МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное автономное образовательное
учреждение высшего образования
ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

Институт математики, механики и компьютерных наук
имени И. И. Воровича

Направление подготовки
01.03.02 — Прикладная математика
и информатика

РЕАЛИЗАЦИЯ РАСПРЕДЕЛЁННОГО ОТКАЗОУСТОЙЧИВОГО ПРОСТРАНСТВА КОРТЕЖЕЙ СРЕДСТВАМИ ЯЗЫКА PYTHON

Выпускная квалификационная работа
на степень бакалавра

Студентки 4 курса
В. М. Ложкиной

Научный руководитель:
старший преподаватель кафедры информатики и вычислительного
эксперимента В. Н. Брагилевский

Допущено к защите:
заведующий кафедры ИВЭ _____ В. С. Пилиди

Ростов-на-Дону
2018

Содержание

Введение

В современном мире распределённые системы являются основой различных объектов, например, веб-сервисов и пиринговых сетей. Поскольку 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, отказу отдельных компонент или системы в целом. Поэтому в распределённых системах поддержке безопасности уделяется особое внимание.

Для повышения надёжности системы случайные и умышленные сбои интерпретируются как Византийские ошибки (в терминах задачи о византийских генералах). В этом случае можно использовать соответствующие отказоустойчивые техники, которые смогут защитить систему и от случайных сбоев, и от умышленных вторжений.

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.

/здесь будет продолжение/

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 неизвестное 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 выклю-

чен 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 [3].

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 боагоприятного 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) [4].

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 $Linda$ [5], 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 [6].

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), описанные в языке $Linda$:

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

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 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.

Распределённая система не может полагаться на какой-то один конкретный узел, так как в случае его отказа восстановить систему будет невозможно [7]. 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.

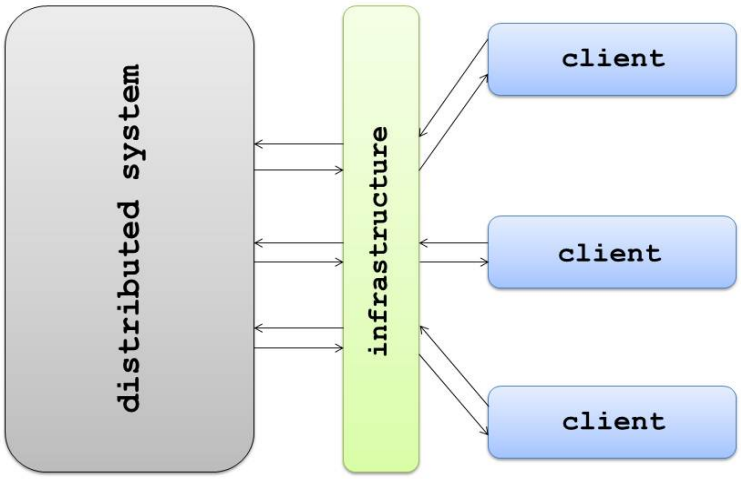

Рисунок 1 — Взаимодействие клиентов с распределённой системой

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 4 модулей:

- модуль $secondary\_functions$ включает в себя вспомогательные функции,
- модуль $client$ реализует клиентский интерфейс,
- модуль $infrastructure$ реализует деятельность координационной инфраструктуры,
- модуль $server$ реализует деятельность сервера $s \in U$.

Рассмотрим реализацию компонент более подробно.

2.2. Модуль secondary_functions

Модуль $secondary\_functions$ включает в себя функции сетевого взаимодействия ($get\_message$, $send\_message$, $connect\_to\_server$), а также функцию создания и записи пространства кортежей в файл ($create\_ts\_file$). Реализации данных функций представлены в листинге .1.

Функции сетевого взаимодействия используют в своей работе возможности модуля $socket$ из стандартной бибилиотеки $Python$ **??**. Функция $connect\_to\_server$ принимает в качестве входного параметра порт, к которому необходимо подключиться. При удачном подключении возвращается сокет, который можно использовать для обмена сообщениямив дальнейшей работе, иначе возвращается объект $None$, сигнализирующий о неудаче.

Функция $send\_message$ используются для отправки данных по сети, 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 $pickle$ **??**, отвечающая за сериализацию объектов. После завершения операции записи в сокет функция $send\_message$ завершает свою работу.

Функция $get\_message$ используются для получения данных из сети, она имеет единственный входной параметр — сокет, из которого осуществляется чтение информации. Если полученное сообщение оказалось пустым, то возвращается 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. Данная 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 записи в файл используется функция $dump$ из модуля $json$ **??**.

2.3. Модуль client

Интерфейс пользователя для взаимодействия с BTS реализован с помощью класса $Client$ из модуля $client$ (см. листинги .2 и .3 в приложении А). 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.

Далее для исполнения операций чтения/записи нужно вызвать соответствующую функцию: $out\_op(t)$, $rd\_op(\bar{t})$ или $in\_op(\bar{t})$. 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 формирует запрос в виде словаря $\{'op' : \text{код операции}, 'tup' : \text{кортеж}\}$ или $\{'op' : \text{код операции}, 'temp' : \text{шаблон кортежа}\}$, соответствующий

её семантике. Сформированный запрос отправляется координационной инфраструктуре. Для завершения операции записи не требуется ответ системы, а операции чтения ожидают, когда инфраструктура вернёт запрошенное значение.

Для корректного завершения работы системы в классе $Client$ реализована функция $stop\_op()$, которая отправляет инфраструктуре запрос о прекращении работы и ожидает ответа об успешном результате операции. По смыслу данная функция не должна являться частью клиентского интерфейса, она была включена в класс $Client$ для удобства.

2.4. Модуль BTS_infrastructure

Теперь опишем протоколы чтения/записи из языка $Linda$, но сделаем операции чтения неблокирующими.

2.5. Операция записи out

Вызов функции $out(t)$ добавляет кортеж t в пространство. На стороне клиента эта операция реализуется с помощью алгоритма 1:

Алгоритм 1. Операция out

```
    {client p}
 1: procedure out(t)
 2:     for all s ∈ Q_w do
 3:         send(s, ⟨ OUT, t⟩)
 4:     end for
 5: end procedure
```

При вызове данной функции клиенту p не нужно ждать ответа от серверов. Будем считать, что операция завершится в тот момент, когда все корректные серверы из кворума записи получат кортеж t.

На стороне сервера операция out реализуется с помощью алгоритма 2:

Алгоритм 2. Операция out

```
    {server s}
 1: upon receive(p, ⟨ OUT, t⟩)
 2:     if t ∉ R_s then
 3:         T_s ← T_s ∪ {t}
 4:     end if
 5:     R_s ← R_s \ {t}
 6: end upon
```

При получении запроса $\langle$ OUT, $t\rangle$ от клиента p сервер s добавляет кортеж t в пространство кортежей только в том случае, если этот кортеж ранее не был из него удалён. Это необходимо для того, чтобы один и тот же кортеж не был удалён дважды.

2.6. Операция чтения rdp

Недеструктивная операция чтения rdp в качестве входного параментра принимает шаблон кортежа $\bar{t}$, возвращает копию соответствующего шаблону кортежа из пространства. На стороне клиента эта операция реализуется с помощью алгоритма 3:

Алгоритм 3. Операция rdp

```
    {client p}
 1: function rdp(t̄)
 2:     T[s_1, ..., s_n] ← (⊥, ..., ⊥)
 3:     for all s ∈ U do
 4:         send(s, ⟨ RD, t̄⟩)
 5:     end for
 6:     repeat
 7:         wait receive(s, ⟨ TS, T_s^t̄⟩)
 8:         T[s] ← T_s^t̄
 9:     until {s ∈ U : T[s] ≠⊥} ∈ Q_r
10:     if ∃t : (|s : t ∈ T[s]| ⩾ f + 1) then
11:         return t
12:     end if
13:     return ⊥
14: end function
```

Клиент p взаимодействует с кворумом серверов чтения, в качестве ответа на запрос он получает от каждого сервера список подходящих под шаблон кортежей (серверная часть реализуется с помощью алгоритма 4), затем клиент p выбирает из них общий кортеж t, который располагается как минимум на $f + 1$ сервере. Если такого кортежа нет, то возвращается спецсимвол $\perp$, обозначающий, что операция не увенчалась успехом. Заметим, что операция rdp не является блокирующей в отличие от её аналога rd в языке $Linda$.

Алгоритм 4. Операция rdp

```
    {server s}
1:  upon receive(p, ⟨ RD, t̄⟩)
2:      T_s^t̄ ← {t ∈ T_s : m(t, t̄)}
3:      send(p, ⟨ TS, T_s^t̄⟩)
4:  end upon
```

2.7. Операция чтения inp

Операция inp является операцией деструктивного чтения, она имеет более сложную реализацию, поскольку один кортеж не может быть удалён из пространства двумя различными вызовами inp. 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. Операции входа в критическую секцию на стороне клиента реализуется с помощью функции $enter(\bar{t})$, описанной в алгоритме 5:

Алгоритм 5. Операция enter

```
    {client p}
 1: function enter(t̄)
 2:     T[s_1, ..., s_|Q_r|] ← (⊥, ..., ⊥)
 3:     for all s ∈ U do
 4:         send(s, ⟨ ENTER, p, t̄⟩)
 5:         T[s] ← T_s^t̄
 6:     end for
 7:     for all s ∈ Q_r do
 8:         wait receive(s, ⟨ GO, p, T_s^t̄⟩)
 9:         T[s] ← T_s^t̄
10:     end for
11:     if ∃t : (|s : t ∈ T[s]| ⩾ f + 1) then
12:         return t
13:     end if
14:     return ⊥
15: end function
```

Операция $enter(\bar{t})$ имеет входной параметр — шаблон кортежа, который требуется удалить в результате исполнения операции inp. Входной параметр нужен для того, чтобы совместить операции входа в критическую секцию и операцию чтения rdp, что позволит уменьшить количество этапов коммуникации на единицу. Сначала клиент p отправляет всем серверам запрос на разрешение войти в критическую секцию, затем ожидает, когда все серверы из кворума чтения дадут положительный ответ на отправленный запрос. При отправке разрешения войти в критическую секцию сервер s также отправляет список подходящих для удаления кортежей (то есть соответствующих шаблону $\bar{t}$). Далее, как и в реализации операции rdp, из всевозможных вариантов, присланных серверами, выбирается и возвращается общий кортеж t, располагающийся хотя бы на $f+1$ сервере, иначе возвращается спецсимвол $\perp$.

Операция выхода из критической секции $exit(\bar{t})$ на стороне клиента имеет более простую структуру: клиент p отсылает всем серверам сообщение о выходе из критической секции, не дожидаясь какого-

либо ответа от них. Реализация данной операции представлена в алгоритме 6:

Алгоритм 6. Операция exit

```
    {client p}
 1: procedure exit(t̄)
 2:     for all s ∈ U do
 3:         wait send(s, ⟨ EXIT, p, t̄⟩)
 4:     end for
 5: end procedure
```

Рассмотрим операции входа в критическую секцию и выхода из неё на стороне сервера. /здесь будет псевдокод и описание соответствующих алгоритмов/

Теперь рассмотрим реализацию операции inp (см. алгоритм 7). В первую очередь клиент p пытается войти в критическую секцию для удаления подходящего под шаблон $\bar{t}$ кортежа. По окончании выполнения функции $enter(\bar{t})$ все серверы из кворума чтения уже разрешили войти в критическую секцию, а клиент p уже выбрал конкретный кортеж t для удаления.

Алгоритм 7. Операция inp

```
    {client p}
 1: function inp(t̄)
 2:     repeat
 3:         t ← enter(t̄)
 4:         if t =⊥ then
 5:             exit(t̄)
 6:             return ⊥
 7:         end if
 8:         d ← paxos(p, P, A, L, t)
 9:         exit(t̄)
10:     until d = t
11:     return t
12: end function
```

Если выбранный кортеж t оказался равен спецсимволу $\perp$, значит в пространстве кортежей нет ни одного подходящего под шаблон $\bar{t}$ кортежа, в этом случае клиент p выходит из критической секции, а в качестве результата возвращается спецсимвол $\perp$, сигнализирующий о неудачном завершении операции inp.

Если выбранный кортеж t не равен спецсимволу $\perp$, то клиент p предлагает серверам удалить именно его. Для того, чтобы удалить кортеж t, соответствующий шаблону $\bar{t}$, из пространства, серверы из кворума чтения должны принять решение, можно ли удалить кортеж t. Для этой цели вызывается функция $paxos$, являющаяся реализацией алгоритма достижения консенсуса в распределённых системах. Данная функция имеет пять параметров:

1. Процесс p, предлагающий значение. В нашем случае это клиентский процесс, вошедший в критическую секцию.

2. Множество Заявителей $P = \{p, s\_1, ..., s\_f + 1\}$, состоящее из клиентского процесса p и $f+1$ сервера. Такой набор гарантирует наличие хотя бы одного корректного заявителя.

3. Множество Акцепторов $A = U$. Все серверы являются Акцепторами.

4. Множество Узнающих $L = \{p\} \cup U$. О принятом решении узнают все серверы и клиентский процесс.

5. Предлагаемое значение t.

В качестве выходного параметра $paxos$ возвращает кортеж d, который было принято удалить из пространства. Далее происходит выход из критической секции. Если полученное значение d совпало с предлагаемым t, то операция inp завершается и возвращает в качестве выходного параметра удалённый кортеж t. Если $d \neq t$, то все вышеописанные выкладки проделываются заново до тех пор, пока не выполнится условие $d = t$.

На момент выхода из функции $paxos$ все Акцепторы уже приняли решение, удалять кортеж t или нет, все Узнающие получили сообщения о решении Акцепторов. Если функция $paxos$ вернула кортеж, то он уже был удалён из пространства кортежей, иначе возвращается спецсимвол $\perp$, сигнализирующий о том, что операция удаления кортежа t не увенчалась успехом. 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 спецсимвол $\perp$, 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).

/здесь будет описание серверной части операции inp/

2.8. Блокирующие операция rd и in

Кроме того, операции чтения являются блокирующими, то есть если в пространстве кортежей на данный момент нет ни одного кортежа, соответствующего шаблону, то процесс, пославший запрос, заблокируется до того момента, пока в пространстве не появится подходящего кортежа.

Блокирующие операции rd и in, соответствующие спецификации языка $Linda$, могут быть реализованы на стороне клиента путём повторного вызова их неблокирующих аналогов rdp и inp до тех пор, пока необходимый кортеж не будет получен.

Заключение

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. Wikipedia, the free encyclopedia : Byzantine fault tolerance. — URL: https://en.wikipedia.org/wiki/Byzantine_fault_tolerance (дата обр. 16.04.2018).
4. *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. — С. 429—433. — ISBN 1-59593-108-2. — DOI: 10.1145/1141277.1141377.
5. Wikipedia, the free encyclopedia : Linda (coordination language). — URL: https://ru.wikipedia.org/wiki/Linda (дата обр. 16.11.2016).
6. Wikipedia, the free encyclopedia : Tuple space. — URL: https://en.wikipedia.org/wiki/Tuple_space (дата обр. 16.04.2018).
7. *Клеппман М.* Высоконагруженные приложения. Программирование, масштабирование, поддержка. — СПб : Питер, 2018. — ISBN 978-5-4461-0512-0.

Приложение А

Листинг .1. Модуль *secondary_functions*

```
1  from socket import socket, AF_INET, SOCK_STREAM, error
2  import _pickle as pickle
3  import json
4
5  def get_message(sock):
6    sock.settimeout(50)
7    msg = b""
8    tmp = b""
9    while True:
10         try:
11           tmp = sock.recv(1024)
12         except error:
13           break
14         if len(tmp) < 1024:
15           break
16         msg += tmp
17   msg += tmp
18   if len(msg) > 0:
19         return pickle.loads(msg)
20   else:
21         return  None
22
23 def send_message(sock, msg):
24   sock.send(pickle.dumps(msg))
25
26 def connect_to_server(port):
27   sock = socket(AF_INET, SOCK_STREAM)
28   i = 50
29   while i:
30         try:
31           sock.connect(('localhost', port))
32         except ConnectionRefusedError:
33           i -= 1
34         else:
35           return sock
36   return None
37
38 def create_ts_file(file_name, init_num, amount=500, length=100):
39   with open(file_name, 'w') as f:
40         list_of_tuples = list()
41         for i in range(init_num * amount, (init_num + 1) * amount
              ):
42           list_of_tuples.append(tuple(j for j in range(i, i +
               length)))
43         json.dump(list_of_tuples, f)
```

Листинг .2. Класс *Client*, модуль *client*

```
1  import logging
2  from secondary_functions import get_message, send_message,
       connect_to_server
3
4  class Client(object):
5
6    def __init__(self, port):
7          self.port = port
8          logging.basicConfig(filename='client_log.txt', level=
               logging.DEBUG, format="%(asctime)s - %(message)s")
9
10
11   def out_op(self, tup):
12         sock = connect_to_server(self.port)
13
14         if sock is not None:
15           send_message(sock, {'op': 'out', 'tup': tup})
16           sock.close()
17           logging.info('OUT: ' + str(tup))
18         else:
19           logging.info('OUT_OP: cannot connect to server')
20
21
22   def rd_op(self, temp):
23         sock = connect_to_server(self.port)
24
25         if sock is not None:
26           send_message(sock, {'op': 'rdp', 'temp': temp})
27           resp = get_message(sock)
28           sock.close()
29           logging.info('RD: ' + str(resp['resp']))
30
31           if resp is None:
32                 return resp
33           else:
34                 return resp['resp']
35         else:
36           logging.info('RD_OP: cannot connect to server')
37           return None
```

Листинг .3. Класс *Client*, модуль *client* (продолжение)

```
1   def in_op(self, temp):
2         sock = connect_to_server(self.port)
3
4         if sock is not None:
5           send_message(sock, {'op': 'inp', 'temp': temp})
6           resp = get_message(sock)
7           sock.close()
8           logging.info('In: ' + str(resp['resp']))
9
10          if resp is None:
11                return resp
12          else:
13                ret     urn resp['resp']
14
15        else:
16          logging.info('IN_OP: cannot connect to server')
17          return None
18
19  def stop_op(self):
20        sock = connect_to_server(self.port)
21
22        if sock is not None:
23          send_message(sock, {'op': 'stop'})
24          resp = get_message(sock)
25          sock.close()
26          logging.info('STOP_OP: ' + str(resp['resp']))
27
28          if resp is None:
29                return resp
30          else:
31                return resp['resp']
32
33        else:
34          logging.info('STOP_OP: cannot connect to server')
35          return None
```