

РОССИЙСКАЯ АКАДЕМИЯ НАУК
ДАЛЬНЕВОСТОЧНОЕ ОТДЕЛЕНИЕ
ВЫЧИСЛИТЕЛЬНЫЙ ЦЕНТР

На правах рукописи

Тарасов Алексей Григорьевич

Система мониторинга вычислительного
кластера расширенной функциональности

05.13.11 — математическое и программное обеспечение
вычислительных машин, комплексов и компьютерных сетей

Диссертация на соискание ученой степени
кандидата технических наук

Научный руководитель
чл.-корр. РАН, д.ф.-м.н.
Смагин С. И.

Хабаровск — 2011

Оглавление

Введение	5
1 Архитектура системы мониторинга расширенной функциональности	22
1.1 Основные задачи систем мониторинга	22
1.2 Система мониторинга расширенной функциональности . . .	25
1.3 Взаимодействие с источниками данных	35
1.4 Представление данных	40
1.5 Механизмы проверки условий и уведомлений о событиях . .	43
1.6 Выводы по первой главе	48
2 Реализация системы мониторинга на языке программирования Java	50
2.1 Соглашения о наименовании классов, методов и членов класса	52
2.2 Пакеты поддержки сбора данных	55
2.3 Пакеты для реализации механизма отклика системы	62
2.4 Вспомогательные пакеты	66
2.5 Сравнение разработанной системы мониторинга с аналогами	71
2.6 Выводы по второй главе	78
3 Практическое применение разработанных подходов	80
3.1 Схема работы типичного программного комплекса, использующего разработанные классы	81
3.2 Расширение возможностей установленной системы мониторинга Ganglia	86
3.3 Взаимодействие с системой паравиртуализации Xen	89
3.4 Взаимодействие с нейросетевым модулем	92
3.5 Пути дальнейшего совершенствования разработанной системы мониторинга	98

3.6 Выводы по третьей главе	99
Заключение	101
Литература	103
Глоссарий	115
Приложения	122
А Подробное описание основных классов Grate	123
Б Пример файла настроек grated	135
В Системы виртуализации	138
Г Искусственные нейронные сети	141

Список принятых сокращений

БД — база данных

БК — вычислительный комплекс

ВП — вычислительный процесс

ИНС — искусственные нейронные сети

ЛВС — локальная вычислительная сеть

ОС — операционная система

ПО — программное обеспечение

ППП — пакет прикладных программ

РВС — распределенная вычислительная среда

СМ — система мониторинга

ЦСД — центр сбора данных

ЭВМ — электронная вычислительная машина

API — Application Programming Interface

GUI — Graphical User Interface

JVM — Java Virtual Machine

MIMD — Multiple Instructions Multiple Data

MPI — Message Passing Interface

PVM — Parallel Virtual Machine

TCP — Transmission Control Protocol

XML — eXtensible Markup Language

Введение

Актуальность темы. Во многих направлениях фундаментальных и прикладных исследований имеется необходимость в решении требовательных к вычислительным ресурсам задач. Такие потребности невозможно удовлетворить при использовании для расчётов только персональных ЭВМ (далее в тексте — рабочих станций). Для решения подобных задач были предложены подходы, позволяющие получить значительно большую производительность. Они основаны на вычислительных комплексах (ВК) различной архитектуры: суперкомпьютерах на базе векторных процессоров¹, вычислительных кластерах [1], высокопроизводительных системах с выделенным вычислительным блоком², различных гибридных вариантах³ и др.

Вне зависимости от выбора архитектуры вычислительного комплекса, разработчикам при его внедрении необходимо решить ряд задач. Одной из них является развёртывание средств управления, в том числе системы мониторинга (СМ).

Управление кластером (здесь и далее в тексте под термином «кластер» подразумевается вычислительный кластер) в значительной степени отличается от управления ВК с общей памятью, поскольку контролируемые ресурсы распределены по нескольким рабочим станциям. Средства управления для кластеров обычно решают такие задачи, как:

- выполнение загрузки и остановка работы вычислительных узлов;
- выполнение непараллельных и параллельных команд на кластере с последующей передачей результата пользователю или администратору ВК;
- получение структурированной и унифицированной информации о со-

¹<http://www.cray.com>

²http://www.nvidia.com/object/tesla_computing_solutions.html

³<http://versi.edu.au/downloads/SuperCGrice.pdf>

стоянии любого вычислительного узла, в частности, таких важных данных, как использование вычислительного времени процессора, оперативной и виртуальной памяти, системы ввода-вывода, коммуникационной среды;

- анализ состояния кластера и его компонент.

Если необходимость выполнения перечисленных задач в реальном времени отсутствует, то любое из указанных выше действий можно реализовать с помощью интерпретируемых языков программирования и стандартных механизмов операционной системы: например, Bourne Shell в Unix, PowerShell в Windows, и других. Однако задачи наблюдения и управления в реальном времени намного более сложны и поэтому требуют специализированных СМ и средств управления, предъявляя при этом более жёсткие требования к времени выполнения управляющих воздействий и накладным расходам при использовании ресурсов ВК.

Традиционно, для исследования вопросов организации сбора данных и контроля ВК, локальных вычислительных сетей (ЛВС) и программных комплексов (ПК) разрабатывалось прикладное программное обеспечение (ПО). Исследованиями в этом направлении, в том числе практической реализацией, занимались такие авторы как Р. Вольски, Я. Фостер, М. Дженера-серет (R. Wolski, I. Foster, M. Genersereth) [49,53,91] и другие. Значительные результаты были достигнуты и в смежных областях: в теории управления и управляющих систем, теории автоматов, системном анализе и системах поддержки принятия решений (А. А. Ляпунов, М. Мортон, С. В. Яблонский, О. И. Ларичев, Э. Кодд, Н. Н. Моисеев [5–8, 31, 65] и другие).

Тем не менее, задачи мониторинга по-прежнему являются сложными в практической реализации. Например, большинство кластеров имеют в своём составе управляющие узлы, к которым могут обращаться пользователи. Задания выполняются на наборе узлов, допуск к которым из публичных сетей запрещён из соображений повышения стабильности работы ВК и защиты информации от несанкционированного доступа к данным. В основном, права пользователей ограничены таким образом, чтобы единственным доступным способом запуска заданий на кластере была система

постановки заданий на выполнение. В подавляющем большинстве случаев эта система естественным образом совмещена с системой планирования заданий. Этот подход позволяет администраторам ВК осуществлять более полный контроль над системой, но вместе с тем затрудняет работу пользователя по отладке своего приложения, поскольку предоставляемой такими системами информации недостаточно.

Мониторинг компонент программно-аппаратного комплекса и оказание управляющих воздействий являются критически важными для организации высокопроизводительных распределённых вычислений. После того как началось выполнение приложения, пользователь и администратор кластера нуждаются в информации о том, как выполняется задание, какое влияние оно оказывает на вычислительную систему в целом. При этом с ростом числа вычислительных узлов в составе ВК возрастает и объём данных, которые приходится анализировать администратору, растёт вероятность ошибки вследствие человеческого фактора. Возникает необходимость в автоматизации рутинных операций по контролю за состоянием ВК.

Для коммерческих суперкомпьютеров и кластеров разработано специализированное ПО, обычно поставляемое производителем ВК, например — IBM Tivoli⁴. Такое программное обеспечение нередко совмещает функции мониторинга с функциями управления и тесно взаимодействует с аппаратным обеспечением ВК, предоставляя информацию со всех уровней мониторинга.

Для вычислительных кластеров, стоимость которых невелика по сравнению с коммерческими ВК, ввиду использования менее специализированного и потому более доступного по цене оборудования, подобное указанному выше ПО для них либо отсутствует, либо его применение затруднено в силу лицензионных ограничений. Поэтому обычно используются различные сторонние программные продукты, в задачи которых входит сбор необходимых данных, обработка и представление полученных результатов в удобном для конечного пользователя виде. Такое ПО доступно, в том числе, и на принципах свободного программного обеспечения, которое

⁴<http://www.ibm.com/software/ru/tivoli/>

успешно применяется в научной среде, в частности, в ВЦ ДВО РАН [25].

Подходы, используемые в системах мониторинга в настоящее время, во многом являются следствием развития концепций построения вычислительных систем. Для понимания особенностей архитектур СМ необходимо рассмотреть основные этапы эволюции подходов к организации ВК.

Стоимость создания и эксплуатации суперкомпьютеров велика, поэтому исторически так сложилось, что они располагались в развитых научных и промышленных центрах. Доступ к их аппаратным ресурсам специалистам из других организаций был затруднён вследствие слаборазвитой инфраструктуры связи, географической удалённости вычислительных систем от нуждающихся в их ресурсах учёных и инженеров, а также других причин. Из-за нехватки машинного времени пользователям таких ВК приходилось длительное время ожидать выполнения своего задания. Это привело к появлению пакетных систем и систем планирования заданий на основе очередей для суперкомпьютеров⁵, что, в свою очередь, позволило более эффективно использовать их.

В 90-е годы прошлого столетия с суперкомпьютерами стали конкурировать ВК, состоящие из набора вычислительных станций. Этому способствовали быстрые темпы развития технологий передачи данных в ЛВС. В итоге вычислительные кластеры стали параллельными вычислительными системами, сравнимыми по производительности с суперкомпьютерными системами с общей памятью [41].

Один из ранних подходов к организации кластеров состоял в использовании вычислительного времени простаивающих рабочих станций. Он и в настоящее время применяется для решения ряда задач, таких как, например, анализ принимаемых из космоса радиосигналов в проекте SETI@Home [36] и численные расчёты процессов свёртывания молекул белка в проекте Folding@Home [59]. Для поддержки таких вычислительных систем был разработан ряд пакетов прикладных программ (ППП) [54, 61]. Это ПО в основном было предназначено для централизованного управления ресурсами и распределения процессов непараллельных заданий по свободным рабочим станциям (далее в тексте — вычислительным узлам),

⁵<http://www.clusterresources.com/products/torque-resource-manager.php>

объединённым в ВК. Задача наиболее эффективного использования разнородной вычислительной среды, состоящей зачастую из рабочих станций с различным аппаратным и программным обеспечением, требовала надёжных механизмов работы в унифицированной виртуальной параллельной машине. В этих целях использовалось такое ПО, как PVM [77], MPI [67] (см. также: LAM⁶, OpenMPI⁷, MPICH⁸) и др.

Эти программные продукты позволили стандартизировать на уровне межгосударственных организаций модель обмена данными между вычислительными узлами в ЛВС. В свою очередь, принятие стандартов привело к тому, что, согласно обновляемому каждое полугодие списку высокопроизводительных систем мира TOP500⁹, наиболее распространёнными на сегодняшний день являются вычислительные кластеры, доля которых превышает 80% от общего числа ВК. В аналогичном списке для стран СНГ доля кластеров равна 98% от общего числа ВК¹⁰.

Некоторые ВК низшего ценового диапазона до сих пор построены с применением ЛВС организации, обычно на базе технологий Fast или Gigabit Ethernet в качестве аппаратного обеспечения коммуникационной среды, используемой совместно с невычислительными станциями. Было показано, что пропускной способности и латентности такой ЛВС достаточно для определённого класса задач [48,58,85]. Однако, чрезмерно загруженная сеть или сеть с неэффективной топологией (с точки зрения пропускной способности) заметно снижает производительность ВК в задачах, требующих обменов данными большого объёма. Поэтому предпочтительным является использование выделенных кластеров с изолированной средой передачи данных [87].

Проект Now института Беркли [45] является одним из общеизвестных примеров таких ВК. В работах [76,86] показано преимущество выделенных вычислительных кластеров по сравнению с системами, в которых применяется неизолированная ЛВС. Кластеры, используемые научными учреждениями для решения широкого круга задач, в основном используют вы-

⁶<http://www.lam-mpi.org>

⁷<http://www.open-mpi.org>

⁸<http://www-unix.mcs.anl.gov/mpi/mpich1>

⁹<http://top500.org>

¹⁰<http://supercomputers.ru>

деленную ЛВС. Независимо от типа ЛВС кластер нуждается в средствах наблюдения за состоянием всех компонент, поскольку производительность всего ВК зависит от корректности функционирования каждого из вычислительных узлов.

Пользователю довольно сложно определить воздействие его задания на вычислительную систему, поскольку возникает необходимость, не имея соответствующего инструментария, выделить влияние приложения из суммарного воздействия всех других выполняемых на кластере заданий. Обычно на практике измеряют все характеристики на узле до и после запуска задания. Этот подход не всегда удобен, если вообще возможен: в ряде случаев пользователь не обладает информацией о том, где конкретно будет выполняться его задание. К тому же, по результатам многочасового выполнения задания в таком случае не всегда ясно, происходит ли, например, ошибочное выделение памяти без возврата его системе в наблюдаемом приложении (т.н. «утечка памяти»), или другом, выполняющемся на том же самом вычислительном узле.

Ситуация значительно усложняется, если задание ставится в очередь на выполнение в РВС (например, организованных при помощи инструментальных комплексов Condor-G [52] или Globus [51]), поскольку пользователь вынужден сначала определить кластер, на выполнение в котором было отправлено его задание, а затем получить доступ к данным мониторинга для конкретных узлов. В системах подобных Globus GRAM [51], пользователь работает с кластером посредством интерфейса постановки задания в РВС. Это создаёт неразрешимую на уровне пользователя задачу, т.к. возможности по мониторингу в таком случае ограничены особенностями службы Globus MDS [49], которая ориентирована, в основном, на предоставление редко обновляемой информации, такой, как число вычислительных ядер, общий объём оперативной памяти вычислительного узла и т.п.

Контроль состояния вычислительной системы и текущего потребления ресурсов необходим для выявления проблем в ВК и проведения работ по организации вычислительного процесса (ВП) с более эффективным использованием имеющихся ресурсов. Механизмы автоматизированного определения неполадок и восстановления системы могут использовать

данные, предоставляемые СМ, для определения, в каком состоянии находится вычислительный узел или кластер в целом [12]. Службы предсказания производительности сети могут использовать данные СМ как входные векторы для модели предсказаний [90]. В свою очередь, сами службы предсказания могут использоваться планировщиками заданий с целью выявления, какие ресурсы ВК и когда возможно использовать для планируемого задания при составлении расписания выполнения заданий.

Также полученные данные могут быть использованы для:

- определения стадии выполнения задания, в т. ч. выявления отказов и сбоев, текущего состояния задания, причин перехода в это состояние;
- анализа производительности и автоматической настройки требований заданий к ресурсам в процессе выполнения (в последние годы это становится более востребованным в связи с развитием т.н. облачных вычислений [56]);
- эффективного управления различными службами ЛВС;
- планирования заданий и предсказания поведения вычислительной системы;
- сбора данных о конфигурациях вычислительных узлов и использовании ВК.

Приведённые выше задачи, возникающие при организации работы ВК, являются одними из самых важных, но ими не исчерпывается множество задач, решение которых облегчается при использовании подходящей СМ.

Изначально СМ создавались для контроля состояния сетевых служб, ввиду их большой значимости в ЛВС и глобальной сети Интернет, например проект Network Weather Service [90]; этой же тематике посвящены работы в Bell Labs [47, 57].

В работе [68] описывается применение мониторинга для отслеживания попыток проникновения в контролируемую сеть; вводится понятие двух классов систем определения вторжения: сетевые (network-based) и узловые (host-based). Первые работают на основе контроля потока данных

(или трафика) в сети; вторые принимают решение об обнаружении попытки проникновения в ЛВС на основе контроля данных с рабочих станций.

В [75] специализированные СМ также делятся на централизованные и распределённые. Первые осуществляют сбор данных в каком-либо одном месте. Вторые не используют единый центр сбора данных: все узлы ВК либо владеют полной информацией о системе, либо существует несколько узлов, ответственных за сбор информации. Там же вводится понятие прямого (direct) и косвенного (indirect) мониторинга. К первому относится непосредственный доступ к данным контролируемого объекта, например, данным ядра операционной системы. Ко второму — доступ к той же информации через посредника, которым может являться некоторый файл, содержащий текстовое или двоичное представление значений набора данных.

Для определения состояния ВК применяются различные механизмы. Например, в работах [12, 13] предлагается распознавать типы состояний вычислительного кластера с помощью искусственных нейронных сетей (ИНС), используя данные системы мониторинга Ganglia. В [38, 93] обсуждается подход с использованием сетей Петри для диагностики состояния контролируемой системы, на основе уведомлений о событиях в СМ с дискретными событиями. Событием является такое состояние в контролируемой системе, которое требует дополнительной обработки.

Обработка событий и уведомление пользователей системы о них могут осуществляться различными способами. Системами с уведомлением о событиях на базе подписки являются ISIS [39], Gryphon [33]. СМ XMLBlaster [73] использует XML¹¹ для описания события и нахождения пути к адресату на базе информации о событии. Альтернативой такому подходу является уведомление всех пользователей системы обо всех событиях, либо уведомление лишь администратора вычислительного комплекса. Последний подход используется при мониторинге системной информации в операционных системах, основанных на Unix, при возникновении любых сообщений ядра ОС.

¹¹<http://www.w3.org/XML/>

В JINI¹² определены события (Distributed Event Specification) и указано как объект в виртуальной машине Java (JVM) регистрируется для получения события, генерируемого объектом какой-либо другой JVM. Там же описывается, как производится уведомление при возникновении события. Система мониторинга JAMM [82] определяет способ поиска событий и генерирующих их объектов или служб с помощью локальной службы каталогов [89].

Задачи, решаемые в JAMM, во многом схожи с основанными на SNMP системами HP OpenView и SGI Performance Co-Pilot¹³. Протокол SNMP является одним из средств контроля состояния сетевых устройств; его подробное описание приведено в RFC 1156¹⁴ и RFC 1157¹⁵. Данный протокол может использоваться как средство доступа к аппаратным счётчикам сетевых устройств. SNMP разрабатывался для управления и мониторинга подключенных к сети устройств и узлов, а не распределённых комплексных систем. Интеграция либо построение приложений мониторинга, основанных исключительно на поддержке SNMP, является трудоёмкой задачей. Вместе с тем, несмотря на некоторые ограничения, SNMP счётчики хорошо вписываются в архитектуры СМ как источники данных (или сенсоры).

Сложной в практической реализации задачей является опрос всех источников данных и выработка удобного и не ресурсоёмкого способа опроса всех известных сенсоров. В [40] приводится описание механизма SQL запросов для получения и агрегирования необходимых данных из сети сенсоров.

Чем более масштабными и сложными становятся распределённые вычислительные среды, например, такие глобальные РВС, как GRID-системы [50], тем более важными и актуальными становятся задачи мониторинга и автоматизации управления. Одним из подходов к решению данной проблемы является использование агентов мониторинга [53] в событийно-управляемом программном комплексе управления РВС.

Обычно под агентом понимается автономный настраиваемый ком-

¹²<http://www.sun.com/jini/specs/>

¹³<http://www.sgi.com/software/co-pilot/>

¹⁴<http://www.ietf.org/rfc/rfc1156.txt>

¹⁵<http://www.ietf.org/rfc/rfc1157.txt>

понент программно-аппаратного комплекса, способный производить контроль и управлять какими-либо другими компонентами в составе РВС. СМ представляют собой набор таких агентов, которые в ряде случаев могут перенастраиваться в процессе выполнения. В задачи агентов входит получение соотнесённой со временем информации о состоянии любой системной компоненты, такой как использование процессора и др. В ряде СМ агенты могут предоставлять дополнительные управляющие возможности, в частности, выполнять безопасным образом ПО и анализировать результат выполнения.

При таком подходе СМ обычно реализована в виде двух компонент. В состав первой входят агенты мониторинга, которые установлены на каждом вычислительном узле. Ко второй относятся сервер и клиент сбора статистики. Например, в системе мониторинга проекта SMILE [88] на каждом вычислительном узле запущен фоновый процесс СМА (Control and Monitoring Agent), который периодически собирает статистику использования ресурсов. Затем полученные данные передаются на центральный сервер управления ресурсами SMA (Systems Management Agent). Причина централизации данных в SMA — простота и эффективность обработки информации на одном сервере. Такой вариант подходит для небольших и средних кластеров, но для ВК со значительным числом узлов необходимы разделение данных по принадлежности к зонам (доменам) и применение иерархической обработки информации.

У описанной выше архитектуры, несмотря на очевидные достоинства, есть и следующие недостатки:

1. SMA является точкой отказа, которая нарушает корректность работы всей системы управления. Эта проблема может быть решена с использованием дублирующего SMA.
2. Централизация без иерархии затрудняет масштабирование СМ на больших кластерах и глобальных РВС. Однако, как показано в [88], для кластеров с числом узлов до 64, эта проблема незначительна.

Чтобы сделать агенты переносимыми, разработчики систем мониторинга явно или неявно внедряют уровень (или слой) абстрагирования от ап-

паратного обеспечения, т. н. HAL (Hardware Abstraction Layer). Этот слой предоставляет данные в унифицированном формате, обычно машинно-независимом.

Для исключения большого числа TCP-соединений и нагрузки на сеть, системы мониторинга часто работают поверх UDP протокола [63, 88]. Тем не менее, часть, либо все операции обменов данными, также могут быть реализованы и поверх TCP [43, 72].

Многие СМ и технологии в их основе создавались для решения конкретных задач, стоявших перед их разработчиками. Они успешно и эффективно решают эти задачи в своих областях. Тем не менее, для ВК, на которых установлено устаревшее ПО мониторинга, зачастую переход на новые системы мониторинга затруднен, а ранее использованная СМ уже не удовлетворяет возросшим требованиям. К тому же некоторые возможности недоступны даже ряду современных СМ. Несмотря на всё многообразие архитектур и ПО СМ, до сих пор отсутствует возможность взаимодействия различных СМ между собой, затруднено добавление возможностей к уже разработанным СМ, сложен переход от контроля на низком к более высокому уровню и наоборот.

Анализ достоинств и недостатков различных систем во время эксплуатации ВК, относящемуся к классу Beowulf-кластеров, выявил необходимость разработки универсальной системы мониторинга, удовлетворяющей требованиям к базовым возможностям, простоте добавления новых модулей и взаимодействию с другими СМ.

Цель работы состоит в разработке архитектуры и реализации СМ расширенной функциональности, которая представляет механизмы взаимодействия сторонних СМ и программных комплексов в рамках единой СМ. Для достижения этой цели были поставлены следующие задачи:

1. Разработать архитектуру СМ, позволяющей производить интеграцию построенных на её основе приложений с различным ПО, в том числе сторонними СМ с учётом современных требований к ним.
2. Реализовать кросс-платформенную СМ, построенной на базе разработанной архитектуры.

3. Примененить предложенные подходы для реализации программных комплексов решения задач как общего характера, так и представляющих интерес для узкоспециализированных приложений.

Объектом исследования являются теория и практика организации систем мониторинга вычислительных кластеров.

Предметом исследования являются методы и инструментальные средства контроля разнородных данных в вычислительном кластере.

Методы исследования. При решении поставленных задач использовались методы системного и прикладного программирования; методы объектно-ориентированного и модульного программирования; технологии построения вычислительных кластеров.

Научная новизна. В диссертации представлен подход к архитектуре СМ, позволяющий осуществлять автоматизированный контроль ресурсов ВК и более простое взаимодействие СМ со сторонним программным обеспечением через механизм отклика на изменения в работе вычислительной системы. Новой возможностью при таком подходе является взаимодействие ПО, разработанного на различных языках программирования.

Разработанная архитектура позволяет наращивать возможности мониторинга, в том числе, добавляя сторонние модули к СМ, не останавливая их работы, а также осуществлять совместный мониторинг и анализ данных, поступающих от различных СМ. Подход к расширению функциональных возможностей уже используемых СМ без их остановки и в рамках единой СМ ранее не был описан.

Для взаимодействия с независимыми модулями СМ, в том числе, реализующим нейросетевые алгоритмы с целью определения текущего состояния вычислительного кластера, предложено использовать авторскую механизм передачи информации о событиях в работе контролируемой системы, при котором подписка на уведомления может выполняться в процессе работы СМ.

Практическая значимость. Разработанное ПО позволяет повысить эффективность и надёжность использования вычислительных кластеров.

Создание и внедрение программных средств, предложенных в диссер-

тации, выполнялись в рамках проекта по внедрению экспериментального кластера ВЦ ДВО РАН [10] и работ по созданию РВС ВЦ ДВО РАН [15]. Разработанная СМ Grate помогает выполнять мониторинг вычислительных задач, производящих расчёты по плановым научно-исследовательским работам.

Предложенная архитектура может использоваться как для разработки СМ вычислительных кластеров (основная область применения разработанной архитектуры), так и для разработки СМ в других областях, например, системах контроля потоков данных в локальной вычислительной сети. В процессе разработки системы были получены результаты, представляющие независимую практическую ценность: выполнено тестирование эффективности функционирования вычислительных кластеров, узлы которых находятся под управлением гипервизора Xen; выявлены значимые для работы СМ метрики.

Доступные исходные коды разработанной системы и документированный интерфейс модулей позволяют адаптировать разработанную СМ под нужды конкретного ВК с меньшими усилиями и накладными расходами, чем при использовании аналогичных по возможностям СМ, благодаря использованию стандартизованных форматов передачи сообщений, необходимых для подписки на уведомления и получение данных.

Соответствие диссертации паспорту специальности. Отражённые в диссертации научные положения соответствуют области исследования специальности 05.13.11 - Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей, включающей исследования моделей, методов, алгоритмов, языков и программных инструментов для организации взаимодействия программ и программных систем (пп. 1, 3 области исследования).

Достоверность полученных в работе результатов и эффективность разработанной архитектуры СМ подтверждается опытом успешной практической эксплуатации разработанного автором диссертации ПО Grate в Вычислительном центре ДВО РАН при мониторинге ряда ресурсоёмких прикладных задач.

Апробация. Результаты работы были представлены на XXX, XXXII

и XXXIII математических школах-семинарах им. Е.В. Золотова (Хабаровск, 2005 г., Владивосток, 2007, 2008 гг.), Всероссийской научной конференции «Научный сервис в сети Интернет: многоядерный компьютерный мир» (Новороссийск, 2007), Межрегиональной научно-практической конференции «Информационные и коммуникационные технологии в образовании и научной деятельности» (Хабаровск, 2008 г.), международной научной конференции «Параллельные вычислительные технологии» (СПб, 2008 г.), XI Всероссийской конференции молодых учёных по математическому моделированию и информационным технологиям (г. Иркутск, 2010), международной научной конференции Russia and Pacific Conference on Computer Technology and Applications (г. Владивосток, 2010 г.).

Публикации. Результаты научных исследований отражены в 12-и научных работах [9, 13, 15, 20, 22, 24, 27, 28], включая 1 статью [9] в журнале, рекомендованном ВАК для опубликования научных результатов диссертации на соискание ученой степени доктора или кандидата наук, 2 препринта [10, 11], 1 свидетельство [26] об официальной регистрации программы в Федеральной службе по интеллектуальной собственности, патентам и товарным знакам. В перечисленных публикациях все результаты, связанные с алгоритмизацией, программной реализацией и проведением вычислительных экспериментов на высокопроизводительных системах, получены автором лично. Результаты вычислительных экспериментов интеграции системы мониторинга и компоненты искусственных нейронных сетей, полученные совместно с Писаревым А. В., являются неделимыми. Из совместных работ и публикаций с Сапроновым А. Ю., Шаповаловым Т. С., Пересветовым В. В., Смагиным С. И., Щербой С. И. в диссертацию включены только те результаты, которые принадлежат непосредственно автору.

Структура и объём работы. Диссертация состоит из введения, трёх глав, заключения, библиографического списка из 94 наименований, глоссария и 4 приложений. Общий объём работы — 131 страница, в том числе 28 рисунков и 3 таблицы.

Личный вклад автора. Автором осуществлена выработка требований к системе мониторинга, разработка трёхуровневой архитектуры и реализация системы мониторинга расширенной функциональности, полу-

чение основных результатов проведённых исследований.

Краткое содержание работы. *Во введении* обосновывается актуальность работы, ставятся цели и задачи исследования, приводятся основные положения работы. Анализируется специфика предметной области, существующие подходы и их реализация.

В первой главе сформулированы требования к архитектуре программного комплекса для мониторинга кластера ВЦ ДВО РАН. Приводится более строгая формулировка задач мониторинга, вводятся основные термины. На основе изложенного материала формулируется постановка задачи диссертационной работы, предлагается новая архитектура СМ. Приводится описание элементов разработанной архитектуры трёхуровневой СМ и принципов их взаимодействия.

В предлагаемой архитектуре информация о значениях контролируемых величин (метрики) предоставляется сенсорами в центры сбора данных (ЦСД) одним из доступных способов, например, широкополосным пакетом данных или ориентированным на соединение протоколом ТСР. ЦСД выполняют предварительную обработку значений метрик, проверку выполнения условий на значения метрик, фильтрацию и генерацию уведомлений о событиях (переходе ВК из одного состояния в другое).

Ретрансляторы событий совмещены с ЦСД и предоставляют клиентским приложениям возможность подписываться на уведомления об изменениях в работе ВК, снимая тем самым необходимость осуществления маршрутизации пакетов данных с вычислительных узлов до конкретного пользователя. Любое приложение имеет возможность как зарегистрироваться для получения уведомления о конкретном событии, так и генерировать новые события. Уведомления обрабатываются в определённом порядке, который зависит от установленных пользователем фильтров.

ЦСД могут обмениваться полученной информацией между собой. Пользователь получает сведения от такого центра посредством явного запроса с указанием интересующих его параметров.

Вторая глава демонстрирует особенности авторской программной реализации СМ, структуру модулей и интерфейсов, которые необходимы для интеграции СМ в более сложные программные комплексы.

Описывается ряд интерфейсов и классов языка программирования Java, объединённых в пакеты для более удобного взаимодействия со сторонним программным обеспечением. Программный комплекс СМ в предлагаемой реализации представляет собой клиент-серверное приложение, состоящее из нескольких модулей. Модули выполняются либо на вычислительных узлах (источники данных), либо на выделенных узлах (ретрансляторы сообщений, ЦСД).

Поскольку вычислительный кластер не является географически распределённым ВК со свободным доступом из внешних сетей к вычислительным узлам, то при создании СМ предполагалось, что его безопасность будет обеспечиваться другими компонентами системного ПО. В большинстве практических реализаций вычислительных систем необходимые средства защиты информации — аутентификация клиентов, авторизация для доступа к ресурсам и т. д. — доступны посредством служб операционных систем или систем диспетчеризации задач.

Третья глава содержит описание применения разработанных подходов, архитектуры СМ и ПО на её базе для решения ряда задач: расширения возможностей установленной ранее СМ для контроля соблюдения температурного режима центральных процессоров ВК; взаимодействия с системой виртуализации Xen и нейросетевым модулем. Там же приводятся примеры других задач, решение которых может быть упрощено при использовании предлагаемых подходов.

В заключении диссертации сформулированы основные результаты работы, намечаются направления дальнейших исследований.

В приложениях А и Б представлены более детализированная информация по программным интерфейсам разработанной СМ Grate, описание формата файла настройки. *В приложениях В и Г* приведены краткие сведения из практики применения систем виртуализации и теории искусственных нейронных сетей соответственно. Данная информация полезна для понимания принципов функционирования компонент СМ, взаимодействие с которыми описывается в третьей главе.

Положения, выносимые на защиту.

1. Архитектура и программная реализация СМ расширенной функ-

циональности обеспечивает мониторинг данных на различных программно-аппаратных платформах под управлением нескольких операционных систем.

2. Использование механизма уведомлений о событиях позволяет осуществлять эффективное взаимодействие СМ со сторонним программным обеспечением для расширения его функциональности.

Автор благодарит Пересветова В.В. и Смагина С.И. за руководство и помощь в процессе выполнения диссертационной работы.

Глава 1. Архитектура системы мониторинга расширенной функциональности

1.1. Основные задачи систем мониторинга

Задачи, решаемые системами мониторинга ВК, можно сформулировать следующим образом.

Пусть N — число контролируемых узлов, на i -ом узле измеряются l_i характеристик (в последнюю декаду, в литературе эти характеристики принято называть метриками), все l_i составляют множество $L = \{l_i\}$, $i = 1, 2, \dots, N$.

Обозначим через $M_i(t) = \{m_{ik}(t)\}$ множество значений характеристик на i -ом узле в момент времени t , $k = 1, 2, \dots, l_i$. Положим $M(t) = \{M_i(t)\}$ — множество значений всех контролируемых характеристик в момент времени t .

Назовем кортеж $S_p = \langle M(t), N, L \rangle$ *состоянием* контролируемой вычислительной системы, где p — порядковый номер состояния.

Значимым событием назовем переход из устойчивого состояния S_p в состояние S'_p , при котором происходит существенное для контролируемых характеристик изменение значений множества $M(t)$, указывающее на важные для пользователя СМ изменения в работе ВК.

Задачами системы мониторинга являются:

1. получение от контролируемой вычислительной системы и сохранение значений $M(t)$;
2. выявление значимых событий;
3. запуск средств реагирования (отклика) на значимое событие.

Все эти задачи сложны и могут быть решены различными способами. Данные о поведении вычислительной системы можно собирать и

предоставлять различными способами, что позволяет разделить все СМ на две группы: система-ориентированные и задание-ориентированные системы мониторинга.

Система-ориентированная СМ может предоставить всеобъемлющую статистику о том, как ведёт себя вычислительная система, например, СМ Ganglia контролирует более чем 40 параметров на каждом вычислительном узле [63,72]. Другим примером СМ такого типа является MonALISA [34,60]. Тем не менее, задачи тонкой настройки приложений и выявления недостатков используемых в них алгоритмов невозможно эффективно решать, основываясь на данных лишь система-ориентированного мониторинга.

Задание-ориентированные СМ не могут заменить система-ориентированные, но позволяют более эффективно использовать данные мониторинга для анализа поведения конкретных заданий. Именно это и определило название данного вида СМ. Наиболее востребованы такие системы в РВС, где могут применяться для более точной планировки заданий, а также для приложений, способных изменять требования к вычислительным ресурсам в процессе выполнения, в т. н. облачных вычислениях [56,62]. Свойствами задание-ориентированных СМ обладают многие планировщики заданий.

Вследствие большего объёма данных мониторинг поведения отдельного задания в РВС или среде вычислительного кластера является зачастую более ресурсоёмкой задачей, чем сбор данных в система-ориентированных СМ.

Пользователь СМ заинтересован в получении информации в течение всего жизненного цикла задания, при этом мониторинг ресурсов может производиться на различных уровнях: всей вычислительной системы, кластера, узла и уровне задания. Первый указанный уровень свойственен глобальным распределённым вычислительным средам (РВС), второй — большинству кластеров, третий — система-ориентированным, последний — задание-ориентированным СМ.

Основными характеристиками, представляющими интерес при мониторинге вычислительного кластера, являются разделяемые между различными задачами ресурсы: количество доступных процессоров, объем исполь-

зуемой оперативной и дисковой памяти, нагрузка на сетевую подсистему, количество пользователей и т. п. Простой и оперативный доступ к этой информации позволяет оценивать текущее состояние ВК, реагировать на изменения в вычислительном процессе, находить причины неисправностей и сбоев, а также использовать полученную информацию в высокоуровневых программных компонентах (диспетчеризация задач, системы виртуализации и др.)

Для пользователя СМ является средством отладки программы и наблюдения за процессом решения своей задачи. Он заинтересован в возможности наблюдать расходуемые его программой ресурсы в реальном времени, выявлять в ней «узкие» места. При запуске параллельных программ пользователю обычно необходимо знать общий уровень загрузки кластера и доступные в этот момент ресурсы ВК. При нарушении работы приложения для анализа причин сбоя ему могут понадобиться данные за некоторый период времени, предшествующий аварийному завершению.

Как администратору, так и пользователю важно получать уведомления об изменениях в работе контролируемой системы, особенно критически важных для функционирования кластера или выполнения приложения. Необходима поддержка уведомлений различными способами: электронная почта, уведомление через клиент мгновенных сообщений и др.

При разработке СМ важно учитывать специфику параллельных приложений, выполняемых на вычислительных кластерах. Такие приложения оказывают наиболее сильную нагрузку на вычислительные ядра, оперативную память и, как правило, на среду передачи данных. Программный комплекс мониторинга должен также иметь возможность для удаленного наблюдения.

В диссертации к архитектуре СМ и программной реализации предъявлялись следующие требования. В дополнение к основным задачам СМ должна:

- иметь возможность уведомлять приложения пользователя о значимых событиях, возникающих при работе ВК;
- быть легко настраиваемой для использования различных источников

данных;

- корректно обрабатывать исключительные ситуации: задержки и ошибки при получении данных, отсутствие либо повреждение необходимой информации и т. п.;
- предоставлять набор библиотек, позволяющих интегрировать СМ в сторонние программные комплексы;
- иметь возможность получать и обрабатывать данные сторонних СМ единообразно и прозрачно для пользователя;
- предоставлять изменяемый графический интерфейс пользователя;
- быть надёжной, стабильной и нетребовательной к ресурсам.

Для удовлетворения этих требований была предложена новая архитектура СМ, объединившая в себе подходы, позволяющие решать такие задачи мониторинга, решение которых затруднено при использовании других СМ.

1.2. Система мониторинга расширенной функциональности

Системы мониторинга состоят из различных структурных элементов, взаимодействующих между собой определённым образом, зависящим от архитектуры СМ и способа её реализации. В диссертации предлагается новая архитектура СМ (более подробно она описывается в следующих разделах этой главы). Весь программно-аппаратный комплекс логически разделён на три слабо-связанных уровня [20], отображённые на рисунке 1.1. Передача данных может осуществляться лишь между соседними уровнями. Это позволяет изменять и дополнять уровни, не нарушая общей работоспособности СМ. Существуют СМ, не выделяющие уровни явно или использующие меньшее число уровней. Но для удовлетворения требований, указанных в предыдущем разделе, минимально возможным числом для распределения структурных элементов, как будет ясно из приведённой ниже структуры СМ, являются три уровня.

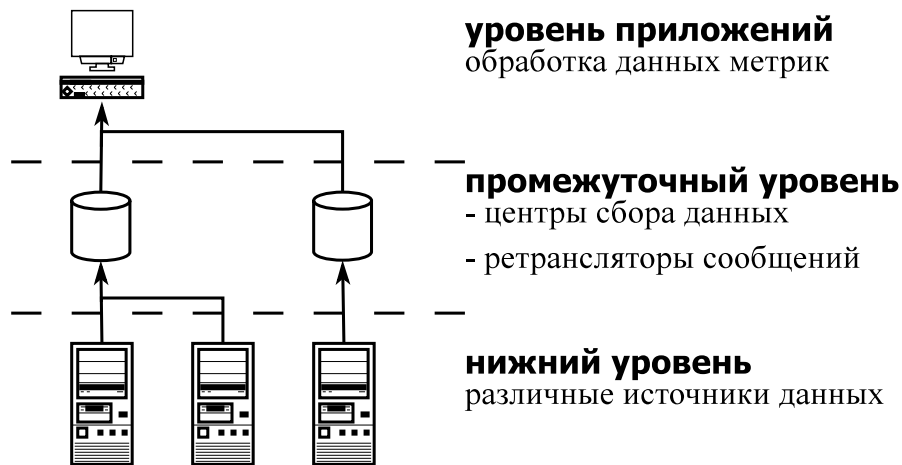


Рис. 1.1: Три уровня системы мониторинга

Нижний уровень ответственен за сбор и представление данных в виде метрик. Источником данных может быть какой-либо сенсор физического или логического устройства, например SNMP-статистика или данные служб сторонних СМ. Функциональность этого уровня мала; в целях более эффективного использования ресурсов контролируемого узла предполагается использование машинно-зависимого ПО.

Промежуточный уровень отвечает за сбор данных с нескольких источников данных, преобразование их к унифицированному формату данных, проверку простейших триггеров и выполнение соответствующих им действий. Этими задачами занимаются *центры сбора данных* (ЦСД). На этом же уровне функционируют *ретрансляторы событий*. Данный уровень можно реализовать на машинно-зависимых или интерпретируемых языках.

Уровень приложений отвечает за анализ данных и представление их пользователю. Быстродействие на данном уровне не так важно, т. к. выполняющееся на нём программное обеспечение может работать за пределами ВК, обращаясь лишь к данным, предоставляемым промежуточным уровнем.

Помимо функциональных возможностей, связанных с мониторингом, на всех уровнях СМ должна предоставлять возможность интеграции с другими программными комплексами через строго определённые интерфейсы взаимодействия.

На рисунке 1.2 представлена общая схема получения и обработки дан-

ных мониторинга в СМ, построенной с использованием предлагаемой архитектуры.

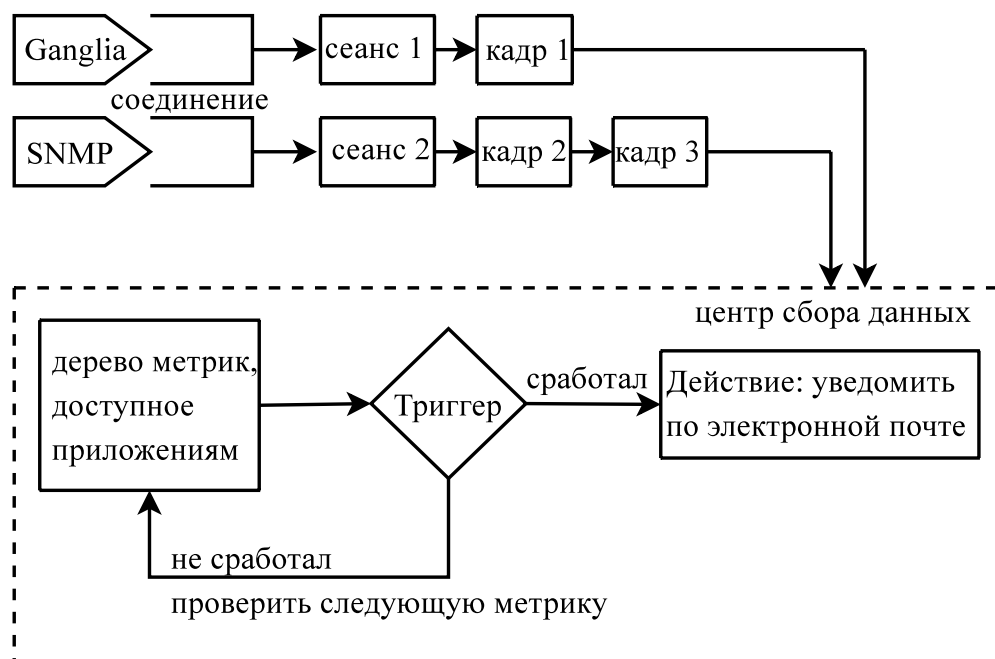


Рис. 1.2: Схема работы системы мониторинга, построенной по предлагаемой архитектуре

В предлагаемой архитектуре *вычислительный узел* отличается от *узла иерархической структуры данных* (ИСД), которая формируется в процессе мониторинга ВК. Если не указано иное, то в тексте под *узлом* понимается второе значение. Трехуровневая архитектура позволяет динамически формировать ИСД, узлом которой может выступать любой определяемый источником данных программный или аппаратный компонент. При этом базовыми единицами хранения данных являются значения метрик.

Метрика — это набор значений определённого типа (целые числа, строки, числа с плавающей запятой и т. п.), отражающий текущее состояние или изменение характеристики в узле ИСД. Метрики бывают статические, т. е. неизменяемые во время работы ОС (например, архитектура аппаратного обеспечения на вычислительном узле, объём физической памяти), и динамические (время работы узла, число операций ввода-вывода в единицу времени и др.). Также они делятся на метрики с накоплением, которые обеспечивают доступ ко всем данным измерений, и одномоментные: хранят только текущее значение. СМ отслеживает непротиворечивость данных по времени, соблюдение порядка помещения данных в метрику. Метрика, в общем случае, хранит набор данных лишь за отно-

сительно небольшой промежуток времени (например, за последний час) и организует эффективный доступ к ним. Остальные данные выгружены во внешнее хранилище в целях уменьшения используемого объёма оперативной памяти. Доступ к ним возможен по запросу пользователя. Каждая метрика существует в рамках определённого узла. Узел имеет изменяемые во времени атрибуты и набор метрик. Отличие атрибута от метрики состоит в том, что атрибут всегда хранит только одно значение. Более строго, *атрибут* — это статичная одномоментная метрика строкового типа. Атрибутами являются вспомогательные данные, используемые при работе с СМ. Например, ими могут являться параметры командной строки, переданные приложению пользователя.

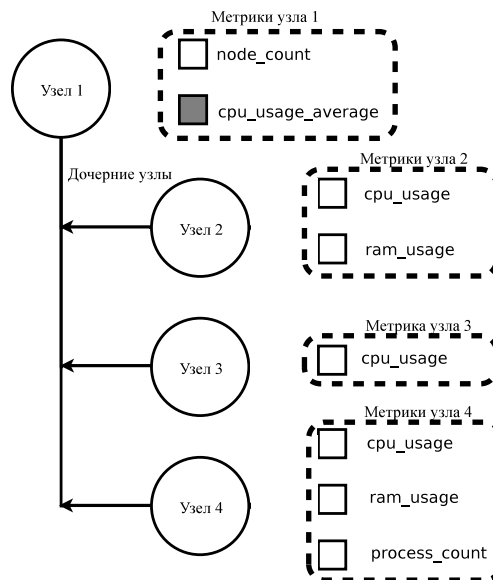


Рис. 1.3: Иерархия узлов системы мониторинга

Каждый узел, за исключением корня иерархии, всегда связан с родительским узлом (рисунок 1.3), а также может иметь набор дочерних узлов. Важным свойством является возможность динамического формирования ИСД, узлом которой может выступать любой определяемый источником данных программный или аппаратный компонент. При мониторинге РВС могут быть сформированы метрики РВС (например, число кластеров), метрики кластеров (число узлов, доля использования всех ресурсов) и другие метрики. При этом, в практической реализации желательно каждой метрике сопоставить символьное имя, сформированное по правилам, схожим с используемыми в спецификации XPath [92] для доступа к узлам XML

документов. Это имя можно использовать в определениях триггеров и для упрощения доступа сторонних приложений к метрикам. При этом каждый ЦСД хранит метрики, принадлежащие некоторому подмножеству узлов.

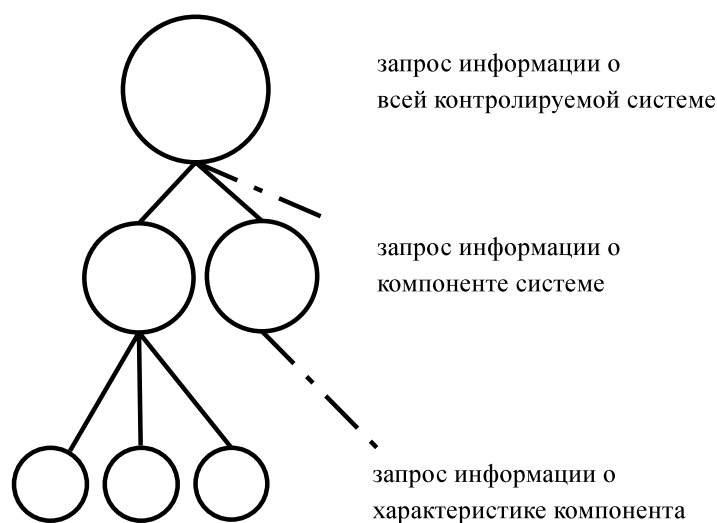


Рис. 1.4: Уровни запроса данных

Как показывают приведённые в работе [63] вычислительные эксперименты по тестированию различных структур сбора данных, использование иерархического сбора данных (N-view, пример на рисунке 1.4) вместо однорангового (1-view) приводит к повышению производительности от 10 до 600 раз для различных уровней обработки статистики мониторинга.

Сборку всех метрик в единое дерево предлагается осуществлять корнем иерархии ЦСД, что позволяет снизить потребление ресурсов на промежуточном уровне. Тем не менее, предлагаемая архитектура не исключает и механизма обмена данными метрик между ЦСД.

ЦСД может производить агрегирование данных и генерировать значения специальной метрики на основе набора однотипных метрик. Например, метрика общей загрузки вычислительного кластера зависит от данных об использовании процессоров всех узлов вычислительной системы. Формирование новых данных производится агрегатором метрик. При наличии соответствующих навыков пользователь СМ может создать необходимые для его специфических задач агрегаторы метрик.

При получении метрик от сенсоров используются соединения, сеансы и кадры.

Кадр — это набор данных, характеризующих метрики и их значения в

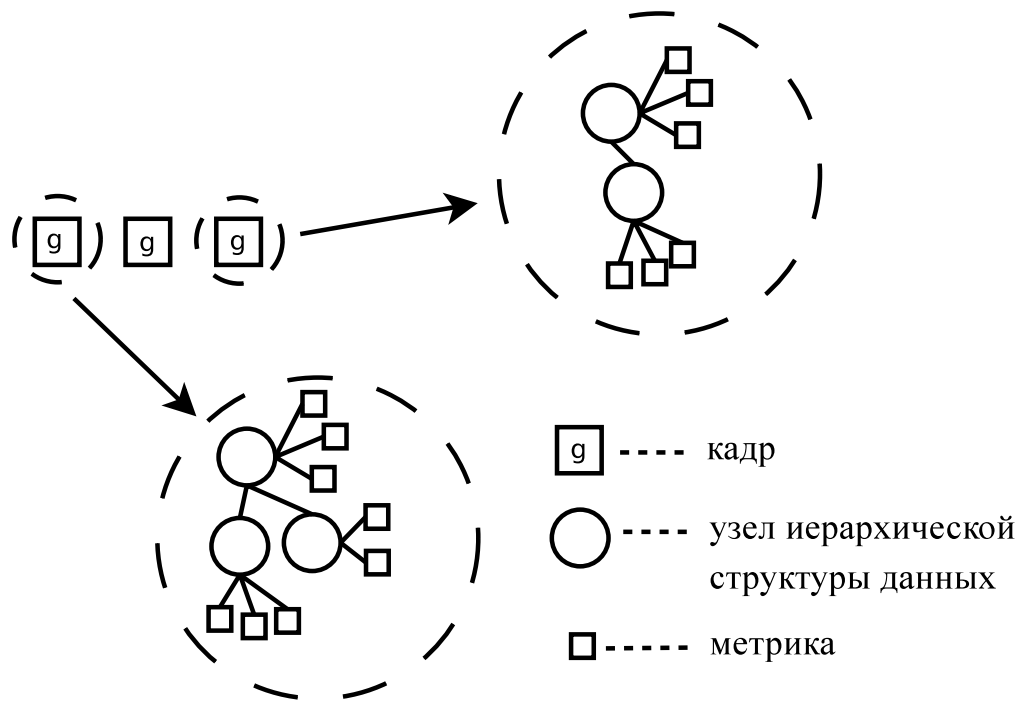


Рис. 1.5: Последовательные кадры

определённый момент времени. В общем случае, в одном кадре содержатся данные для нескольких метрик (см. рисунок 1.5). Отметим, что в описываемой архитектуре СМ полученные значения метрик полагаются неизменными до поступления следующего кадра. Кадр является логическим представлением данных, не зависящим от типа соединения, по которому данные могут передаваться в различных форматах.

Соединение — это инструмент, при помощи которого обеспечивается доступ к данным посредством сетевых коммуникаций, открытия локального файла, запроса к базе данных и т. д. Результатом работы соединения является возможность открытия сеансов.

Соединение можно рассматривать как реализацию механизма получения метрик в неструктурированном формате от сенсора, к которому единожды при инициализации устанавливается подключение (см. рисунок 1.6). В дальнейшем, приложение инициирует открытие сеанса получения данных, используя ранее настроенное соединение. Работа с сеансом может привести к созданию последовательности нескольких кадров. Возможен также вариант, при котором не будет создано ни одного кадра.

Каждый сенсор может производить измерение множества метрик, возможно неодинаковое для различных источников. Через определённые

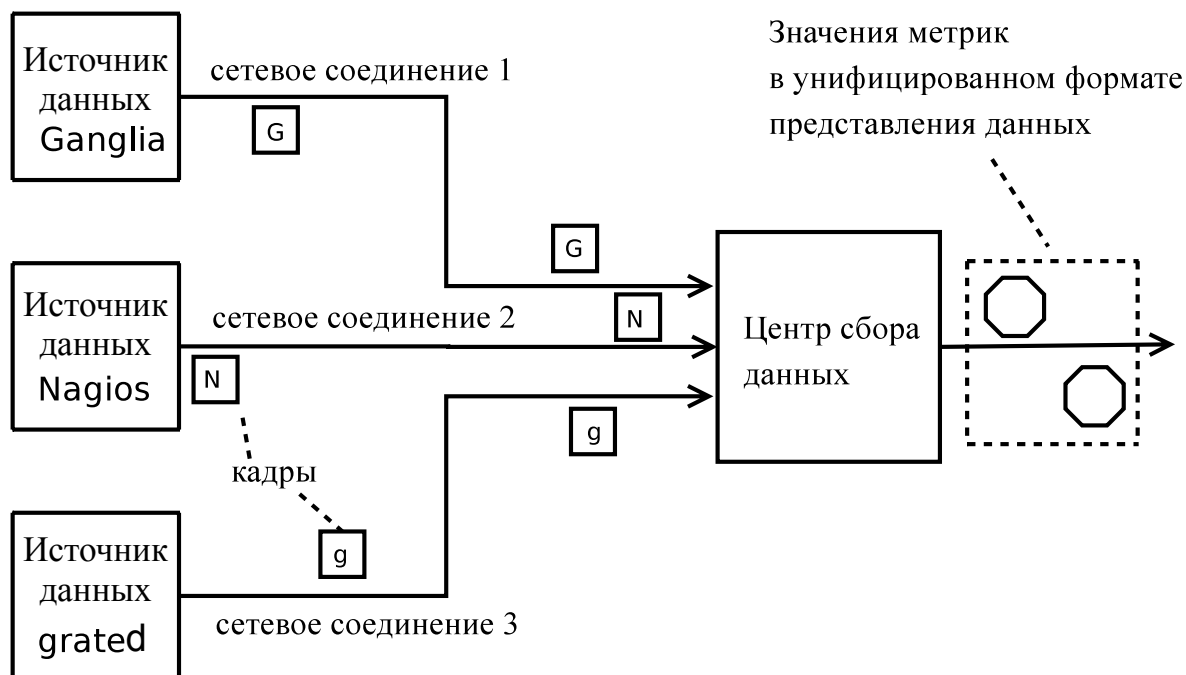


Рис. 1.6: Однотипное соединение с различными источниками данных

промежутки времени источник данных делает мгновенный снимок состояния метрик и передаёт его в процессе сеанса обмена данными на ЦСД, ответственный за обработку информации. Источник данных работает на нижнем уровне архитектуры, ЦСД на промежуточном или уровне приложений.

Сеанс — это объект, идентифицирующий итерацию процесса получения данных. Созданием сеансов занимаются ЦСД.

Следующими важными структурными элементами обсуждаемой архитектуры СМ являются триггер и действие. *Триггер* — это логический элемент, содержащий условие или выражение, истинность которого необходимо проверить для последнего значения метрики. Если условие выполнено, то состояние триггера изменяется, что приводит к выполнению заранее определённых действий. Отметим, что в диссертации под термином «триггер» всегда подразумевается программный логический элемент в том же смысле, что и в области СУБД, но не электронике.

Действие — набор инструкций, реакция на срабатывание триггера. Действия могут быть разнообразными: запись диагностического сообщения в журнал событий; посылка уведомления администратору кластера; запуск

приложений; проверка ещё одного триггера и т. д. Одному триггеру может быть сопоставлено несколько действий и наоборот: одному действию может быть сопоставлено несколько триггеров.

Триггеры делятся на переключаемые и сбрасываемые. Первые находятся в сработавшем состоянии до тех пор, пока условие выполняется, и не производят действий повторно. Вторые выполняют действие каждый раз при поступлении нового значения метрики, если выражение, используемое в определении условия, истинно.

Основная задача триггеров на предварительном этапе анализа данных — проверка выполнения простых условий, наложенных на значения метрик. Предполагается, что проверка триггеров проводится в основном на уровне ЦСД.

В ряде случаев необходимы логические операции над несколькими триггерами, например, выполнение действия, если одновременно выполняются три условия. Возможна проверка всех трёх с использованием последовательной проверки триггеров, однако более производительным решением является введение агрегаторов, которые включают в себя триггеры или другие агрегаторы, каждый из которых проверяет результат выполнения логических операций «И», «ИЛИ» для набора указанных триггеров. Использование масок идентификаторов триггеров при описании агрегатора позволяет динамически добавлять триггеры при включении новых узлов и появлении новых метрик в вычислительном кластере.

Генератор триггеров (или мета-триггер) — объект, создающий триггеры для метрик, подходящих под заданные условия. Необходимость создания этого элемента продиктована динамической структурой иерархии узлов системы мониторинга, при которой часть узлов уже может быть создана (и иметь активные триггеры), другая — отсутствовать и формироваться по мере необходимости. Мета-триггеры облегчают добавление новых узлов в вычислительную систему тем, что позволяют создавать триггеры на базе указанного шаблона без вмешательства пользователя или администратора.

В ранних версиях разработанной архитектуры [16] для выявления событий предлагалось использовать только триггеры. Однако, несмотря на гибкость данного подхода, у него имеются следующие недостатки: необ-

ходимость для пользователя реализовывать проверку условий на уровнях СМ (в том числе в пределах ВК, что повышает нагрузку на вычислительные узлы), реализовывать свой способ уведомления своих приложений о выполнении условий; невозможность структурированной и иерархической обработки события и другие, менее важные.

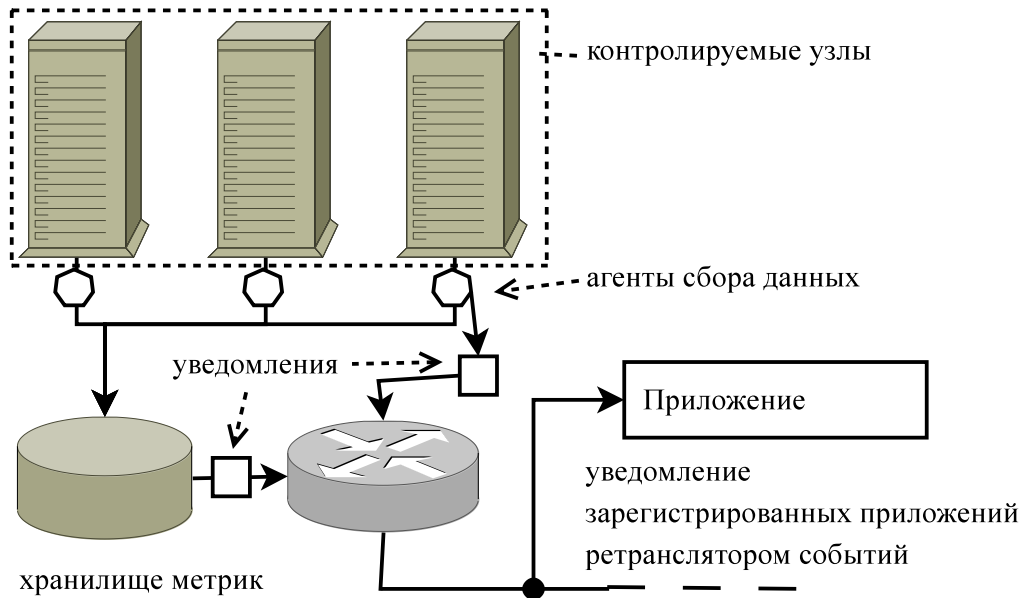


Рис. 1.7: Схема работы системы уведомлений

Поэтому было предложено внести в архитектуру механизм подписки на уведомления о событиях — см. рисунок 1.7. Возможность уведомлений о событиях позволяет устранить перечисленные выше недостатки. Каждый контролируемый узел системы мониторинга может генерировать сообщение о возникшем событии. Уведомление может поставляться в рамках автоматически создаваемой метрики, например, с типом данных «XML-объект». Эта метрика может поставляться как по обычному соединению, по которому передаются кадры данных, так и по выделенному. Если центр сбора данных сохраняет историю данных метрик, то метрика события, как и всякая другая, будет сохранена для дальнейшего анализа.

Пользовательские приложения «подписываются» (регистрируются) на ретрансляторе событий, в качестве которого обычно выступает центр сбора данных уже в процессе работы СМ. При подписке указывается, в уведомлениях о каких событиях заинтересовано приложение и в каком формате предоставлять информацию о них. Основным способом является передача сообщения подписавшемуся приложению в формате XML. Таким

образом, при подписке создаётся фильтр уведомлений на ретрансляторе событий.

ЦСД получает информацию о событии и определяет необходимость уведомления подписчиков, если оно подходит под ранее установленный фильтр. При наличии зарегистрированного приложения ему отправляется уведомление. Далее, с целью проверки подписки сообщение ретранслируется на другие ЦСД.

Клиентское приложение может обработать события, игнорировать их (в этом случае событие передаётся для дальнейшей обработки на ретрансляторе) или же генерировать новые события. Например, при возникновении события отказа сетевой платы, подписавшееся приложение может сгенерировать более конкретные события: «необходимость миграции задач с узла»; «необходимость уведомления администратора»; «необходимость удаления узла из списка ресурсов вычислительного кластера» — которые, в свою очередь, могут быть обработаны другими подписчиками.

Если подписок на одно и то же событие несколько, то все подписавшиеся приложения получают уведомление в порядке, зависящем от приоритета, указанного при подписке. Таким образом, если администратор кластера и технический специалист с помощью клиентского ПО подписаны на событие «уведомление администратора», то оба получают соответствующие сообщения, например, по электронной почте.

Приложения, занимающиеся анализом метрик, выполняются на третьем уровне разработанной архитектуры и принимают данные в структурированном виде от центров сбора данных посредством запроса необходимых данных. В зависимости от реализации запрос может содержать указания по агрегированию необходимых данных: осреднить данные по всем узлам, показать суммарную информацию и другие. В ряде случаев, если определён агрегатор метрик, то при получении центром сбора новых значений автоматически формируется специальная метрика. Приложение может запрашивать эту конкретную метрику для получения обобщённых данных, в других случаях механизм отчёта должен принять запрос в определённом формате и представить отчёт на основе имеющейся информации.

Запрос принимается сервером отчётов, который передаёт этот запрос

элементу структуры СМ, занимающемуся формированием отчётов. После создания ответа сервер выдаёт полученные данные клиенту. На данном этапе также возможно включение пользовательских модулей формирования отчётов, что позволяет разработчикам стороннего ПО получать необходимые сведения в наиболее удобном для них формате.

Такая архитектура позволяет решить все задачи мониторинга, сформулированные в первом разделе данной главы, удовлетворив требования к базовым возможностям системы мониторинга. Разделение элементов СМ по трём уровням позволяет реализовать опрос различных сенсоров единообразно, тем самым обеспечив совместную работу нескольких СМ. Применение механизма подписки обеспечивает взаимодействие СМ со сторонними программными комплексами.

1.3. Взаимодействие с источниками данных

В этом разделе описана схема взаимодействия системы мониторинга с внешними источниками данных в рамках разработанной архитектуры. Приведены элементы архитектуры, относящиеся к процессам получения данных и организации сетевых и иных соединений.

Три уровня системы мониторинга

Базовой концепцией предлагаемой архитектуры является разделение функций системы мониторинга между тремя уровнями: нижним уровнем, промежуточным уровнем и уровнем приложений. Каждый из них предоставляет интерфейс для взаимодействия с другими уровнями и отвечает за решение своих задач, которые были ранее приведены рядом с названием уровня на рисунке 1.1.

Необходимость в разделении функций, реализованных на каждом из уровней, возникла в силу требования обеспечить более простой и удобный способ интеграции СМ, разрабатываемых по данной архитектуре, и уже имеющегося ПО. Также важно было исключить сильную зависимость различных модулей системы друг от друга, что позволило более гибко модифицировать и подстраивать функциональность под нужды конкретного

ВЫЧИСЛИТЕЛЬНОГО КОМПЛЕКСА.

Уровень приложений:
уведомление о событии, действие

Промежуточный уровень:
метрика, триггер, событие

Нижний уровень:
сенсор, сеанс, соединение, кадр

Рис. 1.8: Основные элементы архитектуры для каждого уровня

В итоге, для пользователя системы все данные проходят между уровнями так, как показано ранее на рисунке 1.1, при этом различные структурные элементы СМ работают на логически разделённых уровнях (на рис. 1.8 приведены основные элементы для каждого уровня), формируя потоки данных между уровнями таким образом, что формат передаваемых данных от нижнего уровня архитектуры к более высокому становится унифицированным и структурированным. Уровень приложений получает доступ к любым данным от любого сенсора в едином формате.

Источник данных

Источники данных, или *сенсоры*, располагаются на первом уровне обсуждаемой архитектуры, являясь первым звеном в цепи обработки данных. Их задача состоит в опросе оборудования или программного обеспечения в целях получения текущего значения контролируемых величин и передаче их как значений определённой метрики на более высокий уровень архитектуры.

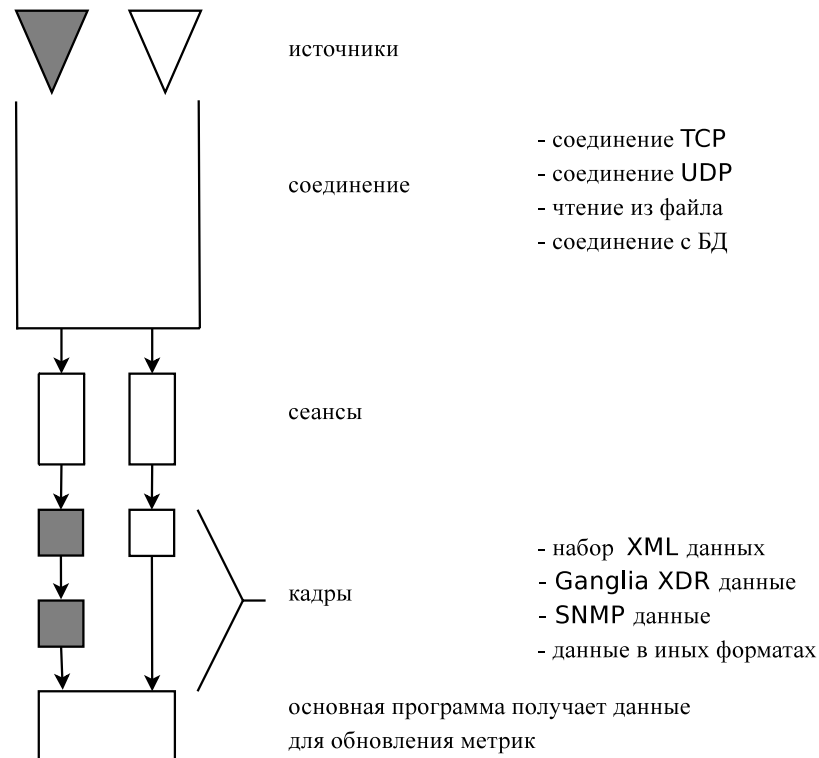


Рис. 1.9: Процесс получения данных в описываемой системе мониторинга

Механизм передачи данных скрывается в объекте соединения, формат данных скрывается в кадре, поэтому сенсор может передавать их во внутреннем представлении, не тратя ресурсы вычислительного узла на приведение данных к унифицированному виду, используемому в ядре системы мониторинга. Общая схема получения и обработки данных от сенсоров приведена на рисунке 1.9.

Разработанная архитектура предполагает, что можно использовать сенсоры других СМ в качестве своих (если соответствующие типы кадров реализованы в ядре системы или загружены на этапе выполнения). Таким образом, для поддержки какого-либо нового типа сенсора обычно необходимо лишь создать обработчик нужного типа кадра, в котором помимо собственно значений наблюдаемой величины будет отметка времени, когда были получены передаваемые величины. Все остальные возможности, предоставляемые предлагаемой архитектурой СМ, будут доступны при работе с данным сенсором, включая поддержку триггеров, агрегацию метрик и другие.

Соединение

Соединение является механизмом подключения к источнику данных, одним из первых этапов процесса получения данных. Использование соединения позволяет создавать сеансы получения данных, независимые от формата передаваемых в них данных. Соединение в рассматриваемой СМ инкапсулирует в себе все параметры, необходимые для подключения к сенсорам. Однако данные не обязательно поступают непрерывно или передаются через такие ориентированные на подключения соединения, как, например, используемые в протоколе TCP.

Модель передачи данных метрик в разработанной СМ можно представить как стандартную модель OSI¹, но с меньшим числом уровней.

- *Уровень соединения*, на котором решаются задачи установки подключения к различным источникам данных посредством единого интерфейса.
- *Уровень сеансов*, идентифицирующий конкретный запрос данных от приложения в рамках соединения. Таким образом, может существовать несколько сеансов получения данных от одного источника данных. Данные от сенсоров, в рамках различных сеансов, могут поступать в различных форматах. Преобразованием этих данных занимается уровень кадров.
- *Уровень кадров*, где полученные с нижнего уровня данные преобразуются в значения метрик, которые можно единообразно читать из других модулей системы.

Процесс получения данных, проходящих через три уровня, также можно представить в виде модели «потребитель-поставщик». Поставщиком выступает источник данных, доступный через уровень сеансов, а потребителем выступает объект на уровне кадров, преобразующий поступившие данные в единый формат метрик. Схематично, эта модель изображена на рисунке 1.10.

¹[http://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994\(E\).zip](http://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994(E).zip)

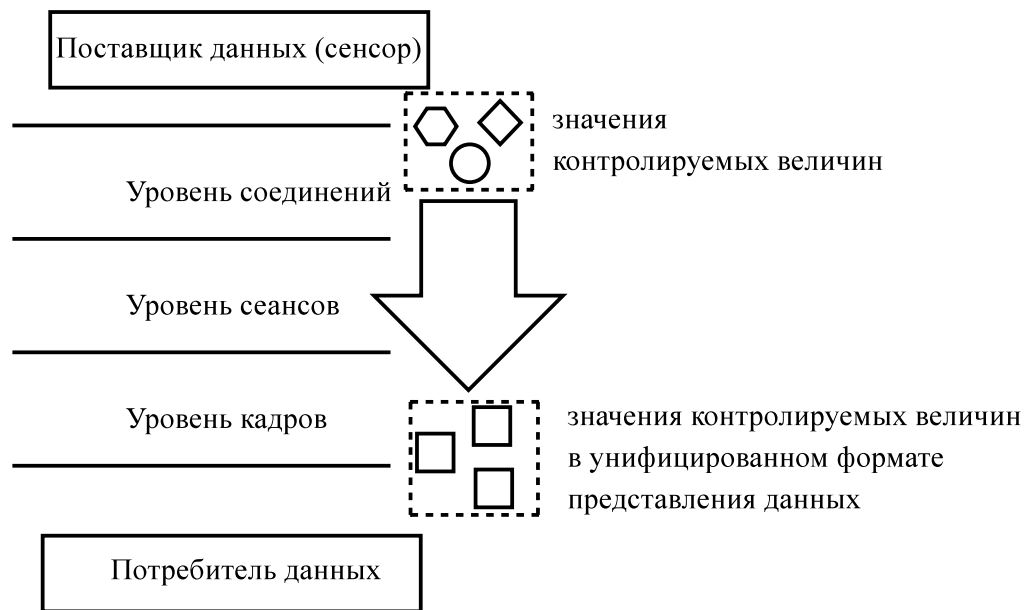


Рис. 1.10: Модель потребитель-поставщик в описываемой системе мониторинга

При практической реализации СМ может возникнуть необходимость явным образом задать соответствие типа поставщика (тип соединения) и типа потребителя (тип кадра). Это никоим образом не сужает область применения системы, но сильно упрощает её архитектуру. Если необходимо автоматически определять тип кадра во время выполнения СМ, то можно ввести объект псевдо-кадра. Он будет пытаться обработать полученную посредством соединения информацию с помощью известных ему объектов-кадров. Первый тип кадра, преобразовавшего данные в метрики без ошибок, будет использоваться для помещения данных в хранилище в дальнейшем, не оказывая влияния на производительность. Альтернативным подходом может являться инкапсуляция всех принимаемых кадров в пакет данных с указанием типа кадра. Такой подход требует доступа к программному коду сенсоров, что не всегда возможно и оправдано.

Сеанс

Сеанс идентифицирует конкретный процесс получения данных из множества, возможных для одного и того же соединения. Это позволяет использовать несколько логически независимых подсистем мониторинга или независимых центров сбора данных в рамках единой СМ. На рисунке 1.11 схематически изображены три сеанса, одновременно открытых для

одного и того же соединения. В различных сеансах получены различные наборы кадров, при этом внутри одного сеанса все кадры передаются в одинаковом формате.

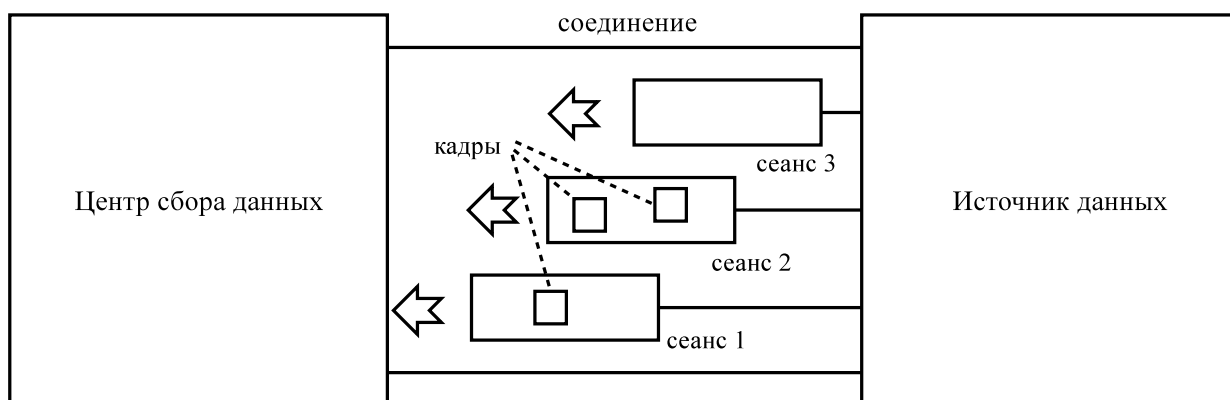


Рис. 1.11: Несколько сеансов, открытых для одного соединения

Принимаемые от сенсора данные передаются по соединению, доставляя в итоге сформированные кадры в ЦСД.

К задачам, решаемым при создании сеанса, могут относиться также корректное создание запроса на получение данных от сенсоров и аутентификация. В варианте СМ, приведенном далее в тексте работы, мы не используем аутентификацию при получении метрик, поскольку авторская программная реализация предполагает использование механизмов ОС для защиты от несанкционированного доступа к данным.

1.4. Представление данных

В этом разделе приведено описание структур данных, которые формируются после опроса сенсоров. Рассмотрены особенности этих структур, важные для реализации системы мониторинга.

Метрика

Базовой единицей хранения данных является метрика. Метрика — это набор значений определённого типа (целочисленного, чисел с плавающей точкой, строкового, двоичного), отражающий текущее состояние или изменение характеристики на конкретном узле. В общем случае, тип метрики может изменяться со временем, однако на практике подобный подход

редко используется в силу того, что тип контролируемых данных обычно известен в момент создания метрик и не предполагает смены в процессе работы СМ.

Метрики бывают статические, т. е. неизменяемые во время работы системы (версия операционной системы на вычислительном узле, объём физической памяти), и динамические (время работы узла, объём доступной физической памяти). Разработанная архитектура предполагает, что все неизвестные метрики по умолчанию статичны и лишь при получении отличающихся данных происходит формирование метрики динамического типа. Это позволяет экономить память при обработке метрики, для которой не известно достоверно, динамична она или нет.

Метрики также делятся на метрики с накоплением (обеспечивают доступ ко всем данным измерений) и одномоментные (хранят только текущее значение). Поскольку у метрики может быть относительно большое число значений, а для проверки триггеров и отображения графиков изменения величин используется сравнительно малое их количество, — хранить их все в оперативной памяти нецелесообразно. Поэтому они выгружаются в какое-либо хранилище данных через определённый в ядре системы интерфейс, организуя доступ к выгруженным значениям по мере необходимости. Таким образом, метрика в общем случае хранит лишь набор данных за относительно небольшой промежуток времени и организует эффективный доступ к ним.

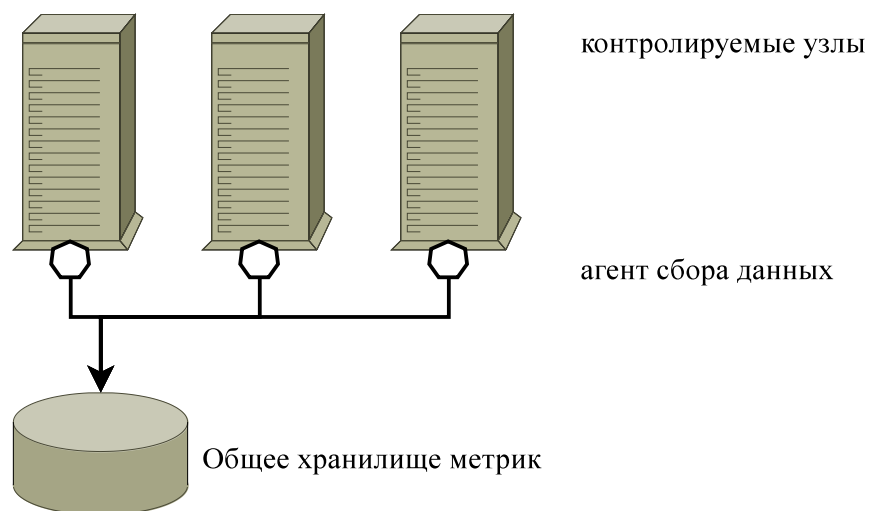


Рис. 1.12: Централизованное хранение и обработка метрик

Программный комплекс отслеживает непротиворечивость данных по времени, соблюдение порядка помещения данных в метрику. При этом, хотя изменение метрики дискретно, значение полагается постоянным и неизменным до следующего полученного значения. Таким образом, формируется кусочно-линейная функция изменения значений метрики, что используется при обработке метрик триггерами, а также агрегаторами метрик — при необходимости сформировать значение при отсутствии у части контролируемых метрик новых значений.

В процессе работы СМ, метрики со всех узлов попадают в один из ЦСД (рисунок 1.12), в котором производится их обработка, проверка условий триггеров и другие операции.

Агрегатор метрик

В некоторых случаях пользователю интересна интегральная характеристика, например, среднее или наибольшее значение из определённого множества каких-либо метрик на текущий момент времени. Подобные возможности легко реализовать на уровне приложения. Однако, если одни и те же вычисляемые характеристики нужны некоторому количеству приложений, то возможность получать необходимые значения на уровне системы мониторинга позволяет снизить общие вычислительные расходы, производя формирование метрик единожды.

По умолчанию система создаёт несколько простых агрегаторов, в которые входят метрики всех вычислительных узлов, входящих в кластер. Пользователь может создать и свои агрегаторы, если того требует его задача. Обычно агрегаторы метрик работают в ЦСД — рисунок 1.13

Объект-агрегат хранит ссылки на все входящие в него метрики. При поступлении новых данных агрегат высчитывает новое значение, которое помещается в итоговую метрику.

Для практических целей были реализованы следующие агрегаторы: «минимальное значение», «максимальное значение», «сумма значений», «среднее значение». Остальные могут быть добавлены своей реализацией интерфейса агрегатора метрик.

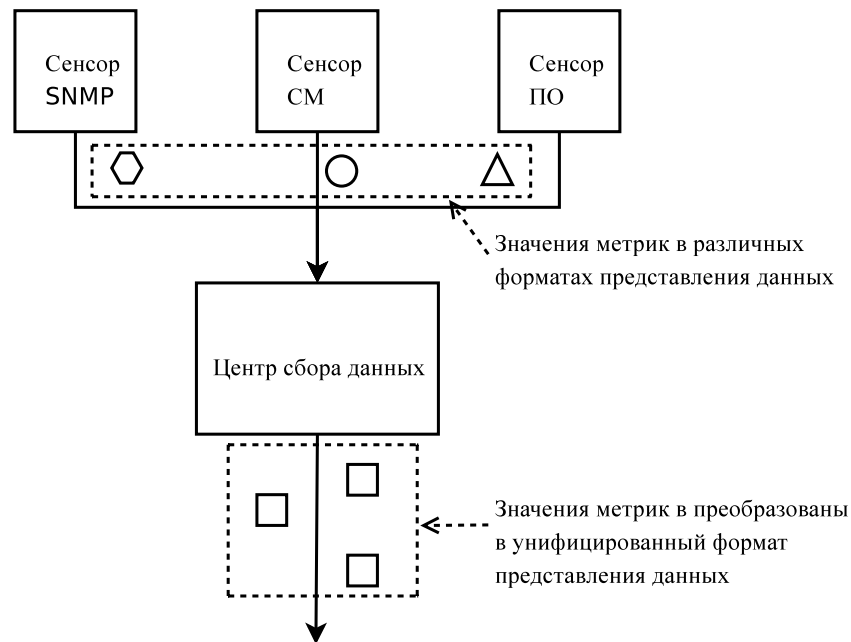


Рис. 1.13: Агрегация метрик в центре сбора данных

1.5. Механизмы проверки условий и уведомлений о событиях

В этом разделе внимание уделяется способам реализации механизма обратного отклика на какие-либо изменения в контролируемых данных. Описаны такие элементы разработанной архитектуры, как: триггеры, события, действия и другие.

Триггер

Для проверки условий, наложенных на метрики, используются триггеры, которые опрашивают одну или несколько метрик, приводя в итоге к выполнению действий, одним из которых, в частности, может являться запуск проверки другого триггера.

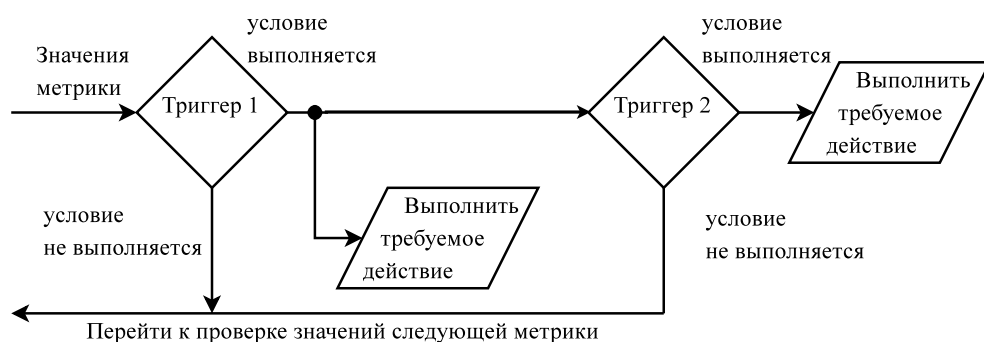


Рис. 1.14: Связь между триггером и действием

Одному триггеру может быть сопоставлено несколько действий и наоборот, что позволяет организовывать достаточно сложные алгоритмы обработки данных, см. рисунок 1.14. В предлагаемой архитектуре триггер не может передать значение другому триггеру без использования объекта-действия. Это позволяет унифицировать обмен данными между элементами, но не позволяет удобным образом использовать сложные условия. Для устранения этого недостатка в архитектуру введены агрегаторы триггеров.

Мета-триггер

Динамическая структура дерева метрик означает, что невозможно на этапе запуска системы мониторинга создать все необходимые триггеры, так как при добавлении нового вычислительного узла в процессе работы ВК появятся новые метрики и на их значения не будет наложено никаких условий.

Чтобы избежать подобной ситуации, нами было введено понятие мета-триггера, т. е. такого объекта, который создаёт для новых метрик необходимые триггеры по какому-либо заданному пользователем правилу. СМ при получении новых данных проверяет существование метрики, для которой пришли данные и при её наличии просто добавляет новые значения в метрику.

Если метрика ранее не была известна, то сначала просматриваются все мета-триггеры и проверяется, не попадает ли метрика под условия, заданные в мета-триггере (в предлагаемой реализации СМ Grate — это совпадение имени метрики и заданной пользователем маски). В случае, если такой мета-триггер существует, то он создаёт триггер для новой метрики.

Агрегатор триггеров

Агрегатор триггеров позволяет выполнять логические операции над множеством триггеров, что позволяет реализовать более сложные схемы обработки и проверки данных.

В качестве примера на рисунках 1.15 и 1.16 изображены примеры агрегации двух триггеров. Рисунок 1.15 — каждый триггер проверяет значе-

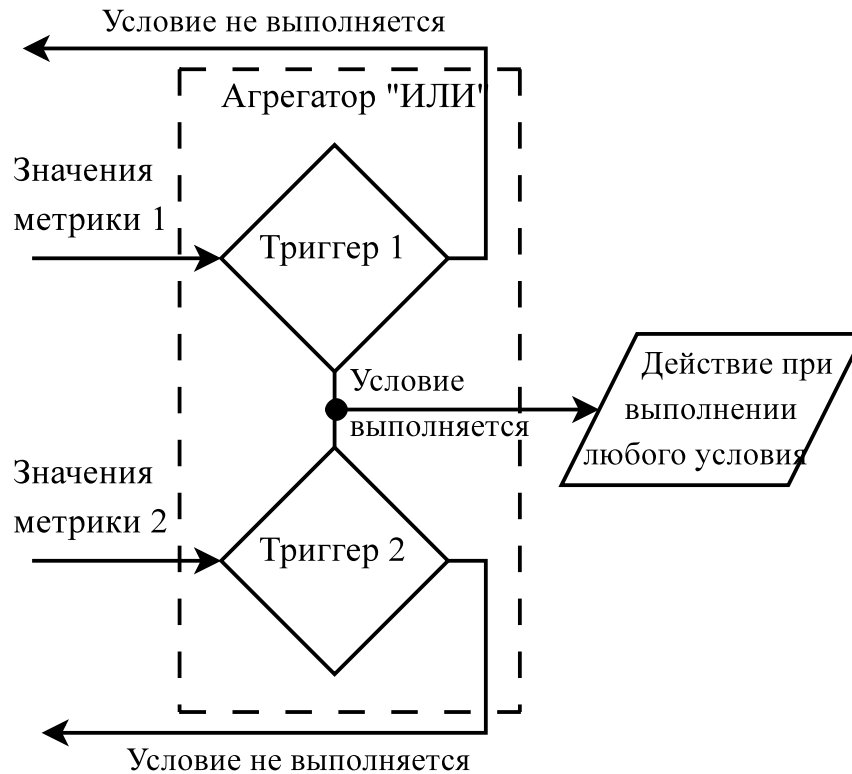


Рис. 1.15: Агрегатор триггеров «ИЛИ»

ние своей метрики, а действие выполняется агрегатором, который реализует логику «ИЛИ». Рисунок 1.16 — оба триггера проверяют значения одной и той же метрики, и агрегатор исполняет действия в случае срабатывания обоих триггеров. Такая схема обработки триггеров может возникнуть, если необходимо определять одновременное превышение значений двух метрик (например, температуры и использования центрального процессора).

Система мониторинга гарантирует, что оба триггера будут проверять значения метрики для одного и того же временного интервала. Таким образом, результат работы агрегатора триггеров всегда корректен.

Пользователь системы может реализовать свои объекты, использующие более сложную логику, если для его задачи недостаточно базовых логических элементов. Агрегаторы позволяют значительно эффективнее использовать ресурсы, исключив при этом излишние действия, единственной задачей которых является передача значения метрики следующему в последовательности триггеру.

Проверку условий триггеров, объединённых агрегатором, можно проводить двумя способами: через равные промежутки времени и при поступ-

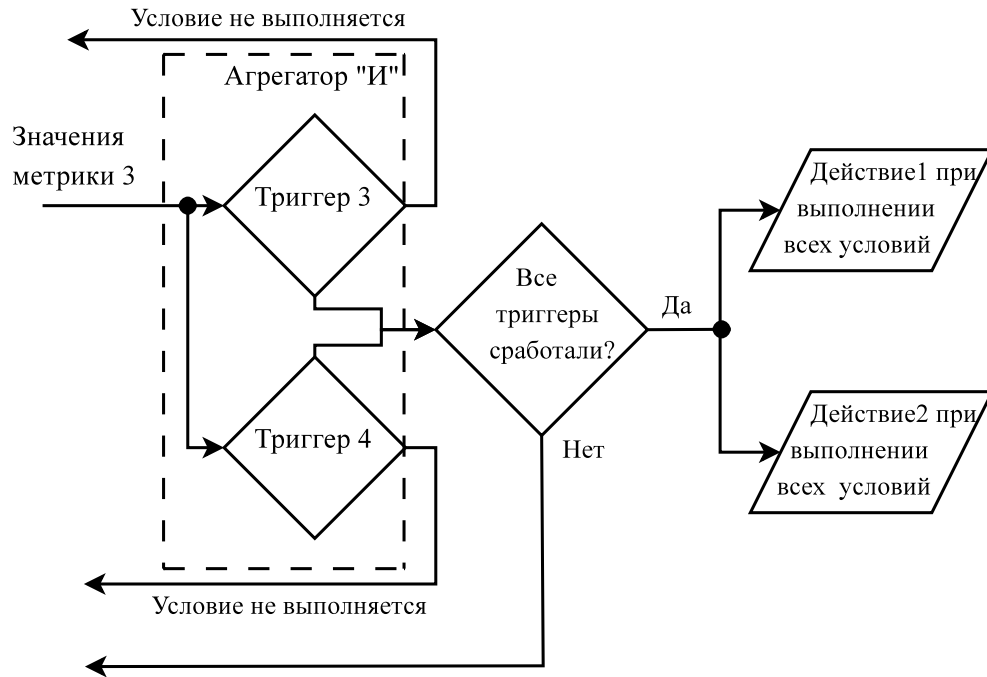


Рис. 1.16: Агрегатор триггеров «И»

лении данных в любую из проверяемых триггером метрик.

При проверке условий через равные промежутки времени потребляется сравнительно много вычислительных ресурсов. При этом не исключена избыточная обработка метрики несколько раз подряд, если новые данные в неё ещё не успели поступить.

Более выгодным с точки зрения производительности является подход, при котором условия триггера проверяются только при поступлении нового значения в метрику. Однако это порождает неопределённость в работе агрегаторов, если одна из метрик, проверяемая его триггерами, ещё не содержит новых данных, а остальные уже получили их. В реализации системы мониторинга Grate (см. вторую главу работы) агрегатору можно указать, как действовать в подобном случае: проверять значения лишь тогда, когда все метрики будут обладать необходимыми данными, или полагать, что у метрик с необновлёнными данными значения остались неизменными. По умолчанию выбирается второй вариант, как более быстро реагирующий на изменения в контролируемых метриках.

Действие

Эти элементы архитектуры используются для реализации обратного отклика системы на какие-либо события в контролируемой вычислительной системе. Действие может быть сколь угодно сложным, хотя для практического применения в большинстве случаев достаточно действий по отправке электронной почты или выполнению системных команд операционной системы, применяемых для управления вычислительным процессом (например, снятие задачи с выполнения).

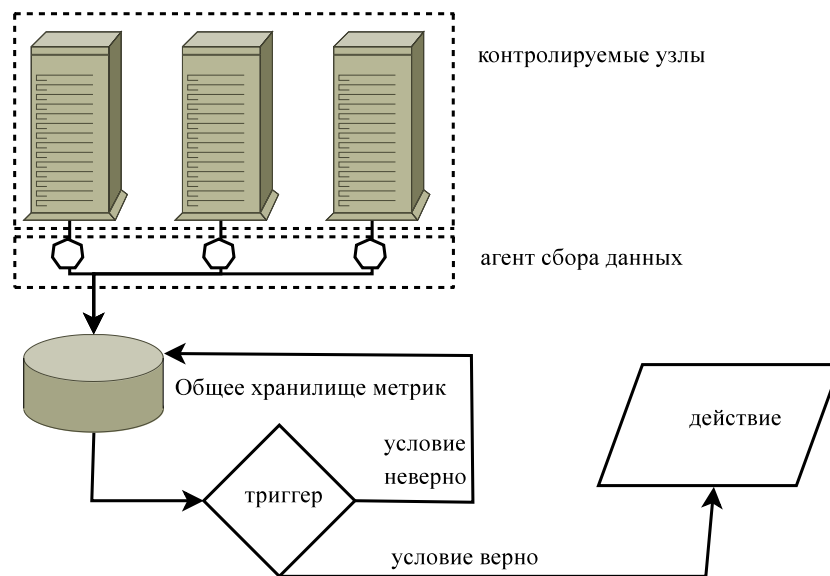


Рис. 1.17: Место объекта-действия в общей схеме обработки данных мониторинга

Для создания своего действия достаточно реализовать соответствующий интерфейс (см. детали реализации во второй главе работы) и определить соответствующие триггеры для проверки значений метрик в центре сбора данных. На рисунке 1.17 схематично отображена взаимосвязь действия и триггера в разработанной СМ.

Событие

Использование лишь триггеров и действий приводит к усложнению схемы обработки каких-либо внештатных ситуаций клиентскими приложениями. Возникает необходимость в таком механизме, который бы позволил разрабатывать ПО обработки подобных ситуаций на различных языках программирования и обладал возможностью динамической установки

фильтров (т. е. приложение пользователя могло бы присоединиться к обработке событий в процессе работы, а не в момент запуска системы мониторинга (рис. 1.7). Подробнее о практической реализации этого механизма рассказывается во второй главе.

Событием в описываемой архитектуре называется переход контролируемого ВК из одного состояния в другое (например, выход из строя одного из вычислительных узлов). В предлагаемой архитектуре предлагается использовать механизм подписки и передавать уведомления о событиях через ретрансляторы событий.

Ретранслятор событий — это служба СМ, позволяющая пересылать событие от источника события к одному или нескольким обработчикам. Использование ретрансляторов позволяет приложению пользователя регистрироваться на одном из ЦСД и получать уведомления обо всех событиях, независимо от того, на каком узле они произошли. Это также упрощает процесс передачи соответствующей информации от вычислительных узлов, которые отправляют событие на ретранслятор, не занимаясь маршрутизацией к заинтересованному в его обработке приложению.

Информацию о событии можно представить в различных форматах, но для большего удобства предлагается использовать являющийся открытым стандартом формат XML, что облегчает переносимость данных между различными архитектурами и обработку их сторонним программным обеспечением.

1.6. Выводы по первой главе

В первой главе сформулированы требования к архитектуре программного комплекса для мониторинга вычислительных кластеров, введены основные термины и приведено описание элементов разработанной архитектуры СМ и принципов их взаимодействия.

1. Уровни архитектуры слабо связаны между собой, взаимодействуя друг с другом через строго определённые интерфейсы. Это позволяет элементам более высоких уровней работать единообразно с нижними.

2. Значимыми и наиболее важными элементами предложенной архитектуры являются метрики, триггеры и события. Источники данных предоставляют информация о значениях контролируемых величин в ЦСД, которые выполняют предварительную обработку данных, проверку выполнения накладываемых на значения метрик условий, генерацию уведомлений о событиях.
3. Механизмы подписки и уведомлений о событиях предоставляют широкие возможности для организации взаимодействия стороннего ПО. Взаимодействие выполняется не только на основе доступа к значениям контролируемых величин, но и доступа к уведомлениям о событиях. В том числе, разнообразное ПО может взаимодействовать между собой, используя СМ как транспорт уведомлений о событиях.

Глава 2. Реализация системы мониторинга на языке программирования Java

В этой главе обсуждаются особенности структуры модулей и интерфейсов программной реализации, которые необходимы для интеграции СМ в более сложные программные комплексы.

СМ разрабатываются на различных языках и с использованием различных библиотек С [37], Java [43, 49, 82], TCL/TK [88]. Часто используются несколько языков в рамках одной системы, например в СМ Alert и клиент, и сервер написаны на языке С, однако часть скриптов, относящихся к web и архивированию данных написаны на языке perl [66].

Нередко ВК состоят из комплектующих от различных производителей или различных по своим характеристикам компонент. Поэтому в такой неоднородной среде важно достичь переносимости программ, что являлось одним из важных факторов при выборе языка программирования Java. Он является простым, объектно-ориентированным, универсальным, независимым от архитектуры, переносимым, высокопроизводительным, многопоточковым и динамичным языком программирования [79]. Помимо этого, Java предоставляет удобный интерфейс к сокетам [69] и широко используется при разработке применяемых в Интернет приложений и бизнес-приложений в сетях Интранет.

Язык программирования Java обладает следующими достоинствами.

- Простота синтаксиса и удобство использования стандартной библиотеки классов (в частности, Java Foundation Classes). Программистам, знакомым с языком С/С++, легко перейти к синтаксису Java.
- Надёжность. Позволяет исключить множество ошибок, связанных с неверным выделением памяти, обращением за пределы массивов и аналогичными ошибками.

- Безопасность. Виртуальная машина Java способна предотвратить множество ситуаций, опасных с точки зрения нарушения целостности операционной системы.
- Платформенная независимость. Байт-код для Java virtual machine (JVM) может исполняться на множестве операционных систем и аппаратных платформ.
- Возможность использования в web-среде.

При выборе языка программирования также учитывались и его недостатки.

- Невысокая производительность в некоторых задачах. В настоящее время, платформа Java не может достичь той пиковой производительности, на которую способен машинно-зависимый код. Однако, в среднем, производительность отличается незначительно¹.
- Объём дополнительных библиотек. Ядро виртуальной машины и стандартная библиотека занимают менее 200 Кб [30]. Но набор удобных для прикладного программиста классов, в т. ч. для поддержки графического пользовательского интерфейса, занимает более 20 Мб. Следует также отметить, что некоторые операционные системы не содержат по умолчанию виртуальной машины либо содержат свою собственную реализацию, которая не всегда соответствует спецификации Java.
- Невозможность использовать для доступа к ряду системных вызовов, поскольку в целях безопасности Java изолирует виртуальную машину от доступа к функциям ОС, что затрудняет создание сенсоров.

Указанные выше недостатки языка программирования Java не оказывают существенного влияния на производительность в задачах мониторинга, что подтверждают результаты применения разработанной CM Grate, приведённые в третьей главе.

¹<http://shootout.alioth.debian.org/>

Программный комплекс Grate, исходный код стабильной версии которого передан в сетевой ресурс фонда открытого программного обеспечения SourceForge², представляет собой библиотеку классов языка Java, предоставляющую базовый набор возможностей, аналогичных таким широко распространённым системам как Ganglia³, Nagios⁴ и др. Это было сделано с целью упрощения совместного использования различных программных продуктов. Вместе с тем, при минимальных изменениях можно использовать разработанное ПО совместно с более специализированными СМ, а также облегчить интеграцию различных СМ друг с другом.

ПК Grate состоит из нескольких независимо функционирующих модулей, написанных на языке Java: grated, grape (Java-апплет, выполняющийся в среде web-браузера), grate-client (приложение для простой визуализации и контроля данных). Модули используют общую библиотеку интерфейсов и классов, позволяющую им работать с метриками, триггерами и прочими элементами разработанной архитектуры СМ.

Для проведения различных численных экспериментов и внедрения СМ на кластере ВЦ ДВО РАН использовалась реализация обсуждаемой архитектуры на языке Java, однако её можно реализовать и на другом высокоуровневом языке программирования.

2.1. Соглашения о наименовании классов, методов и членов класса

Все классы, разработанные в рамках этого проекта и доступные для интеграции в сторонние проекты, именуются в соответствии со следующими правилами.

1. Первый символ в названии класса — «G», например, GFrame. Это позволяет отличать классы и интерфейсы проекта Grate от поставляемых со стандартной библиотекой Java.
2. Каждое смысловое слово, составляющее название класса начинается с большой буквы: GConnectionFile. Наименования классов выбраны

²<http://sourceforge.net>

³<http://ganglia.info>

⁴<http://nagios.org>

таким образом, чтобы реализуемая им функциональность была очевидна и интуитивно понятна.

3. Класс сохраняет в составе названия наименование интерфейса, который он реализует. Например, класс `GFrameXML` реализует интерфейс `GFrame`.
4. Методы классов, в соответствии с традициями языка Java начинаются с маленькой буквы (например: метод `connect()` класса `GConnectionInet`). То же самое справедливо и для членов класса. Методы и члена классов также удовлетворяют приведённому выше пункту 2 для всех слов, кроме первого.
5. Использование символа подчёркивания за некоторыми исключениями, допустимо лишь для внутренних переменных класса.

Большинство объектов, рассмотренных в предыдущей главе, задаётся в виде интерфейсов (например, `GFrame`). На базе интерфейсов реализуется абстрактный класс (например, `GFrameBase`), который в свою очередь расширяется классом, используемым прикладным программистом в своём приложении.

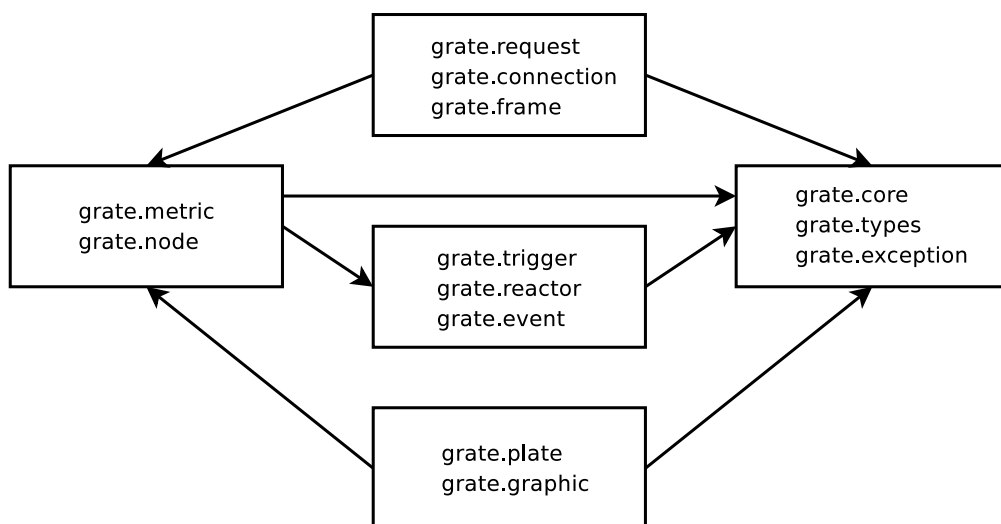


Рис. 2.1: Взаимосвязь пакетов программного комплекса

Классы, реализованные в процессе работы над проектом, располагаются в определённых пакетах. Пакеты условно образуют группы, наиболее

интенсивно взаимодействующих между собой пакетов. На рисунке 2.1 приведена схематичная взаимосвязь модулей СМ Grate, где стрелка указывает на зависимую группу пакетов.

Ниже приведено краткое описание пакетов.

`grate.core` — этот пакет содержит основные системные классы, которые редко используются напрямую пользователем.

`grate.exception` — в этом пакете определены исключения, которые могут генерироваться во время работы с набором классов.

`grate.connection` — содержит набор классов для подключения к источникам данных.

`grate.frame` — содержит реализации интерфейса кадра.

`grate.metric` — содержит реализации метрик и агрегаторов метрик.

`grate.plate` — содержит реализации графических областей.

`grate.graphic` — содержит различные виды графиков, используемые классом `GPlateGraphic`.

`grate.node` — содержит реализацию интерфейса узла.

`grate.types` — содержит определения основных типов данных, которые могут храниться в метриках.

`grate.request` — содержит определения классов, необходимые для формирования запроса на получения данных.

`grate.trigger` — определяет различные триггеры, срабатывающие при определяемых конкретным классом условиях. В этом же пакете реализованы агрегаторы триггеров.

`grate.reactor` — включает набор классов, реализующих некое действие при срабатывании триггера.

`grate.output` — обеспечивает возможность экспорта данных в предоставляемый выходной поток.

`grate.event` — поддержка событий, их генерации, обработки и передачи.

В дальнейших разделах этой главы мы рассмотрим основные классы и интерфейсы, важные для понимания реализации архитектуры СМ. Несущественные подробности, в т. ч. определения классов, их методов, интерфейсов и др. будут пропущены. С более полным их описанием можно

ознакомиться в Приложении А и исходном коде Grate, доступном на условиях лицензии GPL.

Исходные тексты программ, приводимые в данной работе, оформляются следующим образом:

- ключевые слова языка программирования Java **выделяются полужирным начертанием**;
- имена классов, переменных и методов класса пишутся без применения дополнительных стилей.

Например:

```
GConnectionBase connection = new GConnectionInet();
```

2.2. Пакеты поддержки сбора данных

В этом разделе приведено описание пакетов, содержащих классы работы с метриками, формирования иерархической структуры данных, поддержки соединений и т. п.

Ядро системы мониторинга (пакет `grate.core`)

Данный пакет содержит определение и реализацию некоторых интерфейсов, которые широко используются классами других пакетов.

Одним из них является простой вспомогательный класс `G`, задача которого обеспечить доступ к статическому полю `theWorld` другим объектам (метрикам, узлам и прочим логическим сущностям в процессе работы приложения). Другим вспомогательным классом является класс `GPath`, который используется при поиске узлов и метрик в иерархической структуре данных системы мониторинга.

```
public class G {
    public static void init(GWorld newWorld) {
        theWorld = newWorld;
    }
    public static GWorld theWorld = null;
```

}

При добавлении новых значений в метрику важно поддерживать непротиворечивость данных по времени. Например, недопустимо помещать более поздние измерения до более ранних. Поскольку формат хранения данных в метрике не определяется, то возможна ситуация, когда в метрике хранятся сами значения, без временной отметки. Чтобы избежать ошибок, в разработанной СМ полагается, что данные о всех интересующих нас метриках узла приходят одновременно, что удобно для практической реализации в подавляющем большинстве случаев. Вывод о том, стоит ли принять новые данные или игнорировать их принимает класс, реализующий интерфейс `GTimeline`. Помимо этого, `GTimeline` определяет интерфейс для таких функций как: поиск индекса по указанному времени; времени для указанного индекса; минимальный индекс и прочие операции.

Единственный класс, который по умолчанию реализует обсуждаемый интерфейс — это `GTimelineBase`, который в данном случае не является абстрактным. Объекты этого класса организуют работу исходя из того, что данные в метрику должны вноситься последовательно по времени. Данные, не удовлетворяющие этому условию, игнорируются.

Последний интерфейс, определённый в этом пакете — `GWorld`, в задачи которого входят административные функции и централизация операций по созданию метрик, узлов и соединений. Далее в диссертации будут приведены два приложения, в которых этот интерфейс реализован.

Работа с соединениями (пакет `grate.connection`)

Этот пакет содержит реализации интерфейса соединения.

Соединение обеспечивает доступ к данным на физическом уровне. Суть этого интерфейса сводится к предоставлению потока данных (объект, реализующий интерфейс потоков ввода-вывода стандартной библиотеки языка Java), который будет обрабатываться высокоуровневыми классами.

После того как соединение установлено с сенсором, приложение должно открыть сеанс получения данных, вызвав метод `connect()`. В результате

будет возвращён объект, уникальным образом идентифицирующий сеанс в рамках соединения. Этот объект используется в дальнейшей работе с сеансом. Конкретный класс, реализующий данный интерфейс, сам определяет наиболее удобный класс для представления объекта идентификации сеанса.

Класс `GConnectionBase` реализует интерфейс `GConnection` с целью определить статический метод `getDescription()`, возвращающий краткое описание назначения данного типа соединения.

Приложению рекомендуется использовать в работе классы, которые не реализуют напрямую данный интерфейс, а наследуют функционал `GConnectionBase`. Хотя пользователь СМ может реализовать интерфейс `GConnection`, но при этом теряется универсальность и взаимозаменяемость классов при работе с клиентским приложением. В состав программного комплекса по умолчанию входит три класса, расширяющие абстрактный класс `GConnectionBase`: `GConnectionFile`, `GConnectionInet`, `GConnectionMCast`.

Первый организует доступ к файлу, расположенному на доступных файловых системах; второй посредством библиотеки сокетов по протоколу ТСР организует доступ к данным, находящимся на удалённых узлах; третий организует подключение к группе мультивещания. Последний класс может использоваться во внутренних локальных вычислительных сетях для сбора данных аналогично `gmetad`.

Последовательность действий при работе с классами соединений, приведена ниже.

Предварительная подготовка.

```
GConnectionBase connection = new GConnectionInet();
connection.tune();
```

Запрос на получение данных.

```
Object sessionObject = connection.connect();
OutputStream outputStream =
    connection.getOutputStream(sessionObject);
InputStream inputStream =
```

```
connection.getInputStream(sessionObject);
```

Завершение сеанса.

```
connection.disconnect(sessionObject);
```

Помимо приведённых выше классов, в данном пакете находится `GSentinelBase`, назначение которого состоит в периодической инициализации фонового процесса получения данных. Объекты этого класса создают фоновый поток выполнения, который активен только в процессе получения данных, а в остальных случаях находится в приостановленном состоянии. При инициализации объекту передаются класс соединения (производитель данных) и класс кадра (потребитель данных). На каждой итерации `GSentinelBase` обеспечивает установку сеанса с источником данных посредством соединения, создание объекта потребителя данных, обработку полученных данных и уведомление глобального статического объекта, реализующего интерфейс `GWorld`. `GSentinelBase` реализует интерфейс `GSentinel`, метод `process()` которого выполняет указанные выше действия, а оставшиеся методы используются для управления работой потока выполнения.

Реализации интерфейсов кадров (пакет `grate.frame`)

Пакет содержит реализации интерфейса `GFrame`, который задаёт базовую функциональность кадра. Кадр рассматривается как законченный единовременный (в смысле получения данных) набор значений в определённом формате. Приложение устанавливает поток данных для кадра и вызывает одну из функций расшифровки данных кадра. Рекомендуемой функцией является `parse()` с входным параметром типа `GNode`.

Типичный код работы с кадром приведён ниже:

```
InputStream ins = connection.getInputStream(obj);
frame.setInputStream(ins);
if (frame.isDataValid()) {
    frame.parse(G.theWorld.getRoot());
    G.theWorld.commit();
}
```

Прикладному программисту рекомендуется расширять класс `GFrameBase`, который реализует дополнительные два метода:

`getDescription()` — получение краткой информации о типе кадра.

`setInputStream()` — установка соединения.

В предлагаемой реализации несколько классов кадров являются наследниками `GFrameBase`. Например, два из них реализуют обработку данных в формате XML, поставляемом системой мониторинга Ganglia: `GFrameGangliaXML` и `GFrameGangliaXML2`. Различие между ними состоит в способе обработке: первый использует DOM-анализатор XML-документа и поэтому требует несколько больше памяти, второй использует SAX анализатор, но не позволяет работать с полным деревом данных, если это необходимо.

Класс `GFrameGangliaXDR` осуществляет обработку данных Ganglia переданных в формате XDR в соответствии с описанием данных в `protocol.x` [63].

Также определён вспомогательный класс-обёртка `GFrameAuto`, который пытается определить тип кадра на основе переданных ему данных, вызывая метод `isValid()` для различных классов кадров. Если ни один из вызовов не был успешным, кадр игнорируется. Если какой-то из объектов успешно определил формат переданных данных как необходимый ему, `GFrameAuto` делегирует все функции по обработке кадра данному объекту.

Доступ к значениям метрик (пакет `grate.metric`)

Данный пакет определяет метрики, их типы и различные реализации. Все классы, входящие в этот пакет, реализуют интерфейс `GMetric`.

Метрика является основополагающей единицей архитектуры проекта, что отразилось на числе методов в данном интерфейсе. При разработке классов, реализующих `GMetric`, необходимо добиться максимально эффективного доступа к данным с учётом особенностей конкретного приложения, которое использует программный комплекс Grate. В связи с этим было разработано значительное число классов метрик, удобных в конкретных ситуациях и позволяющих достичь максимально эффективного использования памяти и вычислительных мощностей.

К основным классам, использующим данный интерфейс, относятся:

`GMetricBase` — абстрактный класс, реализующий некоторые простые операции, общие для широкого набора классов метрик (получение типа данных, последнего помещённого значения и прочее). В частности, в нём определены следующие полезные функции:

- `init()` — вызывается при начальной инициализации объекта метрик производных классов;
- `placeStringData()` — вспомогательная функция, упрощающая производным классам работу так, что им достаточно реализовать всего лишь функцию `placeData()`, которая вызывается из этой функции.;
- `updateMinMax()` — используется для оптимизации поиска минимального и максимального значения;
- `getEffectiveMetric()` — необходима для реализации механизма динамической смены типа метрики, см. класс `GMetricShell`.

`GMetricStatic` — используется для хранения единственного постоянного значения (например: метрика, содержащая имя узла, или максимальный объём физической памяти, установленный на вычислительном узле). При попытке добавления значения, отличающегося от ранее добавленного, генерируется исключение `GEfficiencyLackException`.

`GMetricShort` — используется для хранения единственного значения, которое, тем не менее, может быть различным в различный момент времени.

`GMetricCommon` — используется для хранения последовательных значений, полученных во время работы `Grate`.

Помимо классов метрик существуют классы-обёртки, которые предоставляют доступ к вышеперечисленным метрикам, однако при этом выполняют дополнительные действия, позволяющие более эффективно организовать доступ к данным или расширить функциональность. В проект входит класс `GMetricShell`, задача которого состоит в эффективном доступе к данным за счёт кэширования. Этот класс является обёрткой для классов,

производных от `GMetricBase`, и выполняет посреднические функции между пользователем и скрытым классом, которому делегирует выполнение функций.

Этот класс широко использует функцию `placeDataEffective()` для хранения данных в скрытом классе. Если же при вызове конструктора этого класса во время инициализации явным образом не указано иное, то создаётся скрытый объект класса `GMetricStatic`. Если при помещении данных в этот объект происходит исключение `GEfficiencyLackException`, то `GMetricShell` вызывает функцию `getEffectiveMetric()`, которая возвращает новый класс, способный более эффективно хранить данные либо значение `null`, в случае когда объект метрики не может определить такой класс. Последняя ситуация приводит к исключению `GAddDataFaultException`.

`GMetricShell` кэширует данные во время операции получения данных по индексу и по времени, что существенно снижает расход вычислительных мощностей во время работы системы, поскольку обычно данные метрик хранятся в динамических структурах, чаще всего оптимизированных для последовательного доступа, а не случайного.

В `grate.metric` описан также и интерфейс `GMetricCacheman`, который используется метриками для сброса значений, хранящихся длительное время из оперативной памяти в хранилище данных, зависимое от реализации (базу данных, файл и др.). Этот интерфейс позволяет также получить ранее сохранённые значения.

В разработанном комплексе по умолчанию используется класс `GMetricCachemanEater`, который является фиктивным хранилищем, т. е. он корректно реализует указанный интерфейс (хотя значения метрик на самом деле нигде не хранит), получить «сохранённые» с его помощью данные невозможно.

Доступ к хранилищу `JRobin`⁵, аналогичному по структуре базам `RRD`⁶ осуществляется с помощью класса `GMetricCachemanJRobin`.

⁵<http://www.jrobin.org/>

⁶<http://www.rrdtool.com/>

Поддержка иерархической структуры сбора данных (пакет `grate.node`)

В этом пакете представлен интерфейс `GNode`, представляющий узел иерархии, формируемой при работе программы. Узел предоставляет методы для доступа к своим метрикам и атрибутам, а также методам поиска метрик и узлов по названию и пути.

Все методы можно разделить на три группы: методы для работы с метриками (получение списка всех метрик, запрос объекта-метрики, добавление и обновление метрик в узле); методы для работы с иерархической структурой узлов; другие методы.

Интерфейс `GNode` реализован в классе `GNodeBase`. Атрибуты узла в нём сохраняются в виде метрики `GMetricShort`, что позволяет также проводить мониторинг атрибутов узла.

2.3. Пакеты для реализации механизма отклика системы

В этом разделе приводится описание пакетов, предоставляющих возможность использования триггеров, действий и уведомлений о событиях.

Работа с триггерами (пакет `grate.trigger`)

В этом пакете определены интерфейсы для работы с триггерами.

По умолчанию, каждая метрика вызывает триггер, сопоставленный ей при добавлении данных в метрику. Для метрики может быть не определён триггер. В таком случае никаких дополнительных действий не предпринимается.

В этом пакете определены классы: `GTriggerMore`, `GTriggerLess`, `GTriggerEqual`. Они проверяют соответственно значение метрики на условие превышения, принятие значения меньше заданной величины и равенства указанной величине. Все создаваемые триггеры могут быть либо сбрасываемыми (выполняют действие при каждом выполнении условий), либо переключаемыми (выполняют действие единожды при первом выполнении условий триггера и не повторяют действие до тех пор пока условия

выполнены). Тип триггера задаётся при его создании с помощью метода `setSignaled()`.

В этом пакете также определены интерфейс и класс мета-триггера с одним единственным методом `check()`, который выполняет проверку для метрики ряда условий, определённых для мета-триггера. При выполнении их создаётся триггер для данной метрики.

Триггеры может создавать как администратор ВК в конфигурационном файле СМ до её запуска, так и пользователи уже во время работы СМ через механизм подписки.

Выполнение действий (пакет `grate.reactor`)

В пакете `grate.reactor` определены возможные действия при выполнении условий триггеров. Все объекты-действия реализуют интерфейс `GReactor`.

В Grate входят три класса, реализующие указанный интерфейс: `GReactorBase`, `GReactorMail` и `GReactorNotify`, которые могут сообщать о переходе контролируемой системы в какое-либо состояние. Первый выводит информацию в ассоциированную с приложением консоль стандартного вывода (т.н. `stdout`), второй — по электронной почте, третий — через механизм событий. При этом все они позволяют указать шаблон сообщения, которое будет отправлено пользователю. В сообщении можно использовать макро-подстановки, например, `{METRIC_VALUE}` или `{NODE}`, которые будут заменены значением метрики, которая вызвала срабатывание триггера, и именем узла соответственно.

Поддержка событий и уведомлений (пакет `grate.event`)

Пакет `grate.event` отвечает за предоставление интерфейсов события и сервера уведомлений. Первый в программном коде представлен интерфейсом `GEvent`, который реализован базовым классом `GEventXML` (хранит все данные в формате XML).

Ниже приведено описание типичного события в разработанной СМ:

```
<event id = 123 timestamp = 46712631236 level = "warn" >
```

```

<system name = "network">
  <interface name = "eth0">
</system>
<src = "node2" />
<broadcast />
<str>Interface eth0 link down</str>
<action default = "ignore" />
<type value = "failure" />
</event>

```

Здесь представлена подсистема (категория), к которой относится событие, идентификационный номер, тип, источник и действие по умолчанию. Последнее — это одно из предопределённых действий (в приведённом примере — «игнорировать»), выполняющихся, если ни одно из приложений не подписано на уведомление о произошедшем событии.

Сервер уведомлений описывается интерфейсом GEventServer, который определяет как производится регистрация и уведомление о событии. Базовой реализацией является GEventServerUDP, осуществляющий посылку необходимых данных по протоколу UDP. При получении нового пакета клиентское ПО должно подтвердить его получение во избежание повторной передачи. Если событие должно быть обработано, но длительное время нет подтверждения от клиентского ПО, то сервер выполняет действие по умолчанию. GEventServerUDP позволяет сохранять события в метрике с типом GType_event, если она создана в ЦСД.

Для упрощения процесса общения клиентского ПО с сервером был также разработан класс GEventClientUDP, который берёт на себя рутинные операции по регистрации на ретрансляторе событий, получению информации от него и созданию новых объектов событий.

В авторской реализации подписка на ретрансляторе событий выполняется при помощи посылки пакета в формате XML по протоколу UDP. Этот протокол был выбран, поскольку не является ориентированным на соединения и поэтому менее затратен для производительности при большом количестве передаваемых пакетов.

Ниже приведён пример пакета с запросом подписки на уведомления о событиях:

```

<filter cookie="ac12h23b">
  <system name = "network" />

```

```

<subsystem name = "*" />
<src name = "*" />
<level name = "warn" />
</filter>

```

После получения запроса сервер создаёт фильтр для событий, относящихся к типу «события сети» (network), любого подтипа и от любого источника уведомлений с уровнем события «предупреждение» (warn) и выше. Затем клиенту высылается пакет содержащий такое же значение сессионного параметра (cookie), что и в оригинальном сообщении, а также статус подписки (подписка выполнена, подписка отклонена):

```

<reply cookie="ac12h23b">
  <status name = "success" />
</reply>

```

Отметим, что адрес отправителя запроса определяется при принятии датаграммы из сокета и не включён в описание фильтра. Сессионный параметр генерируется клиентом произвольно, он предназначен для различения ответов от различных серверов, если приложение клиента работает с несколькими ретрансляторами. Помимо уровня предупреждений поддерживаются также уровень ошибок – «error» и информационных сообщений – «notice». Эти уровни используются для более гибкой фильтрации сообщений, в которых заинтересовано приложение. По умолчанию в пакеты не включается схема документа (т.н. DTD определение), тем не менее такие схемы созданы и доступны для приложений, взаимодействующих с СМ.

В авторской реализации не используется авторизация при подписке, нет ограничений прав доступа приложений к конкретным событиям, поскольку необходимости в этом при использовании СМ для мониторинга ВК не возникает при должной настройке средств защиты ОС. При ориентации СМ на глобальные РВС следует использовать механизмы авторизации и шифрации данных, которые не применяются в описываемой реализации СМ Grate.

Описываемый пакет позволяет также устанавливать триггеры и либо посылать уведомления о выполнении условий триггера, либо выполнять известные СМ действия (класс, реализующий действие, должен находиться или в папке, где установлена СМ, или по одному из системных путей). Со-

здание триггеров выполняется после установки параметров мета-триггера. Процесс установки происходит аналогично подписке на уведомления: посылается пакет с описанием параметров мета-триггера на ЦСД.

Например:

```
<crigger cookie="cc173hb8">
  <metric template = "cluster1/*/CPU_Temperature" />
  <trigger class = "GTriggerMore">
    <option name = "value" value="80.0" />
  </trigger>
  <reactor class = "GReactorMail" />
    <option name = "body" value = "Overheat {NODE}" />
    <option name = "rcpt" value = "cluster-reports@as.khb.ru" />
  </reactor>
  <signaled />
</crigger>
```

В приведённом примере указывается, что для всех метрик температуры центрального процессора, относящихся к узлу ИСД cluster1, необходимо установить триггеры, срабатывающие при превышении определённого значения (класс GTriggerMore). При выполнении условий триггера выполняется отправка электронной почты (GReactorMail) на адрес cluster-reports@as.khb.ru с указанным телом сообщения. Параметр signaled означает, что триггер является переключаемым, по умолчанию все триггеры создаются сбрасываемыми.

Сервер СМ подтверждает успех или не успех установки триггеров. Отметим, что для установки триггера необходимо знать имена классов, ответственные за проверку условий и выполнение конкретного действия. Это не всегда является возможным и удобным для пользователя.

2.4. Вспомогательные пакеты

К вспомогательными пакетам относятся определения возможных типов значений метрик, средства вывода графики и классы поддержки исключений.

Поддержка базовых типов данных (пакет `grate.types`)

В этом пакете определяется интерфейс для хранения значений метрик различных типов, а также набор базовых типов. При необходимости этот набор может быть расширен сторонними разработчиками под свои нужды. При инициализации объекта метрики происходит поиск класса, название которого состоит из префикса `GType_` и указанного типа. Например, `GType_uint32` для типа `uint32`. Если класс с указанным именем не найден, создаётся объект класса с неопределённым типом `GType_unknown`, для которого ряд операции не определён. Но, если существует график, который зарегистрировался для вывода данных значений, то полноценное отображение значений возможно. Для этих значений выполняются все доступные операции в рамках метрик.

Одним из важных методов в интерфейсе `GType` является метод установки значения объекта `setValue()`. Методу передаётся строковое представление, которое метод должен сам преобразовать к нужному ему виду и принять данное значение. Естественно, что не всякое строковое значение может быть приведено к нужному типу. В таком случае поведение класса не определено. Строго говоря, такая ситуация в разработанной архитектуре не допускается на этапе помещения значения в метрику (в этот момент проводится проверка на тип значения).

Операция сравнения двух значений также определена в классах с интерфейсом `GType`. Если выполнение сравнения невозможно, поведение типа не определено. Базовый набор классов `Grate` не проводит проверку на соответствие сравниваемых классов в целях повышения быстродействия системы. Т. е. допустима операция сравнения строки с целочисленным значением. Данный метод использует функции преобразования значения для получения сравнимых величин необходимого типа. Единственное исключение — тип `GType_unknown`. Если сравниваемый объект является экземпляром того же типа `GType_unknown`, то возвращается значение равенства; иначе — любое другое.

Помимо этого многие классы, реализующие интерфейс `GType`, добавляют метод `setValue()` с параметром, отличным от типа `String`, т. е. тем

параметром, который наиболее точно представляет значение данного типа в языке Java.

Ниже приведены все классы базовых типов, созданные в рамках работы над Grate.

`GType_unknown` — класс для значений с неопределённым типом. Помимо реализации основных функций, класс содержит полезный метод `getRealBytes()` который читает из потока в указанный массив байты как знаковые целые (`int`), что удобно — в частности — для работы с форматом XDR.

`GType_timestamp` — временная отметка. К особенностям этого класса можно отнести то, что приведение его к строке даёт время в формате ЧЧ:ММ.

`GType_boolean` — логический тип данных.

`GType_float`, `GType_double` — числа с плавающей точкой одинарной и удвоенной точности.

`GType_int32`, `GType_int64` — знаковые целые.

`GType_uint16`, `GType_uint32` — беззнаковые целые.

`GType_string` — строка.

Последние пять классов содержат статические методы `getValueFromXDR()` и `putValueToXDR()`, которые используются остальными классами этого интерфейса для сохранения в поток и чтения данных из потока в формате XDR.

Дополнительно определён тип `GType_event`, который используется для хранения информации о событии. Этот класс расширяет `GType_string` и предоставляет методы для получения категории события.

Поддержка исключений (пакет `grate.exception`)

Этот пакет расширяет класс `Exception` стандартной библиотеки Java до класса `GException`, который является базовым для остальных классов исключений, порождаемых в процессе работы Grate.

Механизм исключений является одним из подходов к обработке ошибок, при котором в потоке выполнения программы генерируется состояние

исключительной ситуации (исключение), управление передаётся на обработку тому блоку программы, который встретится первым при поиске подходящих обработчиков исключений.

Помимо собственно ошибочных ситуаций обсуждаемая система мониторинга использует исключения для смены типа метрики. Такой подход позволяет эффективно работать с метриками, используя по умолчанию самые простые и быстродействующие типы, переходя к использованию более сложных типов только при необходимости. Подробнее об этом подходе сказано в описании класса `GMetricShell`.

Определены следующие исключения:

- `GAddDataFaultException` — генерируется в ситуации, когда невозможно добавить новое значение в объект класса метрики.
- `GEfficiencyLackException` — генерируется, когда невозможно добавить данные в метрику, поскольку хранение данных в данной метрике неэффективно.
- `GFailedInitException` — генерируется при ошибке инициализации любых объектов `Grate`.
- `GInvalidParamException` — генерируется методами классов при передаче неверных параметров, в случаях, если подобную ситуацию невозможно автоматически обработать иначе.

Представление графической информации (пакет `grate.plate`)

В этом пакете реализуются интерфейсы графической области, которые используются при построении клиентских приложений. Базовые возможности обеспечиваются классом `GPlateBase`.

`GPlateBase` добавляет возможность использовать операцию «перетаскивания» — взаимодействие с элементами управления в графическом интерфейсе пользователя, известное также как `drag and drop`. Для этого определены вспомогательные классы `GLocalTransferable`, реализующий интерфейс стандартной библиотеки Java — `Transferable`, и `GPlateDropListener`, реализующий `DropTargetListener`.

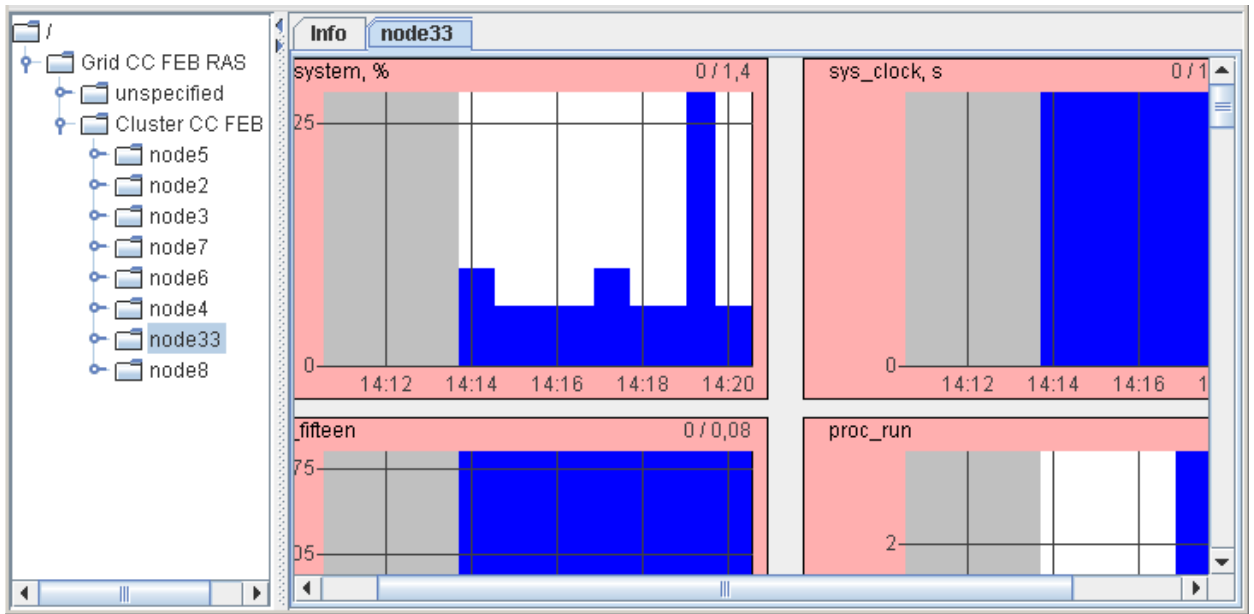


Рис. 2.2: Отображение значений метрик при помощи GPlateGraph

На основе GPlateBase в разработанных приложениях используются два класса: GPlateGraph и GPlateTable. Первый (рисунок 2.2) предназначен для отображения графиков и использует классы пакета `grate.graphic`, второй — для табличного представления данных с использованием компонента `JTable` пользовательского интерфейса `Swing` (рисунок 2.3).

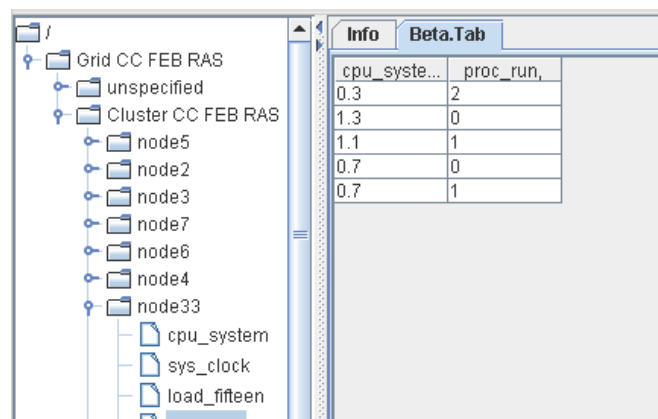


Рис. 2.3: Отображение значений метрик при помощи GPlateTable

К особенностям реализации GPlateGraph следует отнести эффективное использование вычислительных мощностей за счёт предварительного вывода данных в специально выделенную область памяти (т.н. внеэкранный блок памяти). Отображение результата на экране происходит только по завершении процесса формирования графического представления данных.

Поддержка графиков изменения значений метрик (пакет `grate.graphic`)

Содержит реализации интерфейса `GGraphic`, который используется классом `GGraphicPlate` для отображения значений метрик в виде различного типа графиков. Один экземпляр объекта `GGraphic` занимается визуализацией одной метрики.

Помимо собственно функции отображения `draw()`, интерфейс определяет методы, относящиеся к взаимному связыванию объектов графика и метрики. Графики могут быть:

- одномоментные — отображают единственное значение метрики;
- требующие всей рабочей области — используют всю экранную область объекта `GPlateGraph`;
- отображающие целочисленные, вещественные, строковые и неопределённые значения;
- другие, в том числе определённые пользователем.

На основе `GGraphicBase` создано два основных класса `GGraphicBar()` и `GGraphicLine()`, рисующие графики значений метрики в виде столбиков и линий соответственно. При необходимости создания нового вида графиков программисту необходимо реализовать интерфейс `GGraphic`.

Все классы совместно используют экземпляр класса `GGraphContext`, который хранит в себе характеристики рабочей области рисования (масштаб, линейные размеры и т. п.).

2.5. Сравнение разработанной системы мониторинга с аналогами

СМ Grate, согласно введённой в [68] классификации, является гибридной, поскольку большинство негативных процессов в вычислительных кластерах можно определить с использованием сенсоров, находящихся на конкретных узлах сети (как при прямом мониторинге). При этом агенты

мониторинга на узлах могут передавать данные в сеть широковещательно, а модуль сбора не обращаться напрямую к узлам (косвенный мониторинг). Схожим образом реализована, например СМ Ganglia, обеспечение совместимости с которой являлось одним из желаемых результатов при разработке трёхуровневой архитектуры.

Предлагаемая в работе архитектура позволяет создавать распределённые и иерархические СМ. Однако, в базовом варианте, наиболее применимом на вычислительных кластерах с сравнительно небольшим числом вычислительных ядер, иерархическая организация является наиболее удобным способом построения СМ.

СМ Grate использует жёстко заданный путь прохождения события (от источника на ретранслятор, затем к подписавшимся приложениям). Следует отметить, что механизм событий, предлагаемый в настоящей работе, при использовании выделенных ЦСД более масштабируем, чем используемые в СМ XMLBlaster и ISIS.

В созданной СМ данные передаются пользователю через механизм формирования отчётов. Отчёт формируется после запроса к конкретному ЦСД. В одном отчёте могут содержаться метрики из различных систем мониторинга, развёрнутых на нижних уровнях, а также данные аппаратных сенсоров.

Некоторые СМ также динамически формируют дерево метрик и позволяют добавлять и убавлять новые вычислительные узлы из системы мониторинга в процессе работы системы.

Ряд СМ также предоставляют набор API для одного или нескольких языков программирования, позволяя тем самым получить доступ к данным мониторинга из приложений пользователя. Использование подобного API значительно ускоряет разработку параллельных приложений за счёт возможностей профилирования используемых ресурсов [88]. Разработанная СМ предоставляет ряд классов и интерфейсов на языке программирования Java, позволяющих сравнительно легко интегрировать её в другие приложения. В разработанной СМ автор отказался от использования SOAP в силу сложности использования её в ряде языков программирования и невозможности организовать вещание событий базовыми стандартами

SOAP (которые требуют запроса для вызова процедуры и возврата значений), а также недостатка SOAP связанного с отсутствием самоописания данных. Между тем, многие функциональные возможности разработанной СМ потенциально можно получать с использованием SOAP.

Существенными преимуществами Grate по сравнению с другими СМ являются:

- поддержка триггеров или других вариантов проверки определяемых пользователем условий для значений метрик;
- реализация механизма событий;
- возможность построения децентрализованной СМ;
- относительно простое взаимодействие с другими СМ;
- унифицированный доступ к данным различных типов;
- независимость от ОС.

В частности, платформенная независимость от операционной системы достигается использованием языка программирования Java для ядра СМ, что также способствует лучшей интеграции с веб-средой. Поддержка триггеров и событий в системе реализована на достаточном для практического применения уровне. Использование унифицированного внутреннего формата данных для работы с метриками, возможность импорта данных сенсоров других СМ позволяют совместно использовать различные системы мониторинга.

Широко используемая СМ Ganglia предоставляет доступ к таким метрикам, как параметры использования программных и аппаратных ресурсов, а также определённые пользователем пары ключ-значение со всех узлов кластера. Ganglia реализована в виде двух отдельных модулей — демонов. Демон gmetad устанавливается на управляющий сервер. Через заданные промежутки времени он запрашивает необходимую информацию от узлов (где работает gmond) и сохраняет ее в свою базу данных. gmetad может взаимодействовать с другими такими же демонами, чем достигается иерархическая структура мониторинга. Организация потоков данных в

виде иерархии с общим корнем позволяет собирать всю информацию на выделенном узле и таким образом наблюдать за несколькими кластерами или PBC типа GRID.

Недоступность одного узла не влияет на общую работу, поскольку все узлы обладают информацией обо всём кластере, т. к. демоны gmond использует протокол UDP в режиме мультивещания (multicasting) для передачи данных метрик в пределах кластера.

Таким образом, в локальной сети вычислительного кластера демоны gmond, установленные на различные узлы, организуют одноранговую устойчивую систему, в которой каждый узел ожидает информации от «соседних» узлов, а не запрашивает её явно [63]. Этот подход не требует априорного знания о существующих вычислительных узлах, следовательно, gmond может адаптироваться даже к динамически изменяющейся вычислительной системе, используя подход soft-state [70].

gmond может взаимодействовать с gmetad посредством данных, передаваемых по протоколу TCP в формате XML. Это позволяет передавать данные по внешним сетям, в которых запрещено мультивещание.

Особенностью данной системы мониторинга является хранение данных в формате Round-Robin Database (RRD), который обычно применяют для часто меняющихся во времени данных, таких как загрузка сети, температура процессора и винчестера, загрузка процессора и т. д. в различных системах контроля потоков данных в ЛВС. Объём хранимых данных не увеличивается со временем, так как ячейки хранения используются циклически. Использование различных функций консолидации (или агрегации) данных позволяет охватывать большие интервалы времени без чрезмерного увеличения объема базы данных за счет снижения разрешающей способности для различных временных периодов (например, 60 значений за последний час; 24 значения за последний день и т. д.).

К недостаткам этой системы можно отнести невысокую разрешающую способность — большой временной интервал, через который ведется опрос узлов (обычно, раз в одну минуту). Отсутствует возможность использования триггеров событий, сложно организовать недревовидную структуру обмена данными.

CM Supermon [64] во многом аналогична системе Ganglia и Grate. На каждом вычислительном узле располагается агент, передающий данные по протоколу TCP. Данные собираются в результате последовательного опроса всех узлов по порядку. При этом в настройках системы мониторинга заранее определена информация обо всех узлах, которые необходимо контролировать. Важным недостатком является то, что в Supermon не ведётся архив данных измерений, делая невозможным наблюдение за тенденциями изменения метрик. Вследствие этого сенсоры системы требуют очень малых накладных расходов для работы, но CM ограничена в возможностях.

Supermon, как и ряд других систем мониторинга, требует $O(N)$ сетевых соединений, чтобы получить статистику от ВК, где N — число узлов кластера. CM Grate, в ряде случаев (в том числе, например, при использовании сенсоров Ganglia), требует всего одного соединения, поскольку в результате использования постоянных межузловых коммуникаций не требуется дополнительных обменов в момент запроса статистики. Подобное преимущество также свойственно распределённым системам мониторинга, например Argus⁷.

И Ganglia и Supermon используют рекурсивные языки (recursive language) для представления данных мониторинга — XML и S-выражения (S-expressions⁸) соответственно. Это позволяет организовывать контролируемые данные иерархически. Grate умеет экспортировать и импортировать данные мониторинга в различные форматы представления, основным из которых является XML в силу удобства его при отладке сторонних приложений. При иерархической организации поток данных от всех узлов к корню значительно вырастает, но при этом можно достичь эффективного контроля относительно большого количества метрик в кластере.

Несколько отличным образом организована система Parmon [43]. Основные компоненты данной CM — это parmon-server и parmon-client. Первый предоставляет доступ к данным мониторинга, второй является GUI клиентом, который не собирает все данные подобно gmetad, а запрашивает их лишь при необходимости.

⁷<http://argus.tcp4me.com>

⁸<http://people.csail.mit.edu/rivest/Sexp.txt>

Clumon⁹ — система мониторинга, разработанная в NSCA. Она основана на SGI Performance Co-Pilot и позволяет собирать данные с узлов, отслеживать задачи, помещённые в очереди планировщиков задач. Clumon является задание-ориентированной системой мониторинга, тесно интегрированной с PBS. Данная система мониторинга не позволяет производить контроль данных нескольких кластеров в виде единой системы. Другим недостатком является то, что используемый формат представления данных — таблицы SQL — затрудняет организацию иерархических структур сбора данных.

Metacomputing Directory Service (MDS) [49] используется в прикладном ПО организации распределённых вычислительных систем Globus Toolkit [51]. Данная служба была создана и используется в основном для описания ресурсов, а не для наблюдения за их потреблением в некотором промежутке времени. Поэтому структура данной службы в большей степени ориентирована на статичную информацию, такую как тип сетевых интерфейсов, рабочая частота процессора и т.п. MDS не хранит историю изменения метрик. Однако, в отличие от многих CM (и Grate в том числе), MDS поддерживает сложную модель проверки достоверности данных и прав доступа, основанную на сертификатах, что особенно важно для поддержки безопасной работы с распределёнными вычислительными системами через публичные сети.

CM dproc [32] работает на уровне ядра, предоставляя данные мониторинга посредством виртуальной файловой системы proc. Эта CM была разработана для применения на одном узле и не поддерживает мониторинг PBS и кластеров. Её аналогом является lmsensors¹⁰ и другие.

XMLBlaster поддерживает систему уведомлений, которые передаются в XML формате и направляются к адресату с использованием информации, заключённой в самом сообщении. В схожей системе Astrolab путь передачи уведомления также зависит от текущего состояния контролируемой среды. В нескольких работах [44, 80] показывается применимость «диффузионного распространения» событий при фильтрации их на пересылочных узлах с

⁹<http://clumon.ncsa.illinois.edu/doc-info.html>

¹⁰<http://www.lm-sensors.org/>

помощью простых правил.

CM Cougar [40] позволяет получать данные, посылая на сервер запрос в формате SQL. Сервер опрашивает все контролируемые узлы и на основе запроса формирует выборку, которую возвращает пользователю. Недостатком системы является её централизация и зависимость от выделенного сервера, принимающего запросы от клиентского приложения.

Automon является одной из наиболее простых CM. Используется в основном для целей мониторинга наличия компьютеров в вычислительной сети организации. В Automon отсутствует возможность создания триггеров, использования метрик, хранения значений за некоторый промежуток времени и т. п.

Система контроля кластера SCMS из проекта SMILE является одним из наборов ППП для Beowulf-кластеров [88]. Она позволяет выполнять параллельные команды на всех узлах, осуществлять менеджмент узлов и отображать информацию по всем узлам. Для получения данных используются shell-скрипты.

CM SyMon позволяет пользователям контролировать отдельную рабочую станцию. SyMon построена по принципам клиент-серверной технологии, чётко выделяя контролируемый узел и станцию мониторинга [78].

CM Alert состоит из одной или более контролируемых станций, которые функционируют независимо друг от друга [66]. Каждая рабочая станция может выполнять ряд действий, хранит свой файл отчёта об ошибках.

CM NSR (node status reporter) предоставляет стандартный механизм для измерения и доступа к состоянию кластеров рабочих станций [71]. Параллельные приложения могут обращаться к NSR через специальную утилиту или интерфейс NSR.

Ряд CM, также как Grate, поддерживает механизмы триггеров и событий. Например, архитектура JAMM [82] использует модель «производитель — потребитель», схожую с известными событийно-ориентированными службами, такими как служба событий CORBA (Event Service) [94]. Она поддерживает оба режима: запрос данных (query) и вещание (streaming). В CORBA понятие «канал событий» инкапсулировано в производителе, который ответственен за мультиплексирование событий и уведомление заинте-

ресованных объектов. JАММ публикует их в службе каталогов. Клиенты (потребители) ищут сенсоры в службе каталогов, а затем подписываются на необходимые данные через шлюз событий (производитель). Служба каталогов используется для публикации расположения (адрес узла ЛВС) всех сенсоров и их шлюзов. Это позволяет потребителю определить, какие сенсоры сейчас активны, и с каким шлюзом необходимо связаться, чтобы подписаться на данные сенсора. В разработанной СМ используется подход с регистрацией приложения на ретрансляторе событий, который является составной частью программного комплекса и не требует настройки дополнительных служб.

Другие системы мониторинга — такие, как, например, Nagios¹¹, позволяют производить обработку событий и поддерживают макроподстановки, которые можно использовать при обработке событий. События могут быть специфичными для узла, для службы, а также локальными и глобальными. Все обработчики событий указываются в конфигурационном файле до запуска системы. Подписка на события во время работы системы мониторинга невозможна. Nagios изначально создавалась для мониторинга сетевых служб с жёстко заданными состояниями (активен, неактивен, ошибка), поэтому добавление новых, определяемых пользователем событий, затруднено.

Схожим с JАММ образом построены такая система как Pablo [46], работающие с сенсорами приложений. В [91] также описана архитектура с службами, схожими с JАММ.

2.6. Выводы по второй главе

Во второй главе указаны особенности авторской программной реализации СМ на языке программирования Java, описывается структура модулей и интерфейсов, необходимые для взаимодействия СМ с другими программными комплексами.

Особенностями реализации являются:

1. Выбор языка программирования Java для реализации описанной в

¹¹<http://www.nagios.org>

первой главе архитектуры повышает устойчивость работы СМ, но вместе с тем ограничивает доступ к изолированным от JVM системным ресурсам. Источниками данных СМ Grate являются источники данных других СМ и машиннозависимое ПО.

2. СМ Grate предполагает обеспечение безопасности (авторизации, аутентификации, ограничения доступа к портам, используемым разработанной СМ) стандартными механизмами ОС. Это не является критически важным при использовании СМ в пределах одного ВК, но является недостатком при мониторинге глобальной РВС.
3. В связи с потенциальной неоднородностью клиентов, сделан выбор в сторону использования парадигмы REST с XML в качестве формата передачи сообщений, а не SOAP с семейством WS-* протоколов. Для уведомлений о событиях используется вещание от сервера клиентам без необходимости явного запроса (что затруднено при использовании SOAP). Простота описания события позволяет использовать простейшие утилиты обработки XML без необходимости реализации механизмов удалённого вызова процедур (RPC), что расширяет множество языков программирования, которые могут быть использованы для обработки событий.
4. Во этой главе проведено сравнение реализации разработанной архитектуры с существующими программными решениями. Показаны особенности и преимущества СМ Grate в сравнении с рядом других СМ.

Глава 3. Практическое применение разработанных подходов

В третьей главе описано применение построенного с использованием предлагаемых подходов кросс платформенного ПК Grate для решения ряда актуальных задач контроля ВК. В качестве источников данных использована система Ganglia 3.0.6, развёрнутая для мониторинга состояния вычислительного кластера ВЦ ДВО РАН [10], а также разработанные агенты сбора данных, в том числе утилита `grate_metric`.

Использование связки «триггер – действие» позволяет реализовать сложную схему обработки критических и штатных ситуаций, возникающих в процессе работы вычислительной системы, при условии возможности добавления своих объектов-действий в СМ. В частности, третий уровень рассматриваемой архитектуры может использовать срабатывание триггеров как входные параметры для простейшей экспертной системы, вырабатывающей заключение о необходимости последующего воздействия на вычислительный процесс.

Обычно система мониторинга должна просто уведомить каким-либо образом администратора или пользователя при выполнении некоторых условий. Тем не менее, задача автоматизации обработки внештатных ситуаций на ВК так же может быть решена с использованием СМ. Далее будет приведён подход к расширению возможностей установленной СМ для реализации штатного отключения вычислительных узлов при их значительном перегреве, что актуально для неспециализированных комплектующих, где подобная функция отсутствует.

СМ, разработанная в соответствии с приведённой в первой главе архитектурой, позволяет использовать модули для расширения своих возможностей. В качестве примера такого взаимодействия приведено использование модуля искусственных нейронных сетей для определения текущего

состояния вычислительного кластера на основе данных, поставляемых системой мониторинга. Обратный отклик модуля реализован через механизм сообщений.

Уровень приложений может использоваться и для более сложных управляющих операций над контролируемой системой. В качестве примера можно привести задачу повышения эффективности использования вычислительной системы. Ниже будет приведён пример решения подобной задачи для вычислительного кластера, узлы которого работают под управлением гипервизора Xen.

Необходимые для понимания работы соответствующих компонент сведения приведены в приложениях (в том числе: краткие сведения из теории ИНС; описание базовых принципов виртуализации).

3.1. Схема работы типичного программного комплекса, использующего разработанные классы

Ядро разработанного программного комплекса составляет класс Java, реализующий интерфейс GWorld. Экземпляр этого класса осуществляет контроль поступления данных и создание остальных важных для работы комплекса классов, инкапсулируя всю доступную приложению информацию от СМ. На рисунке 3.1 приведена общая архитектура приложения, использующая данный интерфейс. Клиентское приложение может работать, минуя данный класс.

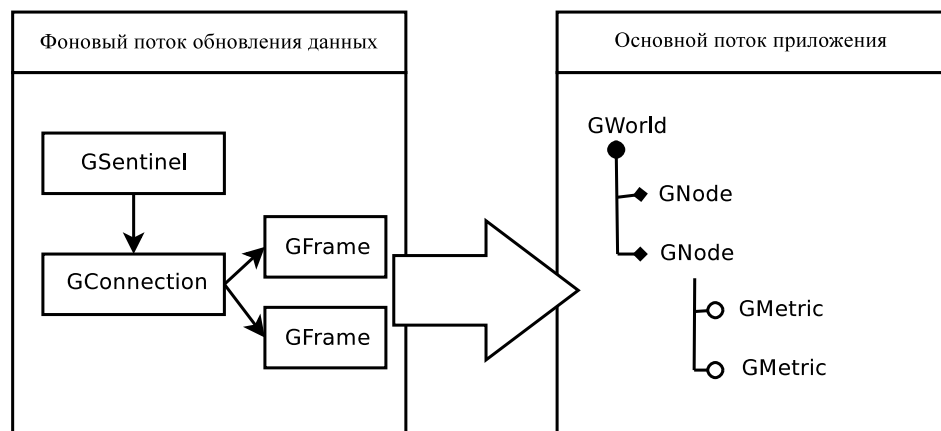


Рис. 3.1: Схема двухпоточного приложения, использующего нашу СМ

Вначале работы создаётся экземпляр необходимого класса, реализу-

ющего интерфейс GWorld. В зависимости от конкретного класса могут потребоваться вызовы дополнительных методов, однако обычно инициализация происходит следующим образом:

```
G.init(new GWorldTiny());
```

G является глобальным объектом, через статическое поле theWorld которого все созданные классы могут обращаться после его инициализации. Использование глобального объекта вместо передачи объекта в качестве параметра вызвано тем, что в типичном приложении создаётся единственный экземпляр GWorld, т. е. он является синглтоном [2], и поэтому нет необходимости в постоянной передаче его как параметра, что уменьшает использование памяти и упрощает определения методов классов.

Клиентская часть приложения вызывает метод createConnection() чтобы создать объект, реализующий интерфейс GSentinel. Этот объект при инициализации получает классы соединения и кадра.

В процессе работы GSentinel с указанной периодичностью создаёт новые сеансы, используя объект соединения и передавая полученный поток данных на обработку в объект кадра.

В объекте кадра происходит обработка поступающих данных и вызов, при необходимости, методов объекта theWorld для создания новых узлов и метрик.

Метрики обновляют свои данные. Если использование памяти превышает некоторый предел, определяемый типом метрики, происходит передача данных во внешнее хранилище с помощью интерфейса GMetricCacheman, запрашиваемого у объекта theWorld.

При успешной обработке поступивших данных вызывается метод GWorld.commit(), чтобы сообщить об изменениях в метриках объектам, которые, возможно, зарегистрированы для уведомления о событии изменения в наборе данных.

Приложение также может проверять уведомления о событиях. Для этого необходимо создать экземпляр класса GEventClientUDP и зарегистрировать объект, методы которого вызываются при получении нового уведомления:

```

GEventClientUDP eventClient =
    new GEventClientUDP(serverAddress);
eventClient.start();
eventClient.register(new GMyEventListener());
eventClient.setFilter(System, Subsystem, Level, Source);

```

В первой строке создаётся новый объект, обеспечивающий взаимодействие с ретранслятором событий. Затем запускается фоновый поток, занимающийся получением уведомлений с сервера. В третьей строке выполняется регистрация экземпляра класса GMyEventListener, который должен соответствовать интерфейсу GEventListener и предоставлять реализацию метода onEvent(). Указанный метод будет вызываться каждый раз при получении очередного уведомления о событии. Последняя строка примера кода устанавливает подписку на ретрансляторе событий с параметрами фильтра, указанными при вызове функции setFilter(). Пользователь может создать несколько фильтров при необходимости, вызвав данную функцию несколько раз с необходимыми параметрами.

Для доступа к значениям метрик использует методы, описанные интерфейсом GMetric, например:

```

GMetric metric = node.getMetric("eth0_available");
if (metric != null) {
    GType lastValue = metric.getLastValue();
    int prType = lastValue.toInt();
    if (prType == -1) {
        GEvent event = new GEventXML(GEvent.WARN);
        GOption option = new GOption("system");
        option.addAttribute("name", "network");
        GOption suboption = new GOption("interface");
        suboption.addAttribute("name", "eth0");
        option.addSubOption(suboption);
        event.addOption(option);
        eventClient.sendNotify(event);
    }
}

```

```

        GEvent event2 = new GEventXML( GEvent.WARN );
        event2.setBodyXML( xmlEventStr );
        eventClient.sendNotify( event2 );
    }
}

```

Этот пример демонстрирует также два варианта формирования тел идентичных событий: данные объекта event формируются при помощи методов интерфейса GEvent, а данные event2 с помощью специфичного для GEventXML метода, где xmlEventStr определено следующим образом:

```

String xmlEventStr = "<system name = \"network\">" +
                    "<interface name = \"eth0\"></system>"

```

В результате будет сформировано событие, тело которого было описано во второй главе. Параметры идентификации события (id) и отметки времени создания (timestamp) генерируются автоматически.

Схожим образом можно получить и доступ к данным полученного события в методе onEvent():

```

public bool onEvent(GEvent event) {
    GOption option = event.getOption("system");
    String ethernetValue = "eth0";
    if (option != null) {
        GOption interfaceOption =
            option.getSubOption("interface");
        if (ethernetName.equals(
            interfaceOption.getAttributeValue("name")))
        {
            /* handle */
            return true;
        }
    }
    return false;
}

```

Здесь мы проверяем наличие опции "system" а затем значение атрибута вложенной опции "interface". Если значение равно "eth0" то выполняем код обработки события.

Для установки триггера можно использовать метод `setCrigger()`, описанный в интерфейсе `GEventClient`. Параметрами являются полные имена классов необходимых триггеров и действий, а также набор опций для каждого из них:

```
GOptions triggerOptions = new GOptions();
triggerOptions.setOption("value", "80.0");
eventClient.setCrigger("grate.trigger.GTriggerMore",
    triggerOptions, "grate.reactor.GReactorLog",
    reactorOptions, "cluster1/*/CPU_Temperature",
    isSignaled);
```

В приведённом выше примере в ЦСД создаётся объект, который генерирует триггеры для метрик `CPU_Temperature` всех дочерних узлов `cluster1` в ИСД (в нашем случае – это узлы кластера). В качестве триггеров выступают экземпляры класса `GTriggerMore` (условие превышения заданного значения), а действием является `GReactorLog` – запись в лог-файл информации о выполнении условий триггера. Булев параметр `isSignaled` определяет тип триггера: сбрасываемый ли триггер (значение `false`) или переключаемый (значение `true`).

Если приложение пользователя написано на других языках программирования, то в нём необходимо либо реализовать поддержку парсера данных в XML формате, либо использовать возможности JNI для взаимодействия с java-классами `CM Grate`.

Для удобства обработки событий сторонними приложениями предлагается также утилита `grate_listener`, которая позволяет запускать указанное в её командной строке приложение с передачей ему параметров через переменные окружения. Например, такая командная строка используется для запуска на выполнение скрипта `action.sh` при обработке событий от ретранслятора событий `CM server.address`:

```
grate_listener -f "system=network,src=*,level=warn" \
-s server.address -a action.sh
```

Фильтр устанавливается ключом командной строки `-f`, после которого через запятую перечисляются пары «параметр = значение». Если какой-то из параметров не задан, то используется символ `*`, означающий что этому параметру удовлетворяет любое значение при проверке соответствия фильтру пришедшего в ЦСД события.

Перед запуском на выполнение указанной команды `grate_listener` задаёт переменные окружения в виде (на примере данных уведомления, представленного во второй главе):

```
event.id = 123
event.timestamp = 46712631236
event.level = warn
event.system.name = network
event.system.interface.name = eth0
event.system.src = node2
event.broadcast =
event.str = Interface eth0 link down
event.action.default = ignore
event.type.value = failure
```

Это позволяет делегировать обработку события даже языкам программирования со слабыми функциональными возможностями, но имеющим доступ к переменным окружения. Отметим, что пустое значение `event.broadcast` позволяет приложениям определить наличие указанной опции в описании события.

Для передачи данных пользовательских метрик в СМ создана утилита `grate_metric`, работа с которой описана в приложении Б. Используя параметры, переданные в командной строке, утилита формирует пакет данных, который направляет в ЦСД.

3.2. Расширение возможностей установленной системы мониторинга Ganglia

На кластере КВЦ-1, собранном в 2004 году из неспециализированных компонент [10] использовалась СМ Ganglia для контроля характеристик ВК.

Невозможность создания отклика системы штатными средствами используемой СМ привела к необходимости расширения возможностей СМ

Ganglia при помощи CM Grate для контроля негативных изменений повышения температуры вычислительных узлов, например, в случае иногда возникающего отказа активного охлаждения. Серверная часть `grated` реализовала большую часть функциональности `gmetad` и была настроена на сбор данных с использованием служб `gmond`, предоставляемых системой Ganglia с предварительно настроенной пользовательской метрикой температуры.

При этом проводился анализ метрик, проверка триггеров, доступны были также другие возможности системы. При превышении заранее указанной величины происходила запись в журнал событий, отправка уведомления администратору кластера. В критическом случае принудительное отключение узла инициировалось аппаратными средствами, однако сигнал о необходимости отключения вычислительного узла также можно использовать для программного отключения средствами ОС.

Тестирование, выполненное при мониторинге до 15 узлов ВК показало, что производительность CM Grate (модуля `grated`) выше, чем у CM Ganglia (модуля `gmetad`). Тем не менее `grated` уступает в объёмах используемой памяти на управляющем сервере [9]. В настоящее время, произвести рост затрат на реальном кластере с числом узлов более 15 не представляется возможным из-за отсутствия у автора доступа к подобным ВК для получения данных мониторинга. Тем не менее, можно оценить рост затрат производительности, исходя из имеющихся фактических данных и результатов моделирование контроля 100 вычислительных узлов по методике оценки масштабирования, использованной разработчиками CM Ganglia.

При использовании источников данных, аналогичных CM Ganglia (`gmond`), и групповой передачи данных (`multicast`, протокол UDP) рост накладных расходов на организацию сетевых подключений для получения значений контролируем с увеличением числа узлов незначителен. Большее влияние на рост вычислительных затрат оказывает проверка условий триггеров, общее количество которых зависят от числа вычислительных узлов. При проведении численных экспериментов данное утверждение подтверждается: для 8 узлов выигрыш в производительности `grated` в сравнении с `gmetad` составляет 45%, при увеличении числа узлов разница уменьшается

по закону близкому к линейному, доходя при моделировании мониторинга 100 вычислительных узлов до 10%. Максимальные затраты на мониторинг составляют менее 0.5% от всего времени работы центрального процессора (ЦПУ) сервера, на котором запускался `grated` (по данным за 41 день статистики использования вычислительного времени процессом `java`, который исполнял `grated`).

Модуль `grated` проводил проверку триггеров (по одному на каждый узел), расширяя тем самым базовые функциональные возможности используемой СМ. Визуализация выполнялась `grate-client` на основе данных, предоставляемых от `grated`, тем не менее веб-сервис СМ Ganglia также мог визуализировать данные, получаемые от `grated`.

Тестирование работы `grated` и `gmetad` проводилось на управляющем узле кластера ВЦ ДВО РАН под управлением WhiteBox Linux 3.0 с ядром 2.4.20-8. Результаты приведены в таблице 3.1, где показаны значения общего времени выполнения T_0 и относительное время загрузки процессора T_p на управляющем узле. Использовалась Sun JRE 1.5.

Таблица 3.1: Сравнение производительности Ganglia и Grate

Выполняемое приложение	кластер, 8 узлов		модель кластера, 100 узлов	
	T_0 , час	T_p , %	T_0 , час	T_p , %
<code>grated</code>	768	0.14	50.8	9.19
<code>gmetad</code>	888	0.23	192	10.6

Важным результатом является то, что расширение возможностей установленной системы произошло без остановки её работы, без перенастройки источников данных. А замена агента сбора данных позволила получить сравнимую или лучшую производительность при большей функциональности. Получить аналогичный результат с использованием других доступных СМ было невозможно.

Тот же подход расширения возможностей СМ можно применить в задачах управления потреблением питания ВК. Серверное оборудование в основном оптимизировано для вычислительных задач, поэтому даже в режиме простоя потребление энергии зачастую незначительно меньше пикового потребления при полной нагрузке во время вычислительного процес-

са. Принудительное отключение питания неиспользуемых узлов в кластере может позволить сократить использование электроэнергии «в холостую». Схожей задачей является и снижение потребления энергии за счёт снижения частоты процессоров и перевода ряда устройств в режим сна. Это особенно важно при нарушениях подачи электроснабжения. Мощность, потребляемая кластером, зачастую настолько велика, что блоки бесперебойного питания неспособны обеспечить длительную работу вычислительных узлов, однако прерывание длительного вычислительного процесса недопустимо. Снижение потребляемой мощности позволяет кластеру дольше работать от резервной батареи питания.

3.3. Взаимодействие с системой паравиртуализации Xen

В ряде случаев приложения пользователей кластера занимают не все вычислительные ресурсы узла, на котором выполняются, в то время как более ёмкие с точки зрения необходимых ресурсов памяти или процессорного времени задачи вынуждены простаивать в очередях в ожидании необходимых узлов.



Рис. 3.2: Неэффективное использование вычислительного узла процессами пользователя

На рисунке 3.2 приведён пример реальной задачи пользователя, которая в силу особенностей вычислительного алгоритма использует не все вычислительные ресурсы узла кластера, тем не менее, занимая его полностью в очереди планировщика заданий.

При использовании системы паравиртуализации Xen на экспериментальном кластере ВЦ ДВО РАН [11] была добавлена возможность миграции выполняющихся процессов с одного вычислительного узла на другой, что позволяет достичь полноценной утилизации ресурсов физического вычислительного узла несколькими виртуальными вычислительными узла-

ми. Пример данных системы мониторинга при более эффективном использовании вычислительных ресурсов приведён на рисунке 3.3

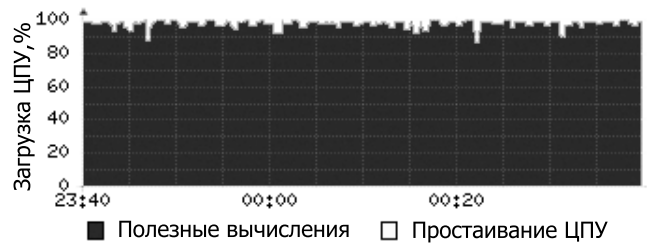


Рис. 3.3: Использование ресурсов вычислительного узла близкое к максимальному

Системы виртуализации позволяют запускать несколько копий операционных систем на одной машине. При этом одна из операционных систем выполняет служебные функции и называется хостовой, а остальные – гостевыми. Миграция выполняющейся копии операционной системы доступна с использованием стандартных средств Xen.

Дополнительная возможность создания контрольной точки выполняющейся операционной системы может использоваться для восстановления работоспособности при сбое в работе вычислительного кластера или отключении электроэнергии.

Для тестирования Xen и его взаимодействия с Grate был создан отдельный логический кластер в составе экспериментального кластера, по пакету программ идентичный основному кластеру, но с модифицированным для поддержки Xen ядром Linux и установленным ПО управления системой виртуализации. Во время тестирования осуществлялся процесс миграции выполняющейся копии гостевой ОС с запущенными вычислительными задачами с одного из физических узлов на другой, обладающий большими доступными ресурсами на момент миграции.

Программный комплекс на базе СМ Grate анализировал использование ресурсов на вычислительном узле, обрабатывая данные, предоставляемые промежуточным уровнем (эффективная загрузка процессора, объём занятой памяти). При необходимости триггером инициировалось выполнение shell-скрипта с командами миграции. При этом не происходило значительного простаивания вычислительного процесса. Как показали тестовые замеры, для малого числа узлов, участвующих в расчёте параллельной за-

дачи с интенсивной передачей данных между узлами, потери были малы. В таблице 3.2 приведены результаты для 3 узлов.

Таблица 3.2: Потери производительности при миграциях гостевой ОС

Число миграций в час	12	6	3	2	0
Производительность в GFlops	8.83	9.73	10.57	10.68	11
Относительные потери производи- тельности в %	19.72	11.55	3.9	2.9	-

Тестирование производилось на тесте HPL¹, что позволило — одновременно с проверкой корректности вычислений — провести измерение производительности ВК. Тест HPL состоит в решении системы алгебраических уравнений указанной размерности методом LU-разложения. Сочетание различных параметров теста HPL позволяет варьировать уровень загрузки вычислительного кластера. Однако, заранее неизвестно, какие значения параметров оптимальны для конкретного кластера и задачи. Следует отметить, что тест HPL достаточно интенсивно использует коммуникационную сеть. Результаты теста осреднялись на основе серии из 10 испытаний.

Для задач, в меньшей мере зависящих от интенсивности обмена данными между вычислительными узлами, потери производительности будут меньше. При отсутствии миграций использование Xen не приводит к заметным потерям производительности: 11 ГФлопс — это экспериментально подтверждённая производительность для кластера с аналогичным набором аппаратного обеспечения [10].

Дальнейшее развитие разработанного ПО может быть связано с реагированием ВК на сбои. В таком случае ряд вычислительных заданий можно сохранить с использованием механизма создания контрольных точек гостевой ОС. Стоит отметить, это возможно только для заданий, не использующих алгоритмы передачи данных, зависящих от таймеров ожидания. В общем случае параллельное приложение невозможно корректно сохранить с возможностью восстановления вычислительного процесса, используя лишь возможности системы виртуализации.

¹<http://www.netlib.org/benchmark/hpl>

3.4. Взаимодействие с нейросетевым модулем

С ростом числа узлов и контролируемых характеристик системному администратору становится всё сложнее в непрерывном режиме следить за негативными изменениями в значениях опрашиваемых характеристик ВК, возникает потребность в автоматизации рутинных операций. При этом многие изменения в работе ВК могут быть обнаружены и в ряде случаев предотвращены с использованием различных систем автоматизированного контроля.

Примером таких обнаруживаемых изменений могут служить последствия неэффективно работающего приложения. Например, довольно типичной ситуацией является неверный выбор семейства функций или топологии потоков данных при использования MPI-обменов в программе пользователя, а также простаивание в ожидании на операциях ввода-вывода. На рисунке 3.4 изображены данные мониторинга на вычислительном узле, где приложение пользователя занималось в основном файловыми операциями, полезные вычисления составляли менее 30 процентов процессорного времени (зона А на графике). Такая ситуация хорошо распознаётся обученной ИНС и может быть предотвращена при своевременном вмешательстве. Например, после рекомендаций, приложение пользователя, запущенное позднее (зона Б), было лишено обсуждаемых недостатков.

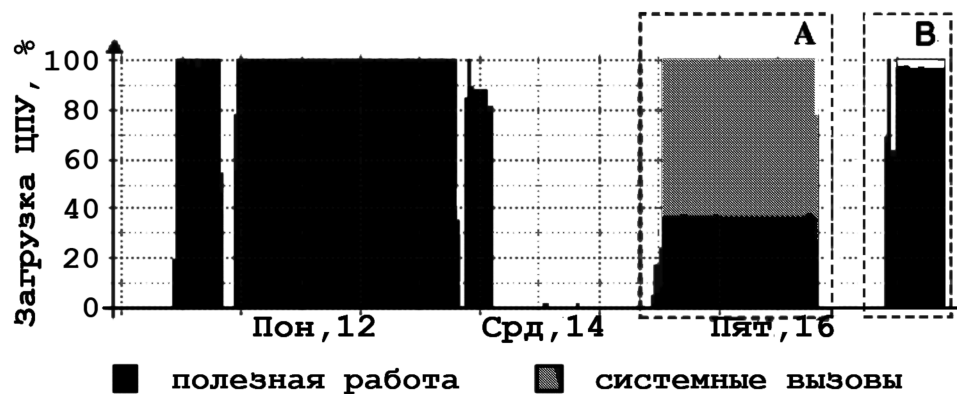


Рис. 3.4: График неэффективного использования ресурсов приложением пользователя

В приведённом примере задание не вредило приложениям других пользователей и не представляло опасности для работы всей вычислительной системы, однако на рисунке 3.5 приведена более серьёзная проблема

с использованием одним из заданий виртуальной памяти в объёме, приведшем к использованию файла подкачки (выделенная пунктиром зона А) и отказу ряда системных приложений (в частности отсутствие данных на графике стало следствием отказа используемой системы мониторинга), а также общему замедлению работы всех приложений вычислительного узла. Очевидно, такое поведение задания пользователя недопустимо и критическая ситуация должна быть предотвращена как можно раньше.

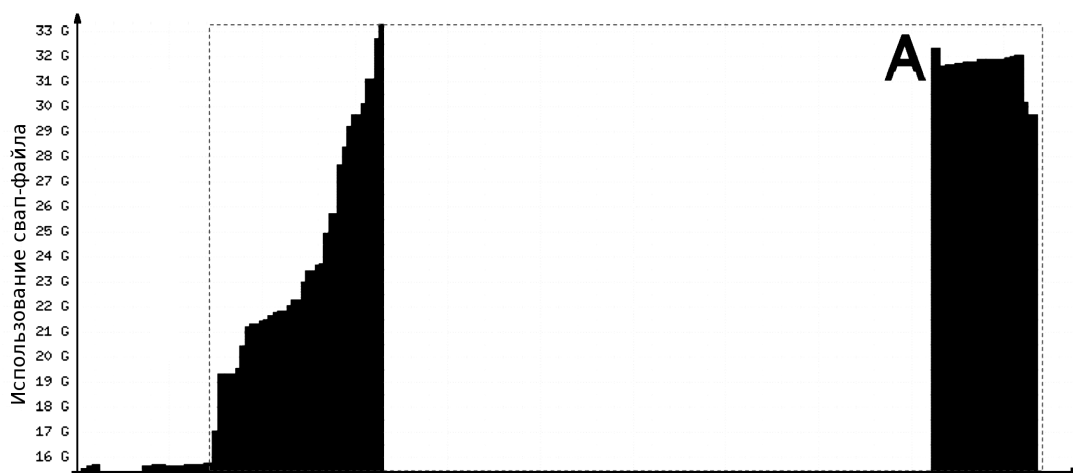


Рис. 3.5: Критическая ситуация на вычислительном узле

В работах [12, 13] для проверки возможности применения ИНС в целях выявления состояния, в котором находится вычислительный узел и ВК в целом, для численных экспериментов были выбраны ИНС Кохонена и встречного распространения. Было показано, что обученная ИНС встречного распространения способна верно определять текущее состояние ВК при заранее неизвестных входных данных. Это обусловило выбор модуля ИНС в качестве компоненты разработанной СМ, ответственной за выявление изменения состояния в ВК.

Схема взаимодействия СМ и модуля ИНС, выполняющегося как независимое приложение, приведена на рисунке 3.6. Тонкими стрелками отображены направления постоянно существующих потоков данных, символы стрелок отображают направления возможной передачи данных и управляющих воздействий. Таким образом в эксперименте участвовали четыре компонента: вычислительный кластер, система мониторинга, приложение-обработчик событий и модуль ИНС.

Вычислительной платформой служил кластер, состоящий из 4 узлов

под управлением операционной системы Linux CentOS v4.4. Модуль ИНС выполнялся как независимое приложение, получающее данные от службы grated. После обработки входных данных через механизм уведомлений система мониторинга получала информацию о событиях (изменениях в состоянии ВК). Приложение-обработчик было зарегистрировано для получения уведомлений, выполнения записи о событиях в журнал и отправки электронной почты администратору ВК.

Приложение-обработчик и служба grated CM Grate выполнялись на управляющем узле, модуль ИНС на отдельной вычислительной станции, обращающейся к разработанной СМ через ретранслятор событий и ЦСД, для отправки уведомлений и получения значений метрик соответственно [27, 28].

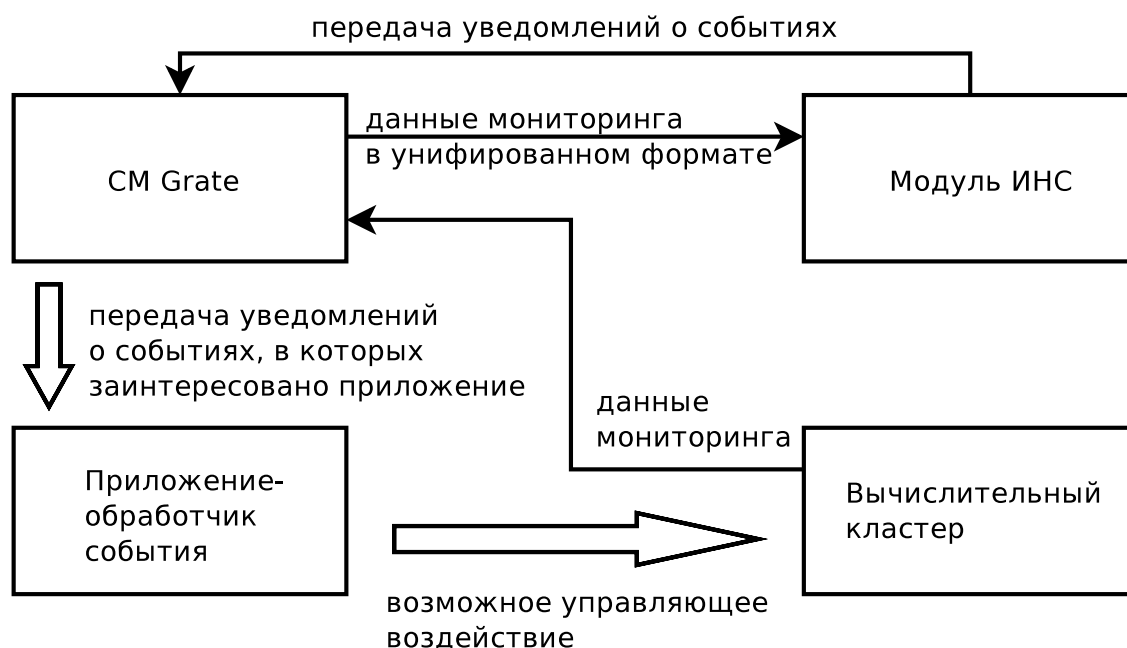


Рис. 3.6: Схема взаимодействия модуля ИНС и CM Grate

Было проведено тестирование корректности функционирования модуля ИНС, взаимодействующего с разработанной СМ посредством уведомлений. Результаты работы модуля ИНС в режиме без взаимодействия с СМ были известны из ранее проведенных исследований и использовались в качестве эталонных.

В процессе предварительного исследования были выделены следующие значимые входные параметры для работы ИНС: доля свободного места в файле подкачки, отношение числа процессов к числу ядер, доля свобод-

ной оперативной памяти, отношение загруженности процессора пользовательскими процессами к системным процессам, доля свободной памяти на жестком диске, число принятых байт на сетевых интерфейсах, число отданных байт и производные от них величины. Такие параметры измерялись как для кластера в целом, так и для каждого узла в отдельности. При обучении ИНС в разработанном ПО возможно варьировать следующие параметры: скорость обучения, необходимая точность (если проводится обучение с проверкой), максимальное число эпох (итераций обучения). В результате обучения методом обратного распространения ошибки была получена двухслойная ИНС, которая способна на основе входных данных для одного вычислительного узла (или для ВК) выдать на выходах значения, указывающие на вероятность нахождения узла (или ВК) в одном из состояний. Под состоянием понимаются значения различных значимых метрик (температура процессора, количество свободной оперативной памяти, число выполняемых процессов и др.). В процессе работы модуль ИНС также запоминал входные параметры, при которых ИНС давала неопределённый ответ, что позволяло в дальнейшем построить на этих данных обучающую выборку и корректировать ИНС. В процессе исследования была выбрана и испытана ИНС встречного распространения с сигмоидальной функцией активации. Более подробную информацию о функционировании ИНС и методиках формировании входных векторов данных, в том числе нормализации входных векторов данных и задании случайных начальных ненулевых весов, можно получить в [4, 29].

Для получения информативной выборки входных параметров необходим большой объем данных по состоянию узлов во время штатной работы ВК. С целью накопления достаточного количества входных данных на экспериментальном вычислительном кластере был выполнен тест HPL. Модуль ИНС опрашивал состояние ВК через систему мониторинга. При этом сеть обучалась для выявления следующих основных классов состояний: «неэффективное использование ЦПУ» (в смысле малой доли полезных вычислений в пространстве пользователя. Это состояние было смоделировано специально разработанной утилитой, поскольку получить его из данных мониторинга теста HPL было невозможно), «штатная работа»

(доля полезных вычислений до 75%, обмен данными между вычислительными узлами невелик), «стадия интенсивных вычислений» (доля полезных вычислений более 75%, значительный обмен данными между узлами ВК), «простой ВК» (нагрузка ЦПУ на контролируемых узлах составляет менее 5%).

Обучение модуля ИНС было произведено с использованием прямого опроса состояния ВК через функции системы мониторинга. Под состоянием понимаются значения различных значимых метрик (температура процессора, количество свободной оперативной памяти, число выполняемых процессов и др.). Полученные сведения были использованы для формирования входных векторов, которые использовались для обучения сети. Далее сходным образом были сформированы входные данные, которые обученная ИНС должна была отнести к какому-либо классу состояний. Более подробную информацию о формировании входных векторов данных можно получить в приложении Г, работах [14, 29].

В рамках исследования взаимодействия нейросетевого модуля с СМ было выполнено более ста тестов HPL с различными параметрами, по методике, указанной в [13], с единственным отличием: данные мониторинга предоставлялись не выборкой из базы данных RRD (Round-robin database) СМ Ganglia, а из разработанной СМ в реальном времени. Для автоматизации тестирования был создан shell-скрипт, который формировал файл описания задания (тест HPL), ставил задание на выполнение через систему диспетчеризации PBS Torque, для каждого вычислительного узла извлекал необходимые данные из полученных от СМ Grate и сохранял результат в файл для дальнейшего анализа корректности работы модуля ИНС. Было проведено обучение ИНС при различных комбинациях входных векторов.

Было проведено обучение ИНС при различных комбинациях входных параметров, полученных системой мониторинга Grate. Результаты работы сети для неизвестных на момент обучения наборов данных представлены в таблице 3.3, они совпадают с приведёнными в работе [13], что говорит о корректности взаимодействия СМ и модуля ИНС.

Таблица 3.3: Доля верно распознанных классов

Число нейронов	Количество циклов обучения						
	10	20	30	50	100	200	300
4	0,76	0,80	0,79	0,80	0,77	0,80	0,77
8	0,97	0,93	0,99	0,99	0,93	0,99	1
12	0,93	0,93	1	1	1	1	1
20	0,94	0,95	0,96	0,96	1	1	1
50	0,93	0,93	0,94	0,96	0,96	0,96	1

Имея на входе значения метрик, ИНС встречного распространения при 12 нейронах и 30 циклах обучения верно отнесла текущее состояние к одному из классов состояний, приведённых выше. При изменении класса состояния, модуль ИНС посылал уведомление о событии в СМ, которая в свою очередь передавала его приложению-обработчику. Также была смоделирована ситуация активного использования файла подкачки с помощью тестовой задачи, активно выделяющей оперативную память без возврата её системе. ИНС успешно распознала критическую ситуацию и уведомила администратора кластера.

В процессе тестирования также была смоделирована ситуация, схожая с приведённой на рисунке 3.5. ИНС успешно распознавала критическую ситуацию и уведомляла администратора кластера, что позволяет судить о применимости модуля ИНС в рамках СМ расширенной функциональности для выявления таких состояний.

В тестировании использовались ИНС с относительно несложной топологией для определения очевидных изменений в работе ВК в автоматизированном режиме. Тем не менее, развитие модуля ИНС позволяет изменять тип и конфигурацию ИНС, не переписывая при этом исходного кода СМ Grate. Более того, уведомления о событиях и данных мониторинга могут обрабатываться одновременно несколькими приложениями (или, например, независимыми модулями ИНС), что позволяет более полно анализировать состояние ВК с применением и других методов и алгоритмов.

3.5. Пути дальнейшего совершенствования разработанной системы мониторинга

Предложенная архитектура СМ обладает недостаточной децентрализацией сбора данных, что ухудшает отказоустойчивость системы при сбое центров ретрансляции событий или центров сбора данных. Из-за централизации растут накладные расходы на обработку триггеров в корне иерархии СМ. Механизм подписки, хотя и прост в реализации и работе, также не является распределённым, что накладывает ограничения на общее число возможных подписчиков.

Ввиду этого, при совершенствовании данной архитектуры СМ для контроля большого числа вычислительных узлов, необходимо предусмотреть несколько ЦСД с вещанием данных, а не запросом их. Как показано ранее [88], указанная проблема не является существенной для ВК с числом узлов до 64.

Следует также отметить возможные сложности при масштабировании системы на такие ВК, поскольку ожидаемый рост объёма передаваемой информации будет требовать большего процессорного времени на обработку данных при некорректно выбранной структуре потоков данных в СМ. Недостатком реализации на языке Java являются большие требования к оперативной памяти, чем при использовании машинно-зависимых программных продуктов в виду системных требований JVM. Использование текстового формата XML для передачи событий хотя и является платформенно-независимым, но при значительном числе сообщений приведёт к значительному росту трафика и вычислительных затрат на упаковку и распаковку сообщений.

Естественным развитием СМ для улучшения производительности является применение более компактных форматов данных передачи сообщений и значений метрик.

В силу политик безопасности и абстрагирования JVM от ОС невозможно реализовать источники данных СМ на языке Java. Существенным улучшением реализации СМ с точки зрения производительности будет использование машинно-зависимых языков, например C/C++.

Недостатком имеющейся реализации с точки зрения пользователя являются недостаточно развитые в базовой реализации графические средства представления и анализа данных, что может затруднить внедрение системы мониторинга. Тем не менее гибкость базовой реализации позволяет изменять внешний вид и свойства отображаемых элементов, а также расширять виды графиков и способы их отображения.

За всё время тестирования и применения на кластере ВЦ ДВО РАН в работе Grate не было никаких ошибок, повлекших за собой нарушение работы вычислительной системы или потерю данных. Тем не менее, вопрос целостности данных в СМ большого ВК является важным и требует более тщательной проработки.

3.6. Выводы по третьей главе

По результатам вычислительных экспериментов сформулируем следующие выводы:

1. Разработанная СМ Grate сравнима по производительности с аналогами, либо превосходит их при выполнении тех же задач. Для упрощения взаимодействия создан набор классов, входящих в СМ Grate и утилиты командной строки (`grate_listener`, `grate_metric`).
2. Показано, что предложенный подход к взаимодействию СМ со сторонними приложениями позволяет организовать совместную работу нескольких приложений для решения различных задач, связанных с контролем ВК.
3. Обученный модуль ИНС в реальном времени позволяет определять классы состояний ВК. В численном эксперименте было исследовано несколько конфигураций ИНС и показано, что при относительно небольшом числе нейронов и циклах обучения модуль способен определять класс состояний с высокой точностью.
4. Взаимодействие с системой виртуализации Xen позволяет достичь повышения эффективности работы ВК путём перераспределения вычислительной нагрузки текущих заданий и освобождения вычисли-

тельных узлов для других заданий. Получены данные потерь производительности при использовании механизмов миграций при интенсивных вычислениях.

Заключение

Перечислим основные результаты диссертационной работы.

1. Разработана новая архитектура СМ, обладающая широким набором функциональных возможностей. Создана её программная реализация на языке программирования Java.
2. Предложен и исследован механизм взаимодействия СМ со сторонним ПО посредством подписки и уведомлений о событиях. Показана эффективность такого взаимодействия в задачах автоматизированного контроля ВК при решении прикладных задач, общих для широкого круга вычислительных кластеров.

Разработанные в диссертации подходы и реализующее их ПО допускают естественное развитие и обобщение для задач систем мониторинга, в том числе глобальных РВС и вычислительных сетей.

СМ Grate является более гибкой и просто расширяемой по сравнению с большинством аналогов: для добавления нового триггера или действия достаточно реализовать интерфейс взаимодействия, определённый в исходном коде Grate. Созданный таким образом модуль можно начинать использовать, не останавливая работу СМ и ВК. Исключение требований к реализации полноценных механизмов удалённого вызова процедур позволяет использовать систему даже в скриптовых языках со слабо развитыми функциональными возможностями.

Разработанная СМ предоставляет унифицированный подход к обработке событий, позволяя приложению подписаться на необходимые события уже во время мониторинга. Возможна также обработка событий с помощью действий и триггеров, заданных в конфигурационном файле.

Таким образом, СМ Grate расширяема в большей степени, чем аналоги: на всех уровнях динамически формирует внутренние структуры и содержит более широкий набор базовых возможностей (система уведомле-

ний, триггеры и др.). В сравнении с рядом аналогов ПО, разработанное по данной архитектуре, превосходит существующие по важнейшим параметрам: либо по набору базовых возможностей, либо по производительности.

Построение программных комплексов, реализующих предложенную архитектуру мониторинга, упрощает создание систем контроля и управления ВК и других вычислительных систем.

Литература

1. Воеводин В. В., Воеводин Вл. В. Параллельные вычисления. — СПб.: БХВ-Петербург, 2002. — 608 с.
2. Гамма Э. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. — СПб.: Питер, 2007 — 366 с.
3. Кохонен Т. Самоорганизующиеся карты. — М.: Бином, 2007. — 655 с.
4. Круглов В. В. Искусственные нейронные сети. Теория и практика. 2-е изд., стереотип. / В. В. Круглов, В. В. Борисов. — М.: Горячая линия-Телеком, 2002. — 382 с.
5. Ларичев О. И. Системы поддержки принятия решений: современное состояние и перспективы развития / О. И. Ларичев, А. Б. Петровский // Итоги науки и техники. — М.: ВИНТИ, 1987. — Т. 21. — С. 131–164.
6. Ляпунов А. А. О некоторых вопросах обучения автоматов // В кн.: Принципы построения самообучающихся систем. — Киев: 1962. — С. 115–118.
7. Ляпунов А. А. О строении и эволюции управляющих систем в связи с теорией классификаций. — Проблемы кибернетики, 1973. — вып. 27. — С. 7–18.
8. Моисеев Н. Н. Математические задачи системного анализа. — М.: «Наука», 1981. — 487 с.
9. Пересветов В. В. Удалённый доступ к вычислительному кластеру ВЦ ДВО РАН / В. В. Пересветов, А. Ю. Сапронов, А. Г. Тарасов, Т. С. Шаповалов // Вычислительные технологии. — Т. 11, специальный выпуск. — Новосибирск: Изд-во ИВТ СО РАН, 2006. — С. 45–51.

10. Пересветов В. В. Вычислительный кластер бездисковых рабочих станций. Препринт №83 / В. В. Пересветов, А. Ю. Сапронов, А. Г. Тарасов. — Хабаровск: ВЦ ДВО РАН, 2005. — 50 с.
11. Пересветов В. В. Организация работы вычислительного кластера в режиме удалённого доступа. Препринт №110 / В. В. Пересветов, А. Ю. Сапронов, А. Г. Тарасов, Т. С. Шаповалов. — Хабаровск: ВЦ ДВО РАН, 2007. — 34 с.
12. Писарев А. В. Нейросетевые компоненты мониторинга вычислительного кластера / А. В. Писарев, В. В. Пересветов // Материалы конференции Информационные и коммуникационные технологии в образовании и научной деятельности (21–23 мая 2008 года, г. Хабаровск). — Хабаровск : Изд-во ТоГУ, 2008. — С. 319–323.
13. Писарев А. В. Модульная нейросетевая система мониторинга вычислительного кластера / А. В. Писарев, А. Г. Тарасов // Материалы межрегиональной конференции Информационные и коммуникационные технологии в образовании и научной деятельности (21–23 сентября 2009 года, г. Хабаровск). — Хабаровск: ТОГУ, 2009 — С. 289–296.
14. Рассел С. Искусственный интеллект: современный подход / С. Рассел, П. Норвиг. — М.: Вильямс, 2007. — 1408 с.
15. Смагин С. И. Опыт построения GRID-сети ДВО РАН / С. И. Смагин, А. Ю. Сапронов, А. Г. Тарасов, Т. С. Шаповалов // Современные информационные технологии для научных исследований. Материалы Всероссийской конференции (Магадан, 20–24 апреля 2008 г). — Магадан: СВНЦ ДВО РАН, 2008. — С. 23–24.
16. Тарасов А. Г. Мониторинг вычислительного кластера с использованием Java-технологий // XXX Дальневосточная математическая школа-семинар имени академика Е.В. Золотова. — Хабаровск: Изд-во ДВГУПС, ИПМ ДВО РАН, 2005. — С. 201.
17. Тарасов А. Г. Доступ к ресурсам вычислительного кластера через сеть Интернет / А. Ю. Сапронов, А. Г. Тарасов, Т. С. Шаповалов // Мате-

- риалы конференции: VII Всероссийская конференция молодых учёных по математическому моделированию и информационным технологиям (1–3 ноября 2006, г. Красноярск). — Красноярск: Изд-во НГУ, ИВТ СО РАН, 2006. — С. 93–95.
18. Тарасов А. Г. Применение системы виртуализации XEN на вычислительных кластерах / А. Г. Тарасов, А. Ю. Сапронов, Т. С. Шаповалов // Материалы конференции: VIII Всероссийская конференция молодых учёных по математическому моделированию и информационным технологиям (27–29 ноября 2007, г. Новосибирск). — Новосибирск: Изд-во НГУ, ИВТ СО РАН, 2007. — С. 106–107.
 19. Тарасов А. Г. Организация GRID-сети ВЦ ДВО РАН / А. Г. Тарасов, С. И. Щерба, Т. С. Шаповалов // Научный сервис в сети Интернет: многоядерный компьютерный мир (24–29 сентября 2007, г. Новороссийск). — Москва: Изд-во МГУ, МГУ, 2007. — С. 94–95.
 20. Тарасов А. Г. Трёхуровневая система мониторинга расширенной функциональности // Параллельные вычислительные технологии: труды международной конференции (Санкт-Петербург, 28 января–1 февраля 2008 г.). — Челябинск: Изд. ЮУрГУ, 2008. — С. 464–469.
 21. Тарасов А. Г. Совместное применение системы мониторинга вычислительного кластера и системы виртуализации XEN // Информационные и коммуникационные технологии в образовании и научной деятельности. Материалы конференции (21–23 мая 2008, г. Хабаровск). — Хабаровск: Изд-во ТоГУ, 2008. — С. 333–338.
 22. Смагин С. И. Web и GRID технологии обеспечения доступа к ресурсам вычислительного кластера ВЦ ДВО РАН / С. И. Смагин, А. Г. Тарасов, Т. С. Шаповалов, В. В. Пересветов, А. Ю. Сапронов // Материалы Межрегиональной конференции Информационные и коммуникационные технологии в образовании и научной деятельности (21–23 мая 2008, г. Хабаровск). — Хабаровск: Изд-во ТоГУ, 2008. — С. 69–76.
 23. Тарасов А. Г. Развитие GRID-сети ДВО РАН / А. Г. Тарасов, Т. С. Шаповалов, А. Ю. Сапронов. // Параллельные вычислительные техноло-

- гии: труды международной конференции (Санкт-Петербург, 28 января–1 февраля 2008 г.). — Челябинск: Изд. ЮУрГУ, 2008. — С. 540.
24. Тарасов А. Г. Расширяемая система мониторинга вычислительного кластера // Вычислительные методы и программирование — Т. 10, № 1. — Москва: Изд. МГУ, 2009. — С. 147–158.
 25. Тарасов А. Г. Применение открытого программного обеспечения на ВЦ ДВО РАН // Информационные и коммуникационные технологии в образовании и научной деятельности. Материалы конференции (21–23 сентября 2009 года, г. Хабаровск). — Хабаровск: Изд-во ТоГУ, 2009. — С. 330–334.
 26. Тарасов А. Г. Grated. Свидетельство о регистрации программы для ЭВМ. РОСПАТЕНТ. — № 2010615286 от 17.08.2010.
 27. Тарасов А.Г. Опыт интеграции системы мониторинга Grate со сторонними программными комплексами // Материалы международной научно-практической конференции Суперкомпьютеры: вычислительные и информационные технологии (30 июня–2 июля 2010 г., г. Хабаровск). — Хабаровск: Изд-во ТоГУ, 2010. — С. 125–132.
 28. Tarasov Alexey G. Integration of computing cluster monitoring system // Proceedings of First Russia and Pacific Conference on Computer Technology and Applications (6–9 September 2010, Vladivostok). — Vladivostok: IACP FEB RAS, 2010. — P. 221–224.
 29. Хайкин С. Нейронные сети: полный курс: Пер. с англ. — М.: Вильямс, 2006. — 1104 с.
 30. Хорстманн К. С. Java 2. Библиотека профессионала. в 2 т. / К. С. Хорстманн, Г. Корнелл. — М.: Вильямс, 2008.
 31. Яблонский С. В. Некоторые вопросы надежности и контроля управляющих систем // Математические вопросы кибернетики. — вып. 1. — М.: Наука, 1988. — С. 5–25.

32. Agarwala S. Resource-Aware Stream Management with the Customizable dproc Distributed Monitoring Mechanisms / S. Agarwala, C. Poellabauer, J. Kong, K. Schwan, M. Wolf // Proc 12th IEEE International Symp. on High Performance Distributed Computing (HPDC-12, Seattle, June 2003). — IEEE Computer Society Press, 2003. — P. 250–259
33. Aguilera M. Matching events in a content-based subscription system / M. Aguilera, R. Strom, D. Sturman, M. Astley, T. Chandra // Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (Atlanta, 1999). — Atlanta: ACM, 1999. — P. 53–61.
34. Ali A. Job monitoring in an interactive grid analysis environment / A. Ali, A. Anjum, J. Bunn, R. Cavanaugh, F. van Lingen, R. McClatchey, H. Newman, W. ur Rehman, C. Steenberg, M. Thomas, I. Willers // Computing in High Energy Physics (Interlaken, 2004). — Interlaken: CHEP, 2004.
35. Anderson E. Extensible, scalable monitoring for clusters of computers / E. Anderson, D. Patterson. // Proceedings of the 11th Systems Administration Conference LISA '97 (San Diego, CA, October 1997). — San Diego: USENIX, 1997. — P. 9–16.
36. Anderson D. SETI@home: An Experiment in Public-Resource Computing / D. Anderson, J. Cobb, E. Korpela, M. Lebofsky, D. Werthimer. // Communications of the ACM. — Vol. 45, No. 11. — 2002. — P. 56–61.
37. Barth W. Nagios System and Network Monitoring. — San Francisco: No Starch Press, 2006. — 462 p.
38. Benveniste A. Diagnosis of asynchronous discrete event systems, a net unfolding approach / A. Benveniste, E. Fabre, C. Jard, S. Haar // IEEE Transactions on Automatic Control. — 2003. — No. 48(5). — P. 714–727.
39. Birman K. P. Exploiting virtual synchrony in distributed systems / K. P. Birman, T. A. Joseph. // Proceedings of the 11th ACM Symposium on Operating Systems Principles. — Austin: ACM, 1987 — P. 123–138.

40. Bonnet P. Towards sensor database systems / P. Bonnet, J. Gehrke, P. Seshadri. // Proceedings of the 2nd International Conference on Mobile Data Management (Hong Kong, 2001). — P. 3–14.
41. Buyya R. High Performance Cluster Computing: Architectures and Systems. — Vol. 1, 1st edn. — Prentice Hall: NJ, 1999. — 849 p.
42. Buyya R. Single system image: need, approaches, and supporting HPC systems // Proceedings of the Fourth International Conference on Parallel and Distributed Processing, Techniques and Applications (PDPTA'97, Las Vegas, 1997). — USA: CSREA Publishers, 1997.
43. Buyya R. PARMON: a portable and scalable monitoring system for clusters // International Journal on Software: Practice and Experience (SPE). — vol. 30. — USA: John Wiley and Sons, 2000. — P. 723–739.
44. Cui X. Efficient fume diffusion spotting in heterogeneous sensor networks / X. Cui, Q. Li, B. Zhao // HeterSanet '08: Proceeding of the 1st ACM international workshop on Heterogeneous sensor and actor networks (Hong Kong, China, 2008). — Hong Kong: AVM, 2008. — P. 31–36.
45. Culler E. D. Parallel Computing on the Berkeley NOW / E. D. Culler, A. Arpaci-Dusseau, R. Arpaci-Dusseau, B. Chun, S. Lumetta, A. Mainwaring, R. Martin, C. Yoshikawa, F. Wong // In Proceedings of the 9th Joint Symposium on Parallel Processing (Kobe, Japan, May, 1997). — Berkley: University of California, 1997. — P. 267.
46. DeRose, L. SvPablo: A Multi-Language Architecture-Independent Performance Analysis System / L. DeRose, D. Reed. // Proceedings of the International Conference on Parallel Processing (ICPP'99, Fukushima, Japan, September 1999). — P. 311–318.
47. Dilman M. Efficient reactive monitoring / M. Dilman, D. Raz // IEEE JSAC. — 2002. — Vol. 20, No. 4. — P. 668–676.
48. Efe K. Parallel Algorithms for Workstation Clusters / K. Efe, P. Uthayopas, V. Krishnamoorthy and T. Dechsakulthorn. // Proceeding

- of 1995 ICPP Workshop on Challenges for Parallel Processing. — US: University Park, 1995. — P. 140–147.
49. Foster, I. A Directory Service for Configuring High-Performance Distributed Computations / S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. // Proc. 6th IEEE Symp. on High-Performance Distributed Computing. — IEEE Computer Society Press, 1997. — P. 365–375.
 50. Foster I. The Grid: Blueprint for a New Computing Infrastructure / Ian Foster, Carl Kesselman, Morgan Kaufmann. — Morgan Kaufmann, 1999. — 701 p.
 51. Foster I. Globus Toolkit Version 4: Software for Service-Oriented Systems // Journal on Computing Science and Technology. — 2006. — Vol.21, No.4. — P. 513–520.
 52. Frey J. Condor-G: A Computation Management Agent for Multi-Institutional Grids / J. Frey, T. Tannenbaum, I. Foster, M. Livny, S. Tuecke // Journal on Cluster Computing. — 2002 — vol. 5. — P. 237–246.
 53. Genersereth M. Software Agents / M. Genersereth, S. Ketchpel // Communications of the ACM. — 1994. — 37(7). — P. 48–53.
 54. Green T. P. DQS User Interface Preliminary Design Document. — Tallahassee: Supercomputer Computations Research Institute, 1993.
 55. Grossberg S. Classical and instrumental learning by neural networks // Progress in theoretical biology. — vol. 3. — New York: Academic Press, 1974. — P. 51–141.
 56. Hayes B. Cloud computing // Communications of the ACM. — 2008. — Vol.7 — P. 9–11.
 57. Jiao J. Towards Efficient Monitoring / S. Naqvi, D. Raz, and B. Sugla // IEEE Journal on Selected Areas in Communications, special issue on recent advances in network management and operations — 2000. — Vol. 5, No. 18. — P. 723–732.

58. Krishnamoorthy V. Partitioning Strategies for Multiplying Dense Matrices on Workstation Networks / V. Krishnamoorthy, P. Uthayopas and K. Efe // Proceeding of First International Workshop on High-Speed Network Computing HiNet'95 (Santa Barbara, April 25,1995). — P. 74–82.
59. Larson S. M. Folding@home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology / S. M. Larson, C. D. Snow, M. R. Shirts, V. S. Pande. Edited by Richard Grant. // Computational Genomics. — Horizon Press, 2002.
60. Legrand I. C. Monalisa: An agent based, dynamic service system to monitor, control and optimize grid based applications / I. C. Legrand, H. B. Newman, R. Voicu, C. Cirstoiu, C. Grigoras, M. Toarta, and C. Dobre. // In Proc. of Computing in High Energy Physics (CHEP, September 2004). — Interlaken, 2004.
61. Litzkow M. Condor — A Hunter of Idle Workstation / M. Litzkow, M. Livny, and M. W. Mutka. // Proceedings of the 8th International Conference on Distributed Computing Systems (June 1988). — P. 104–111.
62. Luis V.M. A break in the clouds: towards a cloud definition / V. M. Luis, R. M. Luis, C. Juan, L. Maik // ACM SIGCOMM Computer Communication Review. — 2008. — Vol. 39, Issue 1. — P. 50–55.
63. Massie M. L. The ganglia distributed monitoring system: Design, implementation and experience / M. L. Massie, B. N. Chun, and D. E. Culler // Parallel Computing. — 2004. — Vol. 30, No. 7. — P. 817–840.
64. Minnich R. G. Supermon: High-performance monitoring for linux clusters / R. G. Minnich and K. Reid // In Proceedings of the Fifth Annual Linux Showcase and Conference (2001).
65. Morton M. S. Decision support systems: an organizational perspective / P. Keen, M. S. Morton. — Mass.: Addison-Wesley Pub. Co., 1978. — 264 p.
66. Moore J. Alert — Cluster Monitoring Package. — Charlottesville: University of Virginia, 1996.

67. MPI Forum MPI: A message-passing interface standard // International Journal of Supercomputer Application. — 1994. — Vol. 3/4, No. 8. — P. 165–416.
68. Mukherjee B. Network intrusion detection / B. Mukherjee, T. L. Heberlein, K. N. Levitt // IEEE Network. — 1994. — Vol. 3, No. 8. — P. 26–41.
69. Шилдт Г. Полный справочник по Java. — СПб: Вильямс, 2009. — 1040 с.
70. Raman S. A Model, Analysis, and Protocol Framework for Soft State-based Communication / S. Raman, S. McCanne // Computer Communication Rev. — 1999. — Vol. 29, No. 4. — P. 15–25.
71. Roder C. Flexible status measurement in heterogeneous environment / C. Roder, T. Ludwig, A. Bode // Proceedings of the Fifth International Conference on Parallel and Distributed Processing, Techniques and Applications (PDPTA'98, Las Vegas, 1998). — CSREA Publishers, 1998.
72. Sacerdoti F. D. Wide area cluster monitoring with ganglia / F. D. Sacerdoti, M. J. Katz, M. L. Massie, and D. E. Culler // In Proceedings of the IEEE Cluster 2003 Conference (1–4 December 2003, Kowloon, Hong Kong). — P. 289–299.
73. Snoeren A. Mesh-based content routing using XML / A. Snoeren, K. Conley, D. Gifford // Operating System Review. — 2001. — Vol. 5, No. 35. — Banff: ACM. — P. 160–173.
74. Sottile M. J. Supermon: A high-speed cluster monitoring system / M. J. Sottile and R. G. Minnich // In Proceedings of the IEEE conference on cluster computing (2002). — P. 39–46.
75. Spafford E. H. Data Collection Mechanisms for Intrusion Detection Systems. Tech. Rep. 2000-08 / E. H., Spafford, D. Zamboni. — West Lafayette: CERIAS, 2000. — 6 p.
76. Sterling T. BEOWULF: A Parallel Workstation for Scientific Computation / T. Sterling, D. J. Becker, D. Savarese, J. E. Dorband, U. A. Ranawake,

- C. V. Packer // Proceedings of ICPP95 (August 14-18, 1995, Urbana-Champaign). — P. 11–14.
77. Sunderam V. S. PVM: A Framework for Parallel Distributed Computing / V. S. Sunderam // Concurrency: Practice and Experience. — 1990. — Vol. 2, No. 4. — P. 315–339.
 78. Sun Microsystems Solstice SyMON 1.1 User's Guide. — Palo Alto: Sun Microsystems Computer Company, 1996. — 186 p.
 79. Gosling J. Java language environment a white paper / J. Gosling, H. McGilton. — Palo Alto: Sun Microsystems Computer Company, 1996. — 56 p.
 80. Terfloth K. Self-sustained Routing for Event Diffusion in Wireless Sensor Networks. Lecture Notes in Computer science / K. Terfloth, J. Schiller. — 2008. — vol. 5321/2008. — Berlin: Springer. — P. 236-241.
 81. Tierney B. A Network-Aware Distributed Storage Cache for Data Intensive Environments / B. Tierney, J. Lee, B. Crowley, M. Holding, J. Hylton, F. Drake // Proceeding of IEEE High Performance Distributed Computing conference (HPDC-8, August 1999). — Redondo Beach: IEEE Computer Society Press, 1999. — P. 185–193.
 82. Tierney B. A Monitoring Sensor Management System for Grid Environments / B. Tierney, B. Crowley, D. Gunter, M. Golfing, J. Lee, M. Thompson // Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing. — Washington: IEEE Computer Society, 2000. — P. 97.
 83. Tierney B. The NetLogger Methodology for High Performance Distributed Systems Performance Analysis / B. Tierney, W. Johnston, B. Crowley, G. Hoo, C. Brooks, D. Gunter. // Proceeding of IEEE High Performance Distributed Computing conference (July 1998, Chicago). — P. 260–267.
 84. Thompson M. Certificate-based Access Control for Widely Distributed Resources / M. Thompson, W. Johnston, S. Mudumbai, G. Hoo, K.

- Jackson, A. Essiari // Proceedings of the Eighth Usenix Security Symposium (Aug. 1999, Washington). — P. 215–227.
85. Uthayopas, P. Parallel Image Velocity Estimation using Workstation Clusters / P. Uthayopas, B. Patwiwat // Proceeding of 5th Annual Conference on Technologies Transfer between Thai Researchers in North America and Thailand (Edmonton, 1996).
 86. Poovarawan Y. High Performance Computing: Needs and Application for Thailand / Y. Poovarawan, P. Uthayopas // In Proceeding of the Third International Conference/Exhibition on High Performance Computing in Asia-Pacific Region (HPC ASIA'99, Bangkok, November 1997).
 87. Uthayopas P. Beowulf Class Cluster: Opportunities and Approach in Thailand / Presented in First NASA Workshop on Beowulf Class Computer Systems (Pasadena, October 1997). — Pasadena: NASA Jet Propulsion Laboratory, 1997.
 88. Uthayopas P. Building a resources monitoring system for SMILE Beowulf cluster / P. Uthayopas, S. Phaisithbenchapol, K. Chongbarirux // Proceedings of the Third International Conference/Exhibition on High Performance Computing in Asia-Pacific Region (HPC ASIA, Singapore, 1998).
 89. Wahl M. Lightweight Directory Access Protocol (v3) / M. Wahl, T. Howes, S. Kille // IETF RFC 2251, 1997. — 12 p.
 90. Wolski R. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing / R. Wolski, N. Spring and J. Hayes // Journal of Future Generation Computing Systems. — 1999. — Vol. 15, No. 5-6. — P. 757–768.
 91. Wolski R. White Paper: Developing a Dynamic Performance Information Infrastructure for Grid Systems / R. Wolski, M. Swamy, S. Fitzgerald. // GridForum, 2000.
 92. Clark J. XML Path Language (XPath). Version 1.0 / J. Clark, S. DeRose // W3C Recommendation, 1999. — 30 p.

93. Yingquan W. Distributed Non-Concurrent Fault Identification in Discrete Event Systems / W. Yingquan, N. H. Christoforos // Proceedings of Multiconference on Computational Engineering in Systems Applications. — Lille, 2003.
94. CORBA Systems Management: Event Management Service. X/Open Document Number: P437 // The Open Group, 1997. — 258 p.

Глоссарий

Вещание (англ. streaming) — механизм пассивного опроса данных, при котором сервер рассылает данные независимо от того, есть ли заинтересованные в них клиенты.

Виртуализация (англ. virtualization) — механизм поддержки нескольких независимо работающих копий операционных систем, возможно одновременно на одном аппаратном обеспечении. Виртуализация позволяет более гибко разделять ресурсы между различными сервисами. Программное обеспечение, управляющее работой таких копий и обеспечивающее их взаимодействие с реальным оборудованием, называется **гипервизором** (англ. hypervisor)

Виртуальная машина Java (Java Virtual Machine, JVM) — программное обеспечение, реализующее абстрактную машину, исполняющую байт-код программ, написанных на языке программирования Java.

Виртуальный параллельный компьютер (Parallel Virtual Computer) — программная среда, запускаемая на вычислительном кластере и моделирующая гомогенную вычислительную систему.

Выделенный узел — одна из вычислительных станций, которой делегированы функции, отличные от вычислений. Иногда на таком узле вычисления запрещены.

Высокопроизводительные вычисления (HPC, High Performance Computing) — прикладная область вычислительной математики, занимающаяся вычислительными алгоритмами, требующих больших затрат машинного времени. В настоящее время высокопроизводительные вычисления в основном производятся на вычислительных

системах, предполагающих распараллеливание вычислительного алгоритма.

Вычислительный кластер — совокупность вычислительных станций, которые в процессе вычислений обмениваются данными посредством среды передачи данных (коммуникационной среды). Наиболее важной характеристикой ВК является вычислительная производительность, измеряемая в операциях с плавающей точкой в секунду (flops, floating-point operations per second) С точки зрения пользователя кластер представляет собой единый программно-аппаратный ресурс. По наиболее известной классификации компьютерных архитектур Флинна такие вычислительные системы относятся к MIMD системам. В рукописи под термином кластер, если не указано иное, понимается вычислительный кластер на базе неспециализированных компонент, т.н. Beowulf-кластер

Вычислительный процесс — процесс решения задачи в вычислительной системе, включающий трансляцию исходной программы во внутреннее представление компьютера, возможно параллельного, пошаговую интерпретацию и выполнение необходимых инструкций.

Вычислительные ресурсы — аппаратные и программные средства для решения прикладных задач. К ним относятся вычислительные узлы, коммуникационная среда, прикладное программное обеспечение и системы хранения данных.

Вычислительный узел (host) — вычислительная машина, являющаяся составной частью вычислительного кластера. Вычислительный узел интегрирует ряд вычислительных ресурсов. Число узлов — это одна из характеристик кластера, определяющих его производительность и особенности при вычислении задач различного типа. Узлы кластера разделяются между различными заданиями, конкурирующими за ресурсы ВК.

В процессе работы пользователь воспринимает кластер как единое целое и составляет программы, получающие искомым результат на

некотором подмножестве узлов кластера, необязательно одинаковом между различными запусками задачи.

Гетерогенность — разнородность вычислительных ресурсов входящих в состав вычислительного комплекса.

Гомогенность — обратна гетерогенности, единообразие вычислительных ресурсов в составе вычислительного комплекса.

Грид (GRID) — распределённая вычислительная среда, структурными единицами которой являются вычислительные кластера и другие системы, разнесённые географически или логически. Гриды позволяют ставить задания на выполнение и получать результат единообразным способом при работе с любым из кластеров, входящих в грид. При этом грид берёт на себя все функции по доставке задания на кластер, постановку в очередь заданий на этом кластере, отслеживание выполнения задания и возвращение результатов пользователю.

демон — в UNIX-подобных операционных системах выполняемая в фоновом режиме программа.

Задание (job) — программа пользователя с описанием необходимых ей для выполнения ресурсов и способа вывода результатов. Большинство выполняемых на кластере заданий являются параллельными приложениями, однако в ряде случаев могут быть и непараллельными. Основные ресурсы, описываемые при определении задания: объём оперативной памяти, общее число распределяемых по вычислительным узлам процессов, свободное дисковое пространство. Наряду с термином задание, в литературе также встречается термин задача.

Число процессоров, объём памяти на вычислительном узле и логическая организация передачи данных влияет на эффективность работы программы пользователя, поэтому при постановке задания в очередь пользователь должен указать такой набор ресурсов, при использовании которых скорость выполнения задания будет наиболее высока для конкретной вычислительной системы.

Запрос данных (query) — механизм активного и часто синхронного опроса сервера, при котором клиент самостоятельно запрашивает у сервера требуемые данные.

Клиент-серверная архитектура — такая архитектура программного комплекса, при котором выделяется отдельное приложение, называемое сервером, в задачи которого входит обслуживание (предоставление каких-либо сервисов) множеству приложений, называемых клиентами посредством определённого программного интерфейса.

Коммуникационная среда — см. среда передачи данных

Крупноблочный параллелизм — подход к реализации параллельных вычислений с редкими обменами промежуточными результатами. При этом параллельными фрагментами являются отдельные задачи, информационно слабо связанные между собой.

Латентность — время между началом передачи данных с источника данных до прибытия первого байта в точку назначения.

Масштабируемость — способность вычислительной системы увеличивать вычислительную мощность без существенного снижения отношения общей производительности к числу узлов при включении в состав вычислительной системы дополнительных узлов.

Мелкозернистый параллелизм — подход к реализации параллельных вычислений, характеризующийся большим количеством параллельно выполняемых, нередко однотипных, вычислений. Характерной чертой мелкозернистого параллелизма является примерно одинаковая интенсивность вычислений и обменов информацией.

Метрика — какая-либо величина, контролируемая системой мониторинга.

Мониторинг вычислительной системы — процесс наблюдения за состоянием ресурсов вычислительной системы, предполагающий

достоверность представляемых данных пользователю за определённый промежуток времени работы системы.

Мультивещание (multicasting) — режим передачи пакетов в сети таким образом, что их получают подключившиеся к каналу мульти-вещания приложения.

Очередь заданий — механизм управления заданиями, при котором формируется набор заданий, ожидающих выполнения. При наличии свободных ресурсов планировщик заданий выбирает и ставит на выполнение простаивающее задание, для которого достаточно свободных ресурсов. Ресурсы маркируются как занятые до тех пор, пока задание не будет выполнено.

Параллельное приложение — приложение, реализующие какой-либо вычислительный алгоритм таким образом, что приложение состоит из нескольких ветвей, или потоков, или процессов, или задач, выполняющихся одновременно.

Планировщик заданий — программный комплекс, управляющий запуском заданий из очереди заданий. На вычислительных кластерах наиболее эффективное использование ресурсов осуществляется в том случае, если ресурсы вычислительных узлов монопольно используются одним заданием, а другие задания выполняются на других свободных узлах. Это исключает накладные расходы на блокировку ресурсов и другие вопросы синхронизации процессов. Планировщик заданий реализует механизм запуска заданий из очереди на выполнение таким образом, чтобы конкуренция за ресурсы между вычислительными процессами в рамках различных заданий отсутствовала во время выполнения.

Прикладное программное обеспечение — программное обеспечение, состоящее из отдельных прикладных программ и пакетов прикладных программ, предназначенных для решения задач пользователей в какой-либо предметной области, а также автоматизированных систем, созданных на основе этих прикладных программ.

Пропускная способность — количество единиц информации, передаваемое или принимаемое в единицу времени.

Распределенные вычисления — способ решения ресурсоемких вычислительных задач с использованием более одной вычислительной станции, объединенных средой передачи данных в распределённой вычислительной среде.

Распределенная вычислительная среда — совокупность распределенных вычислительных станций, коммуникационной сети с, возможно, низкими характеристиками пропускной способности, промежуточного программного обеспечения, предназначенного для управления процессом решения задач в узлах этой вычислительной среды.

Ресурсоемкая задача — задача вычислительного характера, решение которой требует значительных затрат вычислительных ресурсов.

Свободное программное обеспечение (Open source software, OSS) — программное обеспечение, при использовании которого пользователю предоставлена свобода на изучение, распространение и изменение программы при условии соблюдения условий свободных лицензий. Одним из частых условий является невозможность закрытия любых модификаций исходного кода, основанного на ранее открытом исходном коде.

Сенсор — программный или аппаратный компонент предоставляющий данные для одной или более метрик.

Система мониторинга — программное или программно-аппаратное обеспечение, позволяющее производить мониторинг вычислительной системы.

Системное программное обеспечение — это комплекс программного обеспечения, которое управляет аппаратными ресурсами вычислительной системы, предоставляя доступ к ним прикладному ПО пользователя.

Среда передачи данных — программно-аппаратный комплекс, позволяющий параллельному заданию обмениваться данными между различными исполняющимися процессами этого задания. Средой передачи данных может выступать локальная вычислительная сеть на базе промышленных стандартов Ethernet, Infiniband, Myrinet и др. К программной компоненте относится интерфейс обмена сообщениями, например, интерфейс обмена сообщениями MPI (англ. Message Processing Interface). К аппаратной компоненте относится технологическое решение для обмена данными на физическом уровне (число соединений между узлами, вид соединения и другие характеристики).

Триггер — логический элемент системы мониторинга, содержащий условие или выражение, истинность которого необходимо проверить для последнего значения метрики.

Управляющий узел — выделенный узел, обладающий функциями управления заданиями пользователей и ресурсами вычислительной системы.

Application Programming Interface (API) — Интерфейс прикладного программирования. Набор готовых структур и функций, используемых при программировании пользовательских приложений и обеспечивающих правильное взаимодействие между пользовательским приложением и какой-либо средой, например, операционной системой.

Beowulf-кластер — общее название кластеров на неспециализированных компонентах, работающих под управлением операционных систем, распространяемых в открытом коде. Beowulf-кластера масштабируемы путём увеличения числа входящих в него узлов.

Multiple Instructions Multiple Data (MIMD) — Вычислительная система со множественным потоком команд и данных. Один из типов вычислительных машин в классификации Флинна.

Request For Comments (RFC) — открытые стандарты в области развития и работы сетевых протоколов. Текст стандарта разрабатывается инициативной группой по развитию какой-либо технологии и выкладывается в свободный доступ.

Приложение А. Подробное описание основных классов Grate

Пакет `grate.core`

класс `GPath` содержит следующие методы:

`up()` — подняться на уровень выше в иерархии.

`cutRoot()` — убрать из пути самый верхний элемент.

`getRoot()` — получить имя корневого элемента пути.

`toString()` — преобразовать путь в строку.

`add()` — добавить к пути ещё один элемент (спуститься на уровень ниже).

`getLast()` — получить последний элемент пути.

`getPath()` — получить путь без последнего элемента.

```
public interface GTimeline {
    public boolean placeTimeline(long timestamp);
    public int checkTimeline(long timestamp);
    public int getPosForTime(long timestamp);
    public long getTimestampAt(int num);
    public long getMinTimeStamp();
    public long getMaxTimeStamp();
    public long getTimeDelta();
    public void killTimeFirst();
    public void extendMaxTimeTo(long timestamp);
    public int getSize();
}
```

`placeTimeline()` — возвращает значение «ложно», если пакет данных с указанным временем по каким-либо причинам нельзя добавить.

`checkTimeline()` — проверяет, возможно ли добавить указанную вре-

менную метку в данный объект.

`getPosForTime()` — возвращает индекс элемента в метрике, который соответствует значению на указанный момент времени.

`getTimestampAt()` — возвращает временную отметку для указанного индекса.

`getMinTimeStamp()` и `getMaxTimeStamp()` — возвращают соответственно минимальную и максимальную временную отметку, хранимую в данном объекте.

`getTimeDelta()` — возвращает разницу между последней помещённой отметкой и предпоследней, т.е. шаг по времени, который может произвольно изменяться.

`killTimeFirst()` — уничтожает первую метку в контролируемой последовательности временных меток.

`extendMaxTimeTo()` — расширяет последовательность новой временной меткой. Используется, если полученное от источника данных значение эквивалентно последнему значению метрики.

```
public interface GWorld {
    public GNode      getRoot ();
    public GNode      createNode (String name);
    public GMetric     createMetric (GNode owner, String name,
                                     String type, String units);
    public void        createNewConnection ();
    public void        commit ();
    public void        setOption (String opt, String val);
    public String      getOptionString (String opt);
    public GMetricCacheman getCacheman ();
}
```

`getRoot()` — возвращает корневой узел иерархии узлов.

`createNode()`, `createMetric()`, `createNewConnection()` — создают узел, метрику и соединение соответственно. Применение централизованных функций позволяет при необходимости вести статистику использования памяти, числа метрик, и прочее.

`commit()` — вызывается после обработки очередного кадра. Класс, реализующий данный интерфейс сам определяет действия, которые связаны с данным событием.

`getCacheman()` — возвращает объект, используя который метрика может сохранять данные во внешнее хранилище данных.

`setOption()` и `getOptionString()` используются для сохранения и получения каких-либо глобальных параметров, задаваемых пользователем.

Пакет `grate.connection`

```
public interface GConnection {
    public void          tune ();
    public Object        connect ();
    public InputStream   getInputStream (Object obj);
    public OutputStream getOutputStream (Object obj);
    public void          disconnect (Object obj);
    public String        getSourceHost (Object obj);
}
```

`tune()` обычно вызывается приложением сразу после создания экземпляра класса, наследующего этот интерфейс, для настройки параметров соединения.

`connect()` вызывается приложением, когда возникает необходимость открыть новый сеанс для получения данных. Возвращаемый объект уникальным образом идентифицирует сеанс в рамках соединения и должен использоваться в дальнейшей работе с сеансом. Конкретный класс, реализующий данный интерфейс сам определяет наиболее удобный класс для представления объекта идентификации сеанса.

`getInputStream()` — возвращает объект, реализующий интерфейс `InputStream`, который и является поставщиком данных в течение сеанса.

`getOutputStream()` — возвращает объект, реализующий интерфейс `OutputStream`, который используется в специфических случаях, для формирования конкретных запросов к источнику данных.

`disconnect()` — закрывает сеанс, связанный с переданным объектом. Полученные ранее классы потоков данных становятся недействительным.

`getSourceHost()` — используется для получения адрес источника данных, с которым открыт сеанс. В общем случае следует избегать использования данного метода и получать подобную информацию разбирая входной поток. Использование метода оправданно, когда подобную информацию невозможно получить иным путём (например при получении UDP пакета с данными Ganglia в формате XDR).

```
public interface GSentinel {
    public void process ();
    public void stopSentinel ();
    public void startSentinel ();
    public void abortSentinel ();
}
```

Пакет grate.frame

```
public interface GFrame {
    public void setInputStream (InputStream stream );
    public boolean isValid ();
    public void parse (GNode node );
    public void parse (GNode node ,
                      GConnection con , Object obj );
}
```

`setInputStream()` — используется для установки входного потока данных.

`isValid()` — проверяет, являются ли полученные данные теми, которые ожидал получить объект. Если возвращаемое значение `false`, то кадр не обрабатывается. Этот метод по умолчанию возвращает `true`, если такую проверку невозможно осуществить. `parse()` осуществляет обработку полученных данных и обновление данных узлов и метрик с корнем, указанным в параметре. Метод с тремя параметрами по умолчанию вызывает `parse()` с одним параметром и должен использоваться лишь в тех случаях, где без

подобного подхода обойтись невозможно.

Класс `GFrameBase`, дополнительно реализует два метода:

`getDescription()` — получение краткой информации о типе кадра.

`setInputStream()` — установка соединения.

Пакет `grate.metric`

```
public interface GMetric {
    public void placeData(Object obj, long timestamp_s)
        throws GAddDataFaultException;
    public void placeStringData(String str, long timestamp_s)
        throws GAddDataFaultException;
    public G_type[] getLastData(int count);
    public G_type[] getData(int count, long timestamp_start,
                           long timestamp_finish);
    public G_type    getLastValue();
    public G_type    getData(int i);
    public int       getDataCount();
    public String    getUnits();
    public String    getType();
    public Class     getTypeClass();
    public long      getDomain();
    public long      getLastTime();
    public String    getName();
    public GNode     getOwner();
    public G_type    getMinValue();
    public G_type    getMaxValue();
    public String    getMetricPath();
}
```

`placeData()` — сохранить данное значение в метрике. Не рекомендуется использовать напрямую пользователем.

`placeStringData()` — аналогично предыдущему, но значение передаётся в виде строки, которая преобразуется к необходимому типу данных, используемому данной метрикой.

`getLastData()` — возвращает указанное число последних значений. Если общее число значений, хранимых в метрике меньше переданного, возвращается столько, сколько возможно.

`getData()` — возвращает указанное число значений из указанного временного промежутка или единственное значение по индексу. Индекс растёт в порядке помещения данных в метрику (но возможно не в порядке увеличения времени).

getLastValue() — получить последнее сохранённое значение.

`getDataCount()` — получить число хранимых значений.

getUnits() — получить тип пользовательского представления данных.

getType() — получить строковое представление типа.

`getClass()` — получить класс, объекты которого хранят значения для данной метрики.

`getDomain()` — возвращает тип значений хранимых в объектах значений, см. интерфейс `GType`.

getLastTime() — возвращает время последнего индекса метрики.

getName() — возвращает название метрики.

getOwner() — возвращает узел-владелец метрики.

getMinValue() — возвращает минимальное значение хранимое в метрике.

getMaxValue() — возвращает максимальное значение хранимое в метрике.

`getMetricPath()` — возвращает путь к метрике. Путь — это строка, которая однозначно идентифицирует метрику в иерархии узлов.

```
public interface GMetricCacheman {
    public boolean    cache(GMetric metr, GType value,
                           long timestamp);
    public boolean    batchCache(GMetric metr, GType[] values,
                                long[] timestamps);

    public GType      get(GMetric metr, long timestamp_s);
    public GType[]    batchGet(GMetric metr,
                               long timestamp start,
```

```

        long timestamp_end);
    }

```

cache() — передать значение на хранение. Возвращает значение «ложь», если операция не удалась.

get() — получить значение из хранилища, соответствующее указанному времени. batchCache() и batchGet() выполняют те же операции, но для массива значений.

Пакет grate.node

```

public interface GNode {
    public void      addMetric(GMetric metr);
    public boolean   hasMetric(String name);
    public GMetric   getMetric(String name);
    public GMetric   getMetric(int i);
    public GMetric   getMetric(GPath path);
    public GMetric[] getMetricArray();
    public int       getMetricCount();
    public void      updateAttribute(String attr,
                                     String value);

    public String    getAttribute(String attr);
    public String    getName();
    public GTimeline getTimeline();
    public boolean   addNode(GNode node);
    public GNode     getParent();
    public GNode     getNodeByName(String node_name);
    public GNode     getNode(GPath path);
    public LinkedList getChildNodes();
    public void      setParent(GNode parent);
    public GPath     getPath();
}

```

Все методы можно разделить на три группы: методы для работы с метриками (получение списка, метрики, добавление и обновление метрик

в узле), методы для работы с иерархической структурой узлов, другие методы.

`addMetric()` — добавить новую метрику к узлу.

`getMetric()` — получить метрику по названию, индексу или относительному пути.

`addNode()` — добавить дочерний узел к данному узлу.

`getChildNodes()` — получить список дочерних узлов.

`getParent()` — получить родительский узел.

`getPath()` — получить путь в иерархии структур.

`getTimeline()` — получить объект реализующий интерфейс `GTimeline`, который отвечает за соответствие индексов метрики и времени.

`getAttribute()` и `updateAttribute()` — соответственно, для получения и обновления данных атрибута.

Пакет `grate.plate`

```
public interface GPlate {
    boolean    addMetric(GMetric metric);
    void       notifyData();
}
```

`addMetric()` — добавить метрику в область.

`notifyData()` — вызывается внешними классами чтобы уведомить об изменении состоянии метрики (обычно — добавление данных)

Пакет `grate.graphic`

```
public interface GGraphic {
    public void    draw(GGraphContext context);
    public void    setMetric(GMetric metric);
    public GMetric getMetric();
    GType          getMinValue();
    GType          getMaxValue();
    public long    getType();
    public void    tune();
    public boolean isCompatibleWith(GGraphic other);
}
```

```
}
```

`draw()` — вызывается для отображения графика

`setMetric()` — связать объект графика с конкретной метрикой.

`getMetric()` — получить объект метрики.

`getType()` — получить тип графика. Графики делятся на одноментные графики (`G_GRAPH_SNAPSHOT`); графики, требующие всей рабочей области (`G_GRAPH_LONEWOLF`); графики, которые могут отрисовывать целочисленные, вещественные, строковые и неопределённые значения (`G_GRAPH_INT`, `G_GRAPH_FLOAT`, `G_GRAPH_STRING`, `G_GRAPH_UNKNOWN` соответственно). Подробнее о типах данных — см. пакет `grate.types`).

`tune()` — используется для настроек характеристик графика (цвет, толщина линий и т.п.) `isCompatibleWith()` — проверяет совместимость данного графика с указанным. В частности проверяется тип отображаемых значений, тип графика и прочее, определяемое реализацией интерфейса.

`GGraphicBase` добавляет следующие методы:

`adjustY()`, `adjustX()`, `adjustXScale()`, `adjustYScale()` — используемые для настройки масштабов вывода. Последние две функции рекомендуется переопределять, первые две — оставлять неизменными.

Пакет `grate.types`

```
public interface GType {
    public void      setValue (String str);
    public void      setDefault ();
    public double    toDouble ();
    public float     toFloat ();
    public long      toLong ();
    public int       toInt ();
    public String    toString ();
    public long      getDomain ();
    public boolean   isSet ();
    public int       compareTo (GType val);
    public void      getFromXDR (InputStream ins)
```

```

                                throws IOException;
    public void                putToXDR(OutputStream ous)
                                throws IOException;
}

```

setValue() — установить значение.

toDouble(), toLong(), toString() — используются для преобразования значения типа к базовым типам языка Java, double, long, String соответственно.

toFloat(), toInt() — используются для преобразования значения к базовым типам float и int.

getDomain() — используется для получения типа хранимых значений, это может быть:

- G_UNKNOWN — неизвестный тип;
- G_NUMERIC_INT — поддерживающий приведение к целочисленному типу;
- G_NUMERIC_FLOAT — поддерживающий приведение к вещественному типу;
- G_STRING — поддерживающий приведение к строковому типу;
- G_BINARY — зарезервировано для будущего использования;
- G_EXTENDED_TYPE — расширенный тип (например временная метка);
- G_LONG — модификатор точности представления данных.

setDefault() — инициализировать объект значением по умолчанию.

isSet() — возвращает «истина», если значение для данного экземпляра класса установлено.

compareTo() — сравнить с другим значением, если это возможно. Если выполнение сравнения невозможно, поведение типа не определено.

getFromXDR() и putToXDR() — предназначены для вывода значений типа в/из указанного потока в стандарте XDR.

Пакет `grate.trigger`

```
public interface GTrigger {
    public void chainReactor(GReactor reactor);
    public int check(GMetric metric, int op);
    public boolean isSignaled();
    public void setSignaled(boolean type);
}
```

`chainReactor()` добавляет объект действия в очередь действий, которая будет выполнена при выполнении условий триггера.

`check()` вызывается метрикой, для которой установлен данный триггер, чтобы проверить выполнение условий. В качестве параметра `op` передаётся одно из следующих:

- `TRIGGER_OP_UNKNOWN` — произошла неизвестная (неопределённая) операция.
- `TRIGGER_DATA_ADD` — данные добавлены в метрику.
- `TRIGGER_DATA_CACHE` — данные помещены в хранилище данных.
- `TRIGGER_METRIC_EVOLUTION` — метрика сменила свой тип.

`isSignaled()` вызывается для проверки, были ли условия триггера выполнены. Эта функция используется для переключаемых триггеров в целях определения их текущего состояния (о переключаемых триггерах см. раздел 2.3 диссертации). Для триггера, условия которого выполнены, функция вернёт значение `true`.

`setSignaled()` устанавливает тип триггера (сбрасываемый – `false`, переключаемый – `true`). Тип триггера обычно устанавливается при его создании, но может быть изменён в процессе работы, через данный метод.

Пакет `grate.reactor`

```
public interface GReactor {
```

```
public void react(GTrigger trigger , GMetric metric);  
public void tune(GMetric metric);  
}
```

react() вызывается триггером и выполняет действие, которое реализовано конкретным классом.

tune() используется для возможной предварительной инициализации класса (например, уведомление по почте может позволить изменить шаблон сообщения, адресата и прочие параметры).

Приложение Б. Пример файла настроек grated

```

reporter { // используется для получения отчётов
    id="mainReporter" // идентификатор этого объекта отчётов
    type="GReportTCPServer" // тип объекта отчётов
    report-type="GOutputGangliaXML" // формат выходных данных
    // параметры объекта отчётов в зависимости от типа
    report-port=6789 // порт для входящих TCP соединений
}

cacheman { // объект хранения данных
    id="mainCacheman" // идентификатор объекта
    type="GCachemanEater" // тип объекта
}

connection { // соединение для получения данных
    id="connection1" // идентификатор этого соединения
    type="GConnectionMcast" // тип соединения

    // параметры соединения в зависимости от типа
    channel="239.2.11.71" // канал мультитевещения
    port=8649 // порт
    max-packet-size=512 // максимальный размер пакетов
    recieve-buffer=32768 // оазмер принимающего буфера
}

sentinel { // объект для фонового обновления данных
    id="sentinel1" // идентификатор объекта
    type="GSentinelSleepless" // тип
    producer="connection1" // соединение-поставщик данных
    consumer="GFrameGangliaXDR" // потребитель данных
    timeout=5 // тайм-аут между итерациями получения данных
    iterations=0 // число итераций
    // (в данном случае - бесконечное)
}

cluster { // информация о кластере
    id="cluster1" // идентификатор объекта кластера
    name="KVC1" // имя кластера
    owner="CC FEB RAS" // владелец кластера
    url="http://cluster.as.khb.ru" // домашняя страница

```

```

    latlong="unspecified"      // географические широта/долгота
}

grid {
    id="grid1"                // идентификатор РВС
    name="CC FEB RAS grid"    // имя РВС
    authority="http://cluster.as.khb.ru" // домашняя страница
    clusters="cluster1"       // список входящих кластеров
}

// пример использования триггеров

reactor {
    id="overheat"             // идентификатор объекта-действия
    type="GReactorBase"       // тип
    add-message= "Overheat of {NODE} ({METRIC_VALUE} Celsius). "
    // сообщение будет отправлено в лог, а выражения
    // {NODE} и {METRIC_VALUE} будут заменены
    // соответствующими значениями
}

trigger {
    id="tempTrigger"          // идентификатор триггера
    signaled = true;           // переключаемый тип триггера
    type="GTriggerMore"       // тип «больше»
// с каким значением сравнивать
    value=65
// какое действие выполнить при выполнении условия
    reactors="overheat"
}

crigger {
    id="Temperature checker"   // мета-триггер
    set=".* /CPU_temperature"  // фильтр метрик
    triggers="tempTrigger"     // какие триггеры создавать
}

```

Данный файл конфигурации позволяет запустить `grated` с набором триггеров, использованных в первом примере третьей главе для контроля температурного режима вычислительного кластера.

При отсутствии метрики `CPU_temperature` в данных, передаваемых источниками данных СМ, можно формировать её вручную с помощью команды `grate_metric` в виде:

```
grate_metric -t <тип> -m <имя метрики> -v <значение>
```

Поддерживаются такие типы как: знаковое и беззнаковое целое, стро-

ковое значение и числа с плавающей запятой. В имени метрики не должно быть пробелов.

Например, для передачи данных о текущем значении метрики температуры можно использовать следующую команду:

```
grate_metric -t float32 -m CPU_temperature -v 56.2
```

Такая команда передаст в канал группового вещания указанное значение (56.2) для метрики CPU_temperature. Параметры канала группового вещания и другие необходимые параметры берутся из конфигурационного файла grated.conf.

Приложение В. Системы виртуализации

Виртуализация — механизм поддержки нескольких независимо работающих копий операционных систем, возможно одновременно, на одном аппаратном обеспечении. Иными словами, происходит абстракция вычислительных ресурсов через предоставление пользователю стандартного программного интерфейса, что позволяет скрыть используемое оборудование и делегировать доступ к аппаратному обеспечению.

С ростом аппаратных мощностей, развитием многопроцессорных и многоядерных ВК в 90-х годах прошлого столетия стали очевидны перспективы подхода виртуализации, поскольку появилась возможность использования нескольких виртуальных машин на одной физической платформе.

Выделяют два класса виртуализации.

- Виртуализация платформ. Программная абстракция любого вида, запускаемая на реальной программно-аппаратной системе.
- Виртуализация ресурсов. Цель этого вида виртуализации — упрощение представления аппаратных ресурсов для пользователя, например, реализацией блочного устройства хранения данных может являться распределённая файловая система. В рамках нашей работы данный вид виртуализации не представляет интереса и не рассматривается.

Под виртуализацией платформ понимают создание программных систем на основе существующих аппаратно-программных комплексов. Система, предоставляющая аппаратные ресурсы и программное обеспечение, называется хостовой, а абстрагируемые ей системы — гостевыми. Чтобы гостевые системы могли стабильно функционировать на платформе хостовой системы, необходимо, чтобы программное и аппаратное обеспечение хостовой системы предоставляло набор интерфейсов для доступа к его ресурсам.

Основные виды виртуализации платформ зависят от того, насколько полно осуществляется сокрытие аппаратного обеспечения.

- Полная эмуляция (Bochs¹, VirtualBox²). При таком подходе полностью виртуализуется все аппаратное обеспечение, сохраняя гостевую ОС в неизменном виде. Это позволяет эмулировать различные аппаратные платформы на хостовых системах с отличной архитектурой. Однако полностью эмулируемое аппаратное обеспечение существенно снижает быстродействие гостевой системы.
- Частичная эмуляция (VMWare³, VirtualBox). Виртуализуется лишь аппаратное обеспечение, необходимое для запуска гостевой ОС изолированно. Частичная эмуляция требует, чтобы гостевая и хостовая ОС были разработаны для одной аппаратной архитектуры. Этот вид виртуализации значительно производительнее полной эмуляции и широко используется в настоящее время. В целях повышения быстродействия применяется гипервизор — ПО, позволяющее гостевой системе напрямую обращаться к ресурсам аппаратного обеспечения.
- Виртуализация адресного пространства (User-mode Linux⁴). Такой вид виртуализации позволяет совместно использовать ресурсы и изолировать процессы, но не позволяет разделять экземпляры гостевых ОС, поскольку происходит изоляция процессов пользователя на уровне операционной системы.
- Паравиртуализация (Xen⁵). При таком подходе нет необходимости эмулировать аппаратное обеспечение, однако, используется специальное API для взаимодействия с гостевой ОС, что требует модификации её кода. ПО паравиртуализации также имеет свой гипервизор.
- Виртуализация уровня ОС (FreeBSD jails⁶, Solaris containers⁷). Выполняется виртуализация физического сервера на уровне ОС в це-

¹<http://bochs.sourceforge.net/>

²<http://www.virtualbox.org/>

³<http://www.vmware.com/>

⁴<http://user-mode-linux.sourceforge.net/>

⁵<http://www.xen.org/>

⁶<http://www.freebsd.org/doc/handbook/jails.html>

⁷<http://www.sun.com/software/solaris/ds/containers.jsp>

лях создания нескольких защищенных виртуализованных серверов на одном физическом. Гостевая система, в данном случае, разделяет использование одного ядра хостовой ОС с другими гостевыми системами.

В настоящее время виртуализация широко используется для консолидации серверов, что позволяет распределять нагрузку, перенося её с нескольких физических серверов, ресурсы которых используются на 10-20 процентов, на один сервер с гостевыми ОС и общей нагрузкой 80-90 процентов.

Преимуществами использования виртуализации являются возможность:

- поддержки старых ОС в целях совместимости;
- изолирования потенциально небезопасного окружения и процессов;
- ограничения использования ресурсов;
- более полного использования оборудования;
- более полной управляемости (создания резервных копий, слепков состояния) и восстановления после сбоев.

Основным недостатком являются расходы аппаратных ресурсов на поддержку виртуализации. Именно влияние этого недостатка на производительность системы исследовалось в процессе тестирования CM Grate в задаче повышения эффективности использования ВК. Поскольку производительность систем паравиртуализации считается одной из лучших — в сравнении с другими типами виртуализации — а также то, что современные версии ядра Linux поддерживают гипервизор Xen, определило выбор данной системы для установки на экспериментальный кластер ВЦ ДВО РАН.

Приложение Г. Искусственные нейронные сети

Во многих прикладных задачах в настоящее время сложно получить точное решение в заданный промежуток времени. Для решения подобных задач были предложены различные методы, основывающиеся на аналогии с поведением сложных структур и объектов из живой природы, в частности: генетические алгоритмы, метод муравьиных колоний, искусственные нейронные сети и другие.

В *искусственных нейронных сетях* (ИНС) работа проводится с искусственными нейронами. ИНС реализуется с помощью электронных компонентов или моделируется программой.

ИНС обладают следующими важными свойствами:

- ИНС может дообучаться в процессе работы;
- для настройки ИНС применяются связи между нейронами, называемые *синаптическими весами*;
- нейронная сеть может изменять свою топологию в процессе обучения.

Целью обучения является определение синаптических весов нейронной сети для обеспечения такой структуры взаимосвязей нейронов, позволяющей получать результат для наборов входных данных, которым не обучена ИНС.

К значимым преимуществам ИНС для нашей работы, можно отнести:

- способность получать обоснованный результат для входных данных, которые не встречались в процессе обучения;
- адаптивность к конкретной задаче в области применения.

В модели искусственного нейрона можно выделить три основных элемента.

Первый — это набор синапсов или связей, каждый из которых характеризуется своим весом или силой. В частности, сигнал x_j на входе синапса j , связанного с нейроном k , умножается на вес ω_{kj} . Синаптический вес искусственного нейрона может иметь как положительные, так и отрицательные значения.

Второй элемент — это сумматор, представляющий линейную комбинацию входных сигналов, взвешенных относительно соответствующих синапсов.

Третий элемент — это функция активации, которая ограничивает амплитуду выходного сигнала нейрона. Обычно нормированный диапазон амплитуд выхода нейрона лежит на отрезках $[0, 1]$ или $[-1, 1]$.

В описываемой модели нейрона вводится пороговый элемент b_k . Эта величина отражает увеличение или уменьшение входного сигнала, подаваемого на функцию активации. Функционирование нейрона k описывается следующим образом:

$$u_k = \sum_{j=1}^n \omega_{kj} x_j, \quad (\Gamma.1)$$

$$y_k = \phi(u_k + b_k), \quad (\Gamma.2)$$

где:

$x_1, x_2, \dots, x_m$ — входные сигналы;

$\omega_{k1}, \omega_{k2}, \dots, \omega_{km}$ — синаптические веса нейрона k ;

u_k — линейная комбинация входных воздействий;

b_k — пороговый элемент;

$\phi(x)$ — функция активации нейрона;

y_k — выходной сигнал нейрона.

Использование порога b_k обеспечивает эффект аффинного преобразования выхода сумматора u_k . В модели, показанной на рисунке Г.1, постсинаптический потенциал вычисляется следующим образом:

$$\nu_k = u_k + b_k \quad (\Gamma.3)$$

Функции активации $\phi(\nu)$ в Г.2 определяют выходной сигнал нейрона в зависимости от индуцированного локального поля ν . В работах по ИНС выделяют следующие основные типы функций активации: функция

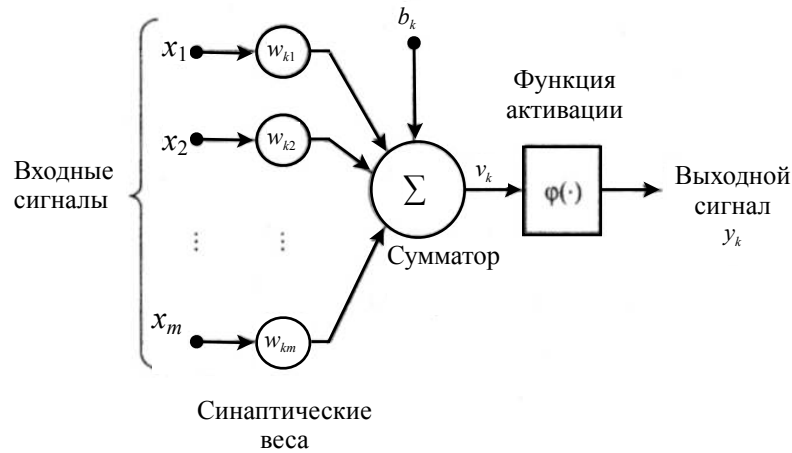


Рис. Г.1: Модель нейрона в ИНС

единичного скачка, кусочно-линейная функция и сигмоидальная [14, 29], которая использована в модуле ИНС в составе CM Grate.

Примером сигмоидальной функции может служить логистическая функция, задаваемая следующим выражением:

$$\phi(\nu) = \frac{1}{1 + \exp(-a\nu)}, \quad (\text{Г.4})$$

где a — параметр наклона сигмоидальной функции.

Область значений функций активации Г.4 обычно представляет собой отрезок от $[0, 1]$, реже — $[-1, 1]$.

Всем нейронным сетям присуща способность к обучению. Обучение — это процесс, при котором свободные параметры нейронной сети настраиваются посредством моделирования входных векторов данных из области применения этой сети.

Не существует универсального алгоритма обучения, подходящего для всех архитектур нейронных сетей. Методы обучения ИНС отличаются друг от друга способом настройки синаптических весов нейронов и способом связи обучаемой нейросети с «учителем» (если он используется). В работах по теории ИНС приведено подробное описание процессов обучения, на которых в своей работе не будем останавливаться подробно, заострив внимание лишь на отдельных существенных деталях.

Часто используемыми являются три фундаментальных класса нейросетевых архитектур: однослойные сети прямого распространения, многослойные сети прямого распространения и рекуррентные сети.

В ИНС существует входной слой узлов источника, информация от которого передается на выходной слой нейронов (*вычислительные узлы* в терминах ИНС). В простом случае такой слой является единственным, и ИНС называется *сетью прямого распространения* или ациклической сетью. Она является однослойной, при этом под единственным слоем подразумевается слой вычислительных узлов.

Существуют и другие классы нейронных сетей прямого распространения, характеризующиеся наличием одного или нескольких скрытых слоев, узлы которых называются скрытыми нейронами. Функция последних заключается в посредничестве между внешним входным сигналом и выходом нейронной сети. Узлы источника входного слоя сети формируют соответствующие элементы входного вектора, которые поступают на нейроны первого скрытого слоя. Выходной вектор N -го слоя используется в качестве входного для слоя $N + 1$.

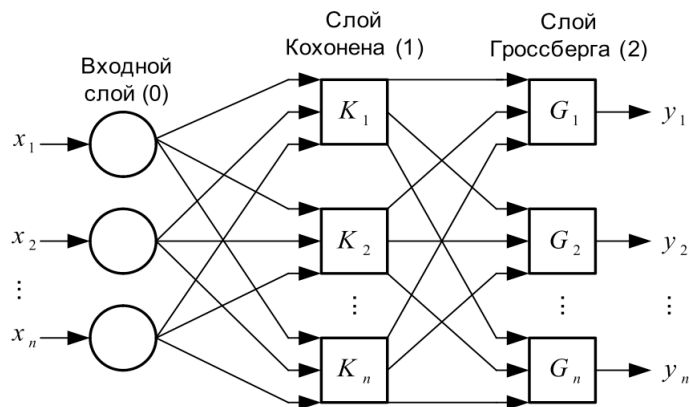


Рис. Г.2: Топология ИНС встречного распространения

Рекуррентная нейронная сеть отличается от сети прямого распространения наличием, по меньшей мере, одной обратной связи. Например, рекуррентная сеть может состоять из единственного слоя нейронов, каждый из которых направляет свой выходной сигнал на входы всех остальных нейронов слоя.

В *ИНС встречного распространения* объединены два хорошо известных алгоритма: самоорганизующаяся карта Кохонена [3] и «исходящая звезда» Гроссберга (outstar) [55]. Их объединение позволяет получить свойства, которых нет ни у одного из них в отдельности. Обобщающая спо-

способность сети позволяет получать обоснованный выход даже при входном векторе, который является неполным.

На рисунке Г.2 приведено изображение структуры ИНС встречного распространения. Нейроны слоя 0 служат точками разветвления и не выполняют вычислений. Каждый нейрон слоя 0 соединен с каждым нейроном слоя 1, называемого слоем Кохонена связью с весом ω_{mn} . Аналогично: каждый нейрон в слое 1 соединен с каждым нейроном слоя 2 связью с весом ν_{np} .

Слой Гроссберга функционирует в сходной манере со слоем Кохонена. Его выход $y^{(2)}$ является взвешенной суммой выходов $y^{(1)}$ слоя Кохонена. Вектор соединяющих весов, обозначенный через V , состоит из весов $\nu_{11}, \nu_{21}, \dots, \nu_{np}$. Выход каждого нейрона слоя Гроссберга есть

$$y_j^{(2)} = \sum_{i=1}^n y_i^{(1)} \nu_{ij}, \quad (\text{Г.5})$$

где $y_j^{(2)}$ — выход j -го нейрона слоя Гроссберга.

В данной работе нейронные сети использовались в независимом модуле, который взаимодействовал с CM Grate через механизм уведомления о событиях.