УДК 681.3

© 2016 г. **В.А. Антипов,** д-р техн. наук,
**О.В. Антипов,** канд. техн. наук,
**А.Н. Пылькин** д-р техн. наук
(Рязанский государственный радиотехнический университет)

# ИНФРАСТРУКТУРА СЕРВИСА УВЕДОМЛЕНИЙ В СИСТЕМАХ ПУБЛИКАЦИЯ/ПОДПИСКА

Рассмотрена инфраструктура сервиса уведомлений, отличающаяся от других прототипов рядом важных характеристик: вместо единой схемы маршрутизации в ней реализован набор алгоритмов маршрутизации, в который можно легко добавлять новые, что позволяет тестировать и сравнивать различные алгоритмы, в том числе вновь разработанные, в единой унифицированной среде, в то время как другие сервисы уведомлений обычно поддерживают фиксированный набор ограничений фильтрации; в разработанной системе используется мощная и гибкая, легко расширяемая структура фильтрации, которая поддерживает оптимизацию маршрутизации; поддерживаются функции воспроизведения прошедших уведомлений и фабрик сервисов, для реализации которых введены события подписки и отмены подписки, которые автоматически публикуются инфраструктурой системы, когда клиент оформляет или отменяет подписку.
***Ключевые слова**:* сервис уведомлений, события, поставщики, потребители, журнал событий, фабрика сервисов, публикация/подписка.

DOI: 10.22250/isu.2016.49.13-24

## Введение

Существующие службы уведомления недостаточно разработаны и исследованы, чтобы использоваться в крупномасштабных распределенных средах, большинство из них или централизованы, или используют простые алгоритмы маршрутизации, которые предполагают, что каждый брокер имеет глобальное знание обо всех активных подписках. Все эти подходы имеют серьезные проблемы масштабируемости в крупных распределенных гетерогенных системах. В связи с этим актуальной задачей является разработка сервиса уведомлений, обеспечивающего возможность улучшения масштабируемости алгоритмов маршрутизации [1 – 4].

Цель работы – разработка инфраструктуры сервиса уведомлений систем Публикация/Подписка, отличающейся от других прототипов наличием характеристик, поддерживающих масштабируемость алгоритмов маршрутизации.

## Общая архитектура

Предлагаемая инфраструктура сервиса уведомлений разработана для создания приложений на основе событий и состоит из взаимосвязанных брокеров событий, которые делятся на две категории. *Локальные брокеры* представляют собой точки доступа к системе и управляют клиентами. Каждый локальный брокер подключен только к одному маршрутизатору. *Маршрутизаторы* не управляют клиентами. Они перенаправляют входящие уведомления соответствующим соседним объектам. Брокеры подключены к маршрутизаторам, а маршрутизаторы соединены друг с другом посредством механизма обмена сериализированными объектами Java через сокеты TCP или посредством локальных запросов.

В инфраструктуре рассматриваемой системы можно реализовать любой из описанных в [5 – 7] алгоритмов маршрутизации: на основе «наводнения», простую маршрутизацию на основе фильтров, маршрутизации на основе идентичности, покрытия и слияния фильтров. Дополнительно можно реализовать использование в алгоритмах маршрутизации рекламных объявлений. Алгоритмы маршрутизации создаются на основе структуры фильтрации, которая использует простую модель пары данных «имя-значение» и поддерживает расширяемый набор атрибутов фильтров.

## Механизм воспроизведения уведомлений

Инфраструктура включает в себя механизм воспроизведения уведомлений, позволяющий потребителям вновь подписываться на прошедшие уведомления, которые сохраняются в системе. Это бывает необходимо при инициализации новых компонентов, например, если инициализируется список текущих котировок акций, за которыми ведется наблюдение, то пользователи обычно не хотят ждать, пока поступят новые котировки для всех отслеживаемых акций, они хотят, чтобы в списке котировок немедленно отображались текущие цены. Для поддержки воспроизведения событий можно подключить журналы. С помощью *журналов* сохраняются и воспроизводятся определенные типы уведомлений, оформляются соответствующие подписки и сохраняются полученные уведомления. В журналах регулярно или при возникновении определенного события проверяются записанные уведомления и удаляются те, которые больше не нужны. Потребитель определяет свое желание получить прошедшие уведомления, указывая *описание воспроизведения* в оформляемой подписке. Для каждой новой подписки публикуется соответствующее *событие подписки*, содержащее подписку и описание воспроизведения. Если журнал получает такое событие подписки, то он определяет, какие сохраненные уведомления соответствуют этой подписке и описанию воспроизведения. Затем каждое из таких уведомлений включается в событие воспроизведения и публикуется журналом. Таким образом, обеспечивается доставка события воспроизведения конкретному потребителю.

## Концепция фабрик сервисов

В крупных системах публикации/подписки может возникнуть ситуация, когда невозможно создать уведомления по всем подпискам потребителей. В таком

случае требуется наличие механизма, который обеспечит создание всех уведомлений, необходимых в данный момент, и блокирует создание уведомлений, для которых нет подписчиков. Это особенно важно при наличии многоэтапных потоков информации. Когда поставщик прекращает публикацию уведомлений, он может также отменить подписку на уведомления, которые получает. В этом случае поставщик этих событий, для которых нет потребителя, должен быть деактивирован. Для автоматического создания и удаления экземпляров поставщиков в разработанной системе используется концепция *фабрик*. В основе этой концепции лежат события подписки и отмены подписки, которые автоматически публикуются инфраструктурой, когда потребитель оформляет или отменяет подписку.

*Фабрика сервисов* управляет определенным набором экземпляров сервиса и оформляет подписку на те события подписки или ее отмены, которые относятся к уведомлениям, за публикацию которых отвечают эти сервисы. Если фабрика получает событие подписки, которое должно обслуживаться ее сервисами, то она либо активирует существующий сервис, либо создает новый, который и выдает необходимые уведомления. Если фабрика получает событие отмены подписки, то проверяет наличие экземпляров сервиса, которые могут быть деактивированы или удалены, а затем выполняет необходимые действия. Этим процессом можно гибко управлять. Фабрики обращаются к журналам при создании нового сервиса или повторной активации существующего.

Разработанная реализация инфраструктуры сервиса уведомлений состоит из классов и интерфейсов. Кратко опишем самые важные из них (рис. 1). Отношения между классами показаны в виде диаграмм на унифицированном языке моделирования UML (Unified Modeling Language).

### Основные классы

**Event.** Класс Event является базовым классом всех уведомлений (рис. 2). AdminEvent – это важный подкласс класса Event, который используется для обмена управляющими сообщениями.

**Filter.** Класс Filter является базовым классом всех фильтров (рис. 3). У каждого фильтра есть уникальный идентификатор, который используется в алгоритмах маршрутизации. Все подклассы класса Filter должны реализовывать метод boolean match (Event e).

**Subscription.** Класс Subscription расширяет класс Filter и является базовым классом для всех подписок. Новый класс Subscription создается на основе класса Filter и (необязательно) класса ReplayDescription (рис. 3).

**Advertisement.** Класс Advertisement является подклассом Filter и служит базовым классом для всех объявлений (рис. 3). Новый класс Advertisement создается на основе класса Filter.

**EventRouter.** Функции маршрутизатора событий реализуются классом EventRouter (рис. 1), перенаправлением уведомлений и актуализацией таблиц маршрутизации управляет RoutingEngine и включает подклассы RoutingEngine и ServerSocket. ServerSocket используется для объединения подключений классов EventBroker и EventRouter. Если подключение установлено, создается класс

EventRouterConnection, обрабатывающий все операции обмена, связанные с этим подключением. Во время создания этого класса EventRouter может подключаться к другому EventRouter по указанному IP-адресу и номеру порта. В настоящее время не поддерживаются другие методы установки подключений, хотя есть возможность создать несколько экземпляров EventRouter в одной виртуальной машине Java, однако такая возможность редко бывает полезной.

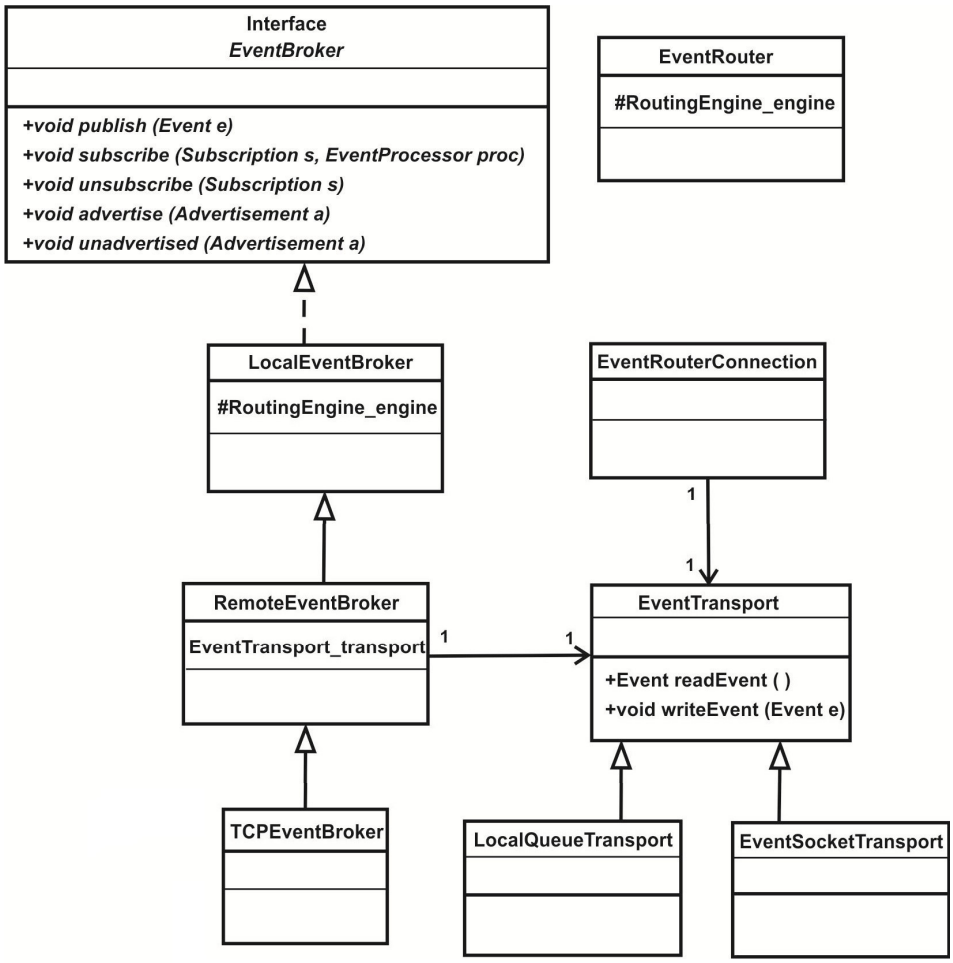


*Рис. 1.* Классы инфраструктуры сервиса уведомлений.

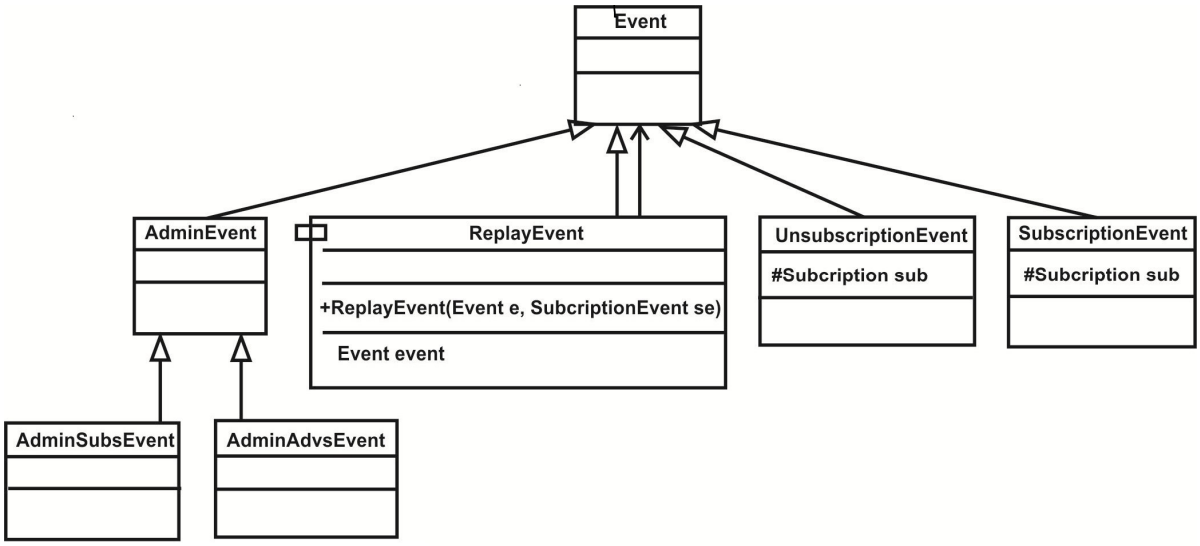


*Рис. 2.* Классы событий.

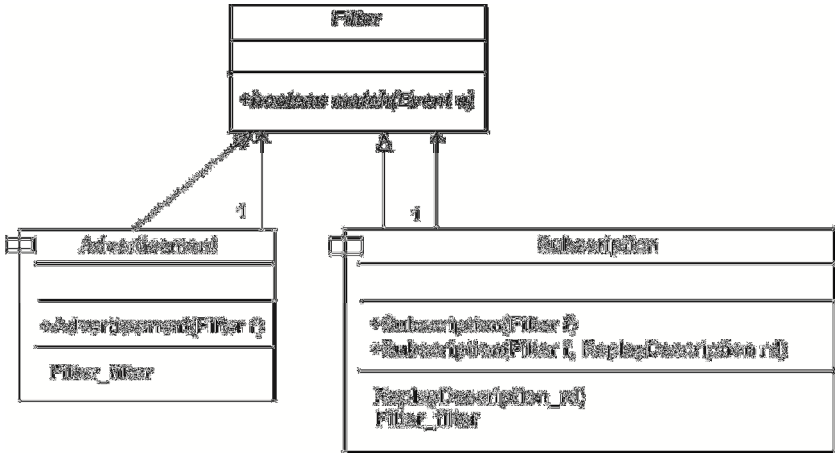


*Рис. 3.* Классы фильтрации.

**RoutingEngine.** Класс RoutingEngine является базовым классом всех алгоритмов маршрутизации (рис. 4). Каждый класс RoutingEngine имеет две таблицы RoutingTable, которые представляют собой соответственно таблицу маршрутизации подписок и таблицу маршрутизации объявлений. Первая таблица используется для маршрутизации уведомлений от поставщиков к потребителям, вторая – для маршрутизации подписок от потребителей к поставщикам. Входящие уведомления обрабатываются в порядке поступления (FIFO). Если обрабатываемое уведомление представляет собой событие AdminEvent, то RoutingEngine соответствующим образом обновляет таблицы маршрутизации и передает надлежащие AdminEvent некоторым из подключенных EventRouterConnection согласно используемому алгоритму маршрутизации. Если уведомление не является событием AdminEvent, RoutingEngine передает EventRouterConnection с соответствующими подписками. Класс RoutingEngine расширяется следующими классами, реализующими соответствующие алгоритмы маршрутизации: Flooding, SimpleRouting, IdentityRouting, CoveringRouting и MergingRouting (рис. 4).

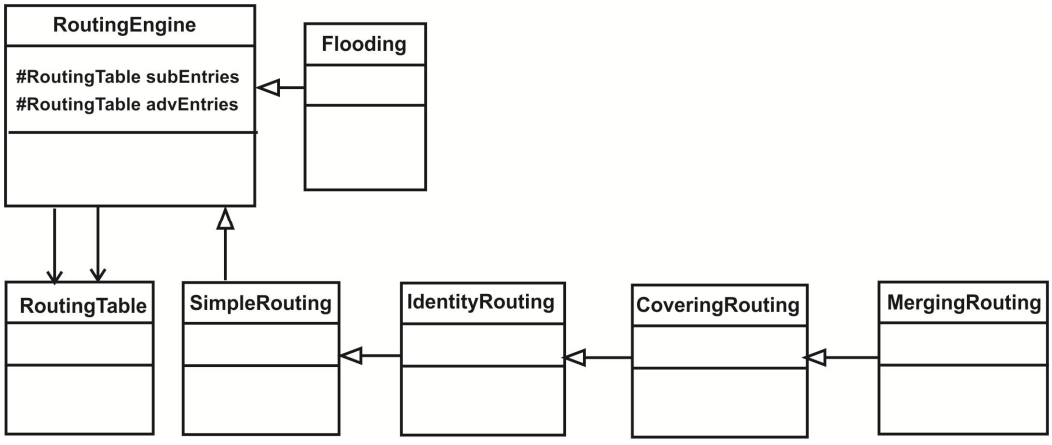


*Рис. 4.* Классы алгоритмов маршрутизации.

**EventRouterConnection.** Класс EventRouterConnection инкапсулирует обмен между EventRouter и EventBroker или другим EventRouter (рис. 1). EventRouterConnection имеет указатель на RoutingEngine и на связанный с ним EventRouter. В RoutingEngine класс EventRouterConnection передает входящие уведомления для дальнейшей обработки. В свою очередь RoutingEngine вызывает

метод void process (Event e) класса EventRouterConnection для отправки соответствующих уведомлений. Класс EventRouterConnection также имеет подкласс EventTransport, который отвечает за текущие низкоуровневые операции обмена.

**EventBroker.** Интерфейсы EventBroker (рис. 1) представляют точки доступа к системе публикации/подписки и поддерживаются API с обычной семантикой:

```
public interface EventBroker {
      void publish (Event e);
      void subscribe (Subscription s, EventProcessor p);
      void unsubscribe (Subscription s);
      void advertise (Advertisement a);
      void unadvertised (Advertisement a);
}
```

**LocalEventBroker** – это EventBroker, который не подключен ни к одному EventRouter (рис. 1), данный класс полезен только при реализации локальной системы публикации/подписки. LocalEventBroker имеет RoutingEngine, который реализует используемый алгоритм маршрутизации.

**RemoteEventBroker** – это LocalEventBroker (рис. 1), дополнительно устанавливающий подключение к EventRouter, по этому подключению идет обмен событиями Event в соответствии с используемым алгоритмом маршрутизации. Само подключение реализуется классом EventTransport.

**TCPEventBroker –** это RemoteEventBroker (рис. 1), который использует EventSocketTransport для подключения к EventRouter по указанному IP-адресу и номеру порта.

**DefaultEventBroker –** это EventBroker, который инкапсулирует экземпляр EventBroker. Он инициализируется с помощью DefaultEventBroker.init (args) в методе main.

**EventTransport.** Класс EventTransport реализует двунаправленное подключение, с помощью которого выполняется обмен экземплярами Event:

```
public abstract class EventTransport {
      public Event readEvent ( );
      public void writeEvent (Event e);
}
```

**EventSocketTransport** – это EventTransport (рис. 1), который использует Socket для обмена событиями Event как сериализированными объектами Java посредством ObjectInputStream и ObjectOutputStream.

**EventQueueTransport** – это EventTransport (рис. 1), который использует локальную очередь для обмена событиями, поэтому данный класс можно использовать только для подключения к партнерам, расположенным на той же виртуальной машине Java.

**EventQueue** – это EventProcessor (рис. 5), который помещает в очередь входящие уведомления. Первое уведомление в очереди можно извлечь с помощью блокирующего метода Event consume() или неблокирующего метода Event tryConsume().

**SubscriptionEvent.** Если потребитель оформляет новую подписку, автоматически создается событие SubscriptionEvent (рис. 2), которое содержит соответствующий класс Subscription. Эта операция выполняется с помощью соответствующего LocalEventBroker.

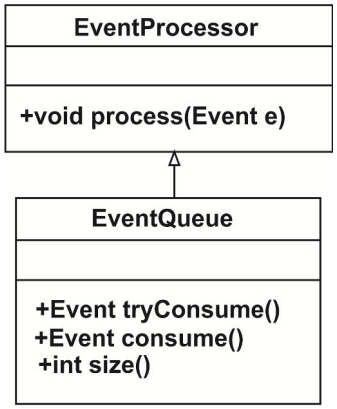


*Рис. 5.* Классы обработчика событий.

**SubscriptionSubscription –** это подписка Subscription, соответствующая событиям SubscriptionEvent, подписка Subscription которых пересекается с заданным фильтром Filter. Кроме того, можно проверить, пересекается ли необязательное описание ReplayDescription подписки с заданным ReplayDescription. SubscriptionSubscription используется журналами и фабриками.

**UnsubscriptionSubscription –** это подписка Subscription, соответствующая событиям UnsubscriptionEvent, подписка которых пересекается с заданным фильтром Filter. UnsubscriptionSubscription используется журналами и фабриками.

**ReplayEvent.** Событие ReplayEvent (рис. 2) публикуется журналами при получении SubscriptionEvent. Оно создается на основе заданного события Event и полученного события SubscriptionEvent. Инфраструктура обеспечивает доставку ReplayEvent только потребителям, имеющим подписку, встроенную в SubscriptionEvent.

## Использование инфраструктуры

Рассмотрим, как можно использовать разработанную инфраструктуру для создания простого распределенного приложения, основанного на механизме публикации/подписки. Пусть у нас есть потенциальные потребители событий ExampleEvent, публикуемых неким поставщиком, который должен быть активным только в том случае, если подписка оформлена хотя бы одним потребителем. Кроме того, новым потребителям должны доставляться десять последних опубликованных ExampleEvent. Это приложение состоит из фабрики, управляющей поставщиком, журналом и, по меньшей мере, одним потребителем. Ниже приводятся классы, необходимые для реализации этого примера и являющиеся частью инфраструктуры, описывается сам пример.

## Реализация события

Класс ExampleEvent расширяет класс Event и содержит метод печати информации. Далее приведена реализация простого события.

*Простое событие*, *определенное пользователем*:

```
public class ExampleEvent extends Event {
      ObjectId _id;
      public ExampleEvent ( ) {
```

```
            _id=new ObjectId ( );
      }
      public String toString ( ) {
            return "Events.example.ExampleEvent: {" + _id + "}";
      }
}
```

## Реализация потребителя

Класс ExampleConsumer реализует потребителя, интересующегося событиями ExampleEvent. Класс реализует интерфейс EventProcessor, предоставляя реализацию метода public void process(Event e). Эта конечная точка привязывается к подписке, соответствующей событиям ExampleEvent.

ExampleConsumer может быть запущен с помощью java Events.example. ExampleConsumer:

*Простой потребитель события*:

```
public class ExampleConsumer implements EventProcessor {
      private EventBroker _broker ;
      private Subscription _sub;
      public ExampleConsumer ( ) {
            _broker = DefaultEventBroker.getEventBroker ( );
            _sub = new Subscription (
                  new EventClassFilter (ExampleEvent.class));
            _broker.subscribe (_sub, this);
      }
      public void process (Event e) {
            System.out.println (e.toString ( )) ;
      }
      public static void main (String args [ ]) {
            DefaultEventBroker.init (args);
            ExampleConsumer consumer = new ExampleConsumer ( );
      }
}
```

## Реализация поставщика

Класс ExampleProducer расширяет класс DefaultEventProducer. Сначала выдается объявление о намерении опубликовать события ExampleEvent, а затем ExampleEvent публикуются каждые 3 секунды. Эта операция выполняется в отдельном потоке во избежание блокировки вызывающего потока. Запустить ExampleProducer можно, вызвав java Events.example.ExampleProducer.

*Простой поставщик событий:*

```
public class ExampleProducer extends DefaultEventProducer {
      boolean _b = true ;
      Advertisement _adv;
      public ExampleProducer ( ) {
            super.setEventBroker (
                  DefaultEventBroker.getEventBroker ( ));
            _adv = new Advertisement (
                  new EventClassFilter (ExampleEvent.class));
            advertise ( _adv );
            Thread th = new Thread ( ) {
                  public void run ( ) {
                        while ( _b ) {
                              publish(new ExampleEvent ( ));
                              try {
                                 Thread.sleep (3000);
                              }
```

```
                        catch (InterruptedException e) { }
                        }
                  }
            };
            th.start ( );
      }
      public void shutdown ( ) {
            _b = false;
            unadvertise ( _adv );
      }
      public static void main(String args[ ]) {
            DefaultEventBroker.init(args);
            ExampleProducer producer = new ExampleProducer( );
      }
}
```

## Реализация журнала

ExampleHistory представляет собой простой журнал, с помощью которого регистрируются и воспроизводятся события ExampleEvent. Если говорить более точно, то осуществляется подписка на ExampleEvent и постоянное отслеживание десяти последних полученных событий. Кроме того, осуществляется регистрация событий SubscriptionEvent, подписка которых соответствует ExampleEvent. Если получено событие SubscriptionEvent, то журнал публикует каждое из записанных событий как ReplayEvent. В приведенном далее примере также показано, как можно использовать анонимный подкласс EventProcessor для передачи соответствующего события в определенный метод. Журнал ExampleHistory может быть запущен с помощью java Events.example.ExampleHistory.

*Простой журнал событий:*

```
public class ExampleHistory {
      EventBroker _broker ;
      Subscription _sub;
      SubscriptionSubscription _subSub;
      int _n = 0;
      Event [ ] history = new Event [10];
      public ExampleHistory ( ) {
            _broker = DefaultEventBroker.getEventBroker ( );
            _sub = new Subscription (
                  new EventClassFilter(ExampleEvent.class));
            _broker.subscribe (_sub, new EventProcessor ( ) {
                  public void process ( Event e ) {
                        processExampleEvent (( ExampleEvent) e );
                  }
            } );
            SubscriptionSubscription _subSub =
                  new SubscriptionSubscription (
                    new EventClassFilter ( ExampleEvent.class ));
      _broker.subscribe(_subSub, new EventProcessor( ){
                  public void process ( Event e ) {
                     processSubEvent (( SubscriptionEvent ) e );
                  }
            } );
      }
      protected void processExampleEvent ( ExampleEvent ee ) {
            history [ _n%10 ] = ee;
            _n++;
      }
      protected void processSubEvent ( SubscriptionEvent se ) {
            ReplayEvent re;
```

```
            ExampleEvent ee;
            for ( int i = 0; i < 10; i++ ) {
                  ee = ( ExampleEvent ) history [ i ];
                  if ( ee! = null ) {
                        re = new  ReplayEvent ( ee, se );
                        _broker.publish ( re );
                  }
            }
      }
      public static void main ( String args [ ] ) {
            DefaultEventBroker.init(args);
            ExampleHistory history = new ExampleHistory ( );
      }
}
```

## Реализация фабрики

ExampleFactory представляет собой простую фабрику, управляющую одним ExampleProducer. Фабрика подписывается на события SubscriptionEvent и UnsubscriptionEvent, связанные с ExampleEvent. Если фабрика получает SubscriptionEvent, то активизирует поставщика, публикующего события ExampleEvent, если он неактивен, и добавляет идентификатор определенной подписки в набор активных подписок. Если она получает UnsubscriptionEvent, то удаляет идентификатор определенной подписки из своего набора активных подписок и деактивирует поставщика, если у него не осталось подписчиков. В примере также показано, как использовать оператор instanceof для демультиплексирования события. ExampleFactory можно запустить с помощью java Evens.example.ExampleFactory.

*Простая фабрика сервисов:*

```
public class ExampleFactory implements EventProcessor {
      EventBroker _broker;
      ExampleProducer _producer;
      SubscriptionSubscription _subSub;
      UnsubscriptionSubscription _unsubSub;
      HashSet _subs = new HashSet ( );
      public ExampleFactory ( ) {
            _broker = DefaultEventBroker.getEventBroker ( );
            _subSub = new SubscriptionSubscription (
                  new EventClassFilter ( ExampleEvent.class ));
            _broker.subscribe (_subSub, this);
            _unsubSub = new UnsubscriptionSubscription (
                  new EventClassFilter ( ExampleEvent.class ));
            _broker.subscribe (_unsubSub, this);
      }
      protected void addSubscription ( SubscriptionEvent se ) {
            if (_producer== null) {
               System.out.println("ExampleFactory:Starting
                                                      Service");
               _producer = new ExampleProducer ( );
            }
            _subs.add ( se.getSubscription ( ).getId ( ));
      }
      protected void removeSubscription(UnsubscriptionEvent ue) {
            _subs.remove ( ue.getSubscription ( ).getId ( ));
            if ( _subs.size ( ) == 0  &&  _producer! = null ) {
               System.out.println("ExampleFactory : Stopping
                                                      Service");
               _producer.shutdown ( );
               _producer = null;
```

```
            }
        }
        public void process ( Event e ) {
            if ( e instanceof SubscriptionEvent ) {
                addSubscription (( SubscriptionEvent ) e );
                return;
            }
            if ( e instanceof UnsubscriptionEvent ) {
                removeSubscription ((UnsubscriptionEvent ) e );
                return;
            }
        }
        public static void main ( String args [ ] ) {
            DefaultEventBroker.init ( args );
            ExampleFactory  ef = new  ExampleFactory ( );
        }
}
```

### Запуск маршрутизатора

Маршрутизатор EventRouter запускается с помощью следующей команды:

```
java Events.EventRouter [ -lport <номер порта> ]
            [ -rport <номер порта> ] [ -rhost <имя хоста> ].
```

Параметр lport указывает порт, который «прослушивается» маршрутизатором на наличие входящих подключений. Если параметр опущен, то используется порт 8020. Если задан параметр rport или rhost, то EventRouter пытается подключиться к родительскому EventRouter, расположенному на указанном хосте (или на локальном хосте, если параметр rhost опущен) и «прослушивает» заданный порт (или порт 8020, если параметр rport опущен) на наличие входящих подключений.

### Процедура использования

Рассмотрим, как выполнить наш простой пример. Сначала необходимо выбрать алгоритм маршрутизации. Для этого нужно отредактировать и скомпилировать класс Config. Можно выбрать любой из описанных алгоритмов [5 – 7]: на основе «наводнения», простую маршрутизацию на основе фильтров, маршрутизации на основе идентичности, покрытия или слияния фильтров с использованием рекламных объявлений или без них. После выбора алгоритма маршрутизации необходимо запустить EventRouter, выполнив команду *java Events.EventRouter.* Затем нужно запустить фабрику командой *java Events.example.ExampleFactory* и активировать журнал *java Events.example.ExampleHistory,* и, наконец, активировать потребителя *java Events.example.ExampleConsumer.*

После короткой задержки потребитель начинает получать опубликованные события ExampleEvent. Если через некоторое время мы запустим второго потребителя, то он также получит 10 (не более) последних экземпляров ExampleEvent, записанных в журнал. Конечно, можно и напрямую запустить поставщика, не используя при этом фабрику: `java Events.example.ExampleProducer.`

### Заключение

Проведен анализ существующих сервисов уведомлений и обоснован выбор реализуемой инфраструктуры на основе событий. В отличие от большинства дру-

гих проектов, разработанный сервис уведомлений поддерживает различные алгоритмы маршрутизации (с «наводнением», а также алгоритмы на основе фильтров, как с объявлениями, так и без них) и позволяет легко добавлять новые алгоритмы и сравнивать их в единой унифицированной среде [4 – 7]. Использование в качестве основы для экспериментов разработанного прототипа позволило повысить достоверность полученных результатов по сравнению с результатами на основе моделирования. Кроме того, предложенная инфраструктура предоставляет базовую поддержку воспроизведения прошедших событий и фабрики сервисов, необходимость использования которых стала очевидной во время реализации приложения торговли акциями, использующего журналы и фабрики.

Рассмотрены вопросы общей архитектуры системы и использования алгоритмов маршрутизации, реализация механизма воспроизведения уведомлений и основных используемых классов, концепция фабрик сервисов. Описывается, как можно использовать данную инфраструктуру для создания простого распределенного приложения на основе архитектуры Публикация/Подписка. Рассмотрена реализация основных механизмов (события, потребителя, поставщика, журнала, фабрики), процедура запуска маршрутизатора и использования приложения.

ЛИТЕРАТУРА


1. *Антипов В.А.*, *Антипов О.В.*, *Пылькин А.Н.* Глава 3: Интеграция распределенных программных приложений на основе коммуникационной парадигмы Публикация/Подписка // Математические и компьютерные методы в технических, гуманитарных и общественных науках: коллективная монография. – Вып. 3. – Пенза; М.: Приволжский Дом знаний; Московский университет им. С.Ю. Витте, 2013. – С. 36-67.
2. *Антипов В.А.*, *Антипов О.В.*, *Пылькин А.Н.* Интеграция распределенных программных приложений на основе маршрутизации по содержимому сообщений // Вестник Рязанского государственного радиотехнического университета. – 2014. –№ 47. – С.75-83.
3. *Антипов В.А.*, *Антипов О.В*., *Чехов А.П.* Сетевая инфраструктура единого информационного пространства виртуальных медицинских организаций // Биомедицинская радиоэлектроника. – 2014. – №7. – С.73-81.
4. *Антипов В.А*., *Антипов О.В.*, *Пылькин А.Н.* Обобщенная структура алгоритмов маршрутизации на основе содержимого сообщений // Вестник Рязанского государственного радиотехнического университета. – 2014. – № 48. – С. 76-82.
5. *Антипов О.В.* Алгоритмы маршрутизации на основе содержимого сообщения. // Математическое и программное обеспечение вычислительных систем: Межвузовский сборник научных трудов. – Рязань: РГРТУ, 2014. – С. 14-25.
6. *Antipov V.A., Antipov O.V., Pylkin A.N.* Dynamic Publish/Subscribe Systems // 2014 international conference on computer technologies in physical and engineering applications. – SPb. – 2014. – P. 11.
7. *Антипов В.А.*, *Антипов О.В*., *Пылькин А.Н.* Алгоритмы маршрутизации сообщений в системах публикация/подписка // Радиотехника. – 2015. – №11. – С.48-54.



*Статья представлена к публикации членом редколлегии Е.Л. Ереминым.*



*E-mail:*
*Антипов Владимир Анатольевич – parfant@rambler.ru;*
*Антипов Олег Владимирович – oleg@anegmetex.com;*
*Пылькин Александр Николаевич – pylkin.a.n@rsreu.ru.*