УДК 004.75

Разработка и оптимизация архитектуры многопользовательского сервера обмена унифицированными сообщениями.

А.А. Петров

Институт проблем передачи информации им. А.А. Харкевича РАН

Введение

В настоящее время большое внимание уделяется техническим задачам распределенных и нагруженных систем. Это связано в первую очередь с ростом количества пользователей сети интернет в последние несколько лет (до 95% всего населения в некоторых европейских странах), а также ее географическим объединением (появляется все больше глобальных единых сервисов, таких как, например, Google). Однако не только количество пользователей диктует новые технические стандарты, но и объемы обрабатываемых и хранимых данных, а также частота их использования. В России по данным ВЦИОМ ежедневно пользуются интернетом более 40% населения, а исследование компании Яндекс показало, что аудитория мобильного интернета растет гораздо быстрее аудитории интернета в целом.

Такой рост меняет и сценарии использования систем, и нагрузку на них. Например, за один день пользователи сети Twitter публикуют более 200 миллионов сообщений, а 900 миллионов пользователей Facebook в день генерируют 3.2 миллиарда сообщений и Like-ов в своих новостных лентах. Таким образом, даже предположив равномерность нагрузки, оказывается, что Facebook обрабатывает более 37 тысяч новых сообщений в секунду, которые в ту же секунду должны быть доступны всем девятистам миллионам пользователей. Этот пример показывает актуальность задачи масштабирования серверных решений по нагрузке и на запись информации (публикацию сообщений), и на ее чтение (доставку пользователям). Подобных примеров в интернете становится все больше, что усиливает интерес к области построения высоконагруженных и распределенных систем.

В данной статье будет описано построение одной из таких систем - сервера обмена унифицированными сообщениями для нужд платформы Penxy. Будут рассмотрены основные принципы построения таких систем, аргументирован выбор платформ и архитектурных решений.

Сравнительный обзор PaaS и IaaS решений, Windows Azure

Одной из важнейших характеристик современных высоконагруженных систем является динамическая масштабируемость. Введем ключевые термины, используемые в статье:

Динамическая масштабируемость *– способность системы в случае роста нагрузки расширять собственные производительные мощности, а также распределять на них необходимые к выполнению операции без потери работоспособности.*

С целью снижения времени, необходимого для такого расширения, создавались автоматизированные системы управления дата-центрами, которые в дальнейшем в силу экономической целесообразности эволюционировали в облачные дата-центры.

Облачные дата-центры *– дата-центры, обеспечивающие автоматическое масштабирование по нагрузке на уровне сервисов и внутренних систем, а также оптимизирующие использование производительных мощностей с целью их перераспределения в случае неравномерных нагрузок. Термин родился как метафора, основанная на изображении дата-центров в виде облаков на диаграмме, и символизирует скрытость внутренней инфраструктуры от внимания разработчиков.*

Облачные дата-центры стали основой для формирования новой ветки развития как моделей коммерциализации (Software as a Service – продажа времени использования программного обеспечения, а не лицензий), так и моделей построения систем в целом. Дата-центры стали предоставлять производительные мощности в виде инфраструктуры (Infrastructure as a Service, IaaS) и в виде платформ (Platform as a Service, PaaS) [1]. Первые решают проблему быстрого запуска дополнительных серверов в случае роста нагрузки. Вторые – предоставляют с одной стороны надежные стандартные программные решения (хранилища данных, распределенные базы данных и пр.) в качестве сервисов, с другой стороны - определяют платформу разработки (язык программирования, API, средства тестирования и отладки и т.д.), удаляя разработчика от решения проблем серверной операционной системы и концентрируя его на уровне приложения.

Горизонтальное масштабирование *– расширение системы путем добавления новых копий оборудования, обрабатывающего запросы или выполняющего задачи, и распределения нагрузки для обеспечения работоспособности системы при ее скачках.*

Вертикальное масштабирование *– расширение системы с точки зрения производительности аппаратного обеспечения конкретного узла. Чаще всего употребляется в смысле отсутствия ограничений программной платформы, т.е. возможности запуска системы на более мощном аппаратном обеспечении в случае увеличения нагрузки.*

Publish-Subscribe *(pubsub-паттерн) – шаблон построения систем обработки данных, основанный на концепции «подписок». В случае «публикации» (вызова метода publish) сообщение будет доставлено всем «подписчикам» (subscribers).*

Брокер сообщений *– архитектурный шаблон построения систем, позволяющих осуществлять прием сообщений от клиентских приложений, их валидацию, обработку и доставку определенной группе пользователей (routing).*

In-memory data-grid *– система хранения и управления данными в оперативной памяти компьютера. Применяются в случаях, когда критичны задержки на запись и чтение информации.*

PaaS и IaaS решения принципиально отличаются тем, на каком уровне разработчик получает возможность управления ресурсами и, соответственно, несет ответственность за верное ими управление (см. Рисунок 1).

Фактически различие наблюдается лишь в трех основных блоках: Runtimes, Security & Integration и Databases. Runtimes означает привязку приложения к конкретной среде исполнения в случае PaaS или же выбор ее в случае IaaS. В случае IaaS на плечи разработчика также ложится работа по развертыванию этой среды исполнения, ее своевременному обновлению и контролю работоспособности. Аналогичные преимущества и проблемы возникают и в остальных двух блоках. В случае Databases в IaaS есть возможность использования, например, баз данных MySQL, однако работы по их поддержке и масштабированию ложатся на плечи разработчика. Это может быть оправдано, например, в случае портирования уже созданного приложения в облако.

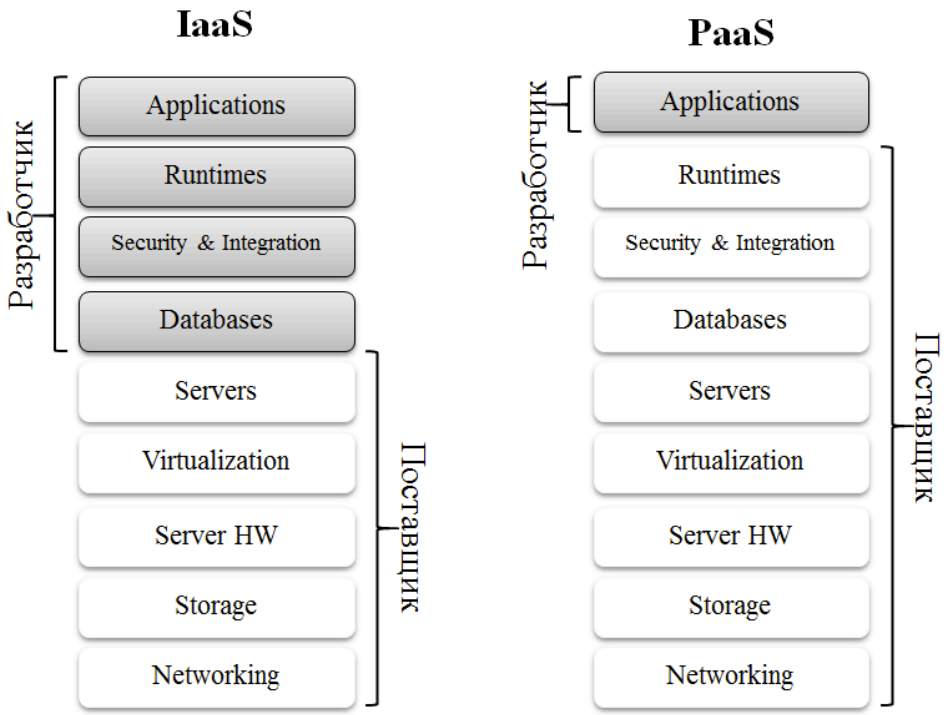

Рисунок 1. *Различия IaaS и PaaS.*

На рисунке 1: Networking – сетевые службы и доступ к сети, Storage – блок хранения данных, Server HW – серверное вспомогательное оборудование, Virtualization – средства виртуализации, Servers – сервера, Databases – базы данных, Security&Integration – блок обеспечения безопасности и интеграции сервисов, Runtimes – среда исполнения, Applications – приложения.

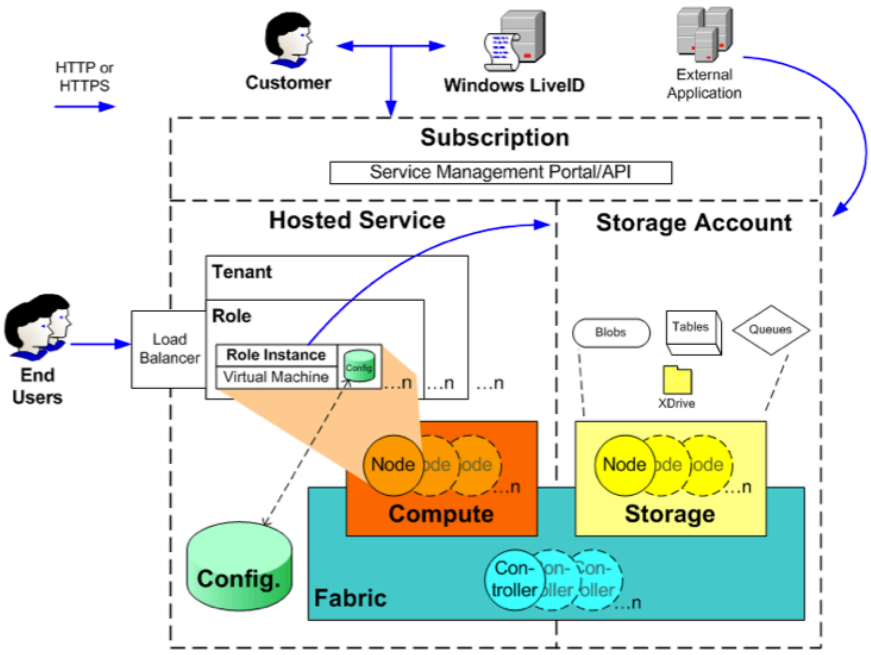

Рисунок 2. *Устройство Windows Azure.*

При выборе платформы для серверного блока Penxy принимались во внимание как сложность и длительность разработки (в этом случае PaaS решения играют немаловажную роль), так и надежность (выбор облачного провайдера с точки зрения надежности предоставляемого сервиса). Наиболее развитым с точки зрения PaaS, а также достаточно гибким (предусматривает гибридную модель использования PaaS и IaaS) и относительно быстрым является Microsoft Windows Azure [2]. Внутреннее устройство логических модулей Windows Azure представлено на Рисунке 2. В качестве среды исполнения в Azure выступает .NET Framework, в качестве баз данных используются встроенные распределенные базы данных Azure SQL. В части Security & Integration в Azure предусмотрено множество средств аутентификации и защиты протоколов общения различных модулей как внутри системы, так и извне. Например, на уровне платформы предусмотрено использование Storage account ключей для доступа к распределенному хранилищу как со стороны внутренних виртуальных машин, так и со стороны внешних приложений.

Требования к брокеру сообщений и протоколу

Система Penxy представляет собой программный продукт, объединяющий в себе на стороне клиентского приложения виртуальный класс и систему записи занятий с возможностью последующего редактирования и использования в образовательном процессе.

Клиентские приложения собирают пользовательские сообщения (отдельные росчерки пера, текстовую информацию, данные о событиях в рамках урока и пр.) в коммуникационные пакеты размером около 500 байт. Частота подобных пакетов-событий может достигать 10 в секунду, при этом интенсивность потока сообщений может быть симметричной по отношению к пользователям (отправляется и принимается сравнимое количество сообщений).

На сервер ложится нагрузка по пересылке этих сообщений, их обработке (совершение определенных действий над учетными записями или правами доступа по факту прихода сообщений определенного типа), индексации (формировании обратного индекса по содержимому пакетов), долгосрочному хранению и поиску.

Важно отметить, что в качестве клиентских приложений могут выступать как мобильные платформы (iOS, Android), стационарные операционные системы (Windows, MacOS, Linux), так и браузеры (Chrome, Safari).

Можно коротко выделить основные требования к брокеру сообщений, составленные на основании исследования и прототипирования системы Penxy:

- Масштабируемость пропускной способности (до 10 сообщений в секунду с возможностью расширения).
- Масштабируемость кол-ва подписчиков (динамическая масштабируемость по количеству подписчиков до 10000 в рамках одной подписки).
- Критичность низкой задержки (минимизация задержки вплоть до 0.1 секунды).
- Большая пропускная способность (около 40к сообщений в секунду).
- Устойчивость к выходу из строя части узлов системы (перестроение без потери работоспособности).

Распределенная система, ограничения CAP-теоремы

Обеспечить растущие потребности в масштабировании, которые при этом не всегда можно предсказать заранее, невозможно средствами одного физического сервера, пусть даже очень мощного. Все подобные задачи требуют распределять нагрузку между несколькими серверами. Это приводит к необходимости иметь дело с целым набором проблем. Необходимо решать задачу синхронизации работы узлов, добиваться согласованности данных, обеспечивать достаточную устойчивость системы в целом и обрабатывать сценарии выхода из строя каждого из узлов.

Тем не менее, чтобы удовлетворить требованию динамической масштабируемости системы необходимо создать кластер в облаке, обладающий свойствами высокой доступности и распределения нагрузки. Очень часто к распределённым системам предъявляют требования полной согласованности данных (Consistency), доступности в любой момент времени на чтение и запись (Availability) и устойчивой к сбоям узлов и потере связности (Partition tolerance). Однако в 2000 году Эриком Брюером была выдвинута гипотеза, впоследствии получившая название **CAP-теоремы**, утверждающая принципиальную невозможность построить распределенную систему, полностью удовлетворяющую всем трем указанным характеристикам [3]. В 2002 году Сетом Джилберт и Нэнси Линч было опубликовано исследование формальной модели асинхронных и синхронных распределённых вычислений, в рамках которой было показано выполнение теоремы CAP в условиях отсутствия синхронизации (общих часов) у узлов распределённой системы и принципиальная возможность компромисса в частично синхронных системах. [4]

Определим основные термины:

Согласованность данных (consistency) — свойство распределенной системы, гарантирующее то, что данные во всех вычислительных узлах в один момент времени не противоречат друг другу. Например, при обновлении или изменении данных на одном из узлов, они должны быть обновлены на всех узлах перед следующим чтением.

Доступность (availability) — свойство распределенной системы, гарантирующее то, что любой запрос к системе завершается корректным откликом.

Устойчивость к разделению (partition tolerance) — свойство распределенной системы, гарантирующее работоспособность системы при потере большого числа пакетов между узлами (фактически – при разделении кластера на части).

Распределенные системы, состоящие из нескольких кластеров (например, разделенные между дата-центрами), обязаны в первую очередь делать ставку на устойчивость к разделению. Что же касается доступности, можно условно выделить два распространенных подхода [5]:

1. Реляционные базы данных, обладающие качеством ACID: Atomicity, Consistency, Isolation, Durability. В терминах CAP теоремы такие системы выбирают согласованность данных. Для достижения целостности данных используются, в том числе, пессимистические блокировки. Они оказывают серьезное влияние на доступность системы, т.к., например, операция записи не считается завершенной, пока информация не будет корректно добавлена в систему (до завершения транзакции). Добавление занимает некоторое время, в течение которого систему можно считать недоступной. Это считается основным недостатком транзакционных систем в целом, однако обеспечивает абсолютную согласованность и целостность данных, что важно, например, в банковской сфере и системах биллинга.
2. NoSQL решения, характеризующиеся высокой доступностью и идущие на компромисс в целостности данных. Обычно это выражается в том, что операция записи в такую систему асинхронна (операция записи считается завершенной сразу после того, как система получила данные, но не обязательно записала их). В таких системах возможна ситуация, когда еще до полного распространения по системе изменений клиент может начать читать устаревшие данные (write-read коллизия). Такие системы называются Eventual Consistent (*англ.* “согласованные в конечном счете”), т.к. они становятся согласованными через некоторое время. В случае если время, через которое данные потребуются на чтение, превосходит период синхронизации системы, NoSQL являются идеальным решением, т.к. при

правильном распределении они практически не имеют предела по производительности на запись (критично, например, в случае высоконагруженных систем логирования).

В ситуации, когда возможно разделить данные «по ключам» - в нашем случае «по подпискам», - можно добиться масштабируемости на запись (write scalability) и без Eventual Consistency. Действительно, в Penxy квантом нагрузки является подписка (замкнутая группа пользователей, общающихся внутри одного урока или лекции), поэтому часть узлов может «не знать» об изменениях внутри конкретной подписки, что снимает с системы требование полной согласованности данных. Достаточно выделить несколько групп узлов, обслуживающих часть данных (в соответствии с их разбиениями). Каждый раз, когда нагрузка на запись будет достигать порогового значения, мы будем увеличивать количествово таких групп, перенося уже существующие данные на соответствующе узлы. Такая стратегия называется шардирование (Sharding). С помощью этого подхода в нашем случае мы можем избежать потери целостности данных в задаче записи новых данных в систему, но по-прежнему остается задача масштабируемости на чтение и обеспечения надежности.

Обеспечения требований по низкой задержке в передаче данных

До этого момента мы не касались требований на высокую пропускную способность и низкую задержку. У большинства реляционных баз данных есть общее узкое место – скорость доступа к диску. Это связано с достаточно низкой скоростью чтения и записи на жесткий диск (HDD в сравнении с SSD). В сценарии Penxy необходимо постоянно обращаться к случайным блокам данных (чтение и запись сообщений в рамках общения внутри класса), сократив при этом временные затраты до минимума (задержки в случае росчерков пера или сообщений, управляющих физической моделью, критичны). Для этой задачи логичным выглядит использование Random Access Memory – оперативной памяти компьютера (т.к. обеспечить кластер винчестерами с SSD невыгодно с экономической точки зрения). С одной стороны это ограничивает нас в максимальном объеме данных, хранимых на узле, и создает проблемы возможной потери этих данных в случае потери электропитания узла. С другой стороны – предоставляет уникальные характеристики производительности. Принимая решение об использовании таких систем, необходимо закладывать

возможность репликации данных на жесткий диск для защиты от потери данных в случае потери электропитания.

В качестве базового узла в нашем решении мы будем использовать NoSQL решение Redis – документо-ориентированное сетевое журналируемое хранилище данных типа «ключ-значение» с открытым исходным кодом, хотя в данном случае можно использовать и его аналоги. Оно работает в режиме in-memory (в RAM) и обеспечивает постоянную низкую задержку на запись и пересылку сообщений, и достаточно большую пропускную способность на единственном узле, поддерживая при этом как функцию регулярного создания копий данных из RAM на диске, так и репликацию типа master-slave на несколько узлов. Redis работает на большинстве POSIX систем, таких как Linux, *BSD, OS X, без каких-либо дополнений. Примечательно, что для управления Redis существует множество библиотек, в том числе и на языке C#, однако официальной версии под Windows до сих пор нет. Портирование на Azure было выполнено самостоятельно в рамках этой работы, опыт портирования был описан в докладе на конференции для разработчиков Microsoft DevCon'12 в мае 2012 года [6]. В дальнейшем планируется реализация шардирования Redis-узлов с простейшим балансировщиком нагрузки в рамках Azure, однако компания Microsoft заявляет о собственных планах в этом направлении, поэтому не исключено, что скоро выйдет официально поддерживаемая версия Redis как часть сервисов облака.

Условие устойчивости к выходу из строя

Самый распространенный способ достижения надежности – использование репликации ведущий-ведомые (master-slave). В этом случае создаются и поддерживаются две или более идентичных копии узлов, одна из которых считается «ведущим» (или «мастером») и принимает все запросы на изменения. После чего синхронно или асинхронно эти данные копируются на ведомых. Одним из достоинств баз данных в оперативной памяти является то, что можно без особых задержек проводить репликацию синхронно: т.е. не завершать атомарную операцию записи, пока измененные данные не будут записаны на «мастер», и на «ведомых». Таким образом, мы избегаем возможных коллизий чтения-записи. При этом потраченное время может быть даже меньше, чем запись на один узел реляционной базы данных, использующей жесткий диск для хранения. Асинхронная репликация при этом так же возможна,

преимущества хранения в оперативной памяти в данном случае проявляются в виде высокой скорости распространения информации по системе (короткий цикл достижения целостности данных).

Еще одним достоинством такой системы является то, что мы можем добиться и произвольной масштабируемости на чтение. Достаточно все операции чтения перенаправить на ненагруженные узлы «ведомых». При синхронной репликации мы можем быть уверены, что в любой момент времени данные на этих узлах актуальны – а значит, целостность данных сохранена, при асинхронной – целостность достигается через некоторое небольшое время. На рисунке 3 представлена схема чтения и записи информации в такую систему в асинхронном режиме.

В случае выхода из строя ведущего узла (например, при падении напряжения или принудительной перезагрузке узла, инициированной платформой Azure) один из ведомых узлов автоматически берет на себя нагрузку по записи и удаляется из списка узлов чтения, становясь, таким образом, новым ведущим. Диагностика работоспособности узлов может идти либо централизованно (контролирующим узлом может быть, например, балансировщик нагрузки в случае, если система состоит из нескольких ведущих узлов, а значит его наличие необходимо), либо посредством организации протокола «слухов», т.е. оповещения всех ведомых узлов системы о всех ошибках общения с ведущим. В случае накопления определенного количества ошибок, на стороне ведомых поведение ведущего признается неудовлетворительным и один из ведомых узлов занимает его место (проставляет данные маршрутизации вызовов, отдает сигнал ведущему на перезагрузку, оповещает ведомых о своей новой роли). Этот вариант алгоритмически более сложный, однако позволяет экономить на центральном администрирующем узле, а также лишен уязвимости потери управляющего узла (в общем случае он также должен состоять из двух реплик для обеспечения надежности). Определение того, какой из ведомых должен занять место ведущего, можно вести, например, по минимальному номеру идентификатора, который ведомый получает, входя в систему (при загрузке из протокола «слухов» определяет максимальный идентификатор в системе и назначает себе ID + 1).

При возвращении в строй, старый ведущий может вернуть себе свою роль или же исполнять функции ведомого. Это зависит от того, был ли выбран в качестве изначального ведущего узла узел с уникальными производительными характеристиками относительно ведомых (например, large instance VM в терминах облаков, против small instance VM ведомых узлов). Такая гетерогенная с точки зрения инфраструктурной конфигурации система оказывается дешевле, однако вызывает

временную деградацию по пропускной способности на запись в случае выхода ведущего из строя.

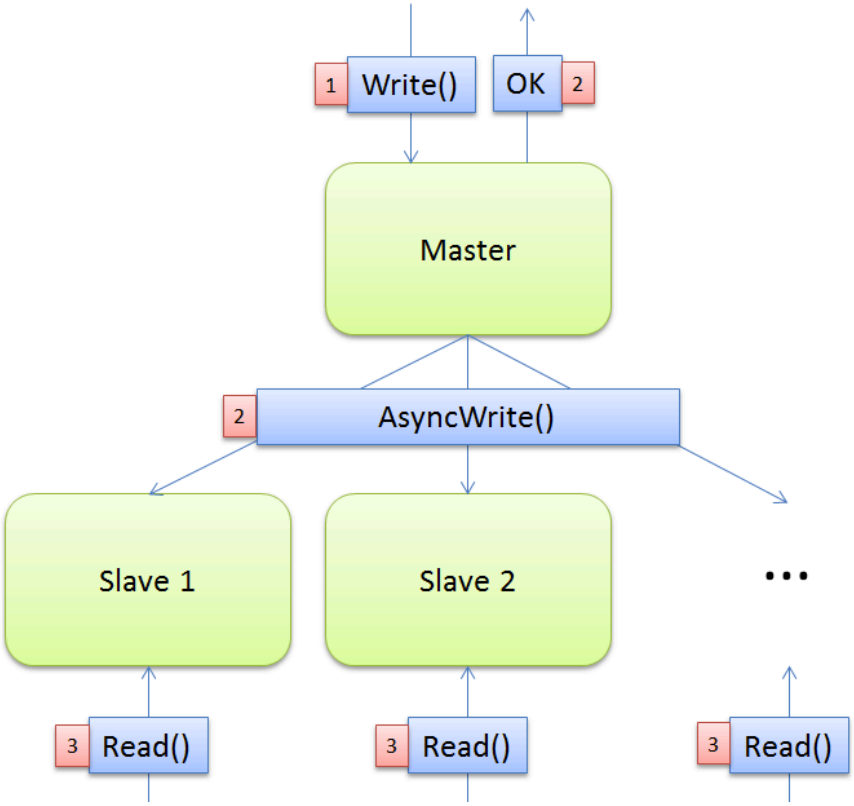

Рисунок 3: *Схема асинхронной репликации узлов.*

Удовлетворение требованиям, оптимизация

Использованием хранилища Redis мы добиваемся постоянной низкой задержки при любой нагрузке на систему, а также большой пропускной способности даже на одном узле Message Broker-а. Используя стратегию Sharding, мы разделяем различные подписки между группами узлов кластера, обеспечивая масштабирование на запись. Внутри каждой группы мы храним несколько синхронных копий данных (репликация ведущий-ведомые), обеспечивая масштабируемость на чтение и возможность «горячей замены», в случае выхода узла из строя (Рисунок 4).

Покажем работу этих принципов на практике. Рассмотрим диаграмму компонентов брокера сообщений (Рисунок 5) и опишем сценарий доставки сообщения отправителя произвольному количеству адресатов.

На этой диаграмме сервера за связь с клиентом отвечают два интерфейсных блока (FrontEnd). Из них за передачи унифицированных сообщений отвечает блок коммуникационных интерйейсов (Communication FrontEnd). Он представляет собой

некоторое (произвольное в зависимости от нагрузки) количество экземпляров узлов, не зависящих от состояния системы. Нагрузка между ними перераспределяется по алгоритму Round Robin, т.к. он является самым простым в реализации и не требует контроля нагрузки на узлах. Каждый запрос от отправителя, поступающий на эти экземпляры, поступает на вход кластера брокера сообщений, обозначенного Redis PubSub. Там в первую очередь происходит «Sharding» запроса – т.е. определяется, какая часть кластера должна отвечать за данный запрос. После этого внутри этой части кластера запрос с сообщением записывается на ведущий узел в очередь доставки и реплицируется на все ведомые узлы. После чего на каждом ведомом узле, для каждого читающего клиента происходит доставка сообщения из очереди.

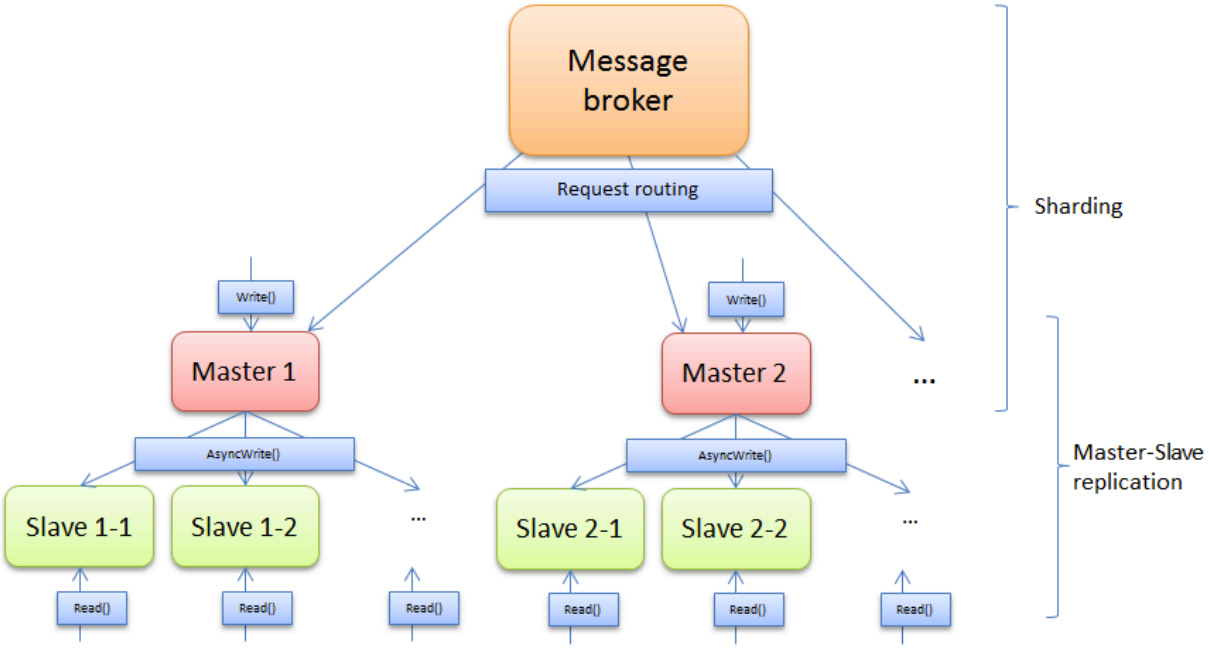

Рисунок 4: *Диаграмма брокера сообщений серверного блока Penxy .*

Таким образом, в случае увеличения нагрузки по количеству клиентов возможно без смены конфигурации запустить несколько новых FrontEnd-ов. В случае увеличения потока сообщений возможно увеличить кол-во экземпляров узлов Message Broker-а.

Рассмотрим функциональное назначение каждого из узлов:

Communication FrontEnd. Данный блок, как уже было описано выше, фактически выполняет функции прокси и маршрутизатора для брокера сообщений, основанного на Redis. Приходящие на него сообщения шардируются по подпискам и отправляются на Redis PubSub. Проводя аналогию с Рисунком 6, Communication FrontEnd фактически выполняет функции блока Message broker. Кроме этого, Communication FrontEnd отвечает за распределение нагрузки внутри узлов Redis, поддерживание соединений с клиентами и авторизацию. Авторизационные ключи

доступа хранятся в аналогичном хранилище Redis на узле Redis Authorization Cache. В зависимости от выставленной степени защиты, возможна авторизация каждого конкретного запроса или SSL-шифрование канала связи между этим узлом и клиентским приложением.

Redis PubSub. Этот блок фактически выполняет функции брокера сообщений и состоит из некоторого числа ведущих узлов Redis, а также их ведомых репликаций. Важно отметить, что каждый из ведущих узлов не подозревает о наличии и состоянии остальных, распределение нагрузки частично ведется на уровне Communication FrontEnd, а управление подписками в Management FrontEnd - через интерфейс Manage RPS.

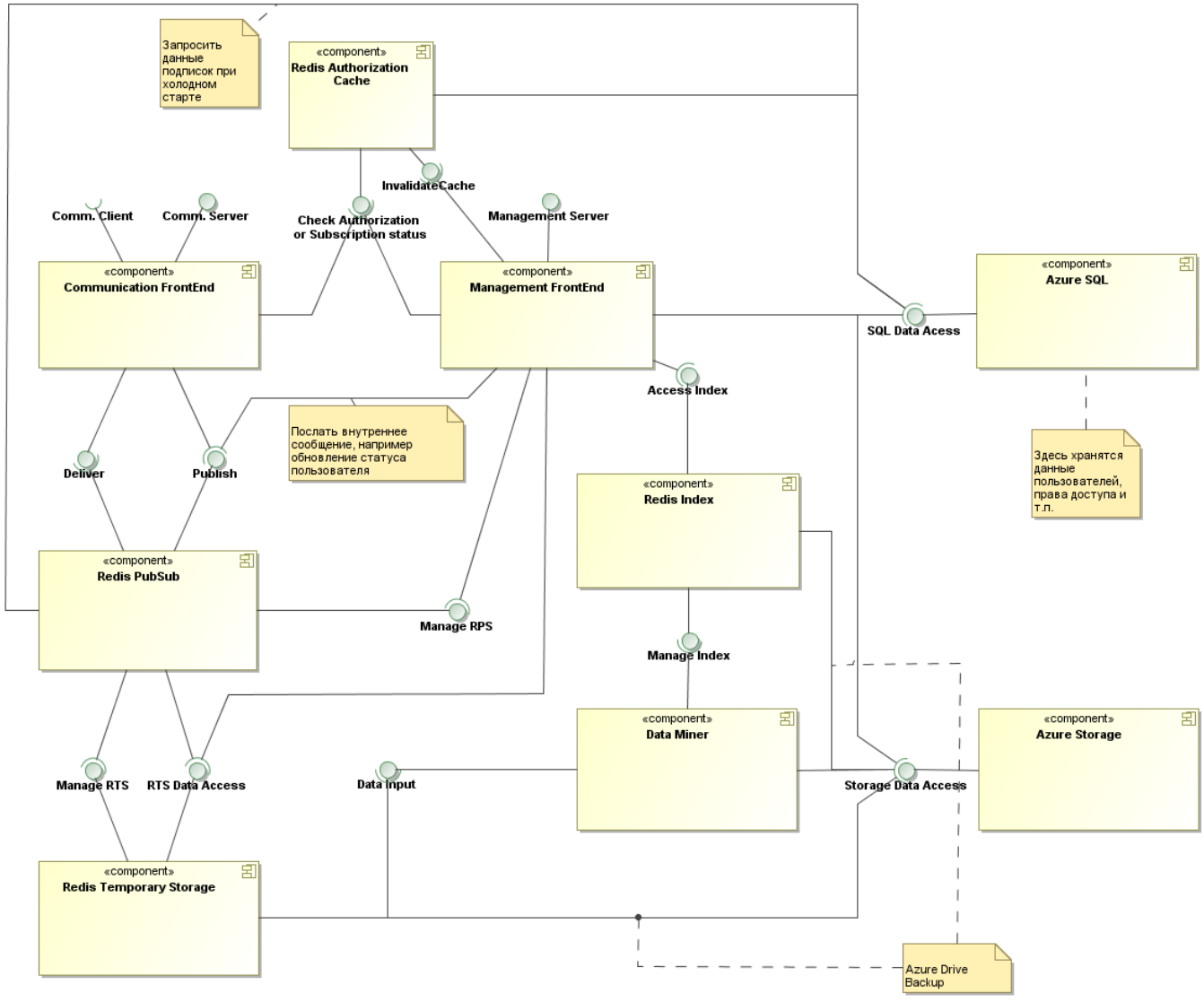

Рисунок 5: *UML-диаграмма компонентов серверного блока Pепху .*

Management FrontEnd. Блок отвечает за

- авторизацию пользователей, обращающихся по интерфейсу Management Server,

- формирование ключей доступа и их инвалидацию,
- управление подписками Redis PubSub,
- формирование поисковых запросов к индексу,
- формирование и управление учетными записями Azure SQL,
- публикацию управляющих сообщений,
- чтение устаревших данных сессий из Redis Temporary Storage в случае, если один из подписчиков потерял на некоторое время связь с сервером, и данные его подписки были удалены из основного Redis PubSub.

Redis Authorization Cache. Блок отвечает за хранение временных авторизационных ключей сессий. Все ключи автоматически обновляются по вызову Invalidate со стороны Management FrontEnd-а, новые ключи получаются из Azure SQL.

Redis Index. Блок отвечает за хранение поискового индекса по системе. В Penxy предусмотрены два отдельных типа индекса, а также фильтрация поисковой выдачи по правам пользователя. Фактически осуществляется поиск по четырем различным индексам для каждого поискового запроса: публичный полнотекстовый индекс (с фильтрацией по правам доступа к конкретному документу), приватный полнотекстовый индекс (индекс содержимого документов, доступных для чтения только этому пользователю, введен для оптимизации поиска), публичный и приватный тег-индексы. Таким образом, система хранит 2N+2 индекс, где N – число пользователей системы. Это позволяет снизить скорость поиска, а также существенно упросить алгоритмы фильтрации по правам, удалив из публичного поиска большое число документов, принадлежащих лишь одному пользователю.

Redis Temporary Storage. Этот блок предназначен для хранения временной информации сессий и передачи данных блоку Data Miner для последующей индексации и складирования в долгосрочное хранилище. Если один из пользователей из-за перебоев сети отключился от подписки, и данные его подписки были удалены Redis PubSub как «доставленные», Management FrontEnd имеет возможность получить потерянные во время отсутствия данные именно в Redis Temporary Storage.

Data Miner. Данный блок предназначен для обработки поступающих данных из подписок в режиме реального времени. В Penxy предусмотрена система автоматического сохранения всех данных в хранилище (часть концепции «облачного учебного пространства»), поэтому данные сразу после доставки пользователей попадают в Data Miner, который формирует обновление поискового индекса.

Azure SQL. Блок, хранящий системные данные: учетные данные, данные о текущих активных подписках, историю подписок, временные и постоянные авторизационные ключи, новостные ленты.

Azure Storage. Блок, хранящий долгосрочные документные данные: пользовательские документы (как уже сформированные, так и еще формирующиеся), все реплики Redis узлов. Как уже описывалось выше, Redis предусматривает репликацию данных на жесткий диск, однако специфика Azure (возможность принудительного уничтожение instance системой) диктует необходимость дополнительной репликации, для которой в Penxy используется Azure Storage.

Из описания может показаться, что для обеспечения функциональности системы необходимо минимум 8 instance-ов в облаке (9 по количеству указанных узлов минус 2 узла Azure-сервисов плюс один узел slave-Redis), однако при малых нагрузках система поддается глубокой интеграции и оптимизации с точки зрения использования производительных ресурсов. Например, в стартовой конфигурации предусматривается лишь 4 сущности в облаке:

1. Сервер Management FrontEnd.
2. Сервер коммуникаций и Redis. Объединяет в себе несколько узлов, построенных на базе Redis (Redis Authorization Cache, Redis PubSub, Redis Index) и Communication FrontEnd.
3. Дублер второй сущности по линии Redis PubSub (slave-Redis).
4. Сервер временного хранилища и формирования индекса: Redis Temporary Storage и Data Miner.

Такая конфигурация с одной стороны уменьшает количество арендуемых производительных мощностей, а с другой – не разрывает связи, критичные для надежности системы (например, при «падении» второй сущности, по прежнему доступно временное хранилище Redis для восстановления утерянных данных).

Заключение

В работе было было показано, как на основе технологий низкого уровня можно по заданным требованиям построить высоконагруженный сервис пересылки, индексации и хранения данных. Рассмотрены ключевые теоретические проблемы масштабирования на запись и чтение, а также отказоустойчивости распределенной облачной системы.

Одним из наиболее важных практических выводов из данной работы может служить то, что распространенное мнение о справедливости CAP-теоремы не является верным, т.к. в случае узких сценариев (например, гранулярности потоков, как в случае подписок внутри урока Penxy) можно достичь всех трех свойств распределенной системы – целостности, к доступности и устойчивости разделению, - одновременно.

Список литературы

1) Gillam, Lee. “Cloud Computing: Principles, Systems and Applications”. Nick Antonopoulos, Lee Gillam. — L.: Springer, 2010. — 379 p. — (Computer Communications and Networks). — ISBN 9781849962407
2) CloudCmp: Comparing Public Cloud Providers, Ang Li Xiaowei Yang Srikanth Kandula Ming Zhang , IMC 2010, Melbourne
3) Brewer, Eric A. “Towards robust distributed systems”. Proceedings of the XIX annual ACM symposium on Principles of distributed computing. — Portland, OR: ACM, 2000. — Т. 19. — № 7. — DOI:10.1145/343477.343502
4) Nancy Lynch and Seth Gilbert, “Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services”, ACM SIGACT News, Volume 33 Issue 2 (2002), pg. 51-59.
5) Jack Belzer. Encyclopedia of Computer Science and Technology - Volume 14: Very Large Data Base Systems to Zero-Memory and Markov Information Source. Marcel Dekker Inc. ISBN 0-8247-2214-0.
6) Windows azure и открытые технологии: полнотекстовый поиск apache lucene, распределенное кэширование данных redis. Конференция TechDays, Матвиенко М.В., http://www.techdays.ru/videos/4435.html