Санкт-Петербургский
Государственный Университет

Кафедра Системного Программирования
Программная Инженерия

Кузнецов Илья Александрович

Модификация инструментария Java AWT в графической подсистеме X11 для удовлетворения требованиям проекта OpenJDK CRaC

Отчёт по производственной практике

Научный руководитель:
ассистент А. П. Козлов

Санкт-Петербург
2022

Содержание

Введение

Java — один из самых популярных языков программирования в наше время, занимающий лидирующие позиции уже больше 20 лет [1]. Так, к 2020 г. уже насчитывалось около 6.8 млн. пользователей Java по всему миру. Данный язык программирования получил широкое распространение в таких популярных или развивающихся индустриях, как встроенные системы, инструменты для разработки, облачные сервисы, анализ данных, мобильная разработка и финансы. А программным обеспечением, написанным преимущественно с его применением, стали веб-сервисы, базы данных, системное обеспечение, инструменты разработчиков и другие [2][3].

Исходя из сфер применения, становится ясно, что на Java-приложения накладываются высокие требования по переносимости, а также производительности и надёжности. На данные характеристики напрямую влияет окружение, в котором система разрабатывается и затем используется. Для Java существует множество реализаций, называемых JDK, которые создаются и поддерживаются различными компаниями и сообществами. Один из них — **OpenJDK** [4], являющийся эталонной реализацией (reference implementation) JDK, одним из контрибьюторов которого является Oracle. Любые изменения в язык, стандартную библиотеку или модель программирования Java реализуются сначала здесь. Он является стандартным для большинства UNIX-систем. В основном его предпочитают как производительную платформу для запуска приложений.

Особенностью работы многих программ на Java является достаточно долгое время запуска (startup), зависящее от сложности приложения. Продолжительный старт связан с архитектурой Java: каждое приложение состоит из множества классов, каждый из которых необходимо сначала загрузить, для чего приходится часто обращаться к относительно медленной вторичной памяти, а затем проинициализировать, что в свою очередь может рекурсивно повлечь дополнительные загрузки.

Для ускорения данных этапов в Java реализована JIT-компиляция. Она зачастую даёт большой прирост производительности, так как основывается на статистике исполнения, но, по сути, является эвристикой, которая иногда может быть неточной. И с другой стороны, JIT-компилятор — это тоже процесс, который требует дополнительные ресурсы взамен на ускорение.

Другая особенность приложений на Java — это необходимость предварительного разогрева приложения (warmup) для достижения оптимальной производительности. Для её решения в Java до 17 версии присутствует экспериментальная AOT-компиляция, которая позволяет предварительно сгенерировать нативный

код и затем исполнить его вместо того, чтобы потратиться на JIT-компиляцию во время исполнения. Таким образом, перед выполнением приложения будет потрачено дополнительное время взамен на более быстрый старт и разогрев программы.

В качестве одного из решений проблем медленного старта и необходимости разогрева приложений на Java на основе OpenJDK был создан проект **CRaC — Coordinated Restore at Checkpoint** [5][6]. Во многом данный проект полагается на уже существующую в UNIX технологию **CRIU — Checkpoint and Restore In Userspace** [7][8], предназначенную для сохранения и восстановления процессов в пространстве пользователя. Инструментарий позволяет сохранить состояние одного или группы процессов, а затем возобновить работу с сохранённой позиции, в том числе после перезагрузки системы или на другом устройстве.

Таким образом, можно заморозить виртуальную машину Java (JVM) с работающим приложением и поместить его в постоянное хранилище как набор файлов. Затем файлы можно использовать для восстановления и запуска JVM и приложения с того места, где они были заморожены.

Отличительной особенностью проекта CRIU является то, что он реализован преимущественно в пользовательском пространстве, а не в ядре. CRIU уже стабильно применяется для решения таких задач, как перезапуск операционной системы без прерывания процессов, миграция изолированных контейнеров в реальном времени, ускорение запуска медленных процессов, обновление ядра без перезапуска служб, сохранение состояния задач на случай сбоя, балансировка нагрузки между кластерными узлами, дублирование процессов и развёртывание на удалённом устройстве; используется в таких системах управления контейнерами, как OpenVZ, LXC/LXD и Docker.

CRaC предоставляет в Java две большие операции, основанные на технологии CRIU.

- **Checkpoint** — непосредственное сохранение состояния процессов в образ (файл) и их прерывание.
- **Restore** — восстановление оригинальных процессов из образа так, будто бы они существовали всё время после Checkpoint, но их не планировала операционная система.

Рассмотрим, как CRaC использует эти возможности для JVM. Предположим, что у нас есть некоторое приложение, исполняемое JVM, которое мы предварительно запустили и еще некоторое время разогревали, а затем сделали

Checkpoint на оптимальной производительности (app heated). Тогда в Restore мы получим запуск приложения с загруженными и проинициализированными классами и готовыми JIT-компиляциями. При этом размер образа, зависящий от потребления приложением ресурсов, получается вполне приемлемым.

Таким образом, CRaC решает обозначенные проблемы с недостаточно быстрым запуском Java-приложений и необходимостью их прогрева после запуска для достижения оптимальной производительности [9].

Для того чтобы создавать гибкие и надёжные образы, CRaC реализует два основных принципа.

1. **Java-приложение управляет своим состоянием и ресурсами** в момент Checkpoint и Restore, используя расширяемый механизм оповещения для того, чтобы решать возможные проблемы с внутренним или внешним состоянием программы. Для этого CRaC API предоставляет несколько удобных интерфейсов ресурса (Resource), позволяющих исполнять произвольный код перед сохранением (beforeCheckpoint) и после восстановления (afterRestore).

2. **CRaC помогает находить опасные состояния** и предоставляет детальную отладочную информацию в случае, если приложение недостаточно хорошо управляет своими ресурсами. Он помогает обнаруживать такие ресурсы операционной системы, как открытые сокеты, соединения, файлы и текущие межпроцессные взаимодействия. В случае, если состояние приложения недостаточно устойчиво для Checkpoint или Restore, CRaC может отменить их, чтобы не нарушать работу системы.

Графическая подсистема в Java — является одним из таких ресурсов, связью с внешней средой, для которого необходимо поддержать корректную реинициализацию соединения с графической подсистемой UNIX, которая по умолчанию руководствуется сетевым графическим протоколом X11. При попытке использовать CRaC для любого графического приложения на Java, это приводит к обнаружению внешних связей и отказу в создании образа, поскольку графические подсистемы никак не взаимодействуют с новой функциональностью OpenJDK.

Базовой графической библиотекой в Java является **AWT (Abstract Window Toolkit)**, которая находится на самом нижнем уровне библиотек Java и представляет связь с протоколом X11. От неё зависят другие, более продвинутые, подсистемы, такие как Swing и JavaFX. Обеспечив в AWT поддержку CRaC, можно решить множество проблем с зависящими от него подсистемами.

1. Цели и задачи

Целью данной работы является добавление начальной поддержки графических приложений в проект OpenJDK CRaC, использующих реализацию AWT для протокола X11 (AWT/X11) в UNIX.

Для достижения цели были поставлены следующие задачи.

1. Провести обзор предметной области:
   (a) графической подсистемы UNIX и протокола X11,
   (b) реализации AWT для протокола X11 в OpenJDK.
2. Реализовать начальную поддержку графических приложений AWT/X11, а именно: сохранение образа JVM, исполняющего графическое приложение, и его восстановление с возможностью создания графического интерфейса приложения.

2. Термины и определения

- **Java Development Kit (JDK)** — комплект разработчика приложений для языка Java, включающий в себя компилятор Java, стандартные библиотеки классов Java, примеры, документацию, различные утилиты и исполнительную систему Java Runtime Enviroment (JRE), в которую входит Java Virtual Machine (JVM).

- **Reference implementation (эталонная реализация)** — реализация программного обеспечения, точно соответствующая спецификации некоторого стандарта, либо созданная для демонстрации этой спецификации в действии.

- **Just-In-Time compilation (JIT-компиляция)** — вид динамической компиляции, позволяющий компилировать промежуточный байт-код виртуальной машины в бинарный во время его выполнения. Требует дополнительных затрат производительности во время исполнения.

- **Ahead-Of-Time compilation (AOT-компиляция)** — вид динамической компиляции, позволяющий предварительно компилировать промежуточный байт-код виртуальной машины в бинарный. Не требует дополнительных затрат производительности во время исполнения.

- **App heated (разогретое приложение)** — точка во времени, означающая достижение приложением оптимальной производительности. Является верхней границей суммы двух временных отрезков — startup (время на запуск) и warmup (время на прогрев).

- **Graphical User Interface (GUI)** — человеко-машинный интерфейс (то есть способ взаимодействия людей с компьютерами), использующий окна, значки, меню и многое другое, которым также можно управлять с помощью различных устройств ввода. Графические интерфейсы пользователя резко контрастируют с интерфейсами командной строки (CLI), которые используют только текст и управляются исключительно с помощью клавиатуры.

- **Java Native Interface (JNI)** — стандартный механизм Java для запуска кода под управлением виртуальной машины Java (JVM), который написан на языках C/C++ или ассемблере и скомпонован в виде динамических библиотек, то есть позволяет не использовать статическое связывание. Это

даёт возможность вызова функций на C/C++ из программы на Java, и наоборот.

- **Toolkit (инструментарий)** — группа библиотек, содержащих набор графических элементов управления, используемых для создания графического интерфейса пользовательских приложений.
- **Singletone (одиночка)** — шаблон проектирования программного обеспечения, который ограничивает создание экземпляра класса единственным экземпляром. Это полезно, когда требуется ровно один объект для координации действий в системе.

3. Обзор предметной области

Целью обзора предметной области является достижение понимания реализации AWT и связанных графических подсистем в Java, использующих сетевой графический протокол X11, а также структуры графической подсистемы UNIX.

3.1. X Window System

На данный момент самой популярной в UNIX графической подсистемой является **X Window System**, сокращённо **X**, — это платформонезависимая, распределенная модульная среда. Современная реализация распространяется свободно X.Org Foundation. Система предоставляет приложениям в UNIX-подобных операционных системах платформу для создания графических пользовательских интерфейсов (GUI).

Система X Window была первоначально разработана командой из Массачусетского Технологического Института (MIT) во главе с Бобом Шейфлером и Джимом Геттисом в 1984 году. В своей работе «The X Window System» [10] они определили основные понятия и принципы, архитектуру и протоколы, а также описали возможности развивающейся системы. С тех пор сетевой графический протокол взаимодействия X Window System пересматривался множество раз. Самый последний выпуск - X11, разработанный ещё в 1987 году, который часто модифицировался и используется в наши дни. На основе этого, разработчикам удалось добиться практически полной обратной совместимости с первыми выпусками протокола. X стал де-факто стандартным графическим движком для UNIX-подобных операционных систем.

Стоит отметить, что долгое время разрабатывается Wayland [11] — другой графический протокол для UNIX, не получивший столько большого распространения, как X11. Как и любая относительно новая технология, он имеет свои положительные стороны и недостатки. К тому же, реализован проект XWayland [12], который позволяет работать приложениям на X11 поверх Wayland. Таким образом, данная технология не представляет большого интереса для работы, так как по умолчанию совместима с X11.

Компания Canonical с начала 2010-х годов долгое время разрабатывала свою замену X Window System в дистрибутиве Ubuntu — графический сервер Mir. Для него была также добавлена совместимость с X Window под названием XMir. В ходе проекта поменялись его приоритеты и цели, за счёт чего разработка не была заброшена, однако полностью заменить проверенный временем X так и не удалось.

X Window System обеспечивает следующие базовые функции графической среды: отрисовку изображения, в том числе с применением аппаратного ускорения, и манипулирование окнами на экране, взаимодействие с устройствами ввода, такими как, например, мышь и клавиатура.

Рассмотрим X Window System с концептуальной точки зрения. Система состоит из двух основных компонентов, так как используется клиент-серверная архитектура. Некоторые клиенты могут представлять собой менеджеров окон или рабочего стола.

3.1.1. X Window Server

X Window Server обменивается сообщениями со множеством клиентов одновременно, которыми выступают графические приложения. На сервере располагаются разделяемые между всеми клиентами драйвера устройств, дисплей, устройства ввода (мышь, клавиатура), шрифты, курсоры, цвета, а также ресурсы каждого из приложений, которые представляются окнами, связанными с ними данными и графическими контекстами. Данные и контексты окон активно разделяются с оконным менеджером, который таким образом поддерживает внешний вид системы.

Сервер принимает запросы от X Window Client на создание и закрытие окон, различные манипуляции с ними, отрисовку графики на дисплее, асинхронно обрабатывает их совместно со считыванием пользовательского ввода, используя указанные в запросе ресурсы, и отправляет обратно клиентам изменения в виде событий. Использование графических и иных аппаратных ускорителей в зависимости от доступности соответствующих драйверов — прерогатива X Window Server.

Все окна и их конфигурации в виде контекстов представляются на стороне сервера в виде дерева. На его верхнем уровне находится самое общее окно — дисплей, который покрывает все физические экраны, подключенные к серверу. Соответственно, он может быть только один. Дисплей разделяется между множеством окон, принадлежащих приложениям. В зависимости от используемых технологий отрисовки, приложения в различной степени делятся своими ресурсами с X Window Server и, как следствие, могут не полностью использовать протокол взаимодействия.

3.1.2. X Window Client

X Window Client, являющийся графическим приложением, в свою очередь может обслуживаться несколькими серверами. Клиент во время работы хранит

у себя числовые идентификаторы ресурсов — атомы, которые используются для общения с сервером и соответствуют хранящимся на его стороне. Такое разделение ресурсов вполне оправдано: для гибкости используемого сетевого протокола, минимального трафика и скорости пересылки данных. X Window Client также кэширует часто используемые ресурсы, запросы и команды на своей стороне.

В ходе своего выполнения, графическое приложение отправляет запросы с числовыми идентификаторами затронутых изменениями ресурсов X Window Server всякий раз, когда оно меняет свое графическое состояние или подвергается его изменению со стороны других процессов. Так X Window Client сообщает серверу, какие графические модификации ему необходимо совершить и с какими ресурсами. X Window Server выполняет эти команды совместно с драйверами устройств и отвечает событиями. Так сервер указывает, как именно следует реагировать приложению на произошедшие изменения. События также включают в себя ввод данных с каких-либо устройств. Стоит отметить, что обработка событий — прерогатива X Window Client.

3.1.3. X Window Manager

Определением конкретных деталей интерфейса пользователя занимается **X Window Manager** — менеджер окон, реализаций которого разработано множество [13]. Их основная функция – обеспечивать управление другими окнами – перемещением, изменением размеров, сворачиванием и разворачиванием окон, отрисовкой обрамлений окон, управление передачей фокуса от окна к окну, а также управление Z-порядком размещения окон. Соответственно, в один момент времени для одного дисплея может быть использован только один менеджер окон.

Управление окнами для X Window System было сознательно отделено от программного обеспечения сервера, обеспечивающего графический дисплей, чтобы пользователь мог выбирать между различными сторонними оконными менеджерами, которые также являются клиентами для X Window Server. Это отличает Linux от Windows и MacOS, где менеджер окон фактически монолитно склеен с остальными частями графической подсистемы.

Реализации X Window Manager отличаются друг от друга несколькими параметрами.

- Возможности настройки внешнего вида и функциональности.
- Потребление оперативной памяти и прочих ресурсов.

- Степень интеграции со средой рабочего стола, предоставляющей более полный набор средств для взаимодействия с операционной системой и различными приложениями.

По этой причине внешний вид программ в среде X может очень сильно различаться в зависимости от возможностей и настроек конкретного оконного менеджера.

3.1.4. Архитектура X Window System

Более наглядно архитектура системы показана на следующей диаграмме.

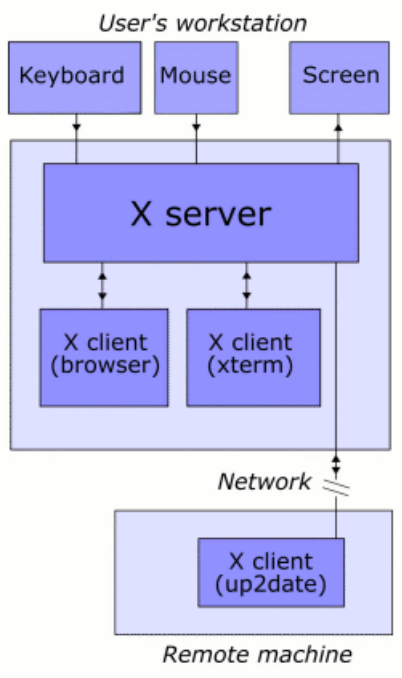

Рис. 1: Материализованная архитектура X Window System.

На данной диаграмме демонстрируется сценарий использования X Window Server с двумя локальными и одним удалённым клиентом, совместно разделяющих одну клавиатуру, мышь и дисплей. Один из клиентов может являться менеджером окон или рабочего стола. Однонаправленное общение X Window Server с клавиатурой и мышью происходит в виде поступления системных вызовов (прерываний) на драйвера соответствующих устройств. В одностороннем порядке

происходит и отображение графики на экран через драйвер этого устройства. Дуплексное общение происходит между сервером и клиентами, в том числе за пределы локальной сети. Для этого применяется графический сетевой протокол X11.

3.1.5. Протокол X11

Соединение по запросно-событийной модели между X Window Server и X Window Client, описанной в предыдущих разделах, обеспечивает сетевой графический протокол **X11**.

Актуальной версией протокола X11 является выпуск 7.7 [14], появившийся в 2012 году. Именно он является центральным компонентом всей системы, так как обеспечивает высокую гибкость, переносимость и, к тому же, на его основе создано множество графических инструментов и библиотек более высокого уровня, многократно расширяющих базовую функциональность, ускоряющих скорость работы и улучшающих опыт использования X Window System как разработчиками, так и пользователями. Именно поэтому обновление протокола происходит консервативно.

В операционных системах UNIX реализация клиентской части сетевого графического протокола X11 по умолчанию представлена в виде одноименной базовой библиотеки, которую также называют **libX11** или **Xlib**, либо её улучшенной версии — **libXCB** или **XCB** (X C-language Binding). Реализованы на C.

libX11 позволяет программам абстрагироваться и не связываться с более низкоуровневыми функциями протокола X11 и за счёт этого обеспечивает сетевую прозрачность X11, то есть позволяет приложениям работать как с локальным X Window Server, так и удалённо, причем делается это незаметно для программы. Таким образом, команды могут передаваться как через разделяемую память или локальное соединение, если клиент и сервер запущены на одном устройстве, так и по сети — при этом клиент и сервер могут быть запущены на разных машинах.

3.2. Связь X Window System с Java

Java Native Interface (JNI) — стандартный механизм Java для запуска кода под управлением JVM, который написан на языках C/C++ или ассемблере и скомпонован в виде динамических библиотек, то есть позволяет не использовать статическое связывание. Это даёт возможность вызова функций на C/C++ из программы на Java, и наоборот.

При помощи JNI базовая библиотека протокола X11 — libX11 подключается к инструментарию AWT и любой другой графической подсистеме Java. Таким

образом, функциональность X Window Client становится доступной на более высоком уровне.

3.3. AWT и связанные графические подсистемы Java

Пользуясь аппаратом JNI, на уровне стандартных или подключаемых библиотек языка Java реализовано большое множество **инструментариев для разработки (Toolkit)**, представляющих графические подсистемы. Те из них, что входят в JDK, обобщенно называют Java Foundation Classes (JFC).

AWT, **Swing** и **JavaFX**, разработанные Sun Microsystems, — являются стандартными инструментариями, которые используются почти в каждом графическом приложении на Java. Они образуют иерархию переиспользования функциональности и выпускались в порядке их перечисления. Все они используют протокол X11 с помощью AWT для работы с графической подсистемой операционной системы.

3.3.1. Abstract Window Toolkit

AWT (Abstract Window Toolkit) — первая и наиболее тяжеловесная технология, которая использует для отрисовки графические возможности операционной системы. Она претерпела большое число изменений и модификаций за свою длинную историю. На сегодня, инструментарий AWT — это фундамент для графических подсистем Java.

Фактически, это слой между реализациями графики в Java и JNI, который вызывает нативные функции протокола X11 и предоставляет два программных интерфейса.

- Собственный набор компонентов графического интерфейса.
- Слой между libX11 и другими графическими подсистемами Java.

Мультиплатформенность AWT обеспечивается при помощи независимости реализаций её графических компонентов и библиотек в Java от нативной части, которая разрабатывается специфично для каждой операционной системы. Для этого была создана обширная система нативных пиров (Peer). Логика их работы показана на следующем рисунке.

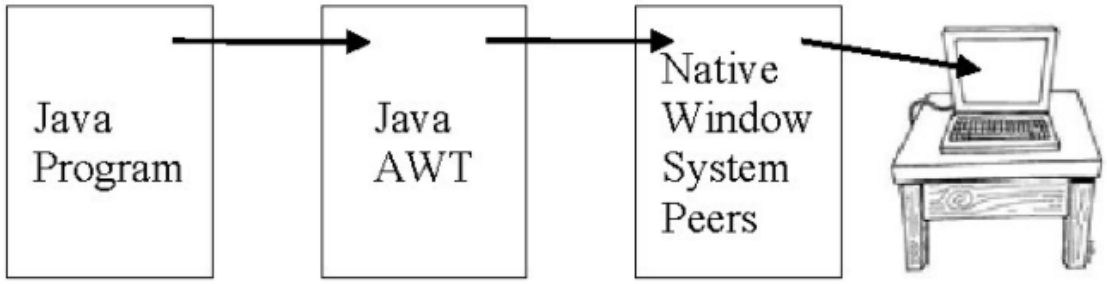

Рис. 2: Пиры в AWT.

Нативные пиры реализуются отдельно для каждого компонента, поэтому добавление в AWT поддержки очередной операционной системы — объемная задача. Более поздние графические подсистемы Java стараются вынести самые ресурсоемкие задачи на сторону X Window Client, оставив только базовое взаимодействие с операционной системой через AWT, поэтому их кроссплатформенность гораздо выше.

Таким образом, AWT использует множество платформно-зависимых пиров для достижения переносимости своих графических элементов. За счёт этого программы получаются похожи на другие в рамках конкретной системы и могут запускаться на разных устройствах. Однако для этого программные интерфейсы компонентов были унифицированы, вследствие чего их функциональность получилась достаточно урезанной.

Представленный в AWT ограниченный набор элементов графического интерфейса частично показан в виде иерархии на следующей диаграмме.

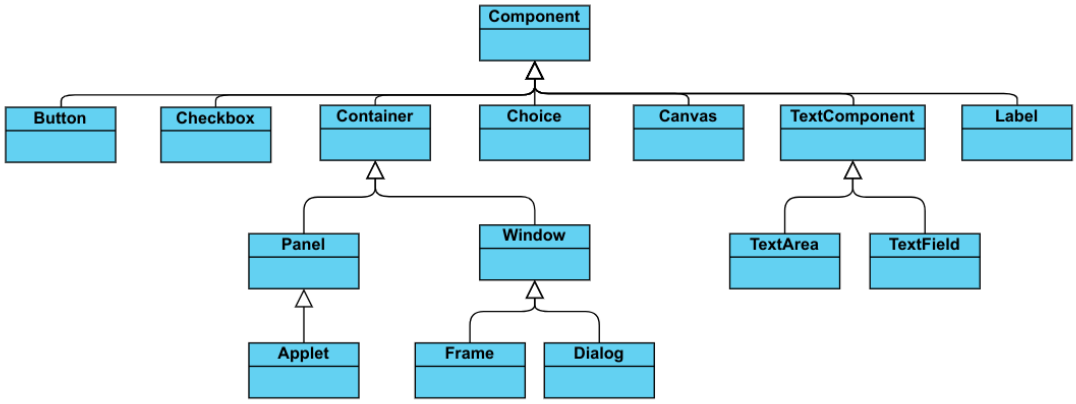

Рис. 3: Сокращённая иерархия компонентов AWT.

В AWT реализованы базовые компоненты, такие как кнопки, надписи, текст, полотна для рисования, элементы с выбором и другие. Среди всех компонентов необходимо выделить контейнеры. Они служат для группировки других графических элементов с помощью менеджеров компоновки и повышения уровня абстракции при разработке соответствующих приложений. Окно (Window) служит базовым классом для всех окон, порождаемых из Java. Разумеется, он также является интерфейсом к соответствующему окну операционной системы. Класс рамки (Frame) предназначен для создания полнофункциональных окон приложений – с полосой заголовка, рамкой, кнопками закрытия, минимизации и максимизации окна.

Графические элементы приложения представляются в виде дерева и обладают такими свойствами, как положение, размер, видимость, доступность, цвета, шрифт. Их рисование на дисплее может инициироваться как самим приложением, так и операционной системой. При этом используется активное взаимодействие по протоколу X11: приложение и X Window Server обмениваются необходимыми для этого ресурсами, запросами и событиями.

Помимо предоставления графических компонентов, AWT реализует в Java графическое ядро и другие важные стандартные библиотеки. Оно состоит из уже упоминавшихся пиров (Peer), различных менеджеров компоновки, геометрических объектов (Point, Rectangle и Polygon), изображений (Image), цветов (Color), шрифтов (Font и FontMetrics). Имеются интерфейсы к устройствам ввода, области уведомлений.

Классы графики (Graphics) и событий (Event) имеют решающее значение для системы рисования и обработки событий AWT. Объект Graphics представляет контекст рисования — без объекта Graphics никакая программа не может рисовать на экране. Объект Event представляет действие пользователя: щелчок мышью, изменение размеров окна, наложение окон и множество других сценариев. Обработчик событий работает асинхронно, уведомляя слушателей. Как рассматривалось ранее, большинство событий приходят от X Window Server в ответ на отправленные запросы по протоколу X11.

Неотъемлемой частью графического ядра является удаление устаревших или временных объектов. Объекты AWT в Java удаляются стандартным сборщиком мусора, но для нативных ресурсов, зависящих от операционной системы, реализовано специальное удаление через JNI (Disposer). Так же делегируется очистка библиотеке libX11.

Таким образом, AWT, являясь слоем между libX11 и другими графическими подсистемами Java, также предоставляет им функциональность графического ядра и позволяет наследоваться от своих компонентов.

3.3.2. Связанные инструментарии Java

Вслед за AWT была разработана новая переносимая графическая библиотека компонентов — **Swing**, полностью написанная на Java и использующая AWT в своей работе. Для отрисовки используется более легковесная технология Java2D на стороне приложения, выступающего X Window Client. Это позволило больше абстрагироваться от протокола X11, оставив только самые важные запросы, затрагивающие графику операционной системы. Как результат, набор стандартных компонентов подсистемы значительно превосходит AWT по разнообразию и функциональности, а также обладает легковесными компонентами и реализует паттерн MVC. Swing позволяет легко создавать новые компоненты, наследуясь от существующих, и поддерживает различные стили и формы.

Флагманской технологией является **JavaFX**, которая позволяет создавать приложения с красивой графикой благодаря использованию аппаратного ускорения и возможностей GPU. Также она обладает кроссплатформенностью и будет работать везде, где установлена исполняемая среда Java (JRE). JavaFX предоставляет большой набор элементов управления, возможности по работе с мультимедиа, двухмерной и трехмерной графикой, декларативный способ описания интерфейса с помощью языка разметки FXML, возможность стилизации интерфейса с помощью CSS. Обеспечена интеграция с библиотеками Swing и, как следствие, с AWT.

Таким образом, более поздние инструментарии исправляют недостатки AWT, создают новые, более гибкие, красивые и легковесные компоненты, максимально дистанцируются от взаимодействия с системой, стараясь выполнять действия по рисованию на клиентской стороне.

3.3.3. Архитектура графических подсистем Java

Исходя из предыдущих разделов, графические подсистемы Java выстраивают одинаковую модель взаимодействия с графикой операционной системы. Для этого они используют функциональность AWT и более низкоуровневую библиотеку libX11. На следующей диаграмме изображена стандартная архитектура графической подсистемы в Java.

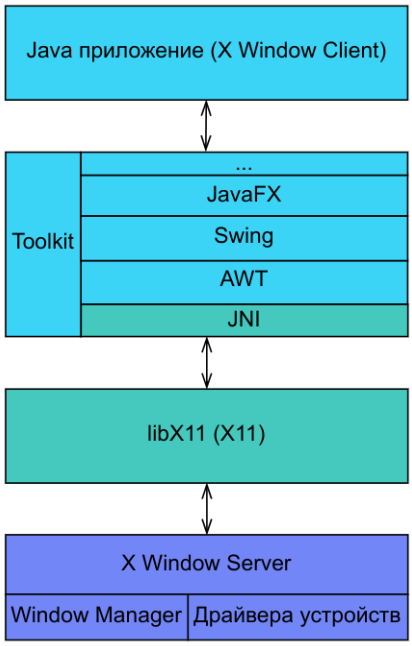

Рис. 4: Абстрактная схема графической подсистемы Java.

На рисунке отмечается, что любое приложение в Java, представляющее X Window Client, общается с X Window Server через графические инструментарии, на самом нижнем уровне которых находится AWT. Он использует JNI, располагающийся над более низкоуровневыми функциями библиотеки libX11, которые обеспечивают общение с X Window Server по протоколу X11.

Подводя итог обзора предметной области, стоит отметить, что модификация AWT/X11 — это важный шаг для поддержки графических приложений в OpenJDK CRaC, поскольку частично будут поддержаны и другие популярные инструментарии Java.

4. Ход работы

4.1. Подготовка окружения

Первым этапом необходимо было подготовить окружение, в котором будет происходить дальнейший поиск ошибок и реализация решения. В него входит подготовка отладочной сборки OpenJDK 17 с проектом CRaC [15] и системной библиотеки libX11, связываемых динамически. Помимо этого, потребовалась конфигурация Xscope и Xtrace — анализаторов трафика по протоколу X11 для более детального анализа общения клиента и сервера, а также логирования ошибок. Для удобства сборки и отладки, были реализованы скрипты [16].

4.2. Приложения для отладки и тестирования

Для отладки ошибок, возникающих в AWT и связанных с ним графических подсистемах OpenJDK из-за наличия активного соединения по протоколу X11 в момент взаимодействия с CRaC, было предложено разработать простейшие графические приложения [16], использующие AWT и Swing. Они же переиспользовались для тестирования.

Данные приложения обеспечивают появление на дисплее одиночного окна с системной рамкой и выполняют операцию Checkpoint — сохранение образа JVM во вторичную память. Затем следует Restore — восстановление образа JVM в процесс и открытие аналогичного окна. После этого приложением можно пользоваться до завершения работы через кнопку закрытия.

4.3. Реинициализация соединения по X11 в AWT с помощью CRaC

При запуске тестовых приложений в первоначальном состоянии наблюдался отказ CRaC от выполнения операции CheckpointRestore, поскольку соединение по X11 не было разорвано. Таким образом, графический интерфейс приложения не закрывался при попытке сделать Checkpoint, что не позволяло корректно остановить JVM.

4.3.1. GraphicsEnvironment, X11GraphicsEnvironment

За открытие и закрытие дисплея (Display), абстракции над соединением в X11, покрывающей всю видимую часть экрана, отвечает низкоуровневая библиотека

libX11. На стороне AWT соответствующую функциональность предоставляет **GraphicsEnvironment**.

От GraphicsEnvironment наследуются соответствующие конкретной системе реализации графического окружения. В зависимости от платформы, создается подходящий singletone-объект. Для рассматриваемой в данной работе UNIX, это **X11GraphicsEnvironment**.

Чтобы обеспечить подключение и отключение от X Window Server с помощью CRaC, было решено добавить соответствующие нативные функции в JNI для AWT/X11 [17]. Для разрыва соединения до Checkpoint была использована стандартная функция библиотеки libX11 — *XCloseDisplay*. Чтобы обеспечить потокобезопасность обращений к деинициализированному дисплею на нативном уровне во время CheckpointRestore, был использован мьютекс для POSIX-потоков, исключающий конкурентный доступ к дисплею [18]. Для повторной инициализации соединения после Restore была использована функция, отвечающая за это по умолчанию в AWT/X11 [19]. В неё была добавлена разблокировка дисплея для других потоков, когда он станет доступен. Данный подход позволил создать реинициализацию соединения по X11, идентичную первоначальному открытию дисплея. А за счёт повторного считывания свойств JVM, восстановление возможно и в другом графическом окружении, как этого требует CRaC.

Чтобы перестроить на новом соединении связанные с нативной частью структуры, хранящие информацию о дисплее (Display), графических устройствах (GraphicsDevice) и экранах (Screen), подключенных к системе, необходимо было реинициализировать после Restore singletone-объект X11GraphicsEnvironment [20], инкапсулирующий данную информацию.

В рамках данной работы, полноценно была поддержана реинициализация X11GraphicsEnvironment. Она вызывается своим суперклассом GraphicsEnvironment, где хранится объект ресурса CRaC [21], участвующий в операции CheckpointRestore. В графическом окружении с помощью виртуальных методов был создан общий интерфейс для реинициализации и на других системах, если CRaC будет их поддерживать в будущем.

4.3.2. XErrorHandlerUtil

В ходе работы над реинициализацией дисплея было обнаружено, что за ним закрепляется обработчик ошибок, управляемый **XErrorHandlerUtil**. В нем кешируются возникающие исключения и хранятся текущие ошибки, поскольку он позволяет обернуть блок кода для их обнаружения и определить логику работы

с ними.

Чтобы поддержать работу обработчика ошибок на новом дисплее после операции CheckpointRestore, было реализовано удаление всех сохраненных ошибок и логическая деинициализация XErrorHandlerUtil до Checkpoint [22]. Его повторная инициализация происходит при создании нового дисплея после Restore в X11GraphicsEnvironment [23].

Теперь при запуске тестовых приложений графический интерфейс закрывался, но возникала остановка работы после Restore: X Window Server при создании нового окна принимал запросы с ресурсами, неизвестными ему после грубого пересоздания соединения. В файлах трассировки протокола X11, полученных при помощи Xscope и Xtrace, наблюдались ошибки *BadWindow*, *BadDrawable*, *BadGC*, *BadMatch*, *BadCursor*, *BadAtom*, а также некорректное завершение работы *Connection reset by client*. Это означало, что в AWT/X11 оставались объекты, в том числе на нативном уровне, требующие удаления и повторной инициализации на новом соединении.

4.3.3. Window

Для исправления возникающих ошибок было решено модифицировать **Window** — компонент AWT. Его статический список *allWindows* содержит в себе все окна, существующие в приложении. Это означало, что на стороне AWT большое число графических объектов оставались нетронутыми.

Каждое окно имеет стандартный метод *dispose*, закрывающий его и все зависящие от него окна. Важной частью данного метода является удаление нативных пиров, мешающих JVM корректно остановится до Checkpoint. Также у окон имеется поле *disposerRecord*, метод *dispose* которого позволяет сделать их недоступными из контекста приложения.

Было решено реализовать последовательное закрытие всех существующих окон приложения как компонентов AWT до Checkpoint, используя описанные методы [24]. Также возвращаются к начальным значениям статистические данные класса.

Таким образом, после внесения данных изменений JVM стала корректно останавливаться во время операции CheckpointRestore, так как нативные пиры отсутствовали. Ошибки *BadDrawable*, *BadGC*, *BadMatch* были исправлены за счёт очистки соответствующих графических объектов при удалении окон. Однако в файлах трассировки протокола X11 все ещё наблюдались проблемы *BadWindow*, *BadCursor*, *BadAtom*, а также некорректное завершение работы *Connection reset by client*. Поиск ещё не очищенных окон был продолжен.

4.3.4. XRootWindow, XWindow, XBaseWindow

Со стороны X11, окна представлены глубокой иерархией наследования, которую затем использует AWT. **XRootWindow**, **XWindow**, **XBaseWindow** являются как раз теми классами, в которых хранится статическая информация, подлежащая реинициализации.

XRootWindow представляет собой singletone-объект, инкапсулирующий главное окно, которое X11 предоставляет графическому приложению. Оно является внутренним объектом X11, поэтому недоступно из AWT. Однако именно от него зависят все остальные окна, поэтому его реинициализация происходит, когда все другие окна AWT/X11 недоступны. Для удаления XRootWindow до Checkpoint был использован метод *destroy* [25]. Для восстановления главного окна после Restore была использована его стандартная инициализация [26].

Так как XRootWindow является последним закрывающимся окном перед Checkpoint, из него вызывается реинициализация XWindow [27] — сброс статических данных к начальным значениям и очистка атомов, необходимых для поддержания связи с Window Manager.

Вверх по иерархии наследования, очистка объектов соединения до Checkpoint переходит от XWindow к XBaseWindow [28], где тоже статически содержится один из атомов.

Стоит отметить, что удаленные атомы инициализируются заново в ленивой форме, то есть во время повторного создания окон.

Таким образом, описанная цепочка реинициализаций была реализована, а ошибка *BadWindow* и некорректное завершение работы *Connection reset by client* были исправлены за счет переоткрытия XRootWindow в правильной последовательности относительно других окон AWT/X11. Это позволило окнам начать восстанавливаться и функционировать на новом соединении, однако в файлах трассировки продолжали наблюдаться проблемы *BadCursor* и *BadAtom*. Это приводило к отсутствию курсора, иконки и панели управления у окон приложения.

4.3.5. Cursor, XGlobalCursorManager

При дальнейшем поиске проблем с использованием неизвестных курсоров на новом соединении, было выявлено, что компонент AWT — **Cursor**, отвечающий за его внешний вид и свойства, кеширует в своих статических массивах когда-либо созданные курсоры. Именно поэтому AWT/X11 пытается их переиспользовать на новом соединении.

Чтобы это исправить, перед Checkpoint было добавлено обнуление всех сохраненных в массивах курсоров [29]. Их реинициализация после Restore происходит автоматически при повторном создании окон, поскольку каждое окно может использовать свой собственный курсор.

За создание предопределенных в AWT курсоров, управлением их нативной частью и связью с X11 отвечает singletone-объект **XGlobalCursorManager**. Так как он тоже кеширует данные, но на нативном уровне, до Checkpoint происходит удаление данного объекта [30]. Его повторная инициализация после Restore происходит самостоятельно при попытке окна создать новый курсор.

Реинициализация Cursor вызывается из Window из-за наличия описанной связи между данными компонентами AWT.

Таким образом, проблема *BadCursor* была решена при помощи очистки закешированных данных курсоров в AWT/X11 как на уровне Java, так и в нативной части. Это позволило окнам корректно их реинициализировать на новом соединении, поэтому они начали отображаться. В файлах трассировки оставалась только ошибка *BadAtom*, у окон приложения отсутствовала иконка и панель управления.

4.3.6. XWM, XAtom

За управление окнами внутри графической оболочки системы отвечает Window Manager. Он использует иконку приложения для отображения не только на окнах, но и на экране приложений, областях уведомлений и состояния, панели задач. Его функциональностью является предоставление окнам системных панелей управления, либо возможности использовать собственные реализации.

В X11 существует класс **XWM**, который инкапсулирует всю логику взаимодействия приложения с системным оконным менеджером. При инициализации определяется используемый в системе Window Manager, протоколы взаимодействия с ним, а также большое количество атомов.

Чтобы поддержать реинициализацию XWM, до Checkpoint была добавлена его логическая деинициализация, заключающаяся в переводе XWM в неопределенное для текущей системы состояние, приостановка работы, а также сброс статических данных и проинициализированных протоколов взаимодействия [31]. После Restore была повторно использована стандартная инициализация [32], которая позволит восстанавливать приложение и в другой графической оболочке, как этого требует CRaC.

Все созданные в ходе работы приложения атомы регистрируются в статических словарях класса **XAtom**. Причем это верно как для атомов XWM, так и

для окон и графического инструментария в целом.

В связи с тем, что зарегистрированные на старом соединении XAtom некорректно использовать на новом, было решено производить очистку статических словарей до Checkpoint [33]. После Restore практически все атомы реинициализируются в ленивой форме, за исключением тех, для которых соответствующая логика не определена. Такими атомами оказались объявленные в XWM, поэтому после Restore для них была реализована повторная реинициализация [34].

С учетом наличия сильной связи между двумя рассмотренными классами, реинициализация XAtom вызывается из XWM в правильном порядке.

Таким образом, произведенные изменения помогли исправить ошибку *BadAtom*, так как все XAtom стали инициализироваться заново на новом соединении, а также вернули панель управления окнам приложения и иконку во все части графической оболочки системы за счет реинициализации XWM.

4.3.7. XToolkit

XToolkit — это графический инструментарий X11, лежащий в основе AWT. Он отвечает за обработку поступающих событий от X Window Server и оповещение подписанных слушателей, синхронизацию данных с нативной частью и сервером, клавиатуру и мышь, нативные пиры и соответствующие им компоненты и окна, а также управление потоками AWT/X11.

В XToolkit тоже производится реинициализация. Перед Checkpoint он блокирует свой бесконечный цикл обработки событий (Event loop), переводя отвечающий за это поток в состояние ожидания, с помощью метода *destroy* удаляет окна X11 и нативные пиры, удостоверяясь в их отсутствии в приложении, закрывает специальные нативные пиры, приводит статические данные к значениям по умолчанию и отключается от клавиатуры и мыши [35]. После Restore восстанавливаются данные, зависящие от нового соединения, подключение к клавиатуре и мыши, разблокируется Event loop и начинается обработка событий [36].

Учитывая назначение XToolkit, было решено сделать его ресурсом CRaC [37], участвующим в операции CheckpointRestore. Из него производится реинициализация Window, XRootWindow, XWM, XErrorHandlerUtil, а также всех зависящих от них классов AWT/X11, как было описано в предыдущих разделах. Порядок выполнения CheckpointRestore выбран неслучайно: он учитывает связи между реинициализируемыми классами так, чтобы все объекты соединения были удалены до Checkpoint только когда они больше не используются, а также восстановлены после Restore перед тем, как они начнут снова использоваться. Некоторые из них инициализируются в ленивой форме самостоятельно,

как было сказано ранее.

4.3.8. Disposer

При последующем анализе решения было выявлено, что перед Checkpoint механизм удаления ресурсов AWT/X11 — **Disposer** не всегда очищал все объекты в соответствующей ему очереди ссылок (ReferenceQueue) до разрыва соединения. Соответственно, в некоторых случаях после Restore старые запросы на очистку отправлялись по новому соединению с X Window Server. Поскольку они не имели смысла в рамках протокола X11, сервер отвечал событием об ошибке, которое приводило к экстренному закрытию нового соединения и, как следствие, завершению приложения.

Поскольку Disposer — это часть механизма сборки мусора в OpenJDK, занимающийся обработкой и очисткой найденных GC недостижимых объектов AWT/X11, то для гарантированного завершения его работы перед Checkpoint была рассмотрена более общая схема.

- Вызов сборки мусора и ожидание получения списка недостижимых объектов в ReferenceHandler от JVM [38].
- Ожидание прихода всех ссылок в соответствующие им ReferenceQueue [39] и затем полная их очистка.

Описанный алгоритм реализован с использованием ресурсов CRaC: путём вызова GC, синхронизации потока CRaC с ReferenceHandler и соответствующими ReferenceQueue. Стоит отметить, что этот подход оказался полезен не только для Disposer, но и других объектов в Java, занимающихся обработкой ссылок [40]. Также он позволил убрать лишний «мусор», ранее попадавший в образ. Таким образом, эта очистка стала обязательным этапом при подготовке приложения к сохранению в образ.

Относительно AWT/X11, данная логика позволила гарантировать выполнение операций удаления объектов соединения по X11 до Checkpoint и решила проблему очистки нативных объектов старого соединения на новом. Для этого был добавлен соответсвующий ресурс CRaC [41].

При дальнейшем тестировании, оптимизации и минимизации кода других проблем выявлено не было. Таким образом, цель работы была достигнута путём реинициализации описанных элементов соединения по X11 и компонентов AWT, а также исправления сопутствующих проблем.

4.4. Архитектура решения

На основе проведенных изменений, описанных в предыдущих разделах, была построена диаграмма классов UML с выделенными компонентами и показаны образовавшиеся связи.

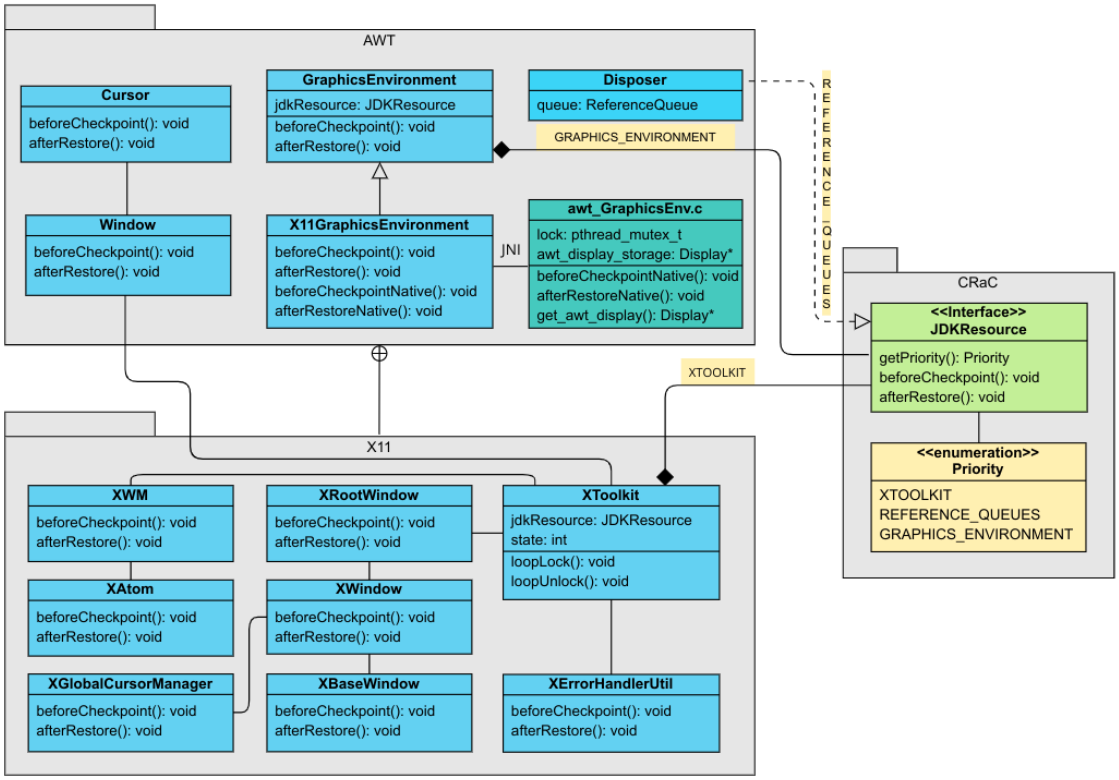

Рис. 5: Архитектура реинициализации соединения по X11 в AWT с помощью CRaC.

Чтобы избежать излишней детализации, на данной диаграмме были изображены не все классы и их нативные части, поля, методы и связи, затронутые модификацией. Однако показано все, что необходимо для представления решения.

Так, на уровень CRaC были добавлены три ресурса — XToolkit [37], Disposer [41] и GraphicsEnrironment [21], обеспечивающих корректность и порядок реинициализации AWT/X11.

5. Результаты

Для добавления начальной поддержки графических приложений в проект OpenJDK CRaC, использующих реализацию AWT для протокола X11 (AWT/X11) в UNIX были выполнены следующие задачи.

1. Проведён обзор предметной области:
   (a) графической подсистемы UNIX и протокола X11,
   (b) реализации AWT для протокола X11 в OpenJDK.
2. Реализована начальная поддержка графических приложений AWT/X11, обеспечивающая сохранение образа JVM, исполняющего графическое приложение, и его восстановление с возможностью создания графического интерфейса приложения. Для этого были модифицированы следующие классы и их нативные части:
   (a) AWT — GraphicsEnvironment, X11GraphicsEnvironment, awt.h, awt_GraphicsEnv.c, Window, Cursor, Disposer;
   (b) X11 — XToolkit, XRootWindow, XWindow, XBaseWindow, XGlobalCursorManager, XWM, XAtom, XErrorHandlerUtil;
   (c) CRaC — JDKResource.

Интеграция изменений в официальный репозиторий проекта CRaC [6] на момент завершения работы над производственной практикой всё еще продолжается из-за неопределенного срока рассмотрения Oracle Contributor Agreement (OCA).

Список литературы

[1] *Самые популярные языки программирования 1965-2021* (2021) // Сайт StatisticsAndData // Просмотрено 20.10.2021
https://statisticsanddata.org/data/the-most-popular-programming-languages-1965-2021/

[2] *Экосистема Java* (2020) // Исследование технологий и рынка // Блог компании JetBrains // Просмотрено 21.10.2021
https://blog.jetbrains.com/ru/idea/2020/09/a-picture-of-java-in-2020/

[3] *Экосистема разработки* (2020) // Исследование главных тенденций и популярных технологий // Блог компании JetBrains // Просмотрено 21.10.2021
https://www.jetbrains.com/ru-ru/lp/devecosystem-2020/

[4] *OpenJDK — open-source реализация языка Java* // Официальная страница проекта // Сайт OpenJDK // Просмотрено 22.10.2021
https://openjdk.java.net/

[5] *Проект CRaC* // Главная страница проекта // Сайт OpenJDK // Просмотрено 25.10.2021
https://openjdk.java.net/projects/crac/

[6] *Проект CRaC* // Исходный код проекта // Сайт GitHub // Просмотрено 25.10.2021
https://github.com/openjdk/crac

[7] *Проект CRIU* // Главная страница проекта // Сайт CRIU // Просмотрено 25.10.2021
https://criu.org/Main_Page

[8] *Проект CRIU* // Исходный код проекта // Сайт GitHub // Просмотрено 25.10.2021
https://github.com/checkpoint-restore/criu

[9] *Производительность CRaC* (2020) // Оценка производительности, сравнение с другими подходами и примеры реализаций // Сайт GitHub // Просмотрено 27.10.2021
https://github.com/CRaC/docs#results

[10] *Robert W. Scheifler, Jim Gettys: The X Window System* (1986) // Описание X Window System // Цифровая библиотека ACM // Просмотрено 5.11.2021
https://dl.acm.org/doi/abs/10.1145/22949.24053

[11] *Графический протокол Wayland* // Альтернативный протокол для UNIX // Сайт Freedesktop // Просмотрено 1.12.2021
https://wayland.freedesktop.org/

[12] *Проект XWayland* // Совместимость протоколов X11 и Wayland // Просмотрено 1.12.2021
https://wayland.freedesktop.org/xserver.html

[13] *X Window Managers* // Список графических менеджеров X // Сайт Xwinman // Просмотрено 2.12.2021
http://www.xwinman.org/

[14] *X Version 11, Release 7.7* // Актуальный выпуск протокола X11 // Сайт Xorg // Просмотрено 3.12.2021
https://www.x.org/releases/X11R7.7/doc/xproto/x11protocol.html

[15] *OpenJDK 17 с проектом CRaC* // Отладочная сборка // Сайт GitHub // Просмотрено 12.04.2022
https://github.com/kznts9v-1lya/crac/tree/x11-dev

[16] *Инструменты для отладки AWT/X11* // Отладка графических подсистем // Сайт GitHub // Просмотрено 13.04.2022
https://github.com/kznts9v-1lya/crac/blob/x11-dev/cracui

[17] *Реинициализация X11GraphicsEnvironment* // Нативная часть метода beforeCheckpoint // Сайт GitHub // Просмотрено 23.05.2022
https://github.com/kznts9v-1lya/crac/blob/x11-dev/src/java.desktop/unix/native/libawt_xawt/awt/awt_GraphicsEnv.c#L813

[18] *Реинициализация X11GraphicsEnvironment* // Нативная часть синхронизации // Сайт GitHub // Просмотрено 23.05.2022
https://github.com/kznts9v-1lya/crac/blob/x11-dev/src/java.desktop/unix/native/libawt_xawt/awt/awt_GraphicsEnv.c#L72

[19] *Реинициализация X11GraphicsEnvironment* // Нативная часть инициализации дисплея // Сайт GitHub // Просмотрено 23.05.2022
https://github.com/kznts9v-1lya/crac/blob/x11-dev/src/java.desktop/unix/native/libawt_xawt/awt/awt_GraphicsEnv.c#L720

[20] *Реинициализация X11GraphicsEnvironment* // Высокоуровневая инициализация // Сайт GitHub // Просмотрено 23.05.2022
https://github.com/kznts9v-1lya/crac/blob/x11-dev/src/java.desktop/unix/classes/sun/awt/X11GraphicsEnvironment.java#L217

[21] *Реинициализация GraphicsEnvironment* // Resource для операции CheckpointRestore // Сайт GitHub // Просмотрено 23.05.2022
https://github.com/kznts9v-1lya/crac/blob/x11-dev/src/java.desktop/share/classes/java/awt/GraphicsEnvironment.java#L114

[22] *Реинициализация XErrorHandlerUtil* // Метод beforeCheckpoint // Сайт GitHub // Просмотрено 23.05.2022
https://github.com/kznts9v-1lya/crac/blob/x11-dev/src/java.desktop/unix/classes/sun/awt/X11/XErrorHandlerUtil.java#L103

[23] *Реинициализация XErrorHandlerUtil* // Вызов инициализации в X11GraphicsEnvironment // Сайт GitHub // Просмотрено 23.05.2022
https://github.com/kznts9v-1lya/crac/blob/x11-dev/src/java.desktop/unix/native/libawt_xawt/awt/awt_GraphicsEnv.c#L763

[24] *Реинициализация Window* // Метод beforeCheckpoint // Сайт GitHub // Просмотрено 23.05.2022
https://github.com/kznts9v-1lya/crac/blob/x11-dev/src/java.desktop/share/classes/java/awt/Window.java#L180

[25] *Реинициализация XRootWindow* // Метод деинициализации // Сайт GitHub // Просмотрено 23.05.2022
https://github.com/kznts9v-1lya/crac/blob/x11-dev/src/java.desktop/unix/classes/sun/awt/X11/XRootWindow.java#L47

[26] *Реинициализация XRootWindow* // Метод инициализации // Сайт GitHub // Просмотрено 23.05.2022
https://github.com/kznts9v-1lya/crac/blob/x11-dev/src/java.desktop/unix/classes/sun/awt/X11/XRootWindow.java#L37

[27] *Реинициализация XWindow* // Метод beforeCheckpoint // Сайт GitHub // Просмотрено 23.05.2022
https://github.com/kznts9v-1lya/crac/blob/x11-dev/src/java.desktop/unix/classes/sun/awt/X11/XWindow.java#L72

[28] *Реинициализация XBaseWindow* // Метод beforeCheckpoint // Сайт GitHub // Просмотрено 23.05.2022
https://github.com/kznts9v-1lya/crac/blob/x11-dev/src/java.desktop/unix/classes/sun/awt/X11/XBaseWindow.java#L40

[29] *Реинициализация Cursor* // Метод beforeCheckpoint // Сайт GitHub // Просмотрено 23.05.2022
https://github.com/kznts9v-1lya/crac/blob/x11-dev/src/java.desktop/share/classes/java/awt/Cursor.java#L57

[30] *Реинициализация XGlobalCursorManager* // Метод beforeCheckpoint // Сайт GitHub // Просмотрено 23.05.2022
https://github.com/kznts9v-1lya/crac/blob/x11-dev/src/java.desktop/unix/classes/sun/awt/X11/XGlobalCursorManager.java#L38

[31] *Реинициализация XWM* // Метод beforeCheckpoint // Сайт GitHub // Просмотрено 23.05.2022
https://github.com/kznts9v-1lya/crac/blob/x11-dev/src/java.desktop/unix/classes/sun/awt/X11/XWM.java#L57

[32] *Реинициализация XWM* // Метод инициализации // Сайт GitHub // Просмотрено 23.05.2022
https://github.com/kznts9v-1lya/crac/blob/x11-dev/src/java.desktop/unix/classes/sun/awt/X11/XWM.java#L1347

[33] *Реинициализация XAtom* // Метод beforeCheckpoint // Сайт GitHub // Просмотрено 23.05.2022
https://github.com/kznts9v-1lya/crac/blob/x11-dev/src/java.desktop/unix/classes/sun/awt/X11/XAtom.java#L144

[34] *Реинициализация XWM* // Метод afterRestore // Сайт GitHub // Просмотрено 23.05.2022
https://github.com/kznts9v-1lya/crac/blob/x11-dev/src/java.desktop/unix/classes/sun/awt/X11/XWM.java#L73

[35] *Реинициализация XToolkit* // Метод beforeCheckpoint // Сайт GitHub // Просмотрено 23.05.2022
https://github.com/kznts9v-1lya/crac/blob/x11-dev/src/java.desktop/unix/classes/sun/awt/X11/XToolkit.java#L236

[36] *Реинициализация XToolkit* // Метод afterRestore // Сайт GitHub // Просмотрено 23.05.2022
https://github.com/kznts9v-1lya/crac/blob/x11-dev/src/java.desktop/unix/classes/sun/awt/X11/XToolkit.java#L290

[37] *Реинициализация XToolkit* // Resource для операции CheckpointRestore // Сайт GitHub // Просмотрено 23.05.2022
https://github.com/kznts9v-1lya/crac/blob/x11-dev/src/java.desktop/unix/classes/sun/awt/X11/XToolkit.java#L230

[38] *Реинициализация Disposer* // Ожидание получения ссылок от GC // Сайт GitHub // Просмотрено 23.05.2022
https://github.com/kznts9v-1lya/crac/blob/x11-dev/src/java.base/share/classes/java/lang/ref/Reference.java#L343

[39] *Реинициализация Disposer* // Ожидание получения ссылок в ReferenceQueue // Сайт GitHub // Просмотрено 23.05.2022
https://github.com/kznts9v-1lya/crac/blob/x11-dev/src/java.base/share/classes/java/lang/ref/ReferenceQueue.java#L217

[40] *Реинициализация Cleaner* // Resource для операции CheckpointRestore // Сайт GitHub // Просмотрено 23.05.2022
https://github.com/kznts9v-1lya/crac/blob/x11-dev/src/java.base/share/classes/jdk/internal/ref/CleanerImpl.java#L46

[41] *Реинициализация Disposer* // Resource для операции CheckpointRestore // Сайт GitHub // Просмотрено 23.05.2022
https://github.com/kznts9v-1lya/crac/blob/x11-dev/src/java.desktop/share/classes/sun/java2d/Disposer.java#L58