УДК 004.415.2

РАЗРАБОТКА ОТКАЗОУСТОЙЧИВОЙ РАСПРЕДЕЛЕННОЙ МНОГОПОЛЬЗОВАТЕЛЬСКОЙ ИГРЫ

Пушкин Иван Александрович

студент
Санкт-Петербургский государственный университет телекоммуникаций им. М.А. Бонч-Бруевича, Санкт-Петербург

author@apriori-journal.ru

Аннотация. В статье описано устройство серверной части отказоустойчивой распределенной многопользовательской игры. Рассмотрен разработанный в ходе проектирования игрового сервера паттерн программирования.

Ключевые слова: golang; docker; отказоустойчивая система; распределенная система; игра; websocket; паттерное программирование.

DEVELOPMENT OF FAULT-TOLERANT DISTRIBUTED MULTIPLAYER GAME

Pushkin Ivan Alexandrovich

student
The Bonch-Bruevich Saint-Petersburg State University of Telecommunications, Saint-Petersburg

Abstract. The article describes the architecture of server side of fault-tolerant distributed multiplayer game and software design pattern developed for the game server of this application.

Key words: golang; docker; fault-tolerant system; distributed system; game; websocket; software design pattern.

Введение

Игровая деятельность в наибольшей мере влияет на развитие интеллектуальных процессов, стимулирует творческое мышление и формирует активную учебно-позновательную деятельность обучающегося, поэтому современное образование включает в себя так называемое игровое обучение. В качестве инструментов игрового обучения могут использоваться компьютерные игры. В данной статье описан процесс разработки обучающей многопользовательской отказоустойчивой распределенной игры [4].

Разработанная в рамках данного проекта игра («многопользовательская змейка») является матричной. Матричные игры – это игры, в которых игровое поле формируются из ячеек – простейших неделимых элементов игры [1].

Данная игра является клиент-серверным приложением. Серверная часть приложения отвечает за коммуникацию игроков, обработку игровых событий, расчет движения объектов. Клиентская часть приложения отвечает за отрисовку игрового пространства и объектов на экране, прослушку команд игрока и отправку их на сервер.

В качестве клиента используется браузер, а сервеная часть данного приложения состоит из сервера авторизации, балансировщика игрового трафика, облака с игровыми серверами с контроллером кластера, легкого веб-сервера для раздачи статического трафика, сервера с базой данных, сервера со средствами разработки и контроллера облака.

Отказоустойчивость данной информационной системы обеспечивается архитектурой ее серверной части, поэтому в статье будет рассмотрена именно серверная часть.

При разработке системы были использованы Linux Ubuntu, Eucalyptus, Go, Docker, PHP, Bash, Git, Nginx и Postgresql.

Постановка задачи

В ходе работы над игрой ставились следующие задачи:

1. Построение грид-системы для данной игры;
2. Разработка игрового модульного сервера, который подошел бы для любой другой подобной игры. Под модульностью имеется в виду заменяемость и независимость элементов игрового сервера друг от друга (зависимость только от интерфейсов);
3. Разработка паттерна программирования [8] для эффективного администрирования игровым сервером групп гоурутин (goroutine [2]) – обработчиков соединений с общими групповыми данными.

Описание игры

После успешной авторизации и обращения к балансировщику игроки подключаются к игровым серверам. Игровые сервера распределяют игроков по комнатам. В одной комнате может играть заданное при запуске игрового сервера количество игроков. Игроки играют за змеек. Цель игры – вырастить самую большую змейку и доминировать в комнате.

Интерфейсы игровых объектов:

```
type (
    // Игровой объект
    Object interface{}

    // Твердый объект
    Resistant interface {
        Object              // Твердый объект — игровой объект
        Strength() float32 // Твердый объект имеет твердость
    }

    // Живой объект
    Living interface {
        Object     // Каждый живой объект — игровой объект
        Resistant  // Каждый живой объект — твердый
        Die()      // Живой объект умирает
```

```
        Feed(int8) // Живой объект ест
    }

    // Неживой объект
    Notalive interface {
        Object                  // Неживой объект — игровой объект
        Break(*playground.Dot) // Неживой объект можно сломать
    }

    // Съедобный объект
    Food interface {
        Object // Съедобный объект — игровой объект
        // Съедобный объект имеет пищевую ценность
        NutritionalValue(*playground.Dot) int8
    }
)
```

Взаимодействие объектов происходит в результате сталкивания движущегося (живого) объекта-агрессора с любым другим объектом на игровом поле.

Архитектура серверной части игры

Серверная часть игры состоит из сервера авторизации, облака игровых серверов с контроллером кластера и балансировщиком нагрузки [5], сервера БД, легкого сервера для раздачи статического трафика, контроллера облака и сервера со средствами разработки (Git и Redmine).

Сервер авторизации принимает запросы на авторизацию и проверяет данные. Для авторизации пользователей используются их учетные записи в социальных сетях. Это удобно для пользователей, так как не требуется регистрация и можно быстро авторизоваться.

На сервере авторизации установлены Apache и PHP, и лежат несколько PHP скриптов, по одному на каждую социальную сеть (vk, fb и ok). При попытке пользователя авторизоваться эти скрипты получают маркер доступа своей социальной сети, выполняют некоторые запросы к

API социальной сети и записывают результат в БД. Затем скрипты отправляют клиенту соль, состоящую из идентификатора пользователя в социальной сети, идентификатора социальной сети и IP-адреса пользователя, и внутриигровой маркер доступа, состоящий из соли и секретного слова:

```
// Вычисление соли
$salt = hash('sha256', sprintf('%s_%s_%s',
     $soc_uid, $soc_id, $_SERVER['REMOTE_ADDR']
));
// Вычисление маркера доступа
$access_token = hash('sha256', $salt.'_'.SECRET);
```

Используя маркер доступа и соль пользователь может работать с балансировщиком игрового трафика, игровыми серверами и получать информацию из БД через REST API. Для работы с веб-сервером для раздачи статического трафика маркер доступа не требуется. К остальным элементам серверной части игроки не имеют доступа.

Балансировщик игрового трафика – это программа, хранящая информацию о всех активных игровых серверах и распределяющая пользователей по игровым серверам на основании их загруженности. Балансировщик игровых серверов написан на Go. Раз в D секунд балансировщик запрашивает у игровых серверов информацию о количестве открытых сетевых соединений и количестве открытых комнат. Балансировщик хранит информацию о каждом игровом сервере в следующей структуре:

```
type SnakeServer struct {
     addr      string    // Адрес игрового сервера
     poolLimit int       // Лимит комнат
     connLimit int       // Лимит соединений на комнату
     poolCount int       // Количество открытых комнат
     connCount int       // Количество открытых сетевых соединений
     ts        time.Time // Время последнего обновления информации
}
```

При обращении игрока балансировщик игровых серверов анализирует ситуацию, выбирает сервер и отправляет адрес и UUID [3] игрового сервера клиенту.

Игровой сервер – это программа рассчитывающая движение и взаимодействие объектов на игровом поле, раздающая игровой трафик игрокам и принимающая команды от них. Для передачи игровых данных используется протокол WebSocket [7].

Игровые сервера запускаются в облаке и уведомляют балансировщик игровых серверов о своем существовании, о лимите комнат и лимите игроков на комнату. Игровые сервера занимаются расчетом игровых событий и раздачей игрового трафика.

Поток игровых данных, получаемых от игрового сервера, для каждого игрока состоит из потока общей информации для всех игроков на сервере, потока общей информации для игроков в конкретной комнате и приватного пользовательского потока данных, который для каждого игрока свой.

Для хранения данных об игроках была выбрана СУБД Postgresql. БД игры состоит из 1 таблицы `players`, в которой хранятся следующие данные:

- `user_ip` – IP-адрес пользователя;
- `soc_id` – идентификатор социальной сети, из под которой пользователь авторизовался;
- `soc_uid` – идентификатор пользователя в социальной сети;
- `token` – маркер доступа полученный от социальной сети пользователя;
- `page_url` – ссылка на страницу пользователя в социальной сети;
- `name` – имя пользователя;
- `last_name` – фамилия пользователя;
- `last_time` – дата и время последней активности;
- `reg_time` – дата и время регистрации пользователя.

В качестве сервера для раздачи статического трафика был выбран Nginx. Здесь следует отметить, что сервер необходимо настроить так, чтобы он заставлял браузер кешировать данные.

Как альтернатива Nginx рассматривался вариант с веб-сервером на Go, который, во-первых, заставляет браузер кешировать данные, а во вторых, держит всю статику в оперативной памяти.

Компоненты системы запускаются в Docker контейнерах, которые запускаются Eucalyptus.

Описание паттерна pwshandler

В ходе работы над игровым сервером был разработан и опробован паттерн программирования pwshandler (Pool WebSocket Handler) [6], предназначенный для администрирования групп (в данной игре – комнат) потоков обработчиков соединений. Альтернативы данному паттерну автором пока найдено не было. Схема pwshandler заложена в ядро игрового сервера.

Данный паттерн можно представить как набор интерфейсов, декларирующих функциональные части сервера:

1. Интерфейс для общих данных комнаты. Общими данными может быть что угодно:

```
type Environment interface{}
```

2. Интерфейс менеджера комнат. Этот объект должен хранить комнаты и управлять ими, решать, когда комната должна быть добавлена, а когда удалена, вызывать фабрики комнат и распределять игроков:

```
type PoolManager interface {
// AddConn должен найти подходящую комнату и вернуть
// общие данные
AddConn(ws *websocket.Conn) (Environment, error)
// DelConn удаляет информацию об игроке из комноты
DelConn(ws *websocket.Conn) error
}
```

3. Интерфейс обработчика соединения. Этот объект занимается работой с соединением. Он получает соединение и общие данные комнаты:

```
type ConnManager interface {
// Handle - это обработчик. data - это общие данные группы
Handle(ws *websocket.Conn, data Environment) error
// HandleError для информирования пользователей об ошибках
HandleError(ws *websocket.Conn, err error)
}
```

4. Интерфейс «проверяльщика». «Проверяльщик» занимается логикой авторизации:

```
type RequestVerifier interface {
// Verify проверяет соединение и возвращает ошибку
// в случае, если что-то не так
Verify(ws *websocket.Conn) error
}
```

Данный паттерн с фабриками для генерации комнат и общих данных комнаты оказался очень удобным на практике и может быть использован не только при разработке игр, но и других приложений с объединением участников (соединений) в группы (или, как в данной игре, комнаты).

Сеанс игры и протоколы

Самые первые запросы – это запросы на легкий сервер за статическими данными игры.

Далее, идет попытка авторизоваться с помощью социальной сети пользователя. Авторизация в выбранных социальных сетях идет примерно одним образом: либо при помощи OAuth, либо при помощи OAuth 2.0. Авторизация различается только количеством получаемых данных пользователя. В итоге, скрипты авторизации получают маркер доступа для работы с API социальной сети и генерируют ответ с солью и маркером для работы с серверной частью игры:

```
{'salt':  '...', 'token': '...'}
```

Получив соль и маркер клиент обращается к балансировщику игровых серверов при помощи обычного HTTP запроса:

`http://addr:port?salt=...&token=...`

Балансировщик отвечает адресом и UUID игрового сервера:

`{'addr': '...', 'uuid': '...'}`.

Далее, клиент, получив адрес игрового сервера и имея соль и маркер, подключается к игровому серверу, также как к балансировщику. Игровой сервер проверяет две вещи: хеш сумму маркера и уникальность маркера. Затем, соединение апгрейдится в WebSocket соединение и начинается процесс игры. Игровой сервер, также как и другие компоненты системы, использует JSON и форму сообщений:

`{"header": HEADER, "data": DATA}`

Заголовки и структуры входящих и исходящих сообщений:

```
const (
     // Определение заголовков сообщений игрового сервера
     HEADER_ERROR   = "error"   // Для информации об ошибках
     HEADER_INFO    = "info"    // Для информационных сообщений
     HEADER_POOL_ID = "pool_id" // Для сообщения о номере комнаты
     HEADER_CONN_ID = "conn_id" // Для сообщения об идентификаторе
игрока
     HEADER_GAME = "game" // Для передачи игровых данных
)

// Структура исходящих сообщений игрового сервера
type OutputMessage struct {
     Header string      `json:"header"`
     Data   interface{} `json:"data"`
}

// Структура входящих сообщений. Здесь используем json.RawMessage,
// чтобы сперва определить вид сообщения, а потом анализировать
// поступившие данные
type InputMessage struct {
```

```
    Header string          `json:"header"`
    Data   json.RawMessage `json:"data"`
}
```

Далее, игровой сервер пересылает пользователю идентификатор комнаты и идентификатор управляемого игроком объекта (змейки). Получив эти данные, клиент, используя соль и маркер доступа, обращается к БД и передает полученные идентификаторы и UUID сервера туда. БД отмечает, что данный пользователь находится на игровом сервере с полученным UUID, в комнате с полученным идентификатором и управляет соответствующим объектом.

Затем, из БД клиент получает идентификаторы объектов соперников, связанные с ними имена пользователей и ссылки на их страницы в социальных сетях:

```
[    {'object_id': 1, name: 'Ivan', link: '...'},
     {'object_id': 2, name: 'Ksenia', link: '...'},
     ...
     {'object_id': N, name: 'Sergey', link: '...'}    ]
```

Далее, клиент работает только с игровым сервером и базой данных.

Заключение

В резульате проделанной работы спроектирована грид-система, разработан игровой сервер и разработан паттерн програмирования pwshandler. Описание паттерна pwshandler можно найти здесь – [6].

Разработанная информационная система, благодаря модульности своих компонентов, без существенных изменений может быть использована при создании других подобных приложений.

Список использованных источников

1. Селянкин В.В., Злыгостев И.С. Технология разработки компьютерных матричных игр // Известия ЮФУ. Технические науки. 2005. № 3. С. 86-90.
2. golang-book.com [Электронный ресурс]. Режим доступа: http://www.golang-book.com (дата обращения: 19.04.15).
3. RFC 4122. A Universally Unique IDentifier (UUID) URN Namespace // The Internet Engineering Task Force [Электронный ресурс]. Режим доступа: https://www.ietf.org/rfc/rfc4122.txt (дата обращения 29.03.2015).
4. Таненбаум Э., Стеен М. Ван. Распределенные системы. Принципы и парадигмы. СПб.: Питер, 2003. 877 с.
5. Токарчук А.М. Применение грид-систем при развертывании web-сайта // Информационно-управляющие системы . 2010. № 3. С. 51-55.
6. Описание паттерна pwshandler [Электронный ресурс]. Режим доступа: https://github.com/ivan1993spb/pwshandler (дата обращения: 21.03.2015).
7. RFC 6455. The WebSocket Protocol // The Internet Engineering Task Force [Электронный ресурс]. Режим доступа: http://tools.ietf.org/html/rfc6455 (дата обращения: 23.03.2015).
8. Крайнова Е.А. Теоретические аспекты паттерного программирования // Вестник Волжского университета им. В.Н. Татищева. 2013. № 2 (21). С. 82-90.