Федеральное государственное автономное образовательное учреждение
высшего образования
«Московский физико-технический институт
(национальный исследовательский университет)»
Физтех-школа Прикладной Математики и Информатики
Кафедра банковских информационных технологий

Направление подготовки: 01.03.02 Прикладная математика и информатика
Направленность (профиль) подготовки: Прикладная математика и компьютерные науки

Исследование и оптимизация способов выполнения операции AllReduce в сети с двухуровневой топологией

(бакалаврская работа)

Студент:
Климовицкий Роман Григорьевич

(подпись студента)

Научный руководитель:
Звонов Денис Владимирович,
магистр, старший преподаватель

(подпись научного руководителя)

Консультант:
Шворин Артем Борисович

(подпись консультанта)

Москва 2021

Аннотация

В данной работе рассматриваются различные способы выполнения коллективной операции AllReduce в сети, имеющей двухуровневую топологию. Целью работы являлось построение отказоустойчивого алгоритма выполнения данной операции, который бы устранял недостатки наивного подхода, основывающегося на коммуникации узлов по принципу «каждый с каждым». Основными задачами работы являлись: изучение подходов к реализации AllReduce, построение концепта системы и реализация ее прототипа на языке программирования, формальное доказательство свойств построенного алгоритма. В результате работы был разработан дизайн системы, который позволил снизить затраты ресурсов на сетевую коммуникацию между узлами, а также предоставить ряд гарантий на отказоустойчивость и корректность выполнения операции AllReduce. Основными этапами исследования являлись: рассмотрение подходов к решению задачи и выработка ключевой концепции промежуточных редукций, разработка механизма выбора узлов-редукторов, построение динамической маршрутизации на уровне узлов и реализация AllReduce в соответствии с выбранной концепцией. Результаты данной работы могут быть использованы для дальнейшей разработки геораспределенной системы выполнения коллективных операций, а также для ее тестирования и внедрения. В частности, написанный на языке Python прототип (приложение А) может стать отправной точкой для реализации готового продукта.

Содержание

Введение

Типичная инфраструктура компаний, предоставляющих георасределённые облачные сервисы в интернете (CDN, стримминг видео, публичные VPN, защита от DDoS), представлена в виде набора площадок, где каждая площадка представляет из себя набор вычислительных узлов в пределах одного дата-центра. Связность узлов внутри площадки очень хорошая (малые задержки, большая пропускная способность, высокое качество) и бесплатная, а связность между площадками осуществляется через открытый Интернет, либо арендованные каналы связи, и обладает намного меньшим качеством, которое постоянно варьируется (меняются задержки, процент потерь пакетов, временами связность может пропадать совсем). Важной особенностью является то, что в результате широкого применения в Интернете ECMP-маршрутизации (маршрут выбирается, исходя из значения хэша от адресов источника и получателя пакета), качество связи между узлом a_1 площадки A и узлом b_1 площадки B может существенно отличаться от качества связи между узлами a_2 и b_2 тех же площадок. Помимо этого в процессе работы могут добавляться новые узлы и площадки, старые узлы могут выходить из строя, каналы связи могут менять свои качественные характеристики и обрываться.

Иногда для решения различного рода задач, связанных с обработкой данных в описанной инфраструктуре, необходимо поддерживать некоторое когерентное состояние на всех узлах сети, которое достижимо путем применения ассоциативной, коммутативной операции редукции к данным, хранящимся на этих узлах. Например, может потребоваться, чтобы каждый узел знал суммарное количество трафика, обрабатываемое во всей сети.

В настоящее время не существует готового инструмента для решения задач подобного вида, который позволял бы работать с данными большого объема, не страдал от динамического изменения состояния системы и учитывал топологию сети. В данной работе мы изучим возможные подходы для решения поставленной проблемы и при необходимости построим собственный алгоритм для выполнения описанной операции редукции.

1 Постановка задачи

Итак, у нас имеется сеть со следующей топологией: есть несколько площадок, каждая из которых состоит из некоторого количества узлов. Считается, что передача данных между узлами разных площадок — «дорогая» операция, а внутри площадок — «почти бесплатная». Одна из задач данной распределенной системы — сбор некоторых счетчиков с узлов разных площадок и обмен ими для поддержания общего состояния по каждому типу счетчиков. Основной операцией, использующейся в решении этой задачи, является операция AllReduce: после ее завершения каждый узел сети должен узнать

результат некоторой редукции по значениям определенного счетчика со всех узлов сети.

Состояние узлов и каналов связи в системе носит динамический характер. В процессе работы могут добавляться новые узлы и площадки, старые узлы могут выходить из строя, каналы связи могут менять свои качественные характеристики и обрываться. Из описанных ограничений формулируется следующая модель отказов, с которой мы далее будем работать. В данной модели отказов возможны следующие отказы оборудования:

- Отказ узла (Node failure) — аварийное завершение работы одного узла системы. С некоторого момента времени узел останавливает свою работу без предупреждения, перестает отвечать на запросы и не отправляет никаких сообщений.
- Отказ канала связи (Link failure) — состояние системы, при котором невозможно доставить сообщение между двумя определенными узлами системы. Может носить временный характер: например, из-за поломки промежуточного сетевого оборудования сетевой инфраструктуре может понадобится некоторое время, чтобы перестроить маршрутизацию.
- Предполагается, что произвольных (византийских [1]) отказов в системе быть не может: каждый из узлов либо действует строго по заданному протоколу, либо не работает (см. Отказ узла).

Помимо всего прочего важным аспектом, который может влиять на множество возможных решений, является набор количественных характеристик системы, в пределах которых она может масштабироваться. Мы будем требовать от нашего решения, чтобы оно работало на стандартном оборудовании при следующих ограничениях:

- в сети находится порядка 100 площадок;
- каждая площадка содержит порядка 100 узлов;
- индивидуальные значения счетчиков каждого узла представляют из себя последовательный набор из порядка 10^5 целых чисел или чисел с плавающей точкой.

В простейшей реализации операции AllReduce общение узлов происходит по принципу «каждый с каждым», однако этот подход не учитывает высокую стоимость передачи данных между узлами разных групп, а также не соблюдает описанные требования по масштабируемости.

Таким образом, основной целью дальнейшей работы является создание алгоритма для выполнения опреации AllReduce, который бы исправлял недостатки наивного подхода и не страдал от изменчивого состояния системы.

Для достижения поставленной цели выпишем задачи, которые позволят нам к ней приблизиться. Для того, чтобы построить алгоритм, обладающий описанными свойствами, нам необходимо:

- изучить возможные способы реализации алгоритма на основе готовых проектов;
- построить концепт системы самостоятельно или на основе существующих решений;
- формально доказать, что предложенный вариант реализации корректен и обладает необходимыми свойствами;
- реализовать прототип системы на одном из языков программирования.

Далее опишем соответствующие шаги для выполнения поставленных задач.

2 Подходы к реализации

В данном разделе разберем основные подходы для реализации описанной операции AllReduce и выберем среди них наиболее подходящий.

2.1 All-Reduce из MPI

В известном интерфейсе для распределенных вычислений MPI (Message Passing Interface) есть одноименная операция All-Reduce [2, с. 238]. Она является аналогом операции Reduce из того же интерфейса с тем отличием, что результат редукции передается всем процессам в группе.

Однако основным недостатком интерфейса MPI является его неустойчивость к отказам процессов. В случае их возникновения текущее выполнение аварийно завершается, и лучшее, что стандартная реализация MPI может предоставить, – это сообщение о возникшей ошибке и инструменты для ее обработки [2, с. 26].

От нашего решения мы будем требовать продолжение работы даже в случае возникновения отказов отдельных узлов. Поэтому мы не будем рассматривать MPI как готовое решение в дальнейшей работе.

2.2 Согласованное в конечном счете хранилище счетчиков

Другой вариант реализации, который можно было бы рассматривать, заключается в поддержании согласованной базы счетчиков. Для этого узлам пришлось бы регулярно обмениваться данными индивидуальных счетчиков. В этом случае при необходимости каждый узел сможет проводить редукцию счетчиков, которые он хранит. Этот подход делится на два приципиально отличающихся случая: 1) каждый узел поддерживает хранилище счетчиков всех узлов системы; 2) узел поддерживает базу, состоящию из несобственного подмножества счетчиков. Каждый из случаев также обладает своими недостатками.

В первом случае база счетчиков со всей системы будет занимать в памяти узла достаточно много ресурсов. Если считать, что значение каждого счетчика – это 64-битное целое число, то для поддержания такого хранилища на каждом узле понадобится

около 7–8 ГБ оперативной памяти: всего в сети около 10^4 узлов, каждый из который обладает набором из 10^5 счетчиков. Итого, все данные займут около $10^4 \cdot 10^5 \cdot 8 \,/\, 1024^3 \approx 7,5$ ГБ. Помимо этого обмен значениями индивидуальных счетчиков между узлами различных площадок не учитывает высокую стоимость передачи данных между ними, поэтому даже в случае поддержания хранилища лишь на подмножестве узлов системы такой метод оказывается слишком требовательным к ресурсам сети и отдельных узлов.

Второй случай в свою очередь не может считаться готовым решением: если разбивать все множество счетчиков на подмножества, необходимо дополнительно продумывать коммуникацию между узлами, обладающими отличными друг от друга данными.

В действительности хранить «чужие» индивидуальные счетчики на узлах из-за их быстрого устаревания нецелесообразно. Гораздо разумнее при получении индивидуальных счетчиков другого узла подсчитывать промежуточную редукцию — благодаря такому подходу можно сэкономить память на узлах, а также вместо значений отдельных счетчиков передавать другим узлам сразу результаты частичных редукций. Использование этой идеи обсудим в следующих двух подразделах 2.3 и 2.4.

2.3 Наивная реализация

Как мы уже упоминали, для реализации операции AllReduce можно применить наивный подход, который состоит в попарном общении узлов друг с другом. При использовании этого подхода каждый узел накапливает результат частичной редукции по мере того, как он получает новые значения индивидуальных счетчиков других узлов. Время от времени накопленное значение сохраняется в качестве актуального результата операции AllReduce, а затем очищается для подсчета нового.

Несмотря на то, что в данном походе не требуется хранить каждый экземпляр индивидуальных счетчиков отдельно, он все еще страдает от большого объема пересылаемых данных между узлами различных площадок, что опять же не учитывает двухуровневую топологию сети и, более того, в наших ограничениях может оказывать экстремально большую нагрузку на сетевую инфраструктуру и сетевые интерфейсы отдельных узлов: узлу необходимо принимать и обрабатывать до $7,5$ ГБ данных для проведения одной итерации редукции (см. подраздел 2.2).

2.4 Частичная редукция в пределах площадок

Прием с подсчетом частичных редукций можно также использовать для того, чтобы сократить сетевую коммуникацию между узлами разных площадок. Для этого в пределах каждой площадки один или несколько узлов могут накапливать частичную редукцию, состоящую из значений счетчиков узлов данной группы, а затем передавать в другие площадки полученный результат. Поскольку операция редукции ассоциативна, итоговый результат операции AllReduce можно «собирать» из таких частичных редукций вместо добавления каждого индивидуального счетчика по отдельности. Коль скоро

вся коммуникация между площадками, относящаяся к непосредственной реализации операции AllReduce, в данном подходе будет состоять лишь из передачи результатов промежуточных редукций, он будет заведомо эффективнее наивного подхода. Поэтому мы возьмем описанную идею за основу, на которой будет строиться дальнейшее наше решение.

3 Математическая постановка задачи

Перед тем, как приступать к описанию алгоритма, введем формальные обозначения и сформулируем задачу в новых терминах. Это необходимо для задания четких требований к реализации. Зафиксированные обозначения нам также помогут ясно выражать утверждения при доказательстве тех или иных свойств построенной системы.

3.1 Двухуровневая топология сети

Определение 3.1. Пусть S – *множество всех узлов сети. Узлом сети* будем называть любой элемент множества S. За $N := |S|$ обозначим количество узлов в сети.

Определение 3.2. Изначально все узлы разбиты на непересекающиеся множества $S_1, S_2, \ldots, S_n$, называемые *площадками.* $\mathcal{S} := \{S_1, \ldots, S_n\}$ — *множество всех площадок.*

Таким образом, выполнено: $\bigsqcup_{k=1}^{n} S_k = S$. Для простоты будем считать, что все площадки не пустые: $\forall k\ S_k \neq \varnothing$.

3.2 Метрика сети

В дальнейшем мы будем работать с сущностями, которые со временем могут изменяться. Поэтому зафиксируем понятие временно́й шкалы, с которой мы далее будем работать, чтобы отразить динамическую природу решаемой задачи:

Определение 3.3. $T \in \{\mathbb{R}_+,\ \mathbb{Z}_+\}$ — *шкала времени* (считаем, что время либо дискретно, либо непрерывно).

Определение 3.4. Каждый из узлов $a \in S$ имеет доступ к некоторой *динамической информации* $\mathrm{Info}_t(a, b) \in I_d$ о качестве непосредственного соединения до всех других узлов $b \in S \setminus \{a\}$, т.е. $\{\mathrm{Info}_t\}_{t \in T}$ — семейство частично определенных, индексированных параметром времени функций вида $\mathrm{Info}_t \colon S \times S \to I_d$.

Определение 3.5. По динамической информации информации считается *метрика качества соединения* $m \colon I_d \to \mathbb{R}_{>0}$. При этом можем определить *метрику непосредственного соединения между узлами* a *и* b *в момент времени* t $m_t \colon S \times S \to \mathbb{R}_+$ так, что

$$m_t(a, b) := \begin{cases} m(\mathrm{Info}_t(a, b)) & ,\ a \neq b \\ 0 & ,\ a = b \end{cases}$$

.

Замечание 3.1 (Физический смысл метрики). В понятие динамической информации может входить целый ряд свойств канала связи: как статических (стоимость передачи данных по данному каналу, его приоритет), так и динамических (RTT [3], потеря пакетов [4], явный флаг о работоспособности канала). «Хорошая» функция метрики качества должна учитывать природу этих данных и отражать реальное качество сетевого соединения: чем ниже метрика, тем выше характеристики канала в совокупности. В том числе, если узел b неработоспособен в момент времени t, то до него невозможно доставить данные, а значит и $\forall a \in S \setminus \{b\}$ значения $m_t(b, a)$ и $m_t(a, b)$ должны быть бесконечно большими по сравнению с метрикой соединения между любыми другими рабочими узлами. Известно [5], что в сетевой инфраструктуре нельзя гарантировать доставку данных за ограниченное время. Однако хотелось бы иметь некоторую уверенность в том, что при достаточно низкой метрике вероятность неуспешной передачи данных стремится к нулю — «хорошая» функция метрики должна учитывать и эту особенность сети. Тем не менее, в данной работе мы абстрагируемся от явного вида данной функции, чтобы сосредоточиться на основной цели по оптимизации выполнения операции AllReduce.

Определение метрики соединения 3.5 вполне естественно: нам ничего не стоит передать данные самому себе, ведь они у нас уже есть, но при этом мы не можем отправить данные кому-то другому совсем бесплатно.

Замечание 3.2. Считается, что почти всегда выполнено следующее:

$$\forall\ i,\ j,\ k \in \{1,\ \ldots,\ n\}\ (i \neq j \implies \forall\ a \in S_i,\ b \in S_j,\ c \in S_k,\ d \in S_k\ m_t(a, b) \gg m_t(c, d))$$

— метрика соединения между узлами разных площадок существенно выше метрики соединения между узлами одной площадки.

3.3 Операция AllReduce

Считается, что у каждого узла системы есть изменяющиеся со временем счетчики. Глобальная задача заключается в применении редуцирующей коммутативной ассоциативной операции к значениям счетчиков всех узлов и распространении результата на все узлы системы.

Определение 3.6. Описанная глобальная операция называется *AllReduce*.

Узлы способны выполнять частичную редукцию полученных от других узлов счетчиков и отправлять полученный промежуточный результат другим узлам. Они также способны применять операцию к полученным частичным результатам.

Определение 3.7. Введем обозначения:

- C — универсум значений счетчиков и результатов их редукций, которыми владеют узлы системы;
- $c_t \colon S \to C$ — функция, возвращающая значение счетчиков определенного узла в момент времени t;
- $|\cdot| \colon C \to \mathbb{N}$ — функция, возвращающая размер элемента множества счетчиков в единицах памяти, аналог объема пересылаемых данных;
- $+ \colon C \times C \to C$ — операция редукции;
- $\sum_{i=k}^{k} c_i := c_k,\ \sum_{i=k}^{m} c_i := \sum_{i=k}^{m-1} c_i + c_m,\ k \neq m,\ \forall i \in \mathbb{N}\ c_i \in C$ — семейство операторов различной валентности для краткой записи редукции, примененной несколько раз;
- $0 \in C$ — нейтральный элемент по операции редукции: $\forall c \in C\ c + 0 = 0 + c = c$.

Мы используем знаки «плюса» и суммирования для наглядности — операция сложения, как правило, коммутативная и ассоциативная, а сумма счетчиков — один из возможных результатов редукции, который необходимо получить в конце выполнения AllReduce. Также для удобства будем использовать инфиксную запись.

Поскольку коммуникация между узлами может занимать любой положительный отрезок времени, отправляемые данные могут произвольным образом терять свою актуальность до ее завершения. Значит любое доступное нам решение не может гарантировать того, что результат работы алгоритма будет всегда равен результату редукции, примененной к актуальным значениям счетчиков. Тем не менее, мы можем требовать более слабых гарантий от решения — для этого определим, что мы будем считать корректным выполнением операции AllReduce.

Определение 3.8. Пусть узел $a \in S$ в момент времени $t \in T$ локально завершил выполнение операции AllReduce и в качестве результата получил значение c_{AR}. Для каждого узла $a_i \in S$ рассмотрим множество $C_i = \{c_t(a_i) \colon t \in [t - \Delta, t]\} \subset C$ из значений индивидуальных счетчиков данного узла, которыми он владел в разные моменты на временно́м отрезке $[t - \Delta, t]$. В частности, если узел a_k в данный промежуток времени не работал или находился в недоступной части сети, будем считать $C_k = \{0\}$. Результат операции AllReduce c_{AR} будем называть *корректным с допустимым временны́м отклонением* $\Delta > 0$, если $\exists \{c_i\}_{i=1}^{N} \colon \forall i \leq N\ c_i \in C_i \wedge c_{AR} = \sum_{i=1}^{N} c_i$.

Замечание 3.3. В частности, если с некоторого момента времени значения счетчиков на узлах перестали меняться, корректная в наших терминах реализация операции AllReduce за ограниченное время должна привести к тому, что результаты редукции на узлах будут согласованы в пределах каждой компоненты сильной связности сети.

Перейдем к формальной постановке задачи по снижению суммарной стоимости выполнения операции AllReduce.

Замечание 3.4. Считается, что верно следующее: $\forall\ c_1, c_2 \in C\ |c_1 + c_2| \leq |c_1| + |c_2|$, т.е. промежуточная редукция не увеличивает размер передаваемых данных.

Определение 3.9. Пусть функция $cost\colon \mathbb{R}_+ \times \mathbb{Z}_+ \to \mathbb{R}_+$ обладает следующими свойствами:

1. $\forall\ l \in \mathbb{Z}_+\ cost(0, l) = 0$;
2. $\forall\ m \in \mathbb{R}_+\ cost(m, 0) = 0$;
3. $\forall\ l \in \mathbb{Z}_+\ \forall\ m_1^{(1)}, m_2^{(1)}, \ldots, m_p^{(1)}, m_1^{(2)},\ m_2^{(2)}, \ldots, m_q^{(2)} \in \mathbb{R}_+$

$$\Big(\sum_{i=1}^{p} m_i^{(1)} < \sum_{j=1}^{q} m_j^{(2)} \implies \sum_{i=1}^{p} cost(m_i^{(1)}, l) < \sum_{j=1}^{q} cost(m_j^{(2)}, l)\Big);$$

4. монотонность по второму аргументу: $\forall\, m \in \mathbb{R}_+\ \forall\, l_1, l_2 \in \mathbb{Z}_+\ (l_1 < l_2 \implies cost(m, l_1) < cost(m, l_2))$.

Тогда *cost* называется *функцией стоимости передачи данных.*

Определение 3.10. Пусть *cost* — некоторая функция стоимости передачи данных, $t \in T,\ a, b \in S,\ l \in \mathbb{N}$. Величину $cost_t(a, b, l) := cost(m_t(a, b), l)$ назовем *стоимостью передачи данных объемом l по каналу связи (a, b) относительно cost в момент времени t.*

Определение 3.11. Пусть *cost* — некоторая функция стоимости передачи данных, $t \in T,\ a, b \in S,\ c \in C$. Величину $cost_t(a, b, c) := cost_t(a, b, |c|)$ назовем *стоимостью передачи счетчиков с относительно cost из узла a в узел b в момент времени t.*

Ставится задача снижения суммарной стоимости передачи данных, необходимой для корректного выполнения операции AllReduce с ограниченным допустимым временны́м отклонением. Из замечаний 3.2 и 3.4 возникает предложение проводить частичную редукцию счетчиков одной площадки на небольшом числе узлов данной площадки и передавать полученный результат в другие площадки для подсчета окончательного результата.

Для реализации предложенного механизма каждый из узлов должен быть способен выполнять следующие задачи:

1. Определять узел своей площадки, на который он передаст данные своих счетчиков для последующей редукции и отправки;
2. Отправлять данные на узлы других площадок таким образом, чтобы суммарная стоимость их передачи была минимальна;
3. Накапливать результаты промежуточных редукций: как внутри площадки, так и промежуточные результаты, пришедшие с других площадок.

Первый пункт из списка решим посредством введния узлов-редукторов, которые будут определяться аналогично выбору DR и BDR из протокола OSPF [6, с. 75] — на них же мы в будущем сможем хранить произвольные данные, отражающие состояние площадки. В разделе 4 сформулируем алгоритм, по которому данный выбор происходит. Реализацию второго пункта обеспечим введением собственного протокола маршрутизации на уровне узлов сети, который опишем в следующем разделе 5. Третий пункт, являющийся завершающим в подсчете результата AllReduce, раскроем в соответствующем разделе 6.

4 Узлы-редукторы

Как мы отметили ранее, для оптимизации выполнения AllReduce разумным решением может стать выполнение промежуточных редукций в пределах площадок. В действительности при условии того, что узел способен справиться с соответствующей нагрузкой, с точки зрения стоимости передачи данных по сети выгоднее всего проводить редукцию на единственном узле в пределах площадки, поскольку при наличии нескольких узлов-редукторов и отсутствии дополнительной коммуникации между ними их результаты промежуточных редукций будут рассылаться на другие площадки по отдельности. Таким образом, основной целью данного раздела будет являться построение алгоритма, позволяющего определить в пределах каждой площадки единственный узел для централизованного выполнения операции редукции на нем.

4.1 Существующие решения

В мире существуют несколько подходов к решению задачи выбора лидера, которые отличаются предоставляемыми гарантиями. Мы рассмотрим два из них, которые имеют широкое применение в современных распределенных системах: выбор лидера в RAFT [7] и выбор DR и BDR в OSPF [6, с. 75].

Алгоритм выбора лидера RAFT гарантирует, что в пределах работающего кластера ни в один момент времени не будет присутствовать более одного лидера. Зачастую данное свойство очень полезно при проектировании систем, в которых важна консистентность. Однако это также накладывает ограничения на доступность системы при разделении сети: меньшая часть кластера не может выбирать нового лидера из-за того, что для перехода в состояние лидерства узлу необходимо получить подтверждение хотя бы у половины узлов сети. С этим же связана другая его особенность: в процессе работы могут возникать «несостоявшиеся» выборы, когда ни один из узлов не набрал необходимый кворум. Чтобы с этим бороться, в RAFT используется прием случайных временны́х интервалов между посылкой сообщений. Однако с увеличением числа узлов, участвующих в выборах, стоит ожидать, что многие узлы будут инициировать новые выборы примерно в одно время, что приведет к частым перезапускам

данной процедуры. Увеличение числа узлов приводит также к сокращению средней продолжительности эпохи, в которой определен лидер, что было описано и доказано в работе [8]. Тем не менее, все эти ограничения оправданы — на основе данного алгоритма в RAFT строится распределенный журнал, который требует высокой степени согласованности.

Другой алгоритм выбора единого узла, который мы рассмотрим, описывается в формате выбора Designated Router (DR) и Backup Designated Router (BDR) в протоколе динамической маршрутизации OSPF [6, с. 75]. Он обладает более слабыми гарантиями по сравнению с предыдущим решением и допускает наличие временны́х промежутков, когда описанный выбор не согласован. Данный алгоритм опирается на предопределенную информацию о том, что при прочих равных выбранный узел определяется однозначно в соответствии со статической информацией. Это позволяет узлам эффективно разрешать конфликты, а также «предсказывать» исход выборов неявно и заранее. Таким образом, данный алгоритм позволяет всем узлам действовать асинхронно, что повышает доступность системы в ущерб согласованности, а также позволяет зайдействовать в алгоритме относительно большое количество узлов. Несмотря на отсутствие гарантий по согласованности, выбор резервного узла не делает отказ текущего «лидера» критичным — в данном случае резервный узел перенимает его ответственность на себя, что в том числе позволяет поддерживать некоторое состояние кластера, например, текущий результат редукции.

С учетом всего вышесказанного было принято решение использовать алгоритм выбора DR и BDR в OSPF в качестве основной идеи для выбора наших узлов-редукторов, поскольку он больше всего подходит под наши задачи: мы не требуем жесткой согласованности в выборе единых узлов и готовы мириться с короткими промежутками времени, когда их может быть несколько. Более того, использование данного решения позволит работать всем узлам по одному и тому же механизму, не заботясь об общем количестве узлов. В случае консенсусного выбора лидера пришлось бы ограничивать набор узлов, на котором он бы выполнялся.

4.2 Алгоритм выбора узла-редуктора

Опишем непосредственно алгоритм, по которому каждый из узлов будет определять текущий узел-редуктор.

Здесь и далее в работе будем описывать различные сообщения, которыми узлы будут обмениваться, в формате protobuf-сообщений из фреймворка Google Protocol Buffers [9], который позволяет сериализовать и десериализовать данные, предназначенные для отправки по сети.

Заведем таймер, по которому каждый узел один раз в промежуток времени Δ_{hb} будет отправлять всем узлам собственной площадки heartbeat-сообщения вида, представленного в листинге 1.

```
message HeartbeatMessage {
    NeighbourState state = 1;
    uint64 start_ts = 2;
}
```

Листинг 1 — Описание heartbeat-сообщения в формате protobuf-сообщения.

В данном сообщении `NeighbourState` — это protobuf-перечисление из листинга 2.

```
enum NeighbourState {
  OTHER = 0;
  REDUCER = 1;
  BACKUP_REDUCER = 2;
}
```

Листинг 2 — Protobuf-перечисление из возможных состояний узла.

Мы хотим добиться того, чтобы каждый из узлов площадки в каждый момент времени, возможно неявно, находился в одном из состояний `REDUCER`, `BACKUP_REDUCER` и `OTHER` и данную информацию вместе со временем начала работы указывал в своих heartbeat-сообщениях. Для этого на каждом из узлов заведем переменные `reducer` и `backup_reducer`, которые будут содержать идентификаторы узлов, которых данный узел считает редуктором и резервным редуктором соответственно, а также последние полученные от них времена начала работы. Таким образом, эти поля будут иметь тип `NeighbourRevision` (листинг 3).

```
class NeighbourRevision:
    def __init__(self, node_id: int, start_ts: int):
        self.id = node_id
        self.start_ts = start_ts
```

Листинг 3 — Определение класса `NeighbourRevision`.

Путем добавления к идентификатору узла отметки времени начала его работы в виде количества секунд, прошедших с начала эпохи [10], мы можем перейти к новой модели отказов, в которой все отказавшие узлы никогда не перезапускаются вновь, а вместо них запускаются новые узлы с новой отметкой времени. Действительно, таким образом, пары «идентификатор-временна́я отметка» для каждого запуска узла будут уникальными.

Изначально полям `reducer` и `backup_reducer` присвоим значения `NeighbourRevision(-1, -1)`, что будет означать, что узел не обладает информацией о том, кто в данный момент является редуктором и резервным редуктором. Значения данных полей определяют состояние, в котором находится данный узел, которое будет добавлено

в его heartbeat-сообщение. Если идентификатор самого узла равен значению идентификатора одного из данных полей, его состояние определяется соответствующим образом, описанным в листинге 4.

```
def __get_state(self) -> reduction_pb2.NeighbourState.V:
    if self.__reducer.id == self.__id:
        return REDUCER
    elif self.__backup_reducer.id == self.__id:
        return BACKUP_REDUCER
    else:
        return OTHER
```

Листинг 4 — Определение состояния узла по значениям его полей `reducer` и `backup_reducer`.

При этом будем стремиться к тому, чтобы как можно чаще выполнялись следующие условия:

- С точки зрения каждого узла ни один из узлов не должен считаться редуктором и резервным редуктором одновременно.
- При условии отсутствия разделения сети [11, с. 2] внутри площадки в один момент времени на ней не должно быть больше одного узла, которого хотя бы один работающий узел считал бы редуктором. Аналогично для резервных редукторов.
- Если на площадке есть хотя бы один рабочий узел, редуктор должен быть выбран на каждом из узлов. Если на площадке есть хотя бы два рабочих узла, на каждом из них должны быть выбраны и редуктор, и резервный редуктор.
- В случае отказа редуктора новым редуктором должен стать текущий резервный редуктор. Соответственно, резервный редуктор должен переизбраться.

Для выполнения условия на единственность редуктора и резервного редуктора заведем правило, по которому будут разрешаться конфликты: если узлу a приходит heartbeat-сообщение от узла b со значением `state`, равным `REDUCER`, а идентификатор узла b больше значения поля `reducer.id` у узла a, то данному полю присваивается новое значение, равное идентификатору узла b. Если при этом значения `reducer` и `backup_reducer` стали равны, значение `backup_reducer` инвалидируется. Аналогично для резервного редуктора с тем лишь отличием, что мы ничего не делаем, если heartbeat-сообщение пришло от узла, которого мы в данный момент считаем редуктором (такая ситуация может возникнуть, когда резервный редуктор еще не успел понять, что сам должен заменить редуктора, но должен отправить очередное heartbeat-сообщение). Нам также важно обновить временну́ю отметку текущих узлов на актуальную, если она изменилась. Если же пришедшее heartbeat-сообщение имеет поле

`heartbeat.state` равное `OTHER`, и оно было отправлено текущим редуктором или резервным редуктором с изменением временно́й отметки, значит данный узел перезапустился и мы обладаем неактуальной информацией, и ее также необходимо инвалидировать. Стоит отметить, что здесь важную роль играет временна́я отметка узла: с ее помощью мы отличаем ситуацию, когда назначенный редуктор или резервный редуктор отказал от той, когда он сам еще не узнал, в каком состоянии он находится.

Таким образом, при возникновении конфликтов больший приоритет в выборе редуктора или резервного редуктора имеет узел с бо́льшим идентификатором. При условии надежной сети (отсутствие потерь, минимальные задержки) такое правило сможет нам гарантировать, что с момента возникновения несогласованности в виде разных редукторов (резервных редукторов) на одной площадке при отсутствии дальнейших отказов узлов пройдет не больше Δ_{hb} времени до наступления момента, когда выбор редуктора (резервного редуктора) станет согласован. Помимо всего прочего, введенное правило в дальнейшем будет влиять на алгоритм выбора редуктора и резервного редуктора при обнаружении отказов.

Другая особенность, которую стоит учитывать при получении heartbeat-сообщения от текущего редуктора или резервного редуктора, заключается в том, что данное сообщение может содержать информацию, не согласующуюся с текущими значениями полей `reducer` и `backup_reducer`. Например, такая ситуация может возникнуть в случае, когда редуктор отказал, текущий узел заметил это и в качестве редуктора установил значение текущего резервного редуктора b, но сам узел b об этом еще не узнал и, продолжая считать себя резервным редуктором, отправил текущему узлу heartbeat-сообщение. Для того, чтобы не инвалидировать значение редуктора в данном случае, заведем поля `redcuer_till_expiration` и `backup_till_expiration`, которые при каждом обновлении редуктора или резервного редуктора будут устанавливаться равными n_e (можно считать $n_e = 1$). В дальнейшем будем использовать эти поля для отличия штатных ситуаций от неполадок, в процессе чего их значения могут уменьшаться. При получении heartbeat-сообщения с ожидаемым состоянием будем присваивать соответствующему полю исходное значение n_e.

Итоговый код обработки heartbeat-сообщения представлен в листинге 5.

В реализации описанного метода используются вспомогательные функции для различных проверок, код которых представлен в листинге 6.

Для определения отказов узлов заведем еще один таймер с временны́м интервалом Δ_{dead}. Несмотря на то, что по замечанию 3.2 качество сети в пределах площадки достаточно хорошее, даже между рабочими узлами единичные пакеты могут терятся или соединение может временно иметь высокую задержку. Поэтому, чтобы свести к минимуму число ситуаций, когда рабочий узел считается отказавшим из-за проблем сети, Δ_{dead} должен превосходить Δ_{hb} в несколько раз. Для определенности возьмем $\Delta_{dead} = 3\Delta_{hb}$. В промежутках между срабатываниями таймера по детекции отказов будем сохранять

```
def handle_heartbeat(self,
                     heartbeat: reduction_pb2.HeartbeatMessage,
                     source_node: int) -> None:
    self.__received_heartbeats[source_node] = heartbeat
    source_revision = NeighbourRevision(source_node, heartbeat.start_ts)
    if heartbeat.state == REDUCER:
        if self.__reducer.id < source_revision.id or \
                self.__is_new_revision(self.__reducer, source_revision):
            self.__set_reducer(source_revision)
            if self.__reducer.id == self.__backup_reducer.id:
                self.__set_backup(NeighbourRevision(-1, -1))
        elif self.__is_same(self.__reducer, source_revision):
            self.__reducer_till_expiration = self.__STATE_EXPIRATION
    elif heartbeat.state == BACKUP_REDUCER:
        if self.__is_same(self.__reducer, source_revision):
            return
        if self.__backup_reducer.id < source_revision.id or \
                self.__is_new_revision(self.__backup_reducer, source_revision):
            self.__set_backup(source_revision)
        elif self.__is_same(self.__backup_reducer, source_revision):
            self.__backup_till_expiration = self.__STATE_EXPIRATION
    elif heartbeat.state == OTHER:
        if self.__is_new_revision(self.__reducer, source_revision):
            self.__set_reducer(NeighbourRevision(-1, -1))
        elif self.__is_new_revision(self.__backup_reducer, source_revision):
            self.__set_backup(NeighbourRevision(-1, -1))
```

Листинг 5 — Обработка приходящих heartbeat-сообщений.

```
@staticmethod
def __is_new_revision(prev: NeighbourRevision, cur: NeighbourRevision) -> bool:
    return prev.id == cur.id and prev.start_ts < cur.start_ts

@staticmethod
def __is_same(node_1: NeighbourRevision, node_2: NeighbourRevision) -> bool:
    return node_1.id == node_2.id and node_1.start_ts == node_2.start_ts
```

Листинг 6 — Вспомогательные функции для проверки актуальности имеющихся данных о редукторе и резервном редукторе.

последние heartbeat-сообщения от каждого из узлов в словарь `received_heartbeats`, перед запуском нового таймера данный словарь будем очищать. Отказавшими будем считать те узлы, от которых за очередной промежуток времени Δ_{dead} мы не получили ни одного heartbeat-сообщения. Если среди таких узлов оказался хотя бы один из узлов с идентификаторами `reducer` и `backup_reducer`, либо же последние heartbeat-сообщения говорят о том, что состояние кого-то из них давно (на протяжении более чем $n_e \cdot \Delta_{dead}$ времени) не согласуется текущими значениями данных полей, то запускает-

ся алгоритм перевыборов (метод `reelect`). Последняя проверка будет осуществляться следующим образом: если при очередном срабатывании таймера Δ_{dead} состояние последнего полученного heartbeat-сообщения от узла, которого мы считаем редуктором, не равно `REDUCER`, уменьшаем значение поля `reducer_till_expiration` на 1. Если значение `reducer_till_expiration` стало отрицательным, значит состояние узла давно не согласуется с ожиданиями и нужно запустить алгоритм перевыборов. Аналогично для резервного редуктора.

Итого, каждый промежуток времени в Δ_{dead} мы будем запускать следующую процедуру:

```
def __check_availability(self) -> None:
    if not self.__is_node_alive(self.__reducer) or \
            not self.__is_node_alive(self.__backup_reducer):
        self.__reelect()
    else:
        if not self.__is_in_state(self.__reducer.id, REDUCER):
            self.__reducer_till_expiration -= 1
        if not self.__is_in_state(self.__backup_reducer.id, BACKUP_REDUCER):
            self.__backup_till_expiration -= 1
        if self.__reducer_till_expiration < 0 or \
                self.__backup_till_expiration < 0:
            self.__reelect()

    self.__received_heartbeats.clear()

def __is_node_alive(self, node: NeighbourRevision) -> bool:
    return node.id in self.__received_heartbeats and \
           node.start_ts == self.__received_heartbeats[node.id].start_ts

def __is_in_state(self,
                  node_id: int,
                  required_state: reduction_pb2.NeighbourState.V) -> bool:
    if node_id not in self.__received_heartbeats:
        return False
    return self.__received_heartbeats[node_id].state == required_state
```

Листинг 7 — Проверка работоспособности редуктора и резервного редуктора по набору полученных heartbeat-сообщений.

Сам алгоритм перевыборов (метод `reelect`) опишем последовательно по шагам:

1. Если среди отказавших или несогласованных узлов есть `reducer`, ставим на его место текущий `backup_reducer` и инвалидируем `backup_reducer` (листинг 8).
2. Определяем множество доступных узлов A как множество ключей словаря `received_heartbeats` (листинг 9).
3. Если доступных узлов помимо текущего нет (считаем, что сообщение до самого

```
if not self.__is_node_alive(self.__reducer) or \
        self.__reducer_till_expiration < 0:
    self.__set_reducer(self.__backup_reducer)
    self.__set_backup(NeighbourRevision(-1, -1))
```

Листинг 8 — Проверка работоспособности текущего редуктора.

```
available_nodes = self.__received_heartbeats.keys()
```

Листинг 9 — Определение множества доступных узлов.

себя узел всегда отправляет успешно), то назначаем редуктором себя, а значение `backup_reducer` инвалидируем. В этом случае алгоритм сразу завершается (листинг 10).

```
if len(available_nodes) <= 1:
    self.__set_reducer(NeighbourRevision(self.__id, self.__start_ts))
    self.__set_backup(NeighbourRevision(-1, -1))
    return
```

Листинг 10 — Обработка случая с единственным рабочим узлом на площадке.

4. Если мы получили хотя бы два heartbeat-сообщения, приступаем к выбору резервного редуктора. Будем выбирать его среди тех узлов, которые не считают себя редукторами и чей идентификатор не равен `reducer.id`: $PBR = A \setminus \{$ `id : received_heartbeats[id].state != REDUCER` $\} \setminus \{$ `reducer.id : receivved_heartbeats[reducer.id].start_ts == reducer.start_ts` $\}$.

   (a) Если среди PBR есть узлы, которые сами себя считают резервными редукторами, в качестве реального редуктора выберем из них узел с наибольшим идентификатором.

   (b) Иначе выберем узел с наибольшим идентификатором среди всех PBR.

   Коль скоро выбор редуктора будет происходить по аналогичной процедуре, заведем отдельную функцию для этого, которую опишем в листинге 11.

   Итоговый код определения резервного редуктора представлен в листинге 12.

5. Далее, если за последний период мы не получили heartbeat-сообщение от узла, соответствующего значению `reducer`, перейдем к выбору редуктора. Чтобы избежать совпадения редуктора и резервного редуктора, удалим из рассмотрения только что выбранный резервный редуктор: $PR = A \setminus \{$`backup_reducer.id`$\}$.

   (a) Если среди доступных узлов есть считающие себя редукторами, выберем среди них узел с наибольшим идентификатором.

   (b) Иначе выберем узел с наибольшим идентификатором среди всех PR.

```
def __elect_from(self,
                 possible: tp.Set[int],
                 self_proclaimed: tp.Set[int]) -> NeighbourRevision:
    new_elect = -1
    if len(self_proclaimed) != 0:
        new_elect = max(self_proclaimed)
    elif len(possible) != 0:
        new_elect = max(possible)

    if new_elect != -1:
        return NeighbourRevision(
            new_elect,
            self.__received_heartbeats[new_elect].start_ts,
        )
    else:
        return NeighbourRevision(-1, -1)
```

Листинг 11 — Метод выбора узла из множества возможных.

```
possible_backups = set(filter(
    lambda node: self.__received_heartbeats[node].state != REDUCER and (
            node != self.__reducer.id or
            self.__received_heartbeats[node].start_ts != self.__reducer.start_ts
    ),
    available_nodes,
))
self_proclaimed_backups = set(filter(
    lambda node: self.__received_heartbeats[node].state == BACKUP_REDUCER,
    possible_backups
))
self.__set_backup(self.__elect_from(possible_backups, self_proclaimed_backups))
```

Листинг 12 — Выбор резервного редуктора.

Код, соответствующий выбору редуктора представлен в листинге 13.

Докажем ряд свойств, которыми описанный алгоритм будет обладать.

Лемма 4.1. *Если у некоторого узла оба поля* `reducer` *и* `backup_reducer` *соответствуют некоторым узлам площадки, то эти узлы различны.*

Доказательство. Напрямую следует из построения: при получении heartbeat-сообщения мы явно проверяем, что обновленные поля не совпали, и инвалидируем менее актуальное; в алгоритме выбора мы исключаем из множества возможных редукторов и резервных редукторов узлы таким образом, что после завершения алгоритма поля не могут совпадать. □

Лемма 4.2. *Пусть с момента времени t ни один из работающих на этот момент времени узлов не отказывал, и новые узлы в работу не вводились. Также положим,*

```
if self.__is_node_alive(self.__reducer):
    return

possible_reducers = set(filter(
    lambda node: node != self.__backup_reducer.id,
    available_nodes,
))
self_proclaimed_reducers = set(filter(
    lambda node: self.__received_heartbeats[node].state == REDUCER,
    possible_reducers,
))
self.__set_reducer(self.__elect_from(
    possible_reducers,
    self_proclaimed_reducers
))
```

Листинг 13 — Выбор редуктора.

что сеть на площадке обладает достаточным качеством для того, чтобы каждая пара рабочих узлов за промежуток времени Δ_{dead} смогла обменяться актуальными (отправленными в данный промежуток) heartbeat-сообщениями, а доставка heartbeat-сообщений занимала не больше Δ_{delay} времени. Тогда не позже, чем через $\Delta_{delay}+9\Delta_{dead}$ времени после t, поля `reducer` *и* `backup_reducer` *каждого из рабочих узлов площадки будут совпадать и соответствовать различным рабочим узлам, и впредь они не будут меняться.*

Доказательство. Возможна ситуация, когда непосредственно перед моментом времени t некоторые узлы отказали, успев отправить свои heartbeat-сообщения. Это может повлиять на следующее выполнение алгоритма выбора. Поэтому положим $t_1 = t + \Delta_{delay} + \Delta_{dead}$ и далее будем работать с ним, чтобы исключить влияние таких сообщений на очередной запуск алгоритма выбора.

Если на площадке в момент времени t_1 был всего один рабочий узел, то либо он уже считал себя редуктором, либо не позже, чем через Δ_{dead} сработает таймер, по которому запустится алгоритм выбора, и, согласно ему, данный узел в качестве редуктора выберет самого себя. Таким образом, для одного узла на площадке лемма доказана.

В ином случае, если на площадке было хотя бы 2 рабочих узла, то не позже, чем через Δ_{dead} после t_1 каждый из узлов будет иметь валидные значения полей `reducer` и `backup_reducer`, соответствующие одному из рабочих узлов площадки. Действительно, за это время он либо получит heartbeat-сообщения от узлов, которых он считал (или посчитал в момент получения) редукторами или резервными редукторами, либо хотя бы раз запустит алгоритм выбора явно, имея информацию о heartbeat-сообщениях каждого рабочего узла. Коль скоро рабочих узлов хотя бы два, в процессе выполнения алгоритма и редуктор, и резервный редуктор будут выбраны из них.

Покажем, что не позже, чем через $4\Delta_{dead}$ после t_1, найдутся узлы, которые будут считать себя редукторами. Заметим, что если в условиях леммы в некоторый момент времени после t на площадке есть хотя бы один узел, считающий себя редуктором, то с этого момента времени такой узел найдется всегда: узел с наибольшим идентификатором среди подобных может перестать быть таковым лишь в двух случаях: 1) узел отказал; 2) он получил heartbeat-сообщение от другого подобного узла с бóльшим идентификатором. По условию отказов с момента времени t не было, а значит первый случай не возможен. Возникновение второго случая означает, что появился другой узел, считающий себя редуктором, а значит мы доказали наше промежуточное замечание. Это также означает, что после появления узла, считающего себя редуктором, резервные редукторы не будут становится редукторами. Значит после появления узла, считающего себя редуктором, и последующего появления узла, считающего себя резервным редуктором, узлы последнего вида также будут существовать всегда — из аналогичных рассуждений. Таким образом, если в момент времени t_1 узел, считающий себя редуктором, существовал, то такой узел будет существовать всегда. Иначе рассмотрим четыре случая:

1. На момент времени t_1 были узлы, считающие себя резервными редукторами и считающие редуктором отказавший узел (либо имеющие невалидное значение соответствующего поля). Тогда в течение отрезка времени Δ_{dead} один из этих узлов либо получит heartbeat-сообщение от узла, считающего себя редуктором (получили, что хотели), либо запустит алгоритм выбора и назначит редуктором себя (аналогично).

2. На момент времени t_1 были узлы, считающие себя резервными редукторами, но все они считали редукторами рабочие узлы. Этот случай сводится к предыдущему с тем лишь отличием, что для запуска алгоритма выбора может понадобится не Δ_{dead} времени, а $\Delta_{dead}(n_e + 1) = 2\Delta_{dead}$, т.к. дополнительное время может понадобиться для разрешения образовавшегося цикла.

3. Узлов, считающих себя резеврными редукторами, на момент времени t_1 не было, но были узлы, считающие резервным редуктором отказавший узел (либо имеющие невалидное значение соответствующего поля). Утверждается, что за следующий отрезок в Δ_{dead} времени хотя бы один из этих узлов запустит алгоритм выбора со словарем `received_heartbeats`, максимальный ключ в котором будет равен идентификатору этого самого узла. Первый такой запуск приведет к тому, что данный узел выберет себя в качестве резервного редуктора. После этого данный случай сводится к одному из предыдущих.

4. Все узлы на момент времени t_1 считали резервным редуктором другой рабочий узел, но не себя. Данный пункт сводится к предыдущему аналогично тому, как

пункт 2 сводится к пункту 1.

Итого, через $4\Delta_{dead}$ времени после t_1 будут узлы, считающие себя редукторами. Значит через $6\Delta_{dead}$ времени после t_1 гарантировано будут узлы, считающие себя резервными редукторами — по аналогичным рассуждениям из пунктов 3-4. Значит в момент времени $t_1 + 6\Delta_{dead}$ все узлы будут иметь валидные значения полей `reducer` и `backup_reducer`, и будут существовать как узлы, считающие себя редукторами, так и узлы, считающие себя резервными редукторами. Это означает, что в момент времени $t_1 + 6\Delta_{dead}$ мы можем выбрать среди узлов, считающих себя редукторами и считающих себя резервными редукторами, узлы с наибольшими идентификаторами. Поскольку они имеют наибольшие идентификаторы среди подобных узлов, никто не сможет изменить их поля `reducer` и `backup_reducer` соответственно. В каждый из следующих двух отрезков времени по Δ_{dead} каждый они отправят heartbeat-сообщения всем другим узлам на площадке, а значит после этого поля `reducer` и `backup_reducer` всех узлов будут соответствовать этим двум узлам — либо потому что эти поля имели значение идентификатора меньше, либо в следствии выполнения алгоритма перевыборов из-за возникновения циклов. Осталось заметить, что алгоритм выбора по истечении таймера с момента времени $t_1 + 8\Delta_{dead}$ больше запускаться не будет — по условию каждый узел в течение времени Δ_{dead} будет получать heartbeat-сообщения с ожидаемыми полями состояния от узлов, которых он считает редуктором и резервным редуктором. Теорема доказана. □

Теорема 4.1. *При наличии качественной сети, обладающей условиями из леммы 4.2, любой отрезок времени длиннее $\Delta_{delay}+9\Delta_{dead}$ с фиксированным набором рабочих узлов будет содержать подотрезок с тем же правым концом, на котором выбор редуктора и резервного редуктора будет согласован, т.е. на нем будут выполняться следующие условия.*

- *С точки зрения любого рабочего узла ни один из узлов не будет считаться редуктором и резервным редуктором одновременно.*
- *При условии отсутствия разделения сети внутри площадки в один момент времени на ней не будет больше одного узла, которого хотя бы один работающий узел считал бы редуктором. Аналогично для резервных редукторов.*
- *При наличии хотя бы одного рабочего узла, редуктор будет выбран на каждом из узлов. При наличии хотя бы двух рабочих узлов, на каждом узле будет выбран резервный редуктор.*

Доказательство. Напрямую следует из лемм 4.1 и 4.2. □

Теорема 4.2. *Пусть в некоторый момент времени t на площадке было больше одного рабочего узла, и выбор редуктора a и резервного редуктора b был согласован, т.е.*

выполнялись все три условия из теоремы 4.1. Пусть при этом сеть на площадке качественная: выполняются уловия на сеть из леммы 4.2, а также любое heartbeat-сообщение либо доходит до всех рабочих узлов с минимальными задержками, либо не доходит ни до кого. Предположим, следующий отказ редуктора a произошел в момент времени $t_{fail} > t$, и до момента t_{fail} резервный редуктор b не отказывал. Тогда если до момента времени $t_{fail} + \Delta_{delay} + 2\Delta_{dead}$ узел b продолжал исправно работать, к этому моменту все узлы площадки будут считать его редуктором.

Доказательство. Заметим, что все рабочие узлы на момент времени t_{fail} либо считали узел b резервным редуктором, либо начали работать недавно и еще не успели получить от него heartbeat-сообщение. При этом узлы второго типа до первого срабатывания таймера Δ_{dead} еще успеют получить от него heartbeat-сообщение — по условию. Рассмотрим момент времени $t_{lh} < t_{fail} + \Delta_{delay}$, когда узел b обработал последнее heartbeat-сообщение от узла a. В течение $2\Delta_{dead}$ времени после момента t_{lh} резервный редуктор b должен будет запустить алгоритм перевыборов, т.к. после второго срабатывания таймера он гарантированно заметит, что не получил ни одного heartbeat-сообщения от узла, которого он считал редуктором. Если до этого момента никто другой не стал редуктором, новым редуктором, будучи резервным редуктором, он назначит себя.

Покажем, что именно так и произойдет: до момента времени $t_{lh} + 2\Delta_{dead}$ никто, кроме узла b, себя редуктором считать не будет. Каждый другой узел, аналогично b, заметит, что редуктор a отказал, а значит по алгоритму перевыборов необходимо заменить его резервным редуктором. Причем, коль скоро по условию теоремы любое heartbeat-сообщение, дошедшее до одного узла, доходит до всех других, запуск алгоритма перевыборов на всех узлах произойдет не раньше, чем через Δ_{dead} времени после $t_{lh} - \Delta_{delay}$. Как мы заметили в самом начале доказательства, при запуске алгоритма перевыборов в словаре `received_heartbeats` любого узла будет содержаться heartbeat-сообщение от узла b, а значит даже если до того момента сам узел b не будет считать себя редуктором, то его посчитает редуктором узел, на котором алгоритм запускается, и, соотвественно, до момента времени $t_{lh} - \Delta_{delay} + 2\Delta_{dead}$ все узлы будут считать редуктором либо b, либо отказавший a, т.е. не себя (за исключением самого b). Тем не менее, во временно́й промежуток между $t_{lh} - \Delta_{delay} + 2\Delta_{dead}$ и $t_{lh} + 2\Delta_{dead}$ у одного из узлов может сработать очередной таймер для проверки пришедших heartbeat-сообщений, а поскольку мы не можем гарантировать, что к началу данного отрезка времени b уже будет считать себя редуктором, последнее полученное от него heartbeat-сообщение может содержать поле `state` равное `BACKUP_REDUCER`, что не будет соответствовать ожиданиям узла, на котором истек таймер. Однако этого не будет достаточно для запуска алгоритма перевыборов, т.к. мы завели поля `reducer_till_expiration` и `backup_till_expiration`, и если их начальные значения по умолчанию положительные, то в данном случае значение поля `reducer_till_expiration` лишь снизится на

1, но отрицательным не станет. Таким образом, до следующего срабатывания таймера на этом же узле узел b уже успеет стать редуктором и отправить heartbeat-сообщение ожидаемого вида. Таким образом, узел b, ранее бывший резервным редуктором, станет редуктором, и на момент времени $t_{lh} + 2\Delta_{dead} < t_{fail} + \Delta_{delay} + 2\Delta_{dead}$ данный выбор будет согласован в пределах площадки. Теорема доказана □

Таким образом, в данном разделе мы рассмотрели различные способы решения задачи выбора лидера и реализовали собственный механизм выбора редуктора и резервного редуктора на основе изученных подходов. Мы также показали, что данный выбор в построенном алгоритме будет становиться согласованным за ограниченное количество времени.

5 Маршрутизация

Для эффективного обмена данными между узлами различных площадок необходимо строить оптимальные маршруты в сети на уровне узлов системы. В данном разделе обсудим решение задачи маршрутизации.

5.1 Постановка задачи маршрутизации

Определение 5.1. Назовем *маршрутом* произвольный кортеж конечной длины большей 1 из узлов сети, а *маршрутом из узла a в узел b* — маршрут, у которого первый и последний элементы равны соответственно a и b.

Определение 5.2. Введем также понятие *множества маршрутов из узла a в узел b*: $\mathrm{Routes}(a, b) := \bigcup_{k=0}^{\infty} (a \times S^k \times b)$. *Множество всех маршрутов* — $\mathrm{Routes} := \bigcup_{a,b \in S} \mathrm{Routes}(a, b)$. *Множество маршрутов от узла a до площадки S_k*: $\mathrm{Routes}(a, S_k) = \bigcup_{b \in S_k} \mathrm{Routes}(a, b)$.

Для однозначности будем отождествлять все маршруты, получающиеся друг из друга увеличением или уменьшением длин подпоследовательностей из одинаковых узлов. Так, например, (a, b, c) и (a, b, b, c) — это один и тот же маршрут, но при этом (a, b, c) и (a, c) — разные (мы не можем полностью удалять цепочку одинаковых узлов из маршрута). Для удобства введем

Определение 5.3. *Приведенным видом* маршрута $r \in \mathrm{Routes}(a, b)$ назовем кортеж $\mathrm{reduced}(r) \in \mathrm{Routes}(a, b)$, который был получен из r последовательным уменьшением длин подотрезков из одинаковых узлов, причем либо любые соседние элементы в $\mathrm{reduced}(r)$ различны, либо его длина равна 2 и $a = b$.

Утверждение 5.1 (Корректность определения приведенного вида маршрута)**.** *Для всех маршрутов $r \in \mathrm{Routes}$ приведенный вид $\mathrm{reduced}(r)$ определен однозначно.*

Доказательство. Заметим, что при удалении одного элемента из подотрезка одинаковых элементов неединичной длины сама последовательность из таких подотрезков не меняется. При этом данный процесс конечен и заканчивается либо тогда, когда длины всех этих отрезков равны 1, либо когда останется лишь один отрезок длины 2. Значит мы можем однозначно определить приведенный вид любого маршрута, который будет выглядеть именно таким образом. □

Определение 5.4. Пусть $r \in \text{Routes}$. Если его приведенный вид состоит из двух одинаковых элементов, будем считать, что *длина данного маршрута* равна 0. В ином случае *длиной данного маршрута* будем называть целое число, равное $|\operatorname{reduced}(r)| - 1$, где $|\operatorname{reduced}(r)|$ — длина кортежа $\operatorname{reduced}(r)$.

Определение 5.5. Пусть $r = (a_1, a_2, \ldots, a_k)$ — маршрут. Тогда *суммарной метрикой маршрута r в момент времени $t \in T$* называется

$$m_t(r) := \sum_{j=1}^{k-1} m_t(a_j, a_{j+1}).$$

Задача маршрутизации заключается в построении маршрутов от узла-отправителя до узла-получателя. Как правило [6, 12], ее решают, сводя к задаче выбора следующего узла в маршруте. Мы также будем формулировать задачу в терминах построения алгоритма выбора следующего узла, причем в силу замечания 3.2 мы будем строить маршруты не до конкретных узлов, а до целых площадок — если мы умеем эффективно передавать данные до произвольного узла площадки узла-получателя, достроить полный маршрут можно тривиальным образом, что не сильно повлияет на суммарную метрику.

Определение 5.6. Пусть $\{t_i\}_{i=1}^{\infty} \colon \forall\ j < k \implies t_j \le t_k$ — неубывающая последовательность моментов времени из T, $\forall i \in \mathbb{N}\ S_{k_i} \in \mathcal{S},\ a_i \in S$. Последовательность узлов $\{\operatorname{real}_{t_i}(a_i, S_{k_i})\}_{i=1}^{\infty}$ назовем *историей выполнения* алгоритма маршрутизации, если на запрос о следущем узле в маршруте от узла a_i до площадки S_{k_i} он вернул узел $\operatorname{real}_{t_i}(a_i, S_{k_i})$ в момент своего завершения t_i. Про сам узел $\operatorname{real}_{t_i}(a_i, S_{k_i})$ будем говорить, что в момент времени t_i он являлся *реальным прокси* узла a_i до площадки назначения S_{k_i}.

Определение 5.7. Пусть $r_{best} = \operatorname{reduced}(\arg\min\{|r| \colon r \in \text{Routes}(a, b),\ b \in S_k,\ m_t(r) = \min_{p \in \text{Routes}(a,b),\ b \in S_k} m_t(p)\})$ — приведенный вид кратчайшего маршрута от узла $a \in S$ до некоторого узла площадки S_k среди всех подобных маршрутов с минимальной суммарной метрикой в момент времени $t \in T$. Второй элемент кортежа r_{best} будем называть *идеальным прокси узла a до целевой площадки S_k в момент времени t.*

Замечание 5.1. Идеальный прокси может быть не единственен. Если существуют маршруты одинаковой минимальной длины от a до S_k с минимальной метрикой, в приведен-

ном виде которых вторые элементы различны, все эти узлы будут считаться идеальными прокси. Обозначение: $\mathrm{Best}_t(a, S_k)$ — множество идеальных прокси.

Утверждение 5.2. *Пусть $a \in S_k$. Тогда идеальным прокси узла a до площадки S_k в любой момент времени будет являться сам узел a.*

Доказательство. Напрямую следует из неотрицательности метрики. □

Утверждение 5.3. *В любой момент времени идеальные прокси существуют.*

Доказательство. Зафиксируем площадку назначения $S_k \in \mathcal{S}$, начальный узел $a \in S$, а также момент времени $t \in T$. Если $a \in S_k$, сам узел a является идеальным прокси, доказывать нечего — см. предыдущее утверждение 5.2. Пусть тогда $a \notin S_k$. В силу определения метрики 3.5, метрика соединения между любыми двумя узлами в момент времени t — неотрицательная функция, поэтому в нашей сети не бывает маршрутов-циклов отрицательной суммарной метрики. Это значит, что если маршрут с минимальной метрикой существует, то он является простым — все его узлы попарно различны. Рассмотрим всевозможные простые маршруты из узла a во все узлы площадки S_k: $\{(a_1, \ldots, a_l) \in \mathrm{Routes}(a, S_k) \mid \forall\ i \neq j\ a_i \neq a_j\}$. Таких маршрутов конечное число, поэтому среди них можно выбрать маршрут с минимальной метрикой, а затем взять кратчайший маршрут из оставшихся — второй узел в нем будет являться идеальным прокси узла a до площадки S_k в момент времени t. □

Оптимизационная задача маршрутизации состоит в том, чтобы реальные прокси были достаточно похожи на идеальные с точки зрения метрики. В случае «идеальной» реализации протокола маршрутизации реальные прокси должны совпадать с идеальными, однако ввиду постоянно изменяющейся динамической информации каналов такой сценарий нереалистичен: после выбора очередного реального прокси для посылки сообщения параметры сети могут изменится таким образом, что выбранный узел не будет включен в оптимальный маршрут. Поэтому мы будем добиваться построения алгоритма, обладающего гарантией *согласованности в конечном счете* [13]. Сформулируем, что именно мы будем понимать под этим требованием в терминах протокола маршрутизации (выбора прокси):

Определение 5.8. Алгоритм выбора прокси обладает гарантией *согласованности в конечном счете*, если и только если после наступления момента времени, начиная с которого значения метрики всех каналов остаются постоянными, обязательно наступит такой момент времени, после которого все реальные прокси будут совпадать с идеальными.

Более формально: пусть $\{\mathrm{real}_{t_i}(a_i, S_{k_i})\}_{i=1}^{\infty}$ — некоторая история выполнения алгоритма. Чтобы алгоритм обладал гарантией *согласованности в конечном счете*, необходимо и достаточно показать, что

$$\exists t_{const} \in T\colon \forall t \geq t_{const}\ \forall a, b \in S\ m_t(a,b) = m_{t_{const}}(a,b) \implies$$
$$\implies \exists i_{EC} \in \mathbb{N}\colon \forall i \geq i_{EC}\ \mathrm{real}_{t_i}(a_i, S_{k_i}) \in \mathrm{Best}_{t_i}(a_i, S_{k_i}). \tag{1}$$

5.2 Таблица маршрутизации

Сведем поставленную задачу к построению таблицы маршрутизации, с помощью которой узлы непосредственно будут выбирать реальные прокси до площадок назначения.

Определение 5.9. *Таблицей маршрутизации* будем называть словарь, ключами которого будут являться площадки $\mathcal{S}$ (например, их индексы, или любые другие идентификаторы), а значениями — экземпляры protobuf-сообщения, описанного в листинге 14.

```
message Route {
    int64 next_hop = 1;
    double metric = 2;
    uint64 length = 3;
}
```

Листинг 14 — Protobuf-сообщение для хранения и передачи данных о реальных прокси в таблице маршрутизации.

Таким образом, мы можем выразить и саму таблицу маршрутизации в виде protobuf-сообщения (листинг 15).

```
message RouteTable {
    map<uint64, Route> routes = 1;
}
```

Листинг 15 — Таблица маршрутизации в виде protobuf-сообщения.

В данном определении `next_hop` — уникальный идентификатор узла. Таким образом, каждый узел системы будет иметь собственную таблицу маршрутизации, в которой будет содержаться информация о прокси `next_hop` в «наилучших» с точки зрения данного узла маршрутах до каждой площадки назначения. Причем для каждого маршрута он также будет явно хранить некоторую оценку `metric` суммарной метрики данного маршрута и его длину `length`.

Имея данную информацию, узел может выбрать в качестве реального прокси узел `next_hop`, соответствующий площадке назначения. Так мы свели задачу маршрутизации к задаче построения таблицы маршрутизации, решение которой опишем в следующем подразделе.

5.3 Построение таблицы маршрутизации

5.3.1 Начальная инициализация

Таблицу маршрутизации будем строить итеративно. Поскольку построение финальной версии может занимать произвольное количество времени, нам необходимо поддерживать все узлы в состояниях, обеспечивающих построение корректных маршрутов, чтобы иметь возможнось отправлять сообщения до окончательной стабилизации всех таблиц. Поэтому мы должны инициализировать все `next_hop` такими значениями, которые позволят строить корректные, но, быть может, неоптимальные, маршруты до площадок назначения. Для этого могут подойти *наивные прокси*:

Определение 5.10. *Наивным прокси узла $a \in S$ до целевой площадки S_k в момент времени $t \in T$* будем называть $\text{naive}_t(a, S_k) := \arg\min\limits_{b \in S_k} m_t(a, b)$ — узел данной площадки, соединение до которого обладает минимальной метрикой.

Коль скоро каждый узел $a \in S$ в каждый момент времени $t \in T$ имеет доступ к динамической информации до всех других узлов $\text{Info}_t(a, b)$ ($b \in S \setminus \{a\}$) и может считать по ней метрику $m_t(a, b)$, то он может определить для себя набор наивных прокси и соответствующим образом инициализировать свою таблицу маршрутизации. С точки зрения метрики полученные маршруты не хуже, чем прямые соединения до произвольного узла площадки назначения, однако выбранные прокси могут не являться идеальными, если есть маршруты через промежуточные площадки или узлы с меньшей суммарной метрикой.

Замечание 5.2. В действительности узел может считать метрику соединений не только до других узлов, но и до себя самого — она равна нулю согласно определению 3.5. Поэтому он сам будет считать себя наивным прокси до своей же площадки.

Несмотря на то, что начальные маршруты могут не являться наилучшими, с этого момента можем считать, что узлы способны выбирать прокси по таблице маршрутизации таким образом, что при последовательном выборе следующего прокси, стартуя из начального узла, мы попадем в площадку назначения. Далее, при внесении изменений в таблицу, мы будем пытаться сохранить этот инвариант. Однако полностью его соблюсти нельзя в силу естественных причин — см. замечание 5.3.

Замечание 5.3. В действительности, если метрика соединений будет постоянно меняться, таблицы маршрутизации могут перестраиваться бесконечно долго. Это может привести к сценарию, в котором из-за постоянного изменения таблиц некоторое сообщение будет перемещаться по различным узлам так, что оно никогда не достигнет площадки назначения. В качестве примера опишем экстремальный случай, который может возникнуть даже в правильно спроектированном алгоритме. Пусть узел $a \in S_i$ считает идеальным прокси до некоторой площадки узел $b \in S_j$ — он отправляет ему некоторые данные для последующей передачи в площадку назаначения. За время, пока данные

шли до узла b, его таблица маршрутизации могла измениться таким образом, что идеальным с его точки зрения уже будет узел площадки S_i (возможно, даже сам a) — и так по кругу. В итоге данные будут ходить между двумя площадками (узлами) и никогда не достигнут площадки назначения.

Коль скоро мы не можем предотвратить возникновение подобных ситуаций, нам нужен инструмент, который позволит бороться с их последствиями — с бесконечно «гуляющими» по сети сообщенями. В качестве решения при любой отправке данных по построенным маршрутам будем использовать стандартный [14, с. 2] прием с добавлением TTL поля: с каждой передачей сообщения будем уменьшать значение TTL на единицу, а если оно достигло нуля, останаливать передачу соответствующего сообщения.

Таким образом, в процессе приведения системы к согласованному состоянию мы можем использовать уже построенные таблицы маршрутизации для передачи данных между узлами. Такая возможность пригодится нам для последующего обмена построенными таблицами маршрутизации для их обновления.

5.3.2 Итеративные обновления

Теперь, когда у нас есть хоть какие-то таблицы маршрутизации, будем их обновлять и стараться привести к состояниям, обеспечивающим построение «хороших» маршрутов. Для начала определим операцию *scatter* — в дальнейшем она нам понадобится.

Определение 5.11. Выполнение операции *scatter* узлом $a \in S$ предполагает передачу некоторых данных, находящихся на узле a, на все узлы системы. Если узел `a` должен выполнить scatter данных `data`, будем обозначать такую операцию следующим образом: `scatter(a, data)`.

Выполнение данной операции обсудим в подразделе 5.3.3, а пока абстрагируемся от ее реализации, чтобы с ее помощью описать механизм обновлений таблиц.

Осуществлять scatter в данном разделе мы будем для таблиц маршрутизации. Поэтому обозначим через `scatterRT(a)` операцию `scatter(a, route_table)`, где `route_table` — текущая таблица маршрутизации узла `a`.

Весь алгоритм будет состоять из двух типов обновлений: *регулярные* и *экстренные*:

- Регулярные обновления — предназначены для улучшения маршрутов и информировании новых (перезапустившихся) узлов:
  - каждый узел `a` выполняет `scatterRT(a)` один раз в определенный промежуток времени;
  - каждый узел $a \in S$ при получении регулярного обновления с таблицей маршрутизации другого узла $b \in S \backslash \{a\}$ для каждой площадки назначения сравнива-

ет сумарные метрики текущего маршрута и возможного маршрута, в котором вторым узлом является b, а также их длины в случае равенства суммарных метрик. В соответствии с полученными результатами узел-получатель обновляет свою таблицу.

- Экстренные обновления — предназначенны для незамедлительного информирования других узлов о значительном ухудшении метрики для ускорения схождения сети после неполадок:

  – для определения значительных ухудшений метрики задается параметр δ_m, который может меняться со временем (в данной работе мы абстрагируемся от конкретной реализации определения текущего значения δ_m);

  – экстренные обновления содержат не всю таблицу маршрутизации узла, а лишь тот маршрут, в котором произошли значительные изменения метрики;

  – экстренные обновления генерируются и отправляются в процессе стандартного механизма перестроения маршрутов, при этом сами экстренные обновления обрабатываются тем же методом `update_route`.

Реализация методов для обработки приходящих обновлений о маршрутах представлена в листинге 16.

До сих пор мы не описали механизмы по защите от отказов узлов: в случае, если узел a принял за прокси до площадки $S_k \in \mathcal{S}$ узел b ($b \notin S_k$), то в случае отказа узла b узел a согласно описанному на данный момент алгоритму продолжит считать узел b прокси до S_k (по крайней мере до тех пор, пока не получит обновление с лучшей суммарной метрикой). Согласно замечанию 3.1 в этом случае он не может являться идеальным прокси до площадки S_k. Значит нынешний алгоритм по умолчанию не может обладать гарантией согласованности в конечном счете при наличии отказов узлов в сети. Для решения данной проблемы заведем еще один таймер для проверки доступности реальных прокси, который будет регулярно проверять значение метрики непосредственного соединения до текущего реального прокси. В случае, если данная метрика достаточно большая для того, чтобы считать прокси отказавшим, нужно выбрать новый реальный прокси и сгененрировать экстренное обновление. В качестве прокси по умолчанию будем использовать наивные прокси — как мы это делали в начальной инициализации таблиц маршрутизации (подраздел 5.3.1).

Покажем, что описанный алгоритм позволяет строить таблицы маршрутизации таким образом, что итоговый алгоритм выбора прокси будет обладать гарантией согласованности в конечном счете.

Замечание 5.4. Если известно, что с некоторого момента времени метрики каналов связи не меняются, то с того же момента времени все отказавшие узлы остаются отказав-

```python
def handle_update(self,
                  updated_routes: network_pb2.RouteTable,
                  source_node: int) -> None:
    emergency_updates: tp.Dict[int, network_pb2.Route] = {}
    for group, route in updated_routes.routes.items():
        self.__update_route(group, source_node, route, emergency_updates)

    self.__scatter_updates(emergency_updates)

def __update_route(self, target_group: int, proposed_next_hop: int,
                   proposed_route: network_pb2.Route,
                   emergency_updates: tp.Dict[int, network_pb2.Route]) -> None:
    proposed_metric = proposed_route.metric + \
                      self.__metric_service.get_direct_metric(proposed_next_hop)
    proposed_length = proposed_route.length + 1
    if target_group not in self.__route_table.routes:
        self.__route_table.routes[target_group] = network_pb2.Route(
            next_hop=proposed_next_hop,
            metric=proposed_metric,
            length=proposed_length,
        )
        emergency_updates[target_group] = self.__route_table.routes[target_group]
        return

    current_route = self.__route_table.routes[target_group]
    old_metric = current_route.metric
    if proposed_metric < current_route.metric or \
            (proposed_metric == current_route.metric and
             proposed_length < current_route.length):
        current_route.next_hop = proposed_next_hop
        current_route.metric = proposed_metric
        current_route.length = proposed_length
    elif current_route.next_hop == proposed_next_hop:
        current_route.metric = proposed_metric
        current_route.length = proposed_length

    if abs(old_metric - current_route.metric) >= \
            self.__get_emergency_metric_delta():
        emergency_updates[target_group] = current_route
```

Листинг 16 — Обработка приходящих обновлений о маршрутах.

шими, а все работоспособные продолжают работать — это следует из того же замечания 3.1.

Теорема 5.1. *Пусть* $\forall (a, b) \in S^2\ \forall t \in T\ \exists t_{success} \geq t$: *в момент времени* $t_{success}$ *узел* b *получил данные, отправленные при помощи операции* `scatterRT(a)`. *Другими словами, на любом суффиксе временно́й шкалы каждый узел хотя бы раз получал информацию о таблицах маршрутизации любого другого узла. В таких условиях описанный алго-*

ритм обладает гарантией согласованности в конечном счете 5.8.

Доказательство. Зафиксируем историю выполнения алгоритма маршрутизации: $\{\text{real}_{t_i}(a_i, S_{k_i})\}_{i=1}^{\infty}$. Пусть с момента времени $t_{fix} \in T$ метрики всех непосредственных соединений между узлами зафиксировались. Покажем, что после t_{fix} наступит момент времени $t_{EC} \geq t_{fix}$, с которого реальные прокси будут совпадать с идеальными.

Докажем лемму, которая понадобится в будущем:

Лемма 5.1. *Пусть* $p_1 \in \text{Best}_{t_{fix}}(a, S_k)$, $r = (a, p_1, p_2, \ldots)$ — *соответствующий данному идеальному прокси маршрут с минимальной метрикой в приведенном виде из определения 5.7. Тогда* $\forall t \geq t_{fix}$ $p_2 \in \text{Best}_t(p_1, S_k)$.

Доказательство. Заметим, что нам достаточно доказать лемму для момента времени t_{fix}, т.к. значения метрики после этого остаются неизменными. Предположим противное: пусть p_2 не является идеальным прокси узла p_1 до площадки S_k в момент времени t_{fix}. Обозначим за r_{suff} суффикс r без первого элемента — это маршрут от узла p_1 до площадки S_k. Тогда

$$\exists\, r_{alt} \in \text{Routes}(p_1, S_k) \colon m_{t_{fix}}(r_{alt}) < m_{t_{fix}}(r_{suff}),$$

причем второй элемент $\text{reduced}(r_{alt})$ не равен p_2. Вспомним, что метрика маршрута определяется (5.5) как сумма метрик последовательно входящих в него соединений:

$$\begin{aligned} m_{t_{fix}}(r_{suff}) = m_{t_{fix}}(r) - m_{t_{fix}}(a, p_1) \implies \\ \implies m_{t_{fix}}(r_{alt}) < m_{t_{fix}}(r) - m_{t_{fix}}(a, p_1) \implies \\ \implies m_{t_{fix}}((a) \cdot r_{alt}) = m_{t_{fix}}(a, p_1) + m_{t_{fix}}(r_{alt}) < m_{t_{fix}}(r). \end{aligned}$$

Таким образом, мы нашли маршрут $(a) \cdot r_{alt}$, являющийся маршрутом от узла a до площадки S_k, суммарная метрика которого оказалась меньше суммарной метрики r. Это противоречит выбору маршрута r по условию. Значит наше предположение оказалось неверным, и узел p_2 действительно является идеальным прокси p_1 до площадки S_k. □

Согласно утверждению 5.3, идеальный прокси для каждой пары «узел-площадка» существует. Поскольку значения метрик не менялись после момента времени t_{fix}, вместе с ними зафиксировались и множества идеальных прокси, а значит и маршруты с наименьшими суммарными метриками, в которых данные идеальные прокси находятся на втором месте. Будем называть такие маршруты *наилучшими* для определённой пары «узел-площадка».

Зафиксируем некоторую площадку назначения $S_k \in \mathcal{S}$. Поскольку обновление маршрутов до разных площадок полностью независимо (не считая использование построенных маршрутов для обмена сообщениями), достаточно доказать теорему про идеальные прокси до одной фиксированной площадки.

Для удобства введем понятие *реальной метрики узла a до площадки* S_k — ей будет являться значение, хранящееся в соответствующей записи таблицы маршрутизации узла a на текущий момент времени. Докажем еще одну вспомогательную лемму:

Лемма 5.2. *Обозначим за* U *все обновления, отправленные до момента времени* t_{fix}*, которые при этом на момент времени* t_{fix} *не дошли до всех узлов. Утверждается, что с момента времени* t_{fix} *любое значение реальной метрики будет равно конечной сумме слагаемых* $\sum_{i=1}^{q} m_i$*, каждое из которых удовлетворяет хотя бы одному из следующих свойств:*

1. m_i *равняется значению метрики одного из прямых соединений между узлами;*
2. m_i *является реальной метрикой одного из узлов на момент времени* t_{fix}*;*
3. m_i *равняется значению метрики, содержащемуся в одном из обновлений из множества* U*.*

Доказательство. Поскольку на интервалах между обработками обновлений никакие реальные метрики не меняются, нам достаточно доказать, что утверждение выполняется после каждой обработки обновления тем или иным узлом. Множество таких моментов времени счетно и упорядочено по времени, а значит для него можно применить метод математической индукции.

1. База. В момент времени t_{fix} реальные метрики являются суммой из одного слагаемого типа 2 — утверждение леммы выполнено, база индукции доказана.

2. Переход. Пусть в момент времени t некоторая реальная метрика изменилась. Это могло произойти только в результате обработки пришедшего обновления (мы уже упоминали, что в условиях теоремы узлы не подвержены отказам, хотя это бы и не помешало доказательству леммы). Вспомним, как происходит изменение метрики после обработки обновления — вместо старого значения в таблицу маршрутизации записывается сумма значения метрики непосредственного соединения до узла-источника обновления и значения метрики, содержащегося в самом обновлении. Первое слагаемое имеет тип 1 из условия леммы. Осталось разобраться со вторым. Всего возможно 2 случая: 1) полученное обновление входит во множество U, и тогда второе слагаемое имеет третий тип из условия леммы; 2) обновление отправлено после наступления момента времени t_{fix}, но до наступления момента времни t, а значит содержит реальную метрику некоторого узла, которая по предположению индукции представляет из себя сумму слагаемых одного из трех типов. Значит и реальная метрика после обновления будет иметь нужный вид. Переход доказан. □

Теперь перейдем непосредственно к доказательству теоремы.

Для каждого узла $a \in S$ покажем, что

$$\exists\, t_a \geq t_{fix} \colon \forall\, i \in \mathbb{N}\, \Big((t_i \geq t_a,\ a_i = a,\ k_i = k) \implies \mathrm{real}_{t_i}(a_i, S_{k_i}) \in \mathrm{Best}_{t_{fix}}(a_i, S_{k_i}) \Big), \quad (2)$$

то есть таблица мрашрутизации узла a с момента времени t_a будет содержать запись о маршруте до площадки S_k, в которой значение `next_hop` будет равно идентификатору одного из идеальных прокси, причем *реальная метрика будет равна* $M = m_{t_{fix}}(r_a)$*, где* r_a *— соответствующий наилучший маршрут.*

Множество $\{m_{t_{fix}}(r) \mid a \in S_k,\ r$ — наилучший маршрут от a до $S_k\}$ конечно, поэтому можем воспользоваться методом математической индукции по возрастанию значения суммарной метрики наилучшего маршрута.

1. База. Нулевой метрикой до площадки S_k обладают узлы этой же площадки. В замечании 5.2 мы описали данную ситуацию: такие узлы считают прокси сами себя, и их метрика в таблице маршрутизации равна нулю. Доказывать нечего.

2. Переход. Пусть узлы $L_M = \{a_1,\ a_2,\ \ldots,\ a_m\}$ таковы, что суммарные метрики их наилучших маршрутов до площадки S_k одинаковы и равны M. Предположим, что для всех узлов, имеющих наилучшие маршруты суммарной метрики меньше M, предположение индукции выполнено. Обозначим их за $L_{<M}$. Их конечное число, поэтому можно взять наибольший момент времени среди t_a из (2), где $a \in L_{<M}$ — обозначим его за $t_{<M}$. Таким образом, к моменту $t_{<M}$ все узлы $L_{<M}$ построили маршруты через идеальные прокси, знают суммарные метрики данных маршрутов, и соответствующие записи в их таблицах маршрутизации больше никогда не изменятся. Предположим, что к моменту времени $t_{<M}$ остались узлы, не входящие в множество $L_{<M}$, но в таблицах маршрутизации которых реальное значение метрики маршрута до площадки S_k ниже M. Покажем, что наступит такой момент времени, что для всех таких узлов реальное значение метрики увеличится хотя бы до M. Иначе, если таких узлов изначально не было, перейдем к следующему этапу доказательства.

Итак, пусть описанные узлы существуют. Заметим, что значений каждого типа из леммы 5.2 конечное число. Это означает, что возможных значений реальных метрик, которые бы были меньше M также конечное число. Обозначим данное конечное множество возможных значений реальных метрик за $\Sigma_{<M}$. Докажем еще одну важную лемму:

Лемма 5.3. $\forall m_i \in \Sigma_{<M}\ \exists t_i \geq t_{<M} \colon \forall b \in S \setminus L_{<M}$ *начиная с момента времени* t_i *значение реальной метрики узла* b *будет больше* m_i*.*

Доказательство. Без ограничения общности будем считать, что значения m_i проиндексированы в порядке возрастания. Также заметим, что $0 \in \Sigma_{<M}$, т.к. 0 является суммой из одного слагаемого 0, являющегося значением реальной метрики узла, находящегося на площадке назначения S_k.

Докажем данную лемму методом математической индукции по i.

1. База. $i = 1 \implies m_i = 0$ (метрика неотрицательная). Однако если узел c имеет реальную метрику равную 0, то данный узел принадлежит площадке S_k, а значит $c \in L_{<M}$ — у него фиксированный реальный прокси, равный идеальному (он сам), и данный инвариант не меняется. Значит можно взять $t_1 = t_{<M}$ — с этого момента условие леммы для $m_1 = 0$ будет выполнено.

2. Переход. Пусть условие леммы выполнено для всех $i \leq j$. Докажем, что и для $i = j + 1$ тоже. Пусть $t'_{j+1} = \max\limits_{1 \leq i \leq j} t_i \geq t_{<M}$ (t_i — из условия леммы). К моменту времени t'_{j+1} некоторые узлы из $S \setminus L_{<M}$ могли отправить обновления со значением метрики меньше m_{j+1}. Найдем момент времени $t''_{j+1} \geq t'_{j+1} \geq t_{<M}$, к которому все такие обновления либо дошли до получателей и были обработаны, либо были отброшены по тем или иным причинам: истекший TTL, устаревшая версия обновления, отказы промежуточного оборудования и т.п. По предположению индукции после момента t''_{j+1} у узлов из $S \setminus L_{<M}$ значения реальных метрик превосходят m_j, а значит они не меньше m_{j+1}. Значит и обновления, исходящие от узлов $S \setminus L_{<M}$, содержат суммарную метрику не меньше m_{j+1}. Предположим, что во время или после момента времени t''_{j+1} среди этих узлов найдется такой узел b, что его реальная метрика будет равна m_{j+1}. Пусть при этом реальным прокси узла b на этот момент является узел c. Коль скоро $b \notin L_{<M}$, по предположению индукции основной теоремы:

$$\forall d \in L_{<M}\ m_{t_{fix}}(b, d) + m_{t_{fix}}(r_d) \geq M, \tag{3}$$

где r_d — один из наилучших маршрутов от d до S_k. Рассмотрим 2 случая:

1. $c \in L_{<M}$. Тогда b при получении регулярного обновления от c изменит свою реальную метрику, причем она станет не меньше M. Действительно, узлы меняют реальную метрику только в 3 случаях: 1) при отказе реального прокси, 2) при получении обновления с меньшей суммарной метрикой и 3) при ухудшении метрики маршрута через реальный прокси. Из условия теоремы (каждая пара узлов получает сообщения дург от друга) и замечания 5.4 следует, что на данный момент у нас отказов нет. Из (3) и того, что с момента времени t''_{j+1} обновления, исходящие от узлов $S \setminus L_{<M}$, содержат суммарную метрику не меньше m_{j+1}, следует, что все приходящие обновления не дадут лучшей суммарной метрики. Значит, как только b получит регулярное (или экстренное) обновление от c, он заметит, что суммарная метрика увеличилась до M или выше и обновит свою таблицу маршрутизации.

2. $c \notin L_{<M}$. В этом случае b при получении регулярного обновления от c также увеличит реальную метрику — по тем же причинам, что и в прошлом пункте. Отличие лишь в том, что новое значение реальной метрики может быть мньше M.

Таким образом, если за $t^{(b)}_{j+1} \geq t''_{j+1}$ взять момент получения и обработки узлом b регулярного обновления от своего реального прокси c, то после него реальная метрика

узла b всегда будет больше m_{j+1}. Несмотря на то, что узлов, подобных b могло оказаться несколько, в качествае t_{j+1} можем взять $\max\limits_{b} t_{j+1}^{(b)}$ — с данного момента времени условие леммы гарантированно начнет выполняться. □

С помощью леммы 5.3 мы показали, что найдется момент времени $t_{\leq M}$, после которого реальная метрика всех узлов из $L_M \subset (S \setminus L_{<M})$ будет не меньше M. В действительности для тех узлов из L_M, чей реальный прокси на момент времени $t_{\leq M}$ не будет входить в $L_{<M}$, можем провести аналогичные рассуждения из доказательства леммы 5.3 и показать, что их реальные метрики станут строго больше M (без ограничения общности будем считать, что это уже произошло к моменту времени $t_{\leq M}$). Итого, узлы $a_i \in L_M$ разобьются на 2 вида:

1. На момент времени $t_{\leq M}$ узел a_i имеет реальную метрику M и реальный прокси $c \in L_{<M}$. В этом случае реальный прокси будет совпадать с идеальным, т.к. сумма метрики $m_{t_{fix}}(a_i, c)$ непосредственного соединения и реальной метрики c, которая по предположению индукции равна суммарной метрике наилучшего маршрута от c до S_k, равна M — суммарной метрике наилучшего маршрута от a_i до S_k.

2. На момент времени $t_{\leq M}$ узел a_i имеет реальную метрику строго больше M. Из леммы 5.1 следует, что множество $L_{<M}$ содержит всех идеальных прокси узла a_i — узлы данного множества нашли своих идеальных прокси, а суммарная метрика их наилучших маршрутов меньше M. Так как по предположению индукции реальные метрики узлов $L_{<M}$ совпадают с метриками наилучших маршрутов, первое же обновление пришедшее узлу a_i от идеального прокси приведет к тому, что сам a_i «найдет» идеального прокси и установит значение реальной метрики равным M.

Осталось показать, что полученные записи о наилучших маршрутах в таблицах маршрутизации узлов L_M никогда не пропадут. Похожее рассуждение мы уже проводили в процессе доказательства леммы 5.3 — теперь это уже следует из результата данной леммы и неравенства (3).

Таким образом, переход индукции окончательно доказан, а из предположения индукции (2) тривиально следует утверждение всей теоремы. □

На текущий момент, абстрагируясь от реализации операции scatter, мы можем утверждать, что наше решение задачи маршрутизации готово и соответствует нашим требованиям в виде гарантии согласованности в конечном счете. В последнем пункте данного подраздела обсудим реализацию операции scatter, чтобы окончательно завершить работу по оптимизации сетевого взаимодействия между узлами разных площадок.

5.3.3 Реализация операции scatter

С того момента, как мы заполнили таблицы маршрутизации всех узлов в пункте 5.3.1, узлы способны строить маршруты для отправки данных друг другу. В пункте

5.3.2 мы упоминали, что для построения оптимальных маршрутов нам понадобится такая абстракция, как операция scatter (определение 5.11). В этом пункте обсудим ее реализацию.

В простейшем случае мы могли бы каждую операцию scatter рассматривать как множество независимых сообщений, отличающихся лишь узлами назначения, и каждое такое сообщение доставлять в соответствии с маршрутизацией отдельно. Однако подобный подход не учитывает следующие факторы: 1) у всех этих сообщений одинаковый набор данных для доставки; 2) у некоторых сообщений маршруты доставки будут иметь одинаковый префикс. Такая реализация привела бы к дополнительным расходам ресурсов на сетевое взаимодействие. Постараемся использовать информацию о построенных маршрутах для более оптимальной реализации операции scatter.

Замечание 5.5. В процессе доставки сообщений некоторые из них могут доходить до узлов назначения в порядке отличном от того, в котором они были отправлены. В теории это может нарушить работу алгоритма: узлы могут получать обновления с устаревшими данными и менять свою таблицу маршрутизации в соответствии с ними. Чтобы защититься от подобных ситуаций, для таблицы маршрутизации каждого узла введем понятие *версии* — монотонного счетчика, который поможет проверять актуальность приходящих обновлений. Значение версии будет увеличиваться после каждой рассылки очередной порции данных, и каждое обновление будет помечаться значением последней версии локальных данных. Каждый из узлов для каждой площадки назначения будет хранить последние полученные от других узлов версии данных. Таким образом, для каждого пришедшего обновления (регулярного или экстренного) узел сможет понять, получал ли он по соответствующей площадке назначения от соответствующего узла-источника более поздние обновления, и если получал, он будет игнорировать только что пришедшее.

Чтобы сделать механизм версионирования отправленных данных устойчивым к отказам оборудования, в качестве «версии» необходимо использовать такой монотонный счетчик, значение которого не уменьшится после перезагрузки узла. Для этого подойдет время отправки очередной порции данных (количество секунд, прошедших с начала эпохи [10]). Действительно, вызовы операции scatter упорядочены по времени, а часы каждого отдельного узла могут выдавать монотонно возрастающие по времени значения.

Коль скоро абстракция в виде операции scatter может понадобиться не только для рассылки данных о динамической маршрутизации, будет разумно разделить все вызовы scatter на подгруппы так, чтобы сообщения каждой подгруппы версионировались по отдельности и не влияли друг на друга. Для этого введем пониятие *топика* для каждого сообщения: сообщения, имеющие один и тот же топик будут версионироваться последовательно (старые сообщения будут игнорироваться), а сообщения из разных

топиков будут версионироваться по отдельности: последнее сообщение из топика `'a'` не может быть проигнорировано из-за последнего сообщения из топика `'b'`, даже если второе имело более позднюю версию.

Для реализации описанного выше механизма версионирования заведем словарь `last_seen_topics_ts`, ключами которого будут являться строковые значения топиков, а значениями — другие словари, хранящие последние полученные версии соответствующего топика от различных узлов (ключи внутренних словарей — идентификаторы узлов, значения — версии в виде временны́х отметок). Изначально словарь `last_seen_topics_ts` пуст. Как именно мы будем его использовать, поясним далее, когда будем описывать *общий механизм* обработки сообщения.

Добавим также новое protobuf-сообщение, экземплярами которого будут обмениваться узлы при выполнении операции scatter. Его описание представлено в листинге 17.

```
message ScatterMessage {
  bytes data = 1;
  int64 ttl = 2;
  string topic = 3;
  uint64 source_node = 4;
  uint64 timestamp = 5;
  repeated uint64 dest_groups = 6;
}
```

Листинг 17 — Protobuf-сообщение для реализации операции scatter.

Поясним значение каждого поля из представленных в нем:

- `data` — данные, которые нужно разослать с помощью операции scatter, в случае с другим protobuf-сообщением — уже в сериализованном виде;
- `ttl` — TTL-поле для борьбы с «гуляющими» сообщениями, возникновение которых мы описали в замечании 5.3;
- `topic` — топик, к которому рассылаемые данные относятся;
- `source_node` — идентификатор узла, инициирующий рассылку, нужен для раздельного версионирования сообщений от разных источников;
- `timestamp` — «версия» рассылаемых данных в топике `topic` от узла `source_node`, представлена в виде временно́й отметки, что отражено в названии поля;
- `dest_groups` — площадки назначения, до которых текущий узел должен передать данные.

Узел, инициирующий рассылку заполняет эти поля соответствующим образом: `source_node` — идентификатор данного узла, `dest_groups` содержит идентификаторы

всех групп из $\mathcal{S}$, `topic` равно строковому значению топика, к которому данные относятся, значение `timestamp` устанавливается равным текущему времени: `int(time.time())`. При этом значение `ttl` может варьироваться, но разумным ограничением сверху может служить общее количество узлов в сети N. Далее полученное в процессе инициализации сообщение `message` начальный узел обрабатывает по *общему механизму*. При получении сообщения `message` типа `ScatterMessage` любой другой узел обрабатывает его также по *общему механизму*:

1. Если значение временно́й метки в словаре `last_seen_topics_ts` по ключам `message.topic` и `message.source_node` больше, чем значение временно́й метки самого сообщения `message.timestamp`, сообщение считается устаревшим и игнорируется, а общий механизм завершает работу. Иначе соответствующее значение в `last_seen_topics_ts` обновляется до значения временно́й метки в пришедшем сообщении (листинг 18).

```
if message.topic in self.__last_seen_topics_ts and \
        message.source_node in self.__last_seen_topics_ts[message.topic] and \
        message.timestamp <= \
        self.__last_seen_topics_ts[message.topic][message.source_node]:
    return
if message.topic not in self.__last_seen_topics_ts:
    self.__last_seen_topics_ts[message.topic] = {}
self.__last_seen_topics_ts[message.topic][message.source_node] = \
    message.timestamp
```

Листинг 18 — Проверка актуальности пришедшего сообщения.

2. Создается словарь `next_hops`, ключами которого будут являться идентификаторы узлов сети, а значениями — экземпляры `ScatterMessage`.

3. Значение поля `message.ttl` уменьшается на 1. Если полученное значение данного поля оказывается неположительным, сообщение обрабатывается текущим узлом, но дальше не передается — на этом общий механизм завершается.

4. Если значение TTL осталось положительным, для каждого идентификатора площадки `dest_group` из `message.dest_groups` у таблицы маршрутизации запрашивается текущий прокси `next_hop`:

   (a) Если узел, на котором происходит обработка сообщения, сам находится на площадке `dest_group`, то он запускает обычную рассылку данных `message.data` внутри своей группы и переходит к следующему значению `dest_group`.

   (b) Если в словаре `next_hops` есть запись с ключом `next_hop`, то в поле `dest_groups` соответствующего значения добавляется идентификатор площадки `dest_group`.

(c) Если в словаре такого ключа нет, то по данному ключу добавляется значение `message` с тем лишь отличием, что поле `dest_groups` в нем заменяется на список, состоящий лишь из `dest_group`.

Итогоый цикл по `message.dest_groups` представлен в листинге 19.

```
for dest_group in message.dest_groups:
    if dest_group == self.__group_id:
        self.__scatter_within_group(message)
    else:
        next_hop = self.__router.get_next_hop(dest_group)
        if next_hop is None:
            continue
        if next_hop in next_hops:
            next_hops[next_hop].dest_groups.append(dest_group)
        else:
            next_hops[next_hop].CopyFrom(message)
            del next_hops[next_hop].dest_groups[:]
            next_hops[next_hop].dest_groups.append(dest_group)
```

Листинг 19 — Цикл формирования сообщений для дальнейшей передачи.

5. Для каждой пары ключ-значение из словаря сформированное сообщение передается напрямую соответствующему узлу, идентификатор которого указан в ключе данной пары.

Нетрудно заметить, что при использовании описанного алгоритма для каждого узла назначения данные проходят тот же самый маршрут, что и при применении наивного подхода. Значит корректность выполнения операции scatter подобным образом сводится к корректности построения маршрутов в сети. В то же время мы снизили количество пересылаемых данных по сравнению с наивным подходом: если хотя бы у двух маршрутов есть общий префикс, состоящий хотя бы из 2 узлов, то вместо передачи двух отдельных сообщений мы сформируем одно и перешлем его дальше — таким образом, снизим 1) сетевой трафик хотя бы на `len(data)` байт; 2) накладные расходы на передачу данных, связанные с сетевым стеком.

Завершая построение маршрутизации сети на этом моменте, можем перейти к описанию непосредственной реализации операции AllReduce в разделе 6.

6 Реализация AllReduce

Напомним, решение каких задач мы должны обеспечить для оптимизации выполнения операции AllReduce с точки зрения суммарной стоимости пересылаемых данных:

1. Определение единственного узла-редуктора на площадке для передачи данных счетчиков, последующей редукции и отправки на внешние площадки;

2. Построение оптимальных по суммарной метрике маршрутов для передачи данных между площадками;

3. Проведение промежуточных редукций в пределах площадки на узлах-редукторах и на каждом узле по отдельности для получения глобального результата.

Решение первого пункта мы предложили в подразделе 4.2: мы научились согласованно в конечном счете выбирать редуктор и резервный редуктор, что далее нам поможет эффективно агрегировать данные о счетчиках на минимальном количестве узлов и считать промежуточные редукции по ним. Также из свойства 3 из определения функции стоимости передачи данных 3.9 и теоремы 5.1 можно сделать вывод о том, что в условиях той же теоремы с некоторого момента суммарная стоимость передачи данных при рассылке результатов промежуточных редукций будет минимальной из возможных.

В данном разделе мы завершим описание реализации операции AllReduce, добавив в общий дизайн несколько этапов редукции и правила, по которым они будут выполняться.

6.1 Выполнение промежуточных редукций

Для начала определимся, как мы будем выполнять операцию редукции и хранить промежуточные результаты на произвольных узлах системы.

Чтобы избежать лишнего потребления памяти на хранение счетчиков каждого узла и результатов промежуточной редукции каждой группы, при получении очередных данных будем пытаться применить редукцию к паре из текущего результата `cur_reduction_values` и только что полученных счетчиков. Таким образом, вместо постоянного поддержания состояния в виде значений всех счетчиков, которые должны в итоге составить общий результат редукции, мы будем накапливать этот результат в процессе сбора этих счетчиков.

Проблема, которую приходится решать при использовании данного подхода, заключается в необходимости поддерживать множество источников, счетчики которых были учтены в текущем результате редукции, чтобы не добавлять в результат одни и те же данные дважды. Такая ситуация может возникать из-за различных интервалов таймеров или в результате задержек сети. Для решения данной проблемы мы будем использовать битовую маску `cur_reduction_set`, в которой i-й бит будет равен 1, если в текущем результате редукции учтены счетчики с узла с идентификатором i, и 0 — иначе. К каждому сообщению, содержащему индивидуальный счетчик или промежуточную редукцию, будем добавлять такую же маску (в случае с индивидуальным счетчиком достаточно идентификатор узла), чтобы узел назначения мог определить множество узлов, счетчики которых были учтены в пришедшем сообщении. Итого, если за `ReductionValues` считать тип значения счетчика из C, то формат protobuf-сообщения

примет вид, представленный в листинге 20.

```
message ReductionResult {
  bytes reduction_set_mask = 1;
  ReductionValues values = 2;
}
```

Листинг 20 — Результат промежуточной редукции в формате protobuf-сообщения.

Дальнейшее добавление промежуточного результата редукции к текущему результату может выполняться двумя диаметрально противоположными способами, каждый из которых определенным образом может привнести неточность в конечный результат:

1. Если побитовое «и» между текущей битовой маской и битовой маской из пришедшего сообщения дает ненулевой результат, т.е. множество учтенных узлов не пересекается с множеством узлов из пришедшего сообщения, то текущий результат редукции пересчитывается в соответствии с содержащимся в сообщении значением, а битовая маска дополняется новыми ненулевыми битами при помощи побитового «или». В ином случае, если множества пересекаются, редукция не производится и пришедшие данные игнорируются. Данный способ может привести к тому, что значения счетчиков некоторых узлов системы не будут учтены в глобальном результате редукции из-за того, что будут отброшены. Однако ни один счетчик не будет добавлен в редукцию дважды.

2. Текущий результат редукции дополняется новыми данными всегда, кроме тех случаев, когда результат побитового «и» масок равен значению маски из пришедшего сообщения, т.е. счетчики, составляющие промежуточный результат из данного сообщения, уже были учтены в текущем результате редукции. Аналогично, после выполнения редукции текущая маска добавляется новыми ненулевыми битами с помощью побитового «или». При использовании данного подхода счетчики некоторых узлов могут быть проредуцированы несколько раз, но при этом данные счетчиков любого пришедшего сообщения будут учтены.

Как можно заметить, каждый из этих подходов лучше работает в тех ситуациях, в которых более грубую ошибку порождает другой. Поэтому мы можем построить компромиссное решение, используя тот или иной способ редукции в более подходящих обстоятельствах. Заведем постоянную `MAX_OVERLAY_RATE`, значение которой будет лежать в отрезке $[0, 1]$. Будем считать, что по умолчанию ее значение равно 0.5. Если нам необходимо применить операцию редукции к некоторому промежуточному результату из пришедшего сообщения, будем считать количество ненулевых битов в побитовом «и» двух масок `cur_reduction_set & other_reduction_set` и сравнивать его с количеством ненулевых битов в маске из пришедшего сообщения, помноженным

на `MAX_OVERLAY_RATE`. Если первое значение не превосходит второго, будем проводить редукцию — в этом случае часть повторно учтенных счетчиков от числа счетчиков, учтенных в пришедешем сообщении, не превзойдет `MAX_OVERLAY_RATE`. Иначе редукцию проводить не будем. В частности, если значение `MAX_OVERLAY_RATE` равно 0, то предложенная реализация соответствует первому пункту, рассмотренному выше, а если оно равно 1, то второму.

Нетрудно заметить, что при такой реализации со временем любой результат редукции устареет, так как новые значения счетчиков будут игнорироваться в любом случае. Поэтому должна быть возможность сбросить текущий результат редукции для накопления нового, более актуального. Чтобы это сделать, достаточно приравнять текущий результат редукции к нейтральному элементу $0 \in C$, а также обнулить все биты в битовой маске — таким образом, приходящие сообщения со счетчиками и промежуточными редукциями вновь начнут учитываться.

Код предложенной реализации операции редукции в виде методов класса `Reducer` представлен в листинге 21. Стоит отметить, что для хранения битовых масок и выполнения битовых операций над ними мы используем python-библиотеку bitarray [15], которая реализована на языке C, что позволяет проводить побитовые операции сравнительно быстро.

6.2 Глобальная редукция

Сперва обсудим применение описанного механизма редукции для получения *глобальной* редукции, т.е. непосредственного результата операции AllReduce, корректности которого мы будем добиваться согласно определению 3.8.

Будем считать, что у нас есть абстракция потребителя результата глобальной редукции, который в любой момент времени может принять ее последнее полученное значение, чтобы использовать в своих целях. В простейшем случае потребитель может сохранять последний результат в памяти до востребования. С помощью дальнейшей реализации глобальной редукции мы будем стараться передавать потребителю наиболее корректный результат редукции, полученный с последнего раза.

Текущее накопленное значение редукции будем хранить в поле `global_result`, являющимся экземпляром класса `Reducer`, основные методы которого мы обсудили в подразделе 6.1. Заведем также повторяющийся таймер длительностью Δ_f времени, каждое срабатывание которого будет инициировать завершение текущей итерации подсчета глобальной редукции: если в результате глобальной редукции учтены счетчики всех узлов системы, при срабатывании таймера происходит передача накопленного результата потребителю и очистка текущего результата. Размер временно́го интервала срабатывания данного таймера напрямую зависит от требований, накладываемых на корректность выполнения AllReduce: чем меньше допустимое временно́е отклонение Δ, тем меньше значение Δ_f.

```
def reduce_with(self, other: ReductionResult) -> None:
    if self.__cur_reduction_values is None:
        self.__cur_reduction_values = self.__cur_reduction_values = \
            self.__reduction_strategy.generate_neutral()

    other_reduction_set = deserialize(other.reduction_set_mask)
    other_reduction_set_size = other_reduction_set.count(1)
    overlay = self.__cur_reduction_set & other_reduction_set
    overlay_size = overlay.count(1)
    if overlay_size <= other_reduction_set_size * self.__MAX_OVERLAY_RATE:
        self.__cur_reduction_set |= other_reduction_set
        self.__total_overlay += overlay_size
        self.__total_reduced += other_reduction_set_size
        self.__cur_reduction_values = self.__reduction_strategy.reduce(
            self.__cur_reduction_values,
            other.values
        )

def clear_reduction_result(self) -> None:
    self.__cur_reduction_set = bitarray(self.__total_nodes)
    self.__cur_reduction_set.setall(0)
    self.__cur_reduction_values = None
    self.__total_reduced = 0
    self.__total_overlay = 0
```

Листинг 21 — Методы класса `Reducer` для добавления результата промежуточной редукции и очищения текущего результата.

Для случая, когда на момент очередного срабатывания таймера в Δ_f времени в маске результата глобальной редукции есть нулевые биты (т.е. в ней учтены счетчики не всех узлов), будем на время Δ_w сохранять полученный результат в отдельную переменную `waiting_reducer`. Подобная ситуация может произойти, если в сети возникли большие задержки, из-за которых некоторые отправленные результаты *локальных* редукций не успели дойти до текущего узла до конца очередной итерации подсчета глобальной редукции. Чтобы частично предотвратить последствия таких ситуаций, при их возникновении все новые приходящие результаты локальных редукций в течение следующего времени Δ_w узел будет добавлять не только в обычное поле `global_result`, но и в `waiting_reducer` тоже. Причем если после очередного выполнения операции редукции окажется, что результат `waiting_reducer` стал полным, то он сразу же передается потребителю и очищается. В ином случае это безусловно происходит после истечения времени Δ_w.

Глобальный результат редукции будет пополняться данными *локальных* редукций — значений, полученных путем агрегации счетчиков узлов в пределах каждой отдельной площадки. Механизм, по которому локальные редукции будут накапливаться и рассылаться опишем в следующих подразделах 6.3 и 6.4. В том же подразделе 6.4

мы упомянем отложенные редукции, которые в некоторых случаях позволят избежать получения некорректного результата.

6.3 Локальная редукция

Результатом *локальной* редукции мы будем называть значение редукции, примененной к счетчикам узлов одной площадки. Для его подсчета мы также заведем отдельный экземпляр класса `Reducer` и таймер длительностью Δ_s времени, причем интервал его срабатывания Δ_s должен быть не больше Δ_f. Идея состоит в том, что узлы будут распространять результаты локальных редукций с частотой не реже той, с которой сохраняется и очищается глобальный результат на всех других узлах. Таким образом, при несущественных изменениях длительности задержек в сети на каждую итерацию подсчета глоабльной редукции будет приходиться не менее одного сообщения с результатом каждой локальной редукции. Однако данный таймер не будет явно запускать механизм рассылки наполненного результата — вместо этого он будет генерировать событие ScatterTimeout (ST), которое будет обрабатываться конечным автоматом состояний, который мы обсудим в подразделе 6.4. Отправка результатов локальных редукций на все узлы системы будет осуществляться с помощью операции scatter, реализацию которой мы обсудили в пункте 5.3.3. Соответственно, для таких сообщений будет использоваться уникальный топик.

Еще один тип событий, который будет обрабатывать конечный автомат — входящее сообщение от узла той же группы со значениями его индивидуальных счетчиков. Будем обозначать его как IndividualValues(id, values) (или IV(id, values)), где id — идентификатор узла отправителя, а values — значения пришедших счетчиков.

Эти сообщения также должен кто-то отправлять. Для этого заведем таймер длительностью Δ_{iv}, интервал срабатывания которого также должен быть не больше Δ_s, чтобы все узлы сети успели отправить данные своих счетчиков для локальной редукции. Нужно лишь определиться, кому эти сообщения будут отправляться. Именно для этого мы строили алгоритм выбора редуктора и резервного редуктора в подразделе 4.2: данные индивидуальных счетчиков каждый узел раз в Δ_{iv} времени будет отправлять тем узлам, которых он на текущий момент считает редуктором и резервным редуктором. Лишь в том случае, если редуктор еще не выбран, что встречается довольно редко, он будет отправлять эти данные самому себе и, соответственно, генерировать событие IndividualValues(id, values) для своего экземпляра конечного автомата.

Последний тип событий будет связан с изменением выбора редуктора и резервного редуктора. Будем обозначать такие события следующим образом: TransitionTo(state), где state — одно из значений состояния, в котором может находится узел в терминах выбора редуктора и резервного редуктора: `REDUCER`, `BACKUP_REDUCER` или `OTHER`. Данные события будет генерировать механизм, описанный в соответствующем подразделе 4.2: после каждого изменения хотя бы одного из полей `reducer` и `backup_reducer` он будет

передавать в конечный автомат событие с соответствующим значением state, которое он может определить однозначно по значениям этих полей (см. листинг 4).

Теперь перейдем непосредственно к описанию состояний автомата и возможных переходов между ними.

6.4 Конечный автомат состояний

Еще раз перечислим события, которые наш конечный автомат будет обрабатывать:

- ScatterTimeout (или ST) — срабатывание таймера, по которому ненулевая локальная редукция должна быть распространена на другие площадки;
- IndividualValues(id, values) (или IV(id, values)) — сообщение от узла текущей площадки, содержащее индивидуальные значения счетчиков values узла с идентификатором id;
- TransitionTo(state) (или TT(state)) — переход в состояние state в терминах алгоритма выбора редуктора или резервного редуктора, state может принимать одно из значений: `REDUCER`, `BACKUP_REDUCER` и `OTHER`.

Всего в конечном автомате будет 5 состояний, каждое из которых будет обрабатывать набор из перечисленных событий уникальным образом. Из них 3 основных, соответствующих одноименным состояниям в алгоритме выбора: ReducerState, BackupState, OtherState и 2 промежуточных: TemporaryReducerState и PreBackupState. Каждое из них реализует интерфейс `State`, представленный на листинге 22.

```
1  class State:
2      def __init__(self) -> None:
3          self.context: tp.Optional[ReducerContext] = None
4
5      def handle_individual_values(self, individual_values: IndividualValues) -> None:
6          pass
7
8      def handle_state_transition(self, to_state: NeighbourState.V) -> None:
9          pass
10
11     def handle_scatter_timeout(self) -> None:
12         pass
```

Листинг 22 — Интерфейс состояния узла с объявленными методами для обработки событий.

Поле `context` здесь — экземпляр класса `ReducerContext`, хранящий экземпляр класса `Reducer` для локальной редукции и таймер, порождающий события ScatterTimeout. Он также предоставляет возможность для рассылки данных с помощью опе-

рации scatter. Более того, класс `ReducerContext` является входной точкой для обработки конечным автоматом событий, приходящих извне, таких как: IndividualValues и TransitionTo.

Наглядная иллюстрация переходов между состояниями в конечном автомате представлена на рисунке 1. Рядом со стрелками переходов между состояниями указаны условия для их совершения и краткие описания эффектов, сопровождаемых соответствующие переходы в формате E + action + ... + action / ... / E + action + ... + action, где E — событие, порождающее данный переход (возможно, с некоторыми условиями), action — один из эффектов, соответствующих данному переходу при возникновении данного события, а каждый набор из события и соответствующих ему действий разделен чертой.

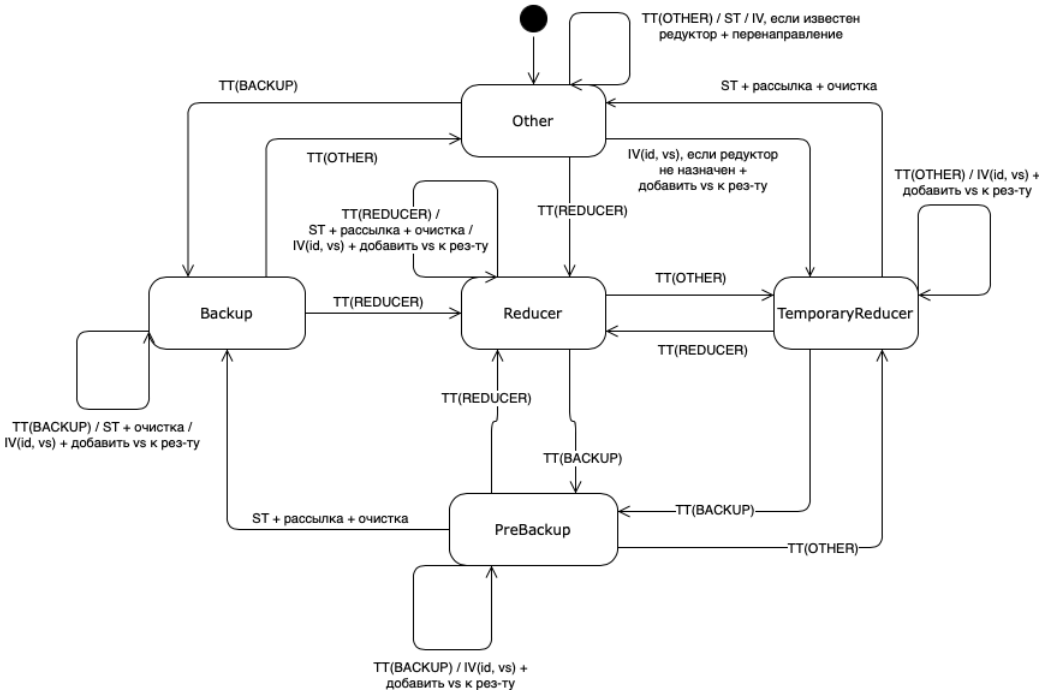

Рисунок 1 — Состояния конечного автомата и переходы в нем.

Разберем по отдельности каждое из упомянутых состояний: какой эффект вызывает обработка различных событий, в какие другие состояния оно может перейти.

6.4.1 Состояние ReducerState

ReducerState — состояние по умолчанию для узла, выбранного в качестве редуктора в пределах группы. Находясь в этом состоянии, узел агрегирует значения приходящих индивидуальных счетчиков от узлов его группы для получения результата локальной редукции, а также регулярно рассылает накопленный результат по узлам всей системы. Приходящие события в этом состоянии обрабатываются следующим образом:

- IndividualValues(id, values): если битовая маска результата на месте id содержит 0,

значения счетчиков values добавляются в текущий результат редукции по обычному механизму, описанному в подразделе 6.1;

- TransitionTo(state): в зависимости от значения state происходит один из переходов:
  - `REDUCER`: событие игнорируется (узел уже находится в нужном состоянии);
  - `BACKUP_REDUCER`: совершается переход в состояние PreBackupState;
  - `OTHER`: совершается переход в состояние TemporaryReducerState;
- ScatterTimeout: текущий накопленный результат локальной редукции рассылается по всем узлам системы с помощью операции scatter, а затем очищается.

Соответствующая реализация интерфейса `State` представлена в листинге 23.

```
class ReducerState(State):
    def __init__(self) -> None:
        super().__init__()

    def handle_individual_values(self, individual_values: IndividualValues) -> None:
        if self.context is not None:
            self.context.local_reducer.add_individual_values(individual_values)

    def handle_state_transition(self, to_state: NeighbourState.V) -> None:
        if to_state == REDUCER or self.context is None:
            return
        elif to_state == BACKUP_REDUCER:
            self.context.transition_to(PreBackupState())
        elif to_state == OTHER:
            self.context.transition_to(TemporaryReducerState())

    def handle_scatter_timeout(self) -> None:
        if self.context is not None:
            self.context.scatter_reduction_result()
            self.context.local_reducer.clear_reduction_result()
```

Листинг 23 — Реализация класса `ReducerState`, соответствующая состоянию ReducerState.

6.4.2 Состояние BackupState

BackupState — состояние по умолчанию для узла, выбранного в качестве резервного редуктора в пределах площадки. Работа узла в данном состоянии аналогична работе в состоянии ReducerState с тем лишь отличием, что при обработке события ScatterTimeout он не рассылает накопленный результат, а просто очищает текущий. Обработка события TransitionTo(state) также отличается:

- Если state равно `BACKUP_REDUCER`, событие игнорируется;

- Если state равно `REDUCER`, то совершается переход в ReducerState;
- Если state равно `OTHER`, накопленный результат редукции очищается, совершается переход в OtherState.

Реализация интерфейса представлена в листинге 24.

```
class BackupState(State):
    def __init__(self) -> None:
        super().__init__()

    def handle_individual_values(self, individual_values: IndividualValues) -> None:
        if self.context is not None:
            self.context.local_reducer.add_individual_values(individual_values)

    def handle_state_transition(self, to_state: NeighbourState.V) -> None:
        if to_state == BACKUP_REDUCER or self.context is None:
            return
        elif to_state == REDUCER:
            self.context.transition_to(ReducerState())
        elif to_state == OTHER:
            self.context.local_reducer.clear_reduction_result()
            self.context.transition_to(OtherState())

    def handle_scatter_timeout(self) -> None:
        if self.context is not None:
            self.context.local_reducer.clear_reduction_result()
```

Листинг 24 — Реализация класса `BackupState`, соответствующая состоянию BackupState.

Данное состояние, как и общая идея резервного редуктора, предназначено для быстрой смены редуктора: с момента отказа последнего до момента явной смены состояния он будет продолжать накапливать значение локальной редукции, поэтому при становлении редуктором ему не придется агрегировать все данные с нуля.

6.4.3 Состояние OtherState

OtherState — состояние по умолчанию узла, не являющегося ни редуктором, ни резервным редуктором. События в нем обрабатываются следующим образом:

- IndividualValues(id, values): если текущий узел считает редуктором один из узлов площадки, полученное сообщение с индивидуальными счетчиками перенаправляется данному узлу, иначе совершается переход в состояние TemporaryReducerState, и то же событие обрабатывается в новом состоянии;
- TransitionTo(state): в зависимости от значения state происходит один из переходов:
  - `OTHER`: событие игнорируется;

– BACKUP_REDUCER: совершается переход в состояние BackupState;

– REDUCER: совершается переход в состояние ReducerState;

- ScatterTimeout: событие игнорируется.

Код реализации данного состояния представлен в листинге 25.

```
class OtherState(State):
    def __init__(self) -> None:
        super().__init__()

    def handle_individual_values(self, individual_values: IndividualValues) -> None:
        if self.context is None:
            return
        if self.context.designator is None or \
                self.context.designator.get_reducer() < 0 or \
                individual_values.ttl <= 0:
            self.context.transition_to(TemporaryReducerState())
            self.context.handle_individual_values(individual_values)
        else:
            individual_values.ttl -= 1
            self.context.local_sender.send_individual_values(
                individual_values,
                self.context.designator.get_reducer()
            )

    def handle_state_transition(self, to_state: NeighbourState.V) -> None:
        if to_state == OTHER or self.context is None:
            return
        elif to_state == REDUCER:
            self.context.transition_to(ReducerState())
        elif to_state == BACKUP_REDUCER:
            self.context.transition_to(BackupState())
```

Листинг 25 — Реализация класса `OtherState`, соответствующая состоянию OtherState.

Стоит отметить, что здесь мы также применяем подход с добавлением к сообщениям TTL-поля, поскольку мы не можем с уверенностью заверять, что циклов при выборе редуктора не образуется.

Как мы помним, при реализации алгоритма выбора редуктора и резервного редуктора мы пожертвовали согласованностью выбора в пользу доступности. Тем не менее, с помощью описанных перенаправлений сообщений и раннего вхождения в состояние TemporaryReducerState мы можем компенсировать несогласованный выбор. Последние два состояния, которые мы опишем в пунктах 6.4.4 и 6.4.5 также предназначены для того, чтобы бороться с проблемой несогласованного выбора редуктора.

6.4.4 Состояние TemporaryReducerState

TemporaryReducerState — состояние узла, который уже (или еще) не является редуктором, но у которого есть некоторый результат редукции, который может быть необходим для формирования корректного результата глобальной редукции. В это состояние может переходить редуктор с меньшим идентификатором узла при возникновении конфликтов или узел, который еще не успел узнать, кто на площадке является редуктором. При этом мы считаем, что данное состояние временное: коль скоро узел сам не является редуктором, однажды другие узлы перестанут считать его таковым, поэтому при следующем срабатывании таймера, которое породит событие ScatterTimeout, данный узел инициирует рассылку своего результата редукции, а затем перейдет в стандартное состояние OtherState. Итак, опишем его действия при обработке всевозможных событий:

- IndividualValues(id, values): индивидуальные счетчики добавляются к текущему результату редукции по обычному механизму, описанному в подразделе 6.1;
- TransitionTo(state): в зависимости от значения state происходит один из переходов:
  - `OTHER`: событие игнорируется;
  - `BACKUP_REDUCER`: совершается переход в состояние PreBackupState;
  - `REDUCER`: совершается переход в состояние ReducerState;
- ScatterTimeout: текущий накопленный результат локальной редукции рассылается по всем узлам системы с помощью операции scatter, а затем очищается, совершается переход в состояние OtherState.

На листинге 26 представлен код реализации состояния TemporaryReducerState.

6.4.5 Состояние PreBackupState

Состояние PreBackupState аналогично состоянию TemporaryReducerState с теми лишь отличиями, что по истечениии таймера переход совершается в состояние BackupState, и переходы при обработке событий TransitionTo отличаются ввиду того, что узел, находящийся в данном состоянии является резервным редуктором на площадке в терминах алгоритма выбора из подраздела 4.2. Для наглядности опустим описание действий при обработке каждого события, а лишь приложим код реализации (листинг 27).

Описав работу конечного автомата, а также механизм, по которому выполняются локальные и глобальные редукции, мы завершили построение общего алгоритма выполнения операции AllReduce. Для того, чтобы мы могли судить о качестве предложенного решения, опишем и докажем его свойства в следующем разделе 7.

```python
class TemporaryReducerState(State):
    def __init__(self) -> None:
        super().__init__()

    def handle_individual_values(self, individual_values: IndividualValues) -> None:
        if self.context is not None:
            self.context.local_reducer.add_individual_values(individual_values)

    def handle_state_transition(self, to_state: NeighbourState.V) -> None:
        if to_state == OTHER or self.context is None:
            return
        elif to_state == REDUCER:
            self.context.transition_to(ReducerState())
        elif to_state == BACKUP_REDUCER:
            self.context.transition_to(PreBackupState())

    def handle_scatter_timeout(self) -> None:
        if self.context is not None:
            self.context.scatter_reduction_result(True)
            self.context.local_reducer.clear_reduction_result()
            self.context.transition_to(OtherState())
```

Листинг 26 — Реализация класса `TemporaryReducerState`, соответствующая состоянию TemporaryReducerState.

7 Корректность реализации AllReduce

7.1 Гарантии построенного алгоритма

В данном подразделе опишем и докажем гарантии на корректность выполнения операции AllReduce, которые удалось достичь, построив предложенный дизайн системы.

Теорема 7.1. *Пусть с момента времени $t_0 \in T$ выбор редукторов на каждой из площадок согласован, узлы не отказывают, все сообщения в сети доставляются, а время задержек при выполнении операции scatter составляет не более Δ_w, т.е. рассылаемые данные дойдут до всех узлов системы не позднее, чем через Δ_w времени после инициирования рассылки. При этом будем считать, что интервалы таймеров Δ_{iv}, Δ_s, а также задержки на передачу данных между узлами одной площадки таковы, что за любой промежуток времени Δ_s редуктор успевает получить данные индивидуальных счетчиков каждого другого рабочего узла площадки. Тогда переданный потребителю глобальный результат редукции на любом узле в момент времени $t > t_0 + \Delta_f + 2\Delta_w + \Delta_s$ будет являться корректным результатом выполнения операции AllReduce с допустимым временны́м отклонением в $\Delta_f + 2\Delta_w + \Delta_s$.*

Доказательство. Из условия теоремы следует, что любой результат локальной редук-

```python
class PreBackupState(State):
    def __init__(self) -> None:
        super().__init__()

    def handle_individual_values(self, individual_values: IndividualValues) -> None:
        if self.context is not None:
            self.context.local_reducer.add_individual_values(individual_values)

    def handle_state_transition(self, to_state: NeighbourState.V) -> None:
        if to_state == BACKUP_REDUCER or self.context is None:
            return
        elif to_state == REDUCER:
            self.context.transition_to(ReducerState())
        elif to_state == OTHER:
            self.context.transition_to(TemporaryReducerState())

    def handle_scatter_timeout(self) -> None:
        if self.context is not None:
            self.context.scatter_reduction_result(True)
            self.context.local_reducer.clear_reduction_result()
            self.context.transition_to(BackupState())
```

Листинг 27 — Реализация класса `PreBackupState`, соответствующая состоянию PreBackupState.

ции при рассылке будет содержать данные всех работающих узлов на площадке за последний интервал времени в Δ_s. Это в свою очередь означает, что при истечении таймера в Δ_f времени результат глобальной редукции на узле может быть не передан потребителю из-за неполной битовой маски лишь в 2 случаях: 1) некоторый узел отказал, поэтому локальная редукция с его площадки не содержала соответствующий единичный бит в маске; 2) из-за задержек сети за последний отрезок времени в Δ_f результат локальной редукции с некоторой группы еще не успел дойти до текущего узла. Если все узлы системы находились в рабочем состоянии, возможен лишь второй случай, поскольку интервал рассылки результата локальной редукции Δ_s не больше интервала Δ_f, после которого результат глобальной редукции проверяется на полноту, а значит за время последней итерации подсчета глобальной редукции результаты локальных редукций всех групп были отправлены хотя бы один раз, и, коль скоро все сообщения по условию доставляются, они еще не были доставлены лишь из-за возникших задержек.

Итак, передача результата глобальной редукции потребителю в момент времени t могла произойти в 3 случаях. Разберем их далее.

1. Передача потребителю произошла после истечения таймера в Δ_f времени. Это стандартная ситуация, в этом случае битовая маска состоит лишь из единичных битов, а значит результат содержит данные счетчиков всех узлов системы, при этом ни один из них не был учтен дважды — из рассуждений выше результаты

локальных редукций имели одинаковые битовые маски, а значит ни один из них не мог добавиться в глобальную редукцию дважды. Мы также знаем, что в момент времени $t - \Delta_f$ результат глобальной редукции на узле был очищен, а значит в итоговый результат могли попасть лишь те значения локальных редукций, которые были отправлены не раньше $t - \Delta_f - \Delta_w$ — из условия на время задержек. Накопление результата локальной редукции длится Δ_s времени. Таким образом, результат глобальной редукции, переданный потребителю в момент времени t, содержал в себе значения счетчиков всех узлов, каждый из которых присутствовал на соответствующем узле не более $\Delta_f + \Delta_w + \Delta_s$ времени назад относительно t. В данном случае теорема доказана.

2. Передача потребителю произошла из отложенного результата глобальной редукции `waiting_result` после того, как было получено очередное сообщение с результатом локальной редукции и битовая маска отложенного результата стала состоять из единичных битов. Это могло произойти не позже, чем через Δ_w времени после очередного срабатывания таймера в Δ_f времени — иначе текущий отложенный результат передался бы потребителю даже без полной битовой маски. Поэтому, проведя рассуждения аналогичные предыдущему пункту, мы можем заключить, что такой отложенный результат будет содержать значения счетчиков, присутствовавших на узлах не раньше момента времени $t - \Delta_f - 2\Delta_w - \Delta_s$. Аналогично, теорема доказана.

3. Передача потребителю отложенного результата глобальный редукции произошла по истечению таймера в Δ_w времени. Однако из проведенных ранее рассуждений такого не могло произойти: на момент срабатывания таймера в Δ_f времени все результаты локальных редукций были уже либо учтены в глобальной редукции, либо отправлены, а, коль скоро задрежки при рассылке не превосходят Δ_w, до срабатывания соответствующего таймера данные должны были дойти и учесться в отложенном результате.

Таким образом, во всех возможных случаях передачи результата глобальной редукции потребителю условия на его корректность из формулировки теоремы выполняются, поэтому теорему можно считать доказанной. □

Следствие 7.2. *Пусть с момента времени $t \in T$ узлы системы не отказывали. Тогда при соблюдении ограничений на качество сети из леммы 4.1 и теоремы 7.1 наступит момент времени $t_1 > t$ такой, что начиная с него результаты выполнения операции AllReduce будут корректны с допустимым временны́м отклонением в $\Delta_f + 2\Delta_w + \Delta_s$, причем $t_1 - t \leq \Delta_f + 2\Delta_w + \Delta_s + \Delta_{delay} + 9\Delta_{dead}$.*

Доказательство. По теореме 4.2 в течение времени $\Delta_{delay} + 9\Delta_{dead}$ после t на каждой площадке редуктор будет выбран согласовано. Значит к моменту времени $t + \Delta_{delay} +$

$9\Delta_{dead}$ можно применить утверждение теоремы 7.1. В итоге получим, что начиная с момента времени $t + \Delta_f + 2\Delta_w + \Delta_s + \Delta_{delay} + 9\Delta_{dead}$ гарантировано все результаты выполнения операции AllReduce будут корректны с допустимым временны́м отклонением в $\Delta_f + 2\Delta_w + \Delta_s$. □

Следствие 7.3. *Пусть непосредственно до момента времени $t \in T$ в пределах каждой площадки выбор редуктора и резервного редуктора был согласован, а в момент времени t на одной или нескольких площадках назначенный редуктор отказал. Пусть при этом после момента времени t узлы не отказывали, и выполнялись все условия на качество сети из теорем 4.2 и 7.1. Тогда любой результат глобальной редукции, переданный потребителю в момент времени $t_1 > t+\Delta_{delay}+2\Delta_{dead}+\Delta_f+2\Delta_w+\Delta_s$, будет являться корректным результатом выполнения операции AllReduce с допустимым временны́м отклонением $\Delta_f + 2\Delta_w + \Delta_s$.*

Доказательство. Коль скоро выбор редуктора и резервного редуктора в пределах каждой площадки до момента t был согласован, то по теореме 4.2 в течение следующего отрезка в $\Delta_{delay}+2\Delta_{dead}$ времени на каждой площадке, в которой отказали назначенные редукторы, в качестве нового редуктора будет выбран резервный редуктор. Поэтому мы можем применить утверждение теоремы 7.1 к моменту времени $t + \Delta_{delay} + 2\Delta_{dead}$ и получить требуемое. □

7.2 Методы обеспечения наилучшей доставки

В ряде утверждений, сформулированных в подразделе 7.1, в качестве необходимых условий мы накладывали ограничения на качество сети. В частности, мы требовали, чтобы доставка сообщений до конечных узлов при выполнении операции scatter происходила за ограниченное время. Однако, как мы уже упоминали в подразделе 3.2, в реальных сетях такой гарантии добиться невозможно [5].

Причины, по которым сообщение может не доходить до узла назначения, могут быть различны: высокие задержки могут возникать в случаях перегрузок сети, пакеты могут теряться из-за сбоев промежуточного сетевого оборудования или переполнения буферов. Тем не менее, существуют приемы, позволяющие бороться с такими проблемами. Среди данных подходов можно выделить регулировку скорости отправки пакетов для избежания перегрузок и сообщения-подтверждения о доставке для избежания потерь произвольных пакетов. В действительности оба приема используются в сетевом протоколе транспортного уровня TCP [16, 17]. Таким образом, если мы будем использовать данный протокол для общения узла с его прокси в реализации операции scatter, мы сможем повысить уровень надежности доставки данных, дополнительно не перегружая при этом сетевую инфраструктуру.

Однако даже при отправке данных по TCP-соединению могут возникать ошибки, поэтому обеспечивать надежность доставки сообщений необходимо и на более высоком

уровне. В случае с операцией scatter это также можно реализовать с помощью подтверждений от получателя. В частности, узел может регулярно посылать данные каждому из своих прокси, пока каждый из них не подтвердит их получение. После этого момента текущий узел может считать, что данные успешно доставлены до площадок назначения, если его прокси будет работать по тому же алгоритму. Тем не менее, мы оставим детальное решение данной проблемы за рамками работы.

Итого, сформулировав и доказав ранее перечисленные свойства алгоритма, мы можем утверждать, что построили решение, удовлетворяющее поставленным требованиям: при его использовании операция AllReduce завершается корректно, при этом вся система способна переживать отказы и за ограниченное время возвращаться к штатному режиму работы. Можем переходить к перечислению результатов работы и выводам.

Заключение

Результаты

В процессе работы над исходной задачей мы подробно изучили ее проблематику и сумели выработать подходящее решение, которое удовлетворяет поставленным требованиям. Мы также получили ряд полезных результатов по теме работы:

- рассмотрели различные подходы к реализации операции AllReduce, в том числе с применением готовых инструментов, таких, как: интерфейс MPI и согласованное в конечном счете хранилище;
- выработали концепцию решения, использующую идею частичных редукций, которая позволила снизить потребляемые ресурсы системы на хранение и обмен данными счетчиков;
- реализовали алгоритм выбора узлов для промежуточных редукций и доказали, что выбор становится согласованным за ограниченный отрезок времени;
- построили протокол маршрутизации на уровне узлов системы, позволяющий передавать данные по эффективным с точки зрения суммарной метрики маршрутам, а также доказали, что алгоритм маршрутизации в конечном счете строит наилучшие маршруты;
- описали алгоритм выполнения операции AllReduce, использующий метод проведения частичных редукций на узлах-редукторах и конечный автомат состояний, определяющий эффект от событий различного типа;
- доказали, что при соблюдении определенных условий на качество сети предложенная реализация AllReduce корректна, а также то, что при единичных отказах оборудования алгоритм продолжает работать и за ограниченное время снова начинает выдавать корректный результат редукции;

- реализовали основные части системы в виде прототипа на языке Python (приложение А).

Выводы

В данной работе была рассмотрена операция AllReduce и ее реализация в рамках системы, обладающей специфичной топологией сети, которой зачастую обладают инфраструктуры компаний, предоставляющих георасределенные облачные сервисы. Нам удалось предложить более эффективное решение по сравнению с наивным подходом. В основе данного решения лежат две основных оптимизации: подсчет промежуточных редукций и построение наилучших маршрутов в сети. Мы также показали, что алгоритм продолжает свою работу при отказах узлов. Таким образом, нам удалось достичь поставленной цели: мы снизили стоимость передачи данных между узлами системы и обеспечили определенный уровень отказоустойчивости.

Дальнейшая работа

Несмотря на большой объем проделанной работы, тема исследования содержит ряд особенностей для дальнейшего изучения.

Во-первых, стоит отметить, что бóльшая часть данной работы была посвящена теоретической стороне вопроса, а также проектированию общего дизайна системы. Поэтому для того, чтобы полностью убедиться в работоспособности описанной системы, в будущем предстоит провести тестирование в лабораторных условиях, а также на данных, приближенных к реальным. При этом прототип системы, реализованный нами на языке Python (приложение А), может послужить отправной точкой для дальнейшей разработки готового продукта.

Во-вторых, говоря о внедрении проекта в реальную инфраструктуру той или иной компании, не стоит упускать такую деталь реализации, как конфигурация системы. В данной работе мы абстрагировались от задачи введения новых узлов сети в работу и вывода из нее. Мы считали, что все узлы друг про друга знают: в простейшем случае это настраивается статически, а для более продвинутого варианта необходимо разрабатывать отдельный протокол, по которому они будут получать информацию о соседях динамически. Тем не менее, при реализации готовой системы этот аспект должен быть учтен.

В конце концов, проблема обеспечения наилучшей доставки при выполнении операции scatter также может быть исследована дополнительно. В подразделе 7.2 мы рассмотрели несколько подходов к решению данной задачи — в дальнейшем ими можно воспользоваться при реализации готовой системы.

Список литературы

[1] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[2] Message Passing Interface Forum. Mpi: A message-passing interface standard. version 4.0. Technical report, Message Passing Interface Forum, 2021.

[3] National Telecommunications and Colorado Information Administration's Institute for Telecommunication Sciences in Boulder. Definition: round-trip delay time. `https://www.its.bldrdoc.gov/fs-1037/dir-031/_4641.htm`, 2021.

[4] S. Bradner. Benchmarking terminology for network interconnection devices. RFC 1242, RFC Editor, Июль 1991.

[5] Олифер Н. А. Олифер В. Г. *Компьютерные сети. Принципы, технологии, протоколы: Учебник для вузов. 4-е изд.* СПб.: Питер, 2010.

[6] John Moy. Ospf version 2. STD 54, RFC Editor, Апрель 1998. `http://www.rfc-editor.org/rfc/rfc2328.txt`.

[7] John Ousterhouty Diego Ongaro. In search of an understandable consensus algorithm. Technical report, Stanford University, Июнь 2014.

[8] Dongyan Huang, Xiaoli Ma, and Shengli Zhang. Performance analysis of the raft consensus algorithm for private blockchains, 2018.

[9] Protocol buffers. `https://developers.google.com/protocol-buffers`. Просмотрено: 17.03.2021.

[10] IEEE and The Open Group. The open group base specifications issue 7, 2018. `https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_16`. Просмотрено: 19.06.2021.

[11] Nancy A. Lynch Seth Gilbert. Perspectives on the cap theorem, 2012.

[12] Gary Scott Malkin. Rip version 2. STD 56, RFC Editor, Ноябрь 1998. `http://www.rfc-editor.org/rfc/rfc2453.txt`.

[13] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, Январь 2009.

[14] Jon Postel. Internet protocol. STD 5, RFC Editor, Сентябрь 1981. `http://www.rfc-editor.org/rfc/rfc791.txt`.

[15] Ilan Schnell. bitarray. `https://github.com/ilanschnell/bitarray`, 2008.

[16] Jon Postel. Transmission control protocol. STD 7, RFC Editor, Сентябрь 1981. http://www.rfc-editor.org/rfc/rfc793.txt.

[17] M. Allman, V. Paxson, and E. Blanton. Tcp congestion control. RFC 5681, RFC Editor, Сентябрь 2009. http://www.rfc-editor.org/rfc/rfc5681.txt.

ПРИЛОЖЕНИЕ

Ссылка на репозиторий с кодом проекта

https://github.com/regu1yar/qnet-bachelor-thesis