Федеральное государственное автономное образовательное учреждение
высшего профессионального образования
«Южный Федеральный Университет»

Факультет математики, механики и компьютерных наук

Направление подготовки 010400
«Прикладная математика и информатика»

А. А. Тактаров

Реактивный фреймворк для организациии мультиагентных распределенных вычислений

Магистерская диссертация

Научный руководитель:
старший преподаватель
В. Н. Брагилевский

Рецензент:
доцент, к. ф.-м. н.
С. А. Гуда

Ростов-на-Дону
2014

Содержание

Введение

Стремительный рост возможностей технологий беспроводной передачи данных, а также широкое распространение мобильных и встраиваемых устройств способоствовали появлению концепции так называемого «Интернета вещей» (англ. *«Internet of Things»*)[1], которая заключается в объединении всех окружающих людей вещей в огромную вычислительную сеть. Участниками (**агентами**) такой сети являются устройства, которые способны собирать информацию о физической среде, обрабатывать ее и реагировать на изменение состояния других агентов и всей системы в целом. Стабильное функционирование такой сети позволит с огромной скоростью внедрять и использовать такие технологии, как «умные» датчики[2], носимые устройства (англ. *wearable devices*), а также системы автоматизированного управления домом. Кроме того, становление «Интернета вещей» влечет за собой появление принципиально новых потоков информации, тщательный анализ которых позволит улучшать существующие системы здравоохранения, безопасности и контролировать состояние окружающей среды.

Однако, создание подобного рода сети невозможно без наличия функционирующей инфраструктуры, которая бы позволила быстро и эффективно интегрировать новые компоненты. Исходя из распределенной природы описываемой сети, сформулируем необходимые для этого требования:

1. Соблюдение принципа системности при разработке[3]. Продукт должен быть представлен в виде целой системы компонентов, каждый из которых обладает определенной функцией. Такие компоненты автоматически становятся автономными участниками сети.

2. Однородность среды. Компоненты сети должны иметь возможность взаимодействовать между собой, используя стандартизированные протоколы и схемы. Задачи идентификации, обеспечения целостности, конфиденциальности передаваемых данных должны по возможно-

сти быть решены этими протоколами.

3. Открытость используемых технологий. Применение как программных, так и аппаратных решений, которые имеют открытую документацию, лояльные условия использования и одновременно поддерживаются разными разработчиками (обычно целым сообществом), позволяет в определенных случаях решить проблему интеграции компонентов и сократить разрыв между разработкой и запуском в производство. Кроме того, открытые платформы предоставляют широкие возможности для начинающих команд разработчиков, что является благоприятным для формирования рынка.

В данной работе описан процесс реализации и интеграции мультиагентной системы на примере задачи распределенной печати фотографий. В рамках разработанной системы устройство, печатающее фотографию, рассматривается как автономный агент, который обладает состоянием и способен принимать и исполнять задания. Принципы, сформулированные выше, были использованы в качестве основополагающих на этапах проектирования и разработки данного продукта.

Постановка задачи

Целью работы является разработка и развертывание системы, позволяющей организовать моментальную печать фотографий пользователей социальной сети Instagram, распределяя задания печати среди подключенных к системе агентов — **печатных станций**, в состав которых входит печатное устройство — принтер.

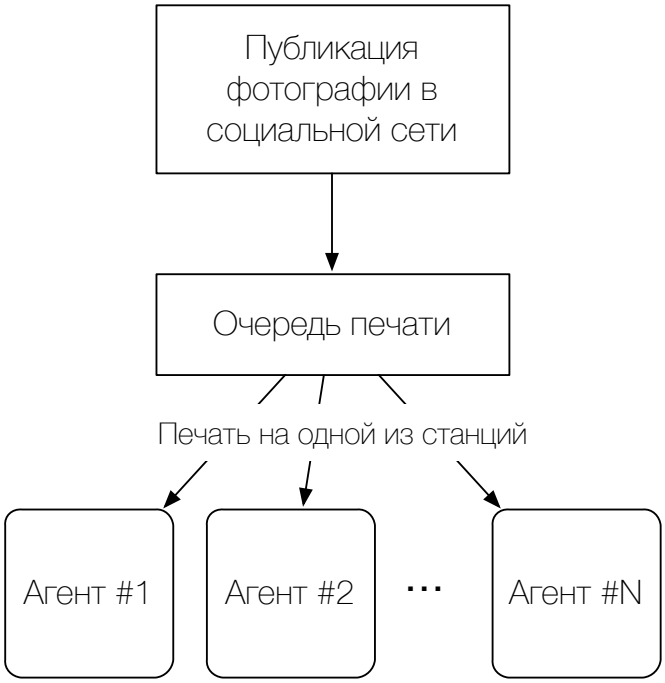

Рис. 1: схема исполнения заданий печати

Фотографии, которые публикуются пользователями социальной сети и удовлетворяют определенным условиям поиска (содержат заранее известную метку — *хештег*), должны автоматически поступать в очередь печати системы. Далее, исходная фотография, прошедшая определенную пост-обработку, печатается на одном из принтеров, входящими в состав печатных станций (рис. 1). Информация о напечатанной фотографии сохраняется в системе для отчетности. Функционирование такой системы позволяет организовать массовую печать фотографий во время проведения мероприятий или для огранизации отложенной печати.

Сформулируем основные требования, предъявляемые к системе:

1. Печатные станции могут быть физически отделены друг от друга, кроме того они могут находиться в разных сегментах сети. Необходим способ организации канала связи между агентами и контроль жизнеспособности этого канала.

2. Необходим интерфейс управления печатными станциями и заданиями печати.

3. Система должна иметь минимальный отклик и максимально быстро реагировать на изменение состояния компонентов. Изменение статуса задания печати (печать может завершиться успешно, а может закончиться неудачей в результате обрыва соединения) должно моментально отражаться в интерфейсе управления заданиями.

Выделим последовательные этапы решения поставленной задачи:

1. Проектирование архитектуры системы: разбиение системы на компоненты, выбор используемых при реализации каждого компонента технологий, построение схемы взаимодействия.

2. Реализация компонентов системы, покрытие отдельных частей функциональными тестами.

3. Решение задач интеграции и развертывании системы, настройка аппаратных средств.

4. Опытное тестирование работы продукта.

1. Архитектура системы

В состав разработанного продукта входят три основных компонента: центральный сервер, агент печатной станции и веб-приложение, предоставляющее интерфейс пользователя (рис. 2).

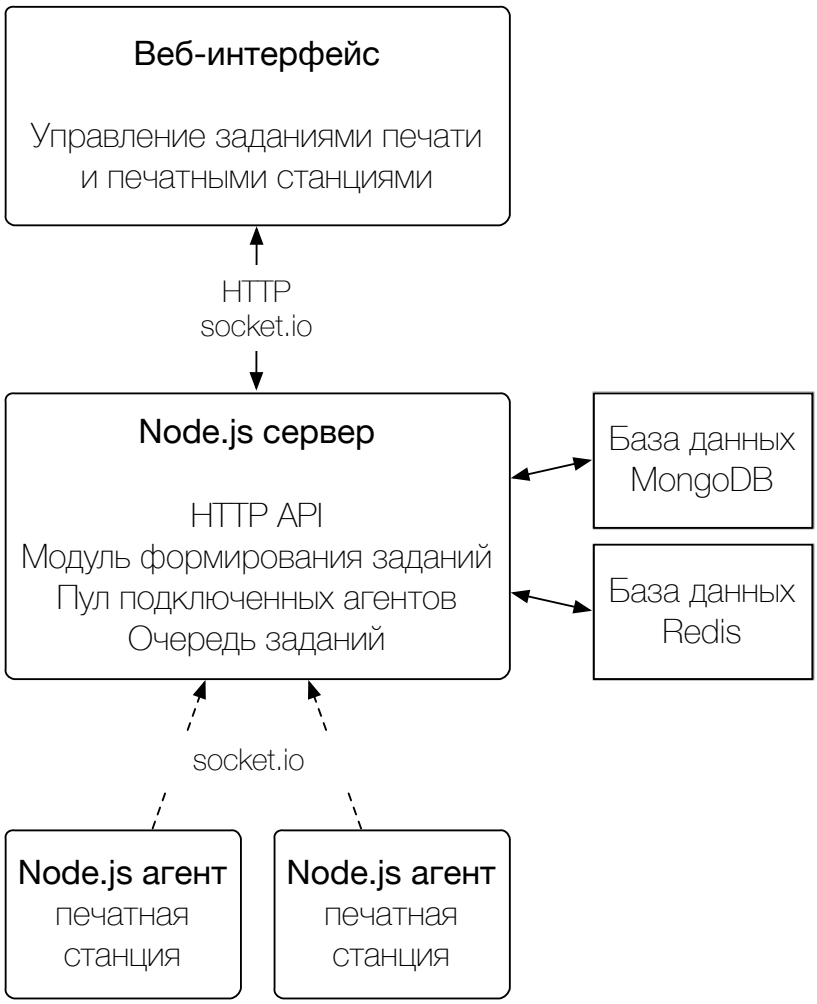

Рис. 2: архитектура системы

Для хранения данных используются базы данных MongoDB и Redis, обращение к которым происходит через центральный сервер. База данных MongoDB содержит информацию о зарегистрированных печатных станциях, администраторах системы, а также хранит историю всех завершенных заданий печати. Средствами MongoDB реализована возможность гибкого поиска и фильтрации данных[4].

База данных Redis, являющаяся быстрым хранилищем типа «ключ-значение», используется для организации очереди заданий печати. Кроме того, благодаря возможности хранения данных в оперативной памяти данная база данных выступает в роли хранилища сессий центрального веб-сервера.

1.1. Центральный сервер

Ядром системы является центральный сервер, в функции которого входит:

1. Управление очередью печати. Модуль формирования заданий используется для поиска в социальной сети новых отмеченных для печати фотографий, которые помещаются в очередь печати. Распечатанные фотографий извлекаются из очереди, а ненапечатанные дополняются сообщением об ошибке для отчетности.

2. Взаимодействие с подключенными по каналу связи агентами. Контроль качества канала связи и авторизация печатных станций.

3. Предоставление прикладного программного интерфейса (API) на основе протокола HTTP.

Данный компонент разработан на языке программирования CoffeeScript[5] и работает на основе асинхронного серверного фреймворка Node.js. Решение по использованию данного инструментария было принято, исходя из следующего:

1. Платформа Node.js предоставляет широкие возможности по использованию низкоуровневых средств (работа с процессами, сокетами, бинарными данными и потоками ввода-вывода), предоставляя для этого удобный интерфейс на языке программирования JavaScript. Кроме того, все операции ввода вывода в Node.js являются асинхронными (т.е. не блокируют исполнение программы), а контроль

завершения происходит с помощью функций обратного вызова и событий. Обработка завершения асинхронных действий реализована в так называемой *очереди обработки событий* (англ. *event loop*)[6], работающей на основе паттерна Проактор[7].

2. Node.js позволяет разработчику использовать сторонние модули благодаря мощному пакетному менеджеру NPM. Простота публикации модулей и открытое сообщество разработчиков по всему миру способствовали развитию огромной инфраструктуры пакетов[8]. Таким образом, проектирование приложений заключается в разбиении на мелкие подзадачи, которые решаются с использованием готовых пакетов, что позволяет оптимизировать процесс разработки.

3. Благодаря тому, что JavaScript является интерпретируемым языком, программы, написанные с использованием Node.js, являются кроссплатформенными. Существует поддержка операционных систем, совместимых с ARM-процессорами, что делает возможным запуск кода даже на встраиваемых устройствах.

4. Язык программирования CoffeeScript является компилируемым в JavaScript языком, расширяющим возможности последнего за счет полноценной поддержки объектно-ориентированной парадигмы и добавления «синтаксического сахара». Синтаксические особенности языка делают возможным реализацию сложных паттернов проектирования, что является важным при разработке больших приложений[9].

1.2. Агент печатной станции

В представленной системе компонентом, который исполняет задания печати, является агент печатной станции, подключенный к центральному серверу. В задачи данного модуля входит:

1. Принятие заданий печати от центрального сервера, представленных

в виде изображения и метаинформации, в которую входят параметры печати и другие вспомогательные данные.

2. Работа с локальной очередью печати подключенного принтера. Постановка на печать полученного изображения.

3. Отправка отчета о статусе завершенного задания.

Компонент разработан на языке программирования CoffeeScript и работает под управлением фреймворка Node.js. В отличие в центрального сервера, который функционирует в режиме демона, подразумевается, что агент может быть запущен по требованию. Более того, одновременно могут быть доступны несколько печатных станций, разнесенных физически и представленных в виде отдельных экземпляров данного компонента.

1.3. Организация канала связи между сервером и агентом

При разработке мультиагентных систем особенно остро встает проблема организации канала связи, который обеспечивает взаимодействие агентов с сервером, распределяющим задания. При проектировании таких систем становится очевидным, что невозможно решить данную проблему, используя только возможности протокола TCP. Во-первых, следует учитывать, что выход в сеть чаще всего всего происходит посредством NAT или межсетевого экрана, что органичивает возможность подключения (в таких условиях необходимо выделять центральный узел, чаще всего расположенный на выделенном сервере). Во-вторых, должен быть способ поддержания длительных сессий между участниками (например, опция keepalive[10] протокола TCP не реализована в старых версиях ядра Linux). Наконец, традиционная схема передачи данных не является удобной при реализации реактивных систем, взаимодействие компонентов в которой обычно организовано в виде двунаправленной передачи сообщений.

В для огранизации канала связи между сервером и печатной станцией в описываемой системе выбор был сделан в пользу свободно распространяемой библиотеки socket.io[11]. Данная библиотека предоставляет следующие возможности:

1. Организация дуплексного канала связи, который абстрагирован от среды запуска и транспорта. Возможно использование socket.io как в серверных и клиентских приложениях (написанных на Node.js), так и в веб-браузере. Для передачи данных могут использоваться технологии WebSockets или JSON Polling (периодический опрос сервера о наличии новых сообщений); переключение между этими транспортами является прозрачным.

2. Автоматический контроль жизнеспособности канала. Это реализовано при помощи отправки серии служебных сообщений (heartbeats) от клиента к серверу и наоборот. В случае обрыва соединения происходит попытка повторного подключения, причем существует стратегия экспоненциального увеличения интервала между неудачными попытками. Такое поведение доступно по-умолчанию, и чаще всего процесс восстановления соединения скрыт от программиста.

Узлы, использующие socket.io, обмениваются друг с другом сообщениями, которые сериализуются для отправки выбранным транспортом. Сообщение можно представить в виде кортежа $(T, a_1, a_2, ..., a_N)$, где T — текстовый идентификатор типа сообщения, $a_1, a_2, ..., a_N$ — сериализованные параметры сообщения.

Библиотека socket.io предоставляет также возможность реализации удаленного вызова процедур (англ. RPC — Remote Procedure Call) благодаря механизму подтверждений (англ. *acknowledgement*). Использование удаленного вызова процедур на примере функции возведения в квадрат представлено в листинге 1. Вызов удаленной функции происходит как отправка сообщения, последним параметром которого является функция обратного вызова, которая будет исполнена, как только удаленная сторона отправит ответ. Реализация удаленной функции также

содержит параметр-функцию, которую необходимо вызвать для возврата значения. Заметим, что этот параметр не является исходной функцией, которая была передана при отправке сообщения. Ее вызов приводит к отправке подтверждающего сообщения, содержащего результат (рис. 3).

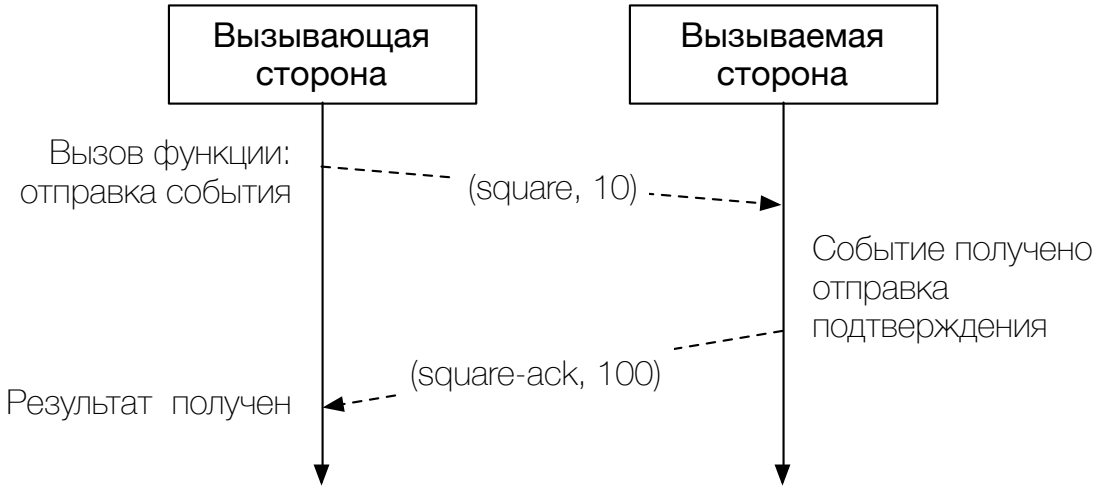

Рис. 3: схема организации удаленного вызова процедур

Листинг 1: Удаленный вызов процедур в socket.io

```
# RPC функция возведения числа в квадрат
socket.on "square", (n, callback) ->
  # Вызов переданной функции выглядит как вызов
  # обычной функции
  callback n * n


# Вызов RPC функции на клиенте выглядит как отправка сообщения.
# Первый аргумент: тип сообщения
# Второй аргумент: параметр сообщения
# Третий аргумент это функция реакции на acknowledgement
socket.emit "square", 10, (result) ->
  # Полученный результат: 100
  console.log result
```

При реализации агента печатной станции были использованы возможности удаленного вызова процедур для организации печати заданий, полученных от центрального сервера.

1.4. Веб-интерфейс

Для удобного управления очередью печати и мониторинга состояния подключенных станций разработан интерфейс, представляющий собой одностраничное веб-приложение, написанное на языке CoffeeScript. Взаимодействие интерфейса происходит через HTTP API центрального сервера с использованием парадигмы REST.

Веб-интерфейс построен с применением паттерна MVC («Model-View-Controller» — «Модель-Вид-Контроллер»)[12], реализованного в клиентском веб-фреймворке Chaplin[13]. Особенность реализации заключается в том, что подгрузка данных с сервера, их обработка, генерация представления этих данных и управление переходами между состояниями веб-приложения полностью происходит на стороне веб-браузера, что позволяет снизить нагрузку с центрального сервера. Кроме того, применение такого подхода означает, что собранное веб-приложение представимо в виде набора статичных файлов, что делает возможным использование кеширования для ускорения загрузки страницы.

2. Особенности реализации

В данной главе затронуты детали реализации отдельных компонентов системы, касающиеся организации печати в рамках печатной станции,очереди печати и описана модель хранимых в системе данных.

2.1. Организация печати

Возможность совершения печати агентом реализована благодаря низкоуровневой работе с системным спулером печати и классу-обертки, позволяющему получать уведомления в виде событий о статусе печати.

2.1.1. Работа с системной очередью печати

Взаимодействие с принтером происходит посредством постановки заданий в системную очередь печати (такую очередь называется *спулером* печати). В UNIX-подобных системах это осуществляется через CUPS («Common Unix Printing System» — система печати в UNIX). Система CUPS была изначально спроектирована так, чтобы обеспечить поддержку печати в условиях модели «клиент-сервер», то есть допускает использование одного принтера сразу несколькими компьютерами в локальной сети.

Ключевым понятием в CUPS является **получатель** (англ. *destination*) печати. Получателем называется доступный по сети принтер вместе с очередью печати. Фактически, любой локальный принтер в CUPS также воспринимается как сетевой. Печать файла представляет собой подключение к определенному получателю и постановку файла в очередь.

Взаимодействие с CUPS происходит через API в виде библиотеки на языке Си[14]. Для обеспечения возможности работы с CUPS из кода агента (реализованного на Node.js) был разработан модуль-обертка на C++. Node.js предоставляет разработчикам средства, позволяющие писать низкоуровневые модули, работа с которыми доступна в контексте

исполнения JavaSript[15].

Задача модуля — предоставление интерфейса для вызова функций CUPS, посредством функций, доступных из JavaScript. При этом следует учитывать, что переданные аргументы должны быть преобразованы из JavaScript-примитивов в типы данных, которые поддерживает CUPS (например, экземпляр JavaScript класса `String` должен быть преобразован к Си-строке). Пободные трансформации в данном модуле реализованы благодаря функция и классам библиотеки V8, которая является движком, используемым платформой Node.js.

Приведем список функций CUPS, поддержка которых была реализована:

`cupsGetDests` Возвращает список всех доступных получателей в локальной системе.

`cupsGetDefault` Возвращает идентификатор получателя по-умолчанию.

`cupsGetJobs` Возвращает список всех заданий печати в очереди заданного получателя.

`cupsPrintFile` Печать файла. Создает новое задание печати в очереди заданного получателя.

`cupsCancelJob` Отменяет заданное задание печати.

Разработанный модуль опубликован в виде пакета в официальном репозитории NPM под названием **cupsidity**. Исходный код модуля доступен в виде открытого репозитория и снабжен документацией[16].

2.1.2. Высокоуровневый класс-обертка

К сожалению, CUPS не предоставляет интерфейса, с помощью которого можно получать асинхронные события по изменению статуса заданий печати, что противоречит принципу реактивности, указанному среди требований к разрабатываемой системе. Единственным способом,

позволяющим узнать, было ли завершено конкретное задание, является периодический вызов функции `cupsGetJobs`.

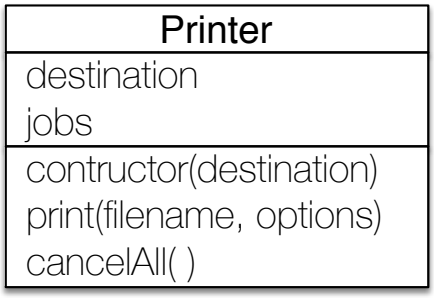

Рис. 4: класс-обертка Printer

Для предоставления асинхронного интерфейса для печати файлов был разработан класс-обертка `Printer` (рис. 4). Конструктор класса принимает параметр `destination`, являющийся названием принтера-получателя.

Листинг 2: Печать файла с помощью класса `Printer`

```
1  # Получаем доступ к принтеру
2  printer = new Printer config.get "printer:name"
3
4  # Печать файла
5  printer.print(filename, config.get "printer:options")
6  .then ->
7    log.info "Job successfully printed"
8    done null
9  .fail (err) ->
10   log.warn "Printing error: #{err}"
11   done "Printing error"
```

Печать заданного файла осуществляется посредством вызова функции `print`, принимающей путь к печатаемому файлу и необходимые параметры печати. Данная функция возвращает объект `Deferred`, который является одним из асинхронных примитивов, входящих в состав библиотеки Q[17]. `Deferred` — это представление завершенности отложенной асинхронной операции, которое может находиться в следующих состояниях: ожидается (англ. *pending*), завершено (англ. *resolved*), завершено

с ошибкой (англ. *rejected*). Пример вызова функции `print` и обработка состояний объекта `Deferred` представлены в листинге 2.

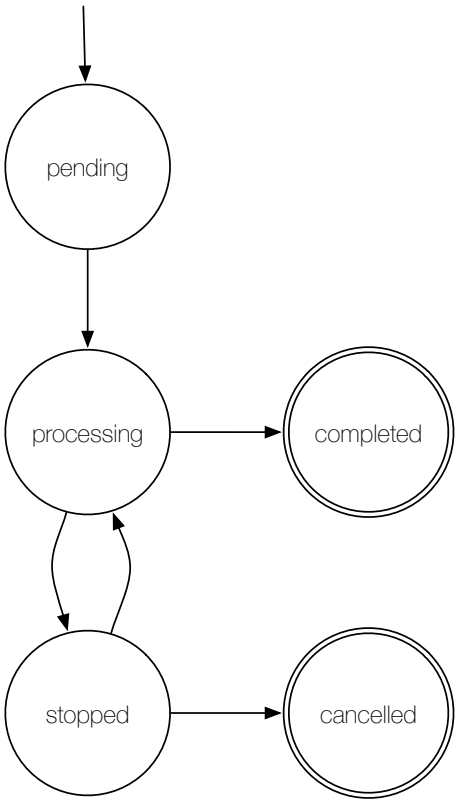

Рис. 5: диаграмма состояний задания печати

Вызов функции `print` приводит к созданию задания печати, которое может находиться в следующих состояниях:

`pending` Задание находится в очереди и ждет обработки. Это начальное состояние задания.

`processing` Задание исполняется (происходит печать).

`completed` Задание завершено (файл напечатан). Это терминальное состояние, которое свидетельствует об успешном завершении задания.

`stopped` Задание приостановлено. Причиной остановки могут быть проблемы, связанные с оборудованием и расходными материалами: закончилась бумаги или краска, связь с принтером прервалась и т.д.

`cancelled` Задание отменено. Терминальное состояние, свидетельствующее о том, что во время печати произошли ошибки.

Возможные переходы между описанными состояниями приведены на рис. 5. Класс `Printer` содержит приватную функцию `stateCheckRoutine`, которая вызывается с определенным интервалом времени и детектирует переход между состояними заданий печати, которые были назначены на заданный принтер. Переход задания в одно из терминальных состояний означает завершение процесса печати и приводит к завершению соответствующего объекта `Deferred`.

Заключение

Список литературы

1. B. Wasik. In the programmable world, all our objects will act as one. http://www.wired.com/2013/05/internet-of-things-2/, 2013.

2. Nest world's first learning thermostat. https://nest.com/.

3. М. Слюсаренко, И. Слюсаренко. Системологический подход к декомпозиции в объектно-ориентированном анализе и проектировании программного обеспечения. 2010.

4. K. Seguin. *The Little MongoDB Book*. 2012.

5. A. MacCaw. *The Little Book on CoffeeScript*. O'Reilly Media, 2012.

6. A. Hall. Understanding the node.js event loop. http://strongloop.com/strongblog/node-js-event-loop/, 2013.

7. C. Kohlhoff. The proactor design pattern: Concurrency without threads. http://www.boost.org/doc/libs/1_47_0/doc/html/boost_asio/overview/core/async.html, 2011.

8. C. Robbins. Npm: innovation through modularity. http://blog.nodejitsu.com/npm-innovation-through-modularity/, 2013.

9. D. Brady et al. Coffeescript cookbook. http://coffeescriptcookbook.com/, 2014.

10. TCP keepalive overview. http://tldp.org/HOWTO/TCP-Keepalive-HOWTO/overview.html.

11. Socket.io real-time bidirectional event-based communication. http://socket.io/.

12. C. Robbins. Scaling isomorphic javascript code. http://blog.nodejitsu.com/scaling-isomorphic-javascript-code/.

13. Chaplin HTML5 application architecture. http://chaplinjs.org/.

14. Introduction to CUPS programming. http://www.cups.org/documentation.php/api-overview.html.

15. Node.js documentation addons. http://nodejs.org/api/addons.html.

16. Cupsidity — CUPS (common unix printing system) native bindings for Node.js. https://github.com/molefrog/cupsidity.

17. Q — a tool for making and composing asynchronous promises in JavaScript. https://github.com/kriskowal/q.