МИНОБРНАУКИ РОССИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ» (НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Кафедра параллельных вычислений

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Мустаков Руслан Набиевич

РАЗРАБОТКА ПОВЕДЕНЧЕСКОГО ОТЛАДЧИКА ДЛЯ СИСТЕМЫ ФРАГМЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ LuNA

Направление подготовки 230100.62 ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА

Руководитель

Перепёлкин В.А.
(фамилия , И., О.)

(уч.степень, уч.звание)

……..
(подпись, дата)

Автор

Мустаков Р.Н.
(фамилия , И., О.)
ФИТ, 9204
(факультет, группа)

……..
(подпись, дата)

Новосибирск, 2013г.

МИНОБРНАУКИ РОССИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ» (НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Кафедра параллельных вычислений

УТВЕРЖДАЮ

Зав. кафедрой Малышкин В.Э.
(фамилия, И., О.)

……
(подпись, дата)

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

Студенту Мустакову Руслану Набиевичу

Направление подготовки 230100.62 ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА

ФАКУЛЬТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Тема: Разработка поведенческого отладчика для системы фрагментированного программирования LuNA.

Исходные данные (или цель работы): проработать необходимую функциональность отладчика в рамках парадигмы фрагментированного программирования с учетом деталей поведенческой отладки; разработать архитектуру отладчика, выполняющую требования переносимости, модифицируемости, распределённости и удалённости; реализовать и интегрировать отладчик с текущей версией исполнительной системы LuNA.

Структурные части работы: Изучение парадигмы фрагментированного программирования. Анализ существующих средств отладки. Интерпретация типичной функциональности отладчиков относительно парадигмы фрагментированного программирования. Разработка архитектуры отладчика. Реализация отладчика. Интеграция отладчика с исполнительной системой LuNA.

Содержание

ВВЕДЕНИЕ

Ошибки в программах. Средства борьбы с ошибками

Известно, что при написании программ на любом языке программирования программистом зачастую допускаются ошибки разного характера. Одной из очевидных и широко применяемых классификаций ошибок является классификация по времени появления:

- *Ошибки времени компиляции* – ошибки, связанные с некорректным использованием синтаксиса языка (*синтаксические* ошибки) или операций (например, несоответствие сигнатуре функции), а также ошибки, возникающие при использовании несуществующих имён, и другие ошибки, которые обнаруживаются компилятором. Для программиста этот класс ошибок отследить наиболее просто, поскольку, как уже было сказано, они обнаруживаются на стадии компиляции программы.
- *Ошибки времени исполнения* – сюда можно отнести ошибки, которые вызывают крах всей программы или её части (системные ошибки, нехватка ресурсов, некорректное использование типов в языках с динамической типизацией и др.), а также ошибки, из-за которых программа работает (или может работать при определенных условиях) некорректно - это недосмотры в реализации, модели или архитектуре программы. Обычно ошибки первого типа отследить и исправить значительно проще, поскольку при крахе система исполнения даёт обширную информацию о возникшей ошибке. Примеры таких ошибок – выход за пределы массива, обращение по некорректному указателю и др. Ошибки, не вызывающие краха программы, исправить гораздо сложнее, поскольку они не всегда легко воспроизводимы и отследить источник такой ошибки в крупной программе непросто.

Для поиска и устранения ошибок времени исполнения используются три средства: тестирование, верификация и отладка.

- *Тестирование* [1] позволяет выявить сам факт наличия ошибки. Процесс тестирования заключается в передаче программе или её части набора входных данных, на который ожидается определенная реакция. Если реакция неожиданная,

значит в программе присутствует ошибка. Тестирование может быть как ручным (проводится человеком), так и автоматизированным (создается программа, имитирующая действия пользователя, создаются подпрограммы для тестирования отдельных модулей и т.п.).

- *Верификация* – способ убедиться в корректности работы программы путём формального анализа и доказательства отсутствия ошибок с использованием математического аппарата. Иногда может проводиться автоматически, чтобы доказать наличие или отсутствие некоторых свойств программы (например, отсутствие мертвых блокировок или ошибок соревнований).
- *Отладка* [1] необходима, чтобы локализовать источник ошибки. Зачастую для этого необходимо иметь возможность узнать о состоянии исполнения программы в определенных точках, историю исполнения программы и т.д. Для реализации подобных возможностей иногда достаточно стандартных средств вывода, но в более сложных ситуациях используются *отладчики*. Отладчик – это программа, предоставляющая пользовательский интерфейс и функциональность, описанную ранее, а также любую другую, которая может помочь локализовать источник ошибки в конкретной системе программирования.

Отдельного рассмотрения требует вопрос ошибок поведения в параллельной программе, а также вопрос отладки этих ошибок[2]. *Поведение параллельной программы* – это множество её возможных сценариев исполнения. Под *сценарием* исполнения понимается порядок исполнения операций в программе при конкретном запуске этой программы. В параллельной программе множество сценариев исполнения огромно, это обуславливается различными задержками при коммуникациях, различным временем прерываний потоков и т.д. При наличии ошибки в параллельной программе может случиться, что при одних и тех же входных данных программа выдаёт разные результаты из-за того, что она отработала по другому сценарию. Подобные ошибки называются *поведенческими*. Существуют следующие типы поведенческих ошибок:

- Мертвые блокировки
- Ошибки соревнования
- Ошибки коммуникаций
- Неоднородность загрузки вычислительного комплекса

- Неиспользование всех вычислительных ресурсов

Способы отладки этих ошибок включают:

- Установку максимального времени выполнения операций, подверженных мертвым блокировкам
- Использование формальных спецификаций работы программы и сравнение специфицированного и реального поведения программы
- Анализ протокола (сценария) работы программы

Фрагментированное программирование

В настоящее время для программирования задач в области высокопроизводительных вычислений используются очень низкоуровневые средства – это языки программирования C/C++[3], Fortran и библиотеки, реализующие стандарт MPI[4] для этих языков. Программировать с использованием таких средств довольно сложно, раз от раза приходится решать ряд однообразных задач системного программирования, не связанных с прикладной задачей. В идеале, хотелось бы, чтобы прикладную задачу мог запрограммировать человек, не обладающий квалификацией программиста, а просто владеющий математическим аппаратом, связанным с предметной областью его задачи. Проект фрагментированного программирования[5] стремится облегчить программирование задач научного моделирования, в которых зачастую требуется использовать большие вычислительные мощности для получения результатов за разумное время. В рамках этого проекта разрабатывается высокоуровневый язык LuNA (Language for Numerical Algorithms)[6] и исполнительная система для этого языка, а также другие необходимые средства.

Можно выделить следующие особенности фрагментированного программирования:

- Фрагментированные алгоритм и программа **сразу параллельные**, нет автоматического распараллеливания. Фрагментированный алгоритм предполагает возможность параллельного исполнения, а фрагментированная программа поступает на исполнение ИС, которая по определению стремится выполнить её параллельно(отображает фрагменты на ресурсы).
- **Корректность** фрагментированных программ: фрагментированные программы собираются из фрагментов – простых частей-блоков, инкапсулирующих фрагменты данных и вычислений исходного алгоритма. Вычисления фрагмента реализуются последовательной процедурой, поэтому отладка отдельно взятого фрагмента сводится к отладке последовательной процедуры. Для фрагмента определено, с какими фрагментами и как именно корректно участвовать в сборке. Зависимости по данным между фрагментами формируют потоковое управление, в рамках которого возможно исполнение фрагментированной программы. Однако, задание самого алгоритма по-прежнему может содержать ошибки и недосмотры со стороны программиста. Именно на поиск источника и устранение таких ошибок направлена разработка отладчика, которой посвящена данная работа.

- Фрагментированная программа содержит не только информационные зависимости, но и **дополнительную информацию** об отображении на ресурсы, поэтому возможно автоматически исполнять эффективно (это основная задача исполнительной системы).
- **Усилия программиста** акцентируются только на важные вещи, а системная часть реализуется исполнительной системой – отпадает необходимость в глубоком изучении тонкостей параллельного программирования, в идеале достаточно изучить фрагментированное программирование чтобы писать хорошие программы.

В контексте фрагментированного программирования многие поведенческие ошибки становятся неактуальными, поскольку человек программирует в высокоуровневых средствах и не имеет доступа к системным компонентам параллельной программы. Однако, системы фрагментированного программирования теоретически могут позволять осуществлять прямое управление распределением ресурсов и порядком выполнения операций. Для объяснения термина «прямое управление» рассмотрим детально понятие управления вычислением алгоритма.

Под *управлением*, вообще, понимается описание того, в каком порядке могут, и в каком не могут быть исполнены операции алгоритма. По описанию алгоритма естественным образом может быть построено *потоковое управление*, которое состоит в том, что операции должны быть выполнены в порядке, не противоречащем информационным зависимостям алгоритма (dataflow). Это управление самое мощное в том смысле, что задаёт максимально широкое поведение, однако оно плохо тем, что автоматический выбор конкретного эффективного сценария затруднителен. Кроме того, реализация потокового управления в чистом виде (т.н. режим интерпретации) накладна. По этим причинам на практике используется т.н. *прямое управление* — управление, которое фиксирует конкретный порядок выполнения операций. Реализация прямого управления гораздо менее накладна, и если порядок, им зафиксированный, эффективен, то и реализация алгоритма в целом получается эффективной. Прямое управление, в частности, может быть реализовано отдельной прикладной (управляющей) программой.

В случае, если нет возможности верифицировать отсутствие нарушений прямым управлением потокового, требуются возможности поведенческой отладки. Кроме того, поведенческая отладка фрагментированных программ может понадобиться, если какие-то

операции имеют побочные эффекты, и система не способна отследить реальные зависимости между такими операциями. Предположим, что программисту требуется наличие строгого порядка исполнения операций записи в один и тот же файл, а параметрами этих операций являются имя файла и строка, которую необходимо в файл дописать. С точки зрения системы, эти операции независимы, но по задумке программиста они должны исполняться по порядку.

Цель работы

Требуется разработать отладчик для системы фрагментированного программирования LuNA (Language for Numerical Algorithms), разработанной и развивающейся в Лаборатории синтеза параллельных программ Института вычислительной математики и математической геофизики СО РАН. Разработка отладчика включает следующие задачи:

- Определение необходимой функциональности для отладки фрагментированной программы и проработка деталей этой функциональности применительно к фрагментированной парадигме.
- Разработка архитектуры отладчика. Архитектура должна удовлетворять следующим требованиям:
  - *Распределённость*. Исполнительная система LuNA является распределённой, поэтому обработка отладочных команд должна производиться распределённо при взаимодействии с модулями исполнительной системы.
  - *Удалённость*. Поскольку фрагментированные программы ориентированы на исполнение на мультипроцессорных системах, в том числе на кластерах, а пользователь отладчика (программист LuNA) работает со своей рабочей станции, необходима возможность отладки программ с этой рабочей станции или, другими словами, должна поддерживаться удаленная отладка.
  - *Переносимость* по отношению к различным версиям исполнительной системы. Поскольку система фрагментированного программирования LuNA активно развивается, различные версии её исполнительной системы значительно отличаются друг от друга, поэтому необходимо разработать такую архитектуру, которая позволяла бы легко интегрировать отладчик с различными версиями исполнительной системы.
  - *Модифицируемость*. Поскольку система фрагментированного программирования LuNA не имеет широкого распространения, представляется затруднительным провести анализ типичных ошибок и, соответственно, вывести наиболее востребованную функциональность отладчика. Поэтому необходимо, чтобы добавление и модификация команд отладчика не являлись проблемой.

- Реализация базовой версии отладчика для текущей версии исполнительной системы.

Актуальность работы следует из необходимости наличия отладочных средств в любой системе программирования. Эта необходимость, в свою очередь, следует из того, что, как показывает практика, при создании хоть сколько-нибудь сложных программ человеком совершаются ошибки.

Новизна работы следует из отсутствия на текущий момент средств отладки фрагментированных программ. Научный интерес работа представляет с точки зрения проработки алгоритма распределенной обработки отладочных запросов.

Глава 1. Функциональность отладчика

В первую очередь в работе необходимо определить, какую функциональность будет реализовывать отладчик языка LuNA, а также понять, как эта функциональность будет ложиться на фрагментированную парадигму и распределенное исполнение.

Сначала необходимо провести обзор и анализ существующих средств отладки в различных системах программирования с целью выяснить, какие возможности отладчика наиболее востребованы и что есть общего в функциональностях отладчиков систем программирования с разными парадигмами. Очень важно рассмотреть этот вопрос детально, поскольку на данном этапе развития системы LuNA, ввиду нераспространенности этой системы, неочевидно, какие ошибки будут возникать у программиста наиболее часто, и мы лишь можем делать предположения. Подробно рассмотрев существующие средства отладки, можно получить представление о том, какие ошибки возникают у программистов в целом в любых системах программирования. Для получения полного и адекватного взгляда были выбраны характерные представители нескольких парадигм программирования – императивной объектно-ориентированной с поддержкой многопоточности (Java), распределенной объектно-ориентированной (Charm++) и распределённой функциональной (Erlang).

1.1 Обзор и анализ средств отладки

1.1.1 Отладка в Java

Java – императивный язык программирования с поддержкой многопоточности в модели языка. Он был выбран, как один из самых популярных языков программирования на сегодняшний день, а также как язык, использующий в основе платформу Java Virtual Machine (JVM) и платформенный подход к отладке. В настоящее время существует множество развивающихся языков, исполняющихся на платформе JVM, – например, Fantom, Clojure, Groovy, Scala и др.

Сначала будут представлены функциональные возможности отладчиков Java, после чего будут рассмотрены некоторые их архитектурные и реализационные детали.

Отладчики языка Java предоставляют графический пользовательский интерфейс, выглядящий по-разному в зависимости от конкретной среды разработки, однако предоставляющий одну и ту же функциональность:

- Установка точек останова (breakpoints) посредством указания строки исходного кода.
- Отображение стека вызовов для всех активных потоков и переход в различные контексты внутри этого стека.
- Отображение списка переменных и их значений в данном контексте.
- Пошаговое исполнение программы внутри отдельных потоков.
- Выборочная остановка части потоков исполнения.
- Установка “наблюдаемых” (watched) выражений – выражений, значения которых пересчитываются и отображаются при изменении текущего контекста.

Теперь перейдем к рассмотрению архитектурных деталей отладчиков Java. Любое средство отладки языка, исполняющегося на платформе JVM (Java Virtual Machine), строится на основе JPDA[7] (Java Platform Debugger Architecture, Архитектура отладчика на платформе Java). JPDA – это многоуровневая архитектура, позволяющая легко создавать отладочные приложения, обладающие переносимостью по отношению к различным реализациям виртуальной машины Java и версиям Java Development Kit (новые версии JDK обычно являются расширением предыдущих). JPDA состоит из трёх слоёв:

- JVM TI – Java VM Tool Interface. Этот слой определяет сервисы, предоставляемые виртуальной машиной для отладки.
- JDWP – Java Debug Wire Protocol. Коммуникационный слой между процессами отладчика и отлаживаемой программы.
- JDI – Java Debug Interface. Определяет высокоуровневые интерфейсы на языке Java, которые могут использоваться для разработки средств удалённой отладки на пользовательской стороне.

1.1.2 Отладка в Charm++

Charm++ - Объектно-ориентированный язык программирования, ориентированный на распределенное исполнение. Расширение языка C++. Программа на Charm++ представляет собой множество объектов (chares), вызовы методов которых осуществляются через передачу сообщений. Сами объекты могут физически располагаться на различных узлах распределенной системы. Данный язык был выбран как один из известных языков в

области высокопроизводительных вычислений, берущий на себя задачу распределения ресурсов, как и системы фрагментированного программирования.

В поставку Charm++ входит отладчик[8] с графическим интерфейсом пользователя. Отладчик предоставляет следующую функциональность:

- Полная остановка исполнения программы и его возобновление.
- Установка точек останова на “точки входа”. Точки входа в терминах Charm++ - это методы, обрабатывающиеся при получении сообщений от других chare-объектов.
- Обзор сущностей программы (например, элементов массивов) и их содержимого на любом узле в любой момент исполнения программы.
- Подключение процессов на отдельных узлах к отладчику GDB (GNU Debugger).
- Остановка исполнения программы на выбранных узлах.
- Возможность записи и воспроизведения сценария работы программы, то есть последовательности обработки сообщений всеми объектами в программе.
- Детальный анализ используемой памяти по типам:
  - Память, выделенная системой;
  - Память, выделенная программистом;
  - Память, выделенная под сообщения;
  - Память, выделенная под chare-объекты;
  - Память, на которую нет указателей (анализ утечек памяти).

1.1.3 Отладка в Erlang

Erlang – функциональный язык программирования, ориентированный на распределенное исполнение. В модели языка присутствует понятие процесса и операторы передачи и получения сообщения. Данный язык интересен для рассмотрения как развивающийся функциональный распределенный язык, получивший применение в высоконагруженных приложениях известных компаний.

В поставку Erlang также входит отладчик[8] с графическим интерфейсом пользователя. Отладчик предоставляет следующую функциональность:

- Установка точек остановка с указанием строки и имени модуля.
- Установка условных точек остановка с указанием строки и имени модуля, а также функции, которая будет вызываться для проверки истинности требуемого условия. Эта функция описывается на самом языке Erlang и принимает список переменных и их значений в текущем контексте.
- Наблюдение за стеком вызовов, переход по контекстам внутри этого стека, возможность активировать сохранение кадров функций с хвостовой рекурсией.
- Мониторинг всех активных процессов в системе (их идентификатор, статус, первая функция, вызванная в текущем модуле, дополнительная информация).
- Подключение к процессу с возможностью пошаговой отладки.

Помимо этого, в стандартной библиотеке Erlang присутствует модуль dbg, который содержит функции, позволяющие отлаживать программу из интерпретатора языка, а также модуль dyntrace, позволяющий проводить динамическую трассировку. Эти модули могут быть использованы для разработки собственных средств отладки.

1.2 Результаты анализа

Средства отладки в различных языках программирования реализуют схожую функциональность, несмотря на существенные отличия в парадигмах программирования. Основными элементами этой функциональности являются:

- Возможность остановки исполнения программы в определенном месте при выполнении определенного условия.
- Возможность исследования состояния программы в целом или её части.
- Возможность изменения состояния программы тем или иным образом.
- Возможность следить за тем, как меняется состояние интересующих частей программы.
- Средства для уменьшения недетерменизма исполнения параллельной программы (поведенческая отладка).

1.3 Модель и семантика исполнения фрагментированной программы

Прежде, чем приступить к описанию функциональности отладчика для системы фрагментированного программирования, необходимо формально описать модель фрагментированной программы и семантику её исполнения, введя необходимую терминологию.

1.3.1 Модель фрагментированной программы

Фрагментированный алгоритм (ФА) – это двудольный ориентированных граф, состоящий из фрагментов данных (ФД) и фрагментов вычислений (ФВ). ФД – это агрегированные переменные единственного присваивания, например подматрицы некоторой матрицы. ФД представляют переменные прикладного алгоритма. ФВ – это операции над ФД. Дуги ФА определяют, какие ФД являются входными, а какие – выходными для данного ФВ. ФВ вычисляет выходные ФД из входных. В целом это известное представление алгоритма как множества переменных и операций над ними с той разницей, что значениями ФД являются не отдельные величины, а агрегаты, например фрагменты матрицы. В реализации ФД обычно реализуется блоком памяти, а ФВ – вызовом процедуры на традиционном языке программирования.

Множества ФД и ФВ описываются с помощью операторов, аналогичных операторам суперпозиции, примитивной рекурсии и минимизации из теории рекурсивных функций, что обеспечивает тьюринг-полноту такого представления.

Язык фрагментированного программирования LuNA располагает средствами во-первых, для описания ФА, а, во-вторых, содержит средства для описания т.н. «рекомендаций» – информации о желаемой схеме исполнения ФА. Рекомендации можно разделить на две группы – низкоуровневые, которые, по сути, представляют собой язык для описания конкретной схемы распределения ресурсов, и высокоуровневые, предназначенные для описания принципиальной схемы распределения ресурсов. На основе высокоуровневых рекомендаций низкоуровневые достраиваются автоматически, хотя программист может определять и низкоуровневые рекомендации, если затрудняется с помощью высокоуровневых сформулировать желаемую схему распределения ресурсов. Совокупность ФА и рекомендаций называется фрагментированной программой (ФП). Формальное определение рекомендаций не приводится, т.к. оно не является существенным с точки зрения данной работы.

ФА является удобным представлением алгоритма с точки зрения автоматизации распределения ресурсов. Во-первых, ФВ – это, как правило, операции без побочных эффектов, что означает возможность независимого исполнения ФВ, для которых вычислены значения их входных ФД, возможность осуществлять миграцию ФД и ФВ с узла на узел без влияния на значения, вычисляемые программой и прозрачно для программиста. Если операция имеет побочные эффекты, то это должно указываться явно, вместе с указаниями о том, как с ней работать – какой должен быть порядок выполнения операций, ограничения на миграцию и т.п. Во-вторых, системе явно виден ход выполнения программы – какие значения ФД уже вычислены, какие ФВ исполняются в настоящий момент. Это что позволяет (частично) прогнозировать и регулировать загрузку процессоров. В-третьих, явно видны информационные связи между ФВ, что позволяет системе оценивать, где, когда и в каком объеме понадобятся коммуникации. В-четвертых, собственно ФА не содержит никакой информации о распределении ресурсов, эта информация содержится только в рекомендациях. Это позволяет решать задачу распределения ресурсов независимо от других подзадач при разработке параллельной программы.

1.3.2 Семантика исполнения фрагментированной программы

Здесь вводится понятие фрагмента кода (ФК):

- ФК – это два набора параметров, называемых входным и выходным и тело ФК.

ФК= $in_1, in_2, \ldots, in_n$, $out_1, out_2, \ldots, out_m$, Body , n>0, m>0

- Тело ФК – это либо ссылка на атомарный ФК (внешний объект по отношению к модели), либо структурированный ФК (стр. ФК)
- стр. ФК – это множество локальных ФД и множество локальных ФВ.

а) Исполнение ФП – то же, что исполнение ФА, с учетом рекомендаций.

б) Исполнение ФА определяется, как исполнение одного ФВ, который является применением одного из ФК, не имеющего ни входных, ни выходных параметров.

в) Исполнение ФВ определяется следующим образом.

1) Входные параметры ФК (если они есть) задают константы и ФД, значения которых вычислены на момент начала исполнения ФК (что такое значение вычислено, см. ниже). Локальные ФД и выходные параметры – это объекты,

значения которых требуется вычислять.

2) Среди множества всех ФВ выделяются те, чьи входные параметры означены. Они называются готовыми к исполнению ФВ. Остальные ФВ называются неготовыми к исполнению.

3) Из множества готовых к исполнению ФВ извлекается один ФВ и исполняется.

4) По мере исполнения ФВ какие-то ФД (или псевдо-ФД «размер кортежа») получают свои значения. Если при этом для каких-то ФВ все входные параметры оказались означенными, то эти ФВ переходят в множество готовых к исполнению.

5) Процесс (шаги c-d) продолжается до тех пор, пока множество готовых к исполнению ФВ не окажется пустым при том, что ни один другой ФВ больше не исполняется.

г) Значение ФД считается вычисленным, если:

1) Для атомарного ФД – его значение выработано некоторым ФВ

2) Для кортежа – если вычислены его размер и значения всех его элементов.

1.4 Функциональность отладчика LuNA

На основе результатов анализа можно составить список команд, которые необходимо поддерживать отладчику. Отметим, что слово “команда” здесь подразумевается в самом общем смысле – это некий запрос для отладчика, поступающий от пользователя. Способ передачи команды не фиксируется. С точки зрения отладчика, программа может находиться в трех состояниях: программа еще не исполнялась, программа исполняется, программа приостановлена. Команды могут требовать, чтобы программа находилась в одном из перечисленных состояний.

Приведём список команд с их интерпретацией относительно фрагментированной парадигмы:

- Запуск программы (**start**). Команду можно исполнить только в случае, если программа находится в состоянии “еще не исполнялась”. Запуск программы означает постановку очередь на исполнение всех фрагментов вычислений, не имеющих входных зависимостей.

- Приостановка исполнения (**pause**). Команду можно исполнить только в случае, если программа находится в состоянии “исполняется”. Приостановка исполнения распределенной программы не так тривиальна, как программы, работающей на одном узле. Приостановка исполнения означает, что каждый узел кластера закончит исполнение текущих атомарных операций и не будет выполнять новые.
- Продолжение исполнения (**resume**). Команду можно исполнить только в случае, если программа находится в состоянии “приостановлена”. Продолжение исполнение означает возобновление работы программы в привычном режиме с учетом команд, выполненных пользователем, пока программа была приостановлена.
- Установка точки останова (**breakpoint**). Команду можно исполнить при любом состоянии программы. Точку останова во фрагментированной программе можно установить как на фрагмент вычислений, так и на фрагмент данных. Рассмотрим подробнее каждый из вариантов. Для фрагментов вычислений точка останова может ставиться на выполнение следующих условий:
  - Система собирается начать исполнять атомарный фрагмент или разворачивать структурированный фрагмент вычислений.
  - Система развернула структурированный фрагмент вычислений.
  - Система закончила исполнение атомарного фрагмента вычислений. Отметим, что хотя условие окончания исполнения структурированного фрагмента вычислений отследить теоретически возможно, но на практике для некоторых реализаций исполнительных систем это может оказаться сложной задачей с точки зрения качественной масштабируемой реализации, поэтому поддержка этого условия не ставится как обязательное требование.

  Для фрагментов данных точка останова срабатывает после того, как фрагмент был вычислен.
- Снятие точки останова (**breakpoint remove**). Команду можно исполнить при любом состоянии программы. После выполнения команды из системы удаляется заданная точка останова.
- Список точек останова (**breakpoint list**). Команду можно исполнить при любом состоянии программы. Команда предоставляет пользовтелю список активных точек

останова.

- Получение информации о программе (**info**). Команду можно исполнить при любом состоянии программы, однако пользователь должен понимать, что результат её работы может оказаться неактуальным в момент его получения, если команда запущена в процессе исполнения программы. Команда предоставляет пользователю информацию о заданном фрагменте данных или вычислений. Информация о фрагменте данных включает:
  - Состояние: не вычислен, вычислен, уничтожен, неизвестно. Последнее может возникнуть в случае, если фрагмент находится в состоянии пересылки с одного узла на другой и никакой узел не может сказать ничего об этом фрагменте.
  - Значение. Включается только, если фрагмент данных вычислен.
  - Расположение. Это список узлов, на которых располагается фрагмент данных. Может быть пустым, если расположение неизвестно или фрагмент еще не был вычислен. Расположение представляется списком, поскольку фрагмент может реплицироваться исполнительной системой.

  Информация о фрагменте вычислений включает его состояние: не готов, готов, исполняется, исполнился, неизвестно. Последнее, как и в случае с фрагментами данных, означает, что ни один узел не может ничего сказать о заданном фрагменте вычислений.

Перечисленные команды составляют ключевую функциональность отладчика, то есть ту функциональность, без которой проводить отладку не представляется возможным. Помимо этой функциональности для отладчика полезно иметь возможность трассировки и возможность проверки истинности утверждений о работе фрагментированной программы. Эту функциональность можно выразить в следующих командах:

- Наблюдение (**watch**). Команду можно исполнить при любом состоянии программы. После выполнения команды исполнительная система начнет собирать информацию о заданном фрагменте данных или вычислений.
- Снятие наблюдения (**watch remove**). Команду можно исполнить при любом состоянии программы. После выполнения команды прекращается заданное наблюдение. При этом накопленная история об объекте сохраняется для дальнейшего анализа.

- Список наблюдаемых объектов (**watch list**). Команду можно исполнить при любом состоянии системы. Команда предоставляет пользователю список активных наблюдений и их объектов.
- История (**history**). Команду можно исполнить при любом состоянии программы. Выводит историю, собранную для фрагмента данных или вычислений, для которого ранее была выполнена команда watch. К каждой операции в этой истории приписывается временная отметка. Для фрагмента вычислений записывается, когда и на каком узле он был вычислен (развернут). Для фрагмента данных записываются операции порождения, пересылки и уничтожения.
- Утверждение (**assertion**). Команду можно исполнить при любом состоянии программы. Команда устанавливает некоторое утверждение, которое должно быть истинным в случае, если программа работает корректно. Система, в свою очередь, должна сообщить пользователю, если это утверждение будет нарушено. Примерами утверждений могут являться утверждения о порядке вычислений некоторых фрагментов вычислений или утверждения о свойствах значений фрагментов данных.

Последние команды составляют поведенческую часть функциональности отладчика. Другими словами, эта функциональность позволяет отследить конкретный сценарий срабатывания программы и убедиться, что этот сценарий соответствует ожиданиям программиста. Однако, эта функциональность позволяет проводить лишь пассивную поведенческую отладку, то есть она не позволяет управлять поведением программы, а позволяет лишь убеждаться в корректности текущего сценария срабатывания. Активная поведенческая отладка фрагментированных программ требует глубокого рассмотрения и выходит за рамки данной работы.

Глава 2. Архитектура отладчика

Поскольку архитектура Java Platform Debugger Architecture (JPDA) уже выполняет ряд требований, представленных к архитектуре отладчика LuNA (а именно: удаленность, модифицируемость и переносимость), и является качественно проработанной и проверенной временем, было решено использовать подобную архитектуру, расширив её с учетом распределенности исполнения фрагментированной программы.

2.1 Java Platform Debugger Architecture

Рассмотрим подробно, что представляет из себя JPDA.

Как было сказано во введении, JPDA состоит из трёх слоёв:

- **Java VM Tool Interface** (JVM TI). Этот слой определяет сервисы, предоставляемые виртуальной машиной для отладки.
- **Java Debug Wire Protocol** (JDWP). Коммуникационный слой между процессами отладчика и отлаживаемой программы.
- **Java Debug Interface (**JDI**)**. Определяет высокоуровневые интерфейсы на языке Java, которые могут использоваться для разработки средств удалённой отладки на пользовательской стороне.

Разработчик отладчика может реализовать любой из перечисленных слоёв независимо от другого. Наиболее высокоуровневым и простым для реализации, очевидно, является слой JDI. Разработав реализацию этого интерфейса, можно получить рабочий отладчик, который может взаимодействовать с реализацией JVM TI, созданной разработчиками виртуальной машины Java.

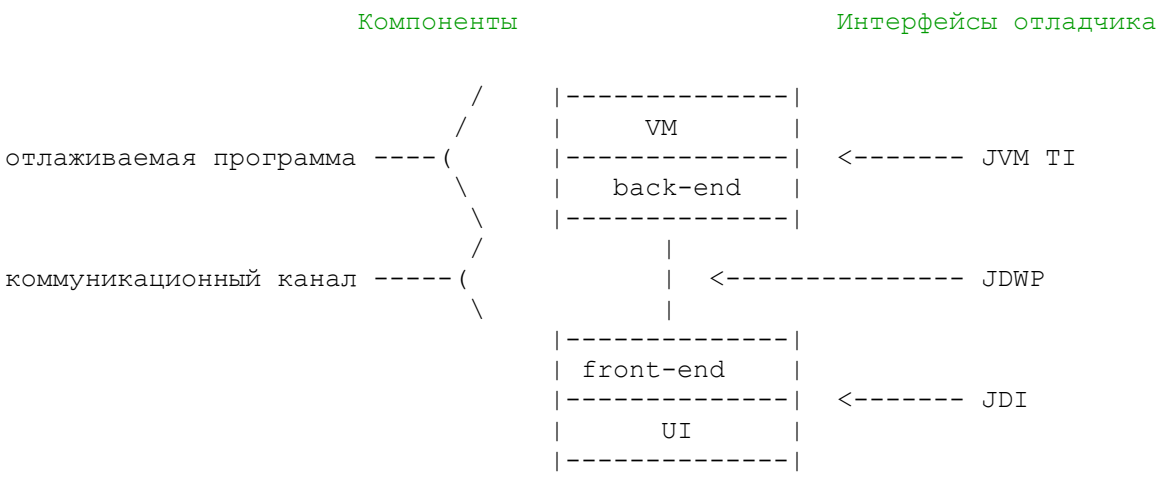

Рисунок 1. Иллюстрация Java Platform Debugger Architecture.

Архитектура JPDA состоит из следующих компонентов:

- **Отлаживаемый процесс**. Он состоит из отлаживаемого приложения, виртуальной

машины, которая его исполняет, и back-end отладчика.

- **Java Virtual Machine** (ВМ). Виртуальная машина, исполняющая отлаживаемое приложение. Напомним, что данная архитектура строится так, чтобы поддерживать широкий спектр реализаций виртуальной машины. Виртуальная машина реализует интерфейс Java Virtual Machine Tool Interface (JVM TI).
- **back-end**. Он отвечает за обработку запросов, поступающих от front-end к отлаживаемому приложению, а также за отправку ответов на эти запросы и оповещение о событиях, на которые подписался front-end. back-end общается с front-end через коммуникационный канал, используя Java Debug Wire Protocol (JDWP). С виртуальной машиной, в свою очередь, back-end общается через интерфейс JVM TI.
- **Коммуникационный канал**. Он связывает front-end и back-end компоненты отладчика. Коммуникационный канал состоит из двух частей:

  а) Коннектор. Это объект JDI, который осуществляет создание соединения между back-end и front-end. JPDA определяет три типа коннекторов:

  1) Ожидающий (listening) коннектор. front-end ожидает входящего соединения от back-end.

  2) Подключающийся (attaching) коннектор. front-end подключается к уже запущенному back-end.

  3) Запускающий (launching) коннектор. Front-end запускает Java-процесс, который будет исполнять код отлаживаемого приложения и back-end.

  б) Транспорт. Он служит непосредственно для передачи информации между back-end и front-end. Конкретный механизм передачи данных не специфицируется. Например, это могут быть сокеты, последовательный канал или разделяемая память. Тем не менее, формат и семантика данных, проходящих через транспортный канал, специфицируется JDWP.
- **front-end**. “Клиентская” сторона отладчика, реализующая интерфейс JDI.
- **Пользовательский интерфейс**. Его детали не специфицируются в архитектуре, поскольку с использованием JDI можно создать любой тип пользовательского интерфейса.

Обоснуем выполнение данной архитектурой части требований, представленных к архитектуре отладчика для языка LuNA. Очевидно, в данной архитектуре выполняется требование удаленности, поскольку обмен данными между отлаживаемым процессом и клиентом происходит через абстрактный транспорт, который может быть реализован на основе сетевых сокетов. Выполнение требования модифицируемости следует из наличия нескольких независимых слоев, изменение реализации которых не приводит к необходимости модифицировать реализацию других слоев. Требование переносимости выполняется, поскольку часть отладчика, неподсредственно встроенная в виртуальную машину, взаимодействует с ней через специфицированный интерфейс. Поэтому различным версиям виртуальной машины достаточно лишь реализовать этот интерфейс, чтобы отладчик мог взаимодействовать с ними.

Таким образом, данной архитектурой выполняются все необходимые требования, кроме распределенности, проработка деталей которой представляет собой немалую сложность. Кроме того, с учетом распределенности исполнения важно сделать так, чтобы реализация интерфейса, предоставляемого исполнительной системе (виртуальной машине языка LuNA), не требовала больших усилий со стороны разработчиков этой системы по организации взаимодействия между узламами для исполнения отладочных команд, иначе требование модифицируемости окажется невыполненным.

Рассмотрим теперь более подробно структуру каждого из компонентов разработанной архитектуры. Для простоты будем рассуждать о трёх компонентах: клиент, транспорт и сервер.

2.2 Клиент

С точки зрения отладчика, клиент – это компонент, основная задача которого – получать команды от пользователя и преобразовывать их в запросы, передавая последние транспорту, а также оповещать пользователя о возникших событиях. Запрос, ответ и событие являются основными сущностями в системе. Для формирования запросов и интерпретации ответов и событий клиент разделяет с сервером представление модели фрагментированной программы, а также абстракции отладчика. В модели фрагментированной программы описываются такие сущности, как сама фрагментированная программа, фрагмент данных и фрагмент вычислений. Абстракции отладчика включают упомянутые ранее запросы, ответы и события, а также любые другие сущности, необходимые для обеспечения взаимодействия между клиентом и сервером.

Каждая из сущностей должна быть сериализуемой, чтобы иметь возможность передать её через соединение, порождённое транспортом.

В отличие от JPDA, здесь у клиента нет некой абстракции виртуальной машины, поскольку LuNA, в отличие от Java, на данный момент не сформировалась как платформа – спецификация исполнительной системы и даже модель фрагментированной программы не зафиксированы строго. В такой ситуации создание абстракции исполнительной системы навредило бы выполнению требования переносимости.

Отметим, что для клиента в архитектуре не фиксируется способ получения команд от пользователя. Они могут передаваться как через графический интерфейс в некоторой среде разработки, так и в текстовом виде интерпретатору команд. Это облегчает разработку пользовательских интерфейсов любого типа.

2.3 Транспорт

Транспорт отвечает за обмен данными между клиентом и сервером. В этот компонент включены две основные абстракции – это, непосредственно, транспорт и соединение. Кроме того, абстракции запроса, ответа и события относятся именно к этому компоненту.

Транспорт представляет собой механизм порождения соединения через некоторую строку-адрес. Интерпретация содержимого этой строки зависит от конкретной реализации транспорта. Например, в случае с сокетным транспортом, строка может содержать пару “адрес” и “порт”, разделённую двоеточием. Если программа исполняется локально и транспорт реализован через разделяемую память, то строка может содержать идентификатор области памяти.

Соединение представляет собой механизм пакетной передачи данных. Если соединение реализовывается на основе некоторого потокового протокола (например, TCP), то необходимо передавать данные так, чтобы имелась возможность разделять пакеты. Простой способ сделать это – отправлять длину пакета перед самим пакетом.

Клиент использует транспорт, чтобы создать соединение с корневым узлом кластера (корневым по отношению к дереву коммуникаций кластера, о котором подробно рассказывается далее). Клиент может как сам ожидать подключения этого узла, так и присоединяться к этому узлу в зависимости от пользовательских настроек.

2.4 Сервер

Сервер отвечает за обработку запросов, поступающих от клиента через транспорт, за формирование и отправку ответов на эти запросы, а также за оповещение клиента о событиях, на которые он подписался.

Как и любая хорошая распределенная архитектура, архитектура отладчика должна обладать масштабируемостью по ресурсам. Важно учесть то, что обычно скорость обмена данными между узлами кластера гораздо выше, а задержка гораздо ниже, чем те же показатели между любым отдельно взятым узлом кластера и пользовательской машиной. Другими словами, связь с клиентом является узким местом в системе. Другим узким местом является наличие точки управления (пользовательской рабочей станции), т.е. необходимости осуществлять рассылку сообщений один-ко-многим и обратно, причем эту необходимость никак не преодолеть. По этим причинам коммуникации между узлами кластера организованы по дереву. Корень дерева осуществляет обмен данными с клиентом. Таким образом, проблема медленной связи с клиентом становится локализованной и не замедляет работу системы при росте числа узлов, как это могло бы быть, если клиент обменивался данными сразу с несколькими узлами. Использование дерева для осуществления коммуникаций позволяет существенно снизить накладные расходы на выполнение коллективных операций, которые необходимо осуществлять при каждом запросе пользователя.

Схематично взгляд на систему представлен на рисунке 2.

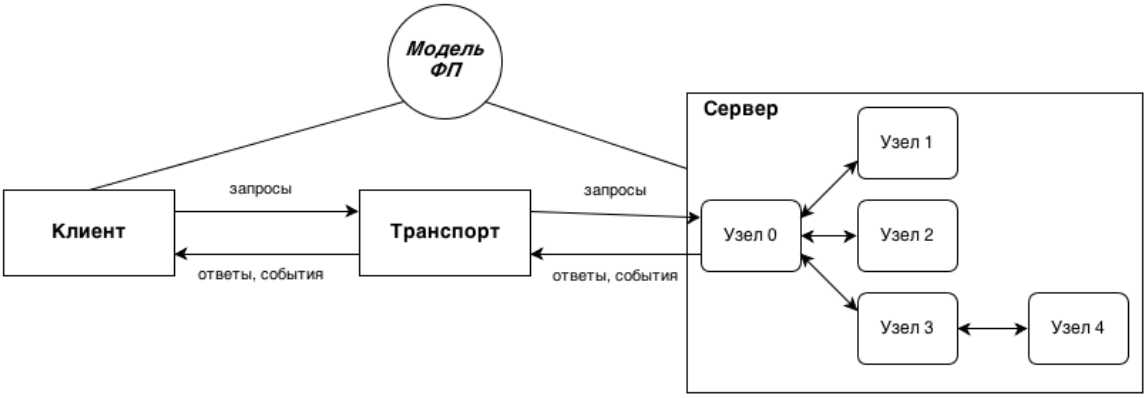

Рисунок 2. Иллюстрация компонентов отладчика.

Теперь необходимо пояснить, как будет осуществляться передача и обработка запросов, приходящих от клиента, а также формирование ответов и оповещение о событиях, на которые клиент подписался. Запросы передаются по дереву от корня к

листьям, при этом, получив запрос, каждый узел обращается к исполнительной системе через определенный интерфейс, чтобы та обработала этот запрос, выдав некоторый ответ. Отметим, что этот интерфейс разработан так, чтобы исполнительной системе не требовалось осуществлять никаких коммуникаций для обработки запросов. Эту ответственность на себя берут модули отладчика.

После того, как прошел процесс обработки запроса, образуется множество узлов, каждый из которых обработал запрос и получил некоторый ответ или ошибку (на самом деле, ошибка является разновидностью ответа). Все эти ответы необходимо собрать в один, общий, который и будет передан клиенту. Для этой цели используется операция редукции, бинарная функция для которой реализуется по-разному в зависимости от типа запроса. Например, в ситуации, если ответ на запрос может представляться двумя значениями – УСПЕХ (0) и ОШИБКА (1) – бинарной функцией будет являться “логическое или”. Другими словами, если на каком-то узле возникла ошибка, то результат исполнения команды в целом будет считаться ошибочным. В ситуации, когда типов ошибок может быть несколько, в качестве кодов следует выбирать степени двойки, и в такой ситуации функцией редукции по прежнему будет “или”, только уже побитовое. Клиент, получив итоговый “флаг” ошибок сможет узнать о возникших типах ошибок применяя операцию “и” к этому флагу.

Опишем структуру ответов и функции их редукции для команд, представленных в предыдущей главе:

- **start**
  - Структура ответа – флаг ошибки. Возможные варианты ошибок – программа уже запущена.
  - Функция для редукции - “или”.
- **pause**
  - Структура ответа – флаг ошибки. Возможные варианты ошибок – программа уже приостановлена.
  - Функция для редукции - “или”.
- **resume**
  - Структура ответа – флаг ошибки. Возможные варианты ошибок – программа не

приостановлена.

- Функция для редукции - “или”.

- **breakpoint**
  - Структура ответа – флаг ошибки. Возможные варианты ошибок – ошибка в параметре, точка останова на заданном объекте уже установлена.
  - Функция для редукции - “или”.

- **breakpoint list**
  - Структура ответа – множество активных точек останова в виде пар (<идентификатор точки останова>, <идентификатор объекта>)
  - Функция для редукции – объединение множеств.
  - Примечание: на самом деле, в качестве функции редукции для данной команды можно использовать функцию, которая просто возвращает свой первый аргумент (функцию выбора). При корректной работе отладчика все узлы должны иметь совпадающие множества активных точек останова.

- **breakpoint remove**
  - Структура ответа – флаг ошибок. Возможные варианты ошибок – неизвестный идентификатор.
  - Функция для редукции - “или”.

- **info <df>** (ситуация, когда аргументом является фрагмент данных)
  - Структура ответа – три компоненты: состояние (не вычислен, вычислен, уничтожен, неизвестно), значение (опционально), расположение (возможно пустое множество идентификаторов узлов).
  - Функция для редукции описывается следующим образом: если i-ая компонента 1-го аргумента известна, то в i-ой компоненте результата будет она, иначе в i-ой компоненте результата будет i-ая компонента 2-го аргумента. Здесь i меняется от 1 до 2. Для получения результирующего расположения (третьей компоненты) используется объединение множеств.
  - Примечание: данная функция редукции рассчитывает, что если разные узлы

знают что-то о заданном фрагменте данных, то они обладают согласованной (одинаковой) информацией для каждой из компонент. То есть, например, не может быть такого, что разные узлы сообщают разное значение для одного и того же фрагмента данных, или что один узел считает, что фрагмент еще не вычислен, а другой, что он уже уничтожен.

- **info <cf>** (ситуация, когда аргументом является фрагмент вычислений)
  - Структура ответа – состояние исполнения заданного фрагмента вычислений (не готов, готов, исполняется, исполнился, неизвестно).
  - Функция для редукции аналогична предыдущей, с учетом того, что компонент в ответе здесь только один.
- Для команд **watch**, **watch remove**, **watch list** ситуация аналогична командам, которые работают с точками останова.
- **history**
  - Структура ответа – отсортированный список пар вида (<временная отметка>, <событие>). Сортировка происходит по временной отметке.
  - Функция для редукции – возвращает список, который может быть получен путём упорядоченного слияния отсортированных списков.
- **assertion**
  - Структура ответа – флаг ошибки. Возможные варианты ошибки – (семантически) некорректное выражение.
  - Функция для редукции - “или”.

Как можно заметить, функция редукции для каждой из команд довольно тривиальна и ее реализация не составляет труда. Идея использования такого подхода вкупе с локальностью требований к исполнительной системе позволяет еще больше упростить добавление новых отладочных команд или, другими словами, улучшить показатель модифицируемости.

Последней нерассмотренной деталью в архитектуре серверной части остался механизм обработки событий. При возникновении на узле события, на которое подписался клиент (точка останова, нарушение утверждения и т.п.), этот узел отправляет информацию о событии корневому. Если в связи с возникшим событием серверной части отладчика

необходимо совершить какие-либо действия, корневой узел порождает запрос, который обрабатывается тем же образом, что и пользовательские запросы. Например, если сработала точка останова, то исполнение программы следует приостановить, и корневой узел отправит запрос на приостановку исполнения (pause). После выполнения требуемых действий корневой узел отправляет событие клиенту, который оповещает о нём пользователя. Локальное оповещение исполнительной системой модуля отладчика происходит по паттерну проектирования Observer[10]. Отметим, что, в общем случае, исполнительная система может оповещать модуль отладчика и о событиях, которые не требуют глобальной реакции. Таким, например, может являться событие появления на узле фрагмента данных, для которого пользователь запросил наблюдение. Модуль отладчика сделает запись о событии с отметкой времени, не тревожа другие узлы.

Получается, чтобы добавить новую команду отладчика, необходимо:

- Добавить функцию в интерфейс взаимодействия отладчика и исполнительной системы и, если требуется, реализовать её для конкретной системы.
- Описать структуру ответа команды.
- Реализовать функции сериализации/десериализации этого ответа.
- Реализовать бинарную функцию для редукции ответа.
- Если требуется, описать структуру необходимых событий, их функции сериализации/десериализации, а также убедиться, что исполнительная система будет оповещать отладчик об этих событиях, когда они возникают.

Задачи по необходимым коммуникациям и использованием этих функций и сущностей на себя берет инфраструктура серверной части отладчика, механизмы работы которой были описаны ранее.

Подведём итог. В главе была описана архитектура отладчика, состоящая из трёх компонентов: клиент, транспорт и сервер. Были описаны сущности и механизмы работы, используемые каждым из компонентов. Доказано выполнение данной архитектурой всех поставленных требований.

Глава 3. Реализация отладчика

Отладчик был реализован с использованием тех же средств, что и текущая версия исполнительной системы – это язык программирования C++, библиотеки MPI и Boost[11]. Полностью готова инфраструктура отладчика, реализованы механизмы, описанные в предыдущей главе. Также на основе этих механизмов реализована простейшая функциональность – команды start, pause, resume. Клиент реализован в виде интерпретатора текстовых команд. Если фрагментированная программа запускается на кластере, требуется, чтобы клиент отладчика был доступен по сети с узлов кластера. В случае с кластерами научных центров зачастую для этого необходимо, чтобы клиент запускался на управляющем узле кластера. В будущем это ограничение можно будет преодолеть, добавив возможность SSH-туннелирования с рабочей станции пользователя через управляющий узел кластера.

Реализация отладчика состоит из трёх модулей: backend, frontend, и common.

3.1 Модуль common

common – модуль, в котором представлены абстракции, совместно используемые модулями backend и frontend. Сюда же входят абстракции транспортного слоя. Основные классы и интерфейсы этого модуля:

- *ITransport* – интерфейс, описывающий транспорт, порождающий соединения. Позволяет либо ожидать подключения, либо устанавливать их, используя строку-адрес, которая может интепретироваться по-разному разными реализациями этого интерфейса.
- *IConnection* – интерфейс, описывающий соединение, порождаемое транспортом. Позволяет осуществлять операции получения и отправки пакетов. Также есть возможность закрыть соединение и проверить состояние открытости.
- *DbgRequest* – абстракция запроса. Состоит из идентификатора запроса, кода команды и данных, интерпретирующихся по-разному в зависимости от команды. Идентификатор запроса должен быть уникальным среди всех запросов, на которые еще не было получена ответа. Идентификатор запроса необходим для возможности асинхронной обработки запросов – в ответ так же включается идентификатор равный идентификатору соответствующего запроса.

- *DbgResponse* – абстракция ответа. Состоит из идентификатора соответствующего запроса, флага ошибки и данных, интерпретирующихся по-разному в зависимости от команды соответствующего запроса.
- *DbgEvent* – абстракция события. Состоит из идентификатора, кода события и данных, интерпретирующихся по-разному в зависимости от кода события. Идентификатор не несет смысловой нагрузки и лишь резервирует место для будущего использования. Его наличие позволяет сделать минимальный размер сообщения-события равным минимальному размеру запросов и ответов (а именно – 48 бит: 32 бита на идентификатор и 16 бит на код). Таким образом, реализации интерфейса IConnection могут проводить проверку целостности сообщения – возникновение пакета меньшей длины означает наличие ошибки.
- *DbgSerializer* – позволяет проводить сериализацию/десериализацию объектов-сообщений.
- *DbgConnection* – обертка над IConnection, позволяющая обмениваться не голыми пакетами, а объектами-сообщениями. Использует DbgSerializer для сериализации/десериализации этих объектов. Реализует паттерн Observer для оповещения о приходящих сообщениях. Интерфейс наблюдателя представляется классом *IDbgConnectionListener*.

Листинг объявлений перечисленных классов представлен в приложении А.

3.2 Модуль frontend

frontend – клиентская часть отладчика. Клиент работает в двух потоках. В первом исполнение происходит по следующему циклу:

1. Считать ввод пользователя.
2. Разобрать ввод пользователя и составить запрос. При возникновении ошибки вывести её пользователю и перейти на шаг 1.
3. Отправить сформированный запрос через транспорт.
4. Дождаться получения ответа пользователя во втором потоке.
5. Интерпретировать полученный ответ и вывести пользователю в удобочитаемом виде. Перейти на шаг 1.

Во втором потоке происходит получение ответов и событий от модуля отладчика,

подключенного к исполнительной системе. Если был получен ответ, об этом оповещается первый поток. Если было получено событие, оно интерпретируется и выводится пользователю в удобочитаемом виде. Преобразование ввода пользователя в абстракцию запроса происходит с использованием паттерна Factory[10]. Обмен сообщений с сервером происходит не напрямую через интерфейс IConnection, а через обертку DbgConnection, которая описывается в модуле common и позволяет осуществлять отправку не голых пакетов, а абстракций сообщений отладчика. Прием сообщений осуществляется через реализацию интерфейса-наблюдателя IDbgConnectionListener.

3.3 Модуль backend

backend – серверная часть отладчика. Этот модуль подключается к исполнительной системе. Центральным классом этого модуля является *LunaDebugger*. Объект этого класса отвечает за старт программы на каждом узле после получения соответствующего запроса от клиента. В конструктор объекта передаются:

- Объект-сервис, через который осуществляется обмен сообщениями между узлами кластера
- Реализация интерфейса взаимодействия с ИС (*IFlowManager*)
- Режим отладки
- Объект типа ITransport, который будет использоваться для создания соединения (IConnection) с клиентом
- Строка-адрес, которая будет передана транспорту

Режим отладки описывается перечислением (enumeration) *DbgMode*, в который входит три значения:

- NO_DEBUG (нет отладки). Это означает, что запуск программы происходит в режиме отсутствия отладочных возможностей. Объект LunaDebugger инициирует начало программы без создания какого-либо соединения и отладочных объектов и после этого не взаимодействует с исполнительной системой.
- LISTENING (клиент ожидает соединения). В этом режиме отладчик создаст соединение, подключившись к клиенту по указанному адресу
- ATTACHING (клиент подключится к программе). В этом режиме отладчик запустить программу немедленно, перед этим начав ожидание соединения (т.е.,

например, открыв сокет на порту, указанном в строке-адресе, в случае, если за интерфейсом ITransport скрывается объект SocketTransport).

После старта в режиме, отличном от NO_DEBUG, объект LunaDebugger порождает ряд объектов-наблюдателей, которые будут оповещать о событиях, на которые необходимо реагировать отладчику. Эти объекты реализуют следующие интерфейсы:

- *IUserListener*. Содержит единственный метод onUserRequest, вызывающийся при получении запроса от пользователя. Является наблюдателем объекта *UserConnection*, который оборачивает IConnection с учетом того, что отладчик может отправлять пользователю лишь ответы и события (методы sendResponse, sendEvent), а получать только запросы (метод onUserRequest у интерфейса IUserListener).
- *IFlowManagerListener*. Наблюдатель объекта IFlowManager, представляющего интерфейс взаимодействия отладчика и ИС. Именно через этот интерфейс IFlowManagerListener исполнительная система оповещает отладчик о событиях, на которые он подписался (точки останова, операции над наблюдаемыми фрагментами и т.п.)
- *IDbgClusterListener*: Через этот интерфейс происходит оповещение отладчика о входящих сообщениях с других узлов кластера. Содержит методы onRequest, onResponse, onEvent, принимающие в качестве параметра числовой идентификатор узла-отправителя и сам объект-сообщение.

Листинг объявлений описанных интерфейсов представлен в приложении Б.

В текущей реализации модуля backend в качестве дерева коммуникаций используется бинарное дерево ввиду простоты его реализации. Скорее всего, это не самый эффективный вариант с точки зрения задержек при обработке запросов, однако такое решение вполне годится для начальной реализации. В будущем необходимо провести анализ и выбор подходящих параметров для дерева.

ЗАКЛЮЧЕНИЕ

В работе приведен анализ отладочных средств языков программирования с различными парадигмами с позиции их функциональности, а также некоторых деталей реализации. В результате анализа была сформулирована необходимая функциональность отладчика. Разработана архитектура отладчика для системы фрагментированного программирования, обеспечивающая необходимые требования: переносимость, модифицируемость, распределённость и удалённость. Реализована и интегрирована инфраструктура отладчика для текущей версии исполнительной системы LuNA, реализована простейшая функциональность.

Результаты работы были представлены на 51-ой Международной научной студенческой конференции «Студент и научно-технический прогресс» (МНСК-2013) в виде устного доклада. Тезисы к докладу опубликованы в сборнике материалов конференции.

На защиту выносятся:

- Определение необходимой функциональности отладчика по отношению к парадигме фрагментированного программирования
- Архитектура отладчика
- Алгоритм распределенной обработки отладочных запросов
- Реализация инфраструктуры отладчика

В дальнейшем планируется:

- Завершить реализацию основной функциональности
- Реализовать функциональность, связанную с пассивной поведенческой отладкой
- Детально проработать вопрос активной поведенческой отладки
- Добавить возможность анализа используемых ресурсов

Список использованных источников

1. Синицын С.В., Налютин Н.Ю. Верификация программного обеспечения: Курс лекций. – М.: МИФИ (ГУ), 2006. – 158 с.
2. Романенко, А. А. Средства отладки параллельных программ для мультикомпьютеров (алгоритмы реализации и разработка отладчика): автореф. дис… канд. техн. наук / Алексей Анатольевич Романенко. – Новосибирск, 2004. – 15 с.
3. Б. Страуструп. Язык программирования C++ = The C++ Programming Language / Пер. с англ. – 3-е изд. – СПб.; М.: Невский диалект – Бином, 1999. – 991 с.
4. MPI Forum [Электронный ресурс]. – Режим доступа: http://www.mpi-forum.org/, свободный.
5. В.А. Перепёлкин. Проблема распределения ресурсов мультикомпьютера в технологии фрагментированного программирования // Научный сервис в сети Интернет: поиск новых решений: Труды Международной суперкомпьютерной конференции (17-22 сентября 2012 г., г. Новороссийск). – М.: Изд-во МГУ, 2012, 752 с., с. 398–401.
6. V.E. Malyshkin, V.A. Perepelkin. LuNA Fragmented Programming System, Main Functions and Peculiarities of Run-Time Subsystem - In: Proceedings of the 11th Conference on Parallel Computing Technologis, LNCS 6873 - pp. 53-61, Springer, 2011
7. Java Platform Debugger Architecture [Электронный ресурс]. – Режим доступа: http://docs.oracle.com/javase/6/docs/technotes/guides/jpda/architecture.html, свободный.
8. Charm++ Debugger Manual [Электронный ресурс]. – Режим доступа: http://charm.cs.illinois.edu/manuals/html/debugger/, свободный.
9. Erlang Debugger Reference Manual [Электронный ресурс]. – Режим доступа: http://www.erlang.org/doc/apps/debugger/, свободный.
10. Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес Приемы объектно-ориентированного проектирования. Паттерны проектирования = Design Patterns: Elements of Reusable Object-Oriented Software. – СПб: «Питер», 2007. – 366 с.
11. Джереми Сик, Лай-Кван Ли, Эндрю Ламсдэйн. C++ Boost Graph Library. – Питер, 2006. – 304 с.

Приложение А. Листинг основных классов модуля common

(рекомендуемое)

Представлены только объявления и только public-компоненты классов.

```
class ITransport {
public:
      virtual ~ITransport() {}

      virtual IConnection* accept() =0;
      virtual IConnection* attach(const std::string& address = "") =0;

      virtual void startListening(const std::string& address = "") =0;
      virtual void stopListening() =0;
};

/// ---------------------------------------------------------------------- ///

class IConnection {
public:
      virtual ~IConnection() {
      }

      virtual void close() =0;
      virtual bool isOpen() const =0;

      virtual void sendPacket(const std::vector<char>& packet)
throw(connection_closed_exception) =0;
      virtual std::vector<char> receivePacket()
throw(connection_closed_exception) =0;
};

/// ---------------------------------------------------------------------- ///

class DbgRequest {
public:
      DbgRequest(uint32_t id, uint16_t command, const std::vector<char>&
data);

      uint16_t getCommand() const;
      uint32_t getId() const;

      const std::vector<char>& getData() const;

      DbgResponse accumulateResponses(const DbgResponse& first,
                  const DbgResponse& second) const;

public:
      static const int SUSPEND = 0;
      static const int RESUME = 1;
};

/// ---------------------------------------------------------------------- ///


class DbgResponse {
```

```
public:
      static const uint16_t SUCCESS = 0;
      static const uint16_t BAD_FRAGMENT_ID = SUCCESS + 1;
      static const uint16_t UNKNOWN_COMMAND = BAD_FRAGMENT_ID + 1;
      static const uint16_t BAD_REQUEST_ID = UNKNOWN_COMMAND + 1;
      static const uint16_t ALREADY_SUSPENDED = BAD_REQUEST_ID + 1;
      static const uint16_t ALREADY_RUNNING = ALREADY_SUSPENDED + 1;

public:
      DbgResponse(uint32_t id, uint16_t errorCode = SUCCESS, const
std::vector<char>& data = std::vector<char>());

      uint16_t getErrorCode() const;
      uint32_t getId() const;

      const std::vector<char>& getData() const;
};

/// ------------------------------------------------------------------- ///

class DbgEvent {
public:
      DbgEvent(uint32_t id, uint16_t eventType, const std::vector<char>& data
= std::vector<char>());

      const std::vector<char>& getData() const;
      uint16_t getEventType() const;
    uint32_t getId() const;
public:
    static const uint16_t PROGRAM_FINISHED = 1;
    static const uint16_t BREAKPOINT = 2;
};

/// ------------------------------------------------------------------- ///

class DbgConnection {
public:
      DbgConnection(IConnection*);

      virtual ~DbgConnection();

      void start();
      void stop();
      bool isRunning() const;

      void sendRequest(const DbgRequest& request);
      void sendResponse(const DbgResponse& response);
      void sendEvent(const DbgEvent& event);

      void addListener(IDbgConnectionListener*);
      void removeListener(IDbgConnectionListener*);
};

/// ------------------------------------------------------------------- ///
```

```
class IDbgConnectionListener {
public:
	virtual ~IDbgConnectionListener() {}

	virtual void onRequest(const DbgRequest& request) =0;
	virtual void onResponse(const DbgResponse& response) =0;
	virtual void onEvent(const DbgEvent& event) =0;
};

/// ------------------------------------------------------------------- ///

class DbgSerializer {
public:
	enum MessageType {
		REQUEST, RESPONSE, EVENT
	};

	static std::vector<char> serialize(const DbgRequest& request);
	static std::vector<char> serialize(const DbgResponse& response);
	static std::vector<char> serialize(const DbgEvent& event);

	static MessageType getMessageType(const std::vector<char>&
serializedData);

	static DbgRequest deserializeRequest(const std::vector<char>&
serializedData);
	static DbgResponse deserializeResponse(const std::vector<char>&
serializedData);
	static DbgEvent deserializeEvent(const std::vector<char>&
serializedData);
};
```

Приложение Б. Листинг основных классов модуля backend

(рекомендуемое)

Представлены только объявления и только public-компоненты классов.

```
enum DbgMode {
      LISTENING, ATTACHING, NO_DEBUG
};

class LunaDebugger {
public:
      LunaDebugger(comm::CommService*, IFlowManager*, DbgMode,
                  const std::string& addr = "", ITransport* userTransport =
0);
      virtual ~LunaDebugger();

      void start();
      void wait();
};

/// ---------------------------------------------------------------------- ///

class DbgClusterTransport {
public:
      DbgClusterTransport(comm::CommService* commService);
      ~DbgClusterTransport();

      size_t getClusterSize() const;
      size_t myRank() const;

      void sendRequest(const size_t rank, const DbgRequest& request);
      void sendResponse(const size_t rank,
                  const DbgResponse& response);
      void sendEvent(const size_t rank, const DbgEvent& event);

      void addListener(IDbgClusterListener*);
      void removeListener(IDbgClusterListener*);
};

/// ---------------------------------------------------------------------- ///

class IDbgClusterListener {
public:
      virtual ~IDbgClusterListener() {}

      virtual void onRequest(size_t senderRank, const DbgRequest&) =0;
      virtual void onResponse(size_t senderRank, const DbgResponse&) =0;
      virtual void onEvent(size_t senderRank, const DbgEvent&) =0;
};

/// ---------------------------------------------------------------------- ///
```

```
class UserConnectionManager {
public:
      UserConnectionManager(ITransport*, const std::string& addr = "",
                  bool attach = false);
      virtual ~UserConnectionManager();

      bool start();
      void stop();
      bool isRunning() const;
      bool isConnected() const;

      void sendResponse(const DbgResponse& response);
      void sendEvent(const DbgEvent& event);

      void addListener(IUserListener*);
      void removeListener(IUserListener*);
};

/// ------------------------------------------------------------------- ///

class IUserListener {
public:
      virtual ~IUserListener() {}

      virtual void onUserRequest(const DbgRequest&) =0;
};

/// ------------------------------------------------------------------- ///

class IFlowManager {
public:
      virtual ~IFlowManager() {}

      virtual void startExecution() =0;
      virtual void pauseExecution() =0;
      virtual bool isRunning() =0;

      virtual BpInfo registerBp(const CfId& cfId, const BpType& bpType) =0;
      virtual void unregisterBp(const BpId& bpId) =0;

      virtual void watchCf(const CfId& cfId) =0;
      virtual void watchDf(const DfId& dfId) =0;

      virtual CfInfo getCfInfo(const CfId& cfId) =0;
      virtual DfInfo getDfInfo(const DfId& dfId) =0;

      virtual int getLastError() =0;

      virtual void setListener(IFlowManagerListener*) =0;

public:
      static const int NO_ERROR = 0;
      static const int BAD_ID = 1;
      static const int NON_LOCAL_FRAGMENT = 2;
      static const int PROGRAM_RUNNING = 3;
};

/// ------------------------------------------------------------------- ///
```

```
class IFlowManagerListener {
public:
      virtual ~IFlowManagerListener() {}

      virtual void onProgramFinished() =0;

      virtual void onBreakpoint(const BpInfo& info) =0;

      virtual void onDfCreated(const DfId&, const DfValue& value) =0;
      virtual void onDfUsed(const DfId&) =0;
      virtual void onDfDestroyed(const DfId&) =0;

      virtual void onCfExecutionStarted(const CfId& cfId,
                  const std::vector<DfInfo>& inputDfsInfo) =0;
      virtual void onCfExecutionFinished(const CfId& cfId,
                  const std::vector<DfInfo>& outputDfsInfo) =0;
};
```