

Содержание

ВВЕДЕНИЕ.....	3
1 АНАЛИТИЧЕСКАЯ ЧАСТЬ.....	5
1.1 Описание области применения.....	5
1.2 Обзор существующих аналогов.....	6
1.3 Подходы к реализации ссылок в программировании.....	7
2 ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ПРОДУКТА.....	8
2.1 Принципы проектирования и реализации.....	8
2.2 Проектирование формата ссылки на удаленный объект.....	9
2.2.1 Развитие идентификатора объекта.....	11
2.3 Проектирование формата передачи сообщений.....	12
2.3.1 Дальнейшие изменения заголовка сообщения.....	14
2.4 Проектирование формата идентификатора запроса.....	15
2.5 Типы сообщений и их схемы данных.....	15
2.5.1 Будущие типы сообщений.....	17
3 РЕАЛИЗАЦИЯ ПРОЕКТА.....	19
3.1 Архитектура.....	19
3.2 Модули репозитория.....	20
3.3 Модули I/O операций.....	21
3.4 Модули исполнения запросов.....	25
3.5 Формат кодирования и его сериализатор.....	26
3.6 Кодогенерация.....	28
3.6.1 Возможности развития средств генерации текста.....	29
4 ТЕСТИРОВАНИЕ ПРОГРАММНОГО ПРОДУКТА.....	30
4.1 Основные понятия и принципы тестирования программного продукта.....	30
4.2 Функциональное демо-тестирование.....	31
4.3 Нагрузочное тестирование.....	32
5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ.....	36
5.1 Руководство по созданию сборки.....	36

5.2 Руководство по конфигурации узла.....	37
ЗАКЛЮЧЕНИЕ.....	39
СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ.....	40
ПРИЛОЖЕНИЕ А.....	
ПРИЛОЖЕНИЕ Б.....	

ВВЕДЕНИЕ

mROA — это современный фреймворк для внедрения объектно-ориентированного удаленного вызова процедур (RPC) в проекты, где требуется вносить контекст исполнения при вызове команд. Он спроектирован так, чтобы вносить наименьший вклад в написание кода, не связанного непосредственно с удаленным вызовом. Для удобства использования, удаленный вызов производится таким же образом, как и вызов обычного метода, через самый обычный метод декларируемый интерфейсом, который пишется непосредственно на C#. Таким образом действительно получается переносить вычисления и другие операции на удаленный сервер, который может иметь куда более другие условия, отличающиеся от условий выполнения кода на клиенте не приводя при этом видимых изменений в клиентский код.

Первоначально, данная задача возлагалась на gRPC, используемая через библиотеку `protobuf-net` [1], которая бы генерировала необходимый код для сервисов и сериализации сообщений вместо привычного `proto` файла. Однако, несмотря на похожее на схожесть с объектно-ориентированным подходом, через gRPC способна предавать только обычные структуры, но не ссылки на какие-либо объекты. Именно для решения этой задачи в первую очередь и был написан этот фреймворк, основной отличительной особенностью которого как раз и является передача, хранение и менеджмент таких ссылок на объекты, располагающиеся в куче в каком-либо рантайме .NET. Кроме того, в процессе разработки так же необходимо было решать задачи, дающие этой куда больше функционала, чем есть в большинстве популярных RPC, таких как обратный вызов или вызов событий, так же в две стороны. В целом получившийся в итоге фреймворк реализует все основные возможности языка программирования C# для использования их по сети.

Сам по себе этот фреймворк не ограничивает в области применения и может быть портирован на множество платформ для использования их особенностей. Однако основными направлениями, для которых изначально

предназначался этот фреймворк это обеспечение взаимодействия микросервисов на стороне бэкэнда, а так же взаимодействие между клиентами и сервером в таких средах как игра, написанная на Unity или какое-либо другое клиент-серверное приложение, например на AvaloniaUI. Именно архитектура одного проекта в Unity и то, как там используются объекты и побудили на создание этого фреймворка, чьи область применения и возможность стали куда больше в последних версиях.

Данный переход от обычных вызовов является довольно логичным и последовательным, как это происходило и в других областях программного обеспечения. Например PowerShell, в котором в отличие от его предшественников в качестве средства обмена информации между разными командами являются своеобразные объекты, имеющие свои методы [2]. Или же в принципе переход к объектно-ориентированному графическому интерфейсу, когда то, что видит пользователь состоит из различных объектов, у каждого из которых есть свой набор функций, который можно использовать у конкретного объекта. Однако надо понимать, что в целом на данный момент для большинства задач достаточно и обычного, не объектного, командного подхода к удаленным вызовам, однако такой подход может внедряться в экспериментальном порядке в различные системы и благодаря удобству реализации mROA уже сейчас может довольно просто и логично применяться в Web API, где при использовании правильного подхода и так уже используются сервисы, декларация которых по большей части совпадает с тем, как это возможно в реализованном фреймворке, что может способствовать его распространению.

1 АНАЛИТИЧЕСКАЯ ЧАСТЬ

1.1 Описание области применения

Изначально, этот фреймворк разрабатывался с целью упрощения взаимодействия уже объектно-ориентированной области — управление внутриигровыми предметами. Однако по мере разработки стало понятно, что подход к адаптации информационной системы не должен отличаться от конкретной области применения получившегося фреймворка.

Стоит уточнить, то благодаря использованию так называемых синглтонов, mROA может так же использоваться и в обычном, функциональном стиле, совместимом с REST API или сервисом gRPC, а не в объектно-ориентированном, но проектирование новых информационных систем с учетом объектно-ориентированного подхода к коммуникации между удаленными узлами может быть в некотором виде сложнее и непривычно, поскольку на данный момент такой подход не встречается массово и потребует перестройки, однако есть области, в которых данный подход значительно больше подходит, нежели процедурный стиль, например вышеупомянутый менеджмент предметами в инвентаре и игровом мире. На самом деле, в играх он может быть применен наиболее простым способом, как это уже частично реализуется даже при помощи уже готовых пакетов, хотя и не на таком уровне, как это реализовано в mROA. Но по крайней мере, объектно-ориентированный подход может использоваться хотя бы для автоматического переноса логики сервисов на удаленных узел в микросервисной архитектуре. Это очень легко можно реализовать при помощи внедрения зависимостей, который уже активно используются в backend разработке и являются стандартом индустрии уже продолжительное время как в новых проектах, так и в проектах, создававшихся во времена монолитной архитектуры. При внедрении зависимостей сервисы уже представлены некими объектами, который представляются, обычно, интерфейсом и могут быть реализованы прокси-объектами, вместо обычных классов с логикой внутри, реализующих некоторый интерфейс.

Хотя данный подход к коммуникации между микросервисами не использует полные возможности библиотеки, но позволяет частично внедрить фреймворк в проект для дальнейшего его использования и в других областях, кроме непосредственного.

Резюмируя всю информацию о ожидаемом применении и use-case, можно выделить функции и возможности, которые необходимо реализовать:

- обычный прямой вызов (унарный вызов);
- реверсивный вызов;
- события;
- автоматическое восстановление соединения без повторной регистрации;
- передачу контекста для конкретного вызова;
- стандартные объекты синглтоны;
- поддержка различных версий клиента и сервера;
- отмена ожидания результата;
- возможность отправлять сообщения без подтверждения получения.

1.2 Обзор существующих аналогов

Конечно же, сама конвенция RPC далеко не нова и данной области существуют уже множество популярных решений, таких как gRPC или например SOAP, но оказывается и в контекстном удаленном вызове уже так же существуют свои фреймворки.

Наверное самым похожим фреймворком на тот, каким является mROA является RPyC, который работает по схожей идеологии, позволяет так же выполнять функции задаваемые клиентом непосредственно в рантайме самого клиента, использует прокси-объекты [3]. Однако этот фреймворк довольно сильно привязан к языку программирования Python и его может быть сложно перенести на других платформах, в отличии от mROA, хотя он и заявляется как фреймворк для Python судя по его названию. По сравнению с ним mROA использует куда более обобщенный протокол, позволяющий реализовать его на

куда большем числе языков, как это например уже доступно у gRPC. При переносе на различные языки протокол не может ограничивать реализацию, а сама реализация будет зависеть исключительно от возможностей самого языка в области обобщенного программирования и оперирования объектами с не ясным при компиляции типом. Это, наверное, будет являться самой сложной задачей, поскольку остальная часть реализации бэкенда mROA не столь сложна.

Если рассматривать саму структуру протокола, использующегося в mROA, то можно найти некие сходства с протоколом SOAP (Simple Object Access Protocol) в части структуры и последовательности передачи параметров метода и его идентификатора. Однако реализация протокола никак не опирается на SOAP или другие технологии, как например gRPC, и является оригинальной, привносящей свои идеи, принципы и возможности.

1.3 Подходы к реализации ссылок в программировании

Первоначально, в языках программирования не было никаких «ссылок» на объекты. Неким прародителем современных ссылок можно назвать указатели, которые являются прямым адресом в памяти, по которому хранится нужный объект. Однако в современных высокоуровневых языках, таких как C#, Golang или Java, обычно, не используются такие прямые указатели. С использованием сборщиков мусора, хотя в C#, Golang и других языках и возможно писать код опираясь на самые простые атомарные указатели.

Использование сборщиков мусора вынуждает предоставлять программисту не «сырой» указатель на объект в куче, а какой-то управляемый сборщиком мусора объект, который когда это необходимо в программе может представлять доступ к соответствующему объекту, как это было бы с указателем. Это обусловлено тем, что объекты в управляемой куче (managed heap) могут перемещаться в памяти в процессе сборки мусора для избегания фрагментации памяти и оптимизации дальнейших аллокаций в куче. Кроме того, ссылка так же содержит данные о типе и его методах. Это можно увидеть в исходных файлах реализации CoreCLR [4]. Похожий принцип используется и

в Java, хотя конкретный алгоритм сборки мусора и может варьироваться в зависимости от сборщика мусора, который в Java можно устанавливать.

Но кроме такого подхода к управлению объектами в памяти так же есть языки программирования, не использующие «обернутые» указатели. Например в таких языках как Golang или C++ сборщики мусора или вовсе отсутствуют, или не перемещают объекты в куче. Но и в таких языках пользователю обычно не предоставляется архитектурозависимый числовой указатель, который можно было бы куда либо записать как число, но по своей сути им являющимся.

Анализируя различные подходы к реализации ссылок, наиболее удобным для реализации ссылок на удаленные объекты на различных узлах можно считать подход, используемый в языках программирования без автоматического перемещения ссылок в памяти, где ссылка, хоть и как то скрыто, представляет собой указатель в памяти. Такой подход минимизирует количество запросов для синхронизации этих самых ссылок на различных узлах, хотя и может задействовать больше памяти для управления этими сетевыми ссылками.

2 ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ПРОДУКТА

2.1 Принципы проектирования и реализации

Для согласованности уже написанного кода и его расширения как в глубину так и в ширь в едином «духе», можно вывести некоторые принципы которые применимы к разработке mROA:

1. Ненавязчивость. Фреймворк должен решать одну задачу — перенос выполнения методов на удалённый узел — и делать это так, чтобы пользователь ощущал минимальное вмешательство в свою работу. Он не должен требовать изменения структуры кода, наследования от специальных классов или ручного управления вызовами. При этом, если разработчику нужна информация, связанная с работой фреймворка — например, контекст вызова — она должна быть доступна простым и понятным способом.

2. Конфигурируемость. Фреймворк построен из минимально связанных модулей, поведение которых можно изменять через внешнюю конфигурацию,

зависимости или стратегии, не изменяя их исходный код. Модули не зависят от контекста использования и работают одинаково в разных конфигурациях и приложениях.

3. Производительность. Удалённые вызовы по своей природе связаны с накладными расходами — сериализацией, сетевой передачей, управлением соединениями. Однако эти издержки должны быть сведены к минимуму, чтобы не оказывать значительного влияния на общую производительность приложения. Это соответствует принципу ненавязчивости: чем меньше фреймворк "замедляет" код, тем прозрачнее его работа.

4. Кодогенерация. Любой однотипный код, необходимый для работы фреймворка, должен генерироваться автоматически. Программист не должен писать вручную прокси, сериализаторы, адаптеры или клиентские заглушки. Эта ответственность лежит на самом фреймворке. Принцип вытекает из ненавязчивости: чем меньше пользователь пишет кода, специфичного для инфраструктуры, тем чище остаётся его бизнес-логика.

Эти 4 принципа по широко используются в фреймворке и позволяют программисту получить лучший пользовательский опыт, чем мог бы быть, если бы они не использовались, например если бы структура была бы монолитной и пользователю самому нужно было описывать прокси-объекты (далее — ПО).

2.2 Проектирование формата ссылки на удаленный объект.

Уникальный адрес объекта, доступного по сети — Complex object identifier (COI, составной идентификатор объекта), состоит на данный момент из 3 частей — сторона владения (OwnershipSide), владелец (Ownership), индекс объекта (ObjectIndex).

Обычно, для реализации ссылок или указателей в языках программирования используется числовой тип без знака максимального размера, возможного для архитектуры процессора. Это необходимо для реализации возможности хранить наибольшее возможное количество объектов в куче. Для 32-х битной архитектуры ограничением является около 4 гигабайт

оперативной памяти, а для современной 64-х битной архитектуры максимальный объем практически неограничен. Но на самом деле для большинства задач вполне достаточно и 32 бит. В случае с управлением удаленными объектами, необходимо хранить не их физическое расположение в памяти какого либо узла, а скорее индекс в некоем полностью управляемом фреймворком репозитории всех управляемых удаленных объектов. Таким образом, более 2 миллиардов объектов для конкретного хоста вполне достаточно в рамках взаимодействия между хостами, ведь обычно, таких объектов используется не так уж и много, чтобы использовать даже 1% от предоставляемых адресных пространств. Хранения даже 31 бит уже будет избыточно и хватит для любого разумного использования библиотеки. Так же стоит уточнить, что в этом индексе для идентификации хранящихся объектов используется всего 31 бит. Первый бит из 32 обозначает, является ли объект синглтоном. Синглтон — это универсальный единственный объект который по умолчанию есть в каждом репозитории. На каждый реализуемый интерфейс может быть всего один синглтон.

Кроме индекса объекта так же хранится идентификатор пользователя, в чьем репозитории хранится объект. На этот идентификатор выделяется 31 бит. Этот показатель тоже является избыточным и не создаст каких либо ограничений для использования фреймворка.

Последней, но не менее важной частью этого составного идентификатора является сторона владения. В текущем стандарте этот бит говорит о том, на какой стороне хранится конечный объект, предоставляющий логику. В случае если он равен 1, это означает что конечный объект хранится на узле, выдающем всем подключившимся узлам их идентификатор. Иначе, он будет храниться на стороне самого удаленного узла. По этому биту легко определить, где конкретно хранится объект.

Complex object identifier layout

Name	Offset	Length
Ownership side	0	1
Ownership	1	31
Index	32	32

Рисунок 2.1 Схема составного идентификатора объекта

2.2.1 Развитие идентификатора объекта

Одним из возможностей для развития является уменьшение размера идентификатора, но таким образом чтобы не создавать значимых ограничений.

Как уже было сказано, две основные значимые части идентификатора являются сильно избыточными. Можно уменьшить размер индекса до 16 бит, что позволит оперировать более чем 65 тысячами объектов, а размер идентификатора владельца уменьшить до 15 бит, что даст возможность работать системе из более чем 32 тысяч узлов (Рисунок 2.2). В реальных условиях использования, количество узлов можно уменьшить еще меньше, но для сохранения выравнивания идентификатора дальнейшее уменьшение размера нецелесообразно.

Complex object identifier layout

Name	Offset	Length
Ownership side	0	1
Ownership	1	15
Index	16	16

Рисунок 2.2 Схема экспериментального составного идентификатора 1

Другим путем развития идентификатора потребует сохранения его размера и изменения его структуры. Возможно, сделать его не просто инструментом для определения объекта по простому числовому индексу, а через некоторый дескриптор, который будет преобразовывать биты, которые

сейчас используются для индекса, в объект, определяемый этими битами логически. Для этого, вероятно, желательно уменьшить размер идентификатора владельца до 14 бит, добавить бит, является ли идентификатор индексным или логическим, в случае если индексным, то следующие 32 бита будут использоваться как индекс. Иначе 48 бит будут использованы как некоторое бинарное представление объекта, которое на стороне, где он расположен должно как либо логически преобразовываться. Например можно передавать какую либо структуру в бинарном формате или напрямую через маршалинг на удаленный хост, где она имеет свою логику внутри (Рисунок 2.3).

Complex object identifier layout

Name	Offset	Length
Ownership side	0	1
Ownership	1	14
Index type	15	1
Index	16	32
Representation	16	48

Рисунок 2.3 Схема экспериментального составного идентификатора 2

Эти два подхода являются частично несовместимыми, хотя и теоретически могут использоваться в различных конфигурациях при небольших доработках текущей версии внутренней архитектуры и абстрагировании от конкретной реализации идентификатора.

2.3 Проектирование формата передачи сообщений

Проведя анализ различных RPC, можно сказать, что нет определенного общего пути к решению вопроса передачи данных между конечными точками. Разные фреймворки используют различные протоколы. Например gRPC работает поверх HTTP/2, а ZeroMQ используется свой трехстадийный протокол обмена сообщениями [5].

Для достижения максимальной гибкости, свободы в использовании и независимости от сторонних стандартов был разработан протокол, позволяющий быстро и эффективно передавать сообщения минимизируя накладные расходы. Однако стоит оговориться, что, благодаря удачной внутренней архитектуре и абстрагированию от конкретного способа передачи сообщений, работоспособность всей системы не зависит от конкретного протокола.

В качестве основного протокола передачи данных был выбран TCP за его легковесность, надежность проверенную десятилетиями и простоту реализации. Но стоит учитывать, что из-за таких особенностей работы протокола как фрагментация необходимо использовать методы, позволяющие нивелировать этот механизм протокола TCP при чтении. В протоколах, которые не разбивают сообщения на несколько частей можно минимизировать размер заголовка, не передавая длину сообщения.

Протокол передачи сообщений, разработанный и используемый на данный момент как основной называется mROAMP (mROA Messaging Protocol). Он является полностью бинарным протоколом и состоит из двух частей — заголовка (header), описывающего необходимые метаданные и тела (body), сами передаваемые данные.

Заголовок состоит из 19 байт (Рисунок 2.4). Первые 16 байт являются уникальным идентификатором запроса. Он может повторяться для разных сообщений и обозначает группу, к которой относится это сообщение. Следующие два бита указывают на длину тела сообщения. Этот протокол поддерживает сообщения длиной не более 65536 байт, однако типичный размер передаваемых сообщения может быть меньше и 128 байт из-за использования CBOR и того, как реализовано кодирование объектов в штатном сериализаторе. Последний бит является типом сообщения. Всего на данный момент 11 типов сообщений.

mROA Message Protocol header layout

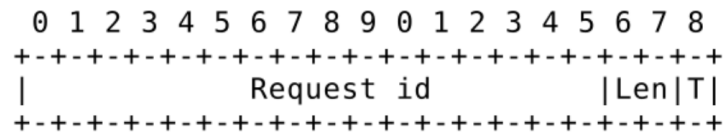


Рисунок 2.4 Схема заголовка сообщения mROAMP

После заголовка, определяющие метаданные сообщения следует само тело с определенной заголовком длиной. Конец тела так же является и концом всего пересылаемого сообщения.

Само тело закодировано в формат, используемый в систему. Система не ограничивает в использовании форматов как бинарных, так и текстовых, однако рекомендуется использовать бинарные форматы для уменьшения длины тела и ускорения парсинга. На данный момент основным форматом передачи данных является CBOR (RFC 8949). Для сериализации и десериализации из за особенностей некоторых конфигураций необходимо реализовать собственную версию сериализатора с использование контекста конечной точки.

2.3.1 Дальнейшие изменения заголовка сообщения

На данный момент такого заголовка, состоящего из 3 частей достаточно, для реализации всех требований к фреймворку, однако в будущем, для расширения возможностей без постоянных изменений схемы заголовка необходимо будет добавить еще одну часть: дополнительную мета информацию, как это например реализовано в http (headers), которая будет идти перед телом сообщения. Для этого планируется изменить формат записи длины сообщения и уменьшение максимального размера сообщения, т.к. оно на данный момент избыточно. Типичный размер сообщения запроса удаленного вызова может быть даже до 128 байт в зависимости от размера параметров. Для обычного вызова, где параметр — это одно целое число, размер тела даже меньше 64 байт при кодировании в CBOR. На данный момент предполагается использовать первый бит длины тела сообщения как индикатор, что вообще есть какие либо дополнительные заголовки. Если этот бит равен единице, то после последнего бита идентификатора типа сообщения будет идти два байта,

содержащие длину заголовочной части, а затем и сами заголовки указанной длины. Такой способ расширения возможностей совместим с текущим способом передачи сообщения.

Так же для идентификации схемы заголовка можно выделить 1 байт после идентификатора типа сообщения, чтобы можно было однозначно трактовать заголовок.

2.4 Проектирование формата идентификатора запроса

Каждое сообщение, пересылаемое между узлами имеет уникальный идентификатор `RequestId`. Он представляет собой последовательность из 128 бит, генерируемых случайно на стороне иницилирующего сообщения. На один и тот же идентификатор `RequestId` может приходиться несколько сообщений, относящихся к одному иницилирующему сообщению.

Изначально для этой цели использовался GUID, однако, для оптимизации кодирования было принято разработать альтернативный уникальный идентификатор. Из-за особенностей реализации генерации случайных чисел в различных системах, время на генерацию нового GUID в Linux системах значительно медленнее, чем это происходит в Windows [6]. Это было учтено при проектировании и реализации уникального идентификатора `RequestId`.

В сравнение со стандартной реализацией GUID в dotnet 9, генерация реализованного уникального идентификатора по результатам BenchmarkDotNet быстрее в более чем 48 раз. Так же степень уникальности выше, за счет использования всех 128 бит для уникальных данных, поскольку в оригинальном стандарте GUID не все биты случайны [7].

2.5 Типы сообщений и их схемы данных

Как говорилось выше, всего на данный момент используется 10 типов сообщений. Они отличаются своей внутренней структурой. 9 из них кроме своего типа так же содержат дополнительные описывающие их значимые данные.

1. Сообщение `FinishedCommandExecution` (идентификатор 1) сигнализирует об успешном завершении удалённого вызова. Оно содержит идентификатор запроса `RequestId` и, при наличии, результат выполнения метода. Результат может отсутствовать, если вызываемый метод не возвращает значения. Сообщение используется для передачи результата вызывающему узлу в асинхронном режиме.

2. Сообщение `ExceptionCommandExecution` (идентификатор 2) указывает на аварийное завершение выполнения вследствие возникновения исключения. Включает в себя `RequestId` и строковое представление исключения, включающее его тип, сообщение и стек вызовов. Длина сообщения не ограничена, что позволяет передавать полную диагностическую информацию. Используется для прозрачной передачи ошибок между узлами.

3. Сообщение `CallRequest` (идентификатор 3) инициирует удалённый вызов метода и состоит из четырёх полей: `RequestId`, `CommandId`, `ObjectId` и `Parameters`. Первые три поля обязательны и не могут быть `null`. `RequestId` — идентификатор запроса; `CommandId` — 32-битный знаковый индекс метода на целевом узле, который должен быть согласован между узлами при различающихся версиях API (вне зависимости от конфигураций API идентификатор -1 зарезервирован под метод `Dispose`); `ObjectId` — идентификатор объекта в формате `ComplexObjectIdentifier` (COI); `Parameters` — массив передаваемых аргументов, который может быть пустым, но не `null`. Параметры `CancellationToken` и `RequestContext` не включаются в сообщение и создаются на стороне получателя.

4. Сообщение `IdAssigning` (идентификатор 4) передаётся в ответ на подключение узла к системе распределённой идентификации и содержит два поля: `Id` — 32-битный знаковый идентификатор узла, и `IndexConfig` — список конфигураций индексов методов, доступных на данном узле. Является первым сообщением, получаемым узлом после регистрации.

5. Сообщение `CancelRequest` (идентификатор 5) используется для отмены ранее отправленного запроса и содержит единственный параметр — `RequestId`, идентифицирующий отменяемый вызов. Предназначено для поддержки отмены асинхронных операций.

6. Сообщение `EventRequest` (идентификатор 6) предназначено для вызова методов без ожидания результата, например, при реализации событийной модели. Его структура полностью совпадает со структурой `CallRequest`, однако ответ на такое сообщение не ожидается.

7. Сообщение `ClientRecovery` (идентификатор 7) применяется при восстановлении соединения после обрыва без создания новой сессии. Имеет ту же структуру, что и `IdAssigning`, и позволяет возобновить работу с сохранением текущего состояния идентификаторов и конфигурации.

8. Сообщение `ClientConnect` (идентификатор 8) является первым сообщением, отправляемым узлом при установлении соединения, и инициирует процесс выдачи идентификатора. Содержит поле `Config` — массив строк, интерпретируемых как параметры подключения. На текущий момент конфигурация не используется, но поле зарезервировано для будущих расширений, аналогично CLI-аргументам.

9. Сообщение `UntrustedConnect` (идентификатор 10) используется для инициализации соединения без гарантии доставки сообщений. Его структура идентична `IdAssigning`. Предназначено для сценариев, где важна производительность, но не требуется надёжная доставка, например, при потоковой передаче событий.

10. Сообщение `ClientDisconnect` (идентификатор 9) не содержит дополнительных данных и сигнализирует о штатном завершении сессии клиентом. Является единственным типом сообщения, не имеющим полезной нагрузки.

2.5.1 Будущие типы сообщений

Хотя текущих 10 типов сообщений и достаточно, но уже планируются новые типы сообщений.

Первый планируемый тип сообщения — Setup, позволяет динамически конфигурировать параметры соединения, задаваемые в ClientConnect.

Последним из планируемым на данный момент типом сообщения является ForwardCallReqeust. Он выполняет функции вызова метода на удаленном узле, но не регистрирует клиента, в отличие от ClientConnect, а затем CallRequest. Этот тип сообщения нужен для создания stateless соединения, не требующего постоянного подключения, похожего на http. Все запросы ForwardCallRequest (FCR) могут выполняться только синглтонами, то есть в нем, в отличие от CallRequest будет отсутствовать ObjectId. У такого типа сообщений есть как преимущества, так и недостатки.

Из преимуществ можно выделить нулевое время на конфигурацию соединения, поскольку после подключения нового клиента запрос будет сразу маршрутизироваться. Однако в связи с особенностями маршрутизации запросов по индексам удаленного узла непосредственно, отсутствие какой либо конфигурации несет с собой риски по несогласованности конфигураций сборок на сервере и клиенте, однако этот недостаток может быть нивелирован, если стабилизировать конфигурацию сервера и добавлять новые сборки в конец текущих индексов.

3 РЕАЛИЗАЦИЯ ПРОЕКТА

Описанный выше протокол взаимодействия между узлами в полной мере позволяет реализовать двусторонний объектный вызов и другие возможности, необходимые для современных систем.

Однако реализация этого протокола является куда более сложной задачей в области архитектуры ПО в целом и конкретных модулей в частности. Необходимо было оптимизировать выполнение самих команд с учетом современных подходов к асинхронности.

3.1 Архитектура

Изначально поставленные требования перед библиотекой составляли лишь удаленный вызов, похожий на gRPC и исполнение запросов с учетом контекста объекта. Для этого была создана первоначальная простая архитектура.

Рассматривая задачи, которые необходимо решить создаваемому фреймворку можно выделить две глобальные задачи: менеджмент состояний и исполнение команд, а так же общение узлом друг с другом.

С развитием фреймворка и ростом числа требований к системе архитектура менялась. Первый набросок фреймворка имел асимметричную архитектуру. Это было сделано для наиболее оптимального решения поставленных задач. Однако, следующим этапом после начальной реализации фреймворка потребовалось реализовать обратный вызов (reverse call). Уже для этого потребовалось изменить архитектуру на симметричную. Это изменение на начальном этапе развития фреймворка стало базой для всех новых возможностей и очертило структуру и общий вид модулей до последней версии. Но стоит подметить, что большие изменения претерпела именно часть, связанная с обменом информацией между узлами, а оставшаяся часть сохранила свой интерфейс или незначительно изменило его.

Это связано с тем, что в сетевом программировании существует ограничение на трансляции одного сетевого пакета для нескольких

потребителей. Передача пакета обычно происходит в порядке очереди запросов от потребителей. Для нивелирования этого эффекта необходимо было создать более высокоуровневый API, позволяющий нескольким конечным точкам запрашивать данные одновременно.

3.2 Модули репозитория

При разработке среды для обеспечения исполнения необходимо было реализовать хранение и доступ к некому множеству различных объектов. Для этой задачи был выбран паттерн Репозиторий.

Паттерн репозиторий — это шаблон проектирования, предназначенный для абстрагирования от конкретного метода хранения данных и предоставления лишь общего интерфейса для доступа к хранящимся данным [8].

Всего в последней версии архитектуры mROA присутствует 7 разных модулей, которые можно считать реализацией паттерна репозиторий:

- ConnectionHub;
- HubRequestExtractor;
- InstanceRepository;
- MultiClientInstanceRepository;
- CancellationRepository;
- CollectableMethodRepository;
- RemoteInstanceRepository.

Эти модули хранят в себе один тип объектов и представляют API для работы с содержащимися в них данными, специфическое для каждого типа данных. Стоит уточнить, что InstanceRepository, MultiClientInstanceRepository и RemoteInstanceRepository предоставляют общий интерфейс, но используются в разных случаях, хотя первые два и могут быть взаимозаменяемы логически.

Не смотря на то, что все эти модули выполняют схожие функции, они реализованы по разному:

- репозитории ConnectionHub, HubRequestExtractor, MultiClientInstanceRepository, CancellationRepository работают по принципу

словаря, то есть значение ключа хэшируется. Поскольку использование `CancellationRepository` подразумевает многопоточную работу, в нем используется `System.Collections.Concurrent.ConcurrentDictionary`, вместо обычного `System.Collections.Generic.Dictionary`.

– репозитории `InstanceRepository`, `CollectableMethodRepository` (Рисунок 3.1), в отличие от вышеперечисленных, хранят значения в сплошном массиве, а доступ к элементам происходит напрямую по индексам. Это значительно ускоряет операции над элементами, предоставляемые этими репозиториями. Для ускорения размещения объектов в `InstanceRepository` применяется связанный список из свободных мест, поэтому время добавления нового элемента константное, в случае если есть свободное место, но растет при увеличении размера хранилища.

```
public class CollectableMethodRepository : IMethodRepository
{
    private readonly List<IMethodInvoker> _methods = new() { MethodInvoker.Dispose };
    private IMethodInvoker[] _baked = Array.Empty<IMethodInvoker>();
    public void AppendInvokers(IEnumerable<IMethodInvoker> methodInvokers)
    {
        _methods.AddRange(methodInvokers);
        _baked = _methods.ToArray();
    }
    public IMethodInvoker GetMethod(int id)
    {
        return _baked[++id];
    }
}
```

Рисунок 3.1 Реализация модуля `CollectableMethodRepository`

– `RemoteInstanceRepository` предоставляет реализацию генерации еще не объявленных прокси-объектов и поиск среди уже сгенерированных.

Модули репозиторий являются самыми одними из самых простых с точки зрения реализации, но они используются в «горячем пути» и они должны быть максимально оптимизированы.

3.3 Модули I/O операций

Как говорилось выше, модули взаимодействия с другими узлами сильно изменились сразу после первой рабочей версии. Необходимо было создать одинаковый как для клиента, так и для сервера высокоуровневый

многопоточный интерфейс взаимодействия между узлами в сети. В итоге получился трехуровневый высокоуровневый query-like интерфейс.

В обычном случае за это API отвечает 3 модуля: StreamExtractor, ChannelInteractionModule и RepresentationModule. Они все работают на разных уровнях абстракции и позволяют гибко получать доступ к данным одного сокета из множества потоков.

StreamExtractor является самым низкоуровневым модулем и ближе всего к непосредственному чтению и записи в сокеты (рисунок 3.2). Он читает все входящие пакеты из соединения, распознает заголовки и создает обертку над входящим запросом — NetworkMessage. NetworkMessage содержит в себе 3 поля: Id, MessageType и Data. Поле Data является массивом из не преобразованных данных. После чтения сообщения, StreamExtractor передает полученный NetworkMessage в DistributionModule, который отвечает за дальнейшее распространение полученного сообщения.

```
public class StreamExtractor
{
    private const int BufferSize = ushort.MaxValue + 19;
    private readonly Stream _ioStream;
    private readonly Memory<byte> _buffer = new byte[BufferSize];
    [2] 3 usages  Mikhail Mitrofanov
    public StreamExtractor(Stream ioStream){...}
    public Action<NetworkMessage> MessageReceived = _ => {};
    [2] 3 usages  Mikhail Mitrofanov
    public async Task SingleReceive(CancellationToken token = default)
    {
        var firstRead :int = await _ioStream.ReadAsync(_buffer, token);

        var meta = MemoryMarshal.Read<NetworkMessage.NetworkMessageMeta>(_buffer.Span);

        var len :ushort = meta.BodyLength;
        var readLen :int = firstRead - 19;

        if (readLen != len)
        {
            var lastPart :Memory<byte> = _buffer[firstRead..(len + 19)];
            await _ioStream.ReadExactlyAsync(lastPart, cancellationToken: token);
        }

        var message = meta.ToMessage(_buffer.Span);

        MessageReceived(message);
    }
    [2] 4 usages  Mikhail Mitrofanov
    public async Task LoopedReceive(CancellationToken token = default){...}
    [2] 1 usage  Mikhail Mitrofanov
    private async Task Send(NetworkMessage message, CancellationToken token = default){...}
    [2] 3 usages  Mikhail Mitrofanov
    public async Task SendFromChannel(ChannelReader<NetworkMessage> channel,
        CancellationToken token = default){...}
    [2] 5 usages  Mikhail Mitrofanov
    public bool IsConnected => _ioStream is { CanRead: true, CanWrite: true };
}
```

Рисунок 3.2 Фрагмент модуля StreamExtractor

Всего есть два разных модуля распространения: ChannelDistributionModule и ExtractorFirstDistributionModule. Эти два модуля различаются внутренней логикой обработки сообщения для определения дальнейшего потребителя.

Самым простым модулем, который использовался большую часть времени является ChannelDistributionModule. Он передает все сообщения в канал входящих сообщений, предоставляемый ChannelInteractionModule.

ChannelInteractionModule является промежуточным слоем в query-like API, который предоставляет один канал для входящих, и два канала для исходящих сообщений: доверенный и недоверяемый. Каналы, которые он предоставляет являются высокопроизводительной реализацией модели производителя и потребителя [9]. Так же этот модуль обрабатывает аварийные ситуации. В случае с отправкой в разорванное соединение он отправляет в свой канал доверенных сообщений запрос на переподключение и ожидает нормализации соединения. Все сообщения, переданные в канал доверенных исходящих сообщений читает StreamExtractor и отправляет в соединение, за которое он ответственен.

После попадания в канал входящих сообщений, это сообщение читает RepresentationModule, который является вершиной этого API. При запрашивании у этого модуля какого либо сообщения, он начинает ожидать поступления в канал входящих сообщений, проверяет на соответствие сообщения с переданным потребителем правилом, и в случае с совпадением с правилом преобразовывает его в необходимый тип, так же определяемый потребителем. RepresentationModule реализует два способа доступа к данным: одиночный и потоковый. В случае с использованием одиночного запроса, при нахождении необходимого сообщения это сообщение и вернется. При использовании потокового исполнения, возвращается Channel, который уже предоставляет исключительно необходимые сообщения без ограничений на количество.

За установление соединения отвечают два класса: `NetworkGatewayModule` для узла, слушающего соединения и `NetworkFrontendBridge` для узла, подключающегося к другому узлу. Эти модули, как и `StreamExtractor`, работают на данный момент для TCP подключений. Для UDP и `UnityTransport` написаны дополнительные модули, реализующие передачу данных по соответствующим протоколам.

`NetworkGatewayModule` используется для прослушивания подключений. Он инициализирует сокеты, создает `StreamExtractor`, `ChannelInteractionModule` и `RepresentationModule`. При подключении нового соединения в зависимости от первого сообщения или создает новую сессию, или восстанавливает прерванное соединение.

`NetworkFrontendBridge` нужен для подключения к удаленному узлу. В отличие от `NetworkGatewayModule` он не создает новые модули, а использует уже существующие.

Вышеописанные два модуля предназначены для передачи доверенных сообщений. Для передачи сообщений без подтверждения по недоверяемому каналу используются другие два модуля: `UdpGateway` и `UdpUntrustedInteraction`. Они, как следует из названий, используют протокол UDP и не вводят дополнительных мер надежности доставки, которые реализованы в TCP или QUIC, в следствие чего могут не всегда доставлять сообщения до требуемого узла. Так же они ограничены в возможностях по инициации сессии — сессию на данный момент можно создать только в модулях, реализующие доверенное подключение.

Для поддержки нативного подключения в Unity также написаны два модуля, реализующие сразу и доверенный, и недоверяемый каналы по протоколу пакета `UnityTransport`, являющимся базовым протоколом для всех сетевых функций присутствующих в Unity: `Netcode for GameObjects` и `Netcode for Entities`. Эти модули называются `UTGateway` и `UTFrontendBridge`. Из-за особенностей работы аллокатора Unity и других аспектов его работы, работа с

модулями отличается от обычного использования `NetworkGatewayModule` или других модулей из основного пакета:

- начало прослушивания подключений должно происходить в основном потоке `Unity`. При использовании его внутри асинхронного метода будет выкинуто исключение;
- чтение входящих сообщений и отправка исходящих, а так же обработка подключений так же должна происходить в основном потоке `Unity`, например в методе `Update`.

Эти правила применимы к обоим I/O модулям, использующим `UnityTransport`.

3.4 Модули исполнения запросов

Последней частью фреймворка является блок исполнения входящих запросов. К модулям, отвечающим за исполнение можно отнести `BasicExecutionModule`, `HubRequestExtractor` и `RequestExtractor`.

Модуль `BasicExecutionModule` занимается над непосредственным управлением запуска метода и отправкой результатов удаленному узлу. `BasicExecutionModule` может быть использован на любом узле, в отличие от сетевых модулей. Он находит требуемый в запросе инвокер, экземпляр объекта, приводит параметры к требуемому инвокером типам, вызывает инвокер и обрабатывает результат метода. Инвокеры в модуль выполнения поставляется через `CollectableMethodRepository`, экземпляр класса поставляется `InstanceRepository` или `MultiClientInstanceRepository`. На один узел, вне зависимости, является ли он «сервером» или «клиентом», нужен всего один `BasicExecutionModule`.

`BasicExecutionModule` не может сам находить запросы на исполнение. Его API используется в `RequestExtractor`, который создает постоянный запрос у `RepresentationModule`. Однако такой запрос создается только в случае, если используется `ChannelDistributionModule` и запросы поступают из общего для всей сессии потока сообщений. Но есть более оптимальный вариант,

минимизирующий накладные расходы на асинхронное ожидание получения нового сообщения: `ExtractorFirstDistributionModule`. Он уже на уровне распространения сообщения проверяет, подходит ли оно для исполнения и напрямую вызывает `RequestExtractor`.

На подключающемся узле, поддерживающем исполнение запросов относящимся только к своим данным необходимо использовать `RequestExtractor`, однако для возможности исполнения в контекстах нескольких удаленных узлов необходим `HubRequestExtractor`, создающий на каждую сессию свой `RequestExtractor` со своим контекстом.

3.5 Формат кодирования и его сериализатор

Как говорилось выше, основным рекомендуемым форматом кодирования объектов является CBOR. У этого формата есть ряд преимуществ, за которые он и был выбран на роль формата передачи данных:

- компактность. Благодаря тому, что CBOR является бинарным форматом, объекты, кодируемые им занимают меньше места, чем закодированные в JSON или XML, использующиеся в JSON-RPC или SOAP и других RPC соответственно;

- поддержка Microsoft. У корпорации Microsoft есть свой низкоуровневый оптимизированный пакет для кодирования и декодирования данных CBOR, имеющую широкую поддержку различных версий .NET;

- производительность. Из за бинарной природы этого формата, для кодирования необходимо меньше служебных токенов, таких как закрывающие скобки в JSON.

- поддержка кодирования множества типов значений. В CBOR доступны для кодирования множество различных типов данных нативно, даже больше, чем в JSON, что позволяет эффективно их кодировать.

Таким образом CBOR является подходящим форматом кодирования, позволяющий в полной мере сделать фреймворк эффективным и производительным. Как известно, формат передачи данных по сети имеет

ключевую роль в производительности такой системы и позволяет минимизировать нагрузку на высоконагруженные сервера, задержку ответа и время на передачу информации [10]. Так же в отличие от ProtoBuf, CBOR не требует какой либо схемы, определенной через сам формат, что и не требуется.

Единственным конкурентом CBOR является формат MessagePacks, который предоставляет большую производительность для типов, которые он может кодировать, но меньшую для тех, которые может кодировать CBOR.

Однако из-за того, что пакет System.Formats.Cbor, разработанный корпорацией Microsoft предоставляет низкоуровневый доступ к записи и чтению на уровне базовых типов, необходимо написать столь же производительную оболочку для кодирования любых типов, передаваемых по сети. Для этого необходимо было реализовать как можно более оптимизированный алгоритм кодирования, минимизирующий повторные дорогостоящие действия. Таким образом был разработан очень оптимизированный стерилизатор работающий по такому алгоритму:

Для сериализации:

1. получить тип объекта.
2. Попытаться найти оптимизированный заготовленный серализатор.
3. В случае если такой сериализатор найден, использовать его. Иначе попытаться записать тип, если он поддерживается CBOR.
4. Если необходимый тип является структурой, то проверить, сохранены ли поля этого типа. Если нет, то получить список полей и сохранить для дальнейшего использования.
5. Получить все значения свойств объекта используя PropertyInfo.
6. Записать их как массив значений. Для каждого свойства проделать все операции с пункта 1.

Для десериализации алгоритм практически такой же, кроме того, что при неизвестном типе создается промежуточный объект, который содержит данные, которые удалось прочесть силами типизации самого формата.

Данный алгоритм позволяет кэшировать дорогостоящие в получении данные и использовать максимально оптимизированные вручную сериализаторы для стандартных типов сообщений, таких как `CallRequest` и `FinalCommandExecution`. В достижении общей высокой производительности фреймворка именно сериализатор и его оптимизации сыграли большую роль.

3.6 Кодогенерация

Кодогенерация (далее — КГ) является ключевой частью многих служебных фреймворков, таких как `gRPC`, `WPF` или `AvaloniaUI` и служит для автоматизированного написания шаблонного кода с минимальным участием программиста в конечном результате. В `C#` КГ является частью пайплайна компиляции проекта и срабатывает после начального считывания всего кода проекта и проведения его семантического анализа.

Всего в получившемся фреймворке КГ выполняет следующие функции:

1. Генерация ПО и реализация внутренней логики удаленного вызова (обычные вызовы методов, а так же полей и событий).
2. Генерация биндеров событий объектов.
3. Генерация провайдеров индексов вызовов.
4. Генерация инвокеров методов.
5. Генерация дополнительных методов интерфейса для внешнего вызова событий.

Для генерации всего вышеперечисленного используются оптимизированные инкрементальные генераторы `IncrementalGenerator`, использующие `API Roslyn` через пакет `Microsoft.CodeAnalysis.Csharp` и `Microsoft.CodeAnalysis.Analyzers`.

Для более высокого уровня абстракции при генерации кода решено было отказаться от использования литералов в качестве носителя основной части генерируемого кода и был реализован свой инструмент для работы со вставкой текста — `mROA.CodegenTools`. `mROA.CodegenTools` позволяет определять все

текстовые составляющие итогового генерируемого кода в виде шаблона, хранящимся отдельно от кода и не засоряя его (Рисунок 3.3).

```
// <auto-generated/>
using mROA;
using System;
using mROA.Implementation;
using System.Collections.Generic;
using mROA.Abstract;

namespace <!L namespaceName>
{
    partial class <!L className>
        : RemoteObjectBase, <!L originalName>
    {
        public <!L className>(int id, IRepresentationModule representationModule, IEndPointContext context, int[] callIndices)
            : base(id, representationModule, context, callIndices)
        {
        }

        <!I methods r sep methodSep><!D methodSep>
        <!D>
    }
}
```

Рисунок 3.3 Пример разметки шаблона для прокси-объекта

Однако из-за ограничений для проектов, содержащих SourceGenerator и IncrementalGenerator, желательно включить исходный код mROA.CodegenTools непосредственно в проект с генераторами.

3.6.1 Возможности развития средств генерации текста

Хотя текущая реализация mROA.CodegenTools уже повышает удобство генерации исходного текста программного кода, но уже присутствуют видимые возможности для улучшения API для взаимодействия с генератором текстов.

На данный момент процесс генерации подразумевается больше в процедурном стиле. Вероятно следующим шагом станет внесение более объектно-ориентированного вида в API этой суб-библиотеки.

Так же для ускорения всей генерации кода с использованием mROA.CodegenTools можно сделать автоматический генератор тех самых классов, о которых говорилось выше для снижения времени необходимого на создание новых шаблонов и их клонирование.

4 ТЕСТИРОВАНИЕ ПРОГРАММНОГО ПРОДУКТА

4.1 Основные понятия и принципы тестирования программного продукта

В связи со сложностью именно межмодульного тестирования, обычные юнит-тесты хотя и могут тестировать конкретные модули в отдельности, проверяя их и так атомарные функции, но обычно требуется комплексное тестирование взаимодействия модулей вместе при решении конкретных задач. Так же стоит уточнить, что хотя сам фреймворк и является открытым для возможности написания своих модулей и широких возможностей по конфигурации, но практически никакие модули не даются пользователю в непосредственное использование, кроме модулей реализации соединения и репозитория объектов для получения ПО. Поэтому для проверки во время разработки работоспособности всей системы в целом после каких либо изменений было выбрано демо-тестирование.

Демо-тестирование — это способ проверки работоспособности программного обеспечения, при котором корректность функционирования системы подтверждается через запуск и демонстрацию реальных сценариев взаимодействия. В отличие от автоматизированных тестов, демо-тестирование не полагается на формальные ассерты или покрытие кода, а основано на визуальной и логической проверке поведения системы в составе работающего приложения. Оно особенно эффективно на этапах прототипирования, когда цель — показать, что ключевые функции (например, удалённый вызов метода, сериализация сообщений, обработка ошибок) работают в условиях, близких к реальным. Демо-тестирование позволяет оценить не только корректность, но и прозрачность, производительность и удобство использования фреймворка без необходимости создания сложной тестовой инфраструктуры.

Кроме тестов функциональных тестов, реализованных через демо-тестирование так же использовались нагрузочные тесты для проверки накладных расходов фреймворка на удаленный вызов метода. Оно также

позволяет оценить, как на производительности сказываются изменения внутренней реализации модулей и изменение архитектуры в целом.

4.2 Функциональное демо-тестирование

Для тестирования работоспособности всех необходимых функций в была создана предметная область и API, описывающее её. В качестве такой предметной области была выбрана печать страниц на принтере от выпуска принтера до самой напечатанной страницы. В этой предметной области 3 интерфейса и один класс: IPrinterFactory, IPrinter, IPage и MyData.

Интерфейс IPrinterFactory представляет собой абстракцию некоего завода по производству принтеров IPrinter и поставщика данных о них. IPrinter уже является абстракцией самого принтера, предоставляющей возможности по печати и предоставлению информации о себе. IPage является абстракцией напечатанной страницы, которая может только предоставить свои данные в бинарном виде.

IPrinterFactory содержит в себе 5 методов:

1. Create — создание нового принтера, принимая как параметр его название.
2. Register — регистрация принтера в базе всех принтеров.
3. GetPrinterByName — выдача принтера с необходимым названием среди зарегистрированных.
4. GetFirstPrinter — первый среди зарегистрированных принтеров.
5. CollectAllNames — список имен всех зарегистрированных принтеров.

IPrinter содержит 5 методов, событие и свойство:

1. Свойство Resource — текущий ресурс принтера (число с плавающей запятой).
2. GetName — метод, возвращающий название принтера.
3. Print — отправка на печать текста. Кроме самого текста для печати среди параметров так же есть тестовые параметры: someParameter, context и

cancellationToken. Последние два являются контекстом запроса и токеном отмены выполнения.

4. Событие OnPrint — оповещает о печати страницы и в качестве параметров несет саму новую страницу IPage и контекст вызова.

5. SomeoneIsApproaching — служит для оповещения принтера о приближении кого либо к нему. В качестве параметра требуется имя сотрудника. Этот метод является недоверяемым и отмечен соответствующим атрибутом.

6. SetFingerPrint — устанавливает что то наподобие идентификатора для данного принтера. В параметрах содержится сам отпечаток в качестве массива байтов.

7. IntTest — тестовый метод для проверки сериализации структур.

IPage предоставляет только один метод, возвращающий данные страницы в бинарном формате — GetData. Он не принимает параметров.

Кроме вышеперечисленных классов и интерфейсов в этой демонстрационной сборке так же есть и инструменты для нагрузочного тестирования ILoadTest. Он содержит 5 методов, однако используется только два — метод Next, принимающий целое число и возвращающее следующее и AsyncTest, используемый для проверки отмены выполнения . Благодаря этому интерфейсу даже в обычном демо-тестировании удастся в меньшей мере оценить производительность и влияние нововведений во время разработки на её изменение. Этот же метод используется и в нагрузочном тестировании.

4.3 Нагрузочное тестирование

Как говорилось выше, нагрузочное тестирование происходит с той же сборкой, что и функциональное демо-тестирование. Однако нагрузочное тестирование предоставляет возможность оценивать производительность и накладные расходы фреймворка в куда более сложных условиях.

Для нагрузочного тестирования сначала подключаются 100 клиентов, после чего они в течение 10 секунд непрерывно вызывают метод Next у

ILoadTest, считаю количество вызовов за 10 секунд. После чего все суммы от разных потоков складываются и делятся на количество секунд теста. Таким образом вычисляется общий RPS (Request per second) системы. Были проведены аналогичные тесты для gRPC, ASP.NET и Axum. Все они считаются высокопроизводительными платформами для работы с сетевыми запросами, в особенности Axum, который является одной из самых быстрых Web фреймворков из всех существующий по тестам TechEmpower.

Тестирование производилось на процессора AMD Ryzen 7 7700X, 64 гигабайта DDR5 RAM, Ubuntu 25.04.

После множественных запусков тестов были собраны данные о максимальном RPS для различных веб фреймворков. Все данные представлены в таблице 4.1.

Таблица 4.1-результаты нагрузочного тестирования

Веб-фреймворк	Пиковый RPS
gRPC	180000
ASP.NET	360000
Axum	490000
mROA	420000

Такая высокая производительность по сравнению с другими фреймворками, написанными на C# была достигнута благодаря оптимизациям в области кеширования при сериализации, предварительным выделением памяти в сериализаторе, оптимизацией асинхронных задач и других улучшений.

4.4 Дополнительные виды тестирования

Хотя основным методом тестирования было демо-тестирование, но на ранних стадиях разработки и ближе к последним нововведениям так же были написаны тесты для проверки преобразований ComplexObjectIdentifier из структуры в число, а так же для структуры RequestId. Эти тесты помогли добиться работоспособности этих двух уникальных структур-идентификаторов.

Таблица 4.1-Тест-кейс 1

Test Case №	1
Приоритет теста	Высокий
Название тестирования/Имя	Преобразование и обратное преобразование COI
Резюме испытания	Необходимо проверить правильность преобразования COI в 64-битное число и обратно в структуру
Ожидаемый результат	Совпадение исходного идентификатора с восстановленным
Фактический результат	Совпадение исходного идентификатора с восстановленным
Статус	Pass

Таблица 4.1-Тест-кейс 2

Test Case №	2
Приоритет теста	Высокий
Название тестирования/Имя	Равенство RequestId
Резюме испытания	Необходимо проверить правильность оператора сравнения для двух RequestId для правильной коммутации запросов
Ожидаемый результат	Совпадение сгенерированного идентификатора с созданным искусственно из его данных
Фактический результат	Совпадение сгенерированного идентификатора с созданным искусственно из его данных
Статус	Pass

Помимо использования тестов функциональности, так же были широко использованы тесты производительности с использованием BenchmarkDotNet. Использование именно BenchmarkDotNet позволило судить а какой либо значительной разнице производительности в различных вариантах процедуры, поскольку используемые им алгоритмы позволяют невелировать всевозможные [11] Таким образом были оптимизированы сериализация, генерация идентификаторов, репозиторий инвокеров и другие аспекты низкоуровневой

производительности, позволившие уменьшить накладные расходы на удаленный вызов до минимума.

Именно благодаря BenchmarkDotNet были исследованы возможности по использованию своего arena-аллокатора для сериализации для снижения нагрузки на сборщик мусора dotnet, однако на данный момент не получилось создать такой ручной аллокатор, который был значительно эффективнее, чем использование обычного выделения памяти в куче через `new byte[]`. Для малых объемах, как выяснилось в ходе тестов, стандартный аллокатор среды dotnet является невероятно быстрым, хотя такие тесты и не учитывают накладные расходы на последующие операции высвобождения выделенной памяти. Данное направление является перспективным, хотя вероятно, и не даст очень большого прироста в производительность напрямую, хотя может в целом хорошо сказать на конечном приложении, поскольку не будет создавать дополнительную нагрузку да сборщик мусора в сценариях с высокочастотными операциями активно работающие с памятью.

5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

5.1 Руководство по созданию сборки

Определение API для взаимодействия между узлами является самой простой частью внедрения mROA в проекты в качестве средства связи удаленных узлов. При разработке фреймворка был сделан упор на удобство использования в проектах без необходимости менять архитектуру и реализовывать какие либо дополнительные механизмы для коммуникации.

В качестве примера создания такой сборки API можно рассмотреть реализацию демонстрационной сборки, использовавшейся в функциональном тестировании. Рассмотрим интерфейс IPrinter. Для того, чтобы сделать его пригодным для передаче по сети, нужно сделать как минимум два шага: добавить этому интерфейсу атрибут SharedObjectInterface и унаследовать его от интерфейса IShared. Основным критерием тут является именно SharedObjectInterface, поскольку Ishared могут наследовать и другие интерфейсы, наследуемые конечным интерфейсом с возможностью передачи удаленному узлу. IShared лишь сигнализирует о том, что методы, определяемые этим интерфейсом необходимо подготовить для использования сетевых вызовов.

Методы, содержащиеся в интерфейсе могут возвращать все базовые типы, в том числе списки и массивы из них, другой интерфейс, отмеченный атрибутом SharedObjectInterface и подходящий по критериям выше, а так же тип асинхронного выполнения метода Task и Task<T>. На параметры распространяются те же ограничения. Параметры с типом RequestContext и CancellationToken не передаются по сети, а генерируются на конечном узле.

Для вызова метода по недоверенному каналу, необходимому методу нужно добавить атрибут Untrusted. Такой метод имеет такие же ограничения на параметры, как и обычные, однако возвращаемый тип должен быть void или Task.

В интерфейсе так же могут быть определены события. Их необходимо обозначить через `Action<...>`. Они имеют стандартные требования к параметрам. Для использования событий интерфейс должен быть разделяемым (partial). В дополнение необходимо будет реализовать дополнительные методы, необходимые для вызова соответствующих событий фреймворком.

На данный момент так же поддерживаются и реализация свойств в интерфейсах. На тип свойства действуют стандартные требования к возвращаемому типу метода.

Для создания такой сборки необходимо добавить в проект как минимум два пакета: `mROA` и `mROA.Codegen`. Для согласованности формата передачи данных и версии сериализатора в общую для узлов сборку так же можно добавить сам сериализатор, хотя он и не будет использоваться. На данный момент единственным реализованным сериализатором является `MROA.Cbor`. Все эти пакеты можно добавить через выполнение команды в терминале:

```
dotnet add mROA
dotnet add mROA.Codegen
dotnet add mROA.Cbor
```

5.2 Руководство по конфигурации узла

Конфигурация и соединение узлов является самой сложной частью использования `mROA`, однако этот процесс необходимо обозначить только при старте приложения. Конфигурация является самым сложным этапом, из того, в котором пользователь фреймворка сам принимает участие. Для обеспечения множества зависимостей в последней версии используется библиотека Microsoft — `Dependency Injection`. Поэтому для возможности конфигурирования в конечный проект необходимо кроме ссылки на сборку `API` так же добавить пакет `Microsoft.Extensions.DependencyInjection`.

Как у базового узла, так и у подключающегося есть как общие используемые модули, так и свои уникальные. К общим можно отнести `CborSerializationToolkit`, `BasicExecutionModule`, `RemoteInstanceRepository`,

CollectableMethodRepository, GeneratedCallIndexProvider, CancellationRepository. Всех их необходимо добавить через Services.AddSingleton.

При конфигурировании базового узла, которые может обрабатывать запросы нескольких удаленных узлов необходимо так же добавить BackendIdentityGenerator, NetworkGatewayModule, UdpGateway или UTGateway (опционально), ConnectionHub, HubRequestExtractor, CreativeRepresentationModuleProducer, MultiClientInstanceRepository. Так же есть возможность сконфигурировать DistributionModule. В зависимости от выбранного типа распространения или ExtractorFirstDistributionModule или ChannelDistributionModule, а так же соответствующий DistributionOptions. По умолчанию для каждой сессии будет создавать свой асинхронно слушающий экстрактор запросов.

При конфигурировании подключающегося узла необходимо кроме прочих добавить EndPointContext, InstanceRepository, UdpUntrustedInteraction (опционально), RepresentationModule, NetworkFrontendBridge, StaticRepresentationModuleProducer, RequestExtractor.

Для изменения формата передачи сообщений необходимо заменить CborSerializationToolkit на какой либо другой. При проектировании нового пайплайна для сообщений необходимо будет заменить RepresentationModule, ChaneledInteractionModule и другие.

Благодаря такой модульности заменять компоненты и разрабатывать новые удастся относительно просто, как это было с недавно разработанным DistributionModule. Так же частично это может быть настолько простым из-за использования DependencyInjection.

ЗАКЛЮЧЕНИЕ


В ходе работы были проведен поиск возможностей по использованию контекстных вызовов по сети, требований к протоколам сетевого и прикладного уровня, архитектуре реализации таких вызовов на С#. Так же были проанализированы конкуренты, позволяющие выполнять схожие операции, хотя и не так сильно развитые в тех аспектах, на которые был сделан упор в получившемся фреймворке.

Создание веб-фреймворка, предоставляющие такие возможности программистам может иметь влияние на разработку различных программного обеспечения для как связанных сетью удаленных компьютеров, так и межпроцессного взаимодействия в некотором роде, как это например реализовано в Language Server Protocol, использующегося для языкового анализа различных исходных текстов для помощи программистам. Применение созданной технологии имеет возможность изменить текущие процессы разработки некоторых видов ПО, хотя и не в равной степени для всех видов программ.

Основным выводом из проделанной работы является успешная реализация возможности использования всех поставленных перед этой библиотекой задач, необходимых для упрощения сетевого взаимодействия между множеством узлов и создание универсального неограниченно расширяемого стандартизированного подхода для удаленного вызова методов.

Хотя на данный момент существует реализация исключительно для платформы DOTNET, однако из-за своей дата-ориентированности и протокола, не зависящего от конкретного языка, вполне возможно реализовать этот же протокол и для других платформ, таких как JVM (Java или Kotlin) в полной мере, Golang или Rust с возможными ограничениями и многих других.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. protobuf-net / [Электронный ресурс] // protobuf-net : [сайт]. — URL: <https://protobuf-net.github.io/protobuf-net/> (дата обращения: 02.09.2025).
2. about_Objects / [Электронный ресурс] // Microsoft Learn : [сайт]. — URL: https://learn.microsoft.com/ru-ru/powershell/module/microsoft.powershell.core/about/about_objects?view=powershell-7.5 (дата обращения: 02.09.2025).
3. Part 1: Introduction to Classic RPyC  [Электронный ресурс] // RPyC : [сайт]. — URL: <https://rpyc.readthedocs.io/en/latest/tutorial/tut1.html> (дата обращения: 02.09.2025).
4. object.h / [Электронный ресурс] // Github : [сайт]. — URL: <https://github.com/dotnet/runtime/blob/main/src/coreclr/vm/object.h> (дата обращения: 02.09.2025).
5. ZeroMQ / [Электронный ресурс] // Wikipedia : [сайт]. — URL: <https://ru.wikipedia.org/wiki/ZeroMQ> (дата обращения: 03.09.2025).
6. Александр @deniaa Сказка про Guid.NewGuid() / Александр @deniaa [Электронный ресурс] // Habr : [сайт]. — URL: <https://habr.com/ru/companies/skbkontur/articles/661097/> (дата обращения: 03.09.2025).
7. P. Leach, M. Mealling, R. Salz A Universally Unique Identifier (UUID) URN Namespace / P. Leach, M. Mealling, R. Salz [Электронный ресурс] // IETF Datatracker : [сайт]. — URL: <https://datatracker.ietf.org/doc/html/rfc4122> (дата обращения: 03.09.2025).
8. Проектирование уровня сохраняемости инфраструктуры / [Электронный ресурс] // Microsoft Learn : [сайт]. — URL: <https://learn.microsoft.com/ru-ru/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design> (дата обращения: 03.09.2025).

9. Библиотека System.Threading.Channel / [Электронный ресурс] // Microsoft Learn : [сайт]. — URL: <https://learn.microsoft.com/ru-ru/dotnet/core/extensions/channels> (дата обращения: 03.09.2025).

10. Шаптала В.С., Солнцев Д.В. О современных форматах обмена данными между компонентами распределенной вычислительной системы [Текст] / Шаптала В.С, Солнцев Д.В // Техника средств связи. — 2019. — № 4. — С. 57-58.

11. Reliability / [Электронный ресурс] // BenchmarkDotNet : [сайт]. — URL: <https://benchmarkdotnet.org/#reliability> (дата обращения: 03.09.2025).

ПРИЛОЖЕНИЕ А

ПРИЛОЖЕНИЕ Б