é possível suportar múltiplos consumidores e múltiplos produtores e a consistência será mantida.
- Transparência de falha - caso os subscribers (ou algum deles) que vão consumir os dados não estejam disponíveis, o servidor de mensageria vai continuar tentando enviar as mensagens.

Arquitetura



Arquitetura - Descrição

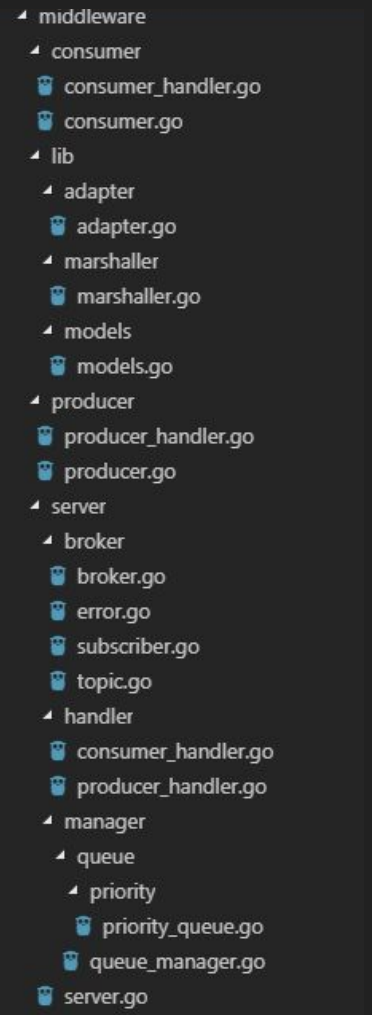
- Producer: possui funções para criar tópico e enviar mensagens para o mesmo;
- Consumer: possui funções para se inscrever em um tópico e receber as mensagens do mesmo;
- Client Handler: possui funções de enviar e receber requisições;
- Marshaller: possui funções de marshall e unmarshall mensagens;
- QueueManager: responsável por guardar as mensagens específicas de cada tópico com controle de concorrência;
- Broker: responsável por armazenar quem são os subscribers inscritos nos tópicos; responsável de pegar as mensagens e enviar para os consumidores; e responsável pelo controle de reenvio de mensagens, tudo com paralelismo e controle de concorrência.

Arquitetura - Interação

- Producer ou Consumer: se comunicam com o marshaller envia/recebe requisições do client handler;
- Client Handler: recebe requisições dos clientes e as enviam/recebe do server handler (producer ou consumer);
- Server Handler: recebe as requisições do client handler, decodificam e enviam para o broker;
- Broker: recebe as mensagens do server handler, manda para o QueueManager, pega a resposta e envia para o consumidor (client handler);
- QueueManager: interage com o broker enviando a próxima mensagem que deve ser enviada (com maior prioridade)

Projeto

- Tipo de Middleware:
  - Middleware Orientado à Mensagem (MOM)
  - Modelo Publisher/Subscriber
- Padrão de Projeto Utilizado
  - Broker



Implementação

- Implementação realizada em Golang
- Utilizou-se marshal/unmarshal de json para converter as mensagens para array de byte
- Utilizou-se a biblioteca 'net' para comunicação dos serviços
- Utilizou-se a biblioteca 'sync' para lidar com concorrência
- Utilizou-se a biblioteca 'container/heap' para implementar fila de prioridades

Implementação

```go
type Publisher interface {
    TopicDeclare(topicName string, maxPriority int)
    Publish(queueName string, content Publishing)
}
```

```go
type Subscriber interface {
    Subscribe(topicName string, identifier string)
    Receive(response interface{}) error
}
```

```go
type producerHanlder struct {
    conn        net.Conn
    jsonEncoder *json.Encoder
    jsonDecoder *json.Decoder
}
```

```go
type consumerHanlder struct {
    conn        net.Conn
    jsonEncoder *json.Encoder
    jsonDecoder *json.Decoder
}
```

Implementação

```
type Broker interface {
    Subscribe(subscriberConnection net.Conn, topicName string, identifier string) error
    Publish(topicName string, messagePrioriy int, data []byte) error
    BroadcastTopic(topic *Topic) error
    BroadcastConn()
    BroadcastNotConn()
    CreateTopic(topicName string, maxPriority int, retry bool) error
}
```

Implementação

```go
type safePriorityQueue struct {
    priorityQueue priority.PriorityQueue
    lock          *sync.RWMutex
}

type queueManager struct {
    queues map[string]*safePriorityQueue
    lock   *sync.RWMutex
}

type QueueManager interface {
    Pop(topic string) (interface{}, error)
    Push(topic string, item interface{}) error
    Len(topic string) (int, error)
    CreateQueue(topic string) error
}
```

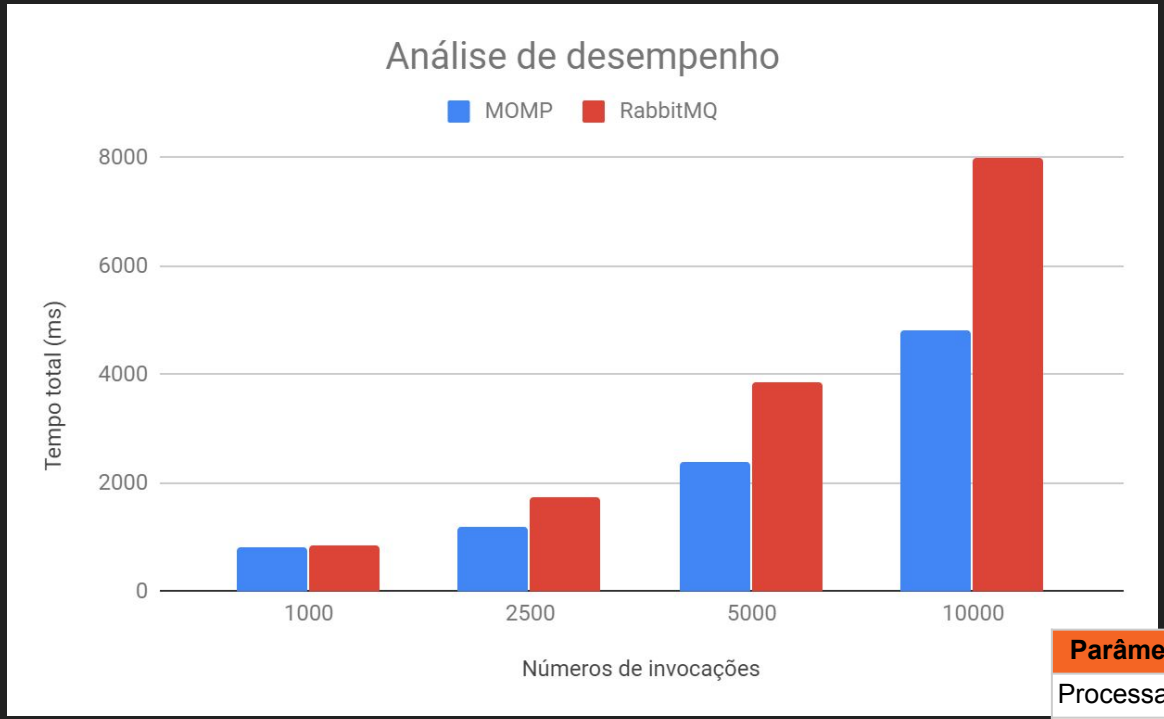
Ambiente de Teste

- Processador Intel Core i5-4590 3.3GHz
- Memória 8GB DDR4
- Sistema Operacional Windows 10
- Middleware utilizado para comparar a performance: rabbitMQ
- Aplicação:
  - Cria eventos como: ad-click, ad-view, misclick, link-click, prod-view
  - Cada evento possui uma prioridade associada

Desempenho

- Avaliação comparativa com o RabbitMQ
- Utilizando uma aplicação que produz eventos aleatórios em um tópico e um consumidor consome estes eventos (end-to-end)
- Avaliando se os eventos produzidos são iguais ao que chega no consumidor (serialização e desserialização), e se os eventos que chegam são primeiro os de maior prioridades para depois os de menores prioridades.
- Analisando o tempo total de execução para 1000, 2500, 5000 e 10000 invocações
- Utilizando o mesmo PC para a realização para manter consistência nas análises

Desempenho



Parâmetros do Sistema	Valor
Processador	Intel Core i5-4590 3.3GHz
Memória	8GB DDR4
Sistema Operacional	Windows 10

Conclusão

- Middleware de mensageria publisher/subscriber onde mensagens possuem prioridades e mesmo quando o consumidor não está ativo, o serviço continua tentando enviar de tempos e tempos
- Limitações
  - Não persiste em memória
  - Caso a conexão do servidor cair, as mensagens não serão enviadas
- Lições aprendidas
  - Importância das etapas de um middleware
  - Tentar sempre garantir a entrega talvez não seja a melhor estratégia