Содержание

[Введение 6](#_Toc296024553)

[1 Системы мониторинга 7](#_Toc296024554)

[1.1 Введение 7](#_Toc296024555)

[1.2 Понятие систем мониторинга 7](#_Toc296024556)

[1.3 Подсистемы мониторинга 8](#_Toc296024557)

[1.3.1 Сбор данных 9](#_Toc296024558)

[1.3.2 Хранение данных 10](#_Toc296024559)

[1.3.3 Анализ данных 10](#_Toc296024560)

[1.3.4 Отчетность 11](#_Toc296024561)

[1.3.5 Оповещения 11](#_Toc296024562)

[1.3.6 Диспетчеризация 12](#_Toc296024563)

[1.4 Классификация систем мониторинга 12](#_Toc296024564)

[1.5 Требования к системам мониторинга 14](#_Toc296024565)

[1.6 Проблемы эксплуатации систем мониторинга 15](#_Toc296024566)

[1.7 Выводы 16](#_Toc296024567)

[2 Модель распределенной системы мониторинга 18](#_Toc296024568)

[2.1 Общие положения 18](#_Toc296024569)

[2.2 Базовая теоретическая модель 18](#_Toc296024570)

[2.3 Модуль мониторинга 19](#_Toc296024571)

[2.4 Система исполнения 20](#_Toc296024572)

[2.5 Код каркаса 21](#_Toc296024573)

[2.6 Прикладной интерфейс программирования 22](#_Toc296024574)

[2.7 Состояние распределенной системы мониторинга 22](#_Toc296024575)

[3 Реализация системы 25](#_Toc296024576)

[3.1 Служба мониторинга 25](#_Toc296024577)

[3.1.1 Структура службы мониторинга 25](#_Toc296024578)

[3.1.2 Выбор средств реализации 26](#_Toc296024579)

[3.1.3 Ядро системы 31](#_Toc296024580)

[3.1.4 Транспортная подсистема 55](#_Toc296024581)

[3.1.5 Подсистема исполнения 60](#_Toc296024582)

[3.2 Менеджер модулей 63](#_Toc296024583)

[3.2.1 Общее описание 63](#_Toc296024584)

[3.2.2 Выбор средств реализации 64](#_Toc296024585)

[3.2.3 Уникальный идентификатор модуля 64](#_Toc296024586)

[3.3 Прикладной интерфейс программирования 65](#_Toc296024587)

[3.3.1 Общее описание 65](#_Toc296024588)

[3.3.2 Выбор средств реализации 65](#_Toc296024589)

[3.4 Панель управления 66](#_Toc296024590)

[3.4.1 Общее описание 66](#_Toc296024591)

[3.4.2 Архитектура панели управления 67](#_Toc296024592)

[3.4.3 Модель 68](#_Toc296024593)

[3.4.4 Контроллер 73](#_Toc296024594)

[3.4.5 Представление 74](#_Toc296024595)

[3.5 Описание организации совместной работы 78](#_Toc296024596)

[3.5.1 Skype (skype.com) 79](#_Toc296024597)

[3.5.2 GoogleMail (gmail.com) 79](#_Toc296024598)

[3.5.3 Хостинг проектов Google (goolecode.com) 79](#_Toc296024599)

[Заключение 80](#_Toc296024600)

[Список использованных источников 82](#_Toc296024601)

# Введение

Быстрорастущий уровень современной компьютеризации общества сопровождается появлением нового класса программных инструментов – систем мониторинга. Основная задача подобных решений - систематический анализ и интерпретация протекающих в гетерогенной среде процессов. Полученные в результате мониторинга данные могут быть использованы как для улучшения процесса принятия решений, так и для выявления узких мест исследуемой системы.

Настоящая работа представляет собой исследование современных решений в области мониторинга, оценку их эффективности и применимости согласно выдвинутой модели требований, а также выводы о необходимости появления нового класса инструментов мониторинга, ввиду неготовности существующих решений удовлетворять ранее выдвинутым требованиям.

Кроме того, в работе детально представлена предлагаемая авторами модель распределенной системы мониторинга гетерогенной среды, лишенная недостатков классических клиент-серверных систем. В основе предлагаемой архитектуры лежат механизмы разработки и исполнения дополнительных модулей, а также возможности и свойства отказоустойчивых распределенных систем.

Наконец, в работе подробно описана спроектированная архитектура каркаса мультиплатформенной распределенной системы мониторинга и рассмотренна её реализация с точки зрения современных технологий программирования распределенных систем – высокоуровневых языков и библиотек среднего слоя.

Итогом работы является каркас распределенной системы мониторига, который представляет собой программный комплекс, состоящий из службы мониторинга, менеджера модулей, интерфейса программирования модулей и панели управления.

# Системы мониторинга

## 1.1 Введение

Понятие систем мониторига впервые было рассмотрено в начале 80-х годов XXI века, в период становления и популяризации сетевых операционных систем. В это время, системы представляли собой унифицированные инструменты сетевого администратора, необходимые и пригодные для обнаружения отказа обарудования или построения примитивной статистики использования ресурсов.

В настоящее время, системы мониторинга являются одним из самых насыщенных классов программных систем, предоставляющих пользователю широчайших набор возможностей, ориентированных на различные задачи и среды эксплуатации.

Системы мониторинга, как класс программных систем за почти полувековую историю эволюционировали из примитивных инструментов администратора до универсальных коробочных решений промышленного уровня.

Несмотря на достаточную насыщенность рынка, новые инструменты мониторинга, совмещающие в себе современные подходы и методологии программирования, продолжают появляться, определяя новые вектора развития класса систем в целом.

## 1.2 Понятие систем мониторинга

*Мониторинг* — систематический сбор и анализ информации о состоянии некоторого вычислительного узла или информационного процесса, организованный с определенной целью. Целями мониторинга могут быть: формализация и улучшение процесса принятия решений; детальный анализ и исследование системы на предмет поиска узких, высоконагруженных мест; сокращение времени простоя системы в случае выхода из строя ее основных компонентов и т.п.

Базовая теоретическая модель описывается с помощью понятий вычислительного узла, сервера и агента мониторинга (рисунок 1.1).



Рисунок 1.1 – Базовая модель

Под *вычислительным узлом* гетерогенной среды здесь и далее будем понимать программно-аппаратное устройство, в память которого может быть загружен и затем исполнен код какой-либо сущности мониторинга.

*Агент*, запущенный на определенном узле, представляется активной сущностью, непрерывно наблюдающей за его состоянием и передающей серверу сообщения об изменении этого состояния.

*Сервер мониторинга* — пассивная сущность, предоставляющая агентам ресурсы для приема сообщений, их последующей обработки и хранения, а также реализующий механизмы отчетности и реагирования на нештатные ситуации.

## 1.3 Подсистемы мониторинга

Функционирование любой системы мониторинга можно представить в виде набора взаимосвязанных повторяющихся действий, среди которых наиболее концептуальными являются сбор, хранение и анализ данных, а также отчетность и оповещение. Тогда обобщенная архитектура системы мониторинга будет выглядеть как композиция отдельных подсистем, ответственных за каждое из вышеперечисленных действий (рисунок 1.2).



Рисунок 1.2 – Подсистемы мониторинга

### 1.3.1 Сбор данных

Известно несколько подходов к сбору данных системой мониторинга с удаленных вычислительных узлов гетерогенной среды, среди которых можно выделить два наиболее распространенных: двусторонний и односторонний. Двусторонний метод сбора данных представляет собой исследование и анализ реакции удаленной системы на определенный набор внешних воздействий. При одностороннем методе, исследуемая система сама предоставляет исследователю необходимые данные путем пересылки оповещений об изменении своего внутреннего состояния.

С технической точки зрения двусторонний метод является более предпочтительным при построении систем мониторинга, т.к. требует меньших затрат на реализацию и максимально использует возможности исследуемой операционной среды. В свою очередь, односторонний метод предоставляет больше возможностей по наращиванию функционала конечной системы.

На практике чаще всего используется комбинированный метод сбора данных, который объединяет в себе особенности двух подходов – одностороннего и двустороннего. Комбинированный метод сбора данных характеризуется использованием там, где это возможно, двустороннего подхода и одностороннего во всех остальных случаях.

Подсистему сбора данных разделяют между собой агенты и сервер мониторинга. Все остальные рассмотренные подсистемы относятся только к серверу мониторинга.

### 1.3.2 Хранение данных

Хранение данных, полученных в результате процессов мониторинга, может быть организовано как с использованием средств баз данных, так и на базе простых плоских файлов. Существуют также «задаче-ориентированные» варианты хранилищ, например распределенные высоконагруженные кеши, облачные нереляционные базы данных и т.п. Таким образом, варианты хранения данных можно охарактеризовать как централизованные и децентрализованные.

Очевидно, что децентрализованные или распределенные варианты хранения обладают большей отказоустойчивостью взамен сложности реализации и сопровождения.

Можно сформулировать минимальный, с точки зрения авторов, список требований к подсистеме хранения данных (централизованной или децентрализованной) системы мониторинга:

1. целостность, доступность и безопасность данных;
2. производительность и эффективность примитивных операций ввода вывода;
3. легкость внедрения и сопровождения.

### 1.3.3 Анализ данных

Анализ, оценка и принятие решений могут происходить непосредственно в реальном времени, как реакция на многократное возникновение нештатной ситуации.

Подсистема анализа данных реализует механизмы и примитивы проверки полученных в результате мониторинга данных для выявления текущего состояния исследуемой системы. Подсистему анализа можно представить в виде композиции следующих логических компонент:

1. цепочка правил или предикатов проверки данных;
2. механизмы применения правил;
3. механизмы генерации исключительных ситуаций.

В результате применения цепочки правил к каждым полученным данных, подсистема анализа выявляет текущее состояние удаленной системы, которое может характеризовать работу исследуемой системы как штатную или нештатную.

При определении нештатной работы удаленной системы, подсистемой анализа генерируются исключительные ситуаций, обрабатываемые подсистемой оповещений или диспетчеризации.

Анализ данных мониторинга позволяет выявить тенденции нежелательных ситуаций.

### 1.3.4 Отчетность

Отчеты позволяют визуализировать и кластеризовать данные мониторинга в удобочитаемой форме, пригодной для анализа и просмотра пользователем. Подсистема генерации отчетов позволяет представить данные в виде информации, которую можно распечатать или сохранить в одном из поддерживаемых электронных форматах.

По способу визуализации можно выделить текстовые и графические отчеты. К текстовым отчетам относятся непосредственно текст, списки, таблицы. Этот вид отчетов характеризуется большим объемом информации при компактном отображении, что делает его удобным для хранения. К графическим относятся графики, графы, схемы. Основным достоинством этого вида отчетов является наглядность, быстрота восприятия. Эти отчеты эффективны, когда необходимо динамично и понятно отобразить данные.

### 1.3.5 Оповещения

Подсистема оповещений реализует набор инструментов и механизмов оповещения заинтересованных клиентов о возникновении исключительных ситуаций в исследуемой системе. Под исключительными понимаются ситуации, приводящие к сбою в работоспособности или отказу аппаратной или программной части системы.

В некотором смысле, можно представлять систему оповещения как набор обработчиков исключительных ситуаций, генерируемых подсистемой анализа данных. При этом при непосредственной обработке происходит оповещение определенных шаблоном клиентов, о том, что исследуемая система с текущего момента функционирует в нештатном режиме.

Подсистема оповещения позволяет оператору системы оперативно реагировать на сбои в удаленной работе системы и быстро принимать решения об их устранении.

### 1.3.6 Диспетчеризация

В качестве расширяющей механизм оповещений можно выделить обособленную подсистему диспетчеризации. *Диспетчеризация* – это процесс оперативного контроля, управления, координации какого-либо процесса с использованием оперативной передачи информации между исследуемым объектом и управляющим исследователем.

Система мониторинга реализует подсистему диспетчеризации, если в качестве реакции на возникновение исключительной ситуации она может некоторым образом воздействовать на состояние исследуемой системы. Под воздействием в данном случае понимается удаленное восстановление системы после сбоя и возобновление ее нормальной работы или, если это невозможно, корректное завершение работы исследуемой системы.

## 1.4 Классификация систем мониторинга

В рамках данной работы предлагается следующая классификация систем мониторинга в сфере информационных технологий: по характеру сетевого взаимодействия и по функционалу (рисунок 1.3).



Рисунок 1.3 – Классификация систем мониторинга

По характеру сетевого взаимодействия можно выделить клиент-серверные и распределенные[1] системы мониторинга.

Клиент-серверные или централизованные системы построены по принципу классических сетевых систем с выделенным сервером. В таких системах присутствуют активные и пассивные сущности – агенты и серверы мониторинга соответственно. В качестве примера клиент-серверных систем мониторинга можно привести продукты Zabbix[3], Nagios[4].

В распределенных или децентрализованных системах мониторинга отсутствует понятие сервера, в классическом его понимании. Каждый узел распределенной системы может одновременно являться как сервером, так и агентом мониторинга. Текущее состояние узла и его поведение характеризуются глобальным состоянием распределенной системы [1], которое имеет свойство изменяться под воздействием как внутренних, так и внешних факторов. Единственной в полном смысле распределенной системой мониторинга является проект Ganglia [5].

С точки зрения функционала системы можно выделить системы с расширяемым и нерасширяемым функционалом.

Будем считать, что система мониторинга является системой с расширяемым функционалом, если в её коробочной поставке есть штатные средства и инструменты, позволяющие динамически наращивать функционал целевой системы. Как правило, подобные инструменты динамического расширения функционала реализованы в виде механизмов разработки и исполнения дополнительных модулей или плагинов системы мониторинга на каком-либо языке программирования. Например, системы Nagios, Zabbix, Mon[6]являются системами с расширяемым функционалом.

Системы мониторинга с нерасширяемым функционалом характеризуются фиксированным, достаточно общим базовым набором функций и возможностей, расширение которого возможно только при участии производителя. Инструментами расширения функционала в этом случае являются пакеты обновления системы, предоставляемые производителем. Вышеупомянутый проект Ganglia,а также системы на базе Zenoss [8] являются примерами решений с нерасширяемым функционалом.

## 1.5 Требования к системам мониторинга

Применимость и практическая ценность систем мониторинга определяется их способностью адаптироваться к условиям динамически изменяющихся требований, среди которых декларируются требования к функционалу системы, отказоустойчивости и масштабируемости.

Требования к функционалу системы мониторинга опираются на область ее внедрения и последующей эксплуатации. Необходимость изменения функционала возникает, как правило, вследствие динамики внешних требований. Можно условно разделить внешние требования на две группы — технические и правовые. Под техническими требованиями понимаются требования к функционалу системы, основанные на целях мониторинга, сетевой инфраструктуре[9], оборудовании или программном обеспечении. Под правовыми требованиями понимаются требования законов государства, лицензионных соглашений, корпоративных уставов.

Под *отказоустойчивостью* понимают способность технической системы сохранять работоспособность и правильно функционировать после отказа некоторых ее компонентов, возможно основных. Применительно к системам мониторинга можно дать следующее определение понятию отказоустойчивости. Система мониторинга называется отказоустойчивой, если она продолжает функционировать согласно целям мониторинга после отказа любой компоненты или сущности системы. Известно, что повышение отказоустойчивости достигается за счет избыточности или резервирования наиболее значимых ресурсов системы. Наиболее значимой сущностью в архитектуре систем мониторинга является сервер мониторинга. Поэтому, репликация или резервирование серверов мониторинга – это наиболее предпочтительный способ повышения отказоустойчивости системы мониторинга.

*Масштабируемость* системы мониторинга может измеряться по двум различным показателям. Во-первых, система может быть масштабируемой по отношению к ее размеру, что означает легкость подключения к ней дополнительных клиентов и ресурсов. Во-вторых, система может масштабироваться географически, то есть клиенты и ресурсы могут быть значительно отдалены друг от друга в гео-пространстве. Применительно к системам мониторинга, масштабируемость во всех смыслах определяется способностью взаимодействия географически удаленных узлов, а также легкостью подключения новых вычислительных узлов к системе мониторинга.

Рассмотренные выше требования к системам мониторинга позиционируются авторами как основные и дальнейшие рассуждения относительно применимости того или иного класса систем или инструментов будут проводиться с точки зрения этих требований.

## 1.6 Проблемы эксплуатации систем мониторинга

Основная причина проблем эксплуатации существующих решений в сфере мониторинга заключаются в неспособности последних удовлетворять требованиям к современным системам мониторинга (раздел 1.4). При этом наиболее важным является не столько разовое удовлетворение предъявляемым требованиям, сколько наличие в системе потенциала для возможной ее адаптации к динамике этих требований.

Проблема расширения функционала систем мониторинга была разрешена в системах с поддержкой динамической загрузки модулей, например в системах Zabbix и Nagios, однако остается актуальной для нерасширяемых систем, в частном случае для решений на базе Ganglia. С другой стороны, данная проблема может трактоваться не только как наличие или отсутствие соответствующих механизмов наращивания функционала, но и как уровень их применимости и возможностей. Тогда можно говорить о недостаточной гибкости существующих решений в плане средств расширения функционала, как например, в системах Zabbix и Nagios, где в качестве подобного инструмента используется модель с запуском исполняемых файлов сопровождаемым перехватом стандартных потоков ввода/вывода операционной среды.

Проблема отказоустойчивости характерна только для класса клиент-серверных систем. В распределенных системах она решается на уровне теоретической модели за счет использования методов избыточности, репликации и сериализуемости [1]. Система мониторинга кластеров и гридов Ganglia является хорошим примером действительно распределенной системы.

Проблема масштабируемости системы по отношению к ее размеру также актуальна только для клиент-серверных систем. В распределенных системах мониторинга стоимость подключениях новых вычислительных узлов для системы в целом равна нулю благодаря использованию механизмов балансировки нагрузки, а также сокрытию времени ожидания связи. Так при построении системы мониторинга на базе решения Ganglia можно не заботиться о теоретическом пределе количества подключаемых к системе устройств. В противоположность этому, в технической документации клиент-серверные решения Zabbix и Nagios определено максимальное количество обслуживаемых устройств.

## 1.7 Выводы

Согласно приведенным выше рассуждениям можно сделать вывод о том, что основополагающая проблема эксплуатации современных систем мониторинга заключается в отсутствии на рынке целого класса комбинированных систем, одновременно объединяющих в себе преимущества как распределенных, так и расширяемых систем мониторинга. Кроме того, современные тенденции развития облачных и кластерных решений в области суперкомпьютерных технологий лишь подтверждают необходимость появления подобных инструментов мониторинга.

Таким образом, рассмотренные выше проблемы эксплуатации систем мониторинга, позволяют сделать вывод о неготовности существующих решений комплексно выполнять выдвинутые к ним требования.

Авторами предлагается проект распределенной системы мониторинга и диспетчеризации процессов гетерогенной среды, которая позволяет обеспечить выполнение перечисленных требований.

# 2 Модель распределенной системы мониторинга

## 2.1 Общие положения

Авторами предлагается модель архитектуры распределенной системы мониторинга, которая позволяет обеспечить выполнение рассмотренных в первом разделе требований. Главная идея предлагаемого подхода заключается в использовании механизма разработки и исполнения дополнительных модулей в процессе решения задач мониторинга, а также свойств распределенных систем в процессе эксплуатации.

Предлагаемая методология построения распределенных систем мониторинга является обобщением и объединением существующих классов систем в сфере мониторинга – распределенных систем с расширяемым функционалом.

## 2.2 Базовая теоретическая модель

Базовая теоретическая модель описывается с помощью следующих понятий (рисунок 2.3):

1. вычислительный узел;
2. служба мониторинга;
3. хранилище данных;
4. задача мониторинга.

Под *вычислительным узлом* далее будем понимать программно-аппаратное устройство, в память которого может быть загружен и затем исполнен код какой-либо сущности мониторинга.

*Служба мониторинга*, запущенная на определенном узле, представляется активной сущностью, непрерывно наблюдающей за его состоянием и сохраняющей сообщения об изменении этого состояния в хранилище данных.

*Хранилище* данных представляется пассивной сущностью, предоставляющей службам ресурсы для приема сообщений, их последующей обработки и хранения.



Рисунок 2.3 – Базовая модель

Фактически служба мониторинга и хранилище данных являются основными компонентами любой современной системы мониторинга за исключением небольшого количества программных продуктов, не реализующих клиент-серверную или иную известную сетевую архитектуру, но являющихся инструментами мониторинга (например, утилита ping).

*Задача мониторинга* представляет собой шаблонную проблему получения и анализа некоторой информации о состоянии удаленного узла.

Можно ввести отношение между целью (раздел 1.1 «Понятие систем мониторинга») и задачей мониторинга. Какая-либо цель мониторинга включает в себя одну или более задач мониторинга. При этом какая-либо задача мониторинга может одновременно реализовывать несколько целей мониторинга.

## 2.3 Модуль мониторинга

Под *модулем мониторинга* далее будем понимать абстракцию (рисунок 2.4), характеризующуюся:

1. возможностью исполнения в операционной среде;
2. входными данными, передаваемыми исполняющей системой;
3. выходными данными, возвращаемыми исполняющей системе;
4. интерфейсом, задающим правила исполнения модуля;
5. реализацией, представляющей собой программный код, воплощающий функционал модуля.



Рисунок 2.4 – Абстракция модуля

Понятие модуля мониторинга является следствием отображения задачи мониторинга из предметной области в программную среду.

## 2.4 Система исполнения

Система исполнения модулей мониторинга (рисунок 2.5) реализует генерацию кода каркаса и исполнение модулей мониторинга с использованием ресурсов операционной среды, а также является промежуточным слоем между модулем мониторинга и службой, в рамках которой он запускается. Данный слой позволяет разрабатывать модули без учета специфики физического расположения служб мониторинга, игнорируя такие особенности, как адресация и топология сети.

Система исполнения является определяющей компонентой в модели службы мониторинга и от ее реализации зависит эффективность и применимость системы в целом. Кроме того, система исполнения является платформенно-зависимой частью системы и должна разрабатываться под определенный набор операционных и аппаратных платформ.



Рисунок 2.5 - Система исполнения

## 2.5 Код каркаса

Код каркаса (рисунок 2.6) генерируется системой исполнения на основании текущего глобального состояния распределенной системы и содержит следующие конструкции:

1. инициализации окружения;
2. создания экземпляра модуля мониторинга;
3. исполнения экземпляра;
4. передача параметров и ожиданием возвращаемого результата.



Рисунок 2.6 - Код каркаса

## 2.6 Прикладной интерфейс программирования

Модули мониторинга разрабатываются в терминах предметной области с использованием прикладного интерфейса программирования (API) [10] — высокоуровневого объектно-ориентированного набора инструментов. Прикладной интерфейс программирования (рисунок 2.7) является промежуточным слоем между модулем мониторинга и операционной средой, в которой он запущен. API призван сосредоточить программиста на решаемой задаче мониторинга, скрыв от него подробности реализации сложных моментов, таких как распределенная коммуникация с сервером, маршализация/демаршализация параметров и возвращаемого результата модуля, системные вызовы операционной системы.



Рисунок 2.7 – Взаимодействие модулей и ОС через API

## 2.7 Состояние распределенной системы мониторинга

Известно понятие глобального состояния [1], в соответствии с которым распределенная система функционирует в данное время (рисунок 2.8). В классической трактовке состояние определяется графом связности узлов, расположением запущенных экземпляров модулей и нагрузкой на узлы. В предлагаемой модели сущность распределенного модуля представляет служба мониторинга. Это придает ей некоторые особенности элемента распределенной системы, например:

1. масштабируемость — возможность запуска дополнительного экземпляра;
2. сериализуемость — возможность сохранения текущего состояния службы;
3. переносимость — возможность переноса службы в распределенной среде с сохранением ее внутреннего состояния.

Служба мониторинга характеризуется своим внутренним непротиворечивым состоянием – активным или пассивным. Активное состояние наделяет службу дополнительными обязанностями по отношению к соседним узлам: планирование запусков модулей мониторинга; мониторинг и диспетчеризация процессов исполнения модулей мониторинга; предоставление промежуточного хранилища для пересылаемых сообщений.



Рисунок 2.8 - Состояние распределенной системы

В предлагаемой модели роль нагрузку на узел играет индекс производительности — целое положительное число, определяющее количество свободных вычислительных ресурсов узла по некоторой абсолютной или относительной шкале. Индекс производительности узла совместно с установленным пороговым значением являются рычагами воздействия на глобальное состояние распределенной системы мониторинга.

Службы, запущенные на узлах с индексом производительности ниже порогового значения, подвергаются масштабированию (запуску дополнительных экземпляров, сопровождаемому балансировкой нагрузки), в результате чего распределенная система переходит в более производительное состояние (рисунок 2.9).



Рисунок 2.9 – Изменение состояние системы

# 3 Реализация системы

## 3.1 Служба мониторинга

### 3.1.1 Структура службы мониторинга

Служба мониторинга (рисунок 3.1) представляет собой программный комплекс, обеспечивающий использование ресурсов вычислительной среды, адресацию, поддержание поведения распределенной системы мониторинга (модулей мониторинга, распределенной коммуникации, программной системы в целом).



Рисунок 3.1 - Служба мониторинга

Служба мониторинга распределенной системы состоит из двух основных подсистем:

* подсистемы исполнения, позволяющей планировать и запускать модули мониторинга, а также получать результаты исполнения модулей и сохранять их в хранилище данных;
* транспортной подсистемы, реализующей сетевую модель распределенной системы и включающей в себя механизмы именования объектов, удаленной коммуникации, адресации и балансировки нагрузки.

Служба мониторинга представляет собой распределенное приложение [11], а, следовательно, должна устойчиво функционировать в гетерогенной вычислительной среде.

### 3.1.2 Выбор средств реализации

#### 3.1.2.1 Модель программирования

В процессе выбора средств реализации службы мониторинга были рассмотрены популярные на сегодняшний день технологии построения распределенных систем. Кроме того, были учтены особенности реализации существующих решений в области мониторинга. Авторами были рассмотрены следующие технологии разработки распределенных систем:

* модель программирования на распределенной памяти или механизмы передачи сообщений (PVM [12], MPI [13]);
* распределенные системы объектов (CORBA [14]);
* нетрадиционные языки программирования для распределенных систем (Erlang [15]);
* библиотеки промежуточного слоя (Ice ZeroC [16]).

В итоге, была выбрана объектно-ориентированная платформа среднего слоя Ice (The Internet Communication Engine) от компании ZeroC в силу доминирующего превосходства над аналогами. Платформа Ice снабжена инструментами, API и библиотеками для разработки объектно-ориентированных клиент–серверных приложений. Ice-приложения могут быть написаны на различных языках программирования (Java [17], C++, Python [18], C#, Ruby [19]), запущены под различными операционными системами (Windows NT, Linux, MacOS OS) и аппаратными платформами, а также могут взаимодействовать, используя разнообразные сетевые технологии. В общем случае Ice позиционируется как инструмент RPC (Remote Procedure Call) [20], который достаточно прозрачно применять на практике. Большое количество компаний по всему миру, таких как Skype, HP, Silicon Graphics используют технологию Ice в своих проектах.

Платформа Ice обладает следующими особенностями преимуществами:

* объектно-ориентированная семантика;
* поддержка синхронных и асинхронных вызовов;
* аппаратная независимость;
* языковая независимость;
* операционная независимость;
* безопасность;
* доступность исходного кода;

Кроме того, используемый в Ice объектно-ориентированный подход позволяет переиспользовать уже известные шаблоны проектирования параллельных многопоточных систем без каких-либо исключений.

#### 3.1.2.2 Терминология модели программирования

Для более детального понимания реализации рассмотрим основные понятия и сущности, представленные платформой Ice. Основные термины модели программирования трактуются с помощью типовых понятий из теории по клиент-серверным системам.

Можно выделить два вида программных агентов в платформе Ice:

1. клиент — активная сущность, запрашивающая некоторые ресурсы у сервера;
2. сервер — пассивная сущность, предоставляющая некоторые ресурсы клиенту.

На практике редко встречаются «чистый» сервер или «чистый» клиент. Чаще всего это смешанный клиент-сервер, который одновременно и запрашивает и предоставляет ресурсы.

Любая сущность, запущенная в распределенной системе Ice, определяется так называемым Ice-объектом. Ice-объект – это абстракция, характеризующаяся следующим:

* локальными или удаленным адресным пространством;
* возможностью реакции на удаленный запрос;
* поддержкой репликации;
* несколькими интерфейсами;
* публичными методами интерфейсов имеющих как входные, так и выходные параметры;
* уникальным идентификатором, не совпадающим с любым другим идентификатором объекта в распределенной гетерогенной среде.

Для непосредственной распределенной коммуникации объектов платформа Ice реализует специальные механизмы транспортного уровня, инкапсулированные в объектах типа прокси. Практически, Ice-прокси соответствует одному Ice-объекту и предоставляет для него механизмы и примитивы для организации удаленных вызовов процедур. Прокси – это артефакт, который локален для клиентского адресного пространства. Ice-прокси инкапсулирует следующую информацию, о действиях, совершаемых платформой при вызове удаленной процедуры клиентом:

1. определяет местоположение Ice-объекта;
2. активирует сервер, если он не запущен;
3. активирует Ice-объект на сервере;
4. передает входные параметры Ice-объекту;
5. ждет, когда операция закончится;
6. передает возвращаемые значения или генерирует исключение.

Кроме того, Ice-прокси содержит:

1. адресную информацию, которая позволяет клиентской стороне соединиться с нужным сервером;
2. идентификатор объекта, который является целью запроса;
3. дополнительный идентификатор, который определяет интерфейс объекта, к которому прокси обращается.

Стоит также рассмотреть виды запросов на диспетчеризацию и виды диспетчеризации вызовов.

Платформа среднего слоя Ice поддерживает следующие виды запросов на диспетчеризацию:

* синхронный вызов, при котором клиент, совершивший вызов, повисает на время выполнения процедуры, пока она не закончится;
* асинхронный вызов, сопровождаемый передачей объекта обратного вызова;
* односторонний вызов метода с организацией одностороннего потока сетевого трафика;

Аналогично запросам на диспетчеризацию можно выделить следующие виды диспетчеризации методов:

* синхронная диспетчеризация, при которой серверный поток повисает в ожидании завершения процедуры;
* асинхронная диспетчеризация, эквивалентная асинхронному вызову методов;

Для описания удаленных интерфейсов объектов Ice использует специализированный язык описания спецификаций – Slice (Specification Language for Ice).

Каждый Ice-объект имеет интерфейс с конечным набором операции. Интерфейсы, операции и типы данных, которыми обмениваются сервер и клиент, должны быть описаны на языке Slice. Slice позволяет описывать поведение сервера и клиента, не опираясь на какой-либо язык программирования, благодаря этому написанный код на Slice компилируется в код любого из поддерживаемых платформой языков программирования.

#### 3.1.2.3 Язык программирования

Платформа среднего слоя Ice поддерживает почти все современные языки программирования, среди которых можно отметить наиболее популярные:

* нативные (C++, Objective C [21]);
* управляемые (Java, С#, VB.NET);
* динамические/интерпретируемые (Python, PHP [22]).

Для выбора инструмента программирования рассмотрим основные выдвигаемые к нему требования:

* достаточно высокая производительность языка или платформы;
* кроссплатформенность;
* поддержка ООП-семантики;
* наличие современных и эффективных средств разработки;
* доступность языка или платформы;
* богатая библиотека стандартных модулей и классов.

Кроме того, можно выделить еще один важный критерий выбора – скорость разработки и простота внесения изменений в программный код.

Динамически интерпретируемые языки, такие как Python и PHP, не удовлетворяют требованиям к производительности. Безусловно, современные технологи построения интерпретаторов позволяют им добиваться сравнимой с нативным кодом производительности, однако ядро службы мониторинга является бутылочным горлышком системы и может существенно влиять на поведение и скорость работы всей системы в целом. Поэтому нами рассматривались два варианта – языки Java и С++. Языки на платформе .NET [23] не рассматривались из-за отсутствия кросс-платформенной реализации.

С одной стороны оба языка предоставляют программисту сравнительно одинаковый набор возможностей (ООП, стандартная библиотека классов и модулей). С другой – программы, написанные на Javaи на С++, показывают абсолютно разную, практически несравнимую производительность.

В конечном счете, нами была выбрана платформа Java. Решающим фактором, определившим наш выбор, стала скорость разработки, которая, как известно, на Java значительно выше, чем на C++. Более того, современные виртуальные машины платформы Java (Oracle Hotspot, OpenJDK) со встроенными компиляторами динамического кода (JIT) показывают отличную производительность, порой превосходящую нативное исполнение не в разы, а на порядки. Такой прирост производительности объясняется в первую очередь механизмами динамического профилирования и сборки мусора, которые идеологически недоступны в нативных компиляторах.

Помимо вышеперечисленного, для Java существует большое количество удобных и эффективных сред разработки (Eclipse [24], Netbeans [25], IntelliJIDEA [26]), отладки, тестирования, а также свободных переносимых библиотек для решения широкого круга прикладных задач.

Безусловно, нельзя отвергать тот факт, что участники проекта при выборе языковой платформы основывались на собственный опыт разработки программных систем, где чаще всего применялась платформа Java.

### 3.1.3 Ядро системы

#### 3.1.3.1 Общее описание

Ядро службы мониторинга (рисунок 3.2) реализует базовую, динамически расширяемую программную платформу, в рамках которой запускаются и функционируют основные подсистемы службы. Кроме того, ядро обеспечивает работу загружаемых компонентов службы, а также содержит базовые механизмы и примитивы для их взаимодействия и синхронизации.

Как уже отмечалось, ядро представляет собой динамически расширяемую программную модель, функционал которой может изменяться процессе эксплуатации, посредством загрузки или выгрузки дополнительных компонентов ядра – так называемых *драйверов ядра*.

Функционирование ядра системы реализовано в терминах модели «Цикл событий», смысл которой заключается в бесконечной обработке событий, приходящих системе от внешних клиентов. В качестве внешних клиентов системы выступают драйверы ядра, каждый из которых реализует определенную часть общего поведения и функционала конечной системы.



Рисунок 3.2 – Ядро службы мониторинга

Взаимодействие драйверов не осуществляется напрямую. Вместо этого используется генерация, обработка и передача специальных событий ядру. Событие ядра инкапсулирует тип случившейся внутрисистемной ситуации и содержит необходимые параметры и структуры для ее корректной обработки.

Для обработки событий используются обработчики ядра. Ядро имеет несколько обработчиков, каждый из которых соответствует определенному состоянию ядра.

Помимо модели «Цикл событий», ядро реализует парадигму «Конечный автомат» (Finit State Machine) [27]. Проще говоря, ядро характеризуется своим внутренним состоянием и может переходить из состояния в состояние при обработке некоторого внутрисистемного события.

Основная идея предлагаемого подхода для разработки ядра распределенной службы мониторинга заключается в так называемых публичных драйверах. Помимо основных драйверов системы, именуемых в дальнейшем приватными, существуют дополнительные драйверы – публичные. Приватные драйвера могут использоваться только тем ядром, в адресное пространство которого они загружены, в то время как публичные могут использоваться любыми другими удаленными ядрами, запущенными в гетерогенной среде.

Для придания драйверу ядра публичности достаточно реализовать для него, так называемый *адаптер* драйвера ядра. Адаптер драйвера ядра предоставляет внешним клиентам RPC интерфейc для синхронных и асинхронных вызовов публичных методов драйвера. Такая модель применяется для взаимодействия служб, запущенных в различных адресных пространствах.

Кроме того, для удаленного взаимодействия используются *сессии*. Сессия представляет собой набор доступных адаптеров ядра с публичными интерфейсами. Существуют сессии режима ядра и сессии режима пользователя. Сессии режима ядра устанавливаются между удаленными ядрами. Сессии режима пользователя устанавливаются между ядром и панелью управления.

#### 3.1.3.2 Архитектура ядра

Ядро представляет собой автономный поток исполнения, реализующий модель «Цикла событий». Ядро содержит базовые механизмы и примитивы, необходимые для работы системы, такие как обработка событий, изменение состояния системы, управление драйверами/адаптерами и управление сетевой подсистемой.

Ядро также содержит основные таблицы и КЭШи системы:

1. пул событий ядра;
2. таблица подключенных дочерних узлов;
3. таблица подключенных родительских узлов;
4. кэш исследованных узлов;
5. таблица драйверов ядра;
6. таблица адаптеров ядра;
7. таблица фильтров ядра;
8. таблица наблюдателей ядра.

В каждый момент времени ядро находится в определенном состоянии, менять которое способен только его текущий обработчик, который однозначно определяется состоянием. Кроме того, изменение любых внутренних структур ядра разрешено только потоку ядра. В противном случае – при попытке несанкционированного изменения состояния любым внешним потоком, генерируется внутреннее исключение ядра.

Ядро управляет двумя основными сетевыми адаптерами системы – первичным и вторичным. Сетевые адаптеры ядра используются для реализации транспортного уровня системы – механизма удаленного вызова процедур (RPC).

Ядро службы мониторинга характеризуется внутренним непротиворечивым контекстом ядра, который содержит основную информацию о его текущем состоянии:

1. идентификатор ядра – 32-х байтовая последовательность, однозначно идентифицирующая ядро в гетерогенной среде (раздел 3.1.3.4 «Уникальный идентификатор узла»);
2. прокси первичного и вторичного адаптера, необходимые для установления соединения между ядрами, запущенными в различных адресных пространствах;
3. индекс производительности узла – целое положительное число, определяющее текущую производительность узла по некоторой шкале;
4. состояние ядра – текущее состояние ядра;
5. список дочерних подключенных узлов, используемый системой для адекватной оценки производительности всей распределенной системы в целом;
6. список родительских узлов, используемый исполнительной подсистемой для управления расписанием и запуском модулей мониторинга;
7. базовая информация о вычислительном узле (тип операционной системы, имя и IP-адрес компьютера в сети).

#### 3.1.3.3 Исключения ядра

Для соблюдения парадигмы защищенного программирования [28] авторами была разработана модель исключений ядра.

В предлагаемой архитектуре ядра распределенной службы мониторинга существует два вида внутрисистемных исключений:

1. нативные исключения ядра;
2. внешние исключения, генерируемые драйверами.

Нативные исключения ядра генерируются самим ядром при следующих ситуациях:

1. ошибка инициализации и деинициализации ядра;
2. некорректная обработка события ядра;
3. ошибка загрузки и выгрузки драйвера ядра;
4. недетерминированный переход между состояниями ядра;
5. обработка внешних системных исключений виртуальной машины или операционной системы.

В отличие от нативных исключений, внешние исключения обрабатываются ядром как обычные события (раздел «3.1.3.10 События ядра»). Источниками внешних исключений могут служить следующие ситуации:

1. некорректная работа локального или удаленного драйвера ядра;
2. ошибка в транспортной подсистеме;
3. ошибка в исполнительной подсистеме службы.

Внедрение в программную модель механизма обработки и генерации исключений и применение принципов защищенного программирования позволило обеспечить систему необходимым уровнем отказоустойчивости и предсказуемости.

#### 3.1.3.4 Пул ядра

Поток ядра реализует механизм бесконечной обработки событий, который генерируют ядру драйвера. Событие при генерации попадает в пул ядра - потокобезопасную очередь с фильтрацией и без планирования. Это значит, что события обрабатываются ядром по принципу FIFO. Благодаря использованию потокобезопасного пула, поток ядра имеет особенность «засыпать» при условии, что пул пустой и в текущий момент нет запроса на обработку. При этом первое поступившее событие в очередь его «разбудит» и основной цикл будет продолжен.

Это позволяет экономить ресурсы системы в сетях с небольшой нагрузкой.

Перед непосредственной обработкой события происходит этап фильтрации событий ядра (раздел 3.1.3.4 «Фильтры пула ядра»). С практической точки зрения фильтрация представляет собой последовательное применение цепочки фильтров на каждое событие. В результате – событие может быть отфильтровано либо пропущено.

Псевдокод основного цикла потока ядра для обработки пула событий изображен на рисунке 3.3.



Рисунок 3.3 – Псевдокод потока ядра

#### 3.1.3.5 Фильтры пула ядра

При проектировании модели поведения ядра стало ясно, что прямая реализация классической модели обработки событий имеет свои определенные недостатки и требует формальных доработок и изменений для данной архитектуры. В первую очередь, это касается пула ядра.

В процессе эксплуатации экспериментального прототипа, опытным путем, было выяснено, что не все поступившие в пул ядра события должны быть обработанными.

При определенной последовательности событий в пуле модель переходов между состояниями ядра становилась недетерминированной. Особенно это проявлялось в многопоточной среде исполнения. Для придания ядру детерминированного поведения, нами был разработан механизм фильтрации событий из пула ядра.

Фильтры пула ядра применяются для исключения определенного класса событий из пула до момента их обработки. Фильтры ядра образуют последовательную цепочку, по которой проходит каждое событие из пула. Событие считается отфильтрованным, если хотя бы один фильтр из цепочки сработал.

В текущей реализации доступно два фильтра ядра:

1. фильтр переходов (ToogleFilter);
2. фильтр UDP сообщений (UDPFilter).

Фильтр переходов ориентирован на фильтрацию сообщений об изменении состояния ядра. Семантика работы фильтра подразумевает удаление из пула всех неразрывных последовательностей событий об изменении состояния ядра кроме последнего.

Фильтр UDP сообщений исключает системные сообщения от удаленных служб, если ядро находится в активном состоянии (раздел 3.1.3.5 «Состояния и обработчики ядра»).

Архитектурно, фильтры пула ядра представляют собой реализацию шаблона «Посетитель/Visitor» [28] (диаграмма классов на рисунке 3.4).

Реализация механизмов фильтрации пула событий позволило внести определенную ясность и предсказуемость в поведение ядра. В частности, наибольший эффект от внедрения механизмов фильтрации ожидался в исключении недетерминированности переходов между состояниями ядра.



Рисунок 3.4 – Диаграмма классов пакета фильтров ядра

#### 3.1.3.6 Состояния и обработчики ядра

Для реализации поведения ядра службы в терминах модели конечного автомата авторами были спроектированы и реализованы конечные множества состояний и событий ядра. Существует пять различных состояний ядра системы:

1. активное (active state);
2. пассивное (passive state);
3. сетевое (online state);
4. автономное (offline state);
5. неопределенное (suspense state).

Как уже отмечалось, состояние ядра однозначно определяет обработчик, которым будут обрабатываться события, приходящие ядру из внешней среды. Таким образом, можно утверждать, что в каждый момент времени состояние ядра определяет его поведение.

С точки зрения реализации, и состояние и обработчик ядра представляются шаблоном проектирования «Состояние/State» [28] (рисунок 3.5). При этом оба интерфейса содержат лишь по одному публичному методу. В случае с состоянием – это метод получения текущего обработчика, в случае с обработчиком – это метод обработки следующего события ядра.



Рисунок 3.4 – Диаграмма классов обработчиков и состояний

Переход из состояния в состояние осуществляется только при обработке определенного класса событий. На рисунке 3.6 рассмотрены основные циклы смены состояний у ядра. Более детально про переходы между состояниями рассмотрены в разделах 3.1.3.1 «События ядра» и 3.1.3.12 «Поведение ядра».

При изменении своего состояния ядро оповещает всех заинтересованных в этом событии драйверов. Такие драйвера реализуют специальный интерфейс, о котором подробно будет сказано в разделе 3.1.3.6 «Наблюдатели ядра».

Реализация модели поведения ядра в терминах конечных автоматов оказалась хорошей практикой. Отладка, тестирование и разработка распределенной системы становится более очевидной и понятной для программиста при применении данного подхода.



Рисунок 3.5 – Состояния ядра

#### 3.1.3.7 Наблюдатели ядра

В процессе реализации драйверов ядра (раздел 3.1.3.7 «Драйверы ядра»), стало понятным, что любое изменение состояния ядра должно явно или не явно отражаться на дальнейшем поведении драйверов. Иными словами, каждый в отдельности взятый драйвер ядра должен функционировать только тогда, когда ядро находится в конкретном, пригодном для нормальной работы этого драйвера состоянии.

Для реализации подобного поведения нами был спроектирован и разработан интерфейс наблюдателя ядра, который должны реализовывать все драйвера, нуждающиеся в зависимом от состояния поведении. Интерфейс наблюдателя ядра содержит один публичный метод, оповещающий драйверы об изменении состояния ядра и передающий драйверам признак этого нового состояния.

Можно выделить следующие особенности, которыми характеризуется драйвер, реализующий интерфейс наблюдателя:

1. отложенный (lazy) запуск/останов или активация/деактивация;
2. зависящее от состояния ядра специфичное поведение.

С точки зрения реализации наблюдатель ядра представляет собой шаблон проектирования «Наблюдатель/Observer» [28]. Тогда драйверы, реализующие интерфейс наблюдателя, в терминах шаблон являются подписчиками.

Более подробно о конкретных драйверах, реализующих интерфейс наблюдателя ядра можно прочитать в разделе 3.1.3.7 «Драйверы ядра».

#### 3.1.3.8 Драйверы ядра

Понятие драйвера ядра введено в программную архитектуру службы мониторинга для разделения функциональных особенностей системы на составляющие. Это наделяет систему модульными особенностями - гибкостью и расширяемостью. Для расширения функционала конечной системы достаточно реализовать один или несколько дополнительных драйверов.

Драйверы реализуют функционал основных подсистем службы – транспортной и исполнительной. Можно условно разделить драйверы ядра на две группы – драйверы транспортной подсистемы и драйверы исполнительной подсистемы. Кроме того, существует третья группа драйверов, реализующих функционал самой платформы.

Драйверы транспортной подсистемы реализуют следующие функциональные особенности ядра:

1. обнаружение в сети вычислительных узлов с запущенными службами мониторинга;
2. управление удаленными сессиями;
3. мониторинг доступности сетевой подсистемы узла.

Драйверы исполнительной подсистемы реализуют следующие особенности поведения ядра:

1. планирование процесса запуска моделей мониторинга;
2. запуск модулей мониторинга;
3. получение и обработка результатов запуска модулей мониторинга;
4. управление модулями мониторинга, развернутыми на данном узле.

В свою очередь, функциональные драйверы реализуют следующие особенности:

1. получение системной информации о вычислительном узле;
2. конфигурирование службы мониторинга;
3. удаленное управление службой мониторинга.

В общем случае, драйвер ядра представляет собой сущность, характеризующуюся:

1. специфичным поведением, реализующим некоторую часть общего функционала системы;
2. наличием самостоятельного потока исполнения;
3. возможностью запуска/останова;
4. возможностью активации/деактивации;
5. возможностью загрузки/выгрузки из памяти платформы;
6. возможностью реагировать на изменение ядром своего состояния.

Рассмотрим общий интерфейс драйвера службы мониторинга на рисунке 3.7.



Рисунок 3.6 – Интерфейс драйвера ядра

Обобщение интерфейса драйвера ядра было сделано в первую очередь для однообразной работы ядра с любым типов драйверов. Кроме того, так называемая точка устойчивости, реализованная в виде интерфейса, дает возможность прозрачно наращивать функционал системы.

Любой драйвер системы должен реализовывать как минимум два публичных метода: получение имени драйвера и получение ядра службы. Эти данные необходимы для корректной регистрации драйвера в системе и его последующей инициализации. Стоит отметить, что благодаря обобщенному интерфейсу все драйвера инициализируются и загружаются автоматически.

Кроме того, существуют дополнительные интерфейсы драйверов, которые могут быть реализованы опционально:

1. загружаемый драйвер (рисунок 3.7);
2. активируемый драйвер (рисунок 3.8);
3. запускаемый драйвер (рисунок 3.9).

Каждый из перечисленных интерфейсов наделяет драйвер дополнительным поведением, которое кратко рассмотрено ниже.

Драйвер является загружаемым, если для его нормального функционирования требуется выполнить дополнительный набор действий в процессе загрузки или выгрузки модуля. Фактически, любой модуль является автоматически загружаемым, однако не каждый модуль требует при этом переопределения поведения загрузки и выгрузки.



Рисунок 3.7 – Интерфейс загружаемого драйвера

Следующим этапом после загрузки драйвера является его активация. Активация драйверов ядра происходит после полной загрузки ядра, инициализации всех структур и активации подсистем. Если требуется переопределить поведение драйвера в момент активации, он должен реализовывать соответствующий интерфейс.



Рисунок 3.8 – Интерфейс активируемого драйвера

Наконец запускаемые драйверы характеризуются собственным потоком исполнения и могут быть запущены или остановлены в любой момент эксплуатации системы. Запуск и остановка драйверов тесно связанны с понятием наблюдателя ядра. Как правило, драйверы останавливаются и запускаются только при переходе ядра в определенное ожидаемое состояние.



Рисунок 3.9 - Интерфейс запускаемого драйвера

В текущей реализации системы присутствует полный, по мнению авторов, набор драйверов, обеспечивающих систему полным функционалом.

Диаграмма классов реализованных не текущий момент в системе драйверов ядра приведена на рисунке 3.10.



Рисунок 3.10 – Диаграмма классов драйверов

Краткое описание реализованных драйверов приведено в таблице 3.1.

Таблица 3.1 – Драйверы ядра

|  |  |  |
| --- | --- | --- |
| **Драйвер** | **Интерфейсы** | **Основные функции** |
| «Scheduler» | «наблюдатель»; «запускаемый»;  «активируемый»; | планирование запусков модулей;  управление персональным расписанием;  синхронизация удаленных расписаний; |
| «Resulter» | «наблюдатель»;  «запускаемый»;  «активируемый»; | получение результатов выполнения модулей мониторинга;  обработка результатов;  пересылка результатов в постоянное хранилище; |
| «Networker» | «активируемый»; | мониторинг сетевой подсистемы узла;  оповещение ядра о сбоях в работе сети; |
| «Aliver» | «запускаемый»;  «наблюдатель»; | мониторинг доступности удаленных сессий ядра; |

Продолжение таблицы 3.1

|  |  |  |
| --- | --- | --- |
| **Драйвер** | **Интерфейсы** | **Основные функции** |
| «Configurer» | «загружаемый» | конфигурирование свойств ядра; |
| «Discoverer» | «запускаемый»;  «наблюдатель»; | обнаружение удаленных служб в сети;  генерация событий обновления кеша ядра; |
| «Hoster» | «загружаемый»; | получение информации о вычислительном узле; |
| «Controller» | «загружаемый»; | удаленно управлением поведением ядра; |
| «Sessionier» | «загружаемый»; | создание сессий режима ядра и режима пользователя;  управление удаленными сессиями; |
| «Moduler» | «загружаемый»; | запуск модулей мониторинга;  развертывание модулей мониторинга;  управление доступными модулями; |

#### 3.1.3.9 Адаптеры ядра

Для взаимодействия между драйверами, загруженными в различные адресные пространства, применяются специальные Ice-объекты - адаптеры ядра. При этом для использования удаленного драйвера достаточно в локальном адресном пространстве создать для его адаптера Ice-прокси. Такая связь называется «объект-прокси» и присутствует в архитектуре службы в качестве списков дочерних и родительских узлов (раздел 3.1.3.2 «Архитектура ядра»).

Рассмотрим на примере реализацию интерфейса адаптера драйвера Discoverer на языке описания спецификаций Slice (рисунок 3.11). Более подробно про семантику работы драйвера Discoverer можно прочитать в разделе 3.1.3.2 «Исследователь».



Рисунок 3.11 – Пример реализации интерфейса адаптера ядра

Можно условно разделить адаптеры ядра на две группы:

1. стационарные адаптеры;
2. динамические адаптеры;

Стационарные адаптеры создаются при запуске системы и доступны по уникальному статическому идентификатору на всем протяжении работы приложения. Это сделано для того, чтобы независимо от текущего состояния ядра и распределенной системы в целом иметь доступ к критическим частям службы мониторинга.

В текущей реализации доступно два стационарных адаптера – адаптеры для драйверов Discoverer и Sessionier (раздел 3.1.3.7 «Драйверы ядра»).

Динамические адаптеры генерируются автоматически на основании текущих подключённых к ядру сессий. Динамические адаптеры могут быть доступны только в рамках сессий – сессии режима ядра или сессии режима пользователя (раздел 3.1.3.10 «Сессии ядра»).

#### 3.1.3.10 События ядра

События ядра наряду с его состояниями являются основополагающими компонентами всей программной платформы службы мониторинга. Внедрение в программную архитектуру модели генерации и обработки событий позволило исключить прямые взаимодействия между драйверами системы, тем самым придерживаются шаблоны «Низкая связность» и «Высокое зацепление» [28] в процессе разработки.

Событие ядра представляет собой сущность, характеризующуюся следующим:

1. типом случившейся ситуации;
2. списком параметров;
3. наличием соответствующего обработчика.

С точки зрения реализации событие ядра представляется интерфейсом, представленным на рисунке 3.12.



Рисунок 3.12 – Интерфейс события ядра

Любое событие может быть сгенерировано только двумя сущностями – самим ядром, либо одним из его драйверов. При этом источник события заносит всю необходимую информацию в структуру и передает ее ядру на обработку, вызывая метод ядра из публичного интерфейса.

Диаграмма классов событий ядра, реализованных в текущей версии системы, представлена на рисунке 3.13.



Рисунок 3.13 – Диаграмма классов событий ядра

События инкапсулируют тип возникшей ситуации в своем классе. Иными словами, идентификация типа события происходит на уровне поддержки полиморфизма языковой платформой. Такая реализация с точки зрения авторов является наиболее эффективной и расширяемой.

В таблице 3.2 кратко рассмотрены все события, поддерживаемые системой в текущей реализации.

Таблица 3.2 – События ядра

|  |  |  |
| --- | --- | --- |
| **Событие** | **Источник** | **Описание** |
| InvokationEvent | Scheduler | Абстрактное событие для инкапсулирования запуска модуля мониторинга, для которого сработало расписание (раздел 3.1.4.2 «Планировщик подсистемы исполнения»); |
| KernelReconfiguredEvent | Configurer | Событие, генерируемое при переконфигурации ядра; |
| ExceptionEvent | <any driver> | Событие, инкапсулирующее внешнее исключение драйвера, передаваемое на обработку ядру; |
| ChildSessionSendedEvent | Kernel | Событие, генерируемое ядром при посылке собственной сессии внешнему клиенту в качестве дочерней; |
| ChildSessionRecievedEvent | Kernel | Событие, генерируемое ядром при получении удаленной дочерней сессии узла; |
| ParentSessionSendedEvent | Kernel | Событие, генерируемое ядром при посылке собственной сессии внешнему клиенту в качестве родительской; |
| ParentSessionRecivedEvent | Kernel | Событие, генерируемое ядром при получении удаленной родительской сессии узла; |
| KernelStateChangedEvent | Kernel | Событие, генерируемое ядром при изменении своего состояния; |

Продолжение таблицы 3.2

|  |  |  |
| --- | --- | --- |
| **Событие** | **Источник** | **Описание** |
| ForceStartEvent | Moduler | Событие, генерируемое при принудительном запуске модуля; |
| ChildNodeDiedEvent | Aliver | Событие, генерируемое при обнаружении «мертвой» дочерней сессии; |
| ParentNodeDiedEvent | Aliver | Событие, генерируемое при обнаружении «мертвой» родительской сессии; |
| DiscoverRecievedEvent | Discoverer | Событие, генерируемое при получении пакета обнаружения узлов; |
| ResultRecievedEvent | Scheduler | Событие, генерируемое при получении результат исполнения модуля мониторинга; |
| NetworkEnabledEvent | Networker | Событие, генерируемое при отсутствии сетевой подсистемы узла; |
| NetworkDisabledEvent | Networker | Событие, генерируемое при доступности сетевой подсистемы узла; |
| SchedulerUpdatedEvent | Scheduler | Событие, генерируемое планировщиком, при изменении внутреннего расписания службы; |
| ScheduleTimeComeEvent | Scheduler | Событие, генерируемое планировщиком, при наступлении времени исполнения модуля; |
| SnoopydStartedEvent | Kernel | Событие, генерируемое ядром при запуске службы; |
| SnoopydTerminatedEvent | Kernel | Событие, генерируемое ядром при завершении работы службы; |

С точки зрения авторов, рассмотренный выше набор событий является абсолютно полными и позволяет описать любой сценарий использования.

#### 3.1.3.11 Сессии ядра

Для удаленного взаимодействия между службами мониторинга нами были спроектированы и реализованы так называемые сессии ядра. Сессия ядра представляет собой Ice-прокси удаленного ядра и характеризуется:

1. типом или классом сессии;
2. методом вызовов удалённых процедур;
3. списком публичных драйверов, методы которых можно вызывать удаленно;
4. параметрами сетевого соединения (протокол, адрес, шифрование).

Можно выделить два вида сессий, доступных в текущей реализации: сессии режима ядра и сессии режима пользователя (рисунок 3.14).



Рисунок 3.14 –Диаграмма классов сессий ядра

Сессии режима ядра устанавливаются между двумя удаленными драйверами, в то время как сессии режима пользователя устанавливаются между ядром и панелью управления.

Исходный код описаний интерфейсов на языке Slice для сессии ядра приведен на рисунке 3.15.



Рисунок 3.15 – Исходный код спецификаций сессий ядра

#### 3.1.3.12 Индекс производительности узла

Для численной оценки вычислительных ресурсов узла, на котором запущена служба мониторинга, авторами была разработана шкала оценки ресурсов вычислительной системы.

Основная идея предлагаемой шкалы заключается в нормировании результатов оценки по некоторому базовому значению. В данном случае мы использовали абсолютный показатель производительности системы с процессором Intel™ CoreDuo™ T2300 (1.66 ГГц), доступной оперативной памятью 1 Гб и свободным дисковым пространством 80 Гб. Абсолютная оценка для этой системы составляет *812*.

Абсолютная оценка производительности узла является эвристической величиной и рассчитывается по основным ресурсам вычислительной системы – процессору, памяти, доступному дисковому пространству и текущей загруженности системы.

Любая оценка производительности узла нормируется на это значение. Фактически индекс производительности отражает во сколько раз исследуемая система производительней базовой.

Аналогичный подход используется в тестах оценки производительности компиляторов, виртуальных машин и баз данных SPEC [29].

#### 3.1.3.13 Поведение ядра

При запуске ядро системы находится в «неопределенном состоянии», это длится до тех пор, пока ядро не получит одно из двух типов событий – «сеть доступна» или «сеть не доступна». Эти события переводят ядро в сетевое и автономное состояния соответственно.

Находясь в сетевом состоянии, ядро запускает драйвер Discoverer, который начинает «исследовать» среду на предмет наличия запущенных узлов.

После получения определенного количества пакетов типа Discoverer ядро принимает решение о подключении к какому-либо удаленному ядру. Выбор наиболее подходящего ядра осуществляется по принципу максимального индекса производительности. Эта информация доступна через контекст ядра.

Под подключением в данном контексте понимается обмен удаленными сессиями ядер. При этом одно ядро/узел становится родительским, другое - дочерним.

После осуществления обмена сессиями ядро переходит в следующее возможное состояние – активное или пассивное. Если ядро является родительским, его состояние становится активным, иначе – пассивным.

Автономное состояние ядра характеризуется отсутствием физического соединения с сетевой средой. В этом режиме ядро функционирует с некоторыми ограничениями. Однако при обнаружении сетевого соединения ядро переходит в сетевое состояние.

Активное, пассивное или автономное состояние ядра считаются заключительными и пригодными для нормальной эксплуатации. В то время как сетевое и неопределенное состояния являются промежуточными, своего рода искусственными состояниями.

### 3.1.4 Транспортная подсистема

#### 3.1.4.1 Общее описание

Транспортная подсистема (рисунок 3.16) службы мониторинга реализует основную часть механизмов и примитивов модели распределенного взаимодействия между узлами. Транспортная подсистема представляет собой совокупность следующих логических компонент:

1. ядра;
2. драйверов транспортного уровня;
3. менеджера сессий;
4. многопоточных распределенных алгоритмов.

Транспортная подсистема реализует следующий функционал службы мониторинга:

1. управление удаленными сессиями;
2. мониторинг сетевой активности;
3. именование объектов;
4. адресация;
5. балансировка нагрузки;
6. выбор лидера.



Рисунок 3.16 – Транспортная подсистема

#### 3.1.4.2 Универсальный уникальный идентификатор (UUID)

Для однозначной идентификации объектов в рамках системы авторами использовалось понятие универсального уникального идентификатора (UUID). UUID представляет собой строку из 36-ти символов в формате Unicode (например такую — «5029a22c-e333-4f87-86b1-cd5e0fcce509»).

Стандартная библиотека классов Java уже содержит реализацию UUID в классе java.util.UUID [17].

Понятие универсального уникального идентификатора введено для однозначной идентификации узлов распределенной системы, сообщений, модулей мониторинга, а также любых других сущностей, требующих идентификации в распределенной системе.

Использование строкового идентификатора в данном случае оправдано по нескольким причинам. Во-первых, производительность современных вычислительных систем настолько высока, что они одинаково быстро работают как со строками, так и числовыми данным. Однако использование подобного подхода удовлетворяет требованиям к масштабируемости по отношению к размеру системы. Во-вторых, согласно используемой платформе среднего слоя, все данные, передаваемые по каналам связи, упаковываются в бинарные последовательности и сжимаются, что позволяет снизить объемы сетевого трафика.

#### 3.1.4.3 Уникальный идентификатор узла

Для эффективной работы транспортного уровня системы каждый узел идентифицируется не локальным или физическим адресом, а так называемым уникальным идентификатором узла (NUID). Идентификатор NUID состоит из двух частей — домена и идентификатора в домене. Под доменом распределенной системы мониторинга здесь и далее будем понимать объединенную группу узлов, с запущенными службами мониторинга, способными без каких-либо ограничений взаимодействовать между собой. В некотором смысле домен распределенной системы можно представлять как домен Windows, а узлы распределенной системы как компьютеры в домене [26].

Уникальный идентификатор узла позволяет избежать коллизий на транспортном уровне, разрывает связь между логическим узлом и его реальным адресом в сети.

#### 3.1.4.4 Алгоритм выбора лидера

За основу алгоритма выбора лидера в предлагаемой архитектуре был взят классический алгоритм Чанди-Робертса [31], который основывается на кольцевой топологии сети с односторонней передачей данных. На его базе нами был разработан алгоритм выбора лидера, основанный на широковещательных запросах.

На практике распределенная система всегда образует полный мультиграф. Это необходимо для поддержания таких свойств распределенной системы, как репликация, переносимость и масштабируемость. Построение мультиграфа, достигается за счет посылки специальных служебных сообщений от каждого узла – каждому. В случае с использованием широковещательных протоколов эту операцию можно унифицировать.

Рассмотрим алгоритм выбора лидера:

1. каждый узел отправляет широковещательный запрос в распределенную систему, содержащий свой индекс производительности;
2. каждый узел получает широковещательные запросы от других узлов, содержащие их индексы производительности и сохраняет их в кеш;
3. если новые узлы перестают обнаруживаться в среде или прошел определенный период времени, каждый узел начинает выбирать лидера из узлов, индексы которых записаны в кеше;
4. выбор лидера представляет собой циклический алгоритм, состоящий из:
   1. выбора узла из кеша (с удалением) с максимальным индексом производительности;
   2. попытки подсоединения к выбранному узлу;
   3. если попытка оказалась неудачной, из кеша будет взят следующий узел с максимальный индексом;
   4. цикл повторяется до тех пор, пока состояние узла не изменится (пока лидер не будет выбран).

Алгоритм будет работать даже в случае присутствия одного узла в распределенной сети. Это обусловлено особенностями современных сетевых протоколов: широковещательный запрос получают все узлы сети, без исключения, независимо от отправителя запроса.

#### 3.1.4.5 Обнаружение узлов

Каждый узел (ядро) характеризуется своим внутренним состоянием, которое состоит из:

1. идентификатора узла;
2. имени узла в сети;
3. типом операционной системы, в которой запущен узел;
4. строковыми представлениями адаптеров узла;
5. индексом производительности узла;
6. списком дочерних узлов (идентификаторов);
7. списком родительских узлов (идентификаторов).

С точки зрения авторов этот список является достаточно полным и может гарантировать любое изменение состояния распределенной системы после сбоев.

В данной реализации авторами был спроектирован и реализован механизм обнаружения соседних узлов. Он основан на использовании широковещательных запросов без гарантии доставки.

Для реализации подобного механизма нами был спроектирован специальный драйвер ядра и его стационарный адаптер – Discoverer. Discoverer – драйвер с самостоятельным потоком исполнения, который через определенный интервал посылает в сеть широковещательный запрос, в котором инкапсулирует внутреннее состояние своего ядра.

Таким образом, в распределенной системе каждый отдельно взятый модуль имеет представление обо всей системе целиком и может быть использован для восстановления работоспособности системы после сбоев (репликация, масштабируемость). Схожие подходы применятся в современных методологиях разработки распределенных систем.

#### 3.1.4.5 Сессии транспортной подсистемы

Сессии режима ядра и сессии режима пользователя (раздел «3.1.3.11 Сессии ядра») образуют множество сессий транспортной подсистемы, которые хранятся в так называемых таблицах дочерних и родительских узлов ядра службы.

Для обеспечения отказоустойчивости системы, а также для реализации поддержки механизмов автоматической балансировки нагрузки авторами был спроектирован специальный драйвер ядра – «Aliver» (раздел «3.1.3.8 Драйверы ядра»), который характеризуется самостоятельным потоком исполнения.

Для проверки доступности сессий авторами использовались существующие в платформе среднего слоя механизмы обмена мета-пакетами между распределенными объектами и прокси.

При обнаружении сессии, фактическое соединение через которую стало недоступным, драйвер «Aliver» генерирует ядру события «ParentNodeDied» или «ChildNodeDied», в зависимости от типа сессии. Обработка ядром этого события приводит либо к смене его внутреннего состояния (раздел «3.1.3.6 Состояния и обработчики ядра») либо к балансировке нагрузки.

#### 3.1.4.7 Балансировка нагрузки

Любая распределенная система должна явно или неявно поддерживать механизмы балансировки нагрузки.

При явном подходе балансировка считается отдельным событием и случается либо через определенные периоды времени, либо по наступлению какого-либо другого события, сигнализирующего о том, что система в данный момент функционирует не достаточно эффективно.

При неявном подходе, система балансируется автоматически в процессе выбора лидера. Данный подход использовался авторами при реализации данной системы.

### 3.1.5 Подсистема исполнения

#### 3.1.5.1 Общее описание

Подсистема исполнения службы мониторинга (рисунок 3.17) реализует основные механизмы исполнения модулей мониторинга. Можно выделить следующие компоненты подсистемы исполнения:

1. ядра;
2. драйверов подсистемы исполнения;
3. менеджера модулей.

Менеджер модулей подсистемы исполнения представляет собой совокупность планировщика подсистемы исполнения и обработчика результатов.

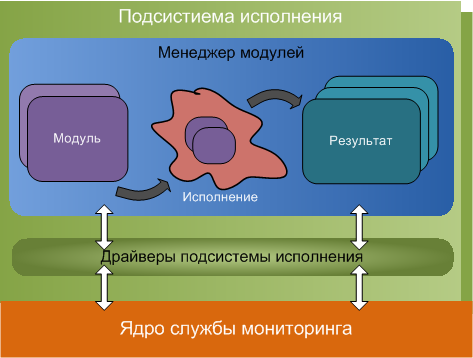


Рисунок 3.17 – Подсистема исполнения

Подсисистема исполнения реализует следующий функционал службы мониторинга:

1. планирование запусков и запуск модулей мониторинга;
2. обработка результатов исполнения модулей;
3. развертывание модулей мониторинга на удаленных узлах.

#### 3.1.5.2 Расписание запусков модулей

В текущей реализации доступно два вида запусков модулей мониторинга:

1. по расписанию;
2. принудительно;

Расписание запусков модуля мониторинга представляет собой последовательность числовых интервалов запусков в секундах. На основании этой последовательности планировщик подсистемы исполнения осуществляет выполнение модулей мониторинга.

Кроме того, существует возможность принудительного запуска, который инициализируется пользователем через панель управления.

Расписание запусков модулей должно сохраняться в энергонезависимую память и иметь возможность восстанавливаться после сбоев или перезагрузок службы мониторинга.

#### 3.1.5.3 Планировщик подсистемы исполнения

Планировщик подсистемы исполнения характеризуется самостоятельным программным потоком, который запускается при переходе ядра в активное состояние и останавливается при выходе из него.

Авторами было решено использовать модель делегирования обязанностей по планированию запусков модулей от дочерних узлов – родительским. Иными словами, планировщик, как сущность, присутствует исключительно на узлах, находящихся в активном состоянии и обслуживает одновременно все дочерние узлы.

Такой подход позволяет добиться эффективного использования распределенных ресурсов вычислительной среды.

#### 3.1.5.4 Развертывание модулей

Развертывание модулей в распределенной системе осуществляется действиями пользователя через панель управления. При этом на каждом узле будет сохранена локальная копия исходного текста модуля, ассоциированная с внутренним уникальным идентификатором (MUID).

Процесс развертывания осуществляется посредством широковещательных запросов в следующей последовательности:

1. передача исходного текста модуля мониторинга;
2. передача списка его параметров;
3. передача расписания модуля.

Модули мониторинга сохранятся в локальном кеше узла именованные своими уникальными идентификаторами.

#### 3.1.5.5 Обработчик результатов подсистемы исполнения

Полученные в результате запусков модулей результаты сохраняются во внутрисистемную очередь для последующей обработки.

Обработчик результатов, характеризующийся самостоятельным потоком исполнения, сохраняет результаты из очереди в ассоциированном с текущим узлом хранилище данных.

В текущей реализации службы мониторинга нет предварительной обработки и анализа результата.

## 3.2 Менеджер модулей

### 3.2.1 Общее описание

Менеджер модулей (рисунок 3.18) представляет собой обособленное приложение, взаимодействующее со службой мониторинга через удаленные RPC-сессии. Служба и соответствующей ей менеджер модулей должны быть запущены на одном узле распределенной системы мониторинга.

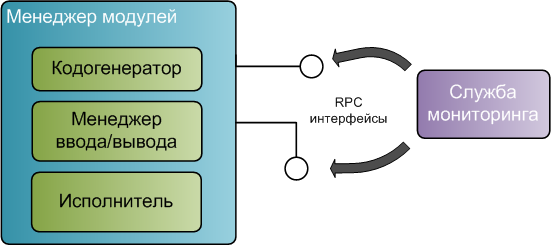


Рисунок 3.18 – Менеджер модулей

Менеджер модулей реализует следующий функционал распределенной системы мониторинга:

1. генерация кода каркаса модулей;
2. исполнение модулей мониторинга в операционной среде;
3. выполнение низкоуровневых файловых операций при работе с модулями.

Авторами было решено разделить операции при работе с модулями на низкоуровневые и низкоуровневые. Например, развертывание модуля инициализируется службой мониторинга, подсистемой исполнения и обрабатывается как низкоуровневая файловая операция модулем мониторинга.

### 3.2.2 Выбор средств реализации

При выборе средств реализации менеджера модулей, авторы в первую очередь ориентировались на языки и средства, выбранные для реализации интерфейса программирования модулей (раздел «3.3.3 Выбор средств реализации»). Это обусловлено возможностью динамического исполнения кода в интерпретируемых языках программирования, таких как Python и PHP.

При запуске модуля мониторинга генерируется так называемый код каркаса, который создает экземпляр модуля и запускает его. Очевидно, что реализовать такое поведение достаточно просто и прозрачно используя одни и те же средства как со стороны программного кода менеджера модулей, так и со стороны кода самого модуля.

В итоге, авторами был выбран интерпретируемый язык общего назначения Python для реализации менеджера модулей.

### 3.2.3 Уникальный идентификатор модуля

Для идентификации модуля в рамках распределенной системы используется универсальный уникальный идентификатор (UUID) именуемый в дальнейшем уникальный идентификатор модуля (MUID). Процедура развертывания модуля на узле, помимо непосредственного сохранения модуля в памяти узла, подразумевает генерацию его уникального идентификатора.

## 3.3 Прикладной интерфейс программирования

### 3.3.1 Общее описание

Прикладной интерфейс программирования позволяет разрабатывать модули мониторинга на основе унифицированного каркаса исходного кода модуля. В текущей реализации интерфейс программирования модулей представляется каркасом с одним публичным методом – «invoke(..)» (рисунок 3.19). Параметры модуля мониторинга могу передаваться как обычная коллекция или список языка Python.

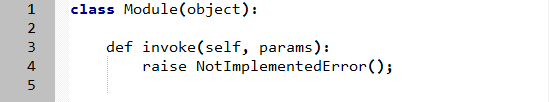


Рисунок 3.19 – Прикладной интерфейс программирования

Интерфейс программирования задает правила исполнения, передачи параметров и возврата результата модуля.

### 3.3.2 Выбор средств реализации

Среди поддерживаемых платформой Ice динамических языков программирования можно выделить два наиболее популярных – Python и PHP, поэтому при выборе языка программирования для интерфейса разработки модулей мониторинга, авторы рассматривали только эти два варианта.

В конечном счете, нами был выбрана платформа языка Python, как наиболее подходящая для реализации механизмов динамического расширения функционала. Язык Python обладает следующими преимуществами по отношению к PHP:

1. более читабельный код и прозрачный синтаксис;
2. ориентированность на различные задачи (не только WEB);
3. большая библиотека стандартных модулей и классов;
4. широкое распространение в сфере администрирования и автоматизации рутинных процессов;
5. наличие и доступность широкого класса сторонних библиотек для решения круга повседневных задач.

## 3.4 Панель управления

### 3.4.1 Общее описание

Панель управления – инструмент управления работой исполняемой среды и визуализации информации процесса и результатов этой работы.

Панель помогает организовать процесс работы, а если более точно этот инструмент является интерфейсом между пользователем и исполняемой средой [32] (рисунок 3.20). Через этот интерфейс задания доставляются от пользователя к узлу, на котором это задание должно быть выполнено и через него же возвращаются результаты работы.

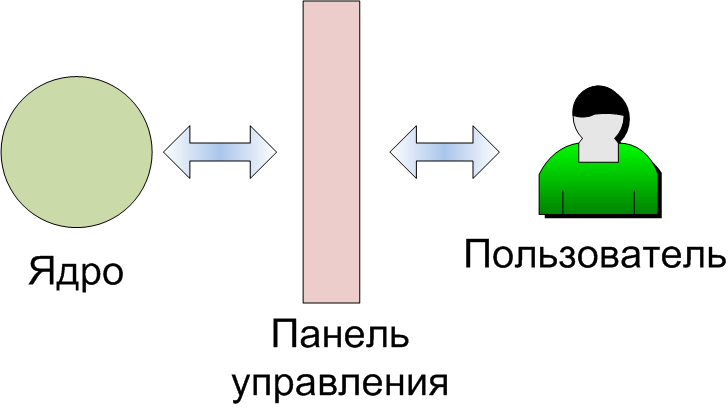


Рисунок 3.20 – Взаимодействие пользователя с ядром

Само приложение обладает большой мобильностью, вследствие отсутствия необходимости установки на жесткий диск компьютера и того, что оно написано на кроссплатформенном языке программирования Java. Конечно, для функционирования программа требует, чтобы на компьютере должна быть установлена виртуальная машина Java.

Точкой входа панели в среду может быть любой узел сети [9], в которой функционирует исполняемая среда. При этом узел, в подавляющей большинстве это персональный компьютер, может быть даже не задействован в этой среде. Другими словами панель может быть запущена на любом компьютере в сети или подсети.

Приложение обладает простым интерфейсом содержащим элементы визуализации информации, такие как дерево узлов, граф домена, списки свойств и результатов, и элементы управления, такие как редактор модулей узла и средства удаленного развертывания модулей, элементы управления состоянием модулей и узлов.

### 3.4.2 Архитектура панели управления

Архитектура панели реализует такую концепцию как Модель – Представление – Поведение (рисунок 3.21). Суть этого шаблона проектирования заключается в том, что  [модель данных](http://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85) приложения, [пользовательский интерфейс](http://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D1%81%D0%BA%D0%B8%D0%B9_%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81) и управляющая логика разделены на три отдельных компонента так, что модификация одного из компонентов оказывает минимальное воздействие на остальные.

Важно отметить, что как представление, так и поведение зависят от модели. Однако модель не зависит ни от представления, ни от поведения. Это одно из ключевых достоинств подобного разделения. Оно позволяет строить модель независимо от визуального представления, а также создавать несколько различных представлений для одной модели.



Рисунок 3.21 – Концепция Модель-Представление-Поведение

### 3.4.3 Модель

Модель в терминах MVC — это не только совокупность кода доступа к данным, но и, как минимум, логика [домена](http://ru.wikipedia.org/wiki/%D0%94%D0%BE%D0%BC%D0%B5%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82) и, возможно, некоторые другие части системы.

#### 3.4.3.1 Получение информации от ядра

Ядро информирует панель управления о своем существовании с помощью драйвера Discoverer. Это драйвер через интерфейс Discover() отправляет минимальный набор информации о ядре. Этот набор информации называется *контекст*. Контекста достаточно чтобы *зарегистрировать* ядро в панели, а также для того, чтобы установить соединение с ядром. После того как соединение будет установлено, через панель можно будет получить более подробную справочную информацию о ядре и адаптеры для управления ядром.

Таблица 3.3 – Содержимое контекста ядра

|  |  |  |
| --- | --- | --- |
| **Ключ для получения** | **Значение** | **Пример** |
| identity | Идентификатор узла | dev/3f759634-59c0-42ae-a923-ad3e61eec769 |
| os | Название операционной системы, на которой запущено ядро | Windows XP |
| parents | Идентификатор родительского узла | dev/92274582-5a96-47a9-942b-676119702ef6; |
| rate | Индекс производительности | 17 |
| primary | Информация для установления сессии до узла | tcp -h 192.168.0.10 -p 10000 |
| secondary | Дополнительная информация для установления сессии до узла | udp -h 192.168.0.10 -p 10000 |
| state | Состояние узла (статус) | ActiveState |
| hostname | Имя узла (компьютера) на котором запущено ядро | Work-PC |
| children | Идентификатор(ы) дочерних узлов | dev/9ea033b6-4a32-4b92-9a70-e57f56bb8d2c; dev/dcee7249-5baa-43c8-a1b2-2f787da599ec; |

Авторами было принято решение, что такое количество информации является достаточным для актуального отображения узла в панели управления.

#### 3.4.3.2 Взаимодействие с ядром системы

Используя информацию из контекста, устанавливается пользовательская сессия до удаленного узла. С помощью этой сессии получаются ссылки на драйверы удаленного ядра. Эти драйверы предоставляют интерфейсы, которые можно использовать для управления и конфигурирования узла.

Таблица 3.4 – Драйверы ядра используемые панелью управления

|  |  |  |
| --- | --- | --- |
| **Драйвер** | **Интерфейсы/методы** | **Описание** |
| Discoverer | discover() | Периодическое оповещение о состоянии удаленного узла через передаваемую информацию |
| Configurer | configuration() | Конфигурирование удаленного узла |
| Hoster | context() | Получение информации об удаленном узле. Отличается от контекста, получаемого с помощью discover(). |
| Sessionier | createUserSession() | Создание сессии до удаленного узла, которая впоследствии дает возможность получить ссылки на другие драйвера |
| Moduler | deploy() | Развертывание модуля на удаленном узле с передачей стартовых настроек |
| force() | Принудительный старт модуля с возможностью передать параметры |
| fetch() | Получение списка модулей узла |
| Scheduler | schedule() | Установления расписания для модуля удаленного узла |
| toggle() | Переключение состояний модулей удаленного узла |
| timetable() | Получения расписания модуля |
| statetable() | Получения таблицы состояний модулей удаленного узла |

Это основные методы для взаимодействия с ядром, но есть еще вспомогательные или, точнее, служебные. Например, такой метод как ice\_ping() есть у всех драйверов [16]. Задача его проста: определение достижимости удаленного узла. В случае недостижимости узла метод генерирует исключительную ситуацию, на которую можно соответственно отреагировать.

#### 3.4.3.3 Домен

В панели управления реализацией модели является класс Domain. Имеется еще несколько классов, которые являются вспомогательными для класса Domain. К ним можно отнести такие классы как DomainController, Discoverer, DiscovererAdapter, но они не являются ключевыми в реализации модели.

К функциям домена можно отнести:

1. Хранение контекста узлов ядра;
2. Хранение ссылок на драйвера удаленного узла;
3. Предоставляет информацию по требованию других компонентов приложения;
4. Информирует пользовательский интерфейс об изменении информации, хранящейся в домене;
5. Контролирует актуальность информации и обновляет ее при необходимости или по требованию;
6. Осуществляет первичное взаимодействие с ядром через драйвер Discoverer.

Вся информация о ядре системы и отдельных узлах, которая визуализируется пользователю, хранится локально в ОЗУ на узле, где запущена панель управления. При этом она не сохраняется в ПЗУ, а в случае перезапуска панели запрашивается у ядра повторно.

Домен содержит адаптер, который предоставляет ядру интерфейс, через который контекст доставляется панели. Это происходит каждые несколько секунд, чтобы панель отображала актуальную информацию.

Полученная информация хранится в динамических массивах, списках, которые в свою очередь содержатся в контейнере Domain (рисунок 3.22).

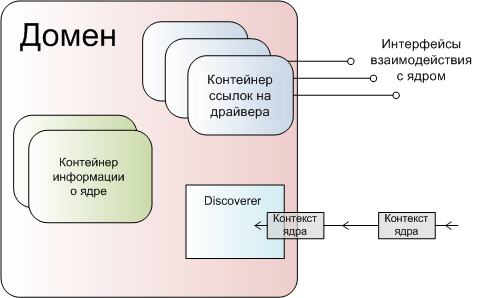


Рисунок 3.22 – Структура домена

Информация кэшируется и если она требуется, то берется из этого кэша, а не запрашивается у ядра каждый раз за исключением ситуаций, когда необходимо принудительно обновить информацию. Имеется два вида обновления информации. Первый вид - это ситуации, когда изменилось состоянии ядра, а точнее его узлов. Тогда ядро через драйвер Discoverer оповещает об этом, рассылая новую информацию в контексте. Используя механизм хэшей, можно определить, изменилась ли информация, и если это произошло, то она обновляется в локальном кэше. Второй вид – ситуации, когда пользователь панели управления изменяет состояния ядра или отдельных его компонентов. И для того, чтобы узнать удачно ли прошли изменения, информация кэша также обновляется, но уже принудительным запросом. Примером второго вида является ситуация запроса пользователем результатов выполнения поставленного задания.

Такой подход позволяет снизить уровень трафика сетевого взаимодействия и повысить отзывчивость панели управления за счет того, что информация в большинстве случаев получается из локального хранилища, а не запрашивается постоянно у ядра.

Еще одним плюсом можно назвать возможность хранить информацию об узлах, которые неожиданно перестали отвечать (сообщать о своей активности). Если узел перестал отвечать, то он помечается как не отвечающий (dead), но информация о нем не удаляется из кэша, а какое-то время хранится. Если до истечения определенного периода времени узел возобновил активность, то его восстанавливают в первоначальное состояние, при этом проверяется, не изменилась ли информация об узле и, если изменилась, то только тогда она обновляется. Если узел так и не возобновил активность, информация о нем удаляется из домена.

Для того чтобы в пользовательском интерфейсе отображалась актуальная информация, необходимо чтобы компонент ответственный за отрисовку обновлял ее своевременно. Постоянно проверять, не изменилась ли информация слишком расточительно с точки зрения ресурсов. И поэтому был использован поведенческий [шаблон проектирования](http://ru.wikipedia.org/wiki/%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) – Наблюдатель. Такое решение было принято по ряду требований к реализации:

* домен должен передавать сообщения интерфейсу, но при этом он не должен знать об его внутреннем устройстве;
* для предотвращения сильных связей между моделью и представлением.

Используемый шаблон полностью справляется с предъявленными требованиями.

### 3.4.4 Контроллер

Контроллер не должен содержать логики приложения (бизнес-логики). Таким образом Контроллер должен выполнять исключительно функцию связующего звена между отдельными компонентами системы.

#### 3.4.4.1 Координатор

В панели управления реализацией контроллера или поведения является класс Coordinator.

Координатор:

1. Устанавливает соответствие между действиями пользователя и изменением информации в домене;
2. Согласовывает информацию из домена и ее отображение в пользовательском интерфейсе
3. Определяет основные действия, определяющие поведение панели вцелом.

Первая заявленная функция координатора является основной.

### 3.4.5 Представление

В реализации представления нет какого-то конкретного класса. Оно реализуется целой группой классов, такие как окна приложения, модели таблиц, деревьев и вспомогательные классов. Но можно выделить основной класс, с которым взаимодействует координатор. Это класс главного окна приложения - MainFrame.

Поскольку панель управления написана на языке Java, то в качестве библиотеки для создания графического интерфейса использовался Swing[29]. Swing предоставляет более гибкие интерфейсные компоненты, чем более ранняя библиотека [AWT](http://ru.wikipedia.org/wiki/Abstract_Window_Toolkit). Также выбранная библиотека является более кросс-платформенной[30].

#### 3.4.5.1 Структура графического интерфейса

В качестве основы для графического интерфейса авторами была концепция многодокументного интерфейса (MDI) [35]. Это одна из стандартных архитектур пользовательского интерфейса Windows-приложений. MDI позволяет работать в приложении не с одним, а сразу с несколькими окнами или документами. Каждое окно показывается в пределах клиентской области основного окна приложения. Это правило не распространяется на модальные окна.

Такая организация интерфейса очень удобна, если необходимо работать с большим количеством однотипных окн. Подразумевается, что в панели управления будет открыто много окн, поэтому такая архитектура была выбрана, как наиболее подходящая.

#### 3.4.5.2 Дерево узлов

Дерево узлов является основным средством отображения структуры ядра (рисунок 3.23). Так же через это дерево можно взаимодействовать с ядром, посредством выполнения над его узлами определенных действий, которые после обработки координатором будут отправлены выбранному узлу или даже группе узлов.

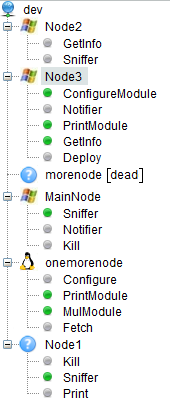


Рисунок 3.23 – Дерево узлов

В дереве можно встретить три вида узлов:

* домен;
* узел;
* модуль узла.

Как было сказано в разделе 3.1.3.2 узел в сети идентифицируется доменом и идентификатором (NUID). Первый вид узлов обозначает домены в сети. Как правило, количество доменов в подсети редко превышает одного.

Ко второму виду относятся вычислительные узлы, которые были описаны в базовой теоретической модели в разделе 2.2. Это дерево не представляет физической структуры узлов, потому что на одном программно-аппаратном устройстве, которое в большинстве случаев является персональный компьютер, можно запустить несколько сущностей ядра. Большая часть информации для дерева берется из контекста ядра, полученного через Discoverer. В качестве имени узла выбирается имя компьютера, а точнее то, что находится под значение hostname. Иконка для узла выбирается в зависимости от значения по ключу OS. Следует заметить, что круг устройств, на котором может использоваться ядро, не ограничивается персональными компьютерами, поэтому определение семейства операционной системы не всегда корректно, если вообще возможно. Если узел перестал информировать о своем существовании, то он помечается как dead. Поскольку узел недоступен, то и нет возможности получить список модулей.

У каждого узла может быть любое количество модулей, в том числе и ни одного. Список модулей можно получить через драйвер Moduler. Через интерфейс fetch() можно получить список, состоящий из MUID и названия модуля. В дереве у каждого узла отображается список всех его модулей. У каждого модуля есть два состояния: активный и неактивный. Это состояние показывает индикатор модуля. Если индикатор зеленый, значит модуль активный, а если серый – неактивный. Статусы модулей получают через драйвер Scheduler через интерфейс statetable(), возвращающий список MUID и статус модуля.

С каждым узлом можно совершать определенный действия. Исключением является узел домена.

Действия доступные для узлов:

* Shutdown – прекратить выполнения ядра на удаленном узле;
* Host Results – вывести список результатов работы всех модулей узла;
* Node Properties – вывести свойства узла, позволяющие просмотреть системные свойства узла и провести конфигурацию ядра.

Действия доступные для модулей узла:

* Force Start – принудительный запуск модуля с параметрами;
* Module Results – вывести список результатов работы конкретного модуля;
* Module Properties – вывести свойства модуля, которые содержат статус модуля, расписание и список параметров.

#### 3.4.5.3 Карта узлов

Карта узлов представляет собой связанный граф [36]. Для удобства карта открывается в отдельном окне (рисунок 3.24).

Важно понимать, что граф отображает не физическую топологию сети и узлов. Здесь связи в графе показывают иерархическую связь, а не физическое соединение. Связь от узла MainNode к Node3 означает, что MainNode является родительским узлом по отношению к Node3. У любого узла должен быть родительский узел, поэтому MainNode связан сам с собой.



Рисунок 3.24 - Карта узлов с установленными родительскими связями

Цвет узлов на графе тоже имеет значение:

* красный – узел в активном состоянии;
* синий – в пассивном;
* желтый – в сетевом;
* серый – узел перестал отвечать (dead)

Первые три состояния полностью соответствуют состояниям ядра из раздела 3.1.3.5. Последнее же состояние неоднозначно. Если узел на графе окрашен в серый цвет, это не значит что ядро в автономном состоянии. Это значит, что текущее состояние ядро неизвестно. На самом деле оно может находиться в любом из возможных состояний.

Граф нарисован с помощью сторонней свободной библиотеки JUNG. Вся информация для отображения графа берется из контекста ядра.

#### 3.4.5.4 Редактор

В панели управления имеется встроенный текстовый редактор. Он необходим для написания исходного кода модуля на языке Python. В редактор можно загрузить уже готовый файл и наоборот сохранить написанный. Из окна редактора можно сразу развернуть командой Deploy написанный или загруженный модуль на удаленный узел. Панель поддерживает множественно развертывание. Команда Deploy вызывает диалоговое окно, в котором предлагается из списка доступных узлов выбрать то или те, на которые необходимо развернуть модуль. В этом же окне сразу можно указать начальный статус модуля, параметры и расписание.

Развертывание модуля происходит через интерфейс deploy(), а выставление расписания и начальных параметров через schedule().

#### 3.4.5.6 Результаты выполнения задач

Результаты выполнения задач узел сохраняет в базе данных, адрес которой указан в его настройках. От туда же их получает панель управления но своими средствами, использую jdbc-драйвера [38].

Результаты для каждого узла или модуля извлекаются из базы согласно настройкам узла и отображаются в виде таблиц в отдельном окне для каждого узла и модуля.

## 3.5 Описание организации совместной работы

В виду того, что проект разрабатывался командой из двух человек, были приняты следующие решения по организации работы в группе разработчиков.

### 3.5.1 Skype (skype.com)

Из-за географической распределенности участников проекта, большинство митингов и коде ревью проходили в режиме «онлайн». Для организации видеоконференций мы использовали популярное приложение VoIP-телефонии Skype.

### 3.5.2 GoogleMail (gmail.com)

Для организации списка рассылки проекта использовались инструменты GoogleMail. Все технические моменты и решения обсуждались посредством переписки на английском языке в закрытом списке рассылки – [snoopy@googlecode.com](mailto:snoopy@googlecode.com). С момента начала проекта было создано 18 почтовых веток общим объемом около140-ти писем.

### 3.5.3 Хостинг проектов Google (goolecode.com)

Площадка для хостинга проектов от Google использовалась для организации репозитория исходного кода, баг-трекераи вики-движка. Проект доступен по ссылке – <http://snoopy.googlecode.com>.

В качестве системы хранения версий использовалась популярная централизованная система SVN [39]. В репозитории хранились исходные тексты проекта, текст пояснительной записки, черновики схем и документов. С момента начала проекта было произведено около 270 ревизий исходного кода.

# Заключение

В результате работы над дипломным проектом была выдвинута и исследована идея построения распределенной системы мониторинга гетерогенной среды. Были изучены аналоги в сфере современных систем мониторинга, произведена оценка их эффективности и применимости согласно выдвинутой модели требований. Вследствие чего, были сделаны выводы о необходимости появления нового класса инструментов мониторинга, в виду неготовности существующих решений удовлетворять выдвигаемым к ним требованиям.

Кроме того, была разработана и формализована модель распределенной системы мониторинга, лишенная недостатков классических клиент-серверных систем и полностью удовлетворяющая выдвинутой модели требований.

На основании формализованной модели был спроектирован и реализован каркас распределенной системы мониторинга и диспетчеризации процессов гетерогенной среды, с применением современных подходов и технологий программирования распределенных систем.

Спроектированное и разработанное в рамках дипломного проекта программное обеспечение каркаса распределенной системы мониторинга может быть использовано для построения высоконагруженных систем мониторинга и дисетчеризации процессов на базе распределенных гетероегенных сетей.

В первую очередь проект ориентирован на использование в коммерческом секторе для обеспечения отказоустойчивости серверов, рабочих станций и встраиваемых устройств, работающих под большой нагрузкой в режиме 24/7.

Можно выделить несколько путей развития проекта:

* разработка дополнительных модулей мониторинга для решений круга повседневных задач;
* реализация программного обеспечения службы мониторинга на нативном ЯП (например, на С++) с целью повышения быстродействия и надежности целевой системы;
* совершенствование компонентов и оптимизация алгоритмов базовой платформы службы мониторинга;
* полномасштабное внедрение и нагрузочное тестирование системы на базе существующей инфраструктуры предприятия, например лаборатории МикроЭВМАлтГТУ;
* сопровождение системы, информационная и техническая поддержка пользователей и администраторов.

# Список использованных источников

1. Таненбаум, Э. Распределенные системы. Принципы и парадигмы / Э. Таненбаум, М. Ван Стеен. — СПб.: Питер, 2003. — 877 с: ил. — (Серия «Классика computerscience»).
2. Буч, Г. Объектно-ориентированный анализ и проектирование с примерами приложений / Г. Буч, Р. Максимчук, М. Энгл, Б. Янг, Д. Коналлен, К. Хьюстон – Вильямс, 2010. – 720 с: ил.
3. Homepage of Zabbix [Электронный ресурс]/ Режим доступа: <http://www.zabbix.com>
4. Nagios [Электронный ресурс] / Режим доступа: http://www.nagios.org
5. Ganglia Monitoring System [Электронный ресурс]/ Режим доступа: http://ganglia.sourceforge.net
6. Mon – Service Monitoring Daemon [Электронный ресурс] / Режим доступа: https://mon.wiki.kernel.org/index.php/Main\_Page
7. Network Monitoring Software Tools with Big Brother by Quest Software [Электронный ресурс] / Режим доступа: http://www.quest.com/big-brother
8. Network Monitoring | Zenoss [Электронный ресурс] / Режим доступа: <http://www.zenoss.com/product/network>
9. Олифер, В. Г. Компьютерные сети. Принципы, технологии, протоколы. Учебник для вузов / В.Г. Олифер, Н. А. Олифер. — СПб.: Питер, 2007. — 960 с.
10. Интерфейс программирования приложений [Электронный ресурс] / Режим доступа: <http://ru.wikipedia.org/Интерфейс_программирования_приложений>
11. What is the Grid? A Three Point Checklist [Электронный ресурс] / Режим доступа: <http://www.mcs.anl.gov/~itf/Articles/WhatIsTheGrid.pdf>
12. PVM: Parallel Virtual Machine [Электронный ресурс] / Режим доступа: <http://www.csm.ornl.gov/pvm/>
13. INTUIT.ru: Учебный курс – Параллельное программирование с использованием технологии MPI [Электронный ресурс] / Режим доступа: <http://www.intuit.ru/department/se/mpitech/>
14. CORBA – official web site [Электронный ресурс] / Режим доступа: <http://www.corba.org/>
15. Erlang – Official Website [Электронный ресурс] / Режим доступа: <http://www.erlang.org/>
16. Henning, Michi. Distributed Programming with Ice [Электронный ресурс]: / Michi Henning, Mark Spruiell; ZeroC, Inc., 2009. - Режим доступа: <http://zeroc.com/download/Ice/3.3/Ice-3.3.1.pdf>
17. Сайт Java [Электронный ресурс] / Режим доступа: <http://java.com/en/>
18. Python Programming Language – Official website [Электронный ресурс] / Режим доступа: <http://www.python.org/>
19. Ruby Programming Language [Электронный ресурс] / Режим доступа: <http://www.ruby-lang.org/en/>
20. Олифер, В. Г. Сетевые операционные системы / В.Г. Олифер, Н. А. Олифер. . — СПб.: Питер, 2009. – 672 с.
21. The Objective-C Programming Language – Mac OS X Developer Library [Электронный ресурс] / Режим доступа: <http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/ObjectiveC/Introduction/introObjectiveC.html>
22. PHP: Hyppertext processor [Электронный ресурс] / Режим доступа: <http://www.php.net/>
23. .Net Framework [Электронный ресурс] / Режим доступа: <http://www.microsoft.com/net/>
24. Eclipse Project [Электронный ресурс] / Режим доступа: <http://www.eclipse.org/>
25. NetBeans Project [Электронный ресурс] / Режим доступа: <http://www.netbeans.org/>
26. Сайт IntelliJIDEA [Электронный ресурс] / Режим доступа: <http://www.jetbrains.com/idea/>
27. Шилдт, Г. Искусство программирования на Java / Г. Шилдт, Д. Холмс – Пер. с англ. – М.: Издательский дом «Вильямс», 2005. – 336 с.
28. Гамма, Э. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Д. Влиссидес. – СПб.: Питер, 2009. – 366 с.
29. Standard Performance Evaluation Corporation [Электронный ресурс] / Режим доступа: <http://www.spec.org/>
30. Руссинович, М. Внутреннее устройство Windows: Windows Server 2003, Windows XP и Windows 2000 / Руссинович М., Соломон Д. Пер. с англ. - 4-е изд. - М.: Издательско-торговый дом "Русская Редакция"; СПб.: Питер; 2005. - 992 с.
31. Параллельное программирование [Электронный ресурс] / Режим доступа: <http://neerc.ifmo.ru/mediawiki/index.php/Параллельное_программирование>
32. Головач, В.В. Дизайн пользовательского интерфейса / В.В. Головач – 2000. – 141 с.
33. Oracle tutorial for swing [Электронный ресурс] / Режим доступа: <http://download.oracle.com/javase/tutorial/uiswing/>
34. JTattoo official website [Электронный ресурс] / Режим доступа: <http://www.jtattoo.net/>
35. MDI [Электронный ресурс] / Режим доступа: <http://ru.wikipedia.org/wiki/Multiple_document_interface>
36. Java Universal Network/Graph Framework [Электронный ресурс] / Режим доступа: <http://jung.sourceforge.net/index.html>
37. Apache log4j [Электронный ресурс] / Режим доступа: <http://logging.apache.org/log4j/>
38. JDBC-driver [Электронный ресурс] / Режим доступа: <http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136101.html>
39. Subversion Project [Электронный ресурс] / Режим доступа: <http://subversion.tigris.org/>
40. Cacti – Official website [Электронный ресурс] / Режим доступа: <http://www.cacti.net/>