Roteador Assíncrono de Latência 0.

ALUNO: Lucas Rangel Gazire
Graduando em Matematica Aplicada

IMECC-UNICAMP

Introdução

O mercado financeiro hoje necessita de ferramentas de alta performance em prol do melhor desempenho de negociação entre a bolsa de valores, corretoras de todo o país e entre bolsas ao redor do mundo. A demanda crescente exige cada vez mais de precisão e agilidade por parte das negociações feitas.

Uma das necessidades é a de registrar os negócios e notícias de maneira que sempre que houver uma oferta de tipo contrário ou uma notícia solicitada chegar o usuário a obtenha da melhor maneira possível. Para isso foi desenvolvido um roteador de mensagens, o RAL0.

Objetivo

O objetivo do RAL0 é fornecer a maior quantidade de informação solicitada em um curto espaço de tempo sem que o usuário sofra algum impacto.

Explicação do funcionamento – RAL0

Um roteador é um “filtro” onde uma determinada mensagem é recebida e, caso exista um assinante, enviada caso contrário é ignorada.

O problema reside na performance de resposta do roteador, para isso utilizamos uma HEAP (uma árvore binária, pode ser de máximo, de mínimo ou de responsabilidade, no nosso caso o último), assim sempre que um elemento chegar a árvore é varrida horizontalmente, ao ser encontrado o tipo, e caso haja especificidade varemos os nós até encontrar o ideal utilizando o HeapSort.

Todo esse processo é feito de maneira assíncrona, ou seja, o usuário envia uma solicitação e as partes (bolsa ou corretora) criam as regras para atender essa solicitação e enviam a resposta para o usuário. Um mesmo usuário pode obter resposta por partes diferentes, o que garante melhor tempo de execução e entrega.

Tecnicamente, o RALO é um roteador de mensagem chave e valor. Existem duas operações básicas:

- Adicionar regra, que é algo como "caso o campo "xpto" seja igual a "foo" e o campo "lalala" seja igual a "lilili", chame esse callback
- Despachar uma mensagem. Quando uma mensagem for despachada, o Ralo checará as regras e chamará os callbacks de acordo com elas. Caso nenhuma regra seja correspondida, o Ralo não faz nada com a mensagem.

Código resumidamente:

```
class Message

{

  string get(const string& key);

     void set(const string& key, value);
```

```
    size_t size();


  FieldsMap::const_iterator find(const string& key) const;

  FieldsMap::const_iterator begin() const;

  FieldsMap::const_iterator end() const;


};


class MessageRouter

{

  int AddRule(const Rule& rule, MessageCallback callback);

  int Dispatch(cons Message& msg);

};
```

Tanto a regra quanto a mensagem são estruturas de chave e valor, somente string. Em C++ seria algo como map<string, string> e em .NET Dictionary<string, string>.

Definições

Mensagem: objeto que contém n campos arbitrários, que são acessados pelo nome. Ex: message.set("key", "value") -> void; message.get("key") -> "value".

Componente: Trecho do sistema que faz parte do terminal e se comunica usando do Ralo.

Provider: componente que fornece informações para outros componentes do terminal. Esse componente responde mensagens de pedido de informação, assinatura etc. Ex: AeCliP, provider que fornece aos módulos consumidores as informações disponíveis através do AeCli.

Condição: par ordenado com o nome do campo e o valor para o qual a condição será considerada satisfeita. Ex: instrument = "PETR4"

Regra: conjunto de condições. Todas as condições devem ser satisfeitas para que a regra seja considerada satisfeita. Ou seja, as condições são tratadas com AND. Ex: instrument = "PETR4" AND InfoType = OrderBook

MessageRouter: outro nome para o RALO, o Roteador de mensagens. É usado por todos os módulos do sistema para comunicação. Mantém a lista de regras e roteia as mensagens para os módulos de acordo com elas.

Messagerouter

O roteador de mensagens (MessageRouter) foi criado devido à necessidade de atender os seguintes requisitos:

- Independência de protocolo de comunicação e de provedor de sinal
- Desacoplamento entre os módulos do terminal
- Performance

Os módulos do terminal são totalmente desacoplados. Um módulo do terminal que precisa de cotação do ativo XYZ não sabe quem responderá à solicitação. Quando um módulo precisa de alguma informação (como o preço de um ativo) ou assinar algo (livro de ofertas, por exemplo), esse módulo monta uma mensagem com a solicitação e faz uma chamada ao MessageRouter para que ele roteie a mensagem. Os provedores que atendem essa solicitação já registraram uma regra para receber esse tipo de solicitação. Exemplo:

- Módulo Provedor que tem informações de ativos da Bovespa registra uma regra para que todas as solicitações com SecurityExchange=XBSP e InfoType=LastPrice sejam roteadas para ele
- Módulo Consumidor prepara uma mensagem com solicitação de último preço (MsgId=1 ; MsgType=Request ; SecurityExchange=XBSP ; Symbol=PETR4 ; InfoType=LastPrice)
- Módulo Consumidor, antes de enviar a mensagem preparada, registra uma regra para receber a resposta (ReferentToMsgId=1)
- Módulo Consumidor envia a mensagem preparada
- Módulo Provedor receberá a mensagem devido à regra previamente registrada. Ele então enviará uma mensagem de resposta com o preço solicitado e com o campo ReferentToMsgId=1.

Note que esse é o funcionamento baixo nível da interação dos módulos com o MessageRouter. Os módulos do terminal usam uma biblioteca que abstrai as partes repetitivas, como inserir uma regra para receber a resposta de uma solicitação.

Segue um exemplo de código de um provider:

```
MessageRouter::Rule conditions;

//

// subscriptions

//

conditions.push_back(

     make_pair(Ae::ControlFields::MsgType,


Ae::ControlFieldValues::MsgType_SubscriptionIncrementalOnly));

router_.AddRule(

     conditions,
```

```
    bind(&PriceGeneratorTabajex::OnSubscriptionMessage,          this,
std::placeholders::_1),

    MultithreadedMessageRouter::ThreadAffinity_SameThreadOnly);



conditions[0] = make_pair(Ae::ControlFields::MsgType,


Ae::ControlFieldValues::MsgType_SubscriptionWithSnapshot);

router_.AddRule(

    conditions,

    bind(&PriceGeneratorTabajex::OnSubscriptionMessage,          this,
std::placeholders::_1),

    MultithreadedMessageRouter::ThreadAffinity_SameThreadOnly);




conditions[0] = make_pair(Ae::ControlFields::MsgType,


Ae::ControlFieldValues::MsgType_CancelSubscription);

router_.AddRule(

    conditions,

    bind(&PriceGeneratorTabajex::OnSubscriptionCancelMessage,    this,
std::placeholders::_1),

    MultithreadedMessageRouter::ThreadAffinity_SameThreadOnly);
```

Uso

A Api usada pelos módulos do terminal seguirá o padrão Continuation-passing para que toda iteração do terminal com a API seja 100% assíncrona e não cause travamento ou impressão de não-responsividade ao usuário.

Continuation-passing style

Continuação que passa o estilo (CPS) é um termo usado dentro para descrever um estilo de programar wherein as funções nunca retornam. Em vez dos valores "de retorno" como no mais familiar estilo direto, uma função escrita no CPS faz exame de um explícito que é significado receber o resultado da computação executou dentro da função. Assim, quando uma sub-rotina é invocada dentro de uma função do CPS, a função de chamada é requerida fornecer um procedimento a ser invocado com o valor "do retorno" da sub-rotina. devido a sua natureza "non-non-intuitive", CPS é usado mais freqüentemente por compiladores do que por programadores. Os compiladores funcionais e da lógica usam frequentemente o CPS como onde um compilador para uma língua imperativa ou processual pôde empregar .

Em , o mais simples de funções do dirij-estilo é a função da identidade:

```
(lambda (x) x)
```

qual no CPS se torna:

```
(lambda (retorno de x) (retorno x))
```

onde *retorno* é claramente o argumento da continuação.

Conclusão

O RAL0, apresenta vantagens em curto e longo prazo pois é uma ferramenta de uso simples e performática, o único problema atual é caso uma solicitação não tenha um atendente ela acaba sendo ignorada, para futuras implementações deveria ser criado um meio de armazenar até chegar uma resposta satisfatória, mas isso demanda uma infra muito maior e um custo alto.