Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение
высшего профессионального образования
«Московский физико-технический институт
(государственный университет)»

Факультет управления и прикладной математики

Кафедра информатики

НАПИСАНИЕ РАСПРЕДЕЛЕННОЙ СИСТЕМЫ КОНСЕНСУСА НА ОСНОВЕ АЛГОРИТМА RAFT И ИЗУЧЕНИЕ ВЛИЯНИЯ СЕТЕВЫХ СБОЕВ НА ПРОИЗВОДИТЕЛЬНОСТЬ

Выпускная квалификационная работа
(бакалаврская работа)

Направление подготовки: 010900 Прикладные математика и физика

Выполнил:
студент 176а группы Родина София Викторовна

Научный руководитель:
д.ф.-м.н., профессор Тормасов Александр Геннадьевич

Научный консультант: Непорада Андрей Леонидович

Москва 2015

Содержание

1 Введение

Представим себе распределенную систему обработки информации, представляющую собой кластер серверов. Если для одного сервера вероятность отказа мала и зачастую при внедрении простых систем ей можно пренебречь, то для кластера серверов вероятность отказа одного из серверов становится в разы больше: MTBF(mean time before failure - наработка на отказ) для одного из N серверов в N раз меньше чем MTBF для одного сервера. Добавим к этому ненадежность сети в виде отказа сетевого оборудования и потери пакетов, отказы жестких дисков, сбои серверного ПО на уровне ОС и приложений. Если верить Google, то для кластера из 1800 машин они говорят о 1000 отказах серверов в течении первого года эксплуатации кластера, то есть 3 отказа в день – и это не считая отказов жестких дисков, проблем с сетью и питанием и т.д. В итоге, если не закладывать отказоустойчивость в ПО распределенной системы, мы получим систему, в которой каждая из указанных выше проблем приводит к отказу системы.

Поэтому задача достижения консенсуса – задача получения согласованного значения группой участников в ситуации, когда возможны отказы отдельных участников, предоставление ими некорректной информации, искажения переданных значений средой передачи данных. В целом сценарии нештатного функционирования компонент распределенных систем можно разделить на два класса:

1. Полный отказ компонента. Характеризуется этот класс проблем тем, что такой отказ приводит к недоступности одного из компонент распределенной системы (или сегментации сети, в случае отказа коммутатора). К этому классу проблем относятся: отказ сервера, отказ системы хранения данных, отказ коммутатора, отказ операционной системы, отказ приложения.
2. Византийская ошибка. Характеризуется тем, что узел системы продолжает функционировать, но при этом может возвращать некорректную информацию. Допустим, при использовании оперативной памяти без ECC может привести к считыванию некорректных данных из памяти, ошибки сетевого оборудования могут приводить к повреждению пакетов и т.п.

Ошибки второго класса намного более сложны в обнаружении и исправлении. В целом, Лесли Лампортом было доказано, что для исправления Византийской

проблемы в N узлах распределенная система должна состоять как минимум из $3N+1$ узлов и должна реализовывать специальный алгоритм консенсуса. Отказоустойчивость на этом уровне требуется по большей части в системах, критичность функционирования которых крайне высока (например, в задачах космической промышленности).

В кластерных вычислениях под отказоустойчивостью обычно понимают устойчивость системы к полным отказам компонент. Для достижения консенсуса в таких системах обычно применяются алгоритмы Paxos и Raft.

2 Обзор

2.1. Понятие распределенной системы

Типичная распределенная система представляет собой набор автономных узлов, состоящих из памяти и процессора, соединенных коммуникационной сетью.

Определение. (Таненбаум) Распределенной системой называется набор независимых компьютеров, представляющийся их пользователям единой объединенной системой.
Определение. (Ж.Тель) Под распределенной системой мы понимаем всякую вычислительную систему, в которой несколько компьютеров или процессоров так или иначе вступают во взаимодействие. Под это определение наряду с глобальными коммуникационными вычислительными сетями попадают локальные сети, многопроцессорные компьютеры, в которых каждый компьютер наделен собственным устройством управления, а также системы взаимодействующих процессов.

Главные характеристики распределенных систем:

1. От пользователей скрыты различия между компьютерами и способы связи между ними.
2. Способ, при помощи которого пользователи и приложения единообразно работают в распределенных системах, независимо от того, где и когда происходит их взаимодействие.
3. Легкость масштабирования.

Особенности распределенных систем:

1. Отсутствие общих физических часов.
2. Отсутствие общей памяти и взаимодействие путем посылки сообщений
3. Асинхронная связь и асинхронное исполнение в качестве базовых моделей.
4. Географическое распределенность.
5. Прозрачность распределенности. Набор независимых компьютеров, которые представляются единым компьютером
6. Отказоустойчивость. Отказ компонента не требует прекращения работы
7. Автономность и гетерогенность. Узлы слабо связаны. Они кооперируют предлагая сервисы совместного решения проблемы.

2.2. Постановка задачи консенсуса

Консенсус (лат. *consensus* — согласие) — способ принятия решений при отсутствии принципиальных возражений у большинства заинтересованных лиц, принятие решения на основе общего согласия без проведения голосования, если против него не выступает никто, либо при исключении мнения немногих несогласных участников.

2.3. Отказоустойчивость

Возможность частичного отказа - характерная черта распределенных систем.

Неполадки обычно выводят из строя только некоторую часть всей системы. И хотя с ростом числа компонентов системы стремительно возрастает и вероятность отказа одного из этих компонентов, столь же стремительно уменьшается вероятность выхода из строя всех компонентов системы.

Отказоустойчивость тесно связана с понятием надежных систем (dependable systems).

Основные требования к надежности:

1. Доступность (availability). Свойство системы находиться в состоянии готовности к работе.
2. Безотказность (reliability). Свойство системы работать без отказов в течение продолжительного времени.
3. Безопасность (safety). Определяет, насколько катастрофична ситуация временной неспособности системы должным образом выполнять свою работу.
4. Ремонтопригодность (maintainability). Определяет, насколько сложно исправить неполадки в описываемой системе.

Говорят, что система отказывает (fail), если она не в состоянии выполнять свою работу. Ошибкой (error) называется такое состояние системы, которое может привести к ее неработоспособности. Причиной ошибки является отказ (fault).

Построение надежных систем тесно связано с управлением отказами. Управление в данном случае означает нечто среднее между предотвращением, исправлением и предсказанием отказов.

Отказоустойчивость (fault tolerance) - способность системы предоставлять услуги даже при наличии отказов.

Система будет называться устойчивой к k отказам, если она останется работоспособной после отказа k компонентов. Обычно наличия k + 1 процессов достаточно, чтобы обеспечить устойчивость к k отказам. Если k из них прекратят работу, будет использован ответ от одного оставшегося.

В случае **византийских ошибок**, когда процесс продолжает работать, рассылая ошибочные или просто случайные сообщения, для обеспечения устойчивость и к k отказам необходимо как минимум 2k + 1 процессов.

В наихудшем случае ошибки в k процессах могут случайно (или даже намеренно) породить одинаковые результаты. Однако остальные k + 1 процессов также дадут одинаковые результаты, так что клиент или устройство голосования смогут все же поверить большинству.

Основная задача алгоритмов распределенного соглашения состоит в том, чтобы привести все безошибочные процессы к согласию по определенным вопросам и к тому, чтобы достичь этого согласия за конечное число шагов.

В теории имеется ряд основополагающих результатов свидетельствующий:

1. Об отсутствии детерминированного консенсуса для асинхронных систем
2. О том, что вероятностные алгоритмы устойчивы по отношению к «доброкачественным» формам неисправности, охватывающим не более половины процессов, и к «злокачественным» формам неисправности, охватывающим не более одной трети процессов
3. О том, что синхронные системы являются более стойкими чем асинхронные

FLP теорема. В работе Фишера, Линч и Патерсона был получен важный результат (FLP теорема), гласящий, что не существует детерминированного алгоритма достижения консенсуса, который был бы устойчив даже по отношению выходу из строя одного процесса.

В дальнейшем было показано что при ослаблении допущений об устройстве системы можно получать практически важные решения, в частности:

1. Ослабленная модель отказа. В модели изначально бездействующих процессов, которая является более слабым типом неисправности, детерминированные алгоритмы могут решать задачи достижения консенсуса и избрания лидера.
2. Ослабленная координация. Существуют задачи, в которых требуется не столь тесная согласованность действий процессов системы, как это требуется для решения проблемы консенсуса. Например, для задачи переименования, существуют решения даже в модели выходящих из строя процессов.
3. Рандомизация. В случае рандомизованных протоколов ослабленное условие завершаемости позволяет получать решения даже в случае

византийского поведения неисправных процессов.

4. Ослабленное условие завершаемости. Если потребовать, чтобы вычисления обязательно завершали только исправные процессы, то решение возможно для византийского поведения неисправных процессов.
5. Синхронность. Очень существенно условие синхронности.

2.4. Алгоритмы достижения консенсуса

Итак, положим у нас есть набор процессов, каждый из которых может предложить некоторое значение. Алгоритм консенсуса гарантирует, что только одно из предложенных значений будет выбрано. Если никакое значение не было предложено, то никакое значение не может быть выбрано. Если же значение было выбрано, то процесс должен иметь возможность узнать это значение. Процесс не может узнать о значении до того, как оно было выбрано.

Мы используем привычную асинхронную невизантийскую модель:

1. Агенты работают с произвольной скоростью, могут давать сбои путем отключения питания и могут перезапускаться. Поскольку все агенты могут отключаться после того, как значение было выбрано и потом перегружаться, решение проблемы невозможно, пока часть информации не “запоминается” этим агентом.
2. Сообщения могут иметь произвольную длину, могут дублироваться, теряться, но не могут быть повреждены.

Самый известный алгоритм для решения поставленной проблемы - алгоритм Paxos. Семейство протоколов Paxos рассматривает варианты решения задачи консенсуса в зависимости от количества вычислителей, количества задержек для получения результата, активности участников, количества отправленных сообщений и типов отказов. Результат отказоустойчивого консенсуса не

определён (то есть, при определённых условиях вычислители не смогут прийти к согласию), однако, гарантируется безопасность (согласованность), а условия, при которых консенсус невозможен, очень редки. К сожалению, многие подробности реализаций этого семейства протоколов малоизвестны, поэтому для реализации протокола компаниям приходится искать собственный подход. И все же, даже существующие решения кажутся непростыми для понимания.

Алгоритм Raft был создан как альтернатива Paxos'у с целью упростить подход к задаче и получить открытый подробный алгоритм решения задачи консенсуса.

В данной работе будет описан алгоритм Raft - относительно новый алгоритм достижения консенсуса в сети ненадежных вычислителей.

3 Алгоритм Raft

3.1. Введение

Алгоритмы консенсуса позволяют набору машин(кластеру) функционировать в качестве согласованной группы, которая может выдерживать перебои в работе части своих членов. По этой причине они играют важную важную роль в построении надежных масштабируемых программных систем. Paxos являлся стандартом де-факто построения таких систем на протяжении последнего десятилетия: большинство реализаций консенсуса базируются на этом семействе алгоритмов.

К сожалению, Paxos считается сложным для понимания, также построение подобной архитектуры требует больших изменений для поддержки реальных систем.

Алгоритм Raft стал результатом научной работы Диего Онгаро(Ph.D. студента Стэндфордского института), целью которой было построение ***легкого для понимания*** алгоритма консенсуса: алгоритм должен не просто работать, но должно быть очевидно как именно он работает. Raft во многом похож на существующие алгоритмы консенсуса, но есть и некоторые важные отличия:

1. ***Сильная форма лидерства:*** все данные направлены только от лидера к остальным серверам. Это упрощает управление журналом копий и делает Raft более понятным.
2. ***Выбор лидера:*** Raft использует рандомизированные таймеры для выбора лидера.

3.2. Основы алгоритма Raft

Реплицированный конечный автомат (replicated state machine).

RSM не отличается от обычного конченого автомата. Он просто исполняет команды, которые считаются необходимыми к исполнению и по сути своей представляет хранилище данных с возможностью эти данные удалять/добавлять/изменять.

Каждый сервер содержит свой RSM и сопряженный с ним журнал копий. RSM обычно работает при помощи журнала копий. Каждый сервер хранит журнал, содержащий набор команд, которые конечный автомат выполняет по очереди. Каждый журнал содержит одни и те же команды в одинаковом порядке, поэтому каждый конечный автомат на любой машине исполняет одну и ту же последовательность команд.

При таком подходе, RSM вычисляет идентичные копии состояний и может продолжать операции даже после того как часть серверов пришли в нерабочее состояние.

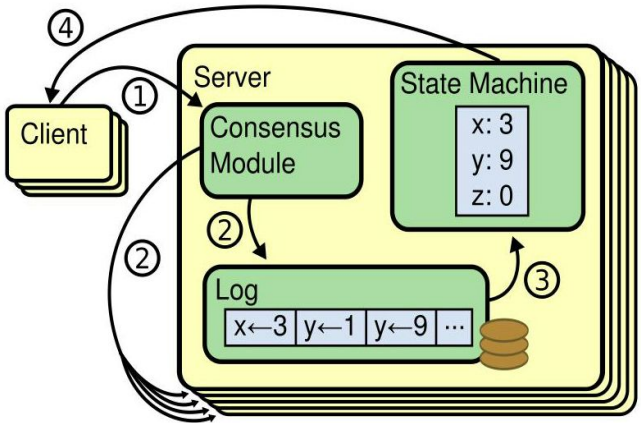


Рис.1. Архитектура RSM.

Raft реализует консенсус прежде всего при помощи выборов лидера. Лидер получает полную ответственность за управление журналом копий. Лидер принимает запросы от клиентов, копирует их на другие сервера и сообщает им когда становится безопасно применять эти запросы к RSM.

При подобном подходе(имеется единственный лидер), Raft декомпозирует задачу достижения консенсуса на 3 независимых подзадачи:

1. Выбор лидера(leader election): новый лидер должен быть выбран, когда старый лидер считается “умершим”.
2. Копирование журнала(log replication): лидер должен принимать запросы от клиентов и копировать их на весь кластер, вынуждая остальных соглашаться со своим решением.
3. Безопасность(safety): если хотя бы один сервер применил конкретный запрос к своему RSM, то никакой другой сервер не может применить другой запрос для того же индекса журнала.

State

Persistent state on all servers:
(Updated on stable storage before responding to RPCs)

currentTerm	latest term server has seen (initialized to 0 on first boot, increases monotonically)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)

Volatile state on all servers:

commitIndex	index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied	index of highest log entry applied to state machine (initialized to 0, increases monotonically)

Volatile state on leaders:
(Reinitialized after election)

nextIndex[]	for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)
matchIndex[]	for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit	leader's commitIndex

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC

Invoked by candidates to gather votes (§5.2).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)

Rules for Servers

All Servers:

- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine (§5.3)
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower (§5.1)

Followers (§5.2):

- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

Candidates (§5.2):

- On conversion to candidate, start election:
  - Increment currentTerm
  - Vote for self
  - Reset election timer
  - Send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If AppendEntries RPC received from new leader: convert to follower
- If election timeout elapses: start new election

Leaders:

- Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts (§5.2)
- If command received from client: append entry to local log, respond after entry applied to state machine (§5.3)
- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower (§5.3)
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§5.3)
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N (§5.3, §5.4).

Сжатое описание алгоритма Raft (параграфы соответствуют [1]).

Raft кластер содержит несколько серверов; 5 - это типичное значение, позволяющее системе справляться с 2 сбоями серверов. В любой момент времени каждый сервер может находиться в одном из 3 состояний: *последователь(follower)*, *кандидат*(*candidate)* или лидер(*leader)*. Во время нормального функционирования системы лидер только один, а все остальные сервера находятся в статусе последователя. Последователи пассивны, они не отправляют никаких запросов, они просто отвечают на запросы лидера. Лидер обрабатывает все запросы клиентов(если клиент подключается не к лидеру, то сервер, к которому подключились, должен направить этого клиента к лидеру). Третье состояние, candidate(кандидат), используется для выбора нового лидера.

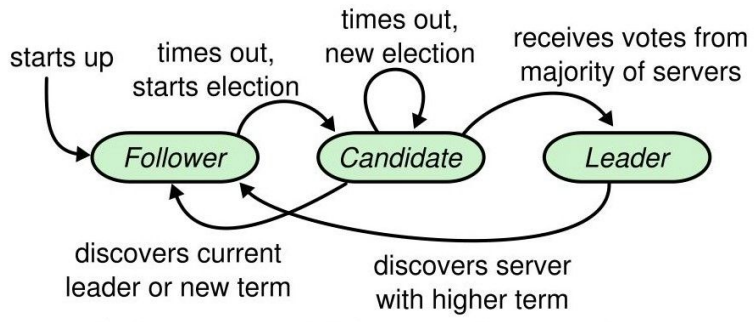


Рис.2. Переходы из одного состояния в другое

Raft делит время на термы(*term*) произвольной длины. Термы нумеруются целыми числами. Каждый терм начинается с *выборов,* в которых один или несколько серверов пытаются стать лидерами. Если кандидат выигрывает голосование, он становится лидером на остаток терма. В некоторых ситуациях выборы могут кончиться разделением голосов. В таком случае терм закончится без лидера и новый терм(с новыми выборами) скоро начнется. Raft гарантирует, что максимум один лидер может быть выбран в данном терме.

Различные серверы могут “увидеть” переходы между различными термами в разное время, а иногда и пропустить выборы или даже целый терм. Термы являются аналогом логических часов в Raft’е и они позволяют серверам обнаруживать устаревшую информацию, например, выбывшего лидера. Каждый сервер хранит *current term*(текущий терм), который монотонно возрастает в течение времени. Текущий терм изменяется всякий раз, когда сервера взаимодействуют друг с другом. Если текущий терм одного сервера меньше, чем другого, тогда сервер обновляет свой текущий терм к большему значению. Если кандидат или лидер обнаруживает, что его терм устарел, то он немедленно возвращается в состояние последователь. Если сервер получает запрос с устаревшим термом, он отклоняет запрос.

Серверы взаимодействуют путем RPC(remote procedure call). Базовая часть алгоритма Raft требует только двух типов RPC: RequestVote RPC и AppendEntry RPC. RequestVote RPC инициируются кандидатами во время выборов. AppendEntry RPC инициируются лидером, чтобы обеспечить т.н. биение(heartbeat).

Выборы лидера (Leader election)

Raft использует механизм биений для начала выборов лидера. Когда сервера начинают работу, они находятся в состоянии последователя. Сервер остается в этом состоянии до тех пор, пока получает действительные(valid) RPC от лидера или кандидата. Лидер отправляет периодические биения(AppendEntry, которые не имеют запросов) всем последователям для поддержки своей власти(authority). Если последователь не получает никакой связи в течение времени, называемого *election timeout,* он решает, что нет больше работающего лидера и инициирует голосование для выбора нового лидера.

Для начала голосования последователь прибавляет единицу к своему текущему терму и переходит в состояние кандидата. Далее он голосует за себя и рассылает RequestVote RPC параллельно каждому из серверов в кластере. Кандидат находится в своем состоянии до тех пор, пока не произойдет одна из 3 вещей: (а) сам кандидат выиграет голосование, (б) другой сервер не утвердит собственное лидерство, (в) текущий период времени кончится без лидера.

Кандидат выигрывает голосование, если он получает голоса от большинства(majority) серверов в кластере для текущего терма. Каждый сервер может проголосовать не больше, чем за одного сервера в каждом терме по принципу fisrt-come-first-served. Правило большинства гарантирует, что максимум один кандидат может выиграть выборы в текущем терме. Когда кандидат выигрывает голосование, н становится лидером. Он отправляет биения другим серверам, чтобы установить свою власть и воспрепятствовать новым выборам.

Ожидая голоса от своих соседей, кандидат может получить AppendEntry RPC от другого сервера, считающего себя лидером. Если текущий терм этого лидера не меньше терма кандидата, тогда кандидат должен признать лидера и вернуться в состояние последователя, иначе запрос должен быть проигнорирован.

Третий возможный исход - отсутствие лидера для текущего терма: если много последователей пытаются стать лидерами одновременно, может случиться разделение голосов. Когда такое происходит, каждый кандидат инициирует новое голосование после истечения таймаута: увеличивает текущий терм на единицу и рассылает новый раунд RequestVote RPC.

Raft использует рандомизированные таймауты для гарантии того, что разделение голосов не будет длиться вечно. Для предотвращения разделения

голосов таймауты выбираются из фиксированного интервала(150-300 мс). Таким образом, в большинстве случаев в данный момент таймер истечет только у одного из серверов.

Копирование журнала(Log replication)

После того, как лидер был выбран, он начинает обрабатывать запросы клиентов. Каждый запрос клиента содержит команду для исполнения RSM. Лидер добавляет команду к своему журналу, затем рассылает параллельно AppendEntry RPC другим серверам, чтобы скопировать запрос на каждый из серверов. Когда запрос был безопасно скопирован(как именно, будет описано ниже), лидер применяет этот запрос к своему RSM и возвращает результат исполнения клиенту. Если последователи работают медленно, дают сбои или сетевые пакеты теряются, лидер снова рассылает AppendEntry RPC. И так длится до тех пор, пока все последователи не сохранят себе все запросы.

Журнал организован следующим образом: каждая запись журнала хранит команду для выполнения на RSM и терм, во время которого эта команда была получена. Значения термов в записях журнала используются, чтобы обнаружить несоответствие между журналами.

Лидер решает когда безопасно применять команду к RSM. Такая запись журнала называется подтвержденной(committed). Raft гарантирует, что все подтвержденные записи журнала будут исполнены в одинаковом порядке всеми серверами кластера.

Лидер хранит индекс последней записи журнала и включает этот индекс во все AppendEntry RPC(включая биения), чтобы другие сервера могли обнаружить обновления. Когда последователь узнает, что данная запись журнала подтверждена, он применяет запрос из этой записи к RSM.

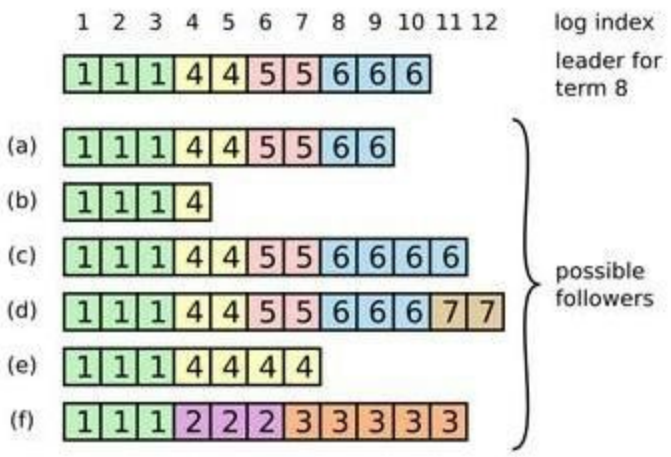
Raft удовлетворяет следующим свойствам:

1. Если 2 записи в различных журналах имеют одинаковый индекс и терм, то они хранят один и тот же запрос.
2. Если 2 записи в различных журналах хранят одинаковый индекс и терм, то все предшествующие им записи также совпадают для этих журналов.

Первое свойство следует из того, что лидер создает максимум одну запись с текущим индексом в текущем терме и того, что запись никогда не меняет своего расположения в журнале.

Второе свойство гарантируется несложной проверкой соответствия(consistency check), проводимой при получении AppendEntry RPC. При отправке AppendEntry RPC, лидер включает индекс и терм записи из своего журнала, которая предшествует новым записям. Если последователь не находит в своем журнале такой записи, то он не принимает новые записи.

Пока система нормально функционирует, журналы лидера и последователи остаются согласованными и проверка соответствия всегда выполняется успешно. Однако, выход из строя лидера, может сделать журнал несогласованными(лидер может не полностью скопировать все записи журнала на кластер).



Возможные состояния журналов последователей.

В Raft’е лидер сохраняет согласованность журналов, вынуждая последователей копировать его собственный порядок записей. То есть все не совпадающие с журналом лидера записи будут перезаписаны лидером.

Чтобы сохранять согласованность своего журнала с последователем, лидер ищет последнюю запись, до которой два журнала совпадают, удаляет все записи в журнале последователя после найденной записи и отправляет последователю свои записи, начиная с найденной точки. Все эти действия происходят в ответ на проверку согласованности, проводимую AppendEntry RPC. Лидер хранит для каждого последователя *next index* - индекс следующей записи, которая будет отправлена данному последователю. Когда лидер впервые избирается, он инициализует каждый next index индексом записи следующей за последней. Если журнал последователя не согласован с журналом лидера, проверка согласованности будет неуспешной. После отказа копировать запись, лидер уменьшает на единицу next index для этого последователя снова отправляет этот AppendEntry RPC. В конце концов, лидер достигнет точки, где журналы

совпадают. Все записи журнала последователя после этой точки будут удалены и новые записи(согласованные с лидером) будут добавлены. После приведения в согласованное состояние, журнал последователя будет оставаться таковым до конца терма.

Используя такой подход, лидеру не приходится прикладывать дополнительные усилия для восстановления согласованности журналов, это происходит автоматически при копировании новой записи или просто осуществлении биения.

Ограничение выборов

Raft использует процесс голосования, чтобы ограничить лидерство для тех последователей, чей журнал не содержит все подтвержденные записи. Кандидат должен получить большинство голосов, чтобы быть избранным лидером, что означает, что каждая подтвержденная запись должна должна присутствовать как минимум у одного сервера из этого большинства. Если журнал кандидата, как минимум, насколько же “свеж”, как и журнал каждого сервера из большинства, тогда кандидат содержит все подтвержденные записи. RequestVote RPC реализует это ограничение: RPC включает информацию о журнале кандидата и сервер не отдает свой голос, если его журнал “новее”.

Raft определяет какой журнал новее сравнивая индекс и терм последней записи этих журналов. Более новым считается тот журнал, у которого терм последней записи больше. Если термы одинаковы, то более длинный журнал считается новее.

Время и доступность

Одним из требований к алгоритму была независимость безопасной работы от времени: система не должна приводить к некорректным результатом только

потому что некоторые события происходят быстрее или медленнее, чем ожидалось.

Выборы лидера это один из аспектов, где время наиболее критично. Raft будет способен выбирать и поддерживать работу до тех пор, пока время в системе удовлетворяет следующим требованиям:

broadcastTime << electionTimeout << MTBF

broadcastTime - это среднее время, которое нужно серверу для отправки RPC кластеру и получения ответов.

electionTimeout - это вышеупомянутый рандомизированный таймаут

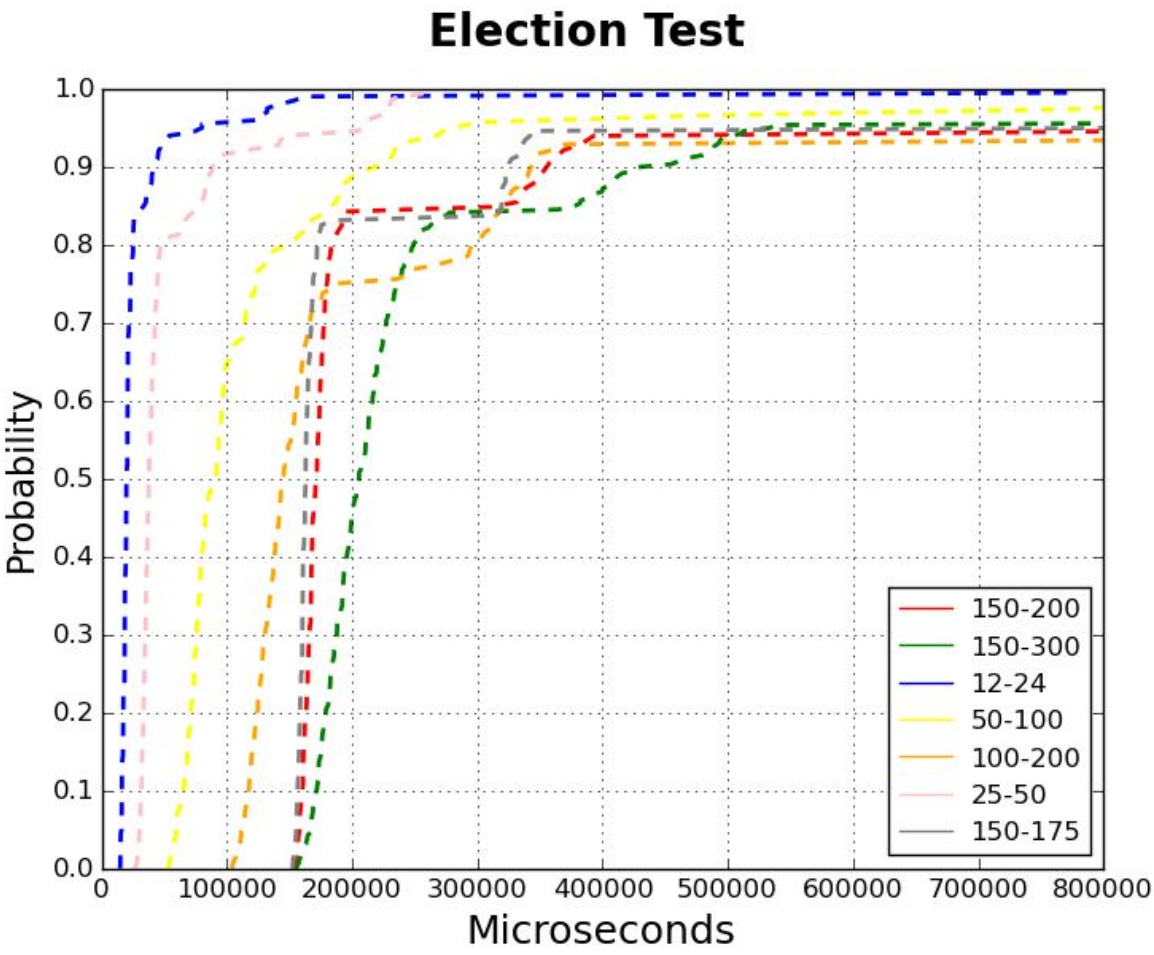
MTBF - mean time before failure - среднее время между сбоями сервера.

Производительность

В нашем исследовании резонно определить производительность как время выборов лидера кластером. Именно это время может критически сказаться на работе нашей системы.

Если вернуться к соотношению времен, то можно заметить, что выбор *electionTimeout* может производиться из широкого интервала значений. Как выбрать правильный промежуток?

Мы провели испытания для 10 различных интервалов значений. На графике видно, что полное отсутствие рандомизации в таймаутах приводит к тому, что выборы могут происходить очень долго(до 10 секунд). Но введение небольшой случайной составляющей меняет ситуацию в лучшую сторону. Также, *electionTimeout* не должен быть слишком маленьким, поскольку при наличии значительных задержек сети выборы могут никогда не кончиться. Результаты испытаний приведены на графике ниже:



Сетевые сбои

Также мы провели исследование, выявляющее влияние сетевых задержек на процесс нормального функционирования системы. При среднем времени выполнения запроса 0.7 мс, среднее время выполнения запроса при наличии константной задержки 20 мс, составляет 1.1 мс(задержка не учтена). Видно, что при наличии задержек, фактическое выполнение запроса происходит несколько дольше из-за наличия перевыборов в системе.

4 Заключение

В данной работе было реализована распределенная система(key-value хранилища) консенсуса на основе алгоритма Raft. Были протестированы несколько возможных интервалов для таймаутов и выбран наиболее оптимальный. Также было исследовано влияние перевыборов на время ответа на запрос клиента.

Литература

[1] “In search of understandable consensus algorithm”, Diego Ongaro

[2] “Распределенные алгоритмы”, Ж. Тель

[3] “Курс лекций по распределенным системам”, Коньков К.А.

[4] “Paxos made simpe”, Leslie Lamport