МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное автономное образовательное
учреждение высшего образования
ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

Институт математики, механики и компьютерных наук
им. И. И. Воровича

Направление подготовки
Прикладная математика и информатика

Кафедра информатики и вычислительного эксперимента

РЕАЛИЗАЦИЯ ИНТЕРВАЛЬНОГО ВРЕМЕНИ В RABBITMQ

Выпускная квалификационная работа
на степень бакалавра студентки
А. А. Мухаррам

Научный руководитель:
ст. преп. В. Н. Брагилевский

Ростов-на-Дону
2015

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Отсутствие глобального соглашения о времени может привести к возникновению ошибок в системе. В таких случаях принято говорить о необходимости синхронизации часов. Синхронизация часов связана с пониманием порядка следования во времени событий конкурентных процессов. Она может быть полезна в тех случаях, когда необходимо синхронизировать обмен сообщениями, контролировать совместное использование ресурсов и выполнение совместной работы несколькими процессами. Таким образом, задача синхронизации в распределенной системе - обеспечить возможность принятия согласованного решения процессами о порядке следования событий.

Одним из вариантов решения данной проблемы является использование логических часов. Этот механизм впервые был предложен и реализован Лэсли Лампортом в 1978 году. В своей статье [1] он указал, что обычно имеет значение не точное время выполнения события процесса, а его порядок.

Лампорт определил логические часы как способ присвоить номер событию, где под номером понимается время, в которое наступило событие. Для каждого процесса P_i определяются часы C_i как функция, которая ставит в соответствие каждому событию a процесса P_i число $C_i(a)$. Система в целом описывается с помощью функции C, которая ставит в соответствие каждому событию b число $C(b)$, где $C(b) = C_j(b)$, если b - событие процесса P_j. С помощью этой функции можно сравнивать время наступления событий в распределенной системе.

Интервальные отметки времени - один из алгоритмов логических часов. В данной работе описан процесс разработки плагина для брокера сообщений RabbitMQ, реализующего мониторинг событий брокера на основе интервальных отметок времени. С помощью данного плагина можно восстановить цепочку событий между любыми двумя событиями системы. Под событием в RabbitMQ будет подразумеваться отправка или получение сообщения.

ГЛАВА 1

ЛОГИЧЕСКИЕ ЧАСЫ

Эта глава будет посвящена рассмотрению синхронизации процессов посредством нескольких алгоритмов логических часов.

Введем основные понятия. Для синхронизации логических часов Лампорт определил не рефлексивное, транзитивное отношение под названием «происходит раньше», которое удовлетворяет следующим 3 условиям:

- Если a и b — события, происходящие в одном и том же процессе, и a происходит раньше, чем b, то отношение $a \to b$ истинно.
- Если a - отправка сообщения одним процессом, а b - это получения этого сообщения другим процессом, то отношение $a \to b$ истинно.
- Если $a \to c$ и $c \to b$, тогда из $a \to b$

Два отдельных события a и b конкурентные, если оба отношения $a \to b$ и $b \to a$ несправедливы.

Основываясь на введенном отношении и функции логических часов, запишем условие: для любых двух событий a и b, если $a \to b$ истинно, то $C(a) < C(b)$. Ясно, что это условие выполняется, если события a и b удовлетворяют одному из условий перечисленных выше.

Рассмотрим последовательность событий, происходящих между тремя процессами, изображенную на рисунке 1.1. Процессы запущены на разных машинах, каждая из которых имеет собственные

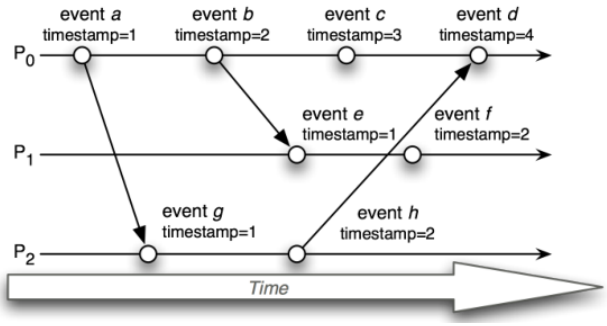

Рисунок 1.1 — неупорядоченная последовательность событий

часы и скорость работы. Каждый процесс поддерживает свой глобальный счетчик, который увеличивается на единицу перед тем, как присвоить новому событию временную отметку. Анализируя временные отметки событий, изображенных на рисунке 1.1, можно заметить несколько особенностей. Событие g, событие изображающее получение сообщения, посланного событием a, имеет такую же временную отметку, что и событие a, хотя совершенно ясно, что оно произошло после события a. Событие e имеет временную отметку меньше, чем событие отправившее ему сообщение (событие b).

1.1. Временные отметки Лампорта

Алгоритм Лампорта исправляет ситуацию, описанную выше, перенумеровывая временные отметки так, чтобы для событий, относящихся к отправке и получению сообщений, выполнялось отношение «происходит раньше».

Правила:

- Каждый процесс имеет счетчик, который увеличивается на единицу перед каждым внутренним событием процесса. События-ми процесса считается получение и отправка сообщений.
- К сообщению при отправке прикрепляется значение счетчика процесса.

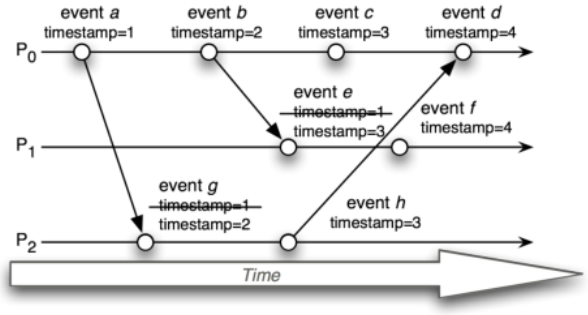

Рисунок 1.2 — упорядоченная последовательность событий

- При получении сообщения, если значение счетчика процесса-получателя меньше временной отметки полученного сообщения, процесс-получатель меняет значение счетчика на значение полученной временной отметки, иначе ничего не меняется.

Применив этот алгоритм к последовательности сообщений, изображенной на рисунке 1.1, мы получим правильно упорядоченный поток сообщений среди событий, связанных причинно-следственной связью (рисунок 1.2). Каждое событие системы теперь имеет временную отметку, которая называется временной отметкой Лампорта. В условиях выполнения правил данного алгоритма, для любых двух событий системы a и b , если отношение $a \to b$ истинно, то $L(a) < L(b)$, где $L(x)$ - временная отметка Лампорта для события x.

1.2. Векторные отметки времени

Благодаря алгоритму Лампорта события в системе имеют единую последовательность, однако из сравнения временных отметок Лампорта $L(a)$ и $L(b)$ нельзя сделать вывод о взаимосвязи между событиями a и b. Причинно-следственная связь может быть установлена посредством векторных отметок времени (vector timestams). Этот алгоритм был независимо предложен Маттерном в 1989 г.(Mattern) и Фиджем в 1991 г. (Fidge).

Векторная отметка времени в системе из N процессов - целочисленный вектор длины N.

Правила для использования векторных отметок времени:

- Каждый процесс в системе имеет свою локальную копию вектора (вектор V_i для процесса P_i), которая инициализируется 0: $V_i[j] = 0, i, j = 1..N$
- Процесс P_j перед каждым новым внутреннем событием увеличивает свою компоненту вектора на единицу: $V_j[j] = V_j[j] + 1$
- Сообщение отправляется процессом P_i вместе с векторной отметкой времени V_i
- При получении сообщения процесс P_j сравнивает полученную временную отметку t со своим локальным вектором поэлементно, устанавливая каждую компоненту локальной временной отметки как максимум из двух значений: $V_j[i] = max(V_j[i], t[i]), i = \overline{1, N}$

Сравниваются векторные отметки по определению:

$$V = V', \text{ если } V[i] = V'[i],\ i = \overline{1, N}$$
$$V \leq V', \text{ если } V[i] \leq V'[i],\ i = \overline{1, N}$$

С помощью алгоритма векторных часов получаем следующие утверждения:

Если отношение $a \to b$ истинно, то $V(a) < V(b)$

Если $V(a) < V(b)$, то отношение $a \to b$ истинно

Два события a и b называются конкурентными, если высказываение $V(a) \leq V(b)$ or $V(b) \leq V(a)$ ложно

Рассмотрим последовательность событий, изображенную на рисунке 1.3. Можно заметить, что события a и e конкурентные,т.к. не каждый элемент одного вектора меньше или равен соответствующему элементу другого, а события b и c взаимосвязаны.

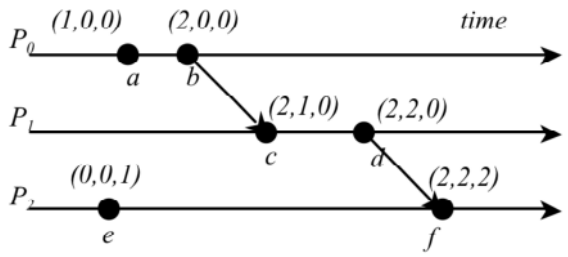

Рисунок 1.3 — векторный алгоритм

1.3. Счетчики дерева интервалов: Логические часы для динамических систем

Хотя отслеживание взаимосвязей между событиями в системах с динамическим изменением числа участников возможно с помощью модифицированного алгоритма векторных часов, в большинстве таких алгоритмов наблюдается чрезмерный структурный рост, а также локализованное удаление не является поддерживаем процессов. Решение этих проблем возможно с использованием алгоритма счетчиков дерева интервалов (Interval Tree Clocks, [2]).

Рассмотрим, следуя [2], возможности алгоритма интервальных отметок времени и понятия, на которых он основывается.

Данному механизму не требуются глобальные идентификаторы, он способен автономно создавать, удалять и повторно использовать их без помощи глобальной координации; любой объект может создать дочерний объект и количество участников может быть сокращено путем присоединения к произвольным парам объектов; метки могут расти или сокращаться, адаптируясь к динамической природе системы.

Механизм отслеживания причинно-следственной связи для алгоритма интервального времени моделируется посредством ряда основных операций: порождения нового процесса ($fork$), события ($event$) и слияния ($join$), воздействующих на интервальные отметки времени (логические часы), чья структура представляет собой пару

(i,e), образованную идентификатором и событием, в котором и содержатся все возможные причинно-следственные связи.

Основная идея алгоритма заключается в том, что идентификатор каждого участника является множеством интервалов, которые используются для увеличения компоненты событие при наступлении внутреннего события процесса и для передачи последующим элементам при порождении нового процесса.

Рассмотрим операции над интервальными отметками времени. При выполнении операция **fork** сохраняется компонента событие, а идентификатор делится на два непересекающихся интервала: $fork(i,e) = ((i_1,e),(i_2,e))$ Операция $peek$ - частный случай операции $fork$: $peek(i,e) = ((0,e),(i,e))$

Операция **event** добавляет новое событие к временной отметке (не анонимной, $(0,e)$) так, что, если $event(i,e) = (i,e')$, то $e < e'$. Обнаружение причинно-следственной связи на основе интервальных отметок осуществляется посредством сравнения компонент событие.

Операция **join** осуществляет слияние двух отметок: $join((i_1,e_1),(i_2,e_2)) = (i_3,e_3)$, где $e_3 > e_2$, $e_3 > e_1$, $e_3 = e_2 \sqcup e_1$, $i_3 = f(i_2,i_3)$, i_3 уникален на уровне системы.

Классические операции отправки, получения и синхронизации реализуются как композиция основных операций:

$$send = peek \circ event$$
$$receive = event \circ join$$
$$sync = fork \circ join$$

В ITC используется исходная метка ($seed$), $(1,0)$, из которой можно получить необходимое число участников N с помощью применения к ней операции $fork$ N раз. Рассмотрим пример, изображенный на рисунке 1.4. В ITC используется графическая нотация, нижний слой отметки времени - изображение идентификатора, верхние - событие. Работа алгоритма начинается с единственного участника ($seed$) с исходной меткой, которая преобразовывается в две временные отметки

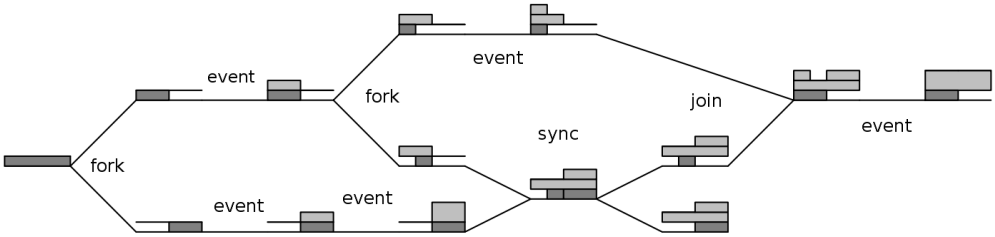

Рисунок 1.4 — интервальные отметки времени

под действием операции fork. На данный момент в системе два участника. В поддереве участника, соответсвующего верхней ветке дерева интервалов,происходит внутреннее событие с последующим за ним порождением нового процесса ($fork$). В поддереве участника, соответствующего нижней ветке, происходит два новых события. В этот момент число участников выросло до трех. Затем один из участников подвергается действию события, в то время как оставшиеся два участника синхронизируются. В результате с помощью операции $join$ происходит объединением двух поддеревьев. Этот пример показывает, как просто ITC адаптируется к числу участников системы.

ГЛАВА 2

RABBITMQ

Мы живем в мире, в котором информация, поступающая в режиме реального времени, постоянно доступна, и написанные нами приложения нуждаются в простом, надежном, быстром способе отправки сообщений многочисленным получателям. А также часто необходимы способы изменения получателей информации, поставляемой нашими приложениями, без постоянного редактирования их кода.

Рассмотрим пример [3]. Разработчик написал аутентификационный модуль для веб-приложения «убийцы», принадлежащего компании, в которой он работает. При каждом обращении к странице, код эффективно взаимодействует с сервером аутентификации, чтобы удостовериться в том, что пользователь действует в рамках прав доступа. Если компания потребует от программиста, написавшего модуль, организовать запись всех удавшихся и неудавшихся попыток аутентификации для дальнейшего анализа данных, то ему потребуется внести изменения в код модуля (предполагается, что с помощью аутентификационного сервера это сделать нельзя). Этого можно было бы избежать при использовании RabbitMQ.

RabbitMQ - брокер сообщений с открытым кодом и сервер с организацией очередей, позволяющий приложениям обмениваться данными с помощью общего протокола или просто ставить в очередь вычисления распределенных рабочих процессов. С помощью

RabbitMQ возможно связывание компонент приложения, реализованных на разных языках программирования.

Теперь представим, что программист построил решение своей задачи на основе RabbitMQ. При каждом обращении к странице, аутентификационный модуль отправляет сообщение, содержащее запрос на авторизацию, брокеру. Аутентификационный сервер подписан на очередь RabbitMQ, в которую поступают эти запросы. Как только запрос будет одобрен, сервер отправляет ответ, который RabbitMQ направляет в очередь, на которую подписан модуль. Так организуется обмен сообщениями между аутентификационным сервером и модулем. При использовании этого решения нет необходимости изменять уже написанный модуль. Все, что теперь нужно сделать, - это написать небольшое приложение, которое связывается с брокером и подписывается на опубликованные модулем запросы.

В настоящее время RabbitMQ используется как небольшими стартапами , так и большими интернет-проектами. Хотя RabbitMQ берет свое начало из области финансовых конгломератов, большинством пользователей RabbitMQ теперь являются технические фирмы.

2.1. Передача сообщений в RabbitMQ

Рассмотрим процесс передачи сообщений в RabbitMQ, основываясь на [3].

Передача сообщений в RabbitMQ осуществлется на основе протокола AMQP. Advanced Message Queuing Protocol (AMQP) - открытый протокол прикладного уровня для промежуточного программного обеспечения, основанное на обмене сообщениями. В 2004 году был запущен проект по созданию AMQP для финансового конгломерата JPMorgan Chase. Отличительными чертами AMQP являются ориентированность на обмен сообщениями, организация очередей, марш-

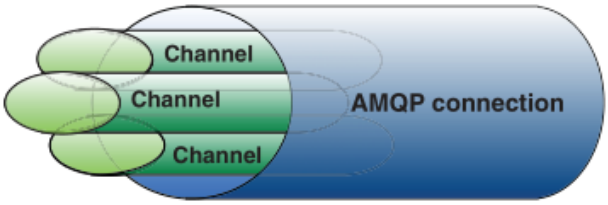

Рисунок 2.1 — каналы и соединение в AMQP

рутизация (точка-точка, публикация и подписка), надежность и безопасность.

Говоря в терминах AMQP, брокер осуществляет маршрутизацию сообщений от поставщика (producer) к подписчику (consumer). Поставщик публикует (отправляет) сообщения посредством RabbitMQ. Сообщение состоит из двух частей: данные, которые передает поставщик, и метка, с помощью которой брокер определяет, кто должен получить копию сообщения. Подписчик соединяется с брокером и подписывается на очередь. RabbitMQ отправляет поступившее в очередь сообщение одному из подписчиков. Теперь сообщение содержит только данные.

Перед тем как отправлять или получать сообщения необходимо установить TCP соединение между приложением и брокером и создать AMQP канал, в котором и будут происходить все действия (публикация, подписка на очередь, получение сообщения и т.д.). Канал - виртуальное соединение внутри настоящего TCP соединения, имеющее уникальный ID. Количество AMQP каналов не ограничено в пределах одного TCP соединения.

Установка и разъединение TCP сессий - дорогая операция для операционной системы. Использование нескольких AMQP каналов, каждый из которых обеспечивает для каждого потока приложения свой собственный путь к брокеру в пределах существующего TCP соединения (рис. 2.1), решение этой проблемы.

В основе AMQP маршрутизации сообщения от поставщика к подписчику (рис. 2.2) лежат три понятия: точка обмена (exchange),

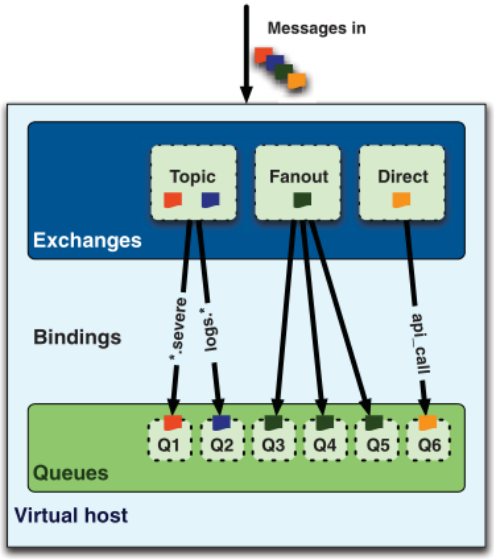

Рисунок 2.2 — AMQP стек

очередь (queue), связь (binding). Точка обмена принимает сообщения от поставщика и направляет в связанную с ней очередь в зависимости от правил, описанных в ее типе. Binding - это связь между точкой обмена и очередью. Сообщения не хранятся в точке обмена, они хранятся в очереди.

Подписчики получают сообщения из очереди одним из двух способов:

1. С использованием AMQP команды $basic.consume$ пользователь подписывается на очередь и получает поступающие в нее сообщения до тех пор, пока не отпишется.

2. С использованием AMQP команды $basic.get$ пользователь отправляет запрос только на одно сообщение из очереди.

Если сообщение поступает в очередь без подписчиков, то оно будет там храниться до тех пор, пока не появится подписчик. В том случае, если у очереди несколько подписчиков, используется алгоритм «$round - robin$», осуществляющий циклическое распределение сообщений между подписчиками. Этот алгоритм может быть изменен с помощью опции $prefetch\_count$ метода $basic\_qos$, в которой устанав-

ливается максимальное количество необработанных получателем сообщений. Каждое сообщение в AMQP всегда отправляется только одному подписчику. В RabbitMQ также поддерживается механизм подтверждения сообщений. Получив сообщение, подписчик с помощью AMQP команды $basic.ack$ отправляет подтверждение RabbitMQ о том, что сообщение было обработано и брокер может удалять его из очереди. Также это можно сделать, установив параметр $auto\_ack = true$ команды $basic.consume$.

Перейдем к рассмотрению точек обмена и их связей с очередями. Как уже было сказано ранее, поставщик публикует сообщение в точке обмена. В AMQP выбрана именно такая концепция для того, чтобы упростить маршрутизацию сообщений на основе определенных правил. Протокол предусматривает четыре типа точек обмена, каждая из которых реализует свой алгоритм маршрутизации: $fanout$, $direct$, $topic$ и $headers$. Правила, из которых следует, какая очередь должна получить сообщение, называются ключами маршрутизации ($routing\_key$). Очередь связывается с точкой обмена с помощью $routing\_key$. Сообщение, отправленное брокеру, тоже имеет $routing\_key$, который будет сопоставляться с ключом маршрутизации, который используется в связях точки обмена и очереди.

Точка обмена $fanout$, как показано на рисунке 2.2, копирует все поступающие в нее сообщения во все связанные с ней очереди. Точка обмена $direct$ копирует сообщение в очередь, чей ключ маршрутизации равен $routing\_key$ сообщения. Брокер может определить точку обмена $direct$, используя вместо имени точки обмена пустую строку, имя очереди будет использоваться как $routing\_key$. Сообщения, отправляемые в точку обмена $topic$, и связи между точкой обмена и очередью должны иметь ключ маршрутизации, состоящий из нескольких слов, разделенных точкой. Используются два специальных символа:

- * заменяет одно слово
- # заменяет любое количество слов, в том числе и нулевое

Точка обмена $topic$ копирует сообщение в очередь, чей ключ маршрутизации равен $routing\_key$ сообщения с учетом выше приведенных специальных символов. Точка обмена $headers$ осуществляет маршрутизацию, основываясь не на $routing\_key$, а на заголовках сообщений.

Введем еще одно важное понятие: виртуальный хост (virtual host, vhost). В пределах каждого RabbitMQ сервера можно создать виртуальный брокер сообщений под названием vhost. Каждый виртуальный хост представляет собой RabbitMQ сервер с его собственными очередями, точками обмена, связями и правами доступа. Это полезно для разделения нескольких пользователей одного RabbitMQ сервера во избежание коллизии имен очередей и точек обмена.

2.2. Язык реализации RabbitMQ: Erlang

RabbitMQ написан на языке Erlang с использованием открытой телекоммуникационной платформы (OTP), фрэймворка для создания приложений на Erlang.

Джо Армстронг (Joe Armstrong) начал проект по созданию Erlang для компании Ericsson в 1986 с целью улучшения разработки телекоммуникационных приложений посредством создания языка наиболее приближенного к их требованиям [4]. В течении следующего года небольшая команда исследователей из компании Ericsson, включая Роберта Вэрдинга (Robert Virding) и Майка Уильямса (Mike Williams), помогала Армстронгу в дальнейшем развитии и усовершенствовании языка. Впоследствии в 1998 вышла бесплатная версия Erlang.

Erlang - функциональный язык общего назначения с сильной динамической типизацией и среда исполнения. Параллельность и передача сообщений - фундаментальные понятия для этого языка. Приложения, написанные на Erlang, часто состоят из 100 или 1000 легковесных независимых процессов, принадлежащих языку программирования, а не операционной системе, и взаимодействующих друг с

другом посредством обмена сообщениями. Erlang упрощает написание распределенных приложений посредством кластера Erlang узлов, а также обеспечивает их отказоустойчивость.

Фрэймворк OTP - одна из отличительных черт языка программирования Erlang. OTP - множество библиотек и шаблонов проектирования для создания масштабируемых, отказоустойчивых, распределенных приложений. Одной из компонент OTP/Erlang является база данных Mnesia, являющаяся приложением, написанным на Erlang. Основная концепция OTP/Erlang - дерево наблюдения ($supervision\,tree$). Эта модель организации процессов основана на идеи рабочих процессов ($workers$), выполняющих вычисления, и процессов наблюдателей ($supervisors$), следящих за поведением рабочих процессов. В дереве наблюдения многие процессы обладают похожей структурой. Например, процессы-наблюдатели следуют аналогичному шаблону поведения. Большинство рабочих процессов являются серверами в модели клиент-сервер, системами с конечным числом состояний или обработчиками событий, такими как программа регистрации ошибок. Понятие поведение ($behaviour$) формализует общую поведенческую модель этих процессов и схоже с понятием интерфейса в объектно-ориентированном программировании. Основная идея заключается в разделении общей части (модуль поведения) и специфической части (функции обратного вызова). Стандартными поведениями OTP/Erlang являются gen_server (сервер в модели клиент-сервер), $supervisor$ (наблюдатель), gen_event (обработчик событий), gen_fsm (автомат с конечным числом состояний), $application$ (поведение для приложения, реализованного согласно принципам OTP).

2.3. Архитектура RabbitMQ

RabbitMQ реализован как приложение OTP/Erlang. Рассмотрим процесс запуска приложения RabbitMQ, основываясь на [**5**].

Как и для любого другого приложения OTP/Erlang в каталоге с исходным кодом RabbitMQ расположен файл, реализующий поведение *application*, rabbit.erl (rabbit - название приложения в данном случае). Приложение запускается при вызове функции rabbit:start/2 (rabbit - имя модуля, совпадающее с именем файла), находящейся в файле rabbit.erl.

Запуск RabbitMQ состоит из серии шагов, которая называется *boot steps*, отвечающей за запуск основных компонент брокера в специальном порядке. Основная идея этой концепции заключается в том, что каждая отдельная подсистема брокера определяет, от каких подсистем она зависит и работу каких подсистем делает возможной при успешном запуске. Например, нет смысла принимать запросы клиентов на соединение, если уровень, отвечающей за маршрутизацию сообщений в очередь, не активирован. Реализация этого механизма заключается в добавлении модулям erlang специальных атрибутов (деклараций), которые определяют, как запускается шаг загрузки. Рассмотрим пример:

```
-rabbit_boot_step({recovery,
                   [{description, "exchange, queue and binding
                     recovery"},
                    {mfa,         {rabbit, recover, []}},
                    {requires,    empty_db_check},
                    {enables,     routing_ready}]}).
```

В данном примере приведен шаг *recovery* с описанием «exchange, queue and binding recovery». Перед запуском этого шага должен быть запущен шаг *empty_db_check*, а после возможен запуск шага *routing_ready*. Аргумент *mfa* говорит о том, что для активизации этого шага необходимо запустить функцию *recover* из модуля *rabbit* со списком аргументов [].

Шаги загрузки могут разделяться на группы. Группа шагов будет обеспечивать запуск другой группы. Например, для запуска *routing_ready* необходим запуск не только шага *recovery*, но и многих

других. Одним из этих шагов является $empty\_db\_check$, который проверяет, что в Mnesia содержатся данные по умолчанию (пользователь guest, например). Некоторые шаги загрузки не требуют и не обеспечивают запуск других шагов. Они сигнализируют о том, что группа шагов завершила запуск, и следующая группа может быть запущена. Пример:

```
{external_infrastructure,
    [{description,"external infrastructure ready"}]}
```

Опишем механизм деклараций $-rabbit\_boot\_step()$. При запуске брокера создается список всех модулей, определенных в загруженных приложениях. Далее этот список сканируется на наличие атрибута $rabbit\_boot\_step$. Найденные атрибуты добавляются в новый список, который обрабатывается и конвертируется в ориентированный ациклический граф, который используется для определения порядка между шагами загрузки.