Сервер обработки сообщений E-message

Оглавление

Бизнес-аналитика

Границы проекта

- Отсутствует обработка дополнительных вложений в сообщение;
- Состав участников диалога утверждается один раз. После создания конференции из неё можно будет только выйти;
- Не предоставляется готовый клиент;
- Предоставляемые соединения не защищаются — данные передаются открытым текстом.

Глоссарий

Сообщение — пересылаемый пользователями текст с дополнительной информацией: время отправки, отправитель.

Диалог — именованная совокупность пользователей, общающихся друг с другом через сообщения.

Система из предметной области

1. Бизнес-процессы

1.1.Выход из диалога

Имя процесса	**Реакция системы**

Выход пользователя из диалога	• Состоит ли указанный пользователь в диалоге?
	  ○ Да, состоит:
	  ▪ Указанный участник является последним?
	  • Да:
	  ○ Удалить диалог.
	  • Нет:
	  ○ Исключить пользователя;
	  ○ Сохранить изменения.
	  ○ Нет, отсутствует:
	  ▪ Уведомить об ошибке

1.2.Отправка сообщения

Имя процесса	**Реакция системы**
Отправка сообщения пользователем	• Собрать сообщение, перевести в состояние **written;** • Сохранить сообщение; • Добавить сообщение в диалог; • Перевести сообщение в состояние **sent;** • Сохранить сообщение; • Отдать пользователю собранный дескриптор.

2. Модель предметной области

2.1.Диаграмма классов

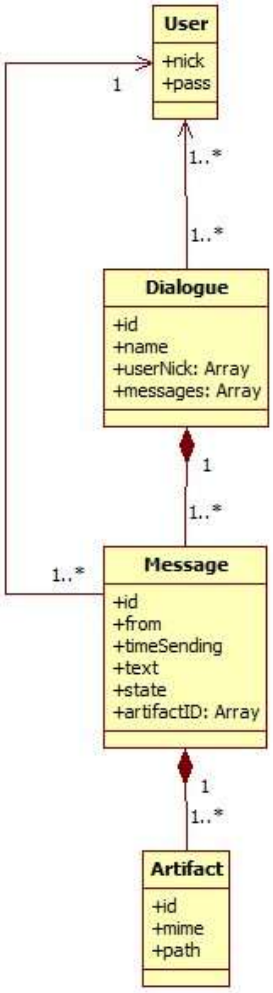

Рисунок 1 — диаграмма классов приложения

2.2.Диаграмма состояний

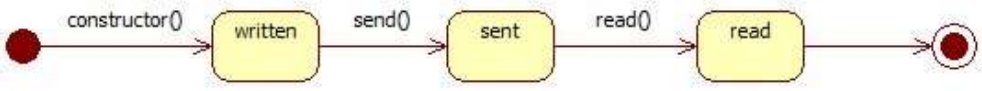

Рисунок 2 — диаграмма состояний сообщения

Системная аналитика

Прецеденты и сценарии

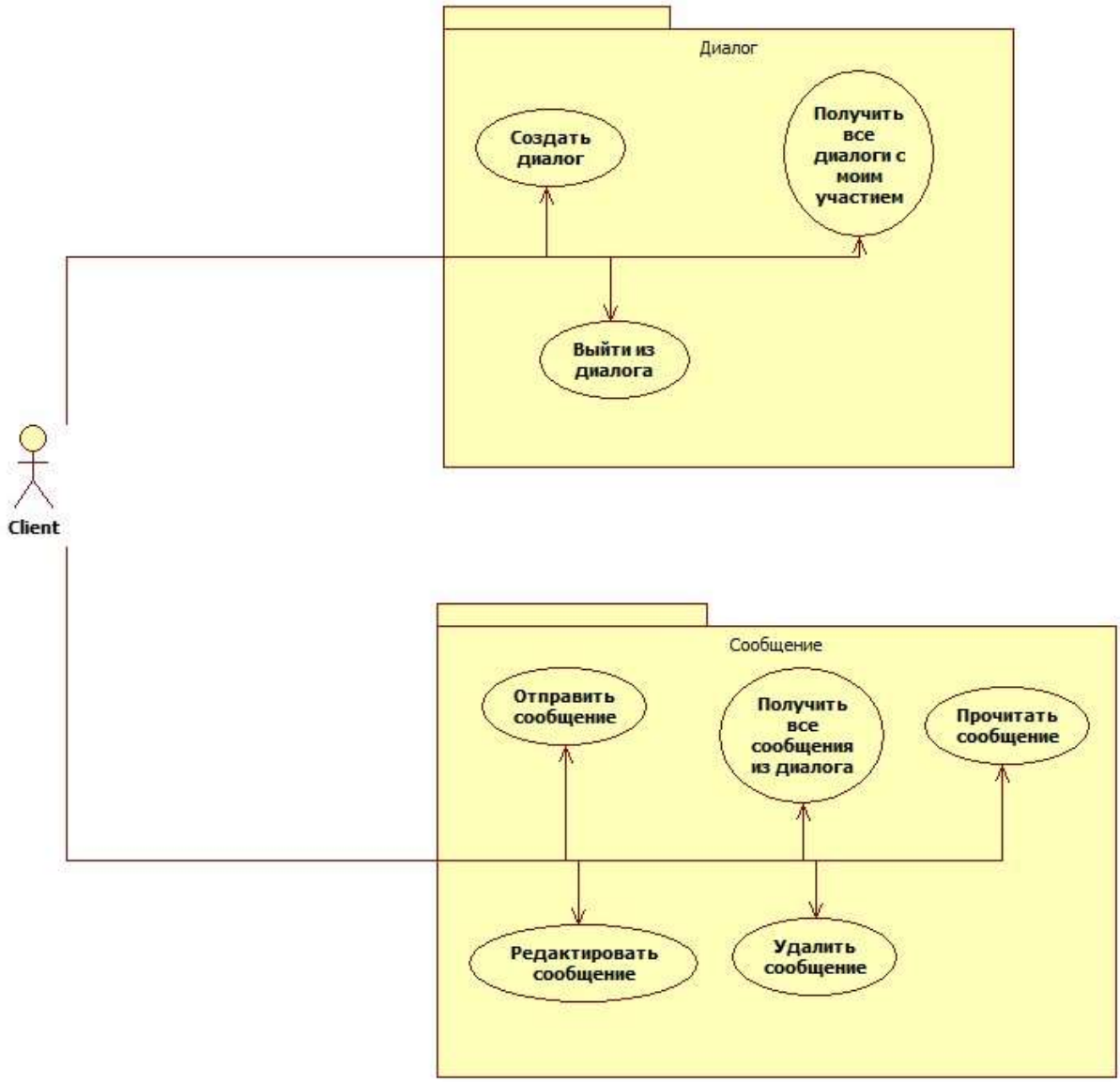

Рисунок 3 — диаграмма прецедентов приложения

Требования проекта

1. Атрибуты качества

Атрибуты.Сервер.Производительность.Многопоточность. Приложение должно эффективно распараллеливать обработку пользовательских запросов.

Атрибуты.Сервер.Производительность.Доступ к модели. Приложение должно кэшировать состояние модели в in memory database.

2. Функциональные требования

Функции.Сервер.Обслуживание_Соединений. Приложение должно слушать указанный в конфигурации порт, при этом иметь множество процессов, обслуживающих запросы клиентов.

Функция.Сервер.Процессы-обработчики. Количество процессов, обрабатывающих входящие запросы задаётся перед запуском сервера из конфигурационного файла.

Диалог.Удаление. Клиенты имеют право только выйти из диалога. Если в диалоге не осталось участников, то он автоматически удаляется со всеми содержащимися сообщениями.

Соообщение.Удаление. При удалении сообщения автоматически уничтожаются его артефакты при наличии.

Архитектура

Рисунок 4 — схематическое представление структуры системы

Система состоит из следующих компонентов:

- Клиентское приложение — отправляет по TCP-соединению текстовые запросы по формату, указанному в разделе **Интерфейс**. Разрыв соединения инициируется клиентом.
- Серверное приложение — получает запросы клиентов, обрабатывает их, отправляет результат обратно;
- СУБД — занимается сохранением изменений, внесённых в формальную модель предметной области.

Интерфейс системы

Интерфейс приложения построен на передаче текстовой информации от клиента по TCP-соединению. Структура любого запроса:

Строка запроса
Пустая строка (разделитель)
Тело запроса

где:

- Строка запроса – строка с именем функции API. Синтаксически идентична атому Erlang;
- Пустая строка – нужна для разделения первой и последней частей. Используется UNIX-стиль разделения (\n);
- Тело запроса — часть, содержащая входные данные функции. Передаётся в формате JSON.
    - Так как разработан stateless-сервер, то практически каждый метод требует авторизации инициатора действия. Для этого передаются ник пользователя и его пароль.

1. Методы User

1.1.create_user

1.1.1. Описание

Метод позволяет сгенерировать нового пользователя в системе. Не требует аутентификации и авторизации.

1.1.2. Тело запроса

```
{
    "nick":"some_user",
    "pass":"some_pass"
}
```

1.1.3. Выход

При успешном исполнении:

```
ok
```

При возникновении ошибки:

```
{
    "type":"error",
    "msg":"some_reason"
}
```

2. Методы Dialogue

2.1.create_dialogue

2.1.1. Описание

Выполняет генерацию нового диалога с указанным перечнем участников.

2.1.2. Тело запроса

```
{
    "nick":"creator name",
    "pass":"creator_pass",
    "name":"name of dialogue",
    "userNicks":["creator",”user2”]
}
```

2.1.3. Выход

При успешном исполнении возвращается сгенерированный объект диалога:

```
{
    "id":1,
    "name":"name_of_dialogue",
    "users":["creator",”user2”],
    "messages":null
}
```

При возникновении ошибки:

```
{
    "type":"error",
    "msg":"some reason"
}
```

2.2.get_dialogues

2.2.1. Описание

Выполняет поиск всех диалогов, в которых состоит инициатор запроса.

2.2.2. Тело запроса

```
{
    "nick":"some_user",
    "pass":"some_pass"
}
```

2.2.3. Выход

При успешном исполнении возвращается массив дескрипторов диалога:

```
{
  "arr": [
    {
      "id": 1,
      "name": "name_of_dialogue",
      "users": [
        "creator",
        "user2"
      ],
      "messages": null
    },
    {
      "id": 2,
      "name": "name_of_dialogue",
      "users": [
        "creator",
        "user5"
      ],
      "messages": null
    }
  ]
}
```


При возникновении ошибки:


```
{
    "type":"error",
    "msg":"some_reason"
}
```


2.3.quit_dialogue

2.3.1. Описание

Осуществляет выход пользователя из диалога. Если диалог станет пустым, то выполнится автоматическое удаление.

2.3.2. Тело запроса


```
{
  "nick": "some_user",
  "pass": "some_pass",
  "id": 8
}
```


2.3.3. Выход

При успешном исполнении может быть два варианта:

- Пользователь был последним в диалоге, тогда система удалит его полностью и вернёт:

```
ok
```

- Пользователь не являлся единственным участником, тогда система вернёт обновлённую сущность диалога:

```
{
  "id": 1,
  "name": "name of dialogue",
  "users": [
    "creator",
  ],
  "messages": null
}
```

При возникновении ошибки:

```
{
    "type":"error",
    "msg":"some_reason"
}
```

3. Методы Message

3.1.send_message

3.1.1. Описание

Преобразует данные сообщения в сущность системы, добавляет сформированный дескриптор в модель, создаёт ссылку в указанном диалоге на новое сообщение.

3.1.2. Тело запроса

```
{
    "nick":"some user",
    "pass":"some_pass",
    "text":"Test 123",
    "artifactID":null,
    "dialogueID":1
}
```

3.1.3. Выход

При успешной отработке функция возвращает дескриптор построенного сообщения:

```
{
	"id": 8,
	"from": "some user",
	"timeSending": 1659251253428,
	"text": "Test 123",
	"state": "written",
	"artifactID": null
}
```

При возникновении ошибки:

```
{
    "type":"error",
    "msg":"some_reason"
}
```

3.2.get_message

3.2.1. Описание

Возвращает одно сообщение по его идентификатору.

3.2.2. Тело запроса

```
{
  "nick": "some user",
  "pass": "some_pass",
  "id": 8
}
```

3.2.3. Выход

При успешной отработке функция возвращает дескриптор сообщения:

```
{
	"id": 8,
	"from": "some_user",
	"timeSending": 1659251253428,
	"text": "Test 123",
	"state": "written",
	"artifactID": null
}
```

При возникновении ошибки:

```
{
    "type":"error",
    "msg":"some reason"
}
```

3.3.get_messages

3.3.1. Описание

Возвращает все сообщения из указанного диалога.

3.3.2. Тело запроса

```
{
  "nick": "some_user",
  "pass": "some_pass",
  "id": 8
}
```

3.3.3. Выход

При успешной отработке сервер возвращает массив дескрипторов сообщений:

```
{
  "arr": [
    {
      "id": 8,
      "from": "vasya",
      "timeSending": 1659251253428,
      "text": "Test 123",
      "state": "written",
      "artifactID": null
    },
    {
      "id": 9,
      "from": "vasya",
      "timeSending": 1659251484494,
      "text": "Test 123",
      "state": "written",
      "artifactID": null
    }
  ]
}
```

При возникновении ошибки:

```
{
    "type":"error",
    "msg":"some_reason"
}
```

3.4.read_message

3.4.1. Описание

Помечает указанное сообщение как прочитанное.

3.4.2. Тело запроса

```
{
  "nick": "some_user",
  "pass": "some pass",
  "id": 8
}
```

3.4.3. Выход

При успешном выполнении возвращается обновлённый дескриптор сообщения:

```
{
      "id": 8,
      "from": "some_user",
      "timeSending": 1659251253428,
      "text": "Test 123",
      "state": "read",
      "artifactID": null
}
```

При возникновении ошибки:

```
{
    "type":"error",
    "msg":"some reason"
}
```

3.5.change_text

3.5.1. Описание

Позволяет редактировать текст указанного сообщения.

3.5.2. Тело запроса

```
{
  "nick": "some_user",
  "pass": "some_pass",
  "id": 8,
  "text":"Napisal normal'noe soobsheniye."
}
```

3.5.3. Выход

При успешном выполнении возвращается обновлённый дескриптор сообщения:

```
{
      "id": 8,
      "from": "some_user",
      "timeSending": 1659251253428,
      "text": "Napisal normal'noe soobsheniye.",
      "state": "read",
      "artifactID": null
}
```

При возникновении ошибки:

```
{
    "type":"error",
    "msg":"some_reason"
}
```

3.6.delete_message

3.6.1. Описание

Удаляет из системы указанное сообщение. Действие допустимо только для автора.

3.6.2. Тело запроса

```
{
  "nick": "some_user",
  "pass": "some_pass",
  "messageID": 8,
  "dialogueID": 1
}
```

3.6.3. Выход

При успешном исполнении:

```
ok
```

При возникновении ошибки:

```
{
    "type":"error",
    "msg":"some_reason"
}
```

Проектирование Серверного приложения

1. Архитектура

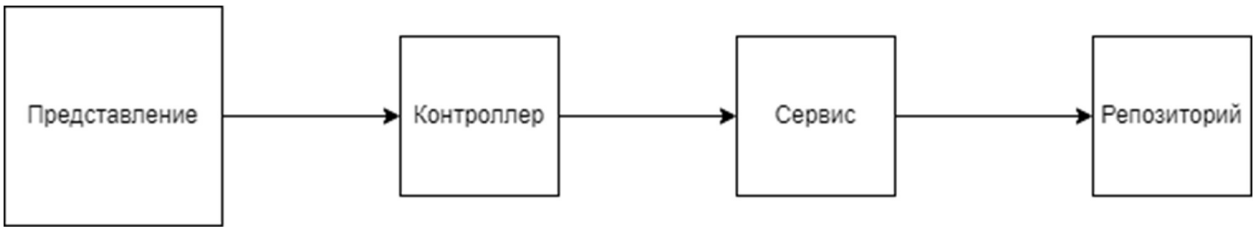

Рисунок 5 — архитектура кода серверного приложения

Приложение состоит из компонентов:

- Представление — слой, отвечающий за общение с клиентами, сериализацию сущностей модели в человекочитаемое представление;
- Контроллер — предоставляет интерфейс воздействия на модель слою представления. Отвечает за сложную логику обработки сущностей — каждая функция может вызвать одну или несколько команд слоя сервисов как для декларированной, так и для других сущностей.
- Сервис — модули данного слоя предоставляют перечень атомарных и не связанных друг с другом функций, направленных на изменение состояния одной декларированной сущности;
- Репозиторий — функции данного слоя занимаются чтением/записью сущностей модели в постоянную память и кэш.

Предоставляемый слоями интерфейс не зависит от его внутренней реализации. Ниже прикреплены диаграммы, демонстрирующие принцип распараллеливания приложения:

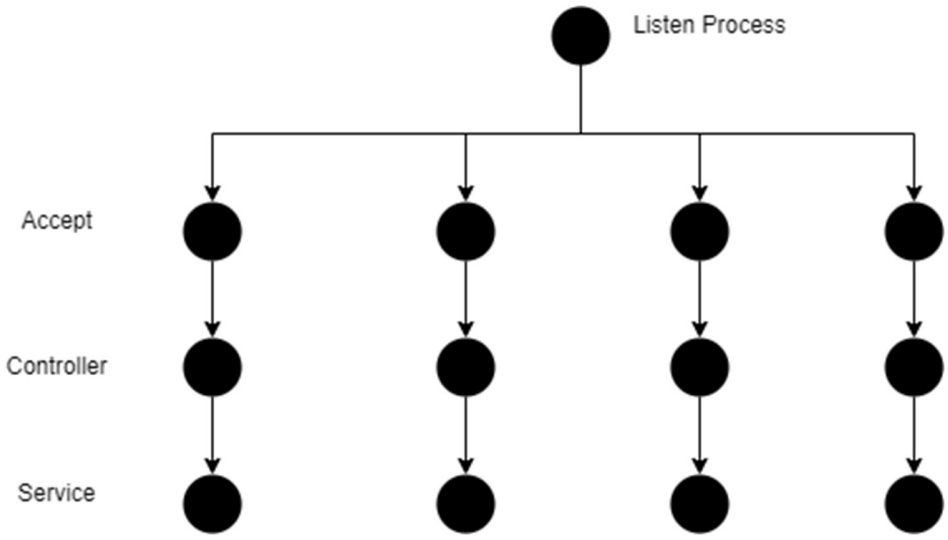

Рисунок 6 — граф управляющих и информационных связей приложения. Показывает расщепление алгоритма работы программы на крупные блоки.

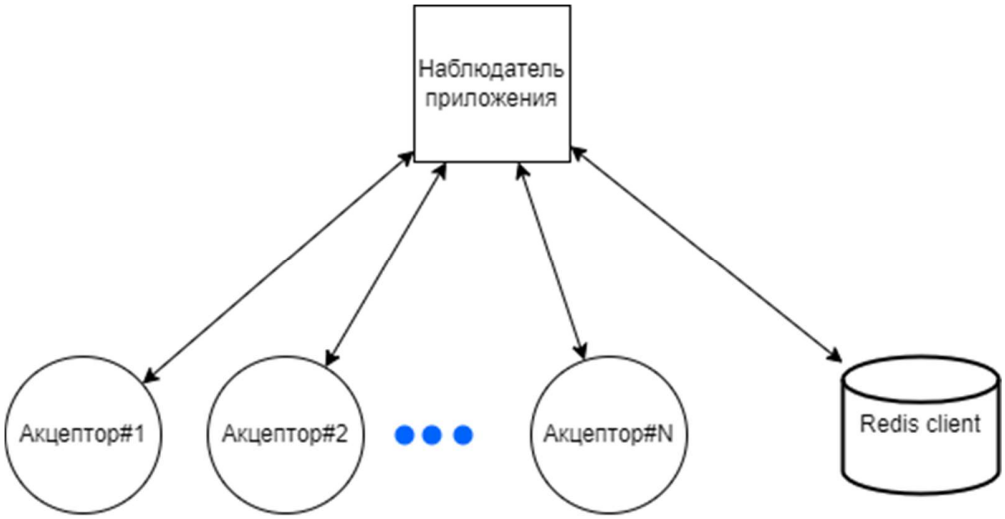

Рисунок 7 — схематическое представление процессов системы. Демонстрирует Erlang-процессы из которых состоит приложение.

Типы процессов:

1. Наблюдатель приложения — занимается инициализацией дочерних процессов, прослушиванием входящих TCP-соединений, распределением их обработки между другими акцепторами;
2. Акцептор — процесс, обслуживающий клиентское соединение. Обслуживание происходит по принципу: «получить запрос, сформировать ответ, отправить ответ». Соединения закрываются по инициативе клиента;
3. Redis client — сторонний процесс Erlang, работа с которым осуществляется акцепторами по паттерну клиент-сервер. Через него происходят все операции с СУБД Redis.

2. Модель данных предметной области

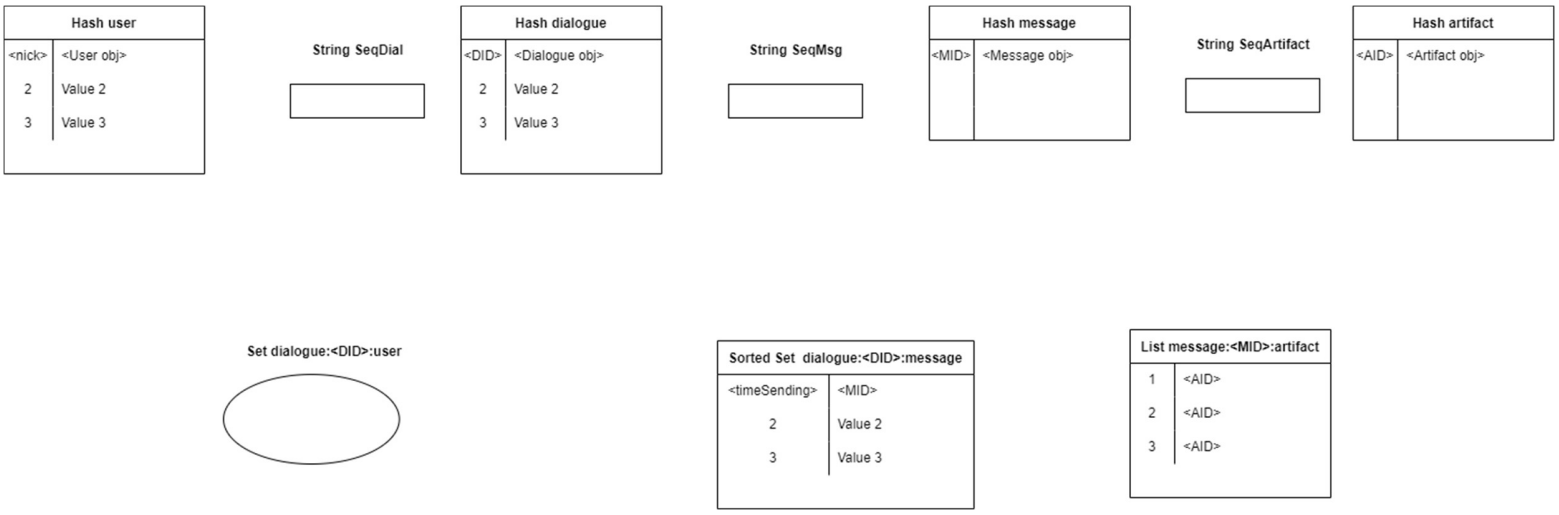

Рисунок 8 — модель данных приложения в системе Redis

Строки с ключами формата Seq<Entity> содержат счётчик, значения которого берутся для формирования нового идентификатора конкретной сущности.

Для хранения сообщений диалога была выбрана структура Sorted Set так как она позволяет извлекать сообщения в порядке их получения сервером без дополнительных затрат на сортировку.

Для связи сущностей Диалог и Пользователь было выбрано множество пользователей для каждого диалога. Структура «Set» удобна, так как позволяет проверять наличие пользователя в диалоге без лишних итераций. Такое решение обеспечивает экономию памяти, но взамен:

- Операция поиска всех диалогов для одного пользователя будет иметь сложность O(n) (n - число всех диалогов), так как нужно перебирать список пользователей для каждого диалога;
- Сложность поиска всех участников диалога будет соответствовать O(n1) (n1 – мощность множества пользователей диалога). В данном случае n1 будет в разы меньше n.

Прочие доступные решения:

- Сделать множество диалогов для каждого пользователя:
  - Поиск всех диалогов пользователя = O(n), где n – мощность множества диалогов;
  - Поиск всех участников диалога = O(n1), где n1 – количество всех пользователей.
- Сделать два множества: множество участников диалога, множество диалогов пользователя:

- Поиск всех диалогов пользователя = O(n), где n – мощность множества диалогов пользователя;
- Поиск всех участников диалога = O(n1), где n1 – мощность множества участников одного диалога.
- Возникает необходимость синхронизации обоих коллекций;
- Возрастают затраты памяти на поддержку состояния модели.

3. Разработка алгоритмов

3.1. Алгоритм сериализации массива записей

При разработке возникла потребность преобразования списка записей Erlang в JSON-массив. Имеющийся инструментарий мог заниматься сериализацией списков примитивных данных.

Была написана функция с хвостовой рекурсией. Алгоритм:

- Рекурсивный обход исходной последовательности:
  - Сколько элементов в списке:
    - Несколько:
      - Расщепить на голову и хвост;
      - Сформировать список результат по принципу: к имеющемуся списку добавить новую голову — голова исходного массива + ”,”.
    - Один:
      - Преобразовать последний элемент, добавить его в качестве головы результата;
      - Выполнить reverse.
- Добавить в начало и конец спец. символы для JSON.

Ниже прикреплена реализация алгоритма:

```
%%Конвертирует массив Erlang-записей в JSON-массив объектов
%%Data - массив записей
%%Encode - callback-функция, конвертирующая каждую запись из списка
%%Encode(type:atom, Rec), где: type - имя записи, Rec - запись
encodeRecordArray(Data,Encode)->
  JSON_List=encodeRecordArray(Data,[],Encode),
  ["{\"arr:\"[" | [JSON_List|"]}"]].

encodeRecordArray([H],Res,Encode)->
  JSON=Encode(H),
  lists:reverse([JSON|Res]);
encodeRecordArray([H|T],Res, Encode)->
  JSON= Encode(H),
  Put=[JSON|","],
  encodeRecordArray(T,[Put|Res], Encode).
```

Так как написанная функция является функцией высшего порядка за счёт получения на вход callback-функции, занимающейся преобразованием каждой записи входной последовательности, то итоговая реализация является обобщённой.

4. Кодирование и отладка

4.1.Процессы

4.1.1. Наблюдатель приложения

Наблюдатель предоставляет внешней Erlang среде следующий интерфейс:

1. Старт:
    a. Описание:
    b. Тело запроса:
        i. Функция
        `e_message:start(Path)`, где:
        `Path` - путь до файла config.json;
        ii. Функция
        `e_message:start()`.
        Путь по умолчанию "priv/etc/config.json".
    c. Выход:
        i. Функция возвращает атом ok.
        Если во время инициализации приложения произойдёт критическая ошибка, то сервер отправит наверх сигналы выхода:
            1. {‘EXIT’, PID, acceptors_start_fail} — ни один из акцепторов не запустился;
            2. {‘EXIT’, PID, repo_start_fail} — клиент БД не запустился.
2. Стоп
    a. Описание.
        i. Наблюдатель сначала начнёт остановку каждого акцептора, если он не ответит в течение 5 секунд, то будет остановлен принудительно.
        После отключается клиент Redis.
    b. Тело запроса:
        i. Кортеж Erlang:
        `{stop,From}`,
        где `From` - PID отправителя.
    c. Выход:
        i. Сообщение ok.

4.1.2. Акцептор

Акцептор кроме API системы предоставляет внутренний интерфейс:

- Стоп

a. Описание:
    i. Процесс постарается завершиться в момент, когда он не занят обслуживанием клиента.
b. Тело запроса:
    i. Сообщенине
    `{stop,From}`,
    где `From` - PID отправителя.
c. Выход:
    i. Сообщение ok.

Акцептор в процессе своей работы вызывает коды слоёв **представления**, **контроллеров**, **сервисов**, **репозиториев**.

4.2.Слои приложения

Так как каждое действие слоя сводится к CRUD-операции интерфейс был построен по следующим принципам:

- На вход принимаются:
    - Объект;
    - Операция чтения — ключи поиска.
- Выходы:
    - Create-операции должны возвращать 1 персистентный объект;
    - Read-операции - фильтры, поэтому они возвращают пустой или заполненный список.
      Если ничего не найдено, то пусть пользователь интерфейса решает является ли ситуация исключительной. Пустое множество результатов означает, что пользователь отправил плохие параметры поиска.
    - Update-операции - возвращают новый объект;
    - Delete – ok;

4.2.1. Слой представления

Слой представления состоит из блока receive, в котором для каждого метода API системы назначается нужная функция-обработчик. Код назначения обработчика:

```
%%обработка клиентских запросов
serve request(Socket, Request,Con)->
  [Fun,ArgsJSON]=parseRequest(Request),
  case Fun of
    create_user->
      create user handler(ArgsJSON,Socket, Con);
    create_dialogue->
      create_dialogue_handler(ArgsJSON,Socket, Con);
    get dialogues->
      get_dialogues_handler(ArgsJSON,Socket, Con);
    quit dialogue->
      quit_dialogue_handler(ArgsJSON,Socket, Con);
    send_message->
      send_message_handler(ArgsJSON,Socket, Con);
    get_message->
      get message handler(ArgsJSON,Socket, Con);
    get_messages->
      get_messages_handler(ArgsJSON,Socket, Con);
    read message->
      read_message_handler(ArgsJSON,Socket, Con);
    change text->
      change_text_handler(ArgsJSON,Socket, Con);
    delete_message->
      delete_message_handler(ArgsJSON,Socket, Con)
  end.
```

Пример функции обработчика:

```
get messages handler(ArgsJSON, Socket, Con)->
  Args = ?json_to_record(get_messages,ArgsJSON),
  #get_messages{nick = Nick, pass=Pass, id = DID}=Args,
  case is_authorised(Nick,Pass,Socket, Con) of
    true->
      D=dialogue_controller:get_dialogue(DID, Con),
      io:format("TRACE server:get_messages_handler/3 D:~p~n",[D]),
      case D of
        {error,_R}->
          handle_error(_R,Socket);
        D->
          Res = dialogue_controller:get_messages(D, Con),
          io:format("TRACE server:get messages handler/3 Messages:~p~n",[Res]),
          handle_request_result(
            Res,
            fun(Y)-> parse:encodeRecordArray(Y,fun(X)-
>?record_to_json(message,X) end) end,
            Socket)
      end;
    false->ok
  end.
```

4.2.2. Слой сервисов

В данном слое выполняется логика обработки сущностей, а также функции обобщённой обработки результатов транзакций репозитория.

Пример обобщённых функций, обрабатывающих результаты транзакций чтения:

```
%%Функция обработки read-операции репозитория.
%%Необходима, если с точки зрения бизнес-процесса результат:
%%    1) обязан быть найден
%%    2) должен быть единственным
extract_single_value(Transaction)->
  case Transaction of
    {error,_R}->{error,_R};
    []->{error,not_found};
    [Res| ]->Res
  end.

%%Функция обработки read-операции репозитория.
%%Необходима, если с точки зрения бизнес-процесса результат:
%%    1) обязан быть найден
%%    2) допускается несколько объектов
extract multiple values(Transaction)->
  case Transaction of
    {error, Reason}->{error, Reason};
    []->{error,not_found};
    Res->Res
  end.
```

Пример функции, обрабатывающий отправку сообщения в диалог:

```
%%Сохранить в БД сообщение
%%Добавить полученный ID в диалог
%%Сохранить диалог
%%Вовзращает персистентное сообщение
add_message(D,M)->
  Fun=
    fun()->
      M_Persisted = message_repo:write(M),
      D_Updated = dialogue:add_message(D,M_Persisted),
      dialogue_repo:update(D_Updated),
      message_repo:update(message:send(M_Persisted)),
      message_repo:read(M_Persisted#message.id)
    end,
  T = transaction:begin_transaction(Fun),
  service:extract single value(T).
```

4.2.3. Слой репозитория

Репозиторий – абстракция для получения данных. Слой предоставляет два интерфейса сервисам:

- Интерфейс CRUD-операций в БД для сущностей, являющихся метафорой на действия с коллекцией данных;
- Интерфейс транзакций для CRUD-операций.

Такой подход позволяет сервисам в рамках одной транзакции выполнять несколько запросов на чтение/запись к СУБД, что положительно сказывается на быстродействии и надёжности приложения.

Модули репозитория поделены на 2 части:

- Код конкретного репозитория к Redis — функции, получающие данные от клиента Redis;
- Код абстрактного репозитория — обёртка над конкретным репозиторием.

Пример абстрактного репозитория сообщений:

```
-module(message_repo).
-include("entity.hrl").
-export([read/2,
         write/2,
         update/2,
         delete/2]).

%%create-операции должны возвращать 1 персистентный объект
%%read-операции - фильтры, поэтому они возвращают пустой или заполненный список
%%update-операции - возвращают новый объект
%%delete - ok
%%ошибка - {error, Reason}

%%Если произошла ошибка соединения с источником -> значит, что вся система
становится бесполезной->
%%ошибка критическая и должна быть послана на самый верх

write(Message, Con)->
  redis_message:write(Con,Message).

read(ID, Con)->
  redis_message:read(Con,ID).

update(Message, Con)->
  redis_message:update(Con,Message).

%%Каскадно удаляются артефакты, так как вне сообщений они не имеют смысла
delete(#message{}=Message, Con)->
  redis_message:delete(Con,Message).
```

Пример конкретного репозитория сообщений

```
-module(redis_message).
-include("entity.hrl").
-include("jsonerl/jsonerl.hrl").
-export([write/2,
         read/2,
         update/2,
         delete/2]).

%% на вход всегда принимаются:
%%      дескриптор соединения, сущность
%%      дескриптор соединения, ключи поиска

%%create-операции должны возвращать 1 персистентный объект
%%read-операции - фильтры, поэтому они возвращают пустой или заполненный список
%%update-операции - возвращают новый объект
%%delete - ok

write(Con,#message{}=Message)->
  {ok,MID}=eredis:q(Con,["INCR","SeqMsg"]),
  Commited = Message#message{id = binary_to_integer(MID)},

{ok,_}=eredis:q(Con,["HSET",atom_to_list(message),MID,?record_to_json(message,Co
mmited)]),
  Commited.

read(Con,MID) when MID =/= -1 ->
  {ok,JSON} = eredis:q(Con,["HGET",atom_to_list(message),MID]),
  [?json_to_record(message,JSON)].

update(Con,#message{id = MID}=Message)->

{ok,_}=eredis:q(Con,["HSET",atom_to_list(message),MID,?record_to_json(message,Me
ssage)]),
  Message.

delete(Con,#message{id=MID})->
  {ok,_} = eredis:q(Con,["HDEL",atom_to_list(message),MID]),
  ok.
```

Пример абстрактного интерфейса транзакций:

```
-module(transaction).
-export([begin transaction/1,
        abort_transaction/0]).

%%Res || {error, Cause}

%%Fun/0 функция с телом транзакции.
%%Тело транзакции - набор действий языка среди которых есть операции,
предоставляемые репозиторием.
%%Каждая функция репозитория - обращение к базе
%%Следовательно, заворачивая множество функций репозитория в транзакции мы
заворачиваем и обращения к базе.
%%Транзакция исполняется если её не отменили.
begin_transaction(Fun)->
  try
    redis_transaction:begin_transaction(Fun)
  catch
    throw:transaction_aborted -> {error, transaction_aborted}
  end.

%%функция внутри должна послать сообщение базе об отмене транзакции и
%%выбросить исключение transaction_aborted
abort_transaction()->
  redis_transaction:abort_transaction().
```

Пример интерфейса транзакций Redis:

```
-module(redis_transaction).
-author("aleksandr_work").
-export([begin transaction/1,
        abort_transaction/0]).

begin_transaction(Fun)->
  case Fun() of
    {error, R}->{error, R};
    Res -> Res
  end.

abort_transaction()->
  throw(transaction_aborted).
```