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.

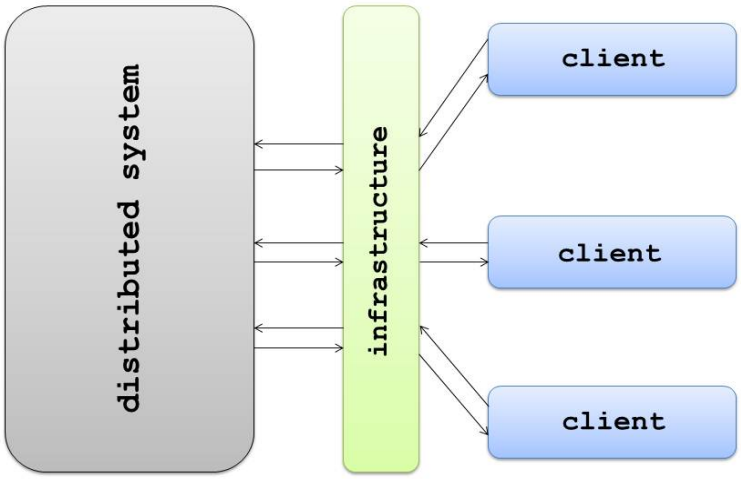


Рисунок 1 — Взаимодействие клиентов с распределённой системой

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 BTS_infrastructure реализует деятельность координационной инфраструктуры,
- модуль BTS_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 в приложении.

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 (полный программный код представлен в приложении). 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': код операции, 'tup': кортеж} или {'op': код операции, 'temp': шаблон кортежа}, соответствующий её семанти-

ке. 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.

Ввиду возможности возникновения ошибок для контроля исполнения операций используется модуль `logging` **??** из стандартной библиотеки языка `Python`, отвечающий за логирование. Создание и настройка конфигурации лога происходит в конструкторе класса `Client` при помощи функции `basicConfig`, устанавливающей имя файла для записи лога, формат выводимых сообщений и уровень логирования. Всего существует пять уровней логирования, в реализации использован уровень `INFO` для вывода информационных сообщений.

## 2.4. Модуль BTS_infrastructure

Координационная инфраструктура системы `BTS` реализована в модуле `BTS_infrastructure` с помощью класса `BTS_infrastructure` (полный программный код представлен в приложении). 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 (см. листинг **??**) 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 множеством кортежей для корректных серверов,
- имя файла, в котором хранится множеством 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 `self.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 **??**). Если изначально количество неисправных серверов 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 **??**).

Метод `start_servers` имеет три входных параметра:

- два индекса, определяющих 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` **??** 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 кворума, которому принадлежит сервер.

Информация об исполненных операциях записывается в лог инфраструктуры.

Вернёмся к методу `run`. После запуска серверов создаётся и настраивается сокет, с помощью которого далее будет происходить коммуникация с клиентскими процессами. Будем обрабатывать клиентские запросы при помощи пула потоков `ThreadPoolExecutor` из модуля `concurrent.futures` **??** стандартной библиотеки `Python`, принимающего в качестве аргумента количество потоков, используемых пулом потоков.

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 добавляется в очередь для исполнения пулом потоков с помощью метода `submit`, принимающего 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 потоков, ожидается завершение всех исполняющихся или ожидающих в очереди задач. После завершения работы пула потоков вызывается метод stop_servers (см. листинг **??**), при исполнении которого всем запущенным серверам посылается запрос о прекращении работы. Информация о проделанных операциях записывается в лог инфраструктуры. На этом работа метода run завершается, возвращается значение True, свидетельствующее об успешном завершении работы координационной инфраструктуры.

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

Рассмотрим метод worker (см. листинг **??**) более подробно. Он выполняет функцию получения и обработки запроса клиента и 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 клиентский запрос был послан уникальным клиентом ( согласно системной модели BTS множество клиентских процессов бесконечно), тогда в качестве идентификатора клиентского процесса можно использовать номер запроса. Это осуществляется при помощи свойства AMOUNT_OF_CLIENTS.

Работу метода self.worker можно разделить на два этапа: получение сообщения от клиента и обработка полученного запроса. Для того, чтобы получить сообщение с запросом от клиента, исполь-

зуется 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 $out$, $rdp$ или $inp$ вызываются методы `out`, `rdp` или `inp`, которые будут рассмотрены позже в разделах **??**, **??** и **??** соответственно.

При получении запроса на исполнение операции $stop$ флаг `THREAD_POOL_ON` устанавливается в значение `False`, после чего прекращается обработка клиентских запросов, система готовится к завершению работы.

## 2.5. Модуль BTS_server

Серверная часть системы `BTS` реализована в модуле `BTS_server` (полный программный код представлен в приложении), 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 кворума, которому принадлежит сервер.

Для обработки аргументов командной строки используется библиотека `argparse` **??**. 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 функ-

ции `read_from_ts_file` (см. листинг **??**). На сервере кортежи хранятся в контейнере `set`, что позволяет исключить наличие идентичных кортежей и сделать проверку на вхождение быстрой, поскольку `set` в качестве базовой структуры данных использует хэш-таблицу **??**.

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

Обработка входящих запросов осуществляется при помощи функции `worker`, программный код которой можно посмотреть в приложении в листинге **??**. 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 связаны с реализацией операции $in$ и будут рассмотрены позже в разделах **??**. Последняя команда `stop`

устанавливает флаг `THREAD_POOL_ON` в значение `False`, что приводит к прекращению работы пула потоков, а затем - к завершению работы сервера. Информация о произведённых действиях записывается в лог.

## 2.6. Операция записи out

Операция $out(t)$ добавляет кортеж $t$ в пространство кортежей. На стороне инфраструктуры эта операция реализуется с помощью функции `out(self, t)`, программный код которой приведён в листинге **??** в приложении.

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 $t$.

На стороне сервера операция $out(t)$ реализуется с помощью функции `out(t)` (программный код приведён в листинге **??** в приложении). При получении запроса на добавление 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 соответствующего шаблону кортежа, не извлекая его из пространства. На стороне инфраструктуры эта операция реализуется с помо-

щью функции `rdp(self, temp, ts=None)`, имеющей два параметра: шаблон кортежа и объект `ts` по умолчанию равный `None`, значение которого будет объяснено позже. При получении запроса на исполнение операции $rd$ данная функция вызывается с единственным первым параметром. Программный код приведён в листинге **??** в приложении.

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 **??**), 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 $Linda$.

## 2.8. 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 1:

**Алгоритм 1.** 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 2:

**Алгоритм 2.** 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 3). 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 3.** 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/

## 2.9. 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

**Листинг .1.** Модуль secondary_functions

```
1  from socket import socket, AF_INET, SOCK_STREAM, error
2  import 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
3         sock = connect_to_server(self.port)
4
5         if sock is not None:
6           send_message(sock, {'op': 'inp', 'temp': temp})
7           resp = get_message(sock)
8           sock.close()
9           logging.info('In: ' + str(resp['resp']))
10
11          if resp is None:
12                return resp
13          else:
14                ret     urn resp['resp']
15
16        else:
17          logging.info('IN_OP: cannot connect to server')
18
19          return None
20
21  def stop_op(self):
22
23        sock = connect_to_server(self.port)
24
25        if sock is not None:
26          send_message(sock, {'op': 'stop'})
27          resp = get_message(sock)
28          sock.close()
29          logging.info('STOP_OP: ' + str(resp['resp']))
30
31          if resp is None:
32                return resp
33          else:
34                return resp['resp']
35
36        else:
37          logging.info('STOP_OP: cannot connect to server')
38
39          return None
```

**Листинг .4.** Класс BTS_infrastructure, модуль BTS_infrastructure

```
1  from secondary_functions import *
2  from subprocess import Popen, CalledProcessError, PIPE
3  from concurrent.futures import ThreadPoolExecutor
4  from itertools import combinations
5  from collections import Counter
6  import logging
7
8  class BTS_infrastructure(object):
9    def __init__(self, port, servers, amount_of_enemies, size_of_QW
       , size_of_QR, right_file, wrong_file):
10     self.INFRASTRUCTURE_PORT = port
11     self.U = servers
12     self.AMOUNT_OF_SERVERS = len(servers)
13     self.AMOUNT_OF_ENEMIES = amount_of_enemies
14     self.TS_FILE, self.WRONG_TS_FILE = right_file, wrong_file
15     self.THREAD_POOL_ON = True
16     self.AMOUNT_OF_CLIENTS = 0
17     self.QW, self.QR = self.U[0:size_of_QW], self.U[-size_of_QR:]
18     logging.basicConfig(filename='infrastructure_log.txt', level=
         logging.DEBUG, format="%(asctime)s - %(message)s")
19     logging.info('Infrastructure on port ' + str(self.
         INFRASTRUCTURE_PORT))
20
21   def start_servers(self, ind1, ind2, file):
22     for i in range(ind1, ind2):
23       if U[i] in self.QR:
24         quorum = self.QR
25       else:
26         quorum = [U[i]]
27
28     try:
29       Popen('python BTS_server.py '+str(i)+' '+str(self.U[i])+' '
           +str(file)+' '+''.join(str(j)+' ' for j in quorum),
           stdout=PIPE, stderr=PIPE, shell=True)
30     except CalledProcessError:
31       logging.info("START_SERVERS: cannot start server "+str(self
           .U[i]))
32     else:
33       logging.info("START_SERVERS: start server "+str(self.U[i]))
34
35   def stop_servers(self):
36     for i in self.U:
37       s = connect_to_server(i)
38       if s is not None:
39         send_message(s, {'op': 'stop'})
40         s.close()
41         logging.info("STOP_SERVERS: stop server "+str(i))
42       else:
43         logging.info("STOP_SERVERS: cannot stop server "+str(i))
```

**Листинг .5.** Класс BTS_infrastructure, модуль BTS_infrastructure (продолжение)

```
1   def worker(self, client, pid):
2     req = get_message(client)
3     try:
4       if req['op'] == 'out':
5         self.out(req['tup'])
6       elif req['op'] == 'rdp':
7         send_message(client, {'resp':self.rdp(req['temp'])})
8       elif req['op'] == 'inp':
9         send_message(client, {'resp':self.inp(pid,req['temp'])})
10      elif req['op'] == 'stop':
11        self.THREAD_POOL_ON = False
12        send_message(client, {'resp': 'ok'})
13      else:
14        logging.info('WORKER: wrong request ' + str(req))
15    except KeyError:
16      logging.info('WORKER: wrong request ' + str(req))
17    client.close()
18
19  def run(self):
20    if self.AMOUNT_OF_ENEMIES >= self.AMOUNT_OF_SERVERS:
21      return False
22
23    self.start_servers(0,self.AMOUNT_OF_ENEMIES,self.
        WRONG_TS_FILE)
24    self.start_servers(self.AMOUNT_OF_ENEMIES,self.
        AMOUNT_OF_SERVERS,self.TS_FILE)
25
26    s = socket(AF_INET, SOCK_STREAM)
27    s.bind(('', self.INFRASTRUCTURE_PORT))
28    s.listen(1)
29
30    with ThreadPoolExecutor(4) as pool:
31      while self.THREAD_POOL_ON:
32        try:
33          s.settimeout(30)
34          client_s, client_addr = s.accept()
35          self.AMOUNT_OF_CLIENTS += 1
36        except error:
37          pass
38        else:
39          pool.submit(self.worker, client_s, self.
              AMOUNT_OF_CLIENTS)
40
41    self.stop_servers()
42    s.close()
43    logging.info('RUN: stop working')
44    return True
```

**Листинг .6.** Класс BTS_infrastructure, модуль BTS_infrastructure (продолжение)

```
  def enter_r(self, pid, temp):
    ts = {i: [None, None] for i in self.U}  # port: (socket, set)

    for i in self.U:
      ts[i][0] = connect_to_server(i)
      if ts[i][0] is not None:
        send_message(ts[i][0], {'op': 'enter', 'pid': pid, 'temp'
           : temp})
        logging.info('ENTER_R: send ' + str(temp) + ' to server '
            + str(i))
      else:
        if i in self.QR:
          return tuple([None, None])
        else:
          ts.pop(i)

    qr = set()

    for i in ts.keys():
      resp = get_message(ts[i][0])
      ts[i][0].close()

      if resp is None or resp['resp'] != 'go':
        logging.info('ENTER_R: wrong respond from server ' + str(
           i))
        ts[i][1] = None
      else:
        ts[i][1] = resp['ts']
        qr.add(i)

    if not set(self.QR).issubset(qr):
      return tuple([None, None])

    return tuple([self.rdp(temp, ts), ts])



  def exit_r(self, pid):
    for i in self.U:
      sock = connect_to_server(i)

      if sock is not None:
        send_message(sock, {'op': 'exit', 'pid': pid})
        sock.close()
      else:
        logging.info('EXIT_R: cannot exit cs with pid: ' + str(
           pid))
```

**Листинг .7.** Класс BTS_infrastructure, модуль BTS_infrastructure (продолжение)

```
1   def out(self, t):
2   for i in self.QW:
3     sock = connect_to_server(i)
4     if sock is not None:
5       send_message(sock, {'op': 'out', 'tup': t})
6       logging.info('OUT: send ' + str(t) + ' to server ' + str(i)
          )
7       sock.close()
8     else:
9       logging.info('OUT: cannot connect to server ' + str(i))
10
11
12  def inp(self, pid, temp):
13  while True:
14    t, ts = self.enter_r(pid, temp)
15    logging.info('INP: enter with pid ' + str(pid) + ', temp ' +
        str(temp) + ', t: ' + str(t))
16    if t is None:
17      self.exit_r(pid)
18      logging.info('INP: exit from cs with pid ' + str(pid))
19      return None
20
21    d = self.paxos(pid, t, ts)
22    self.exit_r(pid)
23    logging.info('INP: exit from cs with pid ' + str(pid))
24    if d == t:
25      break
26
27    return t
28
29
30  def paxos(self, pid, t, ts):
31    for i in ts.keys():
32    ts[i][0] = connect_to_server(i)
33    if ts[i][0] is not None:
34      send_message(ts[i][0], {'op': 'in', 'pid': pid, 'tup': t})
35
36    tup_list = []
37    for i in ts.keys():
38      resp = get_message(ts[i][0])
39      ts[i][0].close()
40      if resp is not None:
41        tup_list.append(resp['resp'])
42      else:
43        tup_list.append(None)
44
45    c = Counter(tup_list)
46    return c.most_common(1)[0][0]
```

**Листинг .8.** Класс BTS_infrastructure, модуль BTS_infrastructure (продолжение)

```
1   def rdp(self, temp, ts=None):
2     if ts is None:
3       ts = {i: [None, None] for i in self.U}  # port: (socket,
           set)
4       for i in self.U:
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
12      qr = set()
13      for i in ts.keys():
14        msg = get_message(ts[i][0])
15        ts[i][0].close()
16        if msg is not None:
17          ts[i][1] = msg['ts']
18          qr.add(i)
19        else:
20          ts[i][1] = None
21
22        logging.info('RDP: receive ' + str(ts[i][1]) + ' from
             server ' + str(i))
23        if not set(self.QR).issubset(qr):
24          return None
25
26    t = None
27    for i in combinations(ts.keys(), self.AMOUNT_OF_ENEMIES + 1):
28      ts_intersection = ts[i[0]][1]
29
30      for j in i:
31        if ts[j][1] is None:
32          ts_intersection = None
33          break
34
35        ts_intersection = ts_intersection.intersection(ts[j][1])
36        if len(ts_intersection) == 0:
37          break
38
39      if ts_intersection is None:
40        continue
41
42      if len(ts_intersection) > 0:
43        t = ts_intersection.pop()
44        break
45
46    return t
```

**Листинг .9.** Модуль BTS_server

```
from secondary_functions import *
import argparse
from concurrent.futures import ThreadPoolExecutor
from collections import deque
from collections import Counter
import logging

TS = set()
RS = set()
QR = deque()

SERVER_ID = 0
SERVER_PORT = 0
SERVERS = []

DECISIONS1 = {}
AMOUNT_OF_DECISIONS1 = 0

DECISIONS2 = {}
AMOUNT_OF_DECISIONS2 = 0

THREAD_POOL_ON = True
IS_ACCEPTED1 = False
IS_ACCEPTED2 = False



def read_from_ts_file(file_name):
  global TS

  with open(file_name, 'r') as f:
    data = json.load(f)
    for i in data:
      TS.add(tuple(i))


def match(temp, tup):
  if len(temp) != len(tup):
    return False

  for i, j in zip(temp, tup):
    if i is None:
      continue
    elif i != j:
      return False

  return True
```

**Листинг .10.** Модуль BTS_server (продолжение)

```
def worker(client):
  global THREAD_POOL_ON

  req = get_message(client)

  if req['op'] == 'out':
    out(req['tup'])
  elif req['op'] == 'rd':
    rdp(client, req['temp'])
  elif req['op'] == 'in':
    inp(client, req['pid'], req['tup'])

  elif req['op'] == 'enter':
    enter_r(client, req['pid'], req['temp'])
  elif req['op'] == 'exit':
    exit_r(req['pid'])

  elif req['op'] == 'accept1':
    accept1(req['pid'], req['port'], req['tup'])
  elif req['op'] == 'accept2':
    accept2(req['pid'], req['port'], req['tup_dict'])

  elif req['op'] == 'stop':
    logging.info('WORKER: stop-message received')
    THREAD_POOL_ON = False

  else:
    logging.info('WORKER: wrong request ' + str(req))

  client.close()


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')
```

**Листинг .11.** Модуль BTS_server (продолжение)

```
1  def rdp(sock, temp):
2    global TS
3    temp_set = set()
4
5    for t in TS:
6      if match(temp, t):
7        temp_set.add(t)
8
9    logging.info('RDP: temp_set for ' + str(temp) + ' : ' + str(
       temp_set))
10   if sock is not None:
11     send_message(sock, {'ts': temp_set})
12   else:
13     return temp_set
14
15 def inp(sock, pid, tup):
16   global TS, RS, QR
17   if QR[0][1] == pid:
18     d = paxos(sock, tup)
19     if d is not None:
20       if d not in TS:
21         RS.add(d)
22         logging.info('INP ' + str(pid) + ': add ' + str(d) + ' to
             RS')
23       TS.remove(d)
24
25 def enter_r(sock, pid, temp):
26   global QR
27   QR.append((sock, pid, temp))
28   if QR[0][1] == pid:
29     logging.info('ENTER_R: ' + str(pid) + ' enter cs')
30     temp_set = rdp(None, temp)
31     send_message(sock, {'resp': 'go', 'ts': temp_set})
```

**Листинг .12.** Модуль BTS_server (продолжение)

```
parser = argparse.ArgumentParser()
parser.add_argument('id', type=int)
parser.add_argument('port', type=int)
parser.add_argument('TSFile', type=str)
parser.add_argument('quorum', nargs='+', type=int)
args = parser.parse_args()
SERVER_ID, SERVER_PORT, ts_file, SERVERS = args.id, args.port,
   args.TSFile, args.quorum

logging.basicConfig(filename=str(SERVER_ID) + 'log.txt', level=
   logging.DEBUG, format="%(asctime)s - %(message)s")
logging.info('Server ' + str(SERVER_ID) + ' on port ' + str(
   SERVER_PORT))
read_from_ts_file(ts_file)
logging.info('Read from file: ' + str(ts_file))
s = socket(AF_INET, SOCK_STREAM)
s.bind(('', SERVER_PORT))
s.listen(1)

with ThreadPoolExecutor(10) as pool:
  while THREAD_POOL_ON:
    try:
      s.settimeout(30)
      client_s, client_addr = s.accept()
    except error:
      pass
    else:
      pool.submit(worker, client_s)

s.close()
logging.info('RUN: stop working')
```