УДК 004.42

МАТЕМАТИЧЕСКАЯ МОДЕЛЬ И ГИБРИДНАЯ АРХИТЕКТУРА КЛИЕНТ-СЕРВЕРНОГО ПРИЛОЖЕНИЯ С ПРОГРАММНЫМ ШЛЮЗОМ-АДАПТЕРОМ

Алпатов А.Н., Студенников М.Р.

ФГБОУ ВО «МИРЭА – Российский технологический университет», Москва, e-mail: aleksej01-91@mail.ru

Развитие сетей передачи данных и внедрение набора протокола Интернет (TCP/IP), а также общее повышение скорости передачи данных привело к развитию распределённых систем, часто реализуемых в рамках трёхзвенных клиент-серверных архитектур. Для организации взаимодействия между клиентским и серверным программным обеспечением было разработано огромное количество методов и подходов, которые были оформлены в виде соответствующих технологий. Такое разнообразие используемых технологий накладывает существенные сложности при организации интеграции программных элементов, написанных на скриптовых языках, таких как Python, которые поддерживают огромный стек технологий для интеграции. Для решения этой задачи в данной работе предлагается способ реализации интеграционного шлюза на основе связки промежуточного ПО Ngnix и Daphne с Gunicorn, для интеграции WSGI и ASGI серверов. Также в работе даётся математическая модель разработанной системы, учитывающая разнородность запросов, протекающих в системе. Предложено математическое описание разработанной системы, с учётом сформулированного разделения системы на области, определяемые используемыми программными слоями, что может быть полезно при управлении клиент-серверной системой или разработке механизмов балансировки нагрузки в распределённых системах. Для оценки свойств разработанной системы и для решения задачи управления системой предложено совместное использование введённых в данной работе параметров точности идентификации объектов и индекса отзывчивости, в рамках расчёта среднего гармонического взвешенного.

Ключевые слова: интерфейсы прикладного программирования, клиент-серверное приложение, взвешенное гармоническое среднее, асинхронное и синхронное взаимодействие, WSGI

MATHEMATICAL MODEL AND HYBRID ARCHITECTURE OF A CLIENT-SERVER APPLICATION WITH A SOFTWARE GATEWAY-ADAPTER

Alpatov A.N., Studennikov M.R.

Federal State Budget Educational Institution of Higher Education «MIREA – Russian Technological University», Moscow, e-mail: aleksej01-91@mail.ru

The development of data networks and the introduction of the Internet protocol suite (TCP/IP), as well as a general increase in data transfer rates, led to the development of distributed systems, often implemented in the framework of three-tier client-server architectures. To organize the interaction between client and server software, a huge number of methods and approaches have been developed, which have been formalized in the form of appropriate technologies. This variety of used technologies imposes significant challenges in organizing the integration of software elements written in scripting languages such as Python, which support a huge stack of technologies for integration. To solve this problem, this paper proposes a way to implement an integration gateway based on a bundle of Ngnix middleware and Daphne with Gunicorn, to integrate WSGI and ASGI servers. Also the mathematical model of the developed system, taking into account the heterogeneity of requests, flowing in the system is given in the work. The mathematical description of the developed system, taking into account the formulated division of the system into the areas determined by the used program layers, which can be useful for client-server system management or development of load balancing mechanisms in distributed systems. To evaluate the properties of the developed system and to solve the problem of managing the system, we propose the joint use of the parameters introduced in this paper, the accuracy of object identification and the responsiveness index, as part of the calculation of the harmonic weighted average.

Keywords: application programming interface, client-server application, weighted harmonic mean, asynchronous and synchronous communication, WSGI

В реалиях современного мира новые сайты в интернете появляются каждый день. Их разработке и разработке веб-приложений посвящена большая часть ИТ индустрии. Часто перед разработчиками ставятся задачи быстрой и бюджетной разработки веб-приложений. Для решениях таких задач хорошо подходит популярный веб-фреймворк Django [1], написанный на Python. К основным достоинствам Django относятся простота его использования, широкий инструментарий, включенный в него, открытость его исходного кода и наличие большого количества документации и статей о нем.

Одним из финальных этапов разработки веб-приложения является его развертывание на сервере. В случае Django-приложений есть несколько важных нюансов их развертки. Большинство веб-серверов не понимают язык Python, поэтому были разработаны два стандарта (интерфейса) WSGI и ASGI.

WSGI – основной стандарт Python, который поддерживает только синхронный код.

ASGI – относительно новый стандарт, который дает доступ к функциям параллельного выполнения кода: WebSocket, Server-Sent Events, HTTP/2 и др. ASGI является наследником WSGI и имеет с ним обратную совместимость.

Для развертывания WSGI-приложения часто используются связки, представленные на рис. 1.

Наибольшей популярностью из них пользуется первая схема. Можно много рассуждать о преимуществах и недостатках Nginx и Apache, поэтому сравним данные сборки по промежуточным веб-серверам.

Связка с использованием промежуточного слоя определённого mod_wsgi, который в настоящее время является одним из популярнейших модулей-адаптеров WSGI Python, продолжительное время активно использовалась разработчиками распределённых систем [2]. Основным преимуществом mod_wsgi является простота развёртывания и его использования на практике, но в то же время главным недостатком mod_wsgi является то, что он может не поддерживаться крупными хостинг-провайдерами, и его можно использовать только с конкретным сервером приложений, а именно Apache. Такое обстоятельство может быть критичным для некоторых проектов.

uWSGI – это программная платформа, которая включает в себя как непосредственно сервер приложения, так и другие механизмы, такие как прокси-серверы, плагины запросов и т.д. [3]. Всё это достигается благодаря модульной конструкции системы. uWSGI был написан специально для Python, поэтому его легко установить через pip. Другими достоинствами данного сервера является то, что он обладает относительно низким порогом вхождения и позволяет создать ещё один промежуточный слой программного обеспечения, что делает возможным снять часть функционала непосредственно с сервера (однако из-за этого необходимо проксировать запросы). Другим несомненным преимуществом данной платформы можно считать, относительную простоту его настройки.

Gunicorn [4] – это HTTP-сервер Python WSGI, который очень похож на uWSGI. Gunicorn – это зрелый, полнофункциональный сервер и менеджер процессов. Достоинства и недостатки у него такие же, как у uWSGI, однако Gunicorn лучше справляется с высокими нагрузками [5], легче настраивается и требует меньше ресурсов [6].

По итогу видим, что популярность связки Nginx-Gunicorn вполне оправдана.

Для развертывания ASGI-приложения официальная документация Django советует использовать следующие ASGI сервера: Daphne, Hypercorn, Uvicorn.

Daphne – это сервер протоколов HTTP, HTTP2 и WebSocket для ASGI и ASGI-HTTP, который был разработан специально для Django Channels, поэтому он считается эталонным [7].

Uvicorn – это быстрый и легкий ASGI-сервер, использующий uvloop и httptools [8]. Uvicorn поддерживает HTTP/1.1 и WebSockets. Uvicorn включает рабочий класс Gunicorn, позволяющий запускать приложения ASGI со всеми преимуществами производительности Uvicorn, а также предоставляет вам полнофункциональное управление процессами Gunicorn. Это позволяет увеличить или уменьшить количество рабочих процессов на лету, плавно перезапускать рабочие процессы или выполнять обновление сервера без простоев.

Hypercorn – это веб-сервер ASGI, основанный на библиотеках sans-io hyper, h11, h2 и wsproto и вдохновленный Gunicorn. Hypercorn поддерживает спецификации HTTP/1, HTTP/2, WebSockets (через HTTP/1 и HTTP/2), ASGI/2 и ASGI/3. Hypercorn может использовать типы рабочих asyncio, uvloop или trio [9].

Использование того или иного сервера для ASGI, помимо их характеристик, обуславливается и тем, какой фреймворк (Django Channels, Quart, FastAPI, BlackSheep) будет привносить асинхронность в проект. Так как в этой статье мы говорим о Django, то будем рассматривать преимущественно Django Channels.

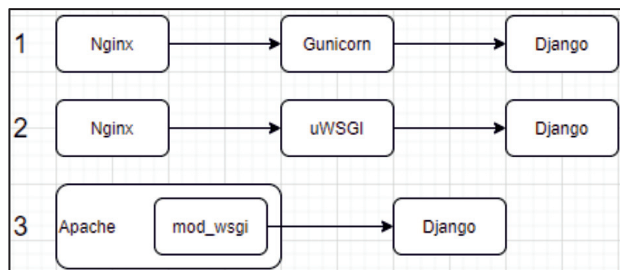

Рис. 1. Популярные схемы развертки WSGI-приложений

Библиотека Django Channels [10] появилась недавно, около 4–5 лет назад. Эта библиотека привнесла в Django асинхронное сетевое программирование. Channels расширяет инструментарий Django и дает возможность работать не только с HTTP протоколом, но и с WebSocket (ws), протоколом чата, IoT-протоколом и другими. Django Channels использует ASGI интерфейс.

При разработке реальных проектов часто бывают ситуации, когда нужно в уже развёрнутый проект включить новые функции. К примеру, в готовое Django-WSGI-приложение необходимо включить Dajngo Channels, которые привносят вместе с собой ASGI. Для подобной ситуации в данной статье предложена конфигурация развертки веб-приложения, которую можно считать эталонной для перехода от WSGI к ASGI в рамках Django-приложений, использующих Django Channels.

Материалы и методы исследования

Рассмотрим случай, когда имеется рабочее развернутое Django-WSGI-приложение, работающее на популярной связке, показанной на рис. 2.

Nginx тут используется как обратный прокси-сервер, а также отвечает за выдачу статического контента. Gunicorn является прослойкой между Nginx и Django. HTTP запросы Nginx передает Gunicorn, который передает их уже в Django, который взаимодействует со слоем базы данных. Возможная конфигурация Nginx и Gunicorn представлена на рис. 3.

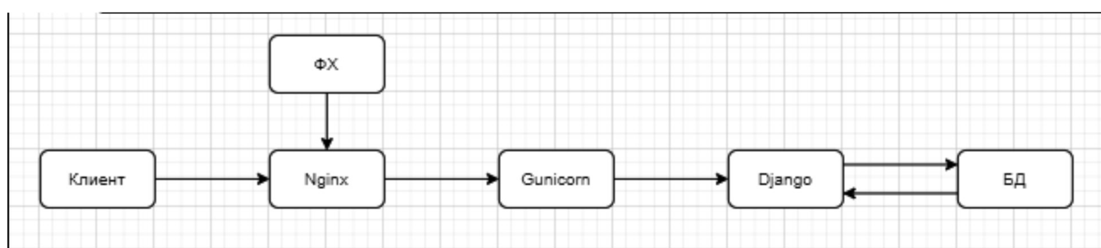

Рис. 2. Текущая архитектура сервера

```
gunicorn.service [Только для чтения]
/etc/systemd/system

1 [Unit]
2 Description=gunicorn daemon
3 Requires=gunicorn.socket
4 After=network.target
5
6 [Service]
7 User=root
8 WorkingDirectory=/home/mixail/DocTracer
9 ExecStart=/home/mixail/DocTracer/doctracer_env/bin/gunicorn \
10         --access-logfile - \
11         --workers 3 \
12         --bind unix:/run/gunicorn.sock \
13         DocTracer.wsgi:application
14 Restart=always
15
16 [Install]
17 WantedBy=multi-user.target
```

```
*DocTracer [Только для чтения]
/etc/nginx/sites-available
gunicorn.service | gunicorn.socket | *DocTracer

1
2 server {
3     listen 80;
4     server_name 0.0.0.0;
5
6     location = /favicon.ico { access_log off; log_not_found off; }
7
8     location /static/ {
9         root /home/mixail/DocTracer;
10    }
11
12    location /media/ {
13        root /home/mixail/DocTracer_MEDIA;
14    }
15
16    location / {
17        include proxy_params;
18        proxy_pass http://unix:/run/gunicorn.sock;
19    }
20 }
```

Рис. 3. Файл конфигурации Nginx и сервис для Gunicorn

Если добавить в приложение Django Channels с каким-либо функционалом и попытаться проверить работу системы в данной конфигурации можно увидеть, что не работают лишь функции, которые используют WebSocket. Это происходит, во-первых, потому, что в конфигурации Nginx не указано, куда отправлять ws/ запросы, во-вторых, если бы мы направили их на Gunicorn, то вероятно получили бы ошибку, так как Gunicorn – HTTP-сервер. Функции же, которые не используют асинхронность, то есть которые работали в WSGI приложении, работают так, как ASGI имеет обратную совместимость.

После установки Daphne настраиваем Nginx и Daphne, как показано на рис. 4, 5.

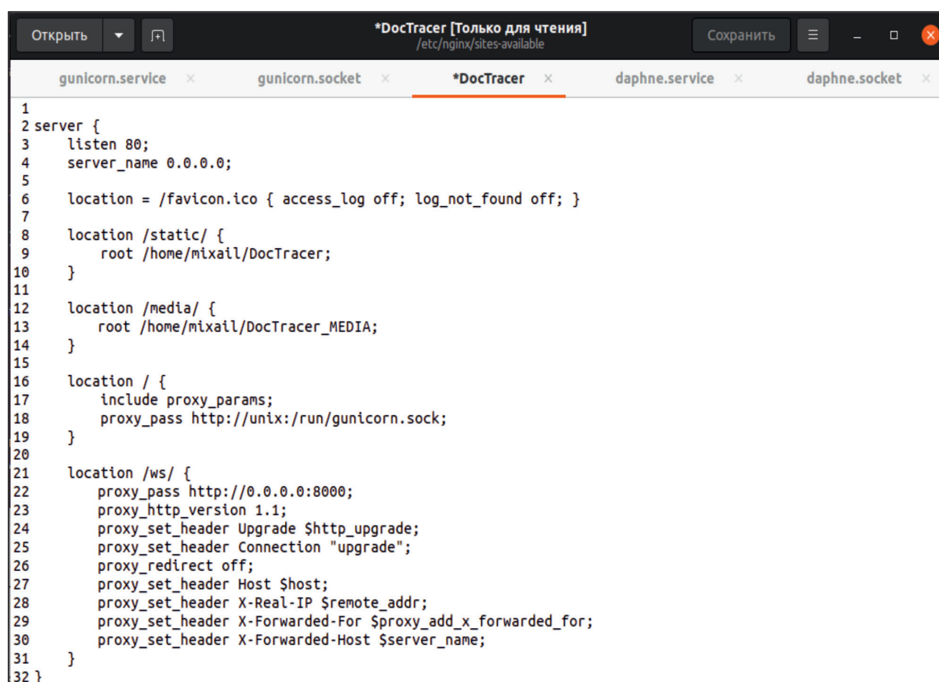

Рис. 4. Файл конфигурации Nginx

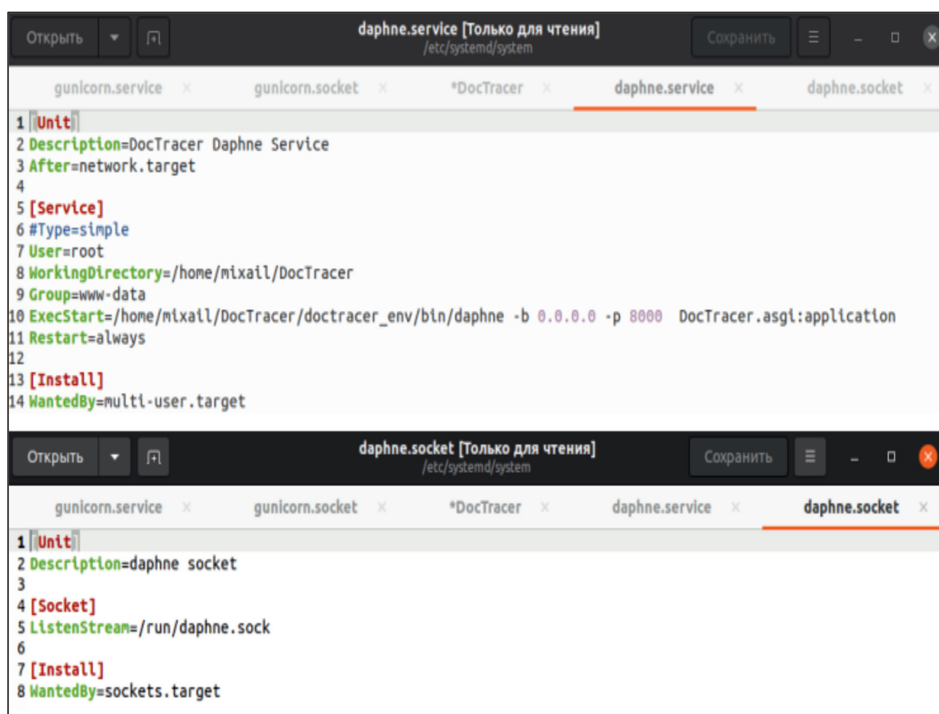

Рис. 5. Файл сокет и файл сервис для Daphne

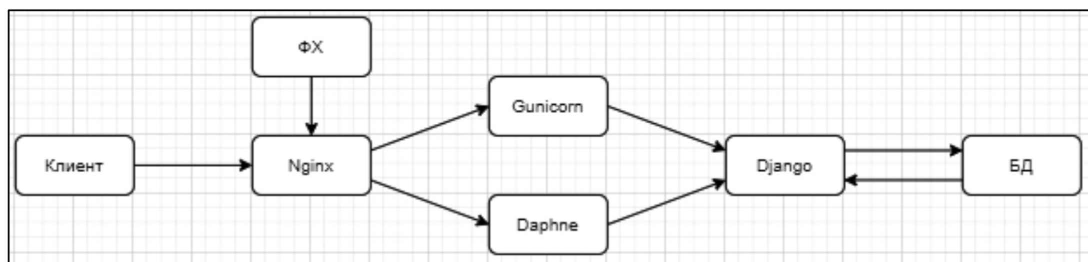

Рис. 6. Гибридная архитектура клиент-серверного приложения со шлюзом-адаптером

Таким образом, получаем видоизменённую гибридную архитектуру, представленную на рис. 6.

В представленной выше архитектуре Nginx не только является обратным прокси-сервером, который отвечает за статический контент, но еще и является балансировщиком нагрузки, отправляя HTTP запросы на Gunicorn, а WS запросы – на Daphne. При такой конфигурации происходит выгодное распределение нагрузки. Так как Daphne способен обрабатывать HTTP запросы, можно впоследствии улучшить архитектуру, сделав Daphne резервным сервером для HTTP.

В качестве второго сервера Daphne был выбран потому, что он разрабатывался специально под Django-Channels и максимально с ним совместим, обладая при этом функциями, которые могут пригодиться при дальнейшей разработке системы и ее масштабировании.

Если в аналогичной ситуации, когда нужно перевести свое приложение с WSGI на ASGI при уже развернутом Nginx+Gunicorn, и это необходимо сделать быстро и не потеряв в производительности, то вместо Daphne можно использовать Uvicorn. В описанной ситуации основным плюсом использования Uvicorn является то, что его настройка очень похожа на настройку Gunicorn. Хотя Uvicorn больше подходит для использования с FastAPI и имеет меньший функционал.

Модель обработки запросов в многоуровневых клиент-серверных системах

Рассмотрим процесс обработки разнородных запросов для разработанной системы. В рамках распределённой системы выделим следующие сущности:

1. Пользователь – это актор, который хочет получить данные либо доступ к функциональности удалённого объекта.

2. Клиент-потребитель – это актор, через который пользователь осуществляет авторизированный доступ к удалённому поставщику сервиса.

3. Поставщик сервиса – это актор, который запущен в рамках отдельного процесса, нежели клиент-потребитель, предоставляющий механизмы доступа к защищённым информационным ресурсам.

4. Защищённый ресурс – данные или бизнес-функционал на удалённом сервисе, которые представляют ценность для конечного пользователя системы.

Распределённая программная система, с учётом вышеобозначенных сущностей, будет определена как

$$Sys = \langle \alpha, Q, \theta, S, r \rangle, \quad (1)$$

где $\alpha = \left\{ \mathrm{A}_n, n = \overline{1, N} \right\}$ – множество пользователей распределённой программной системы;

$Q = \left\{ q_m, m = \overline{1, N} \right\}$ – множество гетерогенных акторов клиентов – потребителей распределённой программной системы;

θ – множество всех запросов в системе;

$S = \left\{ s_\eta, \eta = \overline{1, N} \right\}$ – множество поставщиков сервиса;

${}^{view}r = \left\{ R_i, i = \overline{1, N} \right\}$ – множество представлений ресурса r.

Вместе с тем соотношение $(\exists\alpha)(\exists Q)$ $(\omega = (\alpha, Q))$ определяет упорядоченные пары акторов авторизованных пользователей и аутентифицированных клиентов системы в определённый момент времени. Множество всех запросов для рассматриваемой выше системы может быть определено через представление Куратовского как

$$\left(\theta_i^{ws}, \theta_i^{\mathrm{http}}\right) := \left\{ \left\{ \theta_i^{ws} \right\}, \left\{ \theta_i^{ws}, \theta_i^{\mathrm{http}} \right\} \right\}, \quad (2)$$

где θ_i^{ws} – запросы типа WebSocket, сгенерированные парой $\left\{ \alpha_i, Q_i \right\}$;

θ_i^{http} – запросы типа http, сгенерированные парой $\left\{ \alpha_i, Q_i \right\}$.

Исходя из (2), общее количество запросов в системе в момент времени t определяется как

$$\sum_{i=1}^{N} Q_i^{ws} + \sum_{i=1}^{N} Q_i^{http} , \qquad (3)$$

где Q_i^{ws} – i-й запрос актора типа WebSocket;

Q_i^{http} – i-й запрос актора типа HTTP.

Введём понятие пороговый барьер $\Psi_к$, который будет определяться как программный слой, в рамках многослойной архитектуры, осуществляющий процесс маршрутизации поступающих запросов в систему. В современных системах пороговый барьер может быть определён через архитектурный паттерн Gateway [11]. Пороговый барьер определяет область $\sigma_{barrier}$. Область, определяемую программными слоями перед пороговым барьером, будем называть предпороговой областью σ_{front} барьера $\Psi_к$. Область, определяемую промежуточным программным обеспечением, находящимся за барьером $\Psi_к$, будем называть областью σ_{back} за пороговым барьером. Такое разделение оправдано тем, что в каждой из этих областей слой программного обеспечения, реализованный посредством использования современных фреймворков, может кешировать запросы, предобрабатывать, формировать цепочки обработки, обрабатывать как синхронно, так и асинхронно, а в некоторых случаях даже отклонять соответствующие типы запросов от их дальнейшей обработки [12]. Процесс обработки запросов всех типов в системе будет определяться как

$$f : Q \to Q_{\sigma_{front}} \to Q_{\sigma_{barrier}} \to Q_{\sigma_{back}} \to S , \qquad (4)$$

где S – конечное множество объектов ответа защищенного ресурса r из всего множества представлений Ri.

Отношение между множествами $Q, Q_{\sigma_{front}}, Q_{\sigma_{barrier}}, Q_{\sigma_{back}}, S$ определено через биективное соотношение, что означает, что между множествами определено однозначное соотношение и, следовательно, $\overline{\overline{Q}} = \overline{\overline{Q_{\sigma_{front}}}} = \overline{\overline{Q_{\sigma_{barrier}}}} = \overline{\overline{Q_{\sigma_{back}}}} = \overline{\overline{S}}$.

Далее введём критерий качества K обработки запросов в системе. В рассматриваемой модели разработанной системы запросы, инициатором которых является конечный авторизованный пользователь, обрабатываются удалённым сервисом, следовательно, в самом лучшем случае каждый запрос должен быть однозначно преобразован в объект ответа. Исходя из того, что в каждой из ранее определённых областей происходит воздействие на соответствующий запрос, что приводит к трансформации запроса, что в конечном итоге может привести к снижению точности объектов конечного ответа удалённого сервиса. Соответственно, под точностью в данной работе будем понимать долю объектов запроса, соответствующих определённому классу запроса, которые система однозначно относит к данному классу. В силу того, что в разработанной системе могут существовать два типа запросов θ_i^{ws} и θ_i^{http}, точность идентификации объектов будет определена формулой

$$\varepsilon = \frac{Q}{\sum_{i=1}^{N} ({}^{+}\theta_i^{ws} + {}^{+}\theta_i^{http})} , \qquad (5)$$

где ${}^{+}\theta_i^{ws}$ – однозначно определённые запросы типа θ^{ws} предпороговой, пороговой и запороговой барьерных областей,

${}^{+}\theta_i^{http}$ – однозначно определённые запросы типа θ^{http} предпороговой, пороговой и запороговой барьерных областей,

Q – общее число запросов в системе.

Следовательно, исходя из формулы (5), математическая интерпретация задачи идентификации запросов в системе может быть определена как соотношение

$$\begin{cases} f_0 = \sum_{i=1}^{N} g({}^{+}\hat{\theta}_j) \to \max, j = \overline{1, N}; \\ g({}^{+}\hat{\theta}) \ge 0, \end{cases} \qquad (6)$$

где ${}^{+}\hat{\theta}_j$ – параметр оптимизации, определённый как общее количество однозначно определённых запросов, j-й пары $\{\alpha_j, Q_j\}$.

Каждый запрос, в самом лучшем случае, формирует один ответ, определённый как представление ресурса ${}^{view}r$. При этом пользователь системы заинтересован в представлении всех требуемых ему данных, для поддержания своих бизнес-процессов за один запрос. Однако в силу архитектурных особенностей современных распределённых систем получить требуемую порцию данных за один запрос бывает очень затруднительно. Для решения этой задачи, в зависимости от назначения и архитектурных особенностей системы могут использоваться такие механизмы, как оркестрация и хореография в рамках микросервисной архитектуры, графовые модели программных интерфейсов, например GraphQL и т.д. Исходя из этого, определим новую характеристику как полнота ответа поставщи-

ка сервиса. Данную характеристику можно определить как

$$Z = \frac{1}{S_i}, Z \in (0,1], \quad (7)$$

где S_i – общее число запросов для извлечения i-го ресурса.

Введём понятие индекс отзывчивости распределённой системы, под которым будем понимать степень предоставления ресурсов (данных) поставщиком сервиса. Индекс отзывчивости системы может быть определён как

$$\lambda = \upsilon_+(k) - \upsilon_-(k), k \in [0,1], \quad (8)$$

где $\upsilon_+(k)$ – частота удовлетворения потребности пользователя в ресурсе r за один запрос из пары $\{\theta_i^{ws}, \theta_i^{http}\}$,

$\upsilon_+(k)$ – частота удовлетворения потребности пользователя в ресурсе r за несколько запросов из пары $\{\theta_i^{ws}, \theta_i^{http}\}$.

Соответственно, $\lambda \to \max$ является нашей основной целей, что можно интерпретировать как свойство системы полностью удовлетворять потребность в ресурсах (данных) пользователя за один запрос из пары $\{\theta_i^{ws}, \theta_i^{http}\}$. Максимизация этого параметра позволяет снизить лишние накладные расходы на обслуживание дополнительных запросов к поставщику сервиса.

Исходя из вышеизложенного, критерий качества регулирования разработанной системы можно определить как

$$K(t) = \lambda_{set}(t) - \lambda_{\text{current}}(t), \quad (9)$$

где $\lambda_{set}(t)$ – установленное значение индекса отзывчивости системы,

$\lambda_{\text{current}}(t)$ – действительное значение индекса отзывчивости системы.

Для процесса управления разработанной системы важно определить сбалансированность между двумя отслеживаемыми параметрами, введёнными нами ранее, а именно, между точностью идентификации объектов и индексом отзывчивости. Для решения данной задачи можно использовать метод среднего гармонического взвешенного [13, 14], который определяется следующим образом. Пусть имеется массив вещественных чисел $\{\varepsilon_1, \varepsilon_2, ..., \varepsilon_n\}$, определённый значениями точности идентификации объектов, и массив вещественных чисел $\{\lambda_1, \lambda_2, ..., \lambda_n\}$, сформированный из значений отзывчивости системы, тогда среднее гармоническое взвешенное может быть рассчитано как

$$H = \frac{\sum_{i=1}^{n} \lambda_i}{\sum_{i=1}^{n} \frac{\lambda_i}{\varepsilon_i}} = \frac{n}{\sum_{i=1}^{n} \frac{1}{\varepsilon_i}} = \frac{n}{\frac{1}{\varepsilon_1} + \frac{1}{\varepsilon_2} + ... + \frac{1}{\varepsilon_n}}. \quad (10)$$

Подход, основанный на методе среднего гармонического взвешенного, обладает явным преимуществом, в отличие от, например, невзвешенных средних, в связи с тем, что он учитывает дисбаланс между классами. В рамках нашей задачи явно сформированы два класса значений, определяемых величинами такими, как точность идентификации объектов и индекс отзывчивости, которые при первом взгляде на эти величины могут показаться взаимосвязанными, а в некоторых случаях и взаимодополняющими друг друга. Однако взаимосвязь между этими значениями довольно-таки сложна. В зависимости от наших целей мы можем быть заинтересованы в различных соотношениях между точностью идентификации объектов и индексом отзывчивости, то есть в распределённой переменной. Классическая оценка на основе метода среднего гармонического взвешенного может не давать качественные оценки, например, в случае, если значение класса отзывчивости системы равно 0, то данный метод просто не учтёт значение другого класса и итоговое значение будет равно 0. Помимо этого, существенным недостатком классического метода, является то обстоятельство, что отсутствует какая-либо явная возможность управлять балансом между точностью идентификации объектов и индексом отзывчивости, то есть отсутствует возможность определить, насколько для нас важна точность идентификации объектов, нежели чем индекс отзывчивости. На практике для решения данной задачи либо используются другие методы и оценки, либо разработаны трансформированные методы расчёта среднего гармонического взвешенного. Исходя из вышеизложенного, в данной работе введём коэффициент баланса γ, следовательно, модернизированная оценка взвешенного гармонического среднего будет определена как

$$\hat{H} = \sum_{i=1}^{n} \gamma_i * H_i, \gamma \in [0, n], \quad (11)$$

где γ_i – коэффициент баланса i-го класса (можно отождествить с вероятностью выбора соответствующего класса),

H_i – значение среднего гармонического взвешенного соответствующего класса,

n – общее количество классов.

Так как в рамках нашей задачи выделены 2 класса (n = 2), то модернизированная оценка взвешенного гармонического среднего для данного случая будет определена как

$$\hat{H} = \gamma_1 * H_1 + \gamma_2 * H_1 = \gamma_1 * \frac{n}{\frac{1}{\varepsilon_1} + \frac{1}{\varepsilon_2} + \ldots + \frac{1}{\varepsilon_n}} + \gamma_2 * \frac{n}{\frac{1}{\lambda_1} + \frac{1}{\lambda_2} + \ldots + \frac{1}{\lambda_n}}. \quad (12)$$

Такой подход позволяет посредством установки значения параметров γ_1 и γ_2 балансировать между точностью идентификации объектов и индексом отзывчивости. Так, например, установка $\gamma_1 = \gamma_2 = 1.0$ говорит о сбалансированном весе между точностью идентификации объектов и индексом отзывчивости системы.

Заключение

В данной работе предложен способ создания программных шлюзов для интеграции гетерогенных серверов в рамках клиент-серверного взаимодействия, который основан на использовании распространённых программных компонентов, что упрощает процесс внедрения предложенной системы в существующие проекты, которые написаны с использованием языка программирования Python и фреймворка Django. Предложенный способ даёт возможность интеграции серверов в рамках программного шлюза, основная задача которого заключается в ретрансляции запросов клиентского приложения во внешние удалённые серверы, которые могут быть связаны с программным шлюзом посредством WebSocket и HTTP. Это стало возможным благодаря внедрению и использованию django channels. Такой подход позволяет в существующие Django проекты внедрять как WebSocket клиентов, так и HTTP клиентов, а также даёт возможность обработки как синхронных, так и асинхронных функций, тем самым расширив возможности разработчиков клиентских приложений. Помимо этого, используемые программное обеспечение и библиотеки для формирования данного программного шлюза, а именно стек Ngnix+Gunicorn+Daphne, позволяют осуществить балансировку HTTP запросов клиентов, что, в свою очередь, позволит повысить скорость и стабильность всего клиент-серверного приложения. При таком конфигурировании системы Daphne будет являться резервным сервером/каналом, который может принимать проксированные Ngnix в случае неисправности Gunicorn или миграции со шлюзового протокола WSGI на ASGI. Предложенная математическая модель распределённой системы учитывает характер поступающих запросов, с возможностью тонкой настройки выделенных в данной работе параметров, что можно использовать при решении задачи управления вычислительной нагрузки [13] в рамках клиент-серверного приложения с многослойной архитектурой.

Список литературы

1. The web framework for perfectionists with deadlines | Django. [Electronic resource]. URL: https://www.djangoproject.com/ (date of access: 21.11.2021).

2. Введение в WSGI-серверы: Часть первая. [Электронный ресурс]. URL: https://habr.com/ru/post/426957/ (дата обращения: 21.11.2021).

3. The uWSGI project. [Electronic resource]. URL: https://uwsgi-docs.readthedocs.io/en/latest/ (date of access: 21.11.2021).

4. Gunicorn. [Electronic resource]. URL: https://gunicorn.org/ (date of access: 21.11.2021).

5. Анализ производительности WSGI-серверов: Часть вторая. [Электронный ресурс]. URL: https://habr.com/ru/post/427217/ (дата обращения: 21.11.2021).

6. Настройка Gunicorn и uWSGI, сравнение производительности. [Электронный ресурс]. URL: https://proft.me/2011/08/16/nastrojka-gunicorn-i-uwsgi-sravnenie-proizvoditeln/ (дата обращения: 22.11.2021).

7. Daphne. [Electronic resource]. URL: https://github.com/django/daphne (date of access: 22.11.2021).

8. Uvicorn. [Electronic resource]. URL: https://www.uvicorn.org/ (date of access: 22.11.2021).

9. Tyagi K., Karmarkar A., Kaur S., Kulkarni S., Das R. Crop Health Monitoring System. 2020 International Conference for Emerging Technology (INCET).2020. P. 1–5.

10. Hobson T.C., Doucet M., Leal R.M.F. Django remote submission. Journal of Open Source Software. 2017. Vol. 2. No. 16. P. 366.

11. Akbulut A., Perros H.G. Software versioning with microservices through the api gateway design pattern. 2019 9th International Conference on Advanced Computer Information Technologies (ACIT). 2019. P. 289–292.

12. Sathiyamoorthi V., Bhaskaran M. Data preprocessing techniques for pre-fetching and caching of web data through proxy server. International journal of Computer Science and Network security. 2011. Vol. 11. No. 11. P. 92–98.

13. Marcelino F.J., Escubio M.T., Baquirin R.B. A Comparative Analysis of Quantum Time based on Arithmetic, Geometric, and Harmonic Mean for Dynamic Round Robin Scheduling. Proceedings of 2020 the 6th International Conference on Computing and Data Engineering. 2020. P. 80–85.

14. Z. Li, F. Nie, X. Chang, Y. Yang. Beyond trace ratio: Weighted harmonic mean oftrace ratios for multiclass discriminant analysis. IEEE Trans. Knowl. Data Eng. 2017. Vol. 29. No. 10. P. 2100–2110.