АО «НИИМЭ»

Технология удалённого вызова процедур

2020

ОГЛАВЛЕНИЕ

1. ОБЩЕЕ ОПИСАНИЕ

Удалённый вызов процедур – технология для вызова функции в другом адресном пространстве (на удалённых компьютерах либо в независимой сторонней системе на том же устройстве) [1].

Текущий документ конкретизирует технологию удалённого вызова, предоставляя разработчикам унифицированный набор подходов, вспомогательных технологий и инструментов. Технология подразумевает клиент-серверную архитектуру. Предоставляемая информация может быть применена для разработки как клиентской, так и серверной части. Детали, намеренно не описанные или отмеченные как выходящие за рамки документа, могут быть интерпретированы свободно, если они были согласованы между всеми участниками взаимодействия.

В общем виде технология удалённого вызова подразумевает собой некоторый протокол, соблюдая который участники имеют возможность обмениваться сообщениями в режиме реального времени в понятном и едином формате. В документе могут быть описаны не все правила, т.к. часть из них является зоной ответственности разработчика, и должны согласовываться и конкретизироваться дополнительно.

Текущий документ описывает правила описания интерфейса, правила сериализации, коммуникационные правила, транспортную архитектуру, предлагает унифицированный интерфейс для упрощения взаимодействия между компонентами системы и ускорения разработки решения.

Текущий документ не описывает интерфейсы взаимодействия на уровне бизнес-логики, не описывает требования к защите информации, не описывает требования к сети.

Ключевые аспекты системы:

1. бинарный формат сериализации MessagePack [2];
2. транспортный протокол TCP/IP [3];
3. широкий, стабильный, безопасный и надёжный канал;
4. поддерживается кодогенерация;
5. асинхронные вызовы не предусмотрены.

Бинарный формат сериализации нужен для улучшения быстродействия, т.к. человекочитаемость не требуется. Формат MessagePack был выбран ввиду простоты реализации и компактности сообщений. MessagePack не стандартизован. Существуют и другие бинарные протоколы, в том числе и стандартизованные, например, CBOR [4] или BSON [5], если необходимо выполнить такое требование.

В терминологии TCP/IP сервер – узел, предоставляющий порт, клиент – узел, запрашивающий соединение. В терминологии бизнес-логики в понятия клиента и сервера вкладывается другой смысл. Клиент – инициатор взаимодействия, сервер – сторона, принимающая запрос. Чтобы не смешивать эти два уровня, в терминологии бизнес-логики клиент и сервер будут обозначаться как пользователь (user) и провайдер (provider). Провайдер предоставляет интерфейс для вызова, т.е. является принимающей вызов стороной, пользователь (интерфейса) через этот интерфейс осуществляет вызов функций, т.е. является отправляющей стороной. Пользователь инициирует передачу, формирует запрос и получает ответ, если он предусмотрен. Провайдер обрабатывает запрос, вызывает функцию и отправляет ответ, если он предусмотрен. Роли пользователя и провайдера не закреплены, в процессе работы участники одновременно могут являться и пользователями, и провайдерами.

Предполагается, что канал передачи широкий, стабильный, безопасный и надёжный и это гарантируется извне. Соблюдение этих требований выходит за рамки ответственности текущего документа.

Существует множество решений с использованием метаязыка описания интерфейса (Interface Definition Language, IDL) [6]. Тем не менее, предполагается, что количество компонентов, использующих описываемую технологию, будет невелико и однообразно на уровне языков программирования, поэтому использование промышленного решения для кодогенерации не целесообразно. Тем не менее, описываемая технология предлагает кодогенерацию.

Асинхронные вызовы не предусмотрены на текущем этапе развития технологии. По умолчанию считается, что все вызовы являются синхронными, т.е. после вызова функции обязательно следует ответ или его не будет совсем. Асинхронные вызовы могут быть добавлены по мере развития.

Сценарий удалённого вызова со стороны пользователя:

1) формирование запроса на вызов функции;
2) сериализация запроса;
3) отправка данных по сокету;
4) приём ответных данных;
5) десериализация ответа;
6) выполнение бизнес-логики.

Сценарий удалённого вызова со стороны провайдера:

1) приём запроса;
2) десериализация запроса;
3) диспетчеризация;
4) выполнение бизнес-логики;
5) сериализация ответа;
6) отправка данных по сокету.

2. ЯЗЫК ОПИСАНИЯ ИНТЕРФЕЙСА

2.1. Синтаксис

Для описания интерфейса вызываемых функций без привязки к конкретному языку программирования используется метаязык ZvezdaIDL. После описания интерфейса нужно специальной программой на основе файла с описанием сгенерировать специализированный файл для одного из поддерживаемых языков программирования. В данный момент поддерживаются языки C++ и C#. В случае C++ после компиляции будет сгенерирован заголовочный файл, в случае C# будет сгенерирован файл с расширением cs.

Язык поддерживает следующие типы данных:

Таблица 1. Базовые типы данных

Zvezda IDL	Описание
I8	Целое знаковое число, 1 байт
I16	Целое знаковое число, 2 байта
I32	Целое знаковое число, 4 байта
U8	Целое беззнаковое число, 1 байт
U16	Целое беззнаковое число, 2 байта
U32	Целое беззнаковое число, 4 байта
U64	Целое беззнаковое число, 8 байт
Byte	U8
Bool	Логический тип
String	Строка, UTF-8
Array<T>	Массив данных типа T
Binary	Массив байт (Array<Byte>)
Enum	I32

Также язык поддерживает создание пользовательских типов данных (структур), которые могут быть сформированы из базовых типов. Области, где описываются такие типы данных, обозначаются ключевым словом `Struct`.

Помимо конструкции для описания структур, язык поддерживает описание перечислений, интерфейсов функций, уведомлений и т. д. Каждое ключевое слово открывает описательную область, в рамках которой некоторые другие области открывать запрещается. Перед каждой областью должен идти обязательный комментарий. Каждая область должна иметь имя. Каждая открытая область должна заканчиваться ключевым словом `End`.

Таблица 2. Языковые конструкции

Zvezda IDL	Описание
Struct	Начало описания структуры
Enum	Начало описания перечисления
Api	Начало описания API
Lib	Начало описания общего (библиотечного) блока кода
Function	Начало описания интерфейса функции
Notification	Начало описания интерфейса уведомления
In	Начало описания входных параметров
Out	Начало описания выходных параметров
Error	Начало описания ошибочных статусов
End	Завершение описания области
Import	Подключение внешних файлов описания интерфейса
Version	Текущая версия API

Следует обратить внимание, что все типы данных и все языковые конструкции всегда начинаются с заглавной буквы.

Разрешённые типы:

```
Type = {I8, I16, I32, U8, U16, U32, U64, Byte, String, Array<Type>, Binary,
Bool, <StructName>}
```

Описание структур и перечислений:

```
# Comment
Struct <Name>
      <Name: Type> # Comment
End


#Comment
Enum <Name>
      <Key = Value (I32)> # Comment
End
```

Описание функций и уведомлений:

```
#Comment
Function <Name>
      {Struct, Enum}

      [In, Out]
            <Name: Type> # Comment
      End
      [Error]
            <Key = Value (I32)> # Comment
      End
End

#Comment
Notification <Name>
      {Struct, Enum}

      [In]
            <Name: Type> # Comment
      End
      [Error]
            <Key = Value (I32)> # Comment
```

```
      End
End
```

Описание библиотечного блока кода и API:

```
#Comment
Lib <Name>
      {Struct, Enum}
End

#Comment
Api <Name>
Version=<major : U16>[.<minor : U16>]
      {Struct, Enum, Function, Notification}
End
```

Версия API задаётся разработчиком и ответственность за ведение версии также лежит на разработчике интерфейса. Версия представляет собой или одиночное значение типа U16, либо два значения типа U16 через символ `'.'`. Первое число означает мажорную версию, второе число – минорную. Допускаются только числовые значения версий. Следует обратить внимание, что в текущей версии разделение на мажорную и минорную версии является условным, т.е. версии 1.1.0 и 1.1.1 **несовместимы** друг с другом. Обратная совместимость будет поддержана позже. На сегодняшний день обе стороны могут взаимодействовать друг с другом только в том случае, если версии каждого из них совпадают полностью.

Следует обратить внимание, что комментарии (секция Comment) является обязательной. Комментарии являются однострочными и начинаются с символа `'#'`. Часть строки, идущая после этого символа, игнорируется и может быть использована для текстовых пояснений.

Дополнительно могут быть подключены внешние файлы описания интерфейса через ключевое слово Import. Имя файла пишется вместе с его расширением. Подключение разрешено только в самом начале файла (по аналогии с заголовочными файлами из C++):

```
Import external_file.idl
```

После подключения внешнего файла все описанные в секции Lib конструкции этого файла будут доступны там, где произошло подключение. Следует отметить, что для повторного использования будет доступен только код из области Lib. Если в файле дополнительно была описана область Api, то её элементы импортированы не будут, поэтому рекомендуется в импортируемых файлах описывать только области Lib.

Обращение к импортируемым элементам происходит следующим образом:

```
<LibName>.{<StructName>, <EnumName>}
```

Описание общей структуры файла ZDL:

```
[Import <Name>.zdl]
{Lib, Api}
```

Пример описания интерфейса:

1) Файл common.zdl:

```
# Общие элементы, характерные для нескольких API
Lib Common
      # Информация о пользователе
      Struct UserInfo
            id:   U16    # Идентификатор
            name: String # Имя
```

```
	End
End
```

2) Файл radio.zdl:

```
Import common.zdl

# Интерфейс для компонентов сети
Api Radio
Version=1.00
	# Состояние клиента
	Enum ClientState
		INIT = 1		# Клиент инициализирован
		ACTIVATED = 2	# Клиент активирован
		DEACTIVATED = 3	# Клиент деактивирован
	End

	# Авторизация пользователя в ЭБКС.
	Function Authorize
		In
			info: Common.UserInfo	# Информация о пользователе
			state: ClientState	# Состояние клиента
			password: String	# Пароль
		End
		Error
			OK = 0			# Функция завершена успешно
			INCORRECT_USER = 1	# Неизвестное имя пользователя
			INCORRECT_PASSWORD = 2	# Некорректный пароль
		End
	End
End
```

2.2. Компилятор ZvezdaIDL

Для трансляции языка ZvezdaIDL в какой-либо язык программирования можно воспользоваться специальным компилятором zvezdac, который проводит проверку синтаксиса и формирует выходной файл, соответствующий выбранному языку программирования. При наличии ошибок в файле описания, компилятор выводит в консоль диагностику. Затем этот интерфейс может быть использован при реализации механизма удалённого вызова функций. Компилятор принимает файл описания интерфейса с расширением .zdl. Сгенерированные компилятором файлы редактировать не рекомендуется.

Для того чтобы произвести удалённый вызов функции, на стороне пользователя в коде бизнес-логики в каком-то виде должен произойти вызов этой функции. Т.к. провайдер может находиться в другом адресном пространстве, то возникает необходимость в предоставлении пользователям некоторого интерфейса, идентичного интерфейсу провайдера. При этом, вызов функции внешне для пользователя должен происходить так же, как если бы функция вызывалось локально. Таким образом, возникает необходимость в создании дополнительного слоя абстракции, который обеспечивал бы удобный вызов функций провайдера на уровне бизнес-логики на стороне пользователя. В качестве такой абстракции может выступать набор функций-заглушек (стабов), которые в совокупности предоставят весь необходимый и достаточный интерфейс. Стабы скрывают сам вызов непосредственно и позволяют прозрачно для пользователя произвести вызов той или иной функции, скрыв детали самого вызова.

Стабы с точки зрения пользователя содержат реализацию запросов на вызов функций провайдера. Реализации запросов заключены в одноимённых функциях сервера, что позволяет пользователю вызывать функции без информации об адресном пространстве провайдера. В каждом таком пользовательском стабе происходит следующий ряд действий:

1) сериализация входных параметров;
2) упаковка данных в соответствии с протоколом;
3) передача пакета по сокету.

Стабы с точки зрения провайдера имеют некоторые отличия от пользовательских стабов. Задача таких функций предоставить возможность автоматической диспетчеризации запросов, соблюдение протокола и транспортировку пакетов. Для запуска этих действий эти классы предоставляют дополнительный метод `Start`, который должен быть вызван управляющим потоком и который этот поток захватывает. В каждом провайдерском стабе происходит следующий ряд действий:

1) получение запроса;
2) десериализация входных параметров;
3) диспетчеризация;
4) отправка ответа (если необходимо).

Пример использования заглушек в пользовательском управляющем коде (C++):

```
auto out_params = api.OpenSession("user", "password");
```

Таким образом, компилятор выполняет две основные задачи:

1. переводит описание языковых конструкций из IDL в поддерживаемый язык программирования (всегда),
2. на основании переведённых языковых конструкций формирует интерфейс для вызовов функций (по запросу).

Компилятор предоставляет сервис генерации заглушек на основании файла описания интерфейса. Все заглушки являются методами некоторого сгенерированного класса Api. Содержимое сгенерированного кода меняется в соответствии с TCP-ролью (клиент/сервер) и RPC-ролью (юзер/провайдер). Комбинация этих настроек генерирует уникальный файл со своими особенностями. Синтаксис использования:

```
zvezdac.exe filepath: <path>.zdl toolchain: cpp | cs [rpc-role: user | provider]
[tcp-role: client | server]
```

Пример вызова:

```
zvezdac.exe D:/files/admin_in.zdl cpp user client
```

Если опциональный параметр rpc-role не был установлен, то генерация стабов произведена не будет, но перевод языковых конструкций будет осуществлён. При использовании последнего опционального параметра tcp-role в этом же файле будет дополнительно подготовлен транспорт. Таким образом, после работы компилятора можно получить следующие сущности:

1) для юзера: `<Name>`, `<Name>Api`;
2) для провайдера: `<Name>, I<Name>Api, <Name>Disp, <Name>Runner`.

Для юзера формируется сущность `<Name>`, содержащая перевод IDL на один из поддерживаемых языков программирования, а также `<Name>Api`, предоставляющая стабы для вызова функций из управляющего кода. Для провайдера формируется также сущность, содержащая языковые конструкции `<Name>`, а также интерфейс, который необходимо реализовать провайдеру самостоятельно, т.к. компилятор не обладает информацией о реализации генерируемых функций. Компилятор лишь может предоставить список функций, которые были описаны в файле zdl. Также компилятор предоставляет сущность `<Name>Disp`, которая предоставляет провайдеру возможность провести автоматическую диспетчеризацию поступившего запроса от юзера. Для перевода провайдера в режим ожидания запросов,

необходимо вызвать метод `Run` из `<Name>Runner`, что позволит встать в ожидание новых подключений от пользователя, автоматическую обработку его запросов и отправки ответов.

Более подробное описание интерфейсов этих сущностей приведено в последней главе документа «**Ошибка! Источник ссылки не найден.**».

Компилятор поддерживает зависимости. Зависимость – это подключаемый через директиву Import дополнительный файл. Зависимости обрабатываются перед основным файлом. Для каждого подключаемого файла будет вызван компилятор и сформирован свой файл со структурами данных. Другими словами, для генерации кода из файла zdl достаточно один раз вызвать компилятор для целевого файла, зависимости будут обработаны автоматически, и на в конце будет получен готовый к работе набор файлов. Стоит также отметить два момента: 1) стабы будут сгенерированы только для целевого файла, 2) зависимости не могут быть вложенными, т.е. директивы Import внутри подключаемого файла будут проигнорированы компилятором.

2.3. Кодогенерация для C++

В случае C++ после компиляции будет сформирован заголовочный файл. Каждой конструкции из ZvezdaIDL ставится в соответствие конструкция из C++:

Таблица 3. Сопоставление языковых конструкций ZvezdaIDL и C++

Zvezda IDL	C++
I8	char
I16	short int
I32	int
U8	unsigned char
U16	unsigned short int
U32	unsigned int
U64	unsigned long
Byte	unsigned char
Bool	bool
String	string
Binary	vector<unsigned char>
Array<T>	vector<T>
Struct	struct
Enum	enum class
Api, Server, Client, Function, Notification	namespace
In, Out	namespace
Error	enum class
End	};
Import	#include

Имена функций и уведомлений конвертируется в пространство имён. Входные и выходные параметры представляются в виде структур InParams и OutParams внутри этих пространств имён соответственно. Описание ошибок представляется в виде перечисления в этом же пространстве имён.

Для поддержки механизма сериализации внутри структур InParams и OutParams генерируется дополнительный код. Сериализация осуществляется через вызов метода Serialize. Десериализация происходит через статическую функцию Deserialize.

Пример сериализации:

```
Radio::Authorize::InParams in_params("user", "password");
Binary serialized = in_params.Serialize();
```

Пример десериализации:

```
Binary serialized = "92 A4 75 73 65 72 A8 70 61 73 73 77 6F 72 64";
auto in_params = Radio::Authorize::InParams::Deserialize(serialized);
```

Пример сгенерированного хидера:

```
struct UserInfo {
    std::string name;
};

namespace Radio {

enum class Functions {
    Authorize = 1,
};

enum ClientState {
    INIT = 1,
    ACTIVATED = 2,
    DEACTIVATED = 3,
};

namespace Authorize {

struct InParams {
 UserInfo info;
 ClientState state;
 std::string password;
 InParams(const UserInfo& info,const ClientState& state, const std::string&
password): info(info), state(state), password(password){}
 operator Binary() { // Сериализация }
 InParams(const Binary& bin) { // Десериализация }
};

struct OutParams {
};

enum class Error {
    OK = 0,
     INCORRECT_USER = 1,
    INCORRECT_PASSWORD = 2,
};

}; /// Authorize
}; /// Radio
```

2.4. Кодогенерация для C#

Для C# будет сгенерирован файл с расширением cs, в котором будут сформированы все структуры данных, которые были описаны в соответствующем файле описания интерфейса. Правила сопоставления языковых конструкций представлены ниже в таблице.

Таблица 4. Сопоставление языковых конструкций ZvezdaIDL и C#

Zvezda IDL	C#
I8	sbyte
I16	short
I32	int
U8	byte
U16	ushort
U32	uint
U64	ulong
Byte	byte
Bool	bool
String	string
Binary	byte[]
ArrayT	List<T>
Struct	class
Enum	enum
Import	using

3. КОММУНИКАЦИОННЫЕ ПРАВИЛА

3.1. Формат сообщений

Передаваемые пакеты имеют структуру вида заголовок + данные. Заголовок обязателен и всегда имеет фиксированный размер 10 байт. Заголовок не сериализуется и передаётся как есть. Каждый элемент заголовка закодирован в формате BIG ENDIAN. Пакеты разделяются на входящие и исходящие, которые в контексте документа будут обозначаться общими терминами запрос и ответ. Всего элементов в заголовке 4, тем не менее в зависимости от типа пакета состав заголовка отличается. Отличия показаны в таблице ниже.

Таблица 5. Формат заголовка в RPC-пакете

Запрос	PKG_TYPE	MSG_ID	FUNC_ID	PARAMS_LEN
Ответ	PKG_TYPE	MSG_ID	STATUS	PARAMS_LEN

После заголовка следует опциональное поле с данными.

Коммуникационный пакет для исходящего сообщения имеет вид:

PKG_TYPE : U16 + **MSG_ID** : U16 + **FUNC_ID** : U16 + **PARAMS_LEN** : U32 + [**IN_PARAMS** : tuple]

где PKG_TYPE – тип пакета, FUNC_ID – уникальный идентификатор вызываемой функции, MSG_ID – идентификатор запроса, PARAMS_LEN – размер передаваемых параметров в байтах, IN_PARAMS – последовательность (кортеж) входных параметров размера PARAMS_LEN.

PKG_TYPE нужен для идентификации типа пакета. Существует несколько типов пакета: вызов функции, ответ, уведомление и служебный. Отправка вызова функции подразумевает удалённый синхронный вызов и обязательное ожидание ответа. Отправка ответа может быть осуществлена только в ответ на полученный запрос. Отправка уведомления технически тоже является удалённым вызовом функции, но другого вида. В случае уведомления происходит вызов т.н. callback-функции, после вызова которой не подразумевается получение ответа, поэтому пользователь не должен ожидать никакого ответа от провайдера. Служебный пакет нужен для информирования об ошибках протокола и других нештатных ситуациях. Таким образом, PKG_TYPE может принимать следующие значения:

Таблица 6. Типы пакетов

Исходящий	**PKG_TYPE**
Вызов функции	0x0001
Уведомление	0x0003
Служебный запрос	0x00F1
Входящий	
Ответ	0x0002
Служебный ответ	0x00F2

MSG_ID необходим для сопоставления ответа сервера с запросом для верификации и исключения ошибочной ситуации, когда в открытое соединение отправляется другой пакет. Если нет необходимости отслеживать сообщение, тогда идентификатор может быть установлен равным 0. Ответственность за формирование идентификатора и его обработку лежит на пользователе. MSG_ID может принимать произвольное значение. Провайдер не должен никак его анализировать или изменять MSG_ID в ответном сообщении.

FUNC_ID нужен для однозначного определения запрашиваемой функции. Провайдер должен публиковать список доступных к вызову функций. Ответственность за обеспечение однозначного соответствия между идентификаторами и вызываемыми функциями лежит на провайдере.

PARAMS_LEN характеризует размер данных в байтах, которые расположены после. Указывается размер фактически передаваемых данных, т.е. уже после сериализации.

IN_PARAMS содержит набор входных параметров вызываемой функции. Входные параметры всегда оборачиваются в кортеж, даже в случае secли параметр один. Если же функция не подразумевает никаких входных параметров, тогда поле IN_PARAMS может отсутствовать.

Если после отправки сообщения подразумевается ответ, то провайдер обязан сформировать сообщение вида:

PKG_TYPE : U16 + **MSG_ID** : U16 + **STATUS** : U16 + **PARAMS_LEN** : U32 + [**OUT_PARAMS** : tuple],

MSG_ID в ответе должен иметь значение, которое было установлено в запросе, полученном от пользователя. Значение может быть установлено произвольно. Контроль уникальности идентификаторов лежит на отправляющей стороне.

STATUS информирует пользователя о состоянии выполнения его запроса. Для типа пакета «ответ» статус оповещает пользователя о наличии ошибок на уровне бизнес-логики. В случае успеха статус принимает значение 0x00, в любом другом случае провайдер может информировать пользователя через значения, описанные на уровне бизнес-логики. Интерпретация такого статуса лежит на уровне бизнес-логики и выходит за рамки описываемой технологии.

Таблица 7. Значения пользовательских статусов

Описание	Значение STATUS
Успех	0x0000
Пользовательский статус	0x0001 – 0xFFFF

Если на стороне провайдера во время приёма сообщения обнаружена ошибка, тогда формируется служебный пакет. В таком случае статус отражает ситуацию на протокольном уровне. Задача такого статуса проинформировать пользователя о проблемах, нештатных ситуациях или предоставить какую-то другую служебную информацию если это необходимо для исправления ошибки.

Пользовательский статус, т.е. если это не служебный пакет, отражает бизнес-уровень и может принимать любое значение, которое устанавливается провайдером в соответствии с собственной политикой обработки ошибок. Если пакет служебный, то значения статуса интерпретируются протоколом, т.е. статус не описывает состояние провайдера на уровне бизнес-логики. Значения статусов для пользовательского и служебного пакета могут перекрываться, но т.к. статусы несут разный смысл в зависимости от типа пакета, одинаковые значения будут указывать на разные ситуации.

Таблица 8. Значения служебных статусов

Описание	Значение STATUS	Пояснение
Неверный тип пакета	0x00F1	Если был отправлен запрос, тип пакета должен быть соответствующим.
Нарушена структура пакета	0x00F2	Если присутствуют не все параметры, нарушен порядок параметров или есть лишние элементы.
Нарушена последовательность пакетов	0x00F3	Если в ответ на вызов функции пришёл пакет, не соответствующий ответу или служебному сообщению
Истекло время ожидания	0x00F4	Если после получения запроса прошло указанное количество времени.
Функция не найдена	0x00F5	Если провайдер не смог вызвать запрашиваемую функцию.
Провайдер приостановлен	0x00F6	Если провайдер завершает работу, то приём сообщений приостанавливается и на все запросы юзера отправляется извещение о невозможности принять запрос.
Входные параметры функции неверны	0x00F7	Если переданные параметры не соответствуют заявленным параметрам функции.
Ошибка во время хэндшейка	0x00F8	Если настройки юзера не соответствуют настройкам провайдера API
Неизвестная ошибка	0x00FF	Если произошёл сбой по неизвестной причине.

OUT_PARAMS содержит набор выходных параметров функции или может отсутствовать, если функция не предполагает возвращаемых параметров. При возникновении ошибочной ситуации поле OUT_PARAMS может быть использовано для передачи дополнительной информации, в том числе служебной.

3.2. Согласование конфигураций

Перед началом взаимодействия необходимо провести согласование конфигураций сторон для проверки и/или проведения действий для обеспечения совместимости. Для этого необходимо произвести обмен служебными сообщениями, содержащими информацию о текущих настройках. Если настройки позволяют продолжить работу, тогда соединение считается установленным, в противном случае соединение разрывается. В контексте документа такая процедура может обозначаться как «хэндшейк», что является синонимом процедуры согласования.

Процедуру согласования инициирует юзер. После установки соединения юзер отправляет специальное служебное сообщение следующего формата:

PKG_TYPE : U16 + **MSG_ID** : U16 + **FUNC_ID** : U16 + **PARAMS_LEN** : U32 + [**IN_PARAMS** : tuple]

При формировании сообщения юзер должен установить тип пакета 00F1 (служебный запрос), MSG_ID = 0x0000 и FUNC_ID = 0x0000. Поле IN_PARAMS_LEN зависит от размера поля IN_PARAMS. Поле IN_PARAMS включает в себя три элемента данных:

1) версия протокола RPC (в этой редакции всегда = 1);
2) версия API;
3) название API.

Т.е. необходимо сформировать специальный RPC-пакет следующего формата:

00F1 + 0000 + 0000 + **IN_PARAMS_LEN** : U32 + **IN_PARAMS** : tuple

Особенность сообщения для хэндшейка заключается в отсутствии правил сериализации входных параметров, т.к. в зависимости от версии протокола RPC правила сериализации могут быть впоследствии изменены. Входные параметры представляют собой набор байтов следующей структуры:

RPC_VER : U8 + **API_VER** : U32 + **LEN** : U8 + **NAME** : ArrayU8,

где LEN обозначается длина имени API, а NAME – это имя API, представленное в виде UTF-8 строки в байтовом формате.

Версия API (API_VER) состоит из 4 байтов, которые распределяются следующим образом: мажорная версия [2 байта] + минорная версия [2 байта]. Ответственность за ведение версий является ответственностью разработчика. При изменении версии и разработчик API, и пользователь API обязаны отслеживать корректность управляющей логики на своей стороне.

В текущей версии протокола RPC = 1 мажорная и минорная версии логически неразделимы и образуют единую версию API посредством конкатенации двух значений в одно, которое впоследствии интерпретируется как версия API. В будущих версиях разделение на мажорную и минорную версию и их функциональность считается возможным, и в этом случае логика их обработки будет определена более полно.

Если параметры юзера позволяют установить с ним соединение, то провайдер отвечает служебным сообщением (PKG_TYPE = 0x00F2) с успешным статусом 0x0000. С этого момента соединение считается установленным. В противном случае, провайдер отвечает служебным сообщением со статусом 0x00F8, что кодирует неуспешность хэндшейка. Неуспешность процедуры обуславливается несовпадением хотя бы одного из трёх параметров: версии протокола RPC, версии API или названия API. Неуспешность хэндшейка влечёт за собой закрытие соединения и завершение взаимодействия между юзером и провайдером. Дополнительно в выходных параметрах OUT_PARAMS кодируется конфигурация провайдера, на основании описанной выше структуры, т.е. в ответ юзеру сообщается приемлемая конфигурация, с которой необходимо отправить следующий запрос на хэндшейк, если это возможно. Следует отметить, что провайдер формирует ответ в виде служебного сообщения по правилам той версии протокола, которую он поддерживает на момент получения запроса, т.к. не обязан поддерживать любую другую версию кроме своей.

В будущих версиях протокола допускается добавление механизма динамической настройки версий, т.е. согласование до тех пор, пока не будет найдена поддерживаемая обеими сторонами конфигурация.

Таким образом, процедура успешного хэндшейка выглядит следующим образом:

1. юзер отправляет служебный запрос (PKG_TYPE = 0x00F1);
2. провайдер проверяет параметры юзера;
3. провайдер отвечает служебным сообщением (PKG_TYPE = 0x00F2);
4. обе стороны готовы к работе или работа завершена.

4. ПРАВИЛА СЕРИАЛИЗАЦИИ

В коммуникационном пакете сериализуются только передаваемые параметры в поле IN_PARAMS или OUT_PARAMS. Заголовок сообщения не сериализуется, размер и структура заголовка фиксированы и определены.

Для сериализации и десериализации передаваемых данных используется бинарный формат MessagePack [7]. Типы данных и правила сериализации описаны в соответствующей спецификации [8]. В общем виде формат сериализованного пакета схож с форматом TLV. В качестве реализаций для языков C++ и C# предлагается использовать референсные реализации[9].

Таблица 9. Сопоставление типов ZvezdaIDL и MessagePack

Zvezda IDL	MessagePack
I8	fixint8
I16	fixint16
I32	fixint32
U8	fixuint8
U16	fixuint16
U32	fixuint32
U64	fixuint64
Byte	fixuint8
Bool	bool
String	String
Binary	Binary
Array<T>	Array
Struct	Array
Enum	fixint32
InParams	Array
OutParams	Array
Error	как Enum
Api, Lib, Function, Notification	-

Для каждого типа Zvezda IDL предусмотрен свой формат сериализации, в зависимости от знака, размера, длины вложенных данных и т. п. Для простых типов в общем виде правила сериализации следующие: тэг + значение, где тэг меняет значение в зависимости от разрядности и знаковости. Отрицательные числа кодируются при помощи 8-битного дополнительного кода, т.е. путём добавления единицы к инверсии. Например, число -1 кодируется как 0xFF, а -128 кодируется как 0x80.

Далее по тексту знак сложения ‘+’ обозначает конкатенацию. Например, под выражением 0xD0 + 0x01 подразумевается набор из 2-х байтов ‘D001’.

Таблица 10. Правила сериализации простых типов

Zvezda IDL	Описание	Пример
I8	0xD0 + значение	01 -> D0 01
I16	0xD1 + значение	01 -> D1 00 01
I32	0xD2 + значение	01 -> D2 00 00 00 01
U8	0xCC + значение	01 -> CC 01
U16	0xCD + значение	01 -> CD 00 01
U32	0xCE + значение	01 -> CE 00 00 00 01
U64	0xCF + значение	01 -> CF 00 00 00 00 00 00 00 01
enum	кодируется как I32	
Byte	кодируется как U8	
Bool	кодируется одним байтом: 0xC2, если false, и 0xC3, если true	false -> 0xC2 true -> 0xC3

Для сложных типов существуют особые правила кодирования в зависимости от размера данных. Ниже в таблице описаны правила сериализации, где под 0xLL понимается один байт длины. Длина может кодироваться несколькими байтами или быть объединена вместе с тэгом, если размер данных может быть закодирован несколькими битами.

Таблица 11. Правила сериализации типа string

Размер данных (байт)	Бинарное представление	Пример
[0; 31]	b101XXXXX + данные, где пять младших бит кодируют длину строки	112233 -> 93 112233
[32; 255]	0xD9 + 0xLL + данные	01 02 ... FF -> D9 FF 0102...FF
[256; 65,535]	0xDA + 0xLL 0xLL + данные	0001 0002 ... FFFF -> DA FFFF 0001 0002 ... FFFF

Таблица 12. Правила сериализации типа Binary

Размер данных (байт)	Бинарное представление	Пример
[0; 255]	0xC4 + 0xLL + данные	01 02 ... FF -> C4 FF 01 02 ... FF
[256; 65,535]	0xC5 + 0xLL 0xLL + данные	0001 0002 ... FFFF -> C5 FFFF 0001 0002 ... FFFF

Тип Array представляется в виде кортежа, т.к. может содержать данные разных типов, поэтому все составные типы, такие как InParams, OutParams, на уровне сериализации представляются в виде обычных структур с описанными полями.

Таблица 13. Правила сериализации типа Array

Размер данных (байт)	Бинарное представление	Пример
[0; 15]	b1001XXXX + данные, где четыре младших бита кодируют размер данных массива	01 02 ... 0F -> 9F 01 02 ... 0F
[16; 65,535]	0xDC + 0xLL 0xLL + данные	0001 0002 ... FFFF -> DC FFFF 0001 0002 ... FFFF

Под сериализацией структур и классов подразумевается последовательная агрегация их полей в кортеж (Array) и последующая сериализация. Порядок сериализации описывается в интерфейсе. Допускается сериализация вложенных структур. Принимающая сторона десериализует в известном порядке данные и восстанавливает объект. Таким образом,

принимающая сторона должна заранее знать тип, количество и порядок полей, из которых состоит структура, и быть готовой принять кортеж с заранее известными типами, соответствующими типам ожидаемой структуры.

Например, пусть дана структура MyType следующего вида:

```
struct MyType {
      I8 a;
      I8 b;
      Binary data;
};
```

Пусть a = 1, b = 2, data = ‘AABBCC’. В сериализованном виде структура будет представлена так:

93 D0 01 D0 02 C4 03 AA BB CC,

где 0x93 – байт, соответствующий типу массива (Array), 0xD0 – байт, описывающий тип I8, далее идёт параметр a = 0x01, далее снова описание типа I8 байтом 0xD0 и параметр b = 0x02, затем следует поле data, представленное в виде массива байт, который обозначен байтом 0xC4, затем идёт размер массива 0x03 и сами данные ‘AABBCC’.

Таким образом, итоговый пакет для передачи структуры MyType будет выглядеть следующим образом:

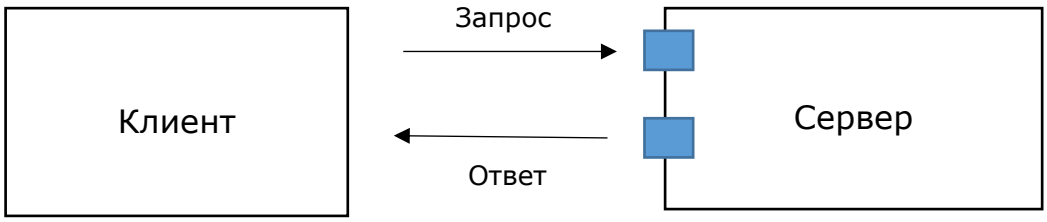
0001 0002 0003 0000000B 91 93 D0 01 D0 02 C4 03 AA BB CC,

где первые 10 байт – заголовок с байтами PKG_TYPE = 0001, MSG_ID = 0002, FUNC_ID = 0003, PARAMS_LEN == 0000000B, следующие 11 байт – это сами данные: первым идёт байт, который всегда описывает кортеж, в котором инкапсулированы входные данные, 0x91 (в этом примере входные данные представляют собой только структуру, т.е. всего 1 элемент), затем идут поля структуры, сериализация которых рассмотрена выше.

5. АРХИТЕКТУРА ТРАНСПОРТА

В терминологии TCP/IP далее по тексту сервером будет обозначаться узел, который предоставляет порт для установки соединения, а клиентом будет называться узел, который эти соединения запрашивает. Предполагается, что сервер при старте открывает 2 TCP-порта для каждого поддерживаемого им API. Подробности будут описаны ниже по тексту.

Рисунок 1. Упрощённая схема системы



Сервер может предоставлять сколь угодно API, поэтому количество портов в общем случае неограниченно. Ответственность за предоставление информации об открытых портах и предоставляемых интерфейсах лежит на интеграторе криптосервиса. В рамках каждого порта должно быть открыто два соединения. Соединения всегда запрашивает клиент, независимо от того является он пользователем или провайдером API. Каждое соединение имеет логическое направление передачи данных. С точки зрения пользователя, одно соединение должно быть использовано только для отправки запросов, а второе соединение только для получения ответов. С точки зрения провайдера, одно соединение должно быть использовано только для приёма запросов, а второе только для отправки ответов. Иными словами, одно соединение только для сообщений типа Request, другое соединение только для сообщений типа Response. Передача по одному сокету должна выполняться в полудуплексном синхронном режиме.

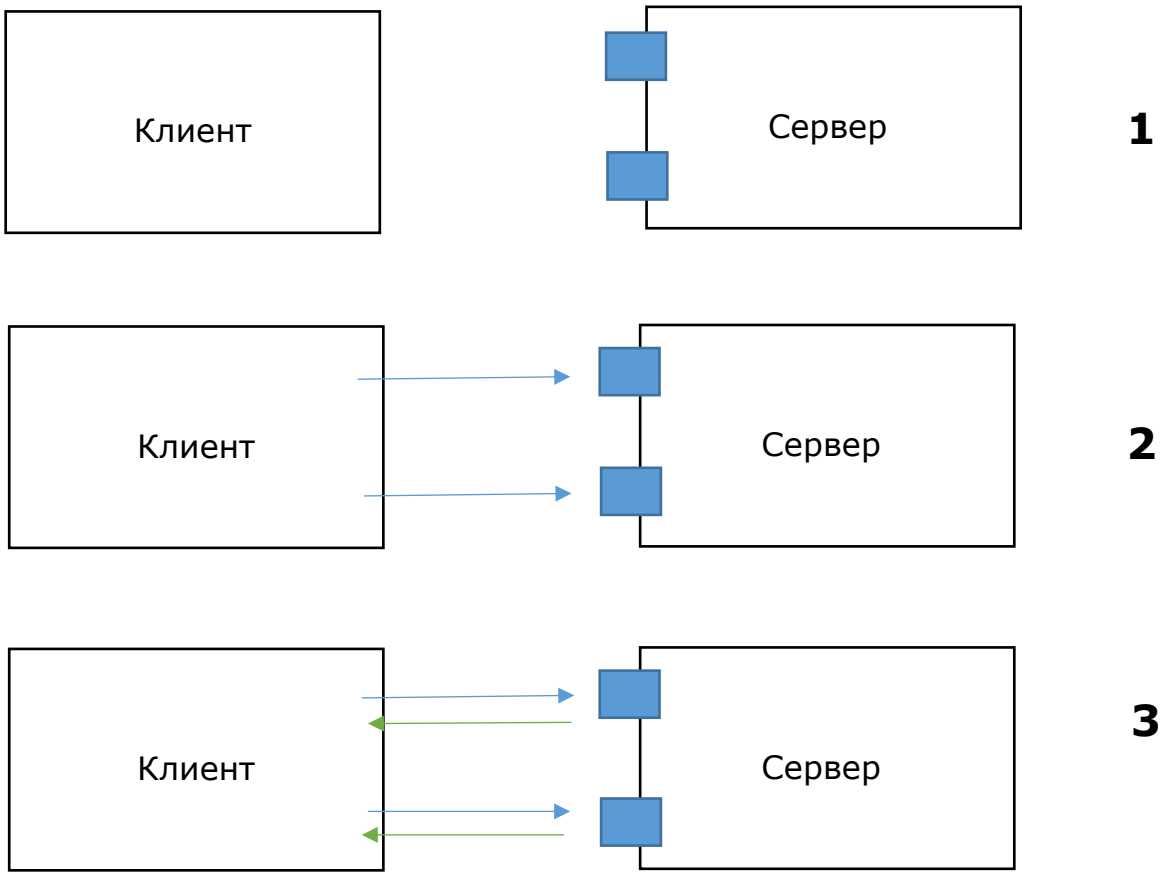
Направление сообщений определяется по чётности: нечётный порт является портом для сообщений типа Request, чётный порт для сообщений типа Response. Например, 127.0.0.1:10001 является портом для сообщений типа Request, а порт 127.0.0.1:10002 является портом для сообщений типа Response.

Передача данных происходит в асинхронном режиме, т.е. при завершении передачи данных, соединение освобождается для других потоков клиента, но не закрывается. Приём данных происходит в асинхронном режиме, т.е. при получении данных сервер производит обработку сообщения и выполняет бизнес-логику, при этом сокет на приём не блокируется и во время выполнения бизнес-логики допускается получение новых данных.

Сценарий открытия соединений с использованием двух API:

1) изначальное состояние системы после настройки и включения компонентов системы, соединения между сервером и клиентом отсутствуют;
2) клиент производит запрос на открытие двух соединений на каждый API;
3) если соединение санкционировано и коммуникационных ошибок не произошло, сервер открывает соединения.

Рисунок 2. Сценарий установки соединения



Синим прямоугольником изображено API

Детали реализации многопоточного взаимодействия клиентов с сервером выходят за рамки текущего документа. Если были соблюдены выше описанные требования и учтены особенности коммуницирующих систем, обе стороны смогут успешно синхронизировать конфигурации, установить все необходимые соединения и безошибочно взаимодействовать друг с другом.

6. ССЫЛКИ

[1]. https://ru.wikipedia.org/wiki/Удалённый_вызов_процедур
[2]. https://msgpack.org/
[3]. https://ru.wikipedia.org/wiki/TCP/IP
[4]. https://tools.ietf.org/html/rfc7049
[5]. http://bsonspec.org/spec.html
[6]. https://en.wikipedia.org/wiki/Interface_description_language
[7]. https://github.com/msgpack/msgpack/blob/master/spec.md
[8]. https://github.com/msgpack/msgpack-c
[9]. https://github.com/neuecc/MessagePack-CSharp