МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное образовательное учреждение высшего образования «Южный федеральный университет»

Институт математики, механики и компьютерных наук им. И. И. Воровича
Кафедра информатики и вычислительного эксперимента

Ложкина Вера Михайловна

РЕАЛИЗАЦИЯ РАСПРЕДЕЛЁННОГО ОТКАЗОУСТОЙЧИВОГО ПРОСТРАНСТВА КОРТЕЖЕЙ СРЕДСТВАМИ ЯЗЫКА PYTHON

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
по направлению 01.03.02 — Прикладная математика и информатика

Научный руководитель —
ст. преп. Брагилевский Виталий Николаевич

Ростов-на-Дону — 2018

Содержание

Введение

На сегодняшний день роль информации в жизни человека стала невероятно велика. Поскольку её объёмы быстро увеличивается, возникает вопрос о хранении, обработке и передаче информации. Для этих целей используются различные информационные технологии. Ярким примером является всемирная сеть Интернет, с её помощью можно быстро произвести обмен данными и получить доступ к необходимой информации.

В современном мире информационные технологии широко используются в социально-экономической сфере (социальные сети, бухгалтерский учёт), обеспечивают функционирование производственных и банковских систем. В основе реализации таких сложных и обширных структур лежат распределённые системы [1—3], они являются основой различных объектов, например, веб-сервисов и пиринговых сетей.

Важной характеристикой распределённой системы является отказоустойчивость — способность правильно функционировать в случае отказа отдельных компонент [4]. В свою очередь компоненты распределённой системы могут быть разъединены в пространстве, поэтому обмен сообщениями в системе осуществляется по ненадёжным каналам связи: сообщения, посылаемые системе или самой системой, могут просматриваться, перехватываться и подменяться. Это приводит к неправильному функционированию или отказу отдельных компонент или системы в целом. Такое поведение может повлечь за собой, например, финансовые потери, поэтому в распределённых системах обеспечению информационной безопасности уделяется особое внимание.

Взаимодействие компонент системы можно рассматривать в рамках задачи византийских генералов [5]. Этот термин используется в криптологии для определения задачи взаимодействия нескольких удаленных объектов. Все объекты получают приказы из одного

центра. Часть объектов, включая центр, могут быть злоумышленниками и вести себя нехарактерным образом (например, совершать деструктивные для системы действия), то есть провоцировать византийские ошибки.

Для повышения надёжности системы случайные и умышленные сбои интерпретируются как византийские ошибки. В этом случае можно использовать соответствующие отказоустойчивые техники, которые смогут защитить систему и от случайных сбоев, и от умышленных вторжений [6—8].

В данной работе рассматривается византийское пространство кортежей — открытая распределённая устойчивая к византийским ошибкам система, основанная на пространстве кортежей без распределённой памяти, в которой процессы взаимодействуют путём обмена сообщениями [6; 9].

Система реализована при помощи языка программирования Python версии 3.6, поскольку данный язык поддерживает несколько парадигм программирования, динамическую типизацию, а также предоставляет удобные высокоуровневые структуры данных [10]. Стандартная библиотека Python включает в себя большой набор полезных функций, что расширяет возможности разработчика и повышает его производительность [11]. Для реализации распределённой системы была использована бесплатная интегрированная среда разработки PyCharm Community Edition 2017.2, распространяющаяся под лицензией Apache 2. Данная среда разработки предоставляет средства для анализа кода и графический отладчик, что облегчает написание кода и поиск ошибок, экономя время разработчика.

Раздел 1 включает в себя определение основных терминов, использованных в данной работе, таких как распределённая система, отказоустойчивость и задача византийских генералов. Раздел 2 посвящён реализации византийского пространства кортежей. Данный раздел включает в себя восемь подразделов. В подразделе 2.1 опи-

сана системная модель византийского пространства кортежей (BTS). В подразделах 2.2-2.5 рассматривается программная реализация распределённой отказоустойчивой системы BTS на языке программирования Python, каждый модуль реализации описывается отдельно. Особое внимание уделяется операциям манипулирования данными, описанным в подразделах 2.6-2.8, поскольку именно их реализация обеспечивает отказоустойчивость системы. Результаты работы реализованной системы, а также их анализ представлены в разделе 3.

1. Распределённые вычислительные системы и отказоустойчивость

Распределённая вычислительная система — это набор независимых компьютеров, реализующий параллельную обработку данных на многих вычислительных узлах [1]. С точки зрения пользователя этот набор является единым механизмом, предоставляющим полный доступ к ресурсам. Существует возможность добавления новых ресурсов и перераспределения их по системе, возможность добавления свойст и методов, но информация об этих событиях скрыта от пользователя.

Одной из важнейших характеристик распределённых систем является отказоустойчивость. Отказоустойчивость — это свойство системы сохранять работоспособность в том случае, если какие-либо составляющие её компоненты перестали правильно функционировать [1]. Компоненты системы могут стать нероботоспособны по различным причинам, например, из-за технологических сбоев или атак безопасности.

Большинство современных распределённых систем имеют характеристики открытых систем. Открытая распределённая система предполагает использование служб, вызов которых требует стандартного синтаксиса и семантики [2]. Такая система может иметь неиз-

вестное количество ненадёжных и неоднородных участников, к тому же участникам не нужно быть активными одновременно (свойство разъединённости во времени) и не обязательно что-то знать друг о друге (свойство разъединённости в пространстве). Связь между узлами распределённой системы является ненадёжной (может прерываться, что повлечёт за собой потерю сообщений), обмен сообщениями может происходить не мгновенно, а с существенной задержкой. Кроме того, любой узел системы может отказать или быть выключен в любой момент времени. Все эти факторы неизбежно приводят к неправильному функционированию системы.

Один из способов улучшить её надёжность — это интерпретировать случайные или умышленные неполадки как византийские ошибки (в терминах задачи византийских генералов), тогда использование отказоустойчивых техник сможет сделать координационную составляющую системы отказоустойчивой и для случайных сбоев, и для умышленных вторжений.

Задача византийских генералов — это задача взаимодействия нескольких удалённых абонентов, получивших сообщения из одного центра, причём часть этих абонентов, в том числе центр, могут быть предателями, то есть могут посылать заведомо ложные сообщения с целью дезинформирования. Нахождение решения задачи заключается в выработке единой стратегии действий, которая будет являться выигрышной для всех абонентов [5].

Формулировка задачи состоит в следующем. Византийская армия представляет собой объединение некоторого числа легионов, каждым из которых командует свой генерал, генералы подчиняются главнокомандующему армии Византии. Поскольку империя находится в упадке, любой из генералов и даже главнокомандующий могут быть заинтересованы в поражении армии, то есть являться предателями. Генералов, не заинтересованных в поражении армии, будем называть верными. В ночь перед сражением каждый из генералов получает от главнокомандующего приказ о действиях во время сражения:

атаковать или отступать. Таким образом, имеем три возможных исхода сражения:

- благоприятный исход: все генералы атакуют противника, что приведёт к его уничтожению и победе Византии.
- промежуточный исход: все генералы отступят, тогда противник не будет побеждён, но Византия сохранит свою армию.
- неблагоприятный исход: некоторые генералы атакуют противника, некоторые отступят, тогда Византийская армии потерпит поражение.

Так как главнокомандующий тоже может оказаться предателем, генералам не следует доверять его приказам. Однако, если каждый генерал будет действовать самостоятельно, независимо от других генералов, то вероятность наступления благоприятного исхода становится низкой. Таким образом, генералам следует обмениваться информацией между собой для того, чтобы прийти к единому решению.

В данной работе рассматривается распределённая система под названием «Византийское пространство кортежей» (BTS — Byzantine Tuple Space) [6]. Все сбои, происходящие в этой системе, интерпретируются как византийские ошибки, для устранения которых используются определённые отказоустойчивые техники.

2. BTS: Византийское пространство кортежей

2.1. Системная модель

Системная модель византийского пространства кортежей BTS предполагает бесконечное число процессов-клиентов $\Pi = \{p_1, p_2, \dots\}$, которые коммуницируют со множеством из n серверов $U = \{s_1, s_2, \dots, s_n\}$ при помощи обмена сообщениями.

Будем полагать, что случайное число клиентов и связка серверов из $f \leqslant \left\lfloor \frac{n-1}{3} \right\rfloor$ штук могут быть подвержены византийским ошибкам: они могут произвольным образом отклоняться от их спецификаций и работать в сговоре, чтобы изменить поведение системы. Такие процессы будем называть неисправными, а правильно работающие процессы — корректными.

Основа распределённой системы BTS — это пространство кортежей, которое также является ядром языка программирования Linda [12], предназначенного для построения эффективных параллельных программ. Кортеж — это структура данных, представляющая собой неизменяемый список фиксированной длины, элементы которого могут относиться к различным типам данных. Два кортежа t_1 и t_2 считаются идентичными, если совпадают их длины, а также типы и значения соответствующих полей. Хранилище кортежей, в котором доступ к элементам может осуществляться параллельно, называется пространством кортежей [9].

Каждый сервер $s \in U$ при запуске получает на вход имя файла, в котором хранится содержимое пространства кортежей. Это необходимо для создания сервером s локальной копии пространства кортежей T_s. Кроме того, создаётся изначально пустое множество кортежей для удаления R_s. Будем полагать, что идентичных кортежей в пространстве не существует, тогда к двум описанным множествам кортежей могут быть применены стандартные операции над множествами.

Для искусственной имитации неисправных серверов будем при запуске подавать им на вход файл с неправильным пространством кортежей. Таким образом, множества T_{s_1} и T_{s_2}, принадлежащие корректному серверу s_1 и неисправному серверу s_2 соответственно, будут иметь пустое пересечение, что повлечёт за собой конфликты и позволит проверить отказоустойчивость системы.

Каждый сервер s реализует три операции манипулирования данными (кортежами):

- out — запись кортежа в пространство кортежей.
- rd — недеструктивное чтение кортежа.
- in — деструктивное чтение (извлечение) кортежа.

Заметим, что операции rd и in не являются блокирующими в отличие от их аналогов в языке пограммирования Linda.

Определим такое понятие, как шаблон кортежа — это кортеж, некоторые поля которого неопределены и не представляют важности. Будем говорить, что кортеж соответствует шаблону, если длина кортежа равна длине шаблона и определённые в шаблоне поля совпадают по типу и значению с соответствующими полями кортежа.

Операция записи out принимает в качестве входного параметра кортеж, все поля которого определены. Операции чтения rd и in принимают в качестве входного параметра шаблон кортежа, по которому производится поиск соответствующих ему кортежей в пространстве.

Благодаря наличию операций чтения/записи, пространство кортежей можно рассматривать как разновидность распределённой памяти: например, одна группа процессов записывает данные в пространство, а другая группа извлекает их и использует в своей дальнейшей работе. Все процессы работают с пространством кортежей по принципу: поместить, изъять, найти. Если один процесс уже изъял какой-либо кортеж из пространства кортежей, то другой процесс не сможет получить доступ к данному кортежу. Чтобы изменить кортеж, находящийся в пространстве кортежей, необходимо для начала изъять его оттуда, а затем поместить изменённый кортеж обратно [3].

Распределённая система не может полагаться на какой-то один конкретный узел, так как в случае его отказа восстановить систему будет невозможно [4]. Это означает, что выполнение любой операции манипулирования кортежами не может производиться только на одном из имеющихся узлов. Для решения этой проблемы разобьём множество U на кворумы и получим кворум-систему, тогда каждая операция чтения/записи будет выполняться в соответствующем кворуме,

что обеспечит правильное функционирование системы в случае отказа f узлов.

Кворум-система представляет собой множество кворумов серверов $\mathcal{Q} \in 2^U$, в котором каждая пара кворумов из $\mathcal{Q}$ пересекается на достаточно многих серверах и всегда есть кворум со всеми корректными серверами. Существование пересечений между кворумами позволяет совершенствовать протоколы чтения/записи, позволяя поддерживать целостность разделённой переменной, даже если эти операции были выполнены в разных кворумах системы.

Будем полагать, что $n > 3f + 1$. Разделим кворумы на два типа ($\mathcal{Q} = \mathcal{Q}_r \cup \mathcal{Q}_w$):

- кворумы чтения ($Q_r \in \mathcal{Q}_r$), мощность каждого кворума $|Q_r| = \left\lceil \frac{n+f+1}{2} \right\rceil$ серверов,
- кворумы записи ($Q_w \in \mathcal{Q}_w$), мощность каждого кворума $|Q_w| = \left\lceil \frac{n+f+1}{2} \right\rceil + f$ серверов.

Так как $|Q_r| + |Q_w| > n$, то можно ожидать, что полученное при чтении значение будет наиболее актуальным, поскольку хотя бы один из узлов кворума $|Q_r|$ также участвует в выплнении операции записи.

Взаимодействие клиентов с системой происходит посредством вспомогательной промежуточной координационной инфраструктуры, как показано на рисунке 1.

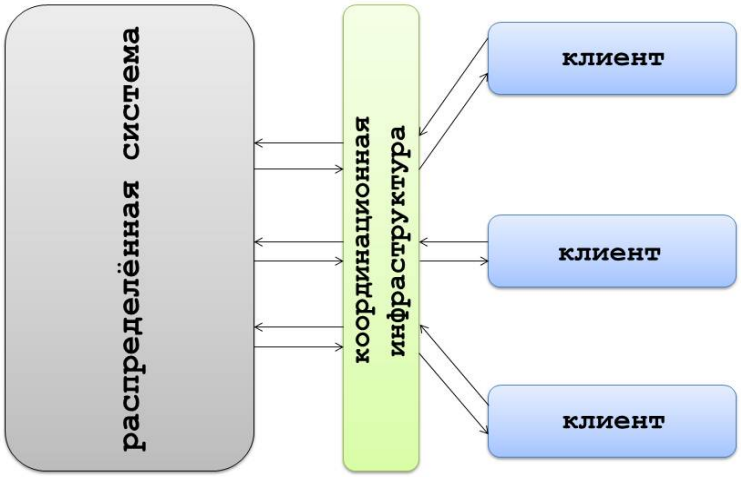

Рисунок 1 — Взаимодействие клиентов с распределённой системой

Когда клиент p посылает запрос системе, этот запрос сначала обрабатывается координационной инфраструктурой и только после этого отправляется системе. При получении ответа от системы координационная инфраструктура также обрабатывает полученную информацию, а затем отправляет ответ клиенту p. Поэтому клиентская часть операций чтения/записи исполняется на стороне координационной структуры, а не на стороне клиента.

Реализация системы BTS состоит из четырёх модулей:

- модуль `secondary_functions` включает в себя вспомогательные функции,
- модуль `client` реализует клиентский интерфейс,
- модуль `BTS_infrastructure` реализует деятельность координационной инфраструктуры,
- модуль `BTS_server` реализует деятельность сервера $s \in U$.

Рассмотрим реализацию компонент более подробно.

2.2. Вспомогательные функции

Вспомогательные функции представлены в модуле secondary_functions. Данный модуль включает в себя функции сетевого взаимодействия (get_message, send_message, connect_to_server), а также функцию создания и записи пространства кортежей в файл (create_ts_file).

Функции сетевого взаимодействия используют в своей работе возможности модуля socket из стандартной бибилиотеки Python [13]. Функция connect_to_server принимает в качестве входного параметра порт, к которому необходимо подключиться (см. листинг 2.1). При удачном подключении возвращается сокет, который можно использовать для обмена сообщениями в дальнейшей работе, иначе возвращается объект None, сигнализирующий о неудаче.

Листинг 2.1. Модуль secondary_functions, функция connect_to_server

```
1  def connect_to_server(port):
2    sock = socket(AF_INET, SOCK_STREAM)
3    i = 100
4    while i:
5      try:
6        sock.connect(('localhost', port))
7      except ConnectionRefusedError:
8        i -= 1
9      else:
10       return sock
11   return None
```

Функция send_message используются для отправки данных по сети (см. листинг 2.2), она имеет два входных параметра: сокет, с помощью которого осуществляется отправка сообщения, и само сообщение, которое требуется отправить. Чтобы преобразовать сообщение к виду, пригодному для записи в сокет, используется функция dumps из модуля pickle [14], отвечающая за сери-

Листинг 2.2. Модуль secondary_functions, функция send_message

```
1 def send_message(sock, msg):
2   return sock.send(pickle.dumps(msg))
```

ализацию объектов. По окончании операции записи в сокет функция send_message завершает свою работу и возвращает количество отправленных байт.

Функция get_message используются для получения данных из сети (см. листинг 2.3), она имеет единственный входной параметр — сокет, из которого осуществляется чтение информации. Во избежание блокировки устанавливается тайм-аут длительностью в десять минут. Если полученное сообщение оказалось пустым, то воз-

Листинг 2.3. Модуль secondary_functions, функция get_message

```
1  def get_message(sock):
2    sock.settimeout(600)
3    msg = b""
4    tmp = b""
5    while True:
6      try:
7        tmp = sock.recv(1024)
8      except error:
9        break
10     if len(tmp) < 1024:
11       break
12     msg += tmp
13   msg += tmp
14   if len(msg) > 0:
15     return pickle.loads(msg)
16   return None
```

вращается объект None, иначе сообщение десериализуется с помощью функции loads из модуля pickle и возвращается функцией get_message.

Функция create_ts_file создаёт пространство кортежей и записывает его в файл. Реализация представлена в листинге 2.4. Дан-

ная функция используется для того, чтобы в дальнейшем каждый сервер системы при запуске смог прочитать информацию из созданного при помощи `create_ts_file` файла и сконструировать локальную реплику пространства кортежей. Имя файла, объём создаваемого пространства, а также длина и значения полей входящих в него кортежей регулируются входными параметрами функции. Для преобразования созданного пространства кортежей в удобный формат и дальнейшей записи его в файл используется функция `dump` из модуля `json` [15].

2.3. Клиентский интерфейс

Интерфейс пользователя для взаимодействия с BTS реализован с помощью класса `Client` из модуля `client` (полный программный код представлен в [16]). Для того, чтобы начать работу, следует создать объект класса `Client`, передав конструктору в качестве входного параметра значение порта, который координационная инфраструктура использует для обмена сообщениями с клиентами.

Ввиду возможности возникновения ошибок для контроля исполняемых операций используется модуль `logging` [17] из стандартной библиотеки языка Python, отвечающий за логирование. Создание и настройка конфигурации лога происходит в конструкторе класса `Client` следующим образом:

Листинг 2.4. Модуль `secondary_functions`, функция `create_ts_file`

```
1 def create_ts_file(file_name, init_num, amount=500, length=100):
2   with open(file_name, 'w') as f:
3     list_of_tuples = list()
4     for i in range(init_num * amount, (init_num + 1) * amount):
5       list_of_tuples.append(tuple(j for j in range(i, i+length)))
6     json.dump(list_of_tuples, f)
```

```
  logging.basicConfig(filename='client_log.txt', level=logging.
     INFO, format="%(asctime)s - %(message)s")
```

При помощи функции `basicConfig` устанавливается имя файла для записи лога, уровень логирования и формат выводимых сообщений.

Для исполнения операций чтения/записи необходимо вызвать соответствующую функцию: `out_op`(t), `rd_op`($\bar{t}$) или `in_op`($\bar{t}$) (см. листинг 2.5).

Листинг 2.5. Модуля `client`, методы `out_op`, `rd_op`, `in_op`

```
1  def out_op(self, tup):
2    self.__op('out', tup)
3    logging.info('OUT: ' + str(tup))
4
5  def rd_op(self, temp):
6    resp = self.__op('rd', temp)
7    logging.info('RD: ' + str(resp))
8    return resp
9
10 def in_op(self, temp):
11   resp = self.__op('in', temp)
12   logging.info('IN: ' + str(resp))
13   return resp
```

Заметим, что операция записи `out_op` в качестве входного параметра принимает кортеж, а операции чтения `rd_op` и `in_op` — шаблон кортежа. Каждая из этих операций исполняется путём вызова приватного метода `__op` (см. листинг 2.6). Внутри данного метода формируется запрос в виде словаря `{'op': код операции, 'tup': кортеж}` или `{'op': код операции, 'temp': шаблон кортежа}`, содержание которого соответствует семантике исполняемой операции. Сформированный запрос отправляется координационной инфраструктуре. Для завершения операции записи ответ системы не требуется, а операции чтения ожидают, когда инфраструктура вернёт запрошенное значение.

Листинг 2.6. Модуль `client`, приватный метод `op`

```
def __op(self, op_type, tup=None):
  sock = connect_to_server(self.port)
  if sock is not None:
    if op_type == 'out':
      req = {'op': op_type, 'tup': tup}
    elif op_type in ['rd', 'in']:
      req = {'op': op_type, 'temp': tup}
    elif op_type == 'stop':
      req = {'op': op_type}
    send_message(sock, req)
    if op_type in ['rd', 'in', 'stop']:
      resp = get_message(sock)
      sock.close()
      if resp is None:
        return resp
      return resp['resp']
    else:
      sock.close()
  else:
    logging.info(str(op_type) + ': cannot connect to server')
    return None
```

Для корректного завершения работы системы в классе `Client` реализована функция `stop_op` (см. листинг 2.7), при исполнении которой инфраструктуре отправляется запрос о прекращении работы и ожидается ответ об успешном результате операции. По смыслу дан-

Листинг 2.7. Модуль `client`, метод `stop_op`

```
def stop_op(self):
  resp = self.__op('stop')
  logging.info('STOP: ' + str(resp))
  return resp
```

ная функция не должна являться частью клиентского интерфейса, она была включена в класс `Client` для удобства.

2.4. Координационная инфраструктура

Координационная инфраструктура системы BTS реализована в модуле `BTS_infrastructure` с помощью класса `BTS_infrastructure` (полный программный код представлен в [16]). Она представляет собой многопоточное приложение, которое обрабатывает запросы клиента и взаимодействует с серверами системы BTS.

Для того, чтобы настроить работу координационной системы, необходимо создать объект класса `BTS_infrastructure` и передать в его конструктор следующие параметры:

- порт, который будет прослушивать инфраструктура и на который будут поступать клиентские запросы,
- список портов, каждый из которых будет прослушивать определённый сервер системы BTS,
- количество неисправных серверов (предателей), которые будут порождать конфликты в системе,
- размер кворума записи,
- размер кворума чтения,
- имя файла, в котором хранится множество кортежей для корректных серверов,
- имя файла, в котором хранится множество кортежей для неисправных серверов.

В теле конструктора значения входных параметров присваиваются соответствующим свойствам класса, устанавливается значение флага `THREAD_POOL_ON` (данный флаг используется при обработке клиентских запросов), значение свойства `AMOUNT_OF_CLIENTS` устанавливается в нуль (данное свойство является счётчиком поступивших

клиентских запросов), множество серверов U разбивается на кворумы записи Q_w и чтения Q_r, а также настраивается конфигурация лога.

После создания объекта класса для запуска работы координационной инфраструктуры необходимо вызвать метод `run` (см. листинг 2.8). Если изначально количество неисправных серверов

Листинг 2.8. Модуль `BTS_infrastructure`, метод `run`

```
1 def run(self):
2   if self.AMOUNT_OF_ENEMIES >= self.AMOUNT_OF_SERVERS:
3     return False
4   self.start_servers(0, self.AMOUNT_OF_ENEMIES, self.
       WRONG_TS_FILE)
5   self.start_servers(self.AMOUNT_OF_ENEMIES, self.
       AMOUNT_OF_SERVERS, self.TS_FILE)
6   .........
```

было указано неверно, то есть не меньше количества всех имеющихся серверов, то возвращается значение `False`. Иначе работа функции продолжается: происходит запуск корректных и неисправных серверов с помощью метода `start_servers` (см. листинг 2.9). Метод `start_servers` имеет три входных параметра:

Листинг 2.9. Модуль `BTS_infrastructure`, метод `start_servers`

```
1 def start_servers(self, ind1, ind2, file):
2   for i in range(ind1, ind2):
3     try:
4       Popen('python BTS_server.py '+str(i)+' '+str(self.U[i])+' '
           +str(file)+' '+''.join(str(j)+' ' for j in self.U),
           stdout=PIPE, stderr=PIPE, shell=True)
5     except CalledProcessError:
6       logging.info('START_SERVERS: cannot start server ' + str(
           self.U[i]))
7     else:
8       logging.info('START_SERVERS: start server '+str(self.U[i]))
```

- два индекса, определяющие диапазон значений в списке портов `self.U`, которые будут прослушивать запускаемые серверы,
- имя файла, которое будет подано выбранным серверам на вход.

Программная реализация сервера системы BTS представлена в модуле BTS_server, который будет рассмотрен позже в разделе 2.5, данный скрипт BTS_server.py запускается с определёнными параметрами в новом процессе при помощи конструктора класса Popen из модуля subprocess [18] стандартной библиотеки Python. В скрипт передаются следующие аргументы:

- идентификатор сервера,
- порт, который будет прослушивать сервер,
- имя файла с хранящимся в нём пространством кортежей,
- список портов всех серверов.

Вернёмся к методу run. После запуска серверов создаётся и настраивается сокет, с помощью которого далее будет происходить коммуникация с клиентскими процессами (см. листинг 2.10). Клиентские запросы будут обрабатываться при помощи пула потоков ThreadPoolExecutor из модуля concurrent.futures [19] стандартной библиотеки Python, принимающего в качестве аргумента количество потоков, используемых в работе. Пока флаг THREAD_POOL_ON равен True, исполняется следующая последовательность инструкций: ожидается подключение клиента к порту, создаётся соответствующий клиентский сокет, количество полученных клиентских запросов увеличивается на единицу. При удачном завершении описанных инструкций задача с помощью метода submit добавляется в очередь для исполнения пулом потоков. Метод submit принимает в качестве входных параметров имя функции (worker), которую необходимо исполнить в одном из потоков пула, и значения аргументов (клиентский сокет и AMOUNT_OF_CLIENTS), с которыми эта функция будет запущена.

Флаг THREAD_POOL_ON может изменить своё значение только при получении запроса о прекращении работы от клиента. Если это произошло, то новые задачи перестают добавляться в очередь пула

Листинг 2.10. Модуль BTS_infrastructure, метод run (продолжение)

```
def run(self):
  .........
  s = socket(AF_INET, SOCK_STREAM)
  s.bind(('', self.INFRASTRUCTURE_PORT))
  s.listen(1)
  with ThreadPoolExecutor(10) as pool:
    while self.THREAD_POOL_ON:
      try:
        s.settimeout(30)
        client_s, client_addr = s.accept()
        self.AMOUNT_OF_CLIENTS += 1
      except error:
        pass
      else:
        pool.submit(self.worker,client_s,self.AMOUNT_OF_CLIENTS)
  s.close()
  logging.info('RUN: stop working')
  return True
```

потоков, ожидается завершение всех исполняющихся или ожидающих в очереди задач.

На этом работа метода run прекращается, возвращается значение True, свидетельствующее об успешном завершении работы координационной инфраструктуры.

Заметим, что метод run является блокирующим, то есть после запуска инфраструктуры и вплоть до её завершения продолжить дальнейшую работу будет невозможно.

Рассмотрим метод worker (см. листинг 2.11) более подробно. Он выполняет функцию получения и обработки запроса клиента, принимает в качестве входных параметром клиентский сокет и идентификатор клиентского процесса. Вспомним, что в функцию submit (добавление задачи в очередь пула потоков) в качестве второго аргумента функции worker передаётся значение свойства AMOUNT_OF_CLIENTS. Данное свойство используется для подсчёта клиентских запросов, будем полагать, что каждый клиент-

Листинг 2.11. Модуль BTS_infrastructure, метод worker

```
1  def worker(self, client, pid):
2    req = get_message(client)
3    if req is not None:
4      try:
5        if req['op'] == 'out':
6          self.out(req['tup'])
7        elif req['op'] == 'rd':
8          send_message(client,{'resp': self.rdp(req['temp'])})
9        elif req['op'] == 'in':
10         send_message(client,{'resp': self.inp(pid,req['temp'])})
11       elif req['op'] == 'stop':
12         self.THREAD_POOL_ON = False
13         self.stop_servers()
14         send_message(client, {'resp': 'ok'})
15       else:
16         logging.info('WORKER: wrong request ' + str(req))
17     except KeyError:
18       logging.info('WORKER: wrong request ' + str(req))
19     client.close()
```

ский запрос был послан уникальным клиентом (согласно системной модели BTS множество клиентских процессов бесконечно), тогда в качестве идентификатора клиентского процесса можно использовать номер запроса, что и осуществляется при помощи свойства AMOUNT_OF_CLIENTS.

Работу метода worker можно разделить на два этапа: получение сообщения от клиента и обработка полученного запроса. Для того, чтобы получить сообщение с запросом от клиента, используется функция get_message из описанного в разделе 2.2 модуля secondary_functions. После получения запроса определяется код операции, которую необходимо исполнить, вызывается соответствующая функция и при необходимости отправляется ответ клиенту.

При получении запросов на исполнение операций out, rd или in вызываются методы out, rdp или inp, которые будут рассмотрены позже в разделах 2.6, 2.7 и 2.8 соответственно.

При получении запроса на исполнение операции *stop* флаг `THREAD_POOL_ON` устанавливается в значение `False`, после чего прекращается обработка клиентских запросов, вызывается метод `stop_servers` (см. листинг 2.12), при исполнении которого всем запущенным серверам посылается запрос о завершении работы и ожидается ответ об удачном исходе данной операции.

Листинг 2.12. Модуль `BTS_infrastructure`, метод `stop_servers`

```
1  def stop_servers(self):
2    for i in self.U:
3      s = connect_to_server(i)
4      if s is not None:
5        send_message(s, {'op': 'stop'})
6        resp = get_message(s)
7        s.close()
8        if resp['resp'] != 'ok':
9          logging.info('STOP_SERVERS: cannot stop server '+str(i))
10       else:
11         logging.info('STOP_SERVERS: stop server '+str(i))
12     else:
13       logging.info('STOP_SERVERS: cannot stop server '+str(i))
```

2.5. Серверная составляющая системы

Серверная составляющая системы BTS реализована в модуле `BTS_server` (полный программный код представлен в [16]), он представляет собой скрипт, запуск которого приводит в исполнение сервер системы. Как уже было отмечено в разделе 2.4, скрипт запускается со следующими аргументами:

- идентификатор сервера,
- порт, который будет прослушивать сервер,
- имя файла с хранящимся в нём пространством кортежей,
- список портов всех серверов.

Для обработки аргументов используется библиотека `argparse` [20]. Она предоставляет возможности анализа аргументов командной строки, конвертирования стркоовых аргументов в другие объекты программы и вывода информационных подсказок.

Для мониторинга работы сервера настраивается логирование.

При запуске сервер считывает из полученного файла кортежи и создаёт локальную копию пространства кортежей при помощи функции `read_from_ts_file` (см. листинг 2.13). На сервере корте-

Листинг 2.13. Модуль `BTS_server`, функция `read_from_ts_file`

```
1 def read_from_ts_file(file_name):
2   global TS
3   with open(file_name, 'r') as f:
4     data = json.load(f)
5     for i in data:
6       TS.add(tuple(i))
```

жи хранятся в контейнере `set` (переменная `TS`), что позволяет исключить наличие идентичных кортежей и сделать проверку на вхождение быстрой, поскольку `set` в качестве базовой структуры данных использует хэш-таблицу [10].

Далее, как и в случае с координационной инфраструктурой, создаётся сокет, через который будет осуществляться коммуникация с инфраструктурой, запускается пул потоков [21] и начинается обработка входящих запросов.

Обработка входящих запросов осуществляется при помощи функции `worker`, её программная реализация практически идентична одноимённой функции из модуля `BTS_infrastructure` и не нуждается в представлении. В теле данной функции сначала происходит считывание сообщения, полученного от инфраструктуры, далее по содержимому полученного сообщения определяется тип комнады и вызывается соответствующая одноимённая функция. Полный перечень возможных команд приведён ниже:

- `out` — команда операции записи out,

- `rd` — команда операции недеструктивного чтения rd,
- `in` — команда операции деструктивного чтения in,
- `enter` — команда входа в критическую секцию,
- `exit` — команда выхода из критической секции,
- `accept1` — команда первого этапа утверждения удаляемого кортежа,
- `accept2` — команда второго этапа утверждения удаляемого кортежа,
- `stop` — команда о прекращения работы сервера.

Первые три команды связаны с операциями манипулирования кортежами, следующие четыре команды — с реализацией операции in (они будут рассмотрены позже в разделе 2.8). Последняя команда `stop` устанавливает флаг `THREAD_POOL_ON` в значение `False`, что приводит к прекращению работы пула потоков, а затем — к завершению работы сервера.

2.6. Операция записи out

Операция $out(t)$ добавляет кортеж t в пространство кортежей. На стороне инфраструктуры эта операция реализуется с помощью метода `out(self, t)`, программный код которого приведён в листинге 2.14. Для того, чтобы добавить кортеж t в пространство, производится подключение к серверам системы, состоящим в кворуме записи, и в случае удачного подключения отправляется запрос `{'op': 'out', 'tup': t}` на добавление кортежа.

Заметим, что при вызове данной функции ждать ответа от серверов нет необходимости. Будем считать, что операция завершится в тот момент, когда все корректные серверы из кворума записи получат кортеж t.

Листинг 2.14. Модуль BTS_infrastructure, метод out

```
def out(self, t):
  for i in self.QW:
    sock = connect_to_server(i)
    if sock is not None:
      send_message(sock, {'op': 'out', 'tup': t})
      logging.info('OUT: send '+str(t)+' to server '+str(i))
      sock.close()
    else:
      logging.info('OUT: cannot connect to server ' + str(i))
  logging.info('OUT: recorded ' + str(t))
```

На стороне сервера операция $out(t)$ реализуется с помощью функции out(t) (см. листинг 2.15). При получении запроса на до-

Листинг 2.15. Модуль BTS_server, функция out

```
def out(t):
  global TS, RS
  if t not in RS:
    TS.add(t)
    logging.info('OUT: add ' + str(t) + ' to TS')
  try:
    RS.remove(t)
  except KeyError:
    pass
  else:
    logging.info('OUT: remove ' + str(t) + ' from RS')
```

бавление кортеж t добавляется в пространство кортежей только в том случае, если этот кортеж ранее не был из него удалён (то есть его нет во множестве RS). Это необходимо для того, чтобы один и тот же кортеж не был удалён дважды.

2.7. Операция чтения rd

Операция недеструктивного чтения $rd(\bar{t})$ в качестве входного параментра принимает шаблон кортежа $\bar{t}$ и возвращает копию кортежа, соответствующего шаблону, не извлекая его из пространства.

Реализация этой операции на стороне инфраструктуры представлена в виде метода `rdp(self, temp, ts=None)` (полный программный код приведён в [16]), имеющего два параметра: шаблон кортежа и объект `ts` (по умолчанию равный `None`), значение которого будет объяснено позже. При получении запроса на исполнение операции rd функция `rdp` вызывается с единственным первым параметром.

Листинг 2.16. Модуль `BTS_infrastructure`, метод `rdp`

```
1  def rdp(self, temp, ts=None):
2    if ts is None:
3      ts = {i: [None, None] for i in self.QR}  #port: (socket, set)
4      for i in self.QR:
5        ts[i][0] = connect_to_server(i)
6        if ts[i][0] is not None:
7          send_message(ts[i][0], {'op': 'rd', 'temp': temp})
8          logging.info('RDP: send '+str(temp)+' to server '+str(i))
9        else:
10         ts.pop(i)
11     for i in ts.keys():
12       msg = get_message(ts[i][0])
13       ts[i][0].close()
14       if msg is not None:
15         ts[i][1] = msg['ts']
16       else:
17         ts[i][1] = None
18       logging.info('RDP: receive ' + str(ts[i][1]) + ' from
           server ' + str(i))
19   .........
```

Суть работы функции заключается в следующем. Каждому серверу из кворума чтения посылается запрос на исполнение операции $rd(\bar{t})$ (см. листинг 2.16). Ответы от серверов кворума приходят в виде множества подходящих под шаблон $\bar{t}$ кортежей и записываются в словарь `ts`, имеющий структуру: `port: (socket, set)`, где `port` — порт, который прослушивает конкретный сервер, `socket` — сокет, с помощью которого происходит коммуникация, `set` — множество подходящих кортежей, хранящихся на данном сервере. Заметим, что словарь `ts` может передаваться в функцию уже

в готовом виде, этот факт используется в реализации операции $in(\bar{t})$, которая будет рассмотрена в разделе 2.8.

Из всех полученных кортежей выбирается один общий кортеж t, который располагается как минимум на $f + 1$ сервере (см. листинг 2.17). Чтобы найти кортеж t, необходимо подобрать та-

Листинг 2.17. Модуль `BTS_infrastructure`, метод `rdp` (продолжение)

```
1  def rdp(self, temp, ts=None):
2    .........
3    t = None
4    for i in combinations(ts.keys(), self.AMOUNT_OF_ENEMIES + 1):
5      ts_intersection = ts[i[0]][1]
6      for j in i:
7        if ts[j][1] is None:
8          ts_intersection = None
9          break
10       ts_intersection = ts_intersection.intersection(ts[j][1])
11       if len(ts_intersection) == 0:
12         break
13     if ts_intersection is not None:
14       if len(ts_intersection) > 0:
15         t = ts_intersection.pop()
16         break
17   logging.info('RDP: selected ' + str(t))
18   return t
```

кую комбинацию из $f + 1$ сервера, пересечение множеств подходящих кортежей которых будет непусто. Для этого рассматриваются всевозможные сочетания из $f + 1$ сервера при помощи функции `combinations` из модуля `itertools` [22]. Для нахождения пересечений множеств подходящих кортежей используются методы класса `set` [23]. После нахождения кортежа t он возвращается в качестве выходного параметра. Если подходящего кортежа не нашлось, то возвращается объект `None`, свидетельствующий о том, что операция чтения не увенчалась успехом.

Рассмотрим реализацию функции недеструктивного чтения `rdp(sock, temp)` на стороне сервера (см. листинг 2.18). Функ-

ция принимает два входных параметра: сокет и шаблон кортежа. Для того, чтобы найти подходящие под шаблон кортежи, произво-

Листинг 2.18. Модуль BTS_server, функция rdp

```
1  def rdp(sock, temp):
2    global TS
3    temp_set = set()
4    for t in TS:
5      if match(temp, t):
6        temp_set.add(t)
7    logging.info('RDP: temp '+str(temp)+', set '+str(temp_set))
8    if sock is not None:
9      send_message(sock, {'ts': temp_set})
10   else:
11     return temp_set
```

дится один проход по локальной копии пространства кортежей TS, каждый кортеж которого проверяется на пригодность функцией match. Если функция match вернула значение True, то кортеж добавляется во множество подходящих кортежей temp_set, иначе игнорируется.

Функция match(temp, t) проверяет, соответствует ли кортеж t шаблону temp: сначала происходит сравнение длин шаблона и кортежа, в случае их совпадения сравниваются определённые в шаблоне поля с соответствующими полями проверяемого кортежа. Если длины кортежей оказались разными или какое-то определённое поле шаблона не совпало с соответствующим полем кортежа, то возвращается значение False, говорящее о том, что кортеж не подходит под шаблон. Иначе возвращается значение True. Программная реализация данной функции представлена в [16].

Если переданный в функцию rdp сокет оказался не равен значению None, то полученное множество temp_set отправляется инфраструктуре в качестве ответа. Иначе оно возвращается в качестве выходного параметра. Такая архитектура необходима для реализации операции $in(\bar{t})$, которая будет рассмотрена в следующем разделе.

2.8. Операция чтения in

Операция деструктивного чтения in имеет более сложную реализацию, поскольку один кортеж не может быть удалён из пространства двумя различными вызовами in. Этот факт подразумевает использование критических секций для процессов, пытающихся удалить один и тот же кортеж, что, во-первых, обеспечит невозможность удаления кортежа двумя процессами одновременно, во-вторых, позволит всем процессам получить доступ к ресурсу в некоторое время.

Помимо критических секций необходимо использовать алгоритм достижения консенсуса [7], чтобы в результате исполнения операции in из пространства был удалён правильный кортеж. Ярким представителем таких алгоритмов является алгоритм Paxos, предложенный Лесли Лампортом [8].

Paxos — это алгоритм решения задачи консенсуса в сети ненадёжных вычислителей. Компоненты распределённой системы можно разделить на три группы [7]:

- Заявитель (Proposer) — выдвигает «предложения» (какие-то значения), которые либо принимаются, либо отвергаются в результате работы алгоритма.

- Акцептор (Acceptor) — принимает или отвергает «предложение» Заявителя, согласует своё решение с остальными Акцепторами, уведомляет о своём решении Узнающих. Если Акцепторами было принято какое-либо значение, предложенное Заявителем, то оно называется утверждённым.

- Ученик (Learner) — запоминает решение Акцепторов, принятое в результате работы алгоритма консенсуса.

Компоненты распределённой системы могут принадлежать сразу нескольким группам, описанным выше, и вести себя и как Заявитель, и как Акцептор, и как Ученик. Такое распределение ролей в системе гарантирует следующее [8]:

- Только предложенное Заявителем значение может быть утверждено Акцепторами.
- Акцепторами утверждается только одно значение из всех предложенных Заявителями значений (возможно, противоречивых).
- Ученик не сможет узнать об утверждении какого-либо значения вплоть до того момента, пока оно действительно не будет утверждено.

Распределим роли среди компонент системы BTS следующим образом:

- Множество Заявителей состоит только из координационной инфраструктуры. Такой набор гарантирует наличие корректного Заявителя.
- Множество Акцепторов составляют все серверы системы.
- Множество Учеников включает в себя все серверы системы и инфраструктуру.

Рассмотрим реализацию операции $in(\bar{t})$ на стороне инфраструктуры, представленную методом `inp(self, pid, temp)` (см. листинг 2.19). Функция `inp` принимает два параметра: иденти-

Листинг 2.19. Модуль `BTS_infrastructure`, метод `inp`

```
1  def inp(self, pid, temp):
2    while True:
3      t, ts = self.enter_r(pid, temp)
4      logging.info('INP: enter with pid ' + str(pid) + ', temp ' +
          str(temp) + ', t: ' + str(t))
5      if t is None:
6        self.exit_r(pid, temp)
7        logging.info('INP: exit from cs with pid ' + str(pid))
8        return None
9      d = self.paxos(pid, temp, t, ts)
10     if d == t:
11       break
12   logging.info('INP: pid ' + str(pid) + ' deleted ' + str(t))
13   return t
```

фикатор клиентского процесса и шаблон кортежа. В первую очередь инфраструктура пытается войти в критическую секцию системы для удаления подходящего под шаблон `temp` кортежа.

Функция входа в критическую секцию `enter_r(self, pid, temp)` совмещена с операцией недеструктивного чтения $rd(\bar{t})$. Это необходимо для того, чтобы уменьшить количество обращений к системе и, как следствие, ускорить исполнение операции in. Сначала всем серверам системы отправляется запроса на вход в критическую секцию, от каждого сервера ожидается ответ. В качестве ответа возвращается сообщение вида `{'resp': 'go', 'ts': ts}`, где `ts` — это множество кортежей, соответствующих шаблону `temp`. Далее с помощью метода `rdp`, описанного в предыдущем разделе, из полученных множеств `ts` выбирается конкретный кортеж для удаления. Программная реализация метода `enter_r` представлена в [16].

Таким образом, по окончании выполнения функции `enter_r` все серверы уже разрешили войти в свои критические секции, а инфраструктурой уже был выбран кортеж t для удаления.

Если не удалось выбрать подходящий кортеж t, то вызывается метод `exit_r(self, pid, temp)`, который посылает серверам запрос на выход из критической секции без ожидания ответа. После этого метод `inp` возвращает значение `None`, свидетельствующее о том, что подходящего для удаления кортежа нет в пространстве.

Если же выбрать кортеж t всё-таки получилось, то необходимо выяснить, можно ли удалить этот кортеж из пространства. Для этого вызывается метод `paxos(self, pid, temp, t, ts)`, в ходе исполнения которого серверам системы отправляется предложение об удаление кортежа t, после чего от каждого сервера приходит ответ о принятом решении. На основе ответов серверов средствами модуля `collections` [24] выбирается самый популярный ответ (см. листинг 2.20). Если данное значение утвердил хотя бы $f+1$ сервер, оно возвращается в качетсве выходного параметра, иначе возвращается `None`. Заметим, что в случае наличия f и менее неисправных сер-

Листинг 2.20. Модуль BTS_infrastructure, метод paxos

```
1 def paxos(self, pid, temp, t, ts):
2   .........
3   c = Counter(tup_list)
4   mc = c.most_common(1)[0]
5   if mc[1] > self.AMOUNT_OF_ENEMIES:
6     return mc[0]
7   return None
```

веров в качестве ответа от сервера может прийти либо кортеж t (значит он уже был удалён из пространства), либо объект None (удаления не произошло), поскольку ни один сервер не является Заявителем. Таким образом, метод paxos симулирует деятельность инфраструктуры как Заявителя и Ученика.

Данный результат иллюстрирует благоприятный и промежуточный исходы задачи византийских генералов. При благоприятном исходе из пространства кортежей удаляется кортеж, предложенный клиентом, то есть приказ главнокомандующего (клиента) был получен и изучен всеми генералами (серверами), после чего они единогласно приняли решение следовать приказу. При промежуточном исходе кортеж не удаляется, возвращается None, при этом сохраняется целостность пространства кортежей, поскольку на всех корректных серверах удаления не произошло. Иными словами, генералами (серверами) было принято единогласное решение не доверять приказу главнокомандующего (клиента) и отступить (не производить удаление), сохранив при этом армию (целостность пространства кортежей).

На момент выхода из метода paxos все серверы уже сделали выбор, какой именно кортеж удалять. Если функция paxos вернула значение, равное t, то операция in считается успешно завершённой и возвращается кортеж t. В противном случае необходимо произвести ту же последовательность действий, начиная со входа в критическую секцию. Данный процесс продолжается до того момента, пока серверами не будет одобрено предложенное инфраструктурой значе-

ние или пока в пространстве не окажется подходящих для удаления кортежей.

Заметим, что выход из критической секции происходит только в случае неудачного исхода операции удаления, а в остальных случаях этого не происходит, поскольку выход из критической секции на каждом сервере осуществляется локально, и при следующей попытке инфраструктуры войти в критическую секцию не возникает проблем. Такая реализация необходима для избежания блокировок.

Теперь рассмотрим реализацию операции in на стороне сервера. Исполнение операции in на стороне инфраструктуры начинается с попытки входа в критическую секцию, рассмотрим, что происходит на сервере в этот момент. При получении запроса на вход в критическую секцию вызывается функция `enter_r(sock, pid, temp)` (программная реализация приведена в [16]), выполняющая две операции: добавление запроса в очередь и конструирование множества кортежей, подходящих под шаблон `temp`.

Когда инфраструктура запрашивает доступ к критической секции, происходит захват ресурса — типа шаблона кортежа. Тип шаблона кортежа — это кортеж, поля которого равны значениям типов соответствующих полей шаблона. Если тип шаблона уже захвачен другим вызовом `enter_r`, то запрос добавляется в очередь, соответствующую данному ресурсу. Иначе происходит захват типа шаблона и исполняются инструкции, отвечающие за конструирование множества кортежей, соответствующих шаблону, с помощью функции `rdp`.

Далее инфраструктура посылает серверам запрос на удаление конкретного кортежа. При получении данного запроса вызывается функция `inp(sock, pid, temp, tup)` с аргументами: сокет, идентификатор клиентского процесса, шаблон кортежа и кортеж, предложенный для удаления (см. листинг 2.21).

Если клиентский процесс с данным индентификатором `pid` находится в критической секции, начинается исполнение алгорит-

Листинг 2.21. Модуль BTS_server, функция inp

```
1  def inp(sock, pid, temp, tup):
2    global TS, RS, QR, TYPES
3    logging.info('INP ' + str(pid) + ': propose ' + str(tup))
4    if QR[TYPES[temp]][0][1] == pid:
5      d = paxos(sock, temp, tup)
6      if d is not None:
7        if d not in TS:
8          RS.add(d)
9          logging.info('INP '+str(pid)+': add '+str(d)+' to RS')
10       try:
11         TS.remove(d)
12       except KeyError:
13         pass
14       else:
15         logging.info('INP: remove ' + str(d) + ' from TS')
16     exit_r(temp)
```

ма Paxos, реализованного в функции paxos(sock, temp, tup) (полный программный код приведён в [16]).

В реализации используется стандартный Paxos-алгоритм [25]. Принятие решения происходит в два этапа. На первом этапе все серверы получают от инфраструктуры кортеж, который необходимо удалить. Если данный кортеж есть в пространстве кортежей, то он считается утверждённым на первом этапе, иначе утверждается значение None. Далее все серверы обмениваются утверждёнными на первом этапе значениями, в итоге у каждого сервера имеется вектор утверждённых значений (см. рисунок 2). Это осуществляется при помощи функции accept1(p, temp, server_port, tup), которая помещает полученные от серверов значения в соответствующий контейнер (полный программный код приведён в [16]).

Далее следует второй этап: серверы снова обмениваются полученными векторами друг с другом, в итоге у каждого сервера имеется матрица утверждённых значений, как показано на рисунке 3. Данные манипуляции реализованы при помощи функции accept2(p, temp, server_port, tup_dict) (полный программный код при-

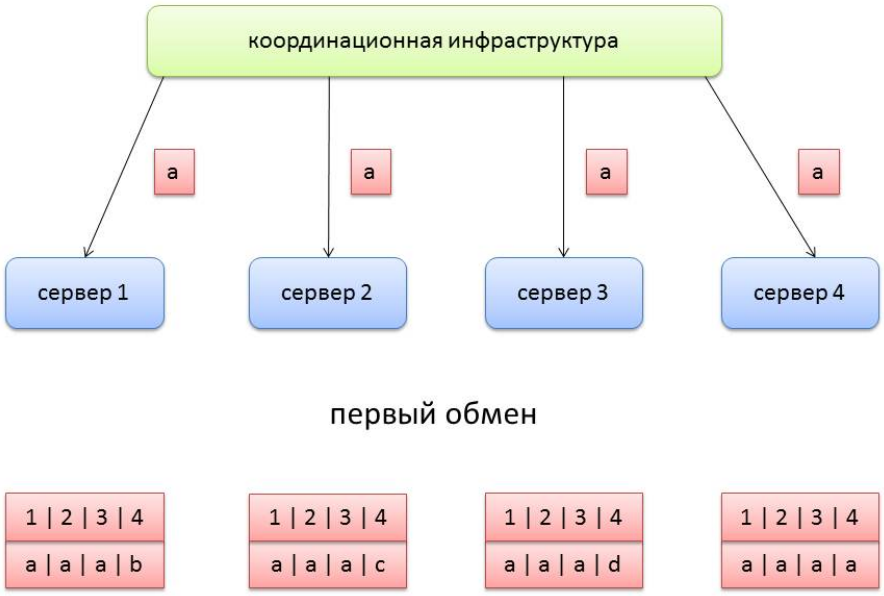

Рисунок 2 — Алгоритм Paxos (1 этап)

ведён в [16]). Далее в каждом столбце матрицы выбирается наиболее часто встречающееся значение, оно помещается в результирующий вектор. В результирующем векторе снова находится значение, которое встречается чаще остальных. Это значение является утверждённым на втором этапе. Таким образом Акцепторами принимается решение. Поскольку все сервера являются и Акцепторами, и Учениками, они уже знают, какое решение было принято, единственный Ученик, который не знает о принятом решении, это инфраструктура, поэтому ей отправляется сообщение о том, какой кортеж был удалён из пространства кортежей.

После завершения работы функции `paxos` возвращённое ею значение удаляется из пространства кортежей. Происходит локальное освобождение ресурса при помощи функции `exit_r(temp, pid=None)`, удаляющей соответствующий запрос из очереди (полный программный код приведён в [16]). На этом этапе завершается исполнение функции `inp`.

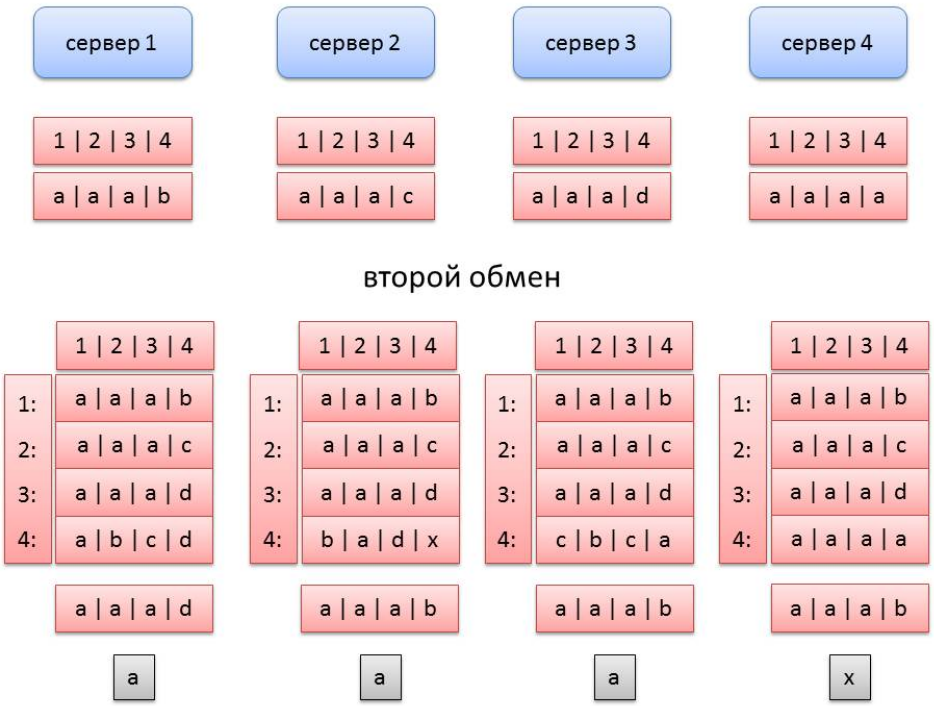

Рисунок 3 — Алгоритм Paxos (2 этап)

3. Полученные результаты

Для проверки корректности работы создадим два файла с кортежами вида:

- $(i, i+1, i+2, i+3, i+4), \quad i \in [0, 50)$ — для корректных серверов,
- $(i, i+1, i+2, i+3, i+4), \quad i \in [50, 100)$ — для неисправных серверов.

Для этого воспользуемся возможностями модуля `BTS_infrastructure`, описанного в разделе 2.4. Запустим систему со следующими параметрами, учитывая ограничения, описанные в разделе 2.1: $n = 5, \quad f = 1, \quad |Q_r| = 4, \quad |Q_w| = 5$, где f неисправных серверов входит и в кворум записи, и в кворум чтения.

Проверим корректность работы функций, реализующих операции out, rd и in, для этого будем просматривать лог-файлы серверов и инфраструктуры. Результаты одного из тестов представлены в таблице 1.

Таблица 1 — Корректность операций out, rd, in

Операция	Запрос	Ответ	Корректность
out	(0, 1, 2, 3, 4)	—	$+$
rd	(None, None, 6, None, 8)	(4, 5, 6, 7, 8)	$+$
in	(9, None, 11, None, 13)	(9, 10, 11, 12, 13)	$+$

Таким образом, при $n = 5, f = 1$ все операции исполняются корректно, что подтверждает отказоустойчивость системы.

Теперь запустим систему со следующими параметрами: $n = 5, \quad f = 3, \quad |Q_r| = 4, \quad |Q_w| = 5$. Ограничение для $f \leqslant \left\lfloor \frac{n-1}{3} \right\rfloor$ не удовлетворено.

Таблица 2 — Корректность операций out, rd, in

Операция	Запрос	Ответ	Корректность
out	(0, 1, 2, 3, 4)	—	$+$
rd	(1, None, None, None, 5)	None	$-$
in	(2, None, 4, None, 6)	None	$-$

Из таблицы 2 видно, что операция out исполняется корректно, а операции rd и in возвращают неправильный результат, поскольку на всех корректных серверах есть запрашиваемое значение. Таким образом, при $f > \left\lfloor \frac{n-1}{3} \right\rfloor$ система продолжает функционировать, но результаты её работы являются неверными. Данный пример демонстрирует отсутствие отказоустойчивости.

Рассмотрим зависимость количества корректно исполненных операций y (%) от числа неисправных серверов f, представленную на рисунке 4.

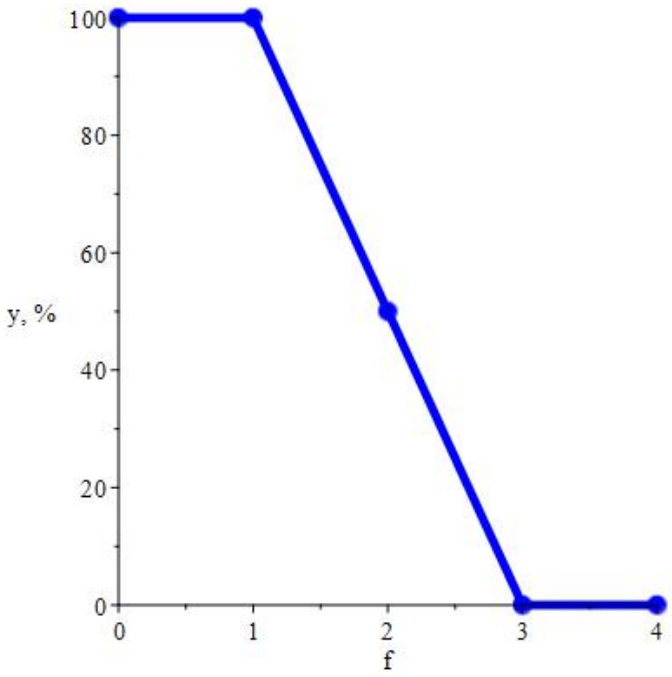

Рисунок 4 — График функции y(f)

Таким образом, отказоустойчивость наблюдается только при условии $f \leqslant \left\lfloor \frac{n-1}{3} \right\rfloor$.

Заключение

В ходе проделанной работы были изучены:

- принципы построения рапределённых систем,
- пространство кортежей как средство построения эффективных параллельных программ,
- техники, обеспечивающие отказоустойчивость рапределённых систем,
- задача византийских генералов и её применение в области распределённых отказоустойчивых систем,
- алгоритма консенсуса Paxos как решение задачи византийских генералов,

- кворум-системы как альтернатива реализации объектов надёжной разделённой памяти,
- возможности стандартной библиотеки языка программирования Python.

В результате проделанной работы была построена и реализована средствами языка Python распределённая отказоустойчивая система под названием «византийское пространство кортежей» (BTS — Byzantine Tuple Space), состоящая из клиентского интерфейса, координационной инфраструктуры и серверной части. Были описаны основные составляющие системы, отвечающие за отказоустойчивость, а также средства языка программирования Python, использованные для их реализации.

Реализованное распределённое отказоустойчивое пространство кортежей было протестировано, результаты тестирования подтвердили корректность исполнения операций манипулирования данными и продемонстрировали отказоустойчивость системы при необходимых условиях.

Таким образом, поставленная задача была полностью выполнена.

Список литературы

1. *Таненбаум Э.*, *Стеен М. ван*. Распределённые системы. Принципы и парадигмы. — СПб : Питер, 2003. — ISBN 5-272-00053-6.
2. *Косяков М. С.* Введение в распределенные вычисления. — 2014. — URL: `https://books.ifmo.ru/file/pdf/1551.pdf` (дата обр. 13.02.2018).
3. *Воеводин В. В.*, *Воеводин В. В.* Параллельные вычисления. — СПб : БХВ-Петербург, 2002. — ISBN 5-94157-160-7.

4. *Клеппман М.* Высоконагруженные приложения. Программирование, масштабирование, поддержка. — СПб : Питер, 2018. — ISBN 978-5-4461-0512-0.

5. Wikipedia, the free encyclopedia : Byzantine fault tolerance. — URL: https://en.wikipedia.org/wiki/Byzantine_fault_tolerance (дата обр. 16.04.2018).

6. *Bessani A. N.*, *Silva Fraga J. da*, *Lung L. C.* BTS: A Byzantine fault-tolerant tuple space // Proceedings of the 2006 ACM symposium on Applied computing. — Dijon, France : ACM, 2006. — С. 429—433. — ISBN 1-59593-108-2. — DOI: 10.1145/1141277.1141377.

7. Консенсус в распределенных системах. Paxos. — URL: https://habr.com/post/222825/ (дата обр. 12.03.2018).

8. *Lamport L.* Paxos Made Simple // ACM SIGACT News (Distributed Computing Column) : в 32 т. — ACM, 11.2001. — С. 51—58. — (4).

9. Wikipedia, the free encyclopedia : Tuple space. — URL: https://en.wikipedia.org/wiki/Tuple_space (дата обр. 16.04.2018).

10. *Ozsvald I.*, *Gorelick M.* High Performance Python. — O'Reilly Media, Inc., 2014. — ISBN 9781449361747.

11. The Python Standard Library. — URL: https://docs.python.org/3.5/library/index.html (дата обр. 20.01.2018).

12. Wikipedia, the free encyclopedia : Linda (coordination language). — URL: https://ru.wikipedia.org/wiki/Linda (дата обр. 16.11.2017).

13. Python 3.6.5 documentation : socket — Low-level networking interface. — URL: https://docs.python.org/3/library/socket.html (дата обр. 15.04.2018).

14. Python 3.6.5 documentation : pickle — Python object serialization. — URL: https://docs.python.org/3/library/pickle.html (дата обр. 15.04.2018).

15. Python 3.6.5 documentation : json — JSON encoder and decoder. — URL: https://docs.python.org/3/library/json.html?highlight=json (дата обр. 15.04.2018).

16. GitHub - verastrass/BTS: Byzantine Tuple Space. — URL: https://github.com/verastrass/BTS (дата обр. 01.06.2018).

17. Python 3.6.5 documentation : logging — Logging facility for Python. — URL: https://docs.python.org/3/library/logging.html (дата обр. 10.04.2018).

18. Python 3.6.5 documentation : subprocess — Subprocess management. — URL: https://docs.python.org/3/library/subprocess.html (дата обр. 01.03.2018).

19. Python 3.6.5 documentation : concurrent.futures — Launching parallel tasks. — URL: https://docs.python.org/3/library/concurrent.futures.html (дата обр. 04.03.2018).

20. Python 3.6.5 documentation : argparse — Parser for command-line options, arguments and sub-commands. — URL: https://docs.python.org/3/library/argparse.html (дата обр. 19.03.2018).

21. *Карпинский А.* Эффективная многопоточность в Python. — URL: https://habrahabr.ru/post/229767 (дата обр. 18.02.2018).

22. Python 3.6.5 documentation : itertools — Functions creating iterators for efficient looping. — URL: https://docs.python.org/3/library/itertools.html?highlight=itertools (дата обр. 20.03.2018).

23. Про Python - Справочник - sets (множества). — URL: http://pythonz.net/references/named/sets/ (дата обр. 19.03.2018).

24. Python 3.6.5 documentation : collections — Container datatypes. — URL: https://docs.python.org/3/library/collections.html (дата обр. 20.03.2018).

25. Wikipedia, the free encyclopedia : Paxos (computer science). — URL: https://en.wikipedia.org/wiki/Paxos_(computer_science) (дата обр. 12.03.2018).