# Sistema de mensagens orientado a eventos

João Francisco B. S. Martins, Victor B. R. Jorge, Alexandre A. R. Pereira

19 de Junho de 2017

## 1 Introdução

Aplicativos de trocas de mensagens se tornaram item essencial na vida moderna. A importância dessas aplicações se tornou tal que o simples fato de algum desses aplicativos ficar indisponível por algum tempo causa impactos econômicos e políticos em larga escala. Apesar de serem utilizados pela maioria das pessoas, nem sempre os usuários dessas plataformas conhecem o aparato tecnológico necessário para que o serviço funcione através da estrutura da internet.

Sendo assim, este trabalho tem por objetivo implementar um protocolo de troca de mensagens. Esse tipo de protocolo pode ser considerado a base tecnológica que viabiliza os aplicativos tão utilizados atualmente. O protocolo será implementado na camada de aplicação utilizando o modelo cliente-servidor combinado com a arquitetura de orientação a eventos. Portanto, teremos um servidor central que lida com as conexões de maneira orientada a eventos e os clientes que consistem de um emissor e um exibidor. Mais detalhes do protocolo serão dados nas sessões a seguir.

## 2 Arquitetura

O modelo cliente-servidor é uma estrutura de aplicação distribuída que negocia as tarefas e cargas de trabalho entre os fornecedores de um recurso ou serviço, identificados como servidores e os que requisitam os serviços, identificados como clientes. A arquitetura orientada a eventos é um padrão de arquitetura de software que promovove a produção, detecção, consumação e reação a eventos e que permite que as aplicações e sistemas sejam contruídas de maneira a facilitar uma maior responsividade. Isto ocorre pois estes sistemas são por desenho normalizadores de ambientes assíncronos ou imprevisíveis.

Um evento pode ser definido como "uma mudança significativa de estado". Um sistema orientado a eventos tipicamente é formado por três componentes principais: Emissores, consumidores e canais.

Os emissores não conhecem os consumidores, não sabem se os consumidores existem, e caso esses existam não sabem como os eventos são usados. Estes detectam, juntam e transferem novos eventos para o destino indicado.

Os consumidores devem tomar uma ação frente a um evento apresentado. A ação pode ser completa ou não. Ele pode conter toda a ação a ser realizada ou pode filtrar, processar e encaminhar o evento para que um outro componente execute a ação adequada.

Os canais são os meios pelos quais emissores e consumidores se comunicam.

Três programas foram desenvolvidos: Um servidor, um exibidor e um emissor. O exibidor e o emissor compõem o cliente. No contexto da arquitetura supracitada o cliente seria o emissor, o servidor seria o consumidor e o canal seria o TCP na internet.

# 3 Emissor

Na arquitetura utilizada neste trabalho, o Emissor é responsável por uma tarefa simples, porém essencial: o envio de mensagens. O emissor pode ser executado com o seguinte comando:

```
$ python emmiter.py server_addr:port <exhibitor>
```

Ao ser executado, ele se conecta ao servidor pelo porto indicado nos parâmetros. Ao conectar-se, o emissor envia a mensagem OI ao servidor. Essa mensagem configura uma espécie de *handshake* entre servidor e cliente. Caso um exibidor seja também passado via linha de comando, ele é informado ao servidor através da mensagem OI. O servidor então retorna uma mensagem dizendo qual é o identificador dado àquele emissor.

O emissor, basicamente, é um programa que executa um loop infinito sempre esperando o usuário escrever uma mensagem. São suportados três formatos de mensagem: (i) (id) MSG, (ii) (CREQ) id e (iii) (FLW). A primeiro formato envia uma mensagem para o exibidor com o identificador informado. A segunda mensagem faz uma requisição ao servidor para que ele envie a lista de pares conectados ao exibidor com o identificador informado. A última desconecta o cliente do servidor e o encerra. Para executar um broadcast, basta enviar uma mensagem com id = 0.

Assim que o usuário escreve uma mensagem, o Emissor verifica qual o tipo da mensagem a ser enviada. Definido o tipo, os parâmetros são validados e o quadro do protocolo é montado de acordo com as especificações e enviado logo em seguida. Importante notar que como o protocolo implementa número de sequência, é necessário atualizar esse valor a cada mensagem enviada. A resposta do servidor (OK, ERRO ou FLW) então é processada. Além disso, ao ser encerrado com uma interrupção de teclado (Ctrl-c), o emissor envia uma mensagem FLW ao servidor e então é encerrado.

# 4 Exibidor

O exibidor também possui duas tarefas simples: tratar e exibir as mensagens recebidas e informar o servidor do recebimento das mensagens. Ele pode ser executado com o seguinte comando:

```
$ python exhibitor.py server_addr:port
```

Ao ser executado, ele segue os mesmos passos descritos no emissor. Conecta-se ao servidor, envia OI e recebe identificador.

A exemplo do emissor, o recebedor também executa em loop infinito. Porém, fica esperando pela chegada de mensagens no socket. Quando uma mensagem é recebida, ela então é processada. O exibidor espera por três tipos de mensagem: (i) FLW - que representa um desligamento do servidor, (ii) MSG - que representa uma mensagem recebida de algum emissor e (iii) CLIST - que contém a lista de exibidores e emissores conectador àquela sessão de mensagens. Ao receber um FLW, o exibidor se encerra porque não há mais um servidor disponível. Ao receber uma MSG ou CLIST, o conteúdo é mostrado na tela e um OK (ACK) é enviado ao servidor. Assim como o emissor, o exibidor também envia um FLW ao ser interrompido por teclado.

# 5 Servidor

O servidor é a estrutura mais complicada desta arquitetura. Ele é responsável por repassar todas as mensagens e tem de lidar com as conexões realizadas por emissores e exibidores. Além disso, ele deve ser capaz de controlar os identificadores e as associações entre emissores e exibidores. Além disso, deve ser capaz de realizar suas tarefas de uma maneira orientada a eventos. O servidor pode ser executado com o seguinte comando:

```
$ python server.py listen_port
```

Ao ser executado, o servidor realiza uma abertura passiva na porta indicada na linha de comando. Além disso, são criadas três estruturas que são de extrema importância para o funcionamento do servidor: uma lista de sockets que guardará os sockets dos clientes conectados e o próprio socket já aberto, um dicionário que mapeia um identificador para um socket e outro dicionário que mapeia um emissor para um exibidor.

A exemplo dos clientes, o servidor também executa em um loop infinito. A cada interação, é verificado se está chegando algum dado no servidor através da função select(). A função select() é um wrapper para uma chamada de sistema blocante que nos permite monitorar eventos de entrada e saída em sockets, arquivos e outras estruturas. Sendo assim, ao chamar a função select para a nossa lista de sockets, o programa fica esperando por um ou mais sockets se tornarem "legíveis", ou seja, que haja dados chegando de algum cliente no servidor. A função então retorna uma lista de sockets com dados a serem lidos. O programa, então, itera sobre essa lista de forma a processar os dados recebidos em todos os sockets que se tornaram "legíveis".

Entretanto, é necessário se diferenciar os sockets presentes na nossa lista. Em um primeiro momento, teremos apenas o socket de recebimento de conexões do servidor. Portanto, caso um dos sockets legíveis seja esse socket, devemos processar a conexão, criar um novo socket para aquele cliente e adicioná-lo à lista. Caso contrário, sabemos que se trata de uma mensagem de algum cliente. Essa mensagem é, então, recebida e lida. Como o servidor faz o papel de middle-man, ele deve ser capaz de processar todos os tipos de mensagem descritas no protocolo.

Ao receber uma mensagem do tipo OI de um cliente, esse cliente é adicionado nas estruturas de controle e é diferenciado entre emissor e exibidor. Essa diferenciação pode ser feita pelo identificador de origem enviado na mensagem. Possíveis erros como um emissor tentar se associar a um exibidor inválido são tratados e exibidos no servidor.

Ao receber mensagens de qualquer outro tipo, é realizada uma checagem de identidade da mensagem. Caso o identificador do emissor seja diferente ao socket designado a ele, a mensagem é rejeitada e a conexão é fechada. Isso evita que haja *spoofing* na troca de mensagens. Além disso, para cada tipo de mensagem há um tratamento diferente. Mensagens de OK e ERRO exibem logs na tela. Mensagem de FLW faz com que o servidor envie uma mensagem OK para o cliente; envie FLW para o exibidor ligado a ele, caso ele seja um emissor; o apague das estruturas de controle do servidor e seus exibidores, caso se aplique; e feche a conexão com o cliente. Mensagem do tipo MSG causa redirecionamento da mensagem e envio de OK ou ERRO para o remetente da mensagem. Interessante notar que há um overhead em se "desmontar" e "montar" o quadro para reenvio. Lembrando que é possível enviar uma mensagem para o identificador de um emissor desde que ele esteja associado a um exibidor. Ademais, a mensagem do tipo CREQ faz com que o servidor processe o payload da mensagem CLIST e a envie para o mesmo destino da mensagem original (incluindo broadcast). Por fim, assim como os clientes o servidor realiza um broadcast da mensagem FLW e fecha todas as conexões ao ser interrompido por teclado.

# 6 Discussão

Aplicativos de troca de mensagens são de extrema importância no mundo atual. Apesar da maioria de seus usuários desconhecerem a tecnologia por trás dessas ferramentas, ela é complexa e deve trazer algumas garantias básicas para seus usuários como a entrega das mensagens e idoneidade dos remetentes.

Além disso, um servidor deve ser capaz de suportar a carga de milhões de conexões para ser efetivo. Uma abordagem que vem sendo adotada atualmente é a orientação a eventos. Essa abordagem é efetiva porque evita o desperdício de recursos computacionais em relação a abordagens como a criação de threads por conexão. Essa última, por exemplo, carrega um overhead grande na criação das threads e é limitada pela capacidade do escalonador do sistema operacional. A orientação a eventos evita esse overhead, já que podemos receber conexões simultâneas em uma só thread. Tudo isso é possível graças à chamada de sistema select() que é capaz de monitorar os sockets e avisar quando há dados em cada um deles.

Ademais, este trabalho foi bastante prazeroso de se implementar devido à real utilidade do protocolo no dia-a-dia. A parte mais desafiadora foi o tratamento de mensagens no servidor, já que cada uma possui uma peculiaridade que deve ser observada com atenção. Por fim, a implementação do protocolo de terminação do servidor também foi trabalhosa, mas foi bem sucedida no final.