Министерство образования и науки Российской Федерации
Московский физико-технический институт (государственный университет)

Физтех-школа прикладной математики и информатики
Кафедра информатики и вычислительной математики

Выпускная квалификационная работа магистра по направлению 03.04.01 «Прикладные математика и физика»

Проектирование и реализация системы автоматической проверки задач курсов по программированию

Работу выполнил:
студент группы M05-908A,
Великанов Олег Владиславович

Научный руководитель:
кандидат физико-математических наук
Хохлов Николай Игоревич

Москва, 2022

Аннотация

В данной работе решается задача построения системы автоматизированного тестирования решений студентов к задачам из курсов по программированию. Такая система необходима для эффективной проверки практических заданий и позволяет увеличивать объем практической работы студентов. В ходе работы спроектирована и реализована проверяющая система, интегрированная с системой дистанционного обучения LMS и с уже существующей системой проверки Ejudge. Построенная система имеет пользовательский интерфейс в LMS и расширяет функционал Ejudge, позволяя гибко добавлять новые механизмы проверки без изменения существующих компонентов системы. В рамках данной работы реализован механизм проверки MPI программ, поддержки которых не было в Ejudge.

Содержание

1 Введение

В наши дни большинство университетских курсов по программированию содержат большой блок практической (лабораторной) работы, которая подразумевает, что студенты решают задачи по программированию: пишут программы, реализуют алгоритмы. Эти программы как и решения практических задач по любому курсу необходимо проверять преподавателям. При этом решение одной и той же задачи у каждого студента будет уникальным, каждый студент имеет свой стиль написания кода. Поэтому ручная проверка таких задач может отнимать много времени. Такая проверка является довольно рутинным процессом, в которых человеку свойственно ошибаться. Хотелось бы иметь возможность как-то автоматизировать этот процесс.

В современных практиках промышленной разработки давно закрепился подход юнит-тестирования. Программу делят на составные части (юниты). В качестве таких частей могут выступать разные средства языка, например функции или классы. Дальше к каждой части формируется свой набор тестовых сценариев. Если говорить о тестировании функций, то тестовый сценарий будет состоять из входных аргументов функции и значения, которое мы ожидаем получить на выходе функции, если применить ее к заданным аргументам. Чтобы протестировать функцию, надо запустить ее на всех наборах аргументов и проверить равенство ожидаемых и полученных при запуске значений.

Программы студентов в рамках учебных курсов решают достаточно узкие задачи: делают какое-то преобразование над входными данными, вычисляют какое-то значение на их основе или реализуют какой-то изучаемый алгоритм над ними. К проверке таких задач можно применить подход юнит-тестирования. Это позволит снять с преподавателей значительную часть работы. Позволит избавиться от ограничения на количество задач, которое может успеть проверить преподаватель. Можно будет увеличить число задач, предлагаемых студентам на практических занятиях. Студент будет сразу получать оценку своей работы. Поэтому такой подход к проверке давно получил популярность и широко используется в университетах.

При использовании этой методики сдача лабораторной работы выглядит следующим образом. Студент отправляет решение в систему автоматизированной проверки. Система запускает программу с разными входными данными, сверяет выходные данные с заранее известным эталоном, выносит вердикт. На решение задачи может быть дано несколько попыток, засчитывается последнее отправленное решение. По итогу фор-

мируется оценка, полученная исключительно автоматическим тестированием. После окончания лабораторной преподаватель может в ручном режиме посмотреть какие-то решения и скорректировать оценку.

В МФТИ активно используется система организации и проверки контестов - Ejudge. В ее терминах контест это набор задач, на решение которых отводится какое-то конечное время. Система имеет открытый исходный код, поддерживает большой набор языков программирования, имеет большой инструментарий для администрирования. На данный момент она развернута во внутренней сети МФТИ, и практические занятия большинства курсов организованы с ее помощью. С другой стороны в последнее время в институте активно развивается своя собственная система управления учебным процессом - LMS (learning management system), построенная на базе Moodle. Она обеспечивает связь преподавателя и студента, позволяет адресно предоставить студентам доступ к учебным материалам самого разнообразного характера, позволяет принимать от студентов и удобно проверять различные типы заданий. В связи с чем появилась идея интеграции систем LMS и Ejudge.

Хотелось бы дать возможность студентам видеть назначенные контесты, отправлять решение задач, отслеживать свою успеваемость в рамках курсов по программированию в том же интерфейсе, который они используют для курсов других кафедр. Преподавателям это даст возможность в одном в интерфейсе поддерживать контакт со студентами, назначать им задания, управлять задачами для практических занятий, видеть успеваемость студентов по конкретному контесту и по всему курсу в целом. Чтобы реализовать такую интеграцию необходимо построить промежуточную систему, которая будет выполнять роль посредника, способного общаться в терминах обеих систем.

Несмотря на то, что система организации контестов Ejudge имеет широкий спектр поддерживаемых языков, с ее помощью невозможно автоматизировать проверку задач по курсу параллельного программирования. В рамка этого курса студенты изучают подходы к распараллеливанию классических алгоритмов вычислительной математики. В качестве языка используется ***C*** вместе с библиотекой MPI. В Ejudge нет поддержки компиляции с использованием утилиты ***mpicc***, запуска программ с использованием ***mpirun***. Ejudge умеет запускать один процесс и сравнивать его stdout с эталоном. Поэтому невозможно независимо проверять выходные данные отдельных процессов. Давно есть запрос на автоматизацию проверки задач этого курса, и сделать это в рамках

Ejudge не представляется возможным.

Учитывая все эти предпосылки, появляется необходимость построить новую систему, которая предоставляла бы единый интерфейс для проверки самых разных задач, при этом внутри себя реализовывала бы разные механизмы проверки. Одним из таких механизмов была бы делегация проверки в Ejudge, а одним из альтернативных способов проверки стала бы проверка MPI программ, которую необходимо реализовать с нуля.

2 Постановка задачи

Целью данной работы является проектирование и построение системы автоматической проверки решений студентов к задачам из курсов по программированию. Система должна иметь свой собственный интерфейс внутри системы LMS. В нем студенты должны иметь возможность отслеживать назначенные задачи, отправлять решения к ним и просматривать результаты проверок.

Должны поддерживаться разные типы задач для разных курсов. Многие типы задач уже поддержаны, и их проверка автоматизирована за счет системы Ejudge. Поэтому новая проектируемая система должна быть интегрирована с системой Ejudge и должна уметь делегировать ей проверку некоторых задач. Полученный по решению вердикт должен транслироваться в интерфейс LMS, чтобы его могли увидеть преподаватели и студенты.

Одним из альтернативных типов задач могут стать задачи из курса параллельного программирования, для которых отсутствует поддержка в Ejudge. В рамках этого курса студенты учатся распараллеливать программы с использованием библиотеки MPI. Проектируемая система должна уметь самостоятельно компилировать, запускать и проверять такие программы с учетом специфики MPI. Должно быть поддержано введение дополнительных ограничений на решения студентов. Так, например, у администратора должна быть возможность выставить ограничение по времени исполнения, по используемой памяти, по количеству использований библиотечных MPI функций. С точки зрения интерфейса в LMS не должно быть видно никаких отличий между такими задачами и задачами, проверяемыми с помощью Ejudge.

Для достижения заданной цели необходимо выполнить следующие подзадачи:

1. Изучить документацию системы Moodle, на базе которой построена система LMS. Понять, какие существуют способы расширить ее функционал и спроектировать в ней необходимый интерфейс.

2. Изучить документацию системы Ejudge. Понять, каким образом с ней могут взаимодействовать другие программы, чтобы отправлять на проверку решения студентов и получать вердикт по ним.

3. Спроектировать архитектуру системы автоматической проверки. Понять, какие модули необходимо реализовать для интеграции уже существующих компонентов.

Спроектированная система должна иметь модульную архитектуру. Должен быть механизм добавления новых способов проверки решений студентов без изменения уже существующих.

4. Спроектировать проверяющий модуль, который будет взаимодействовать с Ejudge и делегировать ему проверку задач.

5. Спроектировать проверяющий модуль для работы с MPI программами.

6. Реализовать спроектированную систему и все запланированные модули. Развернуть ее во внутренней сети МФТИ. Провести тестовые контесты с ее использованием.

3 Обзор области и существующих инструментов

На сегодняшний день в мире существует большое число систем автоматизировонного тестирования, многие из которых имеют открытый исходный код, и которые можно развернуть на своих серверах.

PC^2 - система автоматизировонного тестирования, разработанная в Университете штата Калифорния, Сакраменто, США. Она активно используется для проверки на международных командных соревнованиях по программированию ICPC. Детальное описание системы, включая документацию для администраторов, судей и команд участников, а также все ссылки для скачивания доступны по адресу (https://pc2ccs.github.io/)[1]. Система разрабатывается с 1988 года, на сегодняшний день последняя стабильная версия 9.7.0 датирована 17 марта 2021 года. PC^2 состоит из клиентской и серверной частей, написанных на языке Java. Развертывание системы на собственном сервере требует ручных действий и манипуляций с конфигурационными файлами, которые описаны в документации. Для работы с системой существует графический интерфейс, а также интерфейс командной строки. Система «из коробки» поддерживает множество компиляторов и интерпретаторов. Список компиляторов может быть вручную расширен администратором системы. Для интерпретаторов такой возможности нет, однако в список включены PERL, PHP, Ruby, Python и shell.

Yandex.Contest (https://contest.yandex.ru/) [2] - сервис для онлайн проверки заданий, ориентированный в основном на олимпиады и соревнования по программированию. На базе этой системы проводится ежегодный чемпионат по программированию от Яндекса, а также различные олимпиады, в том числе финальные этапы всероссийской олимпиады по программированию. Yandex.Contest поддерживает более двадцати языков программирования и позволяет администраторам гибко настраивать схему соревнований. Также гибко можно настраивать задания: постановку задачи, набор тестов, критерии оценки. Утверждается, что сервис способен одновременно обрабатывать терабайты данных, благодаря чему способен выдержать одновременную нагрузку более чем от тысячи участников.

DOMjudge - система автоматизировонного тестирования, разработанная Исследовательской Ассоциацией A-Eskwadraat Утрехтского университета, Нидерланды. Система поставляется в виде серверного ПО и клиентского web-интерфейса. Основным преимуществом данной системы является возможность легко настраивать процесс параллель-

ной обработки несколькими серверами. Это дает возможность горизонтально масштабировать систему при растущей нагрузке. Проверка может осуществляться на серверах из внешней сети. То есть можно использовать облачные решения для развертывания системы. Установка и настройка системы происходит с помощью скриптов. Имеется возможность конфигурировать систему через ручное редактирование файлов. Однако стоит отметить, что «из коробки» поддерживается не так много языков: C, C++, Java, Haskell.

4 Проектирование системы

4.1 Расширение функционала LMS

Проектируемая система подразумевает наличие пользовательского интерфейса в системе LMS. Система LMS была запущена в МФТИ весной 2020 года для обеспечения процесса дистанционного обучения. Она построена на базе образовательной платформы с открытым исходным кодом Moodle. За счет модульной архитектуры Moodle позволяет расширять свой функционал с помощью реализации сторонних плагинов. Реализация плагина подразумевает разработку программного кода на скриптовом языке PHP. Плагины в Moodle делятся на определенные типы, каждый из которых направлен на изменение или расширение стандартного поведения системы в определенной области. Судя по документации [3] на данный момент в последней версии системы поддерживается несколько десятков типов плагинов. Среди них есть и те, которые позволяют расширить интерфейс сдачи задания на проверку.

С помощью такого плагина можно переопределить логику, по которой происходит проверка задания. Тестирование программы может занять длительное время, может накопиться очередь тестирования из-за ограниченности ресурсов проверяющей системы. Поэтому плагин должен взаимодействовать с проверяющей системой в асинхронном режиме. Самым простым решением будет разработать REST API, который будет иметь два метода. Первый - POST метод для отправки решения и информации о задаче, которую студент решает. Второй - GET метод для получения текущего статуса проверки и информации об оценке. Когда студент отправляет свое решение на проверку, плагин будет вызывать POST метод, передавая решение студента и информацию о решаемой задаче. Затем периодически асинхронно обновлять статус проверки с использованием GET метода.

4.2 Способы программного взаимодействия с Ejudge

Одним из способов проверки решений в проектируемой системе должна стать проверка посредством Ejudge. Для этого необходимо понять, какие существуют способы программного взаимодействия с ней. Архитектурно Ejudge состоит из серверного ПО, а также набора CGI скриптов, которые предлагается установить на заранее подготовлен-

ный web сервер. CGI скрипты обеспечивают наличие удобного интерактивного пользовательского веб-интерфейса. Им могут пользоваться преподаватели, студенты и администраторы. Однако для программного взаимодействия с Ejudge будет проще в обход сайта взаимодействовать с сервером напрямую.

Действия, доступные пользователям в веб-интерфейсе, можно осуществлять напрямую через интерфейс командной строки, что дает возможность для программного взаимодействия с системой. Доступ к серверу турниров из командной строки осуществляется через консольную утилиту. Если система устанавливалась в соответствии с официальной документацией [4], то абсолютный путь к бинарному файлу утилиты - `"/opt/ejudge/bin/ejudge-contests-cmd"`.

Основные варианты использования:

Напечатать версию системы Ejudge и время компиляции системы:

```
$ ejudge-contests-cmd -version
```

Напечатать краткую подсказку об использовании программы

```
$ ejudge-contests-cmd -help
```

Для отправки решения на проверку используется команда

```
$ ejudge-contests-cmd <contest_id> submit-run <session_file> <problem_name>
<lang> <submission_file_path>
```

Аргументы, которые необходимо передать:

- contest_id - id контеста, которому принадлежит проверяемая задача
- session_file - файл авторизации
- problem_name - имя задачи в рамках контеста
- lang - язык программирования, на котором написано проверяемое решение
- submission_file_path - путь к файлу с решением

При успешной отправке на проверку команда вернет submit_id - внутренний идентификатор отправки, с помощью которого можно отслеживать ее результат.

Для отслеживания статуса проверки:

```
$ ejudge-contests-cmd <contest_id> run-status <session_file> <submit_id>
```

Для получения полного отчета о проверке:

```
$ ejudge-contests-cmd <contest_id> dump-report <session_file> <submit_id>
```

4.3 Архитектура

При проектировании архитектуры любой системы надо опираться на основные требования, которые к ней предъявляются. Для интеграции с LMS система должна иметь REST API с двумя методами для отправки решения на проверку и получения актуального статуса проверки. Должно быть реализовано несколько методов проверки решений, а также должна быть возможность добавить новый метод проверки без изменения имеющихся. Первыми двумя механизмами проверки должны стать проверка посредством Ejudge и проверка MPI задач.

Проверка задач может занимать длительное время, поэтому взаимодействие пользователя с системой должно быть асинхронным. Во время проведения контестов или контрольных работ ожидается рост числа запросов на проверку. Система должна быть устойчива к этому и не переставать обрабатывать запросы пользователей при их растущем числе. Должна быть возможность горизонтально масштабировать систему при необходимости. Также пользователям на постоянной основе должна быть доступна история посылок и результатов проверок. Целостность и долговечность этой информации не должна зависеть от сбоев системы. Все вышесказанное может быть достигнуто с помощью следующей архитектурной схемы (рис. 1) .

Рассмотрим ее более подробно. Внутрь системы LMS интегрируется плагин, который расширяет интерфейс сдачи задания и добавляет возможность отправлять программу на проверку в проектируемую систему. Для отправки он использует POST метод `/submit`. В теле запроса передается информация о проверяемой задаче, а также решение студента. В ответ на отправку решения клиент получает уникальный идентификатор посылки. Далее плагин может получить актуальный статус проверки с использованием GET метода `/submission_status`, передавая в качестве аргумента полученный ранее идентификатор.

«Сердцем» системы является сервис `coordinator` (блок coordinator рис. 1). Именно он предоставляет публичный REST API интерфейс, который будет использоваться LMS плагином. Помимо этого `coordinator` отвечает за хранение истории проверок. Состояние проверок должно быть согласованным и долговечным. Для этого сервис должен использовать реляционную базу данных, обладающую свойствами ACID. На каждую новую посылку в таблице должна создаваться новая запись, соответствующая этой проверке. При этом должен использоваться тот же идентификатор, который ранее получил

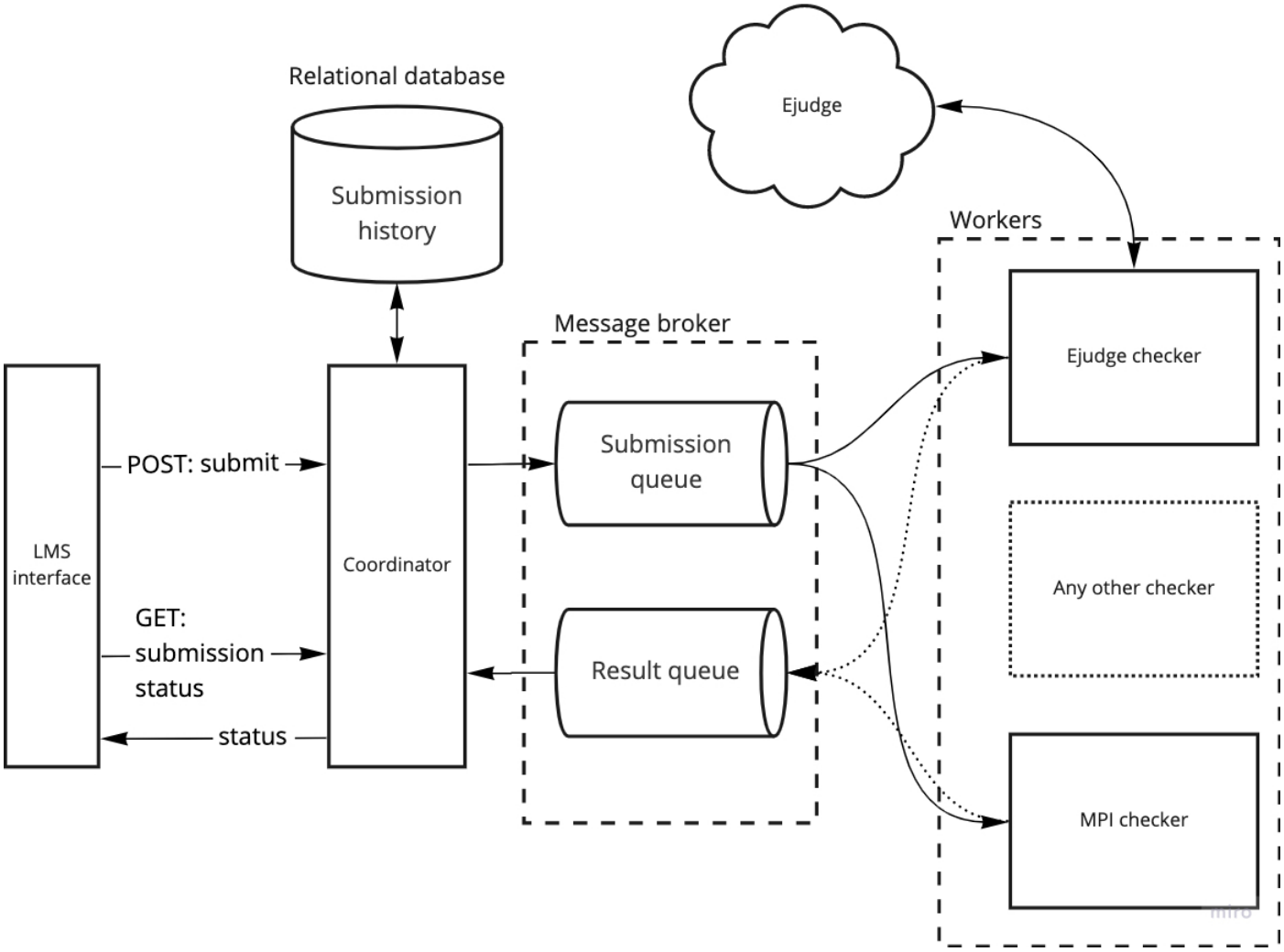

Рис. 1: Архитектура системы

клиент, отправивший решение на проверку. В дальнейшем при получении результата проверки `coordinator` должен обновлять статус в таблице.

Также `coordinator` хранит информацию о задачах и тестовых сценариях, которые необходимы для вынесения вердикта. Это может быть набор тестовых сценариев, ожидаемые выходные данные программы, флаги компиляции, ограничения по времени и используемой памяти.

Полученное на проверку решение `coordinator` должен отправить в проверяющий модуль, соответствующий задаче (блок workers рис. 1). Он мог бы отправлять решение напрямую в сервис проверки через REST API, однако при таком подходе добавление нового способа проверки потребует изменений в коде сервиса `coordinator`. Нужно бу-

дет добавлять ветку кода, связанную с отправкой решения в новый сервис проверки. Также стоит отметить, что между `coordinator` и сервисами проверки должно быть асинхронное взаимодействие, так как проверка задач может занимать существенное время. Если делать это взаимодействие синхронным, то придется долгое время держать отдельное открытое tcp соединение на каждую задачу. Такой подход совершенно не масштабируется.

Альтернативой мог бы стать подход с двумя методами: один для отправки на проверку, другой для получения актуального статуса. Однако в таком случае сервисы проверки должны были бы хранить состояние с информацией о посылках. Поэтому наиболее правильным решением в данной ситуации будет использование брокера сообщений и событийно-ориентированный подход. Получив решение на проверку, `coordinator` обогащает его информацией о задаче, формирует архив и отправляет его в очередь посылок (блок submission queue рис. 1). С другого конца очереди на обновления подписываются сервисы проверки. На основе заголовков сообщения, которые проставляет `coordinator`, можно установить, какому сервису проверки сообщение предназначается. Говоря иными словами, на основе заголовков происходит перенаправление (роутинг) сообщения соответствующему подписчику.

Получив решение на проверку, сервис проверки в соответствии со своей логикой выносит вердикт. Это может занять длительное время, однако как только вердикт вынесен, сообщение с вердиктом отправляется в другую очередь в рамках того же брокера сообщений (блок result queue рис. 1). У этой очереди один подписчик - `coordinator`. Он вычитывает вердикты из очереди и обновляет состояние посылки в базе данных.

Стоит отметить, что такая архитектура позволяет легко масштабировать количество сервисов проверки даже одного типа. Вычитывание посылок из очереди может конкурентно осуществлять неограниченное количество сервисов. Также такая архитектура устойчива к падениям проверяющих сервисов за счет гарантий, которые предоставляют брокеры сообщений. Посылка будет храниться в очереди до тех пор, пока вердикт по ней не будет отправлен назад в `coordinator`.

5 Реализация системы

5.1 Используемые технологии

Архитектура, описанная в главе [4], подразумевает реализацию сервиса координатора и отдельных проверяющих модулей. Это должны быть программы-сервисы, которые можно один раз запустить и ожидать, что они будут постоянно работать, обслуживая входящие сообщения. В качестве языка программирования для этих сервисов был выбран Python 3. Такой выбор обусловлен широким распространением языка в академической среде. Код, написанный на Python, легко читать и понимать, легко вносить правки. Кроме того у языка большое сообщество пользователей, существует огромное количество библиотек для работы с различными внешними технологиями. Последний пункт про библиотеки крайне важен в рамках данной работы, так как архитектурная схема подразумевает использование внешней реляционной базы данных и брокера сообщений.

На сегодняшний день существует большое количество различных реляционных систем управления базами данных (РСУБД). Согласно [5] их число на данный момент приближается к ста. Тем не менее между ними существуют отличия, опираясь на которые и нужно делать выбор. Важно отметить, что данный проект носит исключительно некоммерческий характер, поэтому в нем должны быть использованы только свободно распространяемые технологии с открытым исходным кодом. Также при выборе стоит учитывать наличие и качество библиотек для работы с базой для того языка программирования, который используется в проекте. Наиболее популярной и распространенной РСУБД с открытым исходным кодом на данный момент является PostgreSQL. Именно она используется в качестве базы данных в данном проекте.

Выбор брокеров сообщений менее обширный. Наибольшей популярностью на сегодняшний день пользуется Apache Kafka, однако для целей нашего проекта ее функционал и возможности будут излишни. За это придется заплатить легкостью администрирования и скоростью настройки. Поэтому выбор пал на RabbitMQ. Это брокер сообщений с открытым исходными кодом, реализующий функционал очередей, и дающий гарантии относительно целостности и долговечности данных. «Из коробки» предоставляется панель администратора, в которой вручную можно управлять очередями, а также отслеживать ключевые метрики системы. Также большим плюсом является

наличие асинхронного клиента на языке Python, который позволяет обрабатывать входящие сообщения независимо с использованием корутин, которые появились в языке начиная с версии 3.4.

5.2 Реализованные сервисы

В рамках данной работы были реализованы два сервиса проверки. Первый - `ejudge-checker` - должен осуществлять проверку с помощью Ejudge. Второй - `mpi-checker` - должен самостоятельно проверять MPI задачи. Также было реализовано «сердце» системы - сервис `coordinator`.

Сервис `coordinator` решает три основные задачи. Во-первых, он реализует REST API для отправки решения на проверку и получения актуального статуса проверки. Для этого был использован фреймворк Django [6], позволяющий быстро поднять HTTP сервер и удобно настроить маршрутизацию входящих запросов. Во-вторых, он отвечает за хранение состояния проверок, и гарантирует целостность и долговечность этого состояния. Для этого он использует реляционную базу данных PostgreSQL. Каждой проверке присваивается уникальный идентификатор и создается соответствующая запись в таблице. В-третьих, сервис координирует работу проверяющих модулей. На основе полученной от LMS информации он формирует архив со всеми необходимым для проверки файлами. Затем отправляет полученный архив в очередь, соответствующую нужному проверяющему модулю. Одновременно с этим он постоянно слушает обновления из очереди результатов проверок.

Сервисы `ejudge-checker` и `mpi-checker` являются проверяющими модулями. Они похожи с точки зрения интерфейса. Оба сервисы подписываются на очередь сообщений и вычитывают сообщения по соответствующему ключу. Далее они осуществляют проверки и формирует сообщение с результатом проверки, которое отсылают в другую очередь. Однако два сервиса совершенно по-разному реализуют описанный интерфейс.

`Ejudge-checker` устроен достаточно просто. Он принимает архив, достает из него файл с решением, получает информацию о том, к какой задаче в терминах Ejudge это решение относится и с использованием интерфейса командной строки отправляет решение на проверку в Ejudge. Затем он периодически опрашивает Ejudge об актуальном статусе проверки, а дождавшись результата, формирует сообщение с результатом для координатора.

`MPI-checker` устроен чуть более сложно. Модуль проверки умеет компилировать, запускать и сравнивать выходные данные MPI программ с ожидаемыми. При его реализации было поддержано введение дополнительных ограничений на решения. У администратора системы есть возможность выставить ограничение на решение по времени исполнения и по используемой памяти. Также есть возможность ограничить набор используемых MPI функций и число их использований. Сравнение реальных выходных данных и ожидаемых производится отдельно для каждого процесса.

Для реализации требования на ограничение по времени и памяти был использован менеджер нагрузки Slurm [7]. Он позволяет управлять распределенными ресурсами, аллоцировать их на выполнение задач, и запускать задачи с использованием выделенных ресурсов. Slurm поддерживает очередь выполнения задач. Как только в системе есть достаточно свободных ресурсов для выполнения следующей задачи из очереди, он приступает к ней. Взаимодействие со Slurm чем-то похоже на взаимодействие с Ejudge. Сначала задача посылается на выполнение, а затем необходимо отслеживать ее статус и ждать, пока Slurm ее выполнит. «Из коробки» Slurm позволяет настроить для каждой задачи отдельные лимиты по времени исполнения и используемой памяти.

Все реализованные сервисы доступны в публичном Bitbucket репозитории.[1]

5.3 MPI профайлер

Для реализации ограничения на MPI функции необходимо отслеживать, какие функции используются в решении студента. Наивный подход мог бы заключаться в анализе исходного кода программ для выявления явного использования запрещенных функций. Однако такой подход не позволяет задать ограничение на количество использований определенной функции. Например, мы хотим дать студенту возможность использовать конкретную функцию не более двух раз. Наличие вызова функции в коде программы не противоречит данному ограничению, однако неясно, сколько раз функция будет вызвана, если вызов происходит, например, в цикле. Поэтому единственным способом реализовать данное ограничение будет профилирование программы.

Спецификация Open MPI [8] содержит информацию о профилировании. Любая реализация MPI должна предоставлять интерфейс, который позволял бы разрабатывать сторонние дебагеры, анализаторы производительности и другие инструменты, исполь-

[1] https://bitbucket.org/mipt-mpichecker/

зующие информацию о ходе исполнения программ. Такой интерфейс называется интерфейсом профилирования. Благодаря тому, что он является частью спецификации, гарантируется, что профайлер или дебагер, написанный и протестированный с конкретной реализацией, будет также работать с любой другой реализацией. Согласно спецификации такой интерфейс должен предоставляться следующим образом. Все функции, декларируемые в спецификации MPI, должны быть реализованы и иметь две альтернативных точки входа. Основная функция, декларируемая спецификацией, должна иметь префикс `MPI_`, однако у нее должна существовать пара с префиксом `PMPI_` которая должна быть идентична оригинальной функции. Это позволяет разработать динамическую библиотеку, в которой будут переопределяться функции с префиксом `MPI_`. При этом новые переопределенные функции внутри себя будут вызывать функции с префиксом `PMPI_`. Вокруг этого вызова могут совершаться любые операции, которые нужны разработчикам профайлеров. За счет того, что библиотека динамическая, для подключения профайлера не требуется перекомпиляция оригинальной программы. Достаточно заново произвести динамическую линковку. Такой подход очевидно приводит к появлению накладных расходов. И расходы тем выше, чем сложнее логика, реализованная на слое профилирования.

В открытом доступе был найден только один готовый MPI профайлер с открытым исходным кодом - mpiP [9]. При попытке использовать его для решения нашей задачи выяснилось, что формат вывода результатов профилирования плохо структурирован и было бы достаточно сложно разбирать его программным способом. К тому же количество функций и возможностей профайлера является избыточном для нашей задачи, а значит мы получим накладные расхода на то, что нам не требуется. Поэтому было принято решение реализовать свой собственной профайлер.

Для решения нашей задачи требуется только вести счетчик количества вызовов MPI функций. С точки зрения реализации это делается следующим образом. Для каждой функции из спецификации MPI надо написать обертку, которая будет регистрировать вызов соответствующей MPI функции, а затем вызывать оригинальную функцию с теми же аргументами. Рассмотрим на примере обертки для функции `MPI_Send`.

```
MPI_CALLS = malloc(sizeof(int) * MPI_FUNCTIONS_COUNT);
for (int i = 0; i < MPI_FUNCTIONS_COUNT; i++) {
    MPI_CALLS[i] = 0;
}
...
void REGISTER_CALL(int funcId) {
  MPI_CALLS[funcId]++;
}
...
extern int MPI_Send(const void *buf, int count, MPI_Datatype
   datatype, int dest, int tag, MPI_Comm comm) {
  REGISTER_CALL(108);
  int res = PMPI_Send(buf, count, datatype, dest, tag, comm);
  return res;
}
```

В коде, представленном выше, число 108 это порядковый номер функции `MPI_Send` в спецификации. Стоит отметить, что на данный момент в спецификации MPI задекларировано 146 функций, и написание вручную обертки для каждой из них заняло много времени и привело бы к большому количеству багов. Поэтому этот процесс необходимо было автоматизировать и написать алгоритм кодогенерации. В итоге был разработан Python скрипт, который на основе спецификации генерирует код библиотеки профайлера на языке C.

Последнее требование к проверяющему сервису - возможность независимо оценивать выходные потоки разных MPI процессов. Утилита `mpirun`, используемая для запуска MPI программ, создает один общий поток вывода и пишет в него все сообщения, получаемые от разных процессов. Нет возможности каким-либо образом без изменений кода программы разделить потоки вывода. Важно отметить, что в ходе реализации предыдущего требования уже был разработан собственный профайлер, который будет использоваться для всех запусков MPI программ. Таким образом мы можем интегрировать дополнительную логику во все проверяемые решения, так как в коде профайлера у нас есть возможность добавлять эту логику в слое интерфейса профилирования. В том

числе в разработанном профайлере уже есть переопределение функции `MPI_INIT`, которая вызывается в любой MPI программе гарантированно раньше вызова всех остальных функций. Это то место, где есть возможность переопределить поток вывода и сделать это независимо для каждого процесса. Таким образом можно для каждого процесса настроить перенаправление потока вывода в отдельный файл.

```
static int world_rank = 0;
static int world_size = 0;
...
void REDIRECT_STDOUT() {
    char* fn = (char*)malloc(15);
    sprintf(fn, "stdout_%03d.log", world_rank);
    freopen(fn, "w", stdout);
    free(fn);
}
...
extern int MPI_Init(int *argc, char ***argv) {
        int res = PMPI_Init(argc, argv);
        PMPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    PMPI_Comm_size(MPI_COMM_WORLD, &world_size);
        REDIRECT_STDOUT();
        REGISTER_CALL(82);
        return res;
}
```

После запуска MPI программы с использованием 3 процессов (`mpirun-n3a.out`) получим в текущей директории 3 файла с именами `stdout_001.log`, `stdout_002.log`, `stdout_003.log`.

Реализация MPI профайлера доступна в публичном Github репозитории.[2]

[2]https://github.com/olegvelikanov/mpi-call-stats

5.4 Пример работы системы

Рассмотрим все особенности работы системы, проследив за конкретным примером проверки решения. Будем рассматривать базовую задачу типа Hello world, в которой необходимо, чтобы каждый процесс вывел в консоль свой порядковый номер и общее количество процессов.

Взаимодействие студента с системой происходит в интерфейсе LMS. Студент записан на курсы, каждый из которых поделен на занятия. В рамках занятия может быть представлено несколько тестовых заданий. В рамках этой работы был реализован новый тип тестового задания, который подразумевает отправку решения на проверку в новую систему. Интерфейс сдачи задания выглядит как форма ввода, в которую студент должен поместить свое решение и нажать кнопку "Закончить попытку"(рис. 2).

После этого плагин для Moodle, реализующий новый тип тестового задания, делает сетевой HTTP вызов в сервис `coordinator`. Продемонстрируем данный запрос в виде curl команды. Для краткости опустим тело запроса, которое идет после флага `--data-raw`.

```
   $ curl -X POST 'http://remote.vdi.mipt.ru:55608/api/submit'
-H 'Accept: application/json, text/plain, */*'
-H 'Content-Type: multipart/form-data; boundary=--WebKitFormBoundaryZ3hOqIr7NY8bn6tu'
--data-raw ...
```

Видно, что происходит POST запрос на эндпоинт `/api/submit`. Тип содержимого тела запроса - `multipart/form-data`. В теле запроса передается идентификатор задачи, а также решение студента. В ответ на такой запрос клиент получает json вида `{"id":"81f98d5d-9adf-40ea-b005-1f5967f3d590"}`, который содержит идентификатор посылки.

При обработке такого запроса `coordinator` создает в реляционной базе данных новую запись с уникальным идентификатором, соответствующую новой посылке. Далее сервис определяет, что для проверки задачи надо провести серию тестов с разными входными данными и разным числом MPI процессов. Для каждого теста формируется отдельный zip-архив, содержащий файлы, необходимые для проверки. В случае с рассматриваемой задачей архив содержит в себе две директории: `submission` и `test`. В директории `submission` содержится один файл `submission.c`, в котором находится решение студента. В директории `test` хранятся файлы вида `stdout.001`, которые являются

корректным и ожидаемым выводом i-го процесса. С содержимым этого файла должен сравнивать реальный поток вывода сервис проверки. Когда все zip-архивы сформированы, каждый из них отправляется отдельным сообщением в очередь `submit-queue` брокера сообщений. После того, как все сообщения успешно добавлены в очередь, идентификатор посылки отправляется обратно клиенту, который сделал вызов эндпоинта `/api/submit`.

Важно отметить, что сервис `coordinator` также добавляет к сообщениям заголовки. Так, например, в одном из заголовков содержится информация о типе проверяемой задачи. На основе этого заголовка внутри брокера сообщений происходит роутинг, и сообщение из общей очереди попадает в топик, на который подписан соответствующий типу задачи проверяющий модуль.

В случае с задачей MPI Hello world сообщения попадут в топик, который слушает сервис `mpi-checker`. Сервис обрабатывает сообщения асинхронно. На каждое сообщение создается новая корутина, которая должна проверить решение студента в заданной тестовой конфигурации. Процесс проверки можно разделить на несколько этапов. В рамках первого этапа происходит подготовка рабочей директории. Создается временная директория с уникальным названием, содержимое полученного zip-архив разархивируется в эту директорию, затем валидируется ее структура и проверяется наличие всех необходимых для проверки файлов. На следующем этапе происходит компиляция программы `submission.c` с нужными флагами и с использованием необходимых библиотек. В частности именно на этом этапе происходит линковка MPI профайлера, о котором шла речь в предыдущей главе. Также формируется файл запуска slurm задачи - `sbatch.sh`. Затем следует этап запуска и ожидания окончания выполнения. В рамках него в менеджере slurm регистрируется новая задача, а затем в бесконечном цикле ожидается ее завершение. Любая задача будет либо успешно завершена, либо остановлена при превышении временных или ресурсных ограничений. Последний этап - проверка потока вывода. Читаются файлы вывода процессов и каждый из них сравнивается с ожидаемым. Далее анализируются результаты профилировки. Для каждого процесса проверяется, что не было превышено количество использований MPI функций. В итоге выносится вердикт. Сообщение с вердиктом отправляется в очередь `result-queue` брокера сообщений.

Стоит отметить, что описанный процесс происходит независимо для каждого теста

в рамках одной задачи. В итоге в очереди результатов будут вердикты по каждому из тестов в отдельности. Сервис `coordinator` дождется вердиктов по всем тестам, после чего сможет сделать финальный вердикт по решению студента в целом. Если хотя бы один из тестов не прошел, студент не получает баллов за задачу.

Когда финальный вердикт вынесен, он записывается в базу данных и может быть прочитан оттуда в любой момент времени. Плагин Moodle использует для получения актуального статуса проверки GET запрос, который также можно проиллюстрировать с помощью curl команды.

```
$ curl 'http://localhost:8080/api/status/81f98d5d-9adf-40ea-b005-1f5967f3d590'
```

Ответом на такой запрос является json вида


```
{
    "id": "1f98d5d-9adf-40ea-b005-1f5967f3d590",
    "status": "OK",
    "grade": 1.0,
    "children":
    [
        {
            "id": "4e4b20ff-16c2-471e-9eb0-6869be50fc86",
            "name": "1 process",
            "status": "OK",
            "grade": 1.0,
            "details": {"io": "ok", "time": "331"},
            "options": {"compile_flags": "-lm", "np": 1}
        },
        {
            "id": "5e6a0e5b-fc67-4386-bf3e-942b38bb02b4",
            "name": "2 processes",
            "type": "test",
            "status": "OK",
            "grade": 1.0,
            "details": {"io": "ok", "time": "341"},
            "options": {"compile_flags": "-lm","np": 2}
        },
```

```
        {
            "id": "91634f64-3321-4491-a2e8-74234b717a79",
            "name": "5 processes",
            "type": "test",
            "status": "OK",
            "grade": 1.0,
            "details": {"io": "ok", "time": "425"},
            "options": {"compile_flags": "-lm","np": 5}
        },
    ]
}
```


В нем видно, что было проведено 3 теста с разным числом MPI процессов: 1, 2, 5. Каждый тест прошел успешно, и итоговый вердикт - максимальный балл. Студент видит этот результат в интерфейсе LMS (рис. 3).

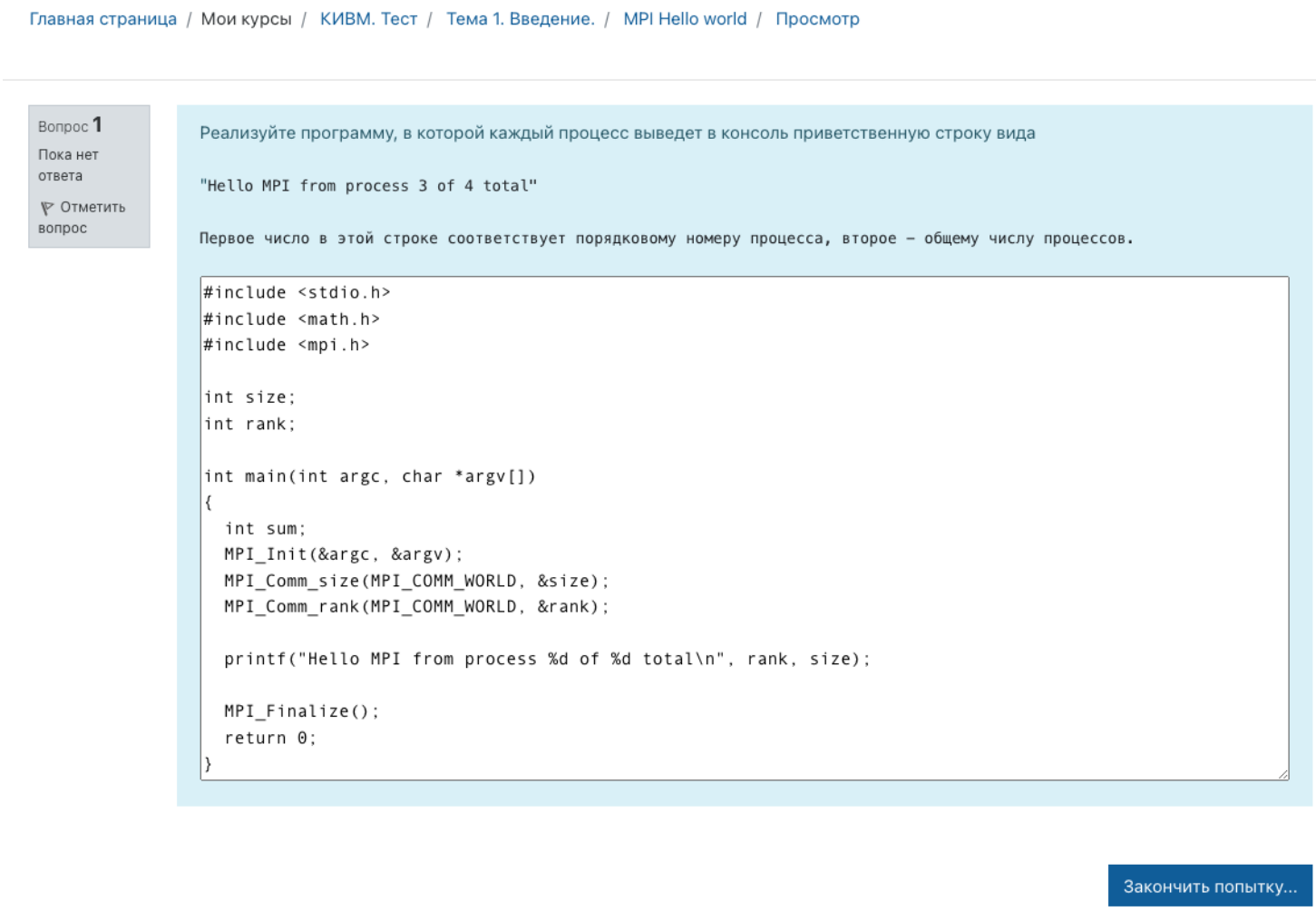

Рис. 2: LMS: форма отправки решения на проверку

Тест начат	Суббота, 28 мая 2022, 12:34
Состояние	Завершено
Завершен	Суббота, 28 мая 2022, 15:53
Прошло времени	3 час. 19 мин.
Баллы	1,00/1,00
Оценка	**10,00** из 10,00 (**100**%)

Вопрос **1**

Выполнен

Баллов: 1,00 из 1,00

Отметить вопрос

Реализуйте программу, в которой каждый процесс выведет в консоль приветственную строку вида

```
"Hello MPI from process 3 of 4 total"
```

Первое число в этой строке соответствует порядковому номеру процесса, второе — общему числу процессов.

```
#include <stdio.h>
#include <math.h>
#include <mpi.h>

int size;
int rank;

int main(int argc, char *argv[])
{
  int sum;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  printf("Hello MPI from process %d of %d total\n", rank, size);
```

Результат: OK

Набор тестов: Correctness

Тест	Результат	Балл	Ответ	Время	Опции
1 process	OK	1	ok	336	{"compile_flags":"-lm","np":1}
2 processes	OK	1	ok	335	{"compile_flags":"-lm","np":2}
5 processes	OK	1	ok	473	{"compile_flags":"-lm","np":5}

Рис. 3: LMS: результат проверки

6 Заключение

Целью работы было построение системы автоматической проверки студенческих программ. Система должна была иметь интерфейс внутри системы дистанционного обучения LMS, используемой в МФТИ. В ходе работы изучена документация проекта Moodle, на базе которого построена LMS. Стали известны методы для расширения ее функционала и создания нового интерфейса сдачи решения задач на проверку. Оказалось, что Moodle позволяет реализовать собственные плагины, которые будут добавлять новый функционал.

Затем была изучена документация Ejudge. Основной задачей было понять, какие существуют способы программного взаимодействия с системой проверки. Выяснилось, что система имеет интерфейс командной строки, который в частности используется веб-сервисом для обработки пользовательских запросов, получаемых через веб-интерфейс. В итоге был построен модуль проверки посредством использования Ejudge, который взаимодействует с Ejudge через этот интерфейс.

Далее была спроектирована архитектура системы автоматической проверки (рис. 1). При разработке архитектурной схемы были учтены все требования, предъявляемые к системе: наличие REST API для отправки решения, поддержка нескольких способов проверки и возможность добавлять новые. Появился список сервисов, которые необходимо реализовать. Также стал понятен список внешних технологий (база данных, брокер сообщений), которые необходимо выбрать из множества доступных и настроить для использования. В итоге в рамках проекта были использованы база данных PostgreSQL и брокер сообщений RabbitMQ.

Последним шагом была реализация спроектированных сервисов. Для этого было выбран язык Python как наиболее простой для понимания и широко распространенной в академической среде. Были реализованы:

1. сервис `coordinator`, который предоставляет REST API для отправки решения на проверку и получения актуального статуса проверки.

2. проверяющий модуль `ejudge-checker`, который делегирует проверку задач системе Ejudge

3. проверяющий модуль `mpi-checker`, который самостоятельно осуществляет проверку MPI задач

Стоит отметить, что для реализации всех требований, предъявляемых к проверке MPI задач, дополнительно был разработан MPI профайлер.

Таким образом, все поставленные в разделе [2] задачи выполнены и цели данной работы достигнуты.

Список литературы

[1] PC2 system Github page. — https://pc2ccs.github.io/.

[2] Yandex.Contest website. — https://pc2ccs.github.io/.

[3] Moodle plugin development documentation. — https://docs.moodle.org/dev/Plugin_types.

[4] Ejudge system installation documentation. — https://ejudge.ru/wiki/index.php/.

[5] Wikipedia: List of relational database management systems. — https://en.wikipedia.org/wiki/List_of_relational_database_management_systems.

[6] Django website. — https://www.djangoproject.com/.

[7] Slurm Workload Manager Documentation. — https://slurm.schedmd.com/documentation.html.

[8] Open MPI specification v3.1. — https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[9] mpiP Github. — https://github.com/LLNL/mpiP.