Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

“САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ”

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

«СИСТЕМА ПЕРЕДАЧИ СООБЩЕНИЙ МЕЖДУ ПОЛЬЗОВАТЕЛЬСКИМИ ПРИЛОЖЕНИЯМИ ОС LINUX»

Автор Каширин Кирилл Сергеевич _____
(Фамилия, Имя, Отчество) (Подпись)

Направление подготовки (специальность)_____
(код, наименование)
09.04.04 - Программная инженерия

Квалификация Магистр
(бакалавр, магистр)*

Руководитель ВКР Ожиганов А.А., д.т.н., профессор _____
(Фамилия, И., О., ученое звание, степень) (Подпись)

К защите допустить

Руководитель ОП Бессмертный И.А., профессор, д.т.н. _____
(Фамилия, И.О., ученое звание, степень) (Подпись)

“_____” _____ 20 ____г.

Санкт-Петербург, 2019 г.

Оглавление

Оглавление 5

Список терминов 8

Введение 9

1 Связующее программное обеспечение 11

1.1 Классификация программного обеспечения 11

1.2 Связующее программное обеспечение, ориентированное на обработку сообщений (Message-oriented middleware) 14

2 Системы обмена сообщениями 16

2.1 Сильно связанные и слабо связанные интерфейсы 17

RPC 17

2.2 Виды архитектур систем передачи сообщений 19

Брокер 19

Без брокера 22

Брокер как служба обнаружения приложений 23

Распределённый брокер 25

Распределённая служба обнаружения приложений 26

2.3 Шаблоны передачи сообщений 28

2.4 Очереди сообщений 29

3 Сравнение систем обмена сообщениями 31

Общая классификация систем передачи сообщений 31

Стандарт AMQP 32

Сравнение ZeroMQ и YAMI4 33
Фиксированные шаблоны обмена против гибкости 34
Безопасность потоков 36
Пропускная способность и приоритетный трафик 37
Общее сравнение различных систем передачи сообщений 39
4 Разработка системы 42
4.1 Требования к системе 42
4.2 ZeroMQ 42
Преимущества 42
Недостатки 43
4.3 Сравнение ZeroMQ и её последователей 44
4.4 Соответствие требованиям, предъявляемым к системе 48
Среда передачи сообщений 48
Исполняемый формат системы, различные интерфейсы доступа 49
Затрачиваемые ресурсы 49
Надёжность передачи 50
4.4 Архитектура системы 52
Сервис имён 53
Архитектура взаимодействий между приложениями 56
4.5 Сериализация сообщений 57
5 Результаты разработки системы 62
Заключение 66
Список литературы 67

Список терминов

Системное программное обеспечение — программное обеспечение, взаимодействующее напрямую с аппаратными средствами и внешними устройствами вычислительных машин.

Пользовательское программное обеспечение — программное обеспечение, не взаимодействующее напрямую с аппаратурой, а использующее интерфейсы, предоставляемые операционной системой или другим системным программным обеспечением.

Связующее программное обеспечение — класс программного обеспечения, который может объединять различные системные и пользовательские приложения между собой, предоставляя им доступ к общему интерфейсу взаимодействия друг с другом или с другими частями программной инфраструктуры.

Сообщение — в контексте данной работы, какая-либо информация, представленная в виде набора байт, которые могут передаваться по различным каналам связи внутри или между вычислительными устройствами.

API — программный интерфейс приложения, позволяющий одному приложению получать доступ к другому приложению. Обычно представляет собой описание набора различных функций, параметров и форматов данных [1].

Введение

При проектировании программных систем большого размера и сложности неизменно возникает необходимость разбиения данных систем на различные части. Это позволяет увеличить эффективность дальнейшей разработки и поддержки программной системы, а также упростить добавление в неё новых подсистем и компонентов.

Однако с появлением так называемой модульной структуры системы сразу же появляется и проблема связи частей системы между собой. Для реализации этой связи в программном обеспечении могут использоваться различные решения, которые описываются термином **связующее программное обеспечение, или middleware**.

Связующее программное обеспечение во всём своем многообразии используется для самых различных целей, которые так или иначе объединяют различные программные системы в единое целое, взаимодействие между частями которого осуществляется средствами этого связующего программного обеспечения.

Одной из задач, которые могут выполнять различные средства связующего программного обеспечения, является передача сообщений. Связующее программное обеспечение, которое отвечает за передачу сообщений, выделяется в отдельный класс связующего программного обеспечения — **связующее программное обеспечение, ориентированное на обработку сообщений, или message-oriented middleware (MOM)**.

Данная работа представляет собой разработку программной системы передачи сообщений между пользовательскими приложениями ОС Linux. В ходе выполнения данной работы проводится исследование и сравнение различных решений в области связующего программного обеспечения, ориентированного на обработку сообщений с целью выбора наилучшего решения для разработки системы передачи сообщений, теоретическое

обоснование различных аспектов архитектуры системы, а также описана сама разработка системы передачи сообщений.

Для лучшего понимания материала и всестороннего рассмотрения вопроса данная работа разделяется на несколько разделов:

1. В первом разделе рассматривается место связующего программного обеспечения среди всех видов программного обеспечения, а также рассматриваются различные виды связующего программного обеспечения и задачи, которые оно может выполнять.
2. Во втором разделе рассматриваются различные параметры связующего программного обеспечения, ориентированного на обработку сообщений, проводится сравнение различных архитектур построения систем данного типа.
3. В третьем разделе проводится сравнение и классификация представителей связующего программного обеспечения, ориентированного на обработку сообщений, между собой, а также обосновывается выбор конкретной системы для использования в разработке нашей системы передачи сообщений.
4. В четвёртом разделе рассматриваются различные аспекты архитектуры разрабатываемой системы, учитывающие достоинства и недостатки выбранного связующего программного обеспечения.
5. В последнем разделе приводятся результаты разработки системы, исследуется эффективность её работы и её соответствие поставленным в начале работы требованиям к системе.

Все определения и специальные термины, использованные далее в данной работе, описаны в соответствующем разделе Термины. Для лучшего понимания текста работы рекомендуется сперва ознакомиться с ним.

1 Связующее программное обеспечение

1.1 Классификация программного обеспечения

Разговор про связующее программное обеспечение следует начать с того, какое место оно занимает в общей классификации программного обеспечения. Программное обеспечение можно разделить на три части:

- Системное.
- Прикладное, или пользовательское.
- Связующее, или промежуточное (в международной практике принят термин middleware).

Связующее программное обеспечение — это программное обеспечение, которое может связывать различные части как системного, так и пользовательского программного обеспечения между собой, предоставляя разным сервисам и приложениям общий интерфейс доступа к каким-либо функциям других приложений или самой операционной системы.

Связующее программное обеспечение служит для взаимодействия различных систем и компонентов как системного, так и прикладного программного обеспечения между собой.

Связующее программное обеспечение может предоставлять функционал и интерфейс для взаимодействия прикладных программ с базами данных, с сетевой инфраструктурой, а также обеспечивать унифицированный доступ и взаимодействие программ в условиях большого разнообразия вычислительных сетей и систем, как в плане программного, так и в плане аппаратного обеспечения [2].

В большинстве представленных на рынке решений связующее программное обеспечение представляет собой какой-либо программный комплекс или инфраструктуру, например: веб-серверы и серверы приложений, мониторы транзакций, сервисные шины и системы

управления содержимым, системы доступа к базам данных, а также *системы, ориентированные на обработку сообщений*.

Также связующее программное обеспечение является базой, на которой разработчики программного обеспечения могут создавать более масштабные программные комплексы, комбинируя приложения, написанные с использованием различных технологий, работающих на разных типах машин. Часто значительно дешевле разрабатывать независимые приложения и подключать их с помощью хорошего промежуточного программного обеспечения, чем разрабатывать единое всеобъемлющее приложение.

Все крупномасштабные программные проекты, как правило, используют связующее программное обеспечение или создают своё собственное [3]. Проблема, которую оно решает, часто кажется простой (надёжно и быстро получить или передать данные), но аспекты связующего программного обеспечения проектов часто становятся наиболее сложными и проблемными частями в системе. Люди продолжают писать свои собственные слои связующего программного обеспечения, потому что использование готовых альтернатив зачастую означает больший объём работы, а не меньший.

Промежуточное программное обеспечение — это широкий спектр сервисов, распределённых между приложениями и операционной системой, которые предоставляют специализированные функции и взаимодействие между распределёнными приложениями [4]. По ряду причин, включая тот факт, что промежуточное ПО является относительно новой категорией программного обеспечения, функциональность различных видов промежуточного ПО не стандартизирована. Например, основная функция промежуточного программного обеспечения для обработки транзакций связана с управлением, но многие продукты для обработки транзакций также включают в себя услуги связи. Поставщики

промежуточного программного обеспечения добавляют функции, обеспечивающие взаимодействие их продуктов с различными аппаратными и программными средствами, а также помогают дифференцировать их продукт от конкурентов.

Поскольку каждый продукт в категории промежуточного программного обеспечения включает в себя различные функциональные возможности и поскольку стандартные функции в каждой категории быстро меняются, трудно найти полезные различия между сервисами промежуточного программного обеспечения.

Успешные продукты и технологии промежуточного программного обеспечения, как правило, занимают определенную нишу: например, подключение компонентов на одной платформе или для одной языковой среды. Несколько универсальных промежуточных программных продуктов (например, IBM MQ Series, BEA Tuxedo), как правило, очень дорогие и очень сложные.

Наконец, существуют некоторые серверы промежуточного программного обеспечения с открытым исходным кодом (главным образом, Java), но они, как правило, являются функциональными программами, не ориентированными на стандарты, или, скорее, реализуют так много различных стандартов, что не гарантируется совместимость даже при использовании одного продукта.

Существуют некоторые стандарты промежуточного программного обеспечения (например, JMS, CORBA), но они ограничены по объему. Например, JMS предназначен исключительно для приложений Java, хотя некоторые JMS-провайдеры создают нестандартные программные интерфейсы для других языков, а CORBA использует сложную объектно-центрированную модель, которая непригодна для многих типов приложений.

1.2 Связующее программное обеспечение, ориентированное на обработку сообщений (Message-oriented middleware)

MOM — это определённый класс связующего программного обеспечения, который поддерживает обмен сообщениями в среде активного взаимодействия разнообразных приложений. Обмен данными осуществляется посредством передачи сообщений и создании очереди сообщений, поддерживающей как синхронные, так и асинхронные взаимодействия между распределёнными вычислительными процессами. Система MOM обеспечивает доставку сообщений, используя надёжные очереди и предоставляя службы каталогов, безопасности и администрирования, необходимые для поддержки обмена сообщениями.

MOM — это прежде всего промежуточное программное обеспечение, которое облегчает взаимодействие между распределёнными приложениями. Хотя MOM поддерживает как синхронный, так и асинхронный обмен сообщениями, он наиболее тесно связан с асинхронным обменом сообщениями с использованием очередей. MOM отправляет сообщения из одного приложения в другое, используя очередь в качестве промежуточного шага. Клиентские сообщения отправляются в очередь и остаются там до тех пор, пока не будут получены приложением сервера. Преимущество этой системы заключается в том, что серверное приложение не обязательно должно быть доступно при отправке сообщения, вместо этого сервер может получить сообщение в любое время. Кроме того, поскольку сообщения могут быть извлечены из очереди в любом порядке, MOM также может облегчить поиск сообщений с использованием схем приоритетов или распределения нагрузки. MOM также может обеспечить уровень отказоустойчивости, используя постоянные очереди, которые позволяют восстанавливать сообщения при сбое системы.

MOM поддерживает сообщения и, следовательно, в первую очередь предназначен для поддержки отложенной связи, в то время как одноранговые и удаленные вызовы процедур (RPC) предназначены для поддержки синхронной связи. Согласно RPC, принимающий сервер должен быть доступен для приёма отправленных сообщений. Если сервер не работает, сообщение не может быть доставлено в это время. MOM, с другой стороны, может отправлять сообщения на неработающие серверы без необходимости их повторной отправки. Сообщения в системе MOM помещаются в очередь и извлекаются всякий раз, когда сервер запрашивает их. Доступен ли сервер в момент отправки сообщения, не имеет значения.

2 Системы обмена сообщениями

Связующее программное обеспечение, ориентированное на обработку сообщений — это концепция, которая включает передачу данных между приложениями с использованием канала связи, который переносит автономные единицы информации (сообщения). В среде связи на основе МОМ сообщения обычно отправляются и принимаются асинхронно. При использовании МОМ, приложения не связаны друг с другом напрямую, отправители и получатели могут ничего не знать друг о друге. Вместо этого они отправляют и получают сообщения в системе обмена сообщениями. Система обмена сообщениями (МОМ) отвечает за доставку сообщений по назначению [5].

В системе обмена сообщениями приложение использует программный интерфейс, предоставляемый МОМ. Клиент обмена сообщениями отправляет и получает сообщения через систему обмена сообщениями, как показано на рисунке 1:

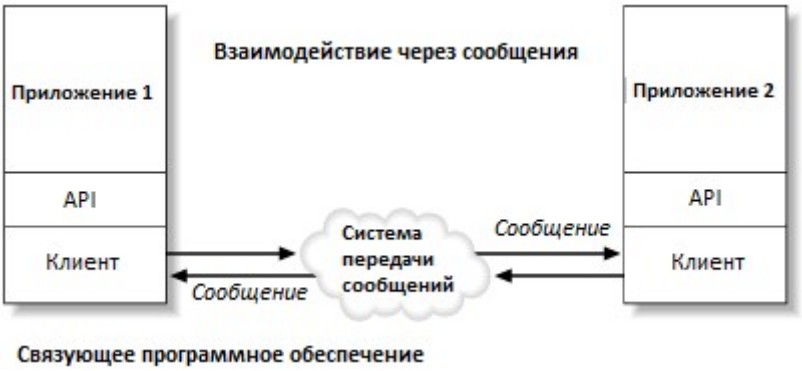

Рисунок 1 — Взаимодействие приложений через систему обмена сообщениями.

Система обмена сообщениями отвечает за управление точками соединения между несколькими клиентами, а также за управление несколькими каналами связи между точками соединения. Система обмена сообщениями обычно реализуется как программный процесс, который обычно называют

сервером сообщений или брокером сообщений. Серверы сообщений обычно могут быть сгруппированы вместе, чтобы сформировать кластеры, которые предоставляют расширенные возможности, такие как балансировка нагрузки, отказоустойчивость и сложная маршрутизация с использованием управляемых доменов безопасности.

2.1 Сильно связанные и слабо связанные интерфейсы

Обмен сообщениями обеспечивает слабосвязанную среду, в которой приложению не нужно знать подробные сведения о том, как связаться с другими приложениями и взаимодействовать с ними. При выборе типа инфраструктуры связи важно учитывать компромиссы между слабо- и сильно связанными интерфейсами, а также с асинхронным и синхронным режимами взаимодействия.

RPC

Программирование в стиле удаленного вызова процедур (RPC) в течение ряда лет находило разумное применение в отрасли. Технологии, в которых преимущественно используется RPC-стиль, включают архитектуру Common Object Request Broker (CORBA), Remote Method Invocation (RMI), DCOM, ActiveX, Sun-RPC, Java API для XML-RPC (JAX-RPC) и Simple Object Access Protocol (SOAP) v1.0 и v1.1.

В программировании в стиле RPC объект и его методы (или процедура и её параметры) «удалены» так, что вызов процедуры или метода может происходить через разделение сети. Приложение использует локальный прокси-сервер, обычно называемый «заглушкой», который имитирует интерфейс удаленного объекта и его методы. Приложение, выполняющее вызовы процедур, называется «клиентом», а удаленная реализация называется «сервером» или «службой». Клиент выполняет то, что выглядит как локальный вызов метода или вызов процедуры, который

фактически перенаправляет данные к удаленной реализации. Аналогично, возвращаемые значения и выходные параметры передаются обратно вызывающей стороне.

Интерфейсы в стиле RPC обладают преимуществами модели программирования, в которой удаленные процедуры представляются таким образом, что имитирует базовую объектную архитектуру соответствующих приложений, что позволяет разработчику делать вызовы «нормального» вида методов на родном языке.

Связь в стиле RPC, как правило, носит синхронный характер. По замыслу программирование в стиле RPC имитирует последовательный поток выполнения, который будет использовать «нормальное» нераспределенное приложение, где каждый оператор выполняется последовательно.

Распределённый синхронный вызов RPC предоставляет возможность немедленного ответа, указывающего, была ли операция успешной. Однако в распределённой системе существуют дополнительные сценарии сбоя, с которыми нераспределенная программа не должна иметь дело. При выполнении синхронной операции между несколькими процессами успех одного вызова RPC зависит от успеха всех последующих вызовов RPC, которые являются частью одного и того же синхронного цикла запроса / ответа. Это делает весь вызов предложением “всё или ничего”. Если по какой-либо причине одна операция не может быть завершена, все другие зависимые операции не будут выполнены [6]. Если один из процессов недоступен, приложение, инициирующее запрос, должно каким-то образом отметить сбой, повторить попытку позже или предпринять другие действия - например, сообщить человеку, что ему просто не повезло, и он должен попытаться вернуться позже. Чтобы компенсировать это, обработка ошибок и логика восстановления должны быть встроены в каждое приложение, использующее программирование в стиле RPC.

2.2 Виды архитектур систем передачи сообщений

На самом базовом уровне системы обмена сообщениями обычно строятся с помощью двух видов архитектур: с брокером или без. **Брокер** — это отдельное приложение, обеспечивающее обработку, принятие и передачу сообщений между клиентами. От вида архитектуры напрямую зависят возможности, которые может предоставлять система сообщений, и её недостатки. Например, архитектура с брокером позволяет обеспечить надёжность передачи сообщений за счёт хранения сообщений внутри брокера, при этом жертвуя пропускной способностью сети из-за необходимости передачи всех сообщений через брокера, который становится “узким местом”. А архитектура без брокера позволяет достичь максимальной скорости передачи сообщений, при этом жертвуя надёжностью доставки. В свою очередь, недостатки этих архитектур зачастую могут быть смягчены средствами, встроенными в саму систему сообщений. Например, системы без брокера могут позволять реализовать брокер самостоятельно, а системы с брокером могут дать возможность создавать распределённых брокеров, уменьшая проблему “узкого места”.
Ниже представлены базовые типы архитектур систем сообщений, рассмотрены их преимущества и недостатки.

Брокер

Архитектура большинства систем обмена сообщениями построена вокруг центрального сервера сообщений (брокера). Такой тип архитектуры можно представить как классическую топологию типа «звезда» [7]. Каждое приложение связано с центральным брокером, ни одно приложение не говорит напрямую с другим приложением, и все общение приложений проходит через брокера.

У этой модели есть несколько преимуществ:

- Во-первых, приложения не должны иметь никакого представления о местонахождении других приложений. Единственный адрес, который им нужен - это сетевой адрес брокера. Затем брокер направляет сообщения в нужные приложения на основе различных критериев (имя очереди, ключ маршрутизации, тема, свойства сообщения и т.д.), а не по информации о физической топологии (IP-адреса, имена хостов).
- Во-вторых, жизни процессов отправителя и получателя сообщений могут не пересекаться. Приложение отправителя может отправить сообщения брокеру и завершить работу. Сообщения будут доступны для приложения получателя в любое время.
- В-третьих, модель брокера в некоторой степени устойчива к сбою приложения. Таким образом, если приложение содержит ошибки и склонно к сбоям, сообщения, которые уже находятся в брокере, будут сохранены, даже если приложение завершится сбоем.

Недостатки брокерской модели также вытекают из её архитектуры:

- Во-первых, это требует излишнего количества передач сообщений.
- Во-вторых, тот факт, что все сообщения должны быть переданы через посредника, может привести к тому, что посредник окажется узким местом всей системы.

Во-первых, давайте посмотрим, как этот сценарий будет выглядеть при реализации с использованием шаблона обмена «запрос-ответ» с архитектурой, основанной на парадигме RPC (удаленный вызов процедуры), когда одно приложение «вызывает» функцию в удалённом приложении. Это делается путем упаковки аргументов функции и отправки их по сети в другое приложение, где аргументы распаковываются и функция обрабатывается. Затем результат снова

упаковывается и отправляется обратно вызывающему приложению, которое распаковывает его и продолжает обработку [8].

Как можно увидеть на рисунке 2, при использовании этой парадигмы нам нужно 12 сетевых передач для выполнения сценария:

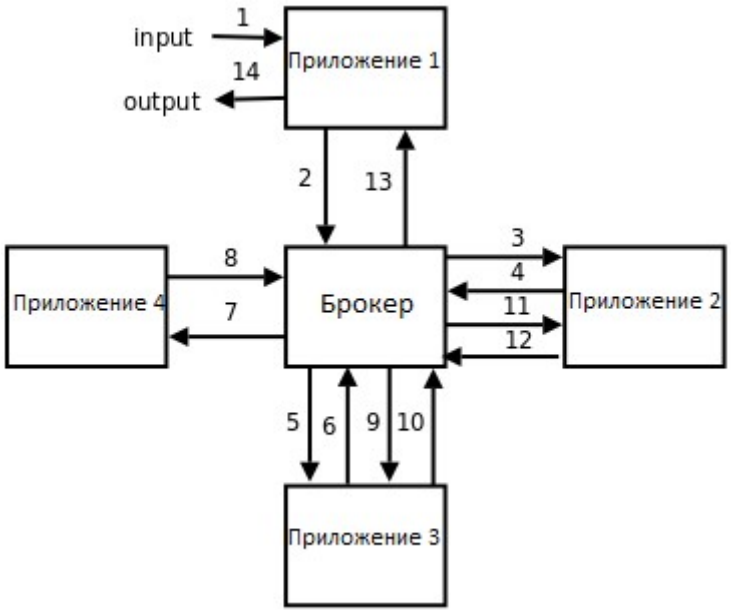

Рисунок 2 — Архитектура с брокером, обмен по протоколу RPC.

Чтобы снизить нагрузку на брокера и устранить задержки, мы можем отказаться от предыдущей модели и реализовать сценарий конвейерным способом. Таким образом, мы можем избежать половины сообщений (возвращающих данные из функций RPC). Такое решение изображено на рисунке 3:

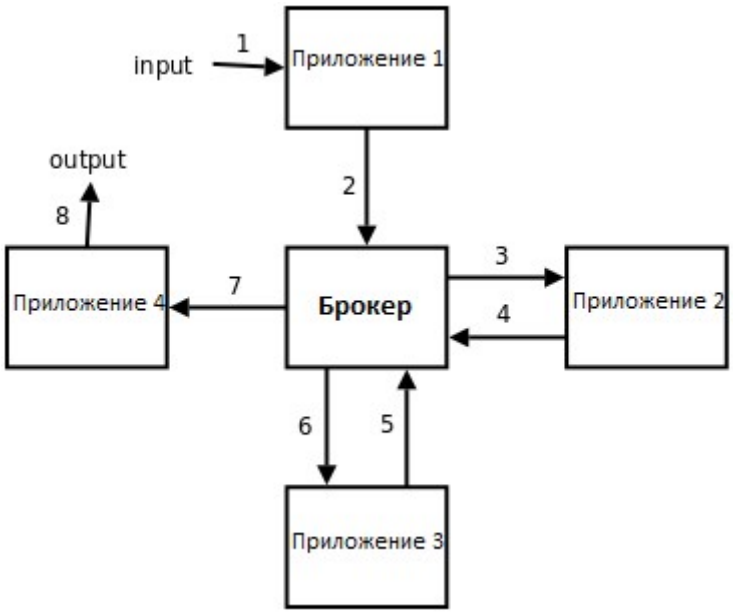

Рисунок 3 — Архитектура с брокером, обмен по протоколу конвейера.

С архитектурой центрального брокера нельзя достичь большей эффективности, чем эта. Если брокер все еще является узким местом или задержка все ещё слишком высока, единственный способ продвинуться вперёд - избавиться от самого брокера.

Без брокера

Рисунок 4 показывает сценарий с приложениями, отправляющими сообщения друг другу без посредника:

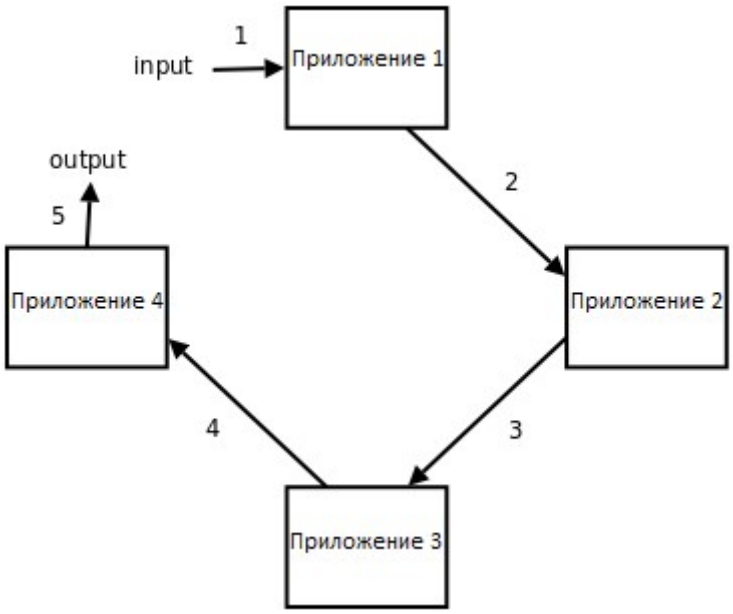

Рисунок 4 — Архитектура без брокера.

Как можно видеть, число передач сообщений уменьшилось до трёх, и в сети нет ни одного узкого места. Такая схема идеальна для приложений с низкой задержкой или высокой скоростью транзакций. Компромиссом является ухудшение управляемости системы. Каждое приложение должно подключаться к приложениям, с которыми оно связывается, и, таким образом, оно должно знать сетевой адрес каждого такого приложения. Хотя это и приемлемо в простом случае, как в нашем примере, в реальной корпоративной среде с сотнями взаимосвязанных приложений, это может стать крайне трудной задачей. В тоже время само число соединений увеличивается, так как каждое приложение должно соединяться со всеми другими приложениями, а не только с брокером.

Брокер как служба обнаружения приложений

Функциональность брокера можно разделить на две отдельные части. Во-первых, у брокера есть список приложений, подключённых к нему. Он знает, что приложение X работает на машине Y и что сообщения,

предназначенные для X, должны отправляться на Y. Он действует как служба каталогов. Во-вторых, брокер осуществляет передачу сообщений. Чтобы решить проблему управляемости, мы можем оставить часть прежней функциональности в брокере, но перенести передачу сообщений на сами приложения. Таким образом, приложение X зарегистрируется у брокера, сообщив ему, что оно работает по адресу Y. Приложение Z, желающее отправить сообщение в приложение X, запросит у брокера местоположение X. Как только брокер ответит, что X находится по адресу Y , Z может создать соединение непосредственно с Y и отправить само сообщение, не беспокоясь о посреднике.

На рисунке 5 изображён данный вид архитектуры:

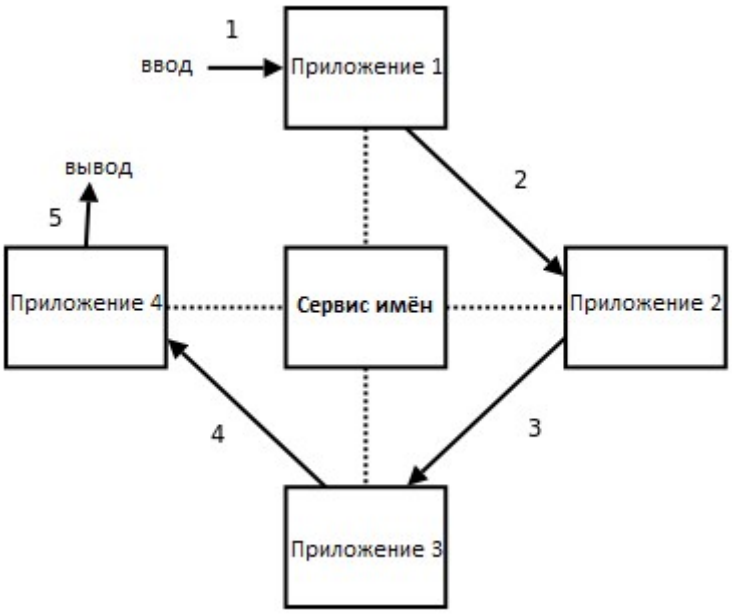

Рисунок 5 — Архитектура со службой обнаружения приложений.

Таким образом, мы можем получить высокую производительность и управляемость одновременно.

Распределённый брокер

Как уже было сказано, у модели брокера есть некоторые преимущества, которых нет у модели без брокера.

Приложение-отправитель и приложение-получатель не обязательно должны иметь пересекающееся время жизни. Сообщения хранятся в брокере, когда отправитель уже выключен, а получатель еще не запущен. Кроме того, в случае сбоя приложения сообщения, которые уже были переданы посреднику, не теряются.

Чтобы достичь такого поведения, вам просто нужно иметь какое-то приложение (брокер) между ними. Следовательно, нельзя избежать двух сетевых передач, чтобы получить сообщение от отправителя к получателю, но все же можно избежать проблемы брокера как узкого места.

Архитектура распределённого брокера, изображённого на рисунке 6, делает именно это:

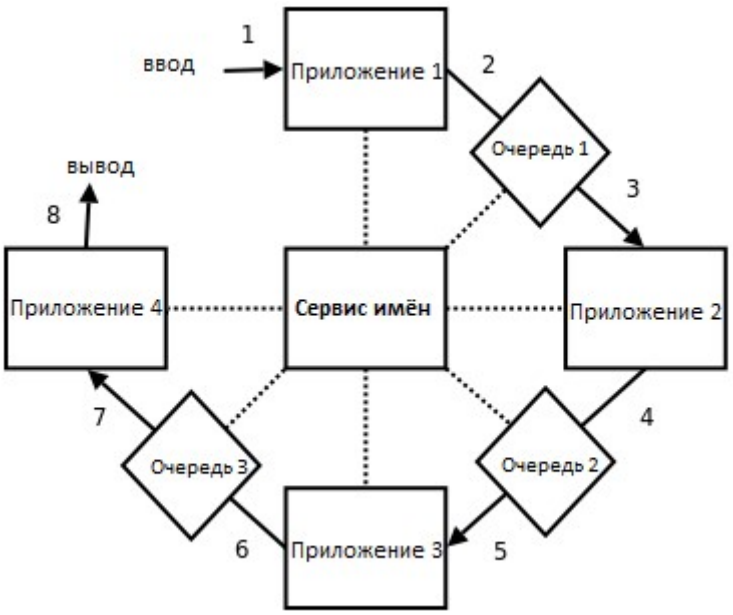

Рисунок 6 — Архитектура с распределённым брокером.

Как показано на диаграмме, каждая очередь сообщений реализована в виде отдельного приложения. Оно может работать на той же машине, что и одно из приложений, к которому он подключается, оно может быть расположено и на совершенно другой машине. Несколько очередей могут работать на одной машине или машина может быть выделена исключительно для размещения одной очереди. Очередь регистрируется брокером и, таким образом, она доступна для всех приложений в сети. Кроме того, очередь представляет собой очень простое программное обеспечение, которое получает сообщения от отправителей и рассылает их получателям. Таким образом, вероятность сбоя намного ниже, чем в реальных приложениях, полных сложной бизнес-логики.

Распределённая служба обнаружения приложений

В некоторых случаях необходимо избегать единой точки отказа. Другими словами, если одна подсистема выходит из строя, другие подсистемы должны продолжать работать. Хотя предыдущая модель полностью распределена по сообщениям, её конфигурация все ещё централизована в службе каталогов. Если происходит сбой службы каталогов или когда она недоступна, происходит сбой системы.

Для решения этой проблемы нам нужен распределённый сервис каталогов. Простейшим примером является случай со статической топологией сети. Идея состоит в том, что после развертывания топология сети производственной линии будет полностью стабильной, так как нет необходимости изменять конфигурацию на всех узлах.

На рисунке 7 показан этот вид архитектуры (маленькие пустые квадраты представляют копии конфигурации):

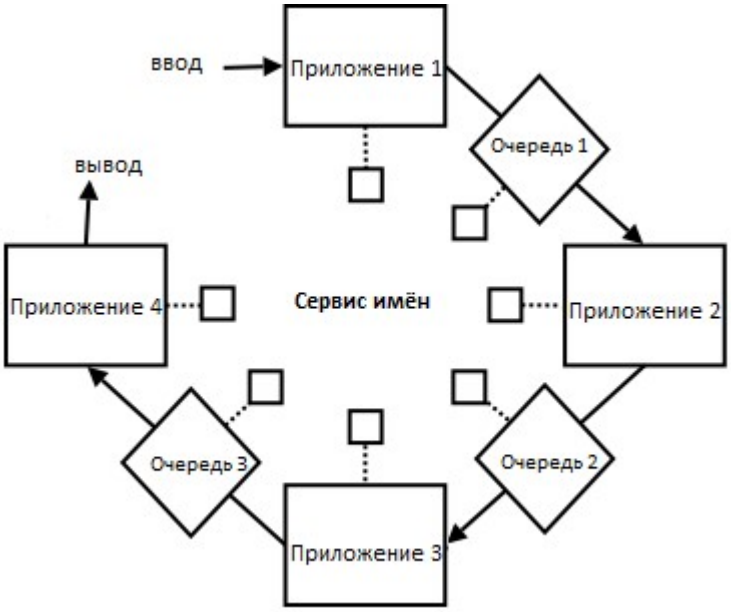

Рисунок 7 — Архитектура с распределённой службой обнаружения приложений.

Тем не менее, многие среды требуют как отсутствия единой точки отказа, так и динамически настраиваемой топологии сети.

В этом случае существует потребность в реальной службе распределённых каталогов. Рисунок 8 показывает описанную архитектуру:

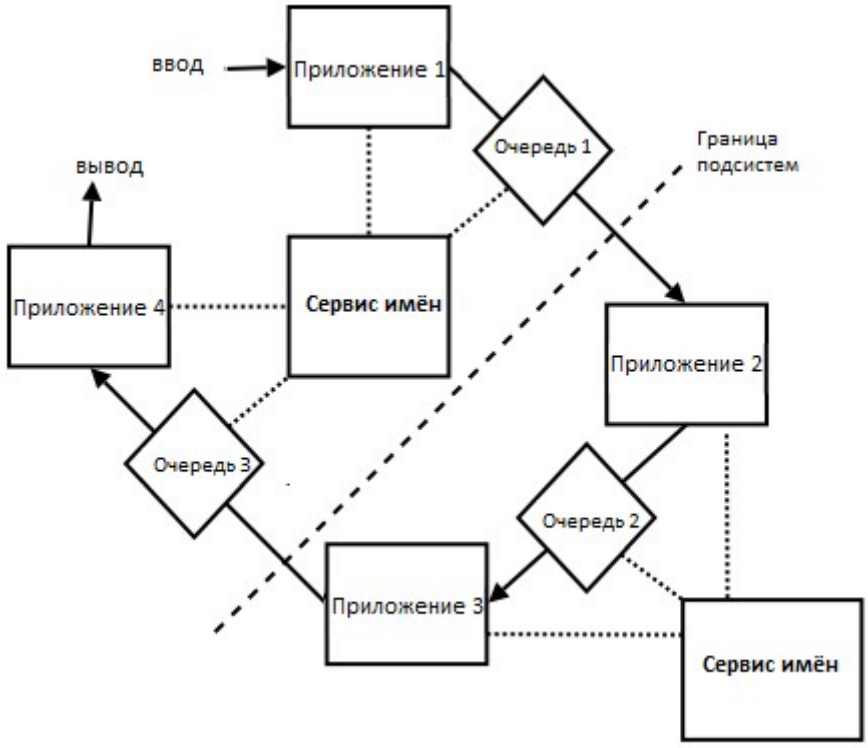

Рисунок 8 — Архитектура со смешанной распределённой архитектурой.

2.3 Шаблоны передачи сообщений

Под **шаблоном передачи сообщений** понимается набор параметров передачи сообщений, в который входят:

1. Протокол передачи сообщений, или последовательность их передачи
2. Число участников обмена сообщениями

Соответственно, шаблоны передачи сообщений можно классифицировать по следующим параметрам:

1. По числу получателей/отправителей:
    - Один отправитель, один получатель (point-to-point).
    - Один отправитель, много получателей (fan-out или pub-sub).
    - Много отправителей, один получатель.
    - Много отправителей, много получателей.
2. По распределению сообщений между получателями:

- Каждый получатель получает каждое отправленное сообщение.
- Каждый получатель по порядку принимает определённое число сообщений (fair-queued).
- Каждый получатель принимает только сообщения того типа, на который подписан (pub-sub).
- Получение сообщений по приоритету, все сообщения принимает наиболее приоритетный получатель.

2.4 Очереди сообщений

Общая модель серверов обмена сообщениями заключается в предоставлении хранилищ данных FIFO на основе дисков или памяти (по-разному называемых очередями, адресатами, темами и т.д.), с которыми одновременно могут работать многие приложения, принимая и отдавая сообщения [9].

Очереди в системах передачи сообщений обычно позволяют реализовывать более сложный функционал, связанный с обеспечением надёжности и качества обслуживания.

Очереди могут отвечать за такой параметр как надёжность доставки, так как хранят в себе все сообщения до тех пор, пока принимающая сторона не сможет их все обработать. Соответственно, системы с очередями по большей части используются в системах с архитектурой с брокером, который и отвечает за организацию очередей. В системах же без брокера очереди могут быть реализованы либо самим программистом, либо они присутствуют в системе неявным образом, и пользователь не может никаким образом получить к ним доступ, хотя и может устанавливать некоторые её параметры, например, размер очереди или время хранения сообщений в ней.

Очереди также могут отвечать за параметры обеспечения качества обслуживания, организуя для разных очередей различные уровни доступа, безопасности и шифрования, а также раздавая очередям различные приоритеты.

Также разные очереди могут поддерживать различные шаблоны обмена, организуя общение с большим количеством различных клиентов.

3 Сравнение систем обмена сообщениями

Общая классификация систем передачи сообщений

Данный тип программного обеспечения обычно предоставляет средства для создания системы сообщений с определёнными требованиями. Такая система может быть реализована в различных топологиях и с различными шаблонами обмена, с возможной поддержкой средств обеспечения надёжности передачи, организации очередей и отложенной доставки сообщений.

Представители данного типа программного обеспечения обычно предоставляют, кроме унифицированного и упрощённого интерфейса передачи сообщений, различные дополнительные возможности, такие как: синхронная или асинхронная доставка сообщений, поддержка различных шаблонов обмена, средства обеспечения надёжности доставки сообщений, поддержка очередей сообщений, маршрутизация сообщений, приоритизация сообщений, поддержка различных протоколов и средств передачи сообщений.

Для систем передачи сообщений существуют различные утверждённые стандарты, регулирующие все аспекты передачи сообщений и возможности, которые может предоставлять данная система. У каждого стандарта обычно существует несколько коммерческих реализаций, предоставляющих дополнительные возможности и особенности конкретной системы. К таким стандартам относятся AMQP, XMPP, STOMP, MQTT [10]. К системам обмена сообщениями можно также отнести системы, основанные на технологии RPC — удалённого вызова процедур. А некоторые системы сообщений не основаны на каком-либо стандарте и используют собственные решения.

Представители:

- RPC:
    - CORBA
    - ZeroC Ice
    - Apache Thrift
- YAMI4
- ZeroMQ
- Стандарт AMQP:
    - RabbitMQ
    - Apache ActiveMQ
    - Apache Qpid
    - OpenAMQ
    - SwiftMQ
- Стандарт XMPP (Jabber)
- Стандарт STOMP
- Стандарт MQTT

Стандарт AMQP

AMQP использует брокер, очереди, нет обратной поддержки разных версий, разные приложения используют разные версии, последняя версия выпущена в 2011-2012 годах, большой объём и сложность стандарта, небольшая активность сообщества разработчиков, неоптимальный бинарный формат кодирования сообщений, v0.9.1 - 69 страниц, v0.10 - 291 страница (на этой реализации основан Qpid), v1.0 - 113 страниц [11].

В решениях на основе стандарта AMQP существуют следующие базовые элементы:

- **Сообщение** является базовым элементом передаваемых данных, брокер никак не влияет на его содержимое и не знает ничего о его структуре. Однако к сообщению могут присоединяться различные

заголовки, которые используются брокером для распределения сообщений по очередям или приоритетам.

- **Точка обмена**, из которой клиенты получают и в которую отправляют сообщения. В этой точке сообщения распределяются в подключённые к ней очереди. При этом от типа точки обмена зависит алгоритм распределения сообщений по очередям. Существуют следующие типы точек обмена:
    - fanout — сообщение передаётся всем очередям, присоединённым к ней;
    - direct — сообщение передаётся в конкретную очередь с именем, совпадающим с ключом маршрутизации, который указывается отправителем при отправке сообщения;
    - topic — по сути, расширение предыдущего типа, в котором сообщение передаётся во все очереди, для которых совпадает маска ключа маршрутизации.
- **Очередь**, в которой сообщения хранятся до тех пор, пока другое приложение их не получит. Очереди могут использоваться совместно разными приложениями или нет, а также могут иметь различные параметры, такие как размер очереди, время хранения, назначение приоритетов и т.п.

Сравнение ZeroMQ и YAMI4

Проект ZeroMQ это развитие AMQP, решение части его проблем. Другим проектом, выступающим наравне с ZeroMQ, является YAMI4. Проекты YAMI4 и ZeroMQ имеют разную предысторию.

ZeroMQ появился в результате разочарования разработчиков в ранее использовавшемся протоколе AMQP. Компания iMatix решила прекратить участие в рабочей группе AMQP и продолжить разработку ZeroMQ с надеждой на то, что в этот раз всё получится лучше. То, что осталось в

ZeroMQ после AMQP — это существующие варианты использования, которые были в значительной степени мотивированы техническими проблемами, возникающими в финансовых системах. Таким образом, в центре внимания ZeroMQ находятся небольшие сообщения, отправляемые по относительно небольшому количеству стабильных каналов связи, например, цены, билеты, торговые запросы и другие финансовые материалы, которые необходимо быстро доставить с одного сервера на другой.

YAMI4, с другой стороны, был создан с учётом опыта, полученного от крупномасштабной распределенной системы управления, которая была разработана для комплекса управления в CERN. Проблемы в таких системах включают в себя большое количество устройств, осуществляющих одновременную связь, использование встроенных систем, наличие неисправных устройств и независимую обработку нескольких сообщений. Содержание данных в таких системах включает в себя широкий диапазон размеров сообщений, от показаний температуры до изображений с камеры и всего, что между ними.

Очевидно, что и YAMI4, и ZeroMQ являются решениями для обмена сообщениями, но, поскольку они были созданы для решения различных проблем, их набор функций также различен. В то время как ZeroMQ в основном занимается системами бизнес-типа, YAMI4 предлагает функции, которые более актуальны в системах управления и мониторинга в режиме реального времени [12].

Фиксированные шаблоны обмена против гибкости

ZeroMQ предлагает несколько шаблонов обмена (например, «запрос-ответ», «публикация-подписка» и т.д.), но ожидается, что пользователь заранее знает, какой шаблон будет использоваться с данным соединением. Например, если пользователь открывает соединение «запрос-ответ», его

нельзя использовать для связи «публикация-подписка». Другими словами, шаблон связи является свойством соединения и фиксируется при создании соединения.

Это кажется справедливым, если мы связываем физические соединения с логическими службами, но в целом вынуждает пользователя работать на неправильном уровне абстракции. Рассмотрим, например, устройство, подобное термометру, к которому можно получить доступ по сети. Очевидно, что мы можем подключиться к этому устройству, и физическое соединение — это то, что позволяет нам выполнять *различные* сценарии, например:

- запросить текущее значение температуры
- удаленно настроить устройство, установив его параметры
- установить автоматический поток обновлений

Каждый из этих сценариев является взаимодействием с одним и тем же логическим устройством, но они требуют разных шаблонов связи — первые два могут быть обменом «запрос-ответ», но последний выглядит как «публикация-подписка».

Ситуация становится еще более интересной, если учесть, что в типичной системе несколькими устройствами термометра можно управлять одним ведущим устройством, и именно это главное устройство подключено к сети. Это означает, что за одной и той же конечной точкой сети существует не только множество шаблонов связи, но и множество логических целей. Нет абсолютно никакой причины создавать несколько физических соединений для обработки всех этих различных взаимодействий с отдельными логическими термометрами, если они управляются одним главным контроллером.

YAMI4 предлагает гибкость связи, не принуждая пользователя выполнять какой-либо конкретный сценарий при создании физического соединения

— каждый шаблон обмена возможен в любое время после установления соединения. Чтобы быть точным, даже несколько одновременных шаблонов обмена возможны по одному соединению в любой момент. Это возможно, потому что YAMI4 тщательно различает физические связи и логические места назначения, поскольку они представляют собой понятия из разных уровней абстракции. Принуждение пользователей рассматривать эти различные концепции как эквивалентные может привести к неудачным проектам или пустой трате ресурсов.

Безопасность потоков

Подход к безопасности потоков является еще одной темой, в которой ZeroMQ и YAMI4 различаются.

В документации ZeroMQ четко указано, что отдельные сокеты не должны совместно использоваться потоками, если только приложение не обеспечивает надлежащую синхронизацию при передаче сокетов из одного потока в другой. В то же время программистам рекомендуется создавать столько сокетов, сколько им нужно — в частности, столько, сколько существует потоков приложений, которые должны создавать и отправлять сообщения. Этот подход раскрывает низкоуровневую природу ZeroMQ.

YAMI4 скрывает коммуникационные ресурсы от пользователя, так что нет прямого взаимодействия между кодом приложения и сокетами (или любыми другими объектами, подобными сокетам). Вместо этого код приложения взаимодействует с агентом, который инкапсулирует управление всеми системными ресурсами, включая сокеты. То есть низкоуровневые ресурсы не предоставляются на уровне кода приложения, и поскольку интерфейс агента полностью ориентирован на многопоточность, код приложения не должен иметь дело с какими-либо проблемами безопасности потока.

В частности, в отличие от ZeroMQ, пользователям YAMI4 не рекомендуется создавать несколько сокетов для одного и того же места назначения только для того, чтобы отразить многопоточность их приложений.

Пропускная способность и приоритетный трафик

ZeroMQ был разработан с единственной и очень четко определенной целью: получить максимально возможную пропускную способность. Действительно, результаты впечатляют, но вы можете проверить некоторые сравнения производительности, чтобы увидеть реальные результаты.

ZeroMQ достигает своей высокой пропускной способности с помощью пакетной обработки сообщений, которая позволяет отправлять несколько последовательных сообщений в виде одного блока, тем самым снижая издержки, связанные с заголовками пакетов и другими низкоуровневыми деталями.

Идея пакетных сообщений работает хорошо, но с очень важным допущением: что сообщения на самом деле образуют последовательность. Это полезно во многих системах бизнес-типа, где сообщения связаны по естественному порядку времени. Такое последовательное упорядочение, однако, означает, что нет места для понятия важности или срочности, то есть *приоритета* сообщений.

Концепция алгоритмов последовательности и пакетирования сообщений, основанная на этой идее, означает, что сообщения не могут опережать другие сообщения. Конечно, программист может использовать отдельные физические соединения для сообщений различной срочности, но, как уже было сказано выше, это работает на неправильном уровне абстракции — и что интересно, в стеке TCP/IP нет ничего, что фактически гарантировало бы, что чем больше важное сообщение получит какую-то особую

обработку, так как TCP-соединения не имеют такого параметра. Это означает, что, хотя ZeroMQ предлагает относительно хорошую пропускную способность, он может доставить ваше самое важное сообщение слишком поздно.

Быстро, но слишком поздно — неправильное решение в системах реального времени, и именно поэтому YAMI4 вводит концепцию приоритета сообщений непосредственно в своем программном интерфейсе.

Сообщения YAMI4 отправляются в исходящую очередь с некоторым приоритетом, и именно приоритет определяет, куда сообщение будет добавлено в очередь — не обязательно в конец.

Введение приоритетов сообщений было сознательным конструктивным решением, которое приносит свои собственные компромиссы с точки зрения дополнительных издержек и вмешательства в методы, такие как пакетирование сообщений. В результате YAMI4 может работать медленнее, чем ZeroMQ, если целью является необработанная пропускная способность, но с тем преимуществом, что он может преобразовать концепцию срочности на уровне приложения в практические условия, доставляя важные сообщения быстрее, чем те, которые могут безопасно ждать своей очереди. Это то, что делает YAMI4 более подходящим для систем реального времени.

Вы можете сравнить, как ZeroMQ и YAMI4 обрабатывают сообщения для достижения своих целей: ZeroMQ склеивает сообщения, образуя большие партии, чтобы быть быстрыми, в то время как YAMI4 преобразует сообщения в более мелкие куски, чтобы быть вовремя — эти две библиотеки, очевидно, имеют разное назначение.

Обратите внимание, что концепция приоритета не мешает YAMI4 сохранять порядок сообщений в пределах одного и того же уровня

приоритета — сообщения с одинаковым приоритетом гарантированно сохраняют порядок в одном и том же канале связи.

Общее сравнение различных систем передачи сообщений

В качестве подведения итогов данного раздела было проведено сравнение некоторых из вышеуказанных решений в области систем передачи сообщений.

Результаты проведённого обзора и сравнения систем передачи сообщений приведены в таблице 1. Как можно видеть из сравнения, каждая система сообщений обладает своими особенностями, выделяющими её на фоне конкурентов [13]. Различные решения отличаются базовой архитектурой (с брокером или без, RPC), поддерживаемыми шаблонами обмена, различными возможностями качества обслуживания (поддержкой безопасности передачи, надёжности доставки и т.п.), типом данных, которые они могут передавать (бинарные данные, объектно-ориентированные, текст, XML), а также транспортным протоколом, через который будут передаваться сообщения [14].

Таблица 1 — Сравнение систем передачи сообщений

Стандарт или реализация	**CORBA**	**ZeroC Ice**	**Apache Qpid**	**YAMI4**	**ZeroMQ**
Протокол	RPC	RPC, Ice Protocol	AMQP	Собственный	Собственный (ZMTP)
Брокер	Да	Да	Да	Опционально	Опционально
Поддержка различных языков программирования	C++, Java, Python	C++, Java, .NET, JavaScript, Python	Брокер: C++, Java Клиенты: C++, Java, Python, .NET, Ruby	C++, Java, .NET, Python	20+
Лицензия	GPL	GPL	Apache License 2.0	GPL, Boost	LGPL
Шаблоны обмена	req/rep, pub/sub	req/rep, pub/sub	point-to-point, fan-out, pub/sub, req/rep	pub/sub, req/rep	point-to-point, fan-out, pub/sub, req/rep

Качество обслуживания (QoS)	Надёжность, Синхронный и асинхронный обмен, Безопасность (SSL/TLS)	Надёжность, Синхронный и асинхронный обмен, Безопасность (SSL/TLS)	Очереди, Надёжность доставки, Безопасность (SASL, TLS)	Приоритеты сообщений, таймауты	Синхронный и асинхронный обмен, таймауты, HWM
Спецификация типа сообщений	Да, объектно-ориентированный	Да, объектно-ориентированный	Да	Да, Динамическая	Нет
Протоколы передачи сообщений	TCP, IPC	TCP, UDP, WebSockets, Bluetooth	TCP	TCP	TCP, IPC, inproc, PGM
Уровень поддержки сообщества	Небольшой	Большой	Средний	Небольшой	Большой

Далее представлены результаты тестирования некоторых систем доставки сообщений. Системы выбирались на основе их максимальных отличий друг от друга. Тестирование проводилось на примере базовых шаблонов обмена: шаблона запрос/ответ, при котором следующий запрос не отправляется до тех пор, пока не получен ответ на предыдущий, и шаблон издатель/подписчик, когда одно сообщение издателя доставляется всем подписчикам.

Шаблон обмена: запрос/ответ, размер сообщения: 4 байта

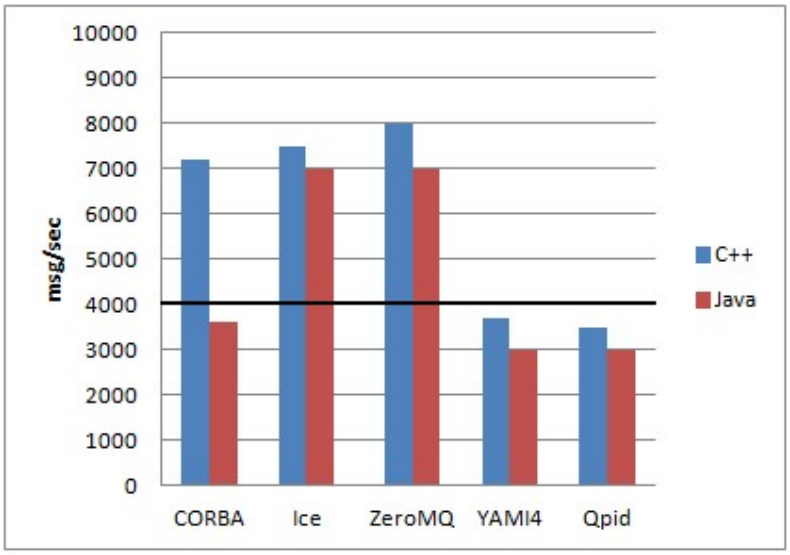

Рисунок 9 — Число сообщений в секунду по протоколу запрос-ответ

По результатам первого теста можно увидеть различия в скорости между системой сообщений на базе брокера (Apache Qpid) и остальными, а также уменьшение скорости передачи у YAMI4 из-за динамической системы типизации сообщений.

Шаблон обмена: публикация, размер сообщения: 8 байт

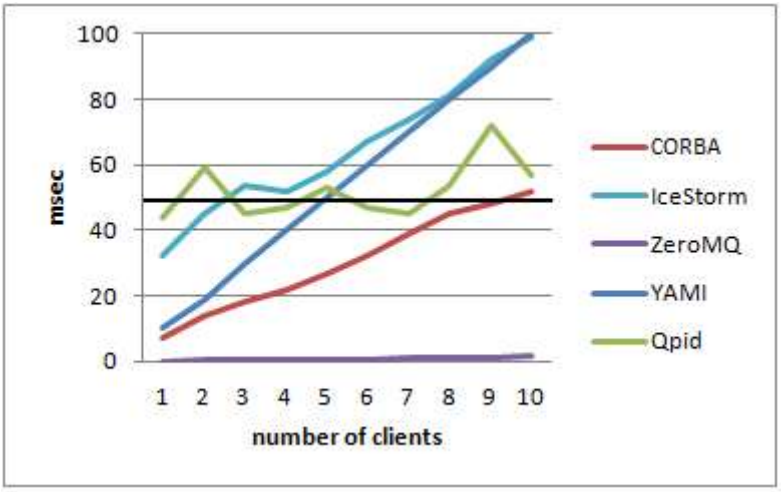

Рисунок 10 — Число сообщений в секунду по протоколу публикаций

4 Разработка системы

4.1 Требования к системе

- Работа внутри одной операционной системы Linux.
- Поддержка различных языков программирования (как минимум, C/C++/Python).
- Простота работы с библиотекой (подключение и API).
- Надёжность передачи сообщений.
- Минимизация затрачиваемых ресурсов.
- Передача не через сетевую подсистему, так как к ней могут применяться правила фильтрации.
- Поддержка шаблонов обмена «запрос-ответ» и «подписка-публикация».

Выполнение всех этих требований будет доказано далее либо в ходе разработки архитектуры в разделе 4.4, либо после проведения тестирования в разделе 5.

4.2 ZeroMQ

Преимущества

- Поддержка большого числа языков программирования. Это соответствует требованию о поддержке языков C/C++ и Python.
- Отсутствие зависимостей.
- Высокая скорость, небольшой размер, экономное использование памяти. Соответствует требованию о минимальном использовании ресурсов системы.

- Большое число уже реализованных шаблонов обмена и различных возможностей построения архитектуры. Соответствует требованию о поддержке шаблонов обмена «запрос-ответ» и «подписка-публикация».
- Выбор транспортного протокола (ipc, tcp, inproc). Соответствует требованию о невозможности использования сетевой подсистемы ОС. Для общения приложений будут использоваться протоколы ipc, которые в ZeroMQ реализуются через unix-сокеты, как известно, работающие через файловую подсистему, а не сетевую. Также этот пункт естественным образом ограничивает область действия системы сообщений до одной машины, как указано в другом пункте требований.
- Открытый исходный код, лицензия LGPLv3. Позволяет бесплатно использовать код ZeroMQ в наших решениях и даже при необходимости модифицировать исходные коды библиотеки.
- Отличная документация и активная поддержка сообщества.
- Не требует дополнительных приложений (например, брокеров).
- Простой, доступный для изучения API, схожий с Unix-сокетами. Данный пункт поможет при выполнении требования о простом встраивании системы передачи сообщений в различные приложения.

Недостатки

- Ничего не знает про данные, которые посылает, следовательно, требует использования отдельного протокола сериализации. Выбор протокола сериализации рассматривается в разделе 4.5.
- Сообщения пересылаются целиком и всё время хранятся в оперативной памяти, а значит, для пересылки сообщений больших размеров может потребоваться реализация отдельных протоколов [15].

- Алгоритм пересылки сообщений не позволяет реализовать приоритизацию сообщений.

4.3 Сравнение ZeroMQ и её последователей

Проект ZeroMQ обладает некоторыми недостатками, которые его авторы захотели исправить в следующем проекте — nanomsg.

Проект nanomsg обладает некоторыми преимуществами по сравнению с ZeroMQ.

Соответствие программного интерфейса стандарту POSIX

Программный интерфейс ZeroMQ создан на основе сокетов Беркли, реализующих стандарт POSIX, однако он не соответствует ему в точности. nanomsg, в свою очередь, стремится к полному соблюдению стандарта POSIX. Во-первых, сокеты представляются в виде чисел, а не указателей на void. Во-вторых, представленные в ZeroMQ «контексты» не используются в nanomsg, что позволяет создать простой программный интерфейс (так как возможно создавать сокеты сразу же без создания «контекста»), а также предоставить возможность использовать библиотеку для взаимодействия различных потоков процесса. В-третьих, функции передачи сообщений полностью соблюдают стандарты POSIX.

Язык реализации

Библиотека nanomsg создана с использованием языка C, а не C++ [16].

По этой причине у nanomsg нет зависимости от среды исполнения C++, что может быть критично, например, во встраиваемых системах с малым объёмом доступной памяти. Соответственно, объём выделяемой приложению памяти резко уменьшается, поскольку вместо контейнеров стандартной библиотеки C++ используется собственная реализация. Всё вышеперечисленное также означает уменьшение фрагментации памяти и так далее.

Встраиваемые протоколы и шаблоны обмена

В ZeroMQ нет программного интерфейса для подключения новых транспортных протоколов (например, WebSockets) и шаблонов обмена (таких как «запрос-ответ» и других). nanomsg представляет внутренний транспортный программный интерфейс, который является основой для создания новых транспортных протоколов и шаблонов обмена.

Также в nanomsg поддерживается новый протокол «опрос», в котором сообщение отправляется множеству получателей и ожидается получение ответа от каждого из них, а также протокол «шина», в котором сообщение каждого участника доставляется всем остальным.

Модель использования многопоточности

В nanomsg объекты не сильно связаны каждый с конкретным потокам, что позволяет решить некоторые проблемы:

- Протокол «запрос-ответ» в ZeroMQ проблематично использовать на практике, так как сокеты зависают, если сообщение теряется. По этой причине пользователю необходимо использовать сокеты типа XREQ и реализовывать повторные запросы самостоятельно. В nanomsg такой функционал уже поддерживается сокетом REQ.
- В nanomsg протокол «запрос-ответ» поддерживает асинхронный обмен, что означает возможность отправки нового запроса без ожидания ответа и получения новых запросов без необходимости ответа на предыдущие.
- В ZeroMQ из-за его реализации многопоточности определённый порядок вызовов функций присоединения и привязки сокетов не работает для транспортного протокола inproc. Также для этого транспортного протокола не работает автоматическое переподсоединение. Эти проблемы также исправлены в nanomsg.

Конечные автоматы

Библиотека nanomsg и её внутренние компоненты реализованы как множество конечных автоматов. Это архитектурное решение позволяет

убрать странный алгоритм отключения, как в ZeroMQ, и тем самым сделать проще разработку библиотеки.

Поддержка Windows IOCP

Ещё одной проблемой в ZeroMQ является использование программного интерфейса сокетов Беркли на системах ОС Windows, где их использование не является наилучшим решением. В случае ZeroMQ использование IOCP потребовало бы переписать большой объём кода и поэтому эта проблема так и не была решена. IOCP обладает большей эффективностью и позволяет использовать некоторые транспортные протоколы (например именованные пайпы), которые нельзя использовать через программный интерфейс сокетов Беркли. Поэтому nanomsg использует IOCP на системах Windows.

Запускаемый по уровню поллинг

Одна из особенностей ZeroMQ является поведение во время поллинга. Основным источником проблем является то, что поллер оповещает пользователя только когда приходит новое сообщение, при этом забывает про все сообщения до этого. nanomsg реализует поллер, который оповещает о поступлении сообщений вне зависимости от того, были ли они доступны в прошлом или нет.

Приоритизация маршрутов

В nanomsg реализуются приоритеты для отправляемых сообщений. Можно указать, чтобы сообщения перенаправлялись к другому получателю в случае, если изначальный получатель не отвечает или не доступен.

Асинхронный DNS

В ZeroMQ DNS-запросы выполняются синхронно, то есть в случае, когда невозможно получить доступ к DNS-серверу, вся библиотека (даже элементы, не использующие DNS) также становится недоступна. В nanomsg эта проблема решается тем, что DNS-запросы выполняются асинхронно.

Zero-Copy

В отличие от ZeroMQ, nanomsg пытается реализовывать настоящие механизмы «нулевого копирования», такие как удалённый прямой доступ в память и использование разделяемой памяти.

Эффективный поиск подписок

ZeroMQ в реализации шаблона «подписка-публикация» для поиска и сравнения тем подписок использует простые префиксные деревья. Однако если пользователь использует большое (несколько миллионов) число подписок, требуется более эффективная структура данных. В nanomsg вместо обычного используется более эффективная версия префиксного дерева.

Унифицированная модель буфера

В ZeroMQ реализована странная система с использованием двух очередей. Передаваемые сообщения хранятся как в самой очереди сообщений, так и в TCP-буферах. Это означает, что если необходимо ограничить объём данных в очереди, то требуется установить опции сокетов и для размера буфера, и для длины очереди. В nanomsg используется только одна очередь для хранения данных.

Несмотря на различные улучшения в указанных библиотеках, проведённое тестирование показывает, что производительность рассматриваемых решений находится примерно на одинаковом уровне. Результаты тестирования можно увидеть в таблице 2. В тестировании рассматривается передача сообщений между двумя приложениями по протоколу «запрос-ответ».

Таблица 2 — Сравнение ZeroMQ и nanomsg

Число	Время передачи, мс

сообщений	ZeroMQ	nanomsg
1	1.1	0.8
10	2.6	2
100	6	5.5
1000	35	33
10000	332	322
100000	3470	3350
1000000	33800	33100

При этом ZeroMQ является гораздо более состоявшимся решением, признанным и поддерживаемым большим сообществом разработчиков, а также по большей части обладает гораздо большим функционалом по настройке параметров передачи сообщений. Тогда как библиотеки nanomsg и nng создают впечатление куда менее поддерживаемых и перспективы их развития куда более туманны.

Поэтому, несмотря на упомянутые недостатки в проекте ZeroMQ, мы считаем его лучшей альтернативой для разработки нашей системы сообщений, чем проекты nanomsg и nng.

4.4 Соответствие требованиям, предъявляемым к системе

Различные аспекты архитектуры создаваемой системы сообщений будут зависеть от требований, которые к ней предъявляются.

Среда передачи сообщений

Требование передачи сообщений не через сетевую подсистему — означает, что для связи будет использоваться протокол IPC в ZeroMQ. ZeroMQ при создании соединений по протоколу IPC создаёт в файловой системе файлы unix-сокетов. Каждое имя такого файла должно быть уникальным, что означает либо необходимость согласования имён таких файлов у разных приложений между собой, либо необходимость поиска алгоритма создания уникальных имён. Первый вариант потребует

лишнего усложнения логики работы библиотеки, поэтому второй вариант выглядит более предпочтительным. Для создания уникальных идентификаторов приложений можно использовать библиотеку libuuid, которая позволяет создавать гарантированно уникальные идентификаторы в рамках одной операционной системы, что как раз соответствует требованиям к системе. Также возникает проблема размещения файлов сокетов в системе. Эту проблему можно решить с помощью уникальной возможности Linux создания так называемых абстрактных сокетов, которые создаются не в файловой системе, а в отдельном пространстве имён. При этом нам не нужно будет следить за неиспользуемыми файлами сокетов, так как абстрактные сокеты удаляются автоматически операционной системой, когда число ссылок на них достигает нуля.

Исполняемый формат системы, различные интерфейсы доступа

Поддержка различных языков программирования и простота работы с библиотекой означает, что библиотека должна быть динамической (DLL), что позволит с лёгкостью встраивать её в программы на любом требуемом языке. Также для простоты работы с библиотекой может потребоваться создать отдельный интерфейс доступа к функциям библиотеки для языка Python.

Затрачиваемые ресурсы

Требование минимизации затрачиваемых ресурсов выполняется благодаря использованию библиотеки ZeroMQ, которая ранее показала отличные результаты по скорости работы на всех требуемых шаблонах обмена в сравнении с другими библиотеками. Поэтому при правильной разработке архитектуры системы скорость работы и малый объём используемых ресурсов системы не должны пострадать.

Надёжность передачи

Требование по надёжности передачи сообщений требует более подробного рассмотрения.

Надёжность, которую мы хотим получить в конкретном случае, тесно связана с видом выполняемой работы. Любое обсуждение надёжности передачи сообщений должны начинаться с ясного заявления о том, какие сценарии обмена сообщениями мы рассматриваем. Обмен сообщениями - это не одна модель, а несколько, и в каждой существует компромисс между затратами и выгодой, и в каждой есть конкретное представление о “надёжности” передачи. Вот некоторые примеры:

- Модель «запрос-ответ», используемая для построения сервис-ориентированных архитектур. Вызывающий абонент отправляет запрос, который направляется в службу, которая выполняет некоторую работу и возвращает ответ. Простейшей проверенной моделью надежности является механизм повторных попыток в сочетании со способностью на стороне службы обнаруживать и правильно обрабатывать дублированные запросы.
- Переходная модель «публикации-подписки», используемая для распространения данных среди нескольких клиентов. В этой модели, если данные потеряны, клиенты просто ждут поступления свежих данных [17]. Хорошим примером будут являться протоколы передачи потокового видео или звука.
- Надёжная модель «публикации-подписки», используемая, когда стоимость потерянных данных слишком высока. В этой модели клиенты подтверждают данные, отправляя ответ отправителю. Если отправителю необходимо, он отправляет данные повторно. Это похоже на протокол TCP.

В ZeroMQ единственным действительно надёжным протоколом является REQ-REP, соответствующий протоколу «запрос-ответ». Данный протокол требует обязательного ответа на каждый запрос и не даёт возможности посылать следующий запрос до получения ответа.

Другой протокол — PUB-SUB, соответствующий протоколу «подписка-публикация», по своей природе не является надёжным и не требует подтверждения принятия сообщения. Обеспечение полной надёжности данного протокола будет означать то, что отправитель будет ожидать ответа от каждого получателя, что может серьёзно замедлить работу всей системы при наличии хотя бы одного медленного получателя, который не успевает принимать сообщения так быстро, как все остальные получатели. По этой причине для данного протокола обмена ZeroMQ по умолчанию отбрасывает сообщения для тех получателей, которые не могут его принять. Существуют несколько решений этой проблемы.

Во-первых, можно сгладить разницу в скоростях отправителя и получателя путём увеличения размера очереди неотправленных сообщений, что позволяет сделать ZeroMQ. По сути, это позволяет накапливаться неотправленным сообщениям до тех пор, пока получатель не сможет их все разобрать или пока не переполнится оперативная память (в которой и хранятся сообщения). В большинстве случаев этого должно быть достаточно, но в крайнем случае, если получатель не отвечает слишком долго, это приведёт к переполнению оперативной памяти, и поэтому этот способ не является полным решением проблемы.

Вторым возможным решением проблемы является использование протокола «запрос-ответ» в шаблоне «подписка-публикация», хотя данный способ и будет куда более затратным по сравнению с обычным протоколом PUB-SUB в ZeroMQ. Как уже было сказано выше, это может привести к тому, что один медленный получатель будет тормозить всю систему. Решением этой проблемы может являться сохранение

неотправленных сообщений отправителем и его попытка переслать их позже.

Естественно, ни один протокол не может обеспечить надёжную доставку сообщений в условиях нестабильной работы сети передачи или самих участников обмена. В случае системы сообщений, работающей только на одной машине, нестабильность работы сети передачи исключается. Также исключается случай, при котором машина аварийно завершает работу, так как в таком случае завершаются и все участники обмена сообщениями, которые после возобновления работы машины смогут снова послать требуемые сообщения.

Единственным вариантом нарушения надёжности передачи сообщения остаётся прекращение работы приложения-получателя. Однако в таком случае приложение-отправитель не получит ответа на свой запрос и сможет либо дождаться возобновления работы получателя, либо продолжить работу, сохранив отправленное сообщение до тех пор, пока получатель не возобновит работу.

4.4 Архитектура системы

В разделе 2.2 были рассмотрены различные подходы к организации архитектуры сообщений. Большинство из них основываются на работе брокера — отдельного приложения, отвечающего за организацию обмена сообщениями. На такой архитектуре строится, возможно, большая часть систем передачи сообщений.

ZeroMQ, в свою очередь, является решением, основанным на архитектуре без брокера, что даёт возможность приложениям общаться напрямую друг с другом.

Однако, при наличии большого числа приложений такой подход крайне затрудняет добавление новых приложений в систему. Причина этого в том, что каждое приложение должно знать адрес (в нашем случае, путь к

сокету) приложения, с которым оно захочет осуществлять обмен сообщениями. Несмотря на то, что наша система будет работать на одной машине, и мы можем и должны будем модифицировать все приложения, которые будут участвовать в обмене сообщениями для того, чтобы они могли использовать нашу систему, при произвольном указании адреса приложения могут возникать конфликты адресов между различными приложениями. Это приводит нас к выводу о том, что нам нужно какое-либо средство согласования имён.

Сервис имён

Во-первых, необходим определённый формат адресов каждого приложения, чтобы каждое приложение имело уникальный адрес в системе. В этом случае каждому приложению необходимо будет знать только уникальное имя приложения, с которым он хочет совершать обмен, для того, чтобы посылать ему сообщения.

Во-вторых, можно использовать отдельный сервис распределения имён, который будет хранить информацию обо всех подключённых к нему приложениях, и выдавать всем приложениям адреса всех остальных приложений. При этом он будет сообщать приложению в том случае, если имя уже занято. Сами приложения при этом будут передавать сообщения напрямую друг другу, что исключает проблему сервиса имён как узкого места.

Такой вид архитектуры изображён на рисунке 9.

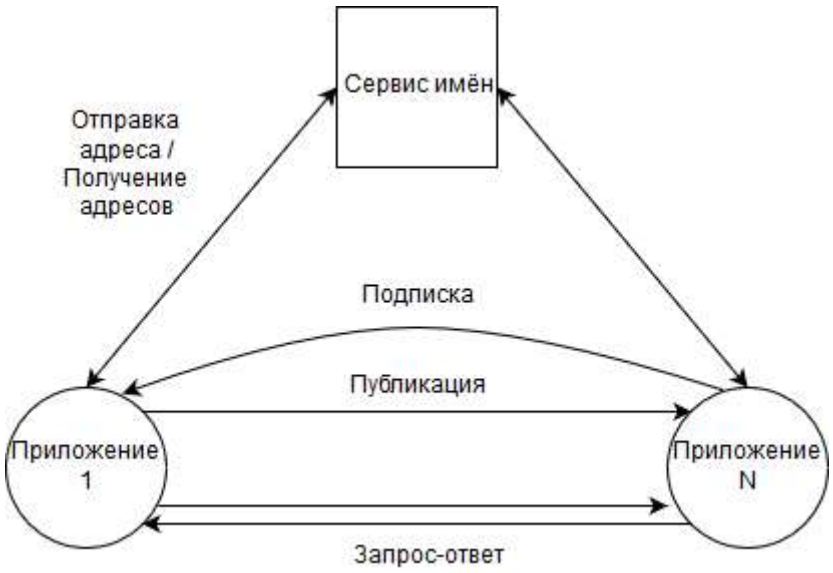

Рисунок 11 — Архитектура разрабатываемой системы передачи сообщений.

Дополнительно сервис имён может следить за состоянием всех подключённых к нему приложений и сообщать другим приложениям, если интересующее их приложение аварийно прекратило работу, не оповестив об этом сервис имён.

Недостатком данной архитектуры можно считать только то, что сервис имён становится дополнительной точкой отказа в системе передачи сообщений. Хотя при его отказе приложения сохраняют возможность передавать сообщения друг другу, они теряют возможность узнавать информацию о появлении новых участников обмена.

Решением данной проблемы может являться то, что один из участников обмена, обнаружив, что сервис имён прекратил работу, может создать процесс сервиса имён заново. При этом все приложения, которые используют систему обмена сообщениями, заново перешлют ему свои сохранённые списки имён, из которых он выберет наиболее актуальный список.

В целом мы приходим к системе, в которой все участники практически равноправны. В такой ситуации можно также рассмотреть различные протоколы автосогласования информации, которые существуют в библиотеке czmq — расширении стандартной библиотеки libzmq.

В czmq существуют следующие средства автосогласования: zbeacon и zgossip. Оба этих протокола отвечают за согласование информации между различными участниками обмена. При этом zbeacon основан на протоколе UDP, а значит его мы рассматривать для наших целей не будем.

Главная проблема zgossip в том, что он использует стандартные средства ZeroMQ для своей работы, а это значит, что какой-то из участников общения должен “привязаться” к определённому адресу (то есть вызвать функцию bind()), а все остальные участники должны к этому адресу подключаться (connect()). В полностью равноправной системе приложениям нельзя будет узнать, кому из них нужно сделать bind(). Поэтому необходимо, опять же, либо отдельное приложение, то есть сервис имён, которое будет “привязываться” к адресу, либо необходим другой способ согласования этого вопроса (например, через средства IPC Linux). Также в протоколе zgossip по умолчанию отсутствует возможность удаления опубликованных записей, а также возможность оповещения участников сети об удалении определённых записей.

Таким образом, мы приходим к необходимости разработки собственного сервиса имён. Данный сервис имён будет выполнять следующие функции:

- хранение имён
- добавление имён
- удаление имён
- получение списка имён
- оповещение подключённых клиентов об удалении имени

Данный сервис имён может быть запущен не только как отдельное приложение-демон, выполняющееся независимо от других, но и может

быть запущен в отдельном потоке любого из участников системы передачи сообщений.

Использование сервиса имён будет опциональным. При отказе от его использования будет подразумеваться, что все приложения, участвующие в процессе обмена, имеют своё собственное уникальное имя, и знают имена всех приложений, с которыми они хотят взаимодействовать. Это может быть актуально для систем небольшого размера, где накладные расходы на поддержку работы сервиса имён могут стать большей проблемой, чем ручное назначение имён, или в системах, где уникальность каждого приложения не имеет большого значения.

Архитектура взаимодействий между приложениями

Каждое приложение, подключающееся к системе передачи сообщений, будет иметь следующую архитектуру:

- 1 сокет типа SUB, который будет подсоединяться к сокетам PUB у всех необходимых приложений. Таким образом будет осуществляться шаблон обмена «публикация-подписка». Из сокета SUB приложение будет получать все сообщения от приложений, на которые оно подписалось.
- 1 сокет типа PUB, к которому будут подключаться все желающие приложения. Этот сокет является одной из внешних точек доступа к приложению, адреса которых известны других приложениям. Приложение будет передавать в этот сокет все свои публикации.
- 1 сокет типа ROUTER, который будет принимать все входящие запросы от других приложений и отправлять им ответы. Этот сокет является второй точкой доступа к приложению, реализующей протокол «запрос-ответ».
- N сокетов типа REQ: для каждого из N приложений, которому требуется послать запрос, создаётся отдельный сокет.

- (опционально) 1 сокет типа REQ для связи с сервисом имён при его использовании.

Схема такого приложения показана на рисунке 10.

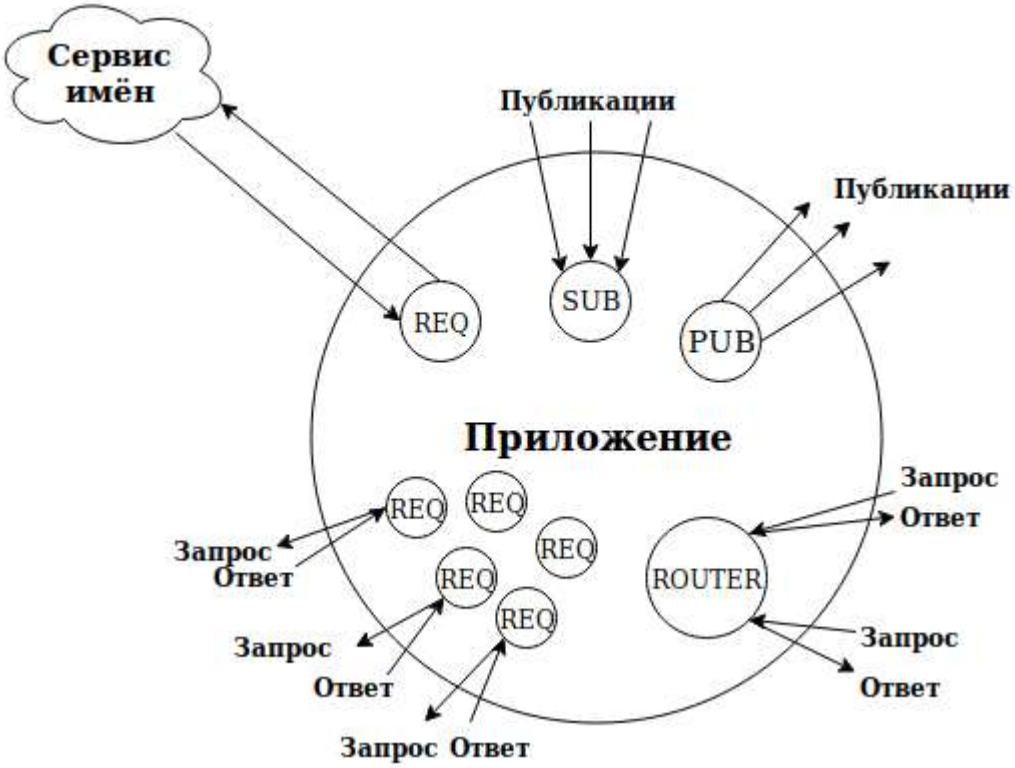

Рисунок 12 — Архитектура приложения, участвующего в обмене сообщениями.

На случай большой разницы в скоростях получателя и отправителя все входящие в приложение сообщения буферизуются и хранятся в отдельных очередях, что позволяет хранить сообщения до тех пор, пока получатель не сможет их обработать.

4.5 Сериализация сообщений

Так как библиотека ZeroMQ не имеет встроенных средств для формализации передаваемых данных, а воспринимает их просто как набор байт, для реализации системы передачи сообщений на ZeroMQ необходимо использовать алгоритм сериализации.

Сами разработчики ZeroMQ предлагают [18] следующие решения:

- zeromq/zproto
- Google Protocol Buffers
- MessagePack
- JSON-GLib
- C++ BSON Library

Из всех вышеперечисленных решений наибольший интерес представляют MessagePack и Protocol Buffers, как наиболее состоявшиеся и распространённые среди разработчиков.

Далее описаны различные особенности указанных протоколов и то, как эти особенности могут повлиять на возможность реализации требований при разработке нашей системы.

Особенности Protocol Buffers:

1. Каждый тип сообщений требует создания отдельного файла и компиляции, что приводит к дополнительным затратам разработчиков на сборку приложения из-за большого числа зависимостей.
2. Каждый тип сообщений создаёт отдельную библиотеку и отдельный набор функций, что может привести к увеличению объёма кода.
3. Код функций одинаковый во всех приложениях для данного типа сообщений, что исключает возможность ошибок, неправильный код не скомпилируется.
4. Легко подсоединить новое приложение, если оно будет использовать уже готовые типы сообщений.
5. Как говорят сами разработчики Protocol Buffers [19], они не предназначены для обработки больших сообщений. Как правило,

при необходимости работы с сообщениями размером более одного мегабайта необходимо рассмотреть альтернативные решения.

Особенности MessagePack:

1. Не требует предварительной спецификации и компиляции типов сообщений.
2. Тип сообщений напоминает json, что означает лучшую читабельность, простой парсинг в языках высокого уровня,
3. Существует возможность, в отличие от Protocol Buffers, создания словарей, которые являются стандартным типом в Python.
4. Одна подключаемая библиотека и, соответственно, один набор функций для всех приложений.
5. Динамическое создание и разбор структуры сообщения при каждой отправке и получении, в целом, также может привести к увеличению объёма кода. Особенно при использовании низкоуровневых языков типа C.

В таблице 3 и на рисунке 11 приведено сравнение эффективности работы указанных протоколов на различных типах сообщений.

Стоит заметить, что Protocol Buffers, в отличие от MessagePack, гораздо более строго типизирован и, например, не даёт возможности создания списка или словаря с элементами различных типов. При сравнении такие типы данных в Protocol Buffers заменялись на аналогичные, например, на вложенные сообщения. Соответственно, уже это может показывать, что при использовании Protocol Buffers может возникнуть больше трудностей при составлении сложных типов данных, чем у MessagePack.

Таблица 3 — сравнение алгоритмов сериализации

	Protocol Buffers	**MessagePack**	Тип сообщения
Упаковка, нс	43	60	1

Распаковка, нс	75	128	
Размер, байт	2	1	
Упаковка, нс	70	99	[1, "string"]
Распаковка, нс	118	171	
Размер, байт	10	9	
Упаковка, нс	146	138	[1, "string", [1, "string"]]
Распаковка, нс	260	229	
Размер, байт	24	18	
Упаковка, нс	192	198	[1,"string", [1,"string"], [1,256,1024,65536,4294967296]]
Распаковка, нс	375	372	
Размер, байт	42	40	

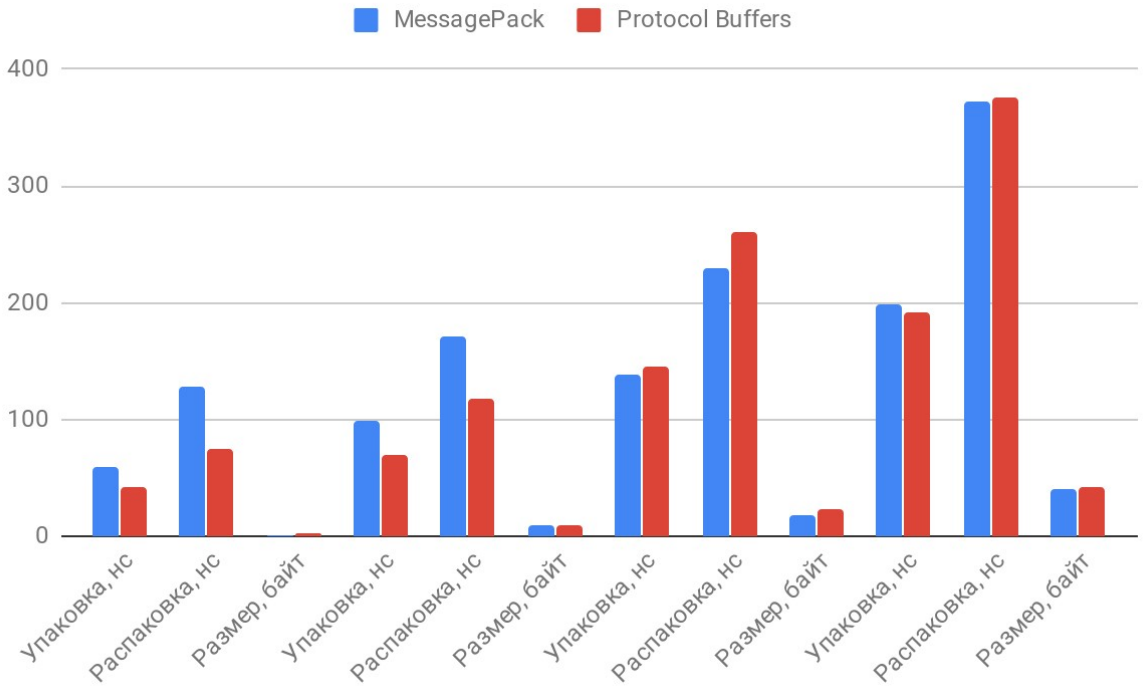

Рисунок 13 — Сравнение алгоритмов сериализации.

Из приведённых сравнений можно заметить следующее:

- MessagePack обеспечивает лучшее сжатие данных. Несмотря на то, что разница во всех случаях составляет единицы байт, при большом числе сообщений эта разница будет накапливаться, что приведёт к гораздо большему использованию памяти нашей системой.
- На простых структурах данных Protocol Buffers обеспечивает лучшую скорость упаковки и распаковки данных, однако при составлении более сложных и больших структур скорость MessagePack практически сравнивается с ним.

По выводам, изложенным выше, было принято решение для разработки нашей системы использовать MessagePack.

5 Результаты разработки системы

По итогам разработки системы были получены следующие результаты:

- Динамическая библиотека на языке C, реализующая весь интерфейс доступа к системе сообщений и взаимодействия с другими приложениями. Данная библиотека с помощью известных средств может быть подключена к программам на практически любом языке программирования. Тот факт, что библиотека разделяемая, означает возможность внесения изменений в библиотеку без необходимости перекомпиляции приложений, которые её используют.
- Интерфейс доступа к библиотеке на языке Python, использующий различные преимущества языка и тем самым позволяющий в упрощённом виде получить доступ ко всем функциям библиотеки.
- Тестовые приложения на языках C и Python, использующие соответствующие библиотеки и успешно взаимодействующие через них.

Схема взаимодействия интерфейса библиотеки и функционала приложений показана на рисунке 12.

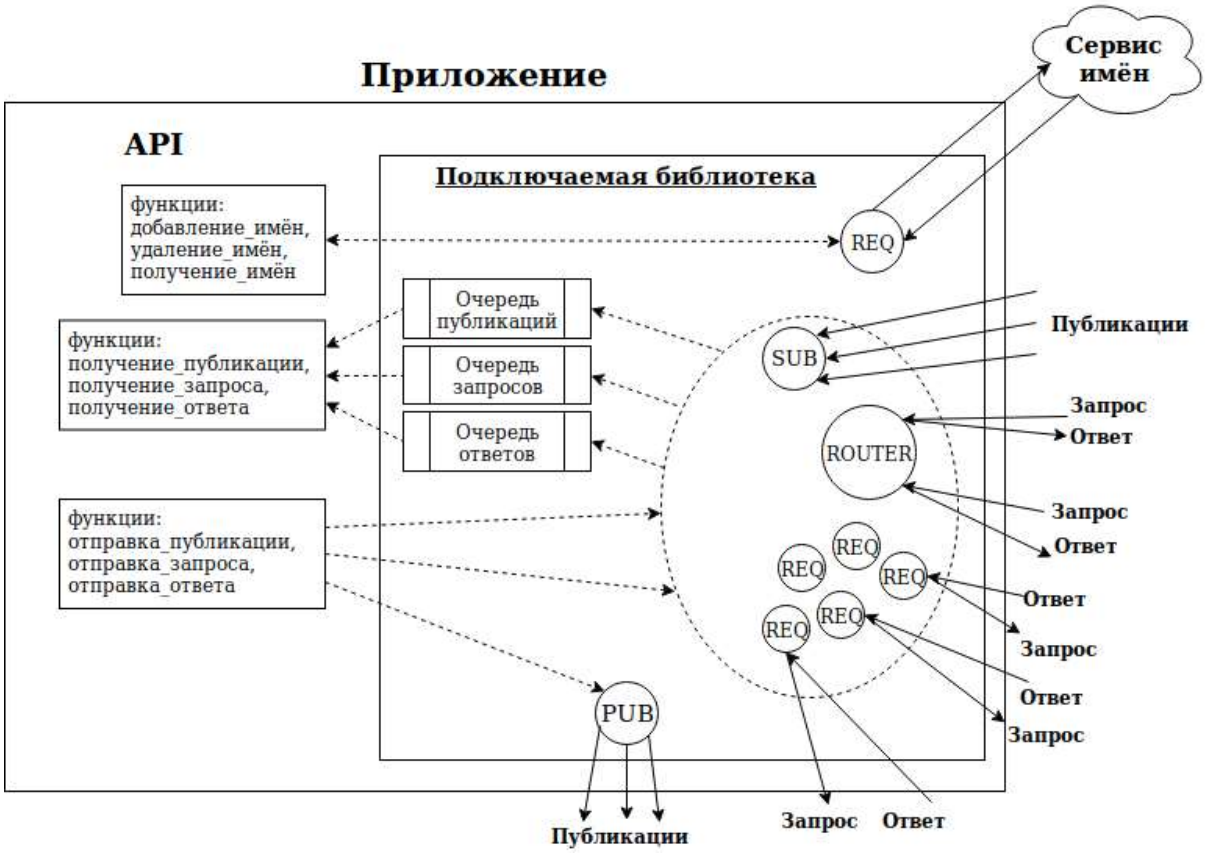

Рисунок 14 — Интерфейс вызовов функций библиотеки.

Интерфейс библиотеки состоит из следующих функций:

- add_name(), del_name(), get_names() — для, соответственно, добавления, удаления и получения имён от сервиса имён. Данные функции отправляют сообщения через сокет REQ, соединённый с сервисом имён, и получают от него ответ.
- publish(), request(), reply() — посылают, соответственно, публикации, а также запросы и ответы приложению с указанным именем. Данные функции отправляют сообщения через сокет PUB для публикаций, через соответствующий сокет REQ — для запросов, и через сокет ROUTER отправляются ответы.

- wait_for_publication(), wait_for_request(), wait_for_reply() — функции получения публикаций, запросов и ответов. Все входящие сообщения хранятся в нескольких очередях, из которых эти функции и получают данные.

По итогам работы также было проведено тестирование производительности разработанной системы на различных топологиях с различными протоколами обмена. Результаты тестирования показаны в таблице 4.

Таблица 4 — Производительность передачи сообщений, протокол «запрос-ответ», полносвязная топология.

Число участников	**Число сообщений**	**Время передачи всех сообщений, мс**
2	10	13
	1000	50
	100000	3987
	1000000	40023
4	10	25
	1000	89
	100000	6547
	1000000	47453
8	10	49
	1000	134
	100000	8967
	1000000	54328
16	10	78
	1000	175

	100000	9845
	1000000	66893
32	10	99
	1000	201
	100000	12565
	1000000	84580
64	10	123
	1000	263
	100000	15108
	1000000	101965
128	10	156
	1000	310
	100000	19842
	1000000	120653

Заключение

В ходе выполнения работы были выполнены все поставленные цели: проведена исследовательская работа в области связующего программного обеспечения, а также была разработана система передачи сообщений.

Данная работа содержит в себе достаточно полную классификацию программного обеспечения, ориентированного на обработку сообщений, что в целом позволяет читателю в краткие сроки ознакомиться с данной сферой, а также помогает принять решение по использованию того или иного программного обеспечения уже в собственных проектах.

С учётом результатов теоретического исследования в ходе выполнения практической части работы была разработана система передачи сообщений, ход разработки которой также может помочь читателям при разработке их собственных решений, так как в данной работе описаны как проблемы, с которыми они могут и будут сталкиваться в ходе разработки, так и некоторые оптимальные методы решения этих проблем.

Разработанная система передачи сообщений также в дальнейшем будет использоваться при производстве различных решений в компании ООО «Эмзиор», где данная разработка уже помогла решить некоторые ранее существовавшие проблемы в области взаимодействия различных программ компании.

Список литературы

1. Dong Jielin, Network Dictionary — Javvin Press, 2007 — 560 с.
2. Etzkorn L.H., Introduction to Middleware: Web Services, Object Components, and Cloud Computing, — CRC Press 2017. — 662 с.
3. Luckham D.C., Event Processing for Business: Organizing the Real-Time Enterprise — John Wiley & Sons, 2011. — 288 с.
4. Gerndt Michael, Performance-Oriented Application Development for Distributed Architectures: Perspectives for Commercial and Scientific Environments — Ios Pr Inc, 2002. — 108 с.
5. David A. Chappell, Enterprise Service Bus — O'Reilly Media, 2004. — 288 с.
6. Simon A.R, Wheeler T., Open Client/Server Computing and Middleware, — Academic Press, 2014. — 288 с.
7. Алиев Т.И., Сети ЭВМ и телекоммуникации — Санкт-Петербург: СПбГУ ИТМО, 2011. — 400 с.
8. Broker vs. Brokerless [Электронный ресурс] — Режим доступа: http://zeromq.org/whitepapers:brokerless (дата обращения: 12.05.2019)
9. Curry Edward, Middleware for Communications — John Wiley & Sons, Ltd, 2004. — 487 с.
10. Johansson Leif, "XMPP as MOM", Greater Nordic Middleware Symposium, University of Stockholm, 2005.
11. What’s wrong with AMQP [Электронный ресурс] — Режим доступа: http://www.imatix.com/articles:whats-wrong-with-amqp (дата обращения: 12.05.2019)
12. YAMI4 vs ZeroMQ [Электронный ресурс] — Режим доступа: http://www.inspirel.com/articles/YAMI4_vs_ZeroMQ.html (дата обращения: 12.05.2019)

13. Middleware trends and market leaders 2011 [Электронный ресурс] — Режим доступа: cern.ch/go/G9RC (дата обращения: 12.05.2019)
14. Таненбаум Э., Уэзеролл Д., Компьютерные сети. 5-е изд. — СПб.: Питер, 2012. — 960 с.
15. ZeroMQ The Guide [Электронный ресурс] — Режим доступа: http://zguide.zeromq.org (дата обращения: 12.05.2019)
16. Differences between nanomsg and ZeroMQ [Электронный ресурс] — Режим доступа: https://nanomsg.org/documentation-zeromq.html (дата обращения: 12.05.2019)
17. E. Curry, D. Chambers, and G. Lyons, "Extending Message-Oriented Middleware using Interception", Third International Workshop on Distributed Event-Based Systems, 2004.
18. ZeroMQ Frequently Asked Questions [Электронный ресурс] — Режим доступа: http://zeromq.org/area%3afaq (дата обращения: 12.05.2019)
19. Protocol Buffers Techniques [Электронный ресурс] — Режим доступа: https://developers.google.com/protocol-buffers/docs/techniques (дата обращения: 12.05.2019)