1DOI: 10.15514/ISPRAS-2026-38(3)-1

К организации системы запросов в распределённой сети с кластерами-кликами

[1] *И.Б. Бурдонов, ORCID: 0000-0001-9539-7853 <igor@ispras.ru>*
[1,2] *Н.В. Евтушенко, ORCID: 0000-0002-4006-1161 <evtushenko@ispras.ru>*
[1] *А.С. Косачев, ORCID: 0000-0001-5316-3813 <kos@ispras.ru>*
[1] *В.Н. Пономаренко, ORCID: 0009-0002-2387-2760 <vera@ispras.ru>*

[1] *Институт системного программирования РАН им. В.П. Иванникова, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25.*

[2] *Национальный исследовательский университет «Высшая школа экономики», 101000, Россия, г. Москва, ул. Мясницкая, д. 20.*

Аннотация. В статье предложена модель кластеризованной распределённой сети для обработки запросов. В такой сети узлы являются вычислительными единицами, и каждый узел может запросить ту или иную услугу, которая может быть оказана каким-то другим, но заранее неизвестным, узлом сети. Для этого осуществляется опрос сети аналогичный поиску в ширину для того, чтобы найти ближайший узел с необходимыми свойствами. При этом обеспечивается достижимость узлов, отсутствие дублирования сообщений, а обмен сообщениями с найденным узлом идёт по кратчайшему пути. Модель нацелена на предотвращение перегрузки сети и ориентирована на минимизацию как времени выполнения опроса, так и размера требуемой для этого памяти в узлах и размеров сообщений. Кластеризованность распределённой сети означает, что сеть состоит из множества кластеров, каждый из которых есть подмножество узлов сети и соединяющих их рёбер. Рёбра графа, которые не принадлежат никакому кластеру, не используются для передачи сообщений, а переход из кластера в кластер осуществляется через общие узлы кластеров. Для того чтобы не было дублирования сообщений, требуется, чтобы граф кластеров был деревом. В данной статье показано, что необходимым требованиям удовлетворяет связный блоковый граф, то есть связный граф, блоки (компоненты двусвязности) которого являются кликами. Эти блоки и выбираются в качестве кластеров.

Ключевые слова: распределенная кластеризованная сеть; опрос сети; достижимость узлов; дублирование сообщений; кратчайший путь; блоковый граф.

Для цитирования: Бурдонов И.Б., Евтушенко Н.В., Косачев А.С., Пономаренко В.Н. К организации системы запросов в распределенной сети с кластерами-кликами. Труды ИСП РАН, том 38, вып. 3, часть 1, 2026 г., стр. 7–32. DOI: 10.15514/ISPRAS–2026–38(3)–1.

On the Query System Organization of a Distributed Network with Clique Clusters

[1] *I.B. Burdonov, ORCID: 0000-0001-9539-7853 <igor@ispras.ru>*
[1,2] *N.V. Yevtushenko, ORCID: 0000-0002-4006-1161 <evtushenko@ispras.ru>*
[1] *A.S. Kossatchev, ORCID: 0000-0001-5316-3813 <kos@ispras.ru>*
[1] *V. N. Ponomarenko, ORCID: 0009-0002-2387-2760 <vera@ispras.ru>*

[1] *Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.*

[2] *National Research University Higher School of Economics, 20, Myasnitskaya st., Moscow, 101000, Russia.*

Abstract. In this paper, a model of a service-oriented architecture is proposed as a clustered distributed network. In such a network, a node can request a service that can be provided by another, but a priori unknown, network node. The process for finding the closest node is organized similar to the breadth-first search. A proposed search ensures the node reachability, allows to avoid the message duplication, and ensures that the shortest path is utilized for the message exchange. The model is aimed at preventing the network overloading and minimizes both time processing and the required memory size. A clustered distributed network has multiple clusters where each cluster is a subset of network nodes and their connecting edges. Network edges that do not belong to any cluster are not used for message transmission, and messages move from cluster to cluster through common nodes. In order to avoid message duplication, the cluster graph is organized as a tree. In the paper, it is shown that the above requirements are satisfied by a connected block graph, i.e., a connected graph whose blocks (biconnected components) are cliques, and these blocks are then selected as clusters of the network.

Keywords: distributed clustered network; service-oriented architecture; node reachability; message duplication; shortest path; block graph.

For citation: Burdonov I.B., Yevtushenko N.V., Kossatchev A.S., Ponomarenko V. N. On the query system organization of a clustered distributed network. Trudy ISP RAN/Proc. ISP RAS, vol. 38, issue 3, part 1, 2026, pp. 7-32 (in Russian). DOI: 10.15514/ISPRAS-2026-38(3)-1.

1. Введение

Рассматривается распределённая сеть, моделируемая связным неориентированным графом без кратных рёбер и петель. Как принято в распределённых сетях, вершины графа будем называть узлами. Узлы соответствуют вычислительным единицам, которые обмениваются между собой сообщениями, посылаемыми по рёбрам графа. Этот граф называется *графом связей*; по умолчанию под графом будет пониматься граф связей.

Сообщение в сети можно разделить на два вида: запрос – ***Request*** и ответ (на запрос) – ***Result***. Сообщение ***Request*** генерируется узлом, который будем называть *начальным узлом*, двигается по некоторому пути в графе и принимается на обработку узлом, который будем называть *конечным узлом*. Сообщение ***Result*** двигается в обратном направлении от конечного узла к начальному узлу. Запрос можно понимать как требование выполнить некие вычислительные действия с параметрами, содержащимися в сообщении ***Request***. Такое сообщение может быть предназначено не какому-то заранее известному узлу, а «кому-нибудь», кто может такой запрос обработать. Это аналогично локальному вызову той или иной процедуры внутри одного узла с той существенной разницей, что вызывающая и вызываемая процедуры находятся в разных узлах сети, и, вообще говоря, всё равно в каком узле находится вызываемая процедура, лишь бы она там была. Коммутация сообщений должна позволять найти такой конечный узел. Запрос предлагается моделировать понятием услуги. В сообщении ***Request*** указывается уникальное имя услуги и набор параметров, а каждый узел сети «знает», какие услуги он может оказывать. Ответное сообщение ***Result*** посылается уже известному начальному узлу.

Каким образом сообщение запроса услуги находит конечный узел? В нашей работе [1] рассматривалась модель распределённой сети, в которой строился ориентированный цикл, проходящий по всем узлам, которые могут оказывать те или иные услуги. Сообщение запроса услуги двигалось по этому циклу, и каждый узел, принимая такое сообщение, определял, может ли он оказать запрашиваемую услугу или не может. Если не может, сообщение пересылалось дальше по циклу. Если может, узел оказывал услугу и посылал ответ, который двигался по тому же циклу, но его параметром было уже не имя запрашиваемой услуги, а идентификатор начального узла. Такая модель позволяла эффективно решить ряд проблем распределённой сети: недостижимость хостов, зацикливание сообщений, перегрузка сети сообщениями (из-за их клонирования), немасштабируемость.

В то же время у этой модели имелся существенный недостаток: путь, который должно пройти сообщение запроса, прежде чем найдёт нужный конечный узел, может оказаться слишком длинным. Попытка решить эту проблему была предпринята в нашей работе [2], где предлагалась кластеризация услуг: множество услуг разбивалось на классы. Вместо одного большого цикла для всех узлов, строилось множество более мелких циклов, по одному на каждый класс услуг. Однако такое решение было лишь частичным: сообщение сначала двигалось по простому пути до цикла нужного класса услуг, а потом по этому циклу, проходя, вообще говоря, далеко не кратчайший путь. Другим общим недостатком обоих вариантов такой модели было то, что она требовала специальной настройки сети для построения цикла в [1] или нескольких циклов по классам услуг и путей, ведущих в тот или иной цикл из узла, не лежащего на цикле в [2]. Также при изменении распределения услуг по узлам сети и (в [2]) кластеризации услуг требовалась перенастройка сети, хотя бы для оптимизации работы сети. Кроме того, при изменении топологии сети усложнялась перенастройка сети (добавление и удаление рёбер и узлов).

В данной работе предлагается другой подход: для того чтобы начальный узел знал, какому конечному узлу посылать запрос услуги, предлагается использовать служебные сообщения опроса. Сначала начальный узел опрашивает сеть, чтобы узнать, какой узел может оказать требуемую услугу, выбирает такой узел в качестве конечного и затем посылает ему сообщение запроса этой услуги (***Request***) по найденному кратчайшему пути. После оказания услуги ответ на запрос (***Result***) двигается по сети по тому же кратчайшему пути, по которому двигалось сообщение ***Request***, но в обратном направлении. В такой модели никакой предварительной настройки сети, связанной с распределением услуг по узлам сети и кластеризацией услуг, не требуется. Нужна только настройка сети для задания её топологии и, соответственно, перенастройка при изменении топологии.

Возникает вопрос: а чем служебные сообщения опроса «лучше» основных сообщений запроса услуги (***Request***) и ответа на запрос (***Result***)? Очевидно ведь, что такой опрос потребует множества пересылок служебных сообщений по сети. Будет ли это компенсироваться тем, что основные сообщения ***Request*** и ***Result*** будут двигаться по кратчайшим путям? Ответ на эти вопросы – в длине сообщений. Предлагается модель, в которой служебные сообщения опроса – это *короткие* сообщения, тогда как основные сообщения могут быть сколько угодно *длинными* за счёт того, что они содержат неограниченные по размеру параметры запроса услуги или ответные параметры, описывающие результат оказания услуги. Кроме того, в предлагаемой модели не происходит дублирования сообщений, когда узел получает одно и то же сообщение несколько раз по дублирующим путям.

Такая сеть в некотором смысле является «идеальной» и не всегда реализуемой по различным причинам. Однако проектируемую (или исследуемую) распределённую сеть можно «проверять» на эффективность, оценивая, насколько её процедуры близки к предлагаемой структуре.

Структура статьи следующая. В разделе 2 формулируются четыре условия, которым должна удовлетворять эффективная работа в сети с опросом, которые, в частности, включают малый диаметр графа. Формулируются пять требований к графу связей, позволяющие выполнить эти условия. Формулируются и доказываются необходимые и достаточные условия достижимости узлов сети и отсутствия дублирования сообщений. Показывается, что граф удовлетворяет указанным пяти требованиям тогда и только тогда, когда он имеет суграф, являющийся блоковым графом. Доказывается формальное утверждение о том, что любой связный граф имеет такой суграф, но в то же время отмечается, что не на любом графе выполняется условие малого диаметра графа. В разделе 3 описывается модель обмена сообщениями при опросе, позволяющая найти кратчайший путь, по которому будут двигаться сообщение запроса услуги (***Request***) и ответное сообщение с результатами оказания услуги (***Result***). Обсуждаются различные варианты такой модели, определяемые теми или иными ответами на восемь возникающих при этом вопросов, и обосновывается выбор предлагаемого решения. Описываются правила коммутации в узлах сети. Приводится таблица примерных значений размеров сообщений и динамических переменных в узлах сети. В Приложении приведены предлагаемые алгоритмы.

2. Требования к топологии графа связей

Рассмотрим предлагаемую модель более детально. Опрос означает, что начальный узел «спрашивает» другие узлы, могут ли они оказать требуемую услугу, рассылая им сообщение ***Question***, содержащее имя услуги. После того, как при опросе конечный узел будет найден, то есть сообщение ***Question*** пройдёт путь *path* от начального узла до конечного узла, конечный узел посылает короткое служебное сообщение ***Yes*** как ответ на сообщение ***Question***. Сообщение ***Yes*** двигается по пути *path* в обратном направлении от конечного узла к начальному узлу. Начальный узел, получив сообщение ***Yes***, посылает по тому же пути *path* в прямом направлении длинное сообщение ***Request***. Наконец, конечный узел, получив сообщение ***Request***, оказывает требуемую услугу, а затем посылает сообщение ***Result*** по тому же пути *path* в обратном направлении до начального узла. Схема обмена этими сообщениями показана на рис. 1.

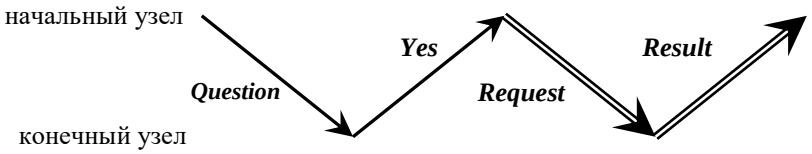

Рис. 1. Короткие (–) служебные и длинные (=) основные сообщения.
Fig. 1. Short (–) service and long (=) main messages.

При таком опросе сети важно соблюдение следующих четырёх условий:

1) Достижимость узлов: каждый узел в принципе может получить сообщение ***Question***, поскольку возможно, что только этот узел может оказать требуемую услугу.
2) Отсутствие дублирования: в данном опросе узел должен получить только одно сообщение ***Question***, во-первых, для того, чтобы узлу не требовалось идентифицировать повторные сообщения с целью их отбраковки, а, во-вторых, чтобы такие повторные сообщения не перегружали сеть.

3) Предотвращение перегрузки сети и кратчайшие пути: опрашивать нужно сначала узлы ближайшие к начальному узлу, и только в том случае, когда конечный узел не найден, расширять зону опроса. Когда конечный узел найден, опрос нужно прекратить, как только это станет возможным. Тем самым, будут найден кратчайший путь, по которому пойдут длинные сообщения ***Request*** и ***Result***. Длина таких путей должна быть достаточно мала. Последнее означает, что диаметр графа должен быть достаточно мал.

Графы связей, в которых возможно выполнение этих условий, близки графам «мир тесен» [3-4]. Название происходит от гипотезы о «шести рукопожатиях» [5-6]: любые два человека на Земле разделены не более чем пятью уровнями общих знакомых и, соответственно, шестью уровнями связей. На математическом языке это означает, что диаметр графа знакомств не превышает 6. В графе «мир тесен» два произвольных узла с большой вероятностью не являются смежными, однако соединены достаточно коротким путём. Графы «мир тесен», во-первых, имеют тенденцию содержать в себе кластеры (подграфы), которые являются кликами (полными подграфами) или почти-кликами. Во-вторых, в графе «мир тесен» расстояние от одного узла до другого в среднем относительно мало. Свойства графа «мир тесен» были обнаружены не только в социальных сетях, но также во многих объектах реального мира, включая карты дорог, пищевые цепочки, электрические сети, сети протекания метаболизма, нейронные сети, сеть телефонных звонков и тому подобным.

Для того чтобы в первую очередь искать конечный узел среди узлов более близких к начальному узлу, опрос происходит в несколько этапов, которые будем называть *раундами*. На раунде 1 начальный узел опрашивает все узлы на расстоянии 1, то есть узлы, с каждым из которых он связан ребром, посылая им сообщение ***Question***. Если требуемый узел не найден, выполняется раунд 2: опрашиваются все узлы на расстоянии 2. И так далее. Если на раунде r требуемый узел не найден, на раунде $r + 1$ опрашиваются узлы на расстоянии $r + 1$.

Обозначим начальный узел через v_0, а узел, опрашиваемый на раунде r, через v_r. На раунде $r > 1$ узел v_{r-1} должен опросить соседние (соединённые с ним ребром) узлы, находящиеся на расстоянии r от узла v_0. Однако среди узлов соседних с узлом v_{r-1} могут быть узлы, находящиеся от узла v_0 на расстоянии меньшим, чем r. Возникает вопрос: каким образом узел v_{r-1} определит среди них узлы, находящиеся на расстоянии r от начального узла v_0?

Для разрешения этого вопроса воспользуемся кластеризацией графа. *Кластерами* будем называть подмножества узлов графа, ни одно из которых не вложено в другое. Там, где это не приводит к двусмысленности, мы не будем различать подмножество узлов и порождённый им подграф (образованный узлами этого подмножества и всеми рёбрами графа, соединяющими пары этих узлов). Будем считать, что в кластере от каждого узла до каждого другого узла можно добраться по пути длиной 1, то есть по ребру. Это значит, что кластер имеет диаметр 1, то есть является кликой. При этом узел может входить в несколько кластеров, то есть кластеры могут иметь общие узлы. Узел v_0 опрашивает узлы всех кластеров, в которые он входит. Все эти узлы находятся на расстоянии 1 от узла v_0. Когда на раунде r сообщение ***Question*** проходит путь $v_0..v_{r-1}$, $r \geq 2$, узел v_{r-1} опрашивает узлы всех тех кластеров, в которые входит он сам и не входит предыдущий узел v_{r-2}, поскольку их узлы находятся от корня v_0 на расстоянии $r - 1$ или меньше (и опрошены на предыдущих раундах).

Когда опрос происходит по этим правилам, сообщение ***Question*** передаётся только по такому ребру, которое принадлежит некоторому кластеру (порождённому им подграфу). Остальные рёбра графа будем называть *лишними*, и их можно удалить. После этого каждое оставшееся ребро принадлежит некоторому кластеру.

Утверждение 1. Для того чтобы гарантировать достижимость узлов, необходимо и достаточно, чтобы после удаления лишних рёбер граф оставался связным.

Доказательство: Очевидно.

□

Утверждение 2. Для того чтобы не было дублирования, необходимо и достаточно, чтобы после удаления лишних рёбер в графе для любого простого (не проходящего дважды через один узел) цикла всего его узлы принадлежали некоторому кластеру.

Доказательство: Необходимость. Пусть нет дублирования, но после удаления лишних рёбер в графе имеется простой цикл $a...b...a$, где узлы a и b не принадлежат одному кластеру. Поскольку каждый кластер является кликой, в этом цикле каждый отрезок $x...y$, проходящий по узлам некоторого кластера, можно заменить ребром xy. Повторяя эту процедуру замены пока возможно, получим простой цикл $ac_1...c_kbd_1...d_ma$, в котором для каждого отрезка xyz длиной 2 узлы x и z не принадлежат одному кластеру. Поскольку узлы a и b не принадлежат одному кластеру, ребро ab, если бы оно было в графе, было лишним, и при удалении лишних рёбер оно удаляется, следовательно, $k \geq 1$ и $m \geq 1$. А тогда у нас есть два пути из узла a в узел b, по которым пройдёт сообщение ***Question*** из начального узла a, если ближайший конечный узел это узел b: $ac_1...c_kb$ и $ad_m...d_1b$. Эти пути разные, поскольку $k \geq 1$ и $m \geq 1$ и в простом цикле все узлы $ac_1...c_kbd_1...d_m$ попарно разные. Но это противоречит отсутствию дублирования.

Достаточность. Пусть после удаления лишних рёбер в графе любой простой цикл проходит через узлы только одного кластера. Покажем, что тогда нет дублирования. Пусть сообщение ***Question*** проходит путь p. Пути длиной 1 не дублируются, поскольку нет кратных рёбер. Покажем, что если длина пути p больше 1, то путь p простой. В таком пути для любых двух подряд идущих дуг ab и bc узлы a и c не принадлежат одному кластеру. Поэтому, если путь p не простой, в нём есть простой цикл длины больше 2, поскольку нет петель и кратных рёбер. А тогда в этом простом цикле должны быть два узла, не принадлежащие одному кластеру. Но это противоречит тому, что цикл, не проходящий по лишним рёбрам, проходит через узлы некоторого кластера. Пусть есть два разных простых пути p и q с общим началом x и общим концом y, по которым двигается сообщение ***Question***. Такие пути можно представить в виде $p = p_1p_2p_3$ и $q = q_1q_2q_3$, где $p_1 = q_1$ общий максимальный префикс путей p и q (такой путь есть, поскольку пути p и q имеют общее начало), путь p_2 минимальное продолжение пути p после префикса p_1 до узла, общего с путём q (такой путь есть, поскольку пути p и q имеют общий конец), а q_2 соответствующее продолжение пути q после префикса q_1. Очевидно, пути p_2 и q_2 имеют ненулевую длину, и, поскольку пути p и q разные и нет кратных рёбер, хотя бы один из путей p_2 и q_2 имеет длину больше 1. Следовательно, на этом пути есть два узла, которые не принадлежат одному кластеру. А тогда эти пути p_2 и q_2 образуют простой цикл, в котором есть два узла, не принадлежащие одному кластеру, что противоречит предположению.

□

В дальнейшем для простоты будем считать, что в графе связей нет «лишних» рёбер. Суммируя, сформулируем *требования* к кластеризованному графу связей:

1) Граф связен.
2) Каждый кластер является кликой.
3) Нет вложенных кластеров.
4) Каждое ребро принадлежит некоторому кластеру.
5) Для любого простого цикла всего его узлы принадлежат некоторому кластеру.

Как было сказано выше, такие требования предъявляются к графу после удаления «лишних» рёбер. Это значит, что на самом деле нам нужно, чтобы требованиям удовлетворял не сам граф, а некоторый его суграф (подграф, содержащий все узлы графа, но, быть, может, не все рёбра). Только рёбра такого суграфа будут использоваться для передачи сообщений. В дальнейшем для простоты изложения под графом связей будем понимать такой суграф.

Формально граф без рёбер, состоящий из одного узла a, образующего один кластер {a}, удовлетворяет требованиям 1-5. Этот случай в дальнейшем не рассматривается, поскольку в такой вырожденной сети вообще нет передачи сообщений.

Утверждение 3. Если в графе, удовлетворяющем требованиям 1-5, больше одного узла, каждый кластер содержит, по крайней мере, два узла.

Доказательство. Допустим утверждение не верно, и есть кластер $\{a\}$, где a узел графа. Тогда узел a не принадлежит никакому другому кластеру, поскольку в такой кластер был бы вложен кластер $\{a\}$, что противоречит требованию 3 (нет вложенных кластеров). Но тогда узлу a не инцидентно никакое ребро, поскольку по требованию 4 ребро соединяет узлы, принадлежащие хотя бы одному кластеру. Поскольку в графе больше одного узла, из узла a недостижим другой узел, что противоречит требованию 1. Мы пришли к противоречию, следовательно, наше допущение не верно, и утверждение доказано.

□

Утверждение 4. Если граф удовлетворяет требованиям 1-5, то общий узел двух разных кластеров, если он есть, единственный и является *шарниром*[1] в графе.

Доказательство: Допустим, утверждение не верно, и есть два кластера A и B с общими узлами x и y, $x \neq y$. Поскольку по требованию 3 нет вложенных кластеров, имеется узел $a_1 \in A \setminus B$, $a_1 \neq x$, $a_1 \neq y$ и узел $b \in B \setminus A$, $b \neq x$, $b \neq y$. Выберем в качестве кластера A максимальный по числу узлов кластер, содержащий узлы x, y, a_1. Пусть A состоит из узлов x, y, a_1, ..., a_n. Поскольку по требованию 2 каждый кластер является кликой, имеется простой цикл $xa_1...a_nybx$. Тогда по требованию 5 должен быть кластер C, содержащий все узлы этого цикла, то есть $A \subset C$, что противоречит требованию 3 (отсутствие вложенных кластеров). Мы пришли к противоречию, следовательно, наше допущение не верно, и утверждение доказано.

□

Двусвязным графом (biconnected graph) называется граф, в котором удаление одного узла вместе с инцидентными ему рёбрами не приводит к нарушению связности графа. Это определение отличается от определения (вершинно) k-связного графа (k-[vertex]-connected graph) для $k = 2$, поскольку в нём дополнительно требуется, чтобы в графе было больше k узлов. Граф, состоящий из двух узлов a и b и соединяющего их ребра ab, двусвязен (biconnected), но не 2-связен (2-connected). Использование терминов «двусвязный» и «2-связный» для обозначения разных, хотя и близких понятий, нельзя признать удачным, однако оно общепринято. В дальнейшем мы будем говорить именно о двусвязности графов, то есть рассматривать также двусвязные графы с числом узлов 2, то есть содержащие одно ребро. *Компонентой двусвязности* или *блоком* называют максимальный по вложенности узлов двусвязный подграф. Для любого связного неориентированного графа двудольный граф, узлы которого это блоки и шарниры, а ребро соединяет шарнир с блоком, которому он принадлежит, является деревом. Все его листья являются блоками, а внутренние узлы это блоки и шарниры. Это дерево называют *деревом блоков и шарниров* исходного графа (*block cutpoint graph*) [7]. *Блоковым графом* (block graph) называется неориентированный граф, в котором каждый блок является кликой [8-9]. Связный блоковый граф называют также *кликовым деревом* (clique tree), поскольку в его дереве блоков и шарниров каждый блок является кликой. Название тоже не очень удачное, поскольку имеется другое понятие с похожим названием: *кликовый граф* или *граф клик* (clique graph), который определяется как граф, узлами которого являются максимальные клики исходного графа, и два узла соединены ребром, если соответствующие клики имеют непустое пересечение [10-11]. В частности, граф клик (clique graph) может быть деревом, но это не кликовое дерево (clique tree). На рис. 2

[1] Шарнир = точка сочленения (cut vertex, separating vertex, articulation point) = узел графа, при удалении которого количество компонент связности возрастает.

показан пример блокового графа (кликового дерева), соответствующие ему дерево блоков и шарниров и граф клик.

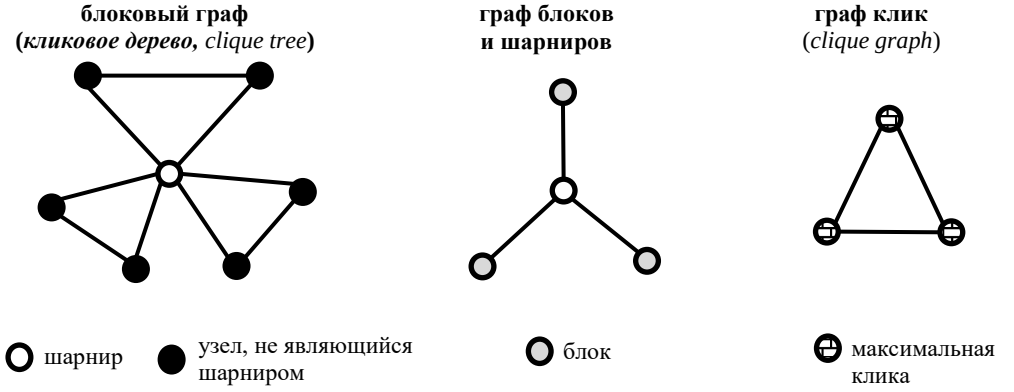

Рис. 2. Отличие кликового дерева от графа клик.
Fig. 2. The difference between a clique tree and a clique graph.

Утверждение 5. Граф удовлетворяет требованиям 1-5 тогда и только тогда, когда он является связным блоковым графом.

Доказательство: Покажем, что граф, удовлетворяющий требованиям 1-5, является связным блоковым графом. Связность следует из требования 1. Нужно показать, что кластер является блоком, и никаких других блоков в графе нет. Сначала покажем, что каждый кластер является блоком. Поскольку по требованию 2 кластер является кликой, а клика двусвязна, нужно показать, что кластер является максимальным по вложенности узлов двусвязным подграфом. Действительно, в противном случае некоторый кластер A строго вложен в некоторый блок H. Тогда в блоке H имеется ребро ab, где узел a принадлежит кластеру A, а узел b не принадлежит кластеру A. Но по требованию 4 ребро ab (оба его конца a и b) должно принадлежать некоторому кластеру B, в частности, узел a принадлежит кластеру B, и, поскольку узел b не принадлежит кластеру A, $B \neq A$. Тогда узел a является общим узлом разных кластеров A и B. По утверждению 4 узел a является шарниром в графе и, следовательно, в блоке H, однако блоки как компоненты двусвязности не могут содержать шарниров. Мы пришли к противоречию и, следовательно, кластер это блок. Теперь покажем, что никаких других блоков, кроме кластеров, в графе нет. Действительно, блок должен быть вложен в некоторый кластер, так как в противном случае в блоке есть два узла, принадлежащие разным кластерам, а тогда между ними в блоке есть путь, проходящий через общий узел двух разных кластеров. Такой узел по утверждению 4 является шарниром, но блок не может содержать шарниров. При этом блок не может быть строго вложен в кластер, поскольку кластер двусвязен, а блок это максимальный по вложенности узлов двусвязный подграф.

Теперь покажем, что блоковый граф удовлетворяет требованиям 1-5, если кластеры это блоки графа. По определению блоковый граф связен (требование 1) и каждый его блок является кликой (требование 2). Поскольку блок это максимальный по вложенности двусвязный подграф, то нет вложенных блоков (требование 3). Поскольку ребро является двусвязным подграфом, а блок это максимальный по вложенности узлов двусвязный подграф, то каждое ребро принадлежит некоторому блоку (требование 4). Выполнение требования 5 следует из того, что простой цикл является двусвязным подграфом и, следовательно, вложен в максимальный двусвязный подграф, то есть блок.

□

В блоковом графе для любой пары узлов существует единственный кратчайший путь, соединяющий их [9,12]. Именно по таким путям будут двигаться сообщения.

Утверждение 6. В любом связном графе есть суграф, являющийся блоковым графом.

Доказательство: Возьмём произвольный остов графа. Он является связным суграфом. В дереве каждый нетерминальный узел является шарниром. Поэтому блок – это два смежных узла. Каждый такой блок является кликой.

□

Формально это утверждение позволяет организовать сеть с опросом на любом связном графе. Но, конечно, не на любом связном графе будет выполняться в полном объёме условие 4: диаметр графа должен быть достаточно мал. Пример блокового графа приведён на рис. 3.

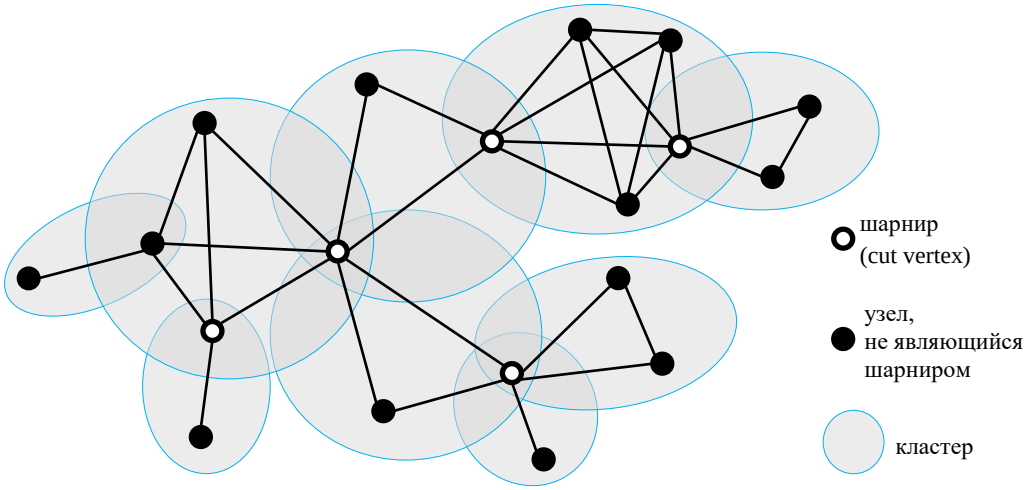

Рис. 3. Пример блокового графа диаметром 4.
Fig. 3. Example of a block graph of diameter 4.

3. Обмен сообщениями

На основе требований 1-5 и утверждений 4 и 5, уточним процедуру опроса. На раунде r через v_i, $i = 0..r$, будем обозначать узел на расстоянии i от начального узла. Начальный узел v_0. Кратчайшие пути, которые проходят сообщения ***Question***, образуют корневое дерево, корнем которого является начальный узел v_0, а внутренние узлы (не листья и не корень) – шарнирами. Корень и листья могут быть как шарнирами, так и не шарнирами. Там, где это не приводит к двусмысленности, мы будем использовать термины «корень» и «начальный узел» как синонимы. Это дерево будем называть *деревом опроса* и обозначать $T(v_0)$. На пути $v_0...v_l$ от корня v_0 до листа v_l дерева $T(v_0)$ узел v_{i-1}, $i = 1..l$, будем называть *родителем* (parent) узла v_i, а узел v_{i+1}, $i = 0..l - 1$, *сыном* (child) узла v_i (сына также называют дочерним элементом или непосредственным потомком). Поддерево дерева $T(v_0)$ с корнем в начальном узле, которое проходится на раунде r, будем обозначать $T(v_0, r)$. В дереве $T(v_0, r)$ все узлы на расстоянии $1..r - 1$ от корня v_0 являются шарнирами. Поддерево дерева $T(v_0, r)$ с корнем в узле v_i будем обозначать $T(v_i, r)$.

Листья дерева $T(v_0, r)$ бывают двух типов: 1) узел v_l, который находится на расстоянии r от корня v_0, то есть $l = r$, 2) для $r > 2$ узел v_l, который является шарниром, находится на расстоянии $l = 1..r - 2$ от корня, а все его сыновья не шарниры. *Пограничным узлом* на раунде r будем называть узел v_i дерева $T(v_0, r)$ на расстоянии $r - 1$ от корня, то есть $i = r - 1$, если у него есть сыновья в дереве $T(v_0)$. Очевидно, все эти сыновья являются листьями 1-го типа

дерева $T(v_0, r)$. Все пограничные узлы, кроме, быть может, корня для $i = 1$, являются шарнирами.

На раунде r сообщение ***Question*** сначала движется по пути от корня через цепочку внутренних узлов (шарниров) либо до листа v_l 1-го типа дерева $T(v_0, r)$, либо до пограничного узла v_{r-1}. В первом случае среди сыновей узла v_l нет шарниров. Поскольку его сыновья находятся на расстоянии $l + 1 < r$, они опрошены на предыдущих раундах, поэтому дальше сообщение ***Question*** не посылается. А во втором случае пограничный узел рассылает дальше сообщения ***Question*** всем узлам кластеров, в которых он входит, кроме того кластера, в который входит родитель (узел от которого он получил сообщение ***Question***), то есть листьям 2-го типа дерева $T(v_0, r)$. Тем самым, на раунде r сообщение ***Question*** проходит по всем дугам дерева $T(v_0, r)$, ориентированного от корня. На раунде r опрашиваются листья дерева $T(v_0, r)$ на расстоянии r от корня, поскольку остальные узлы этого дерева опрошены на предыдущих раундах. В наихудшем случае (когда в сети нет требуемого узла) число раундов будет равно эксцентриситету начального узла (максимальному из всех минимальных расстояний начального узла от других узлов), которое не превышает диаметр графа, и будут опрошены все узлы, каждый по одному разу. Схема иллюстрируется на рис. 4, где рёбра деревьев $T(v_0, 1)$, $T(v_0, 2)$, $T(v_0, 3)$, $T(v_0, 4)$ помечены, соответственно, цифрами 1, 2, 3, 4 и обозначены кластеры K_0, ..., K_{1111}, и путь $v_0v_1v_2v_3v_4$ от начального узла (корня) v_0 до конечного узла v_4.

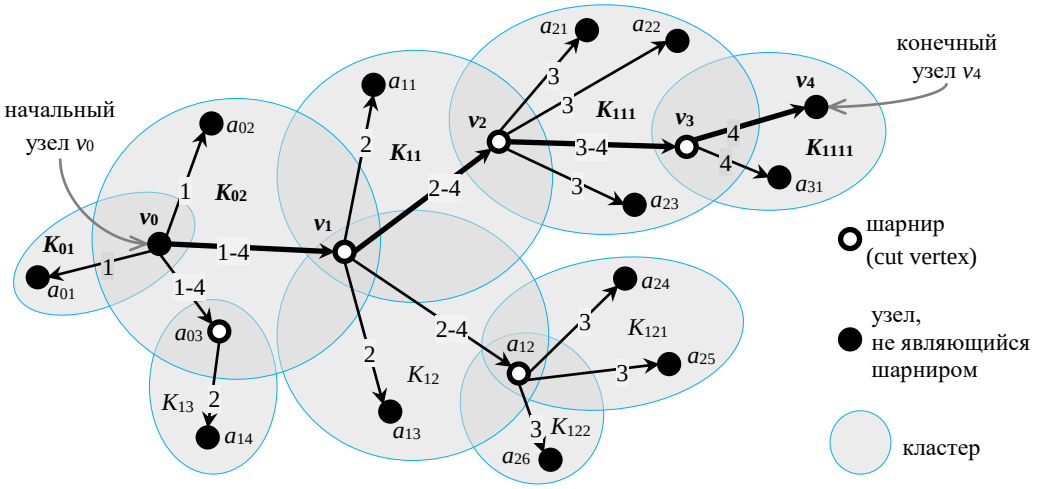

Рис. 4. Схема опроса короткими сообщениями по графу связей.
Fig. 4. Scheme of using short messages when traversing the connection graph.

Рассмотрим процедуру опроса более детально. Набор сообщений, схема обмена сообщениями, параметры сообщений и динамические переменные в узлах сети зависят от решения ряда возникающих при опросе вопросов. Мы анализируем возможные решения для того, чтобы обосновать принятые в нашей модели решения и показать, как предлагаемую модель можно использовать для сравнения с другими моделями.

Вопрос 1: Где запоминать путь? Путь $v_0...v_l$, который проходит сообщение ***Question*** от корня v_0 до листа v_l, нужно запомнить тогда, когда этот лист оказался конечным узлом и отвечает сообщением ***Yes***, которое двигается по тому же пути, но в обратном направлении. Далее сообщение ***Request***, посылаемое корнем после получения ответа ***Yes***, должен пройти тот же путь $v_0...v_l$ в прямом направлении, а после этого сообщение ***Result***, посылаемое конечным узлом, должно пройти этот же путь в обратном направлении до корня. Каким образом было бы желательно запомнить этот путь?

Одним из вариантов решения этого вопроса было бы запоминание пути в параметрах сообщений. Когда сообщение ***Question*** проходит ребро $v_{i-1}v_i$, узел v_i добавляет в конец пройденного пути, запомненного в сообщении, свой идентификатор v_i. Путь от корня до конечного узла помещается конечным узлом в ответ ***Yes***, и оно двигается по этому пути в обратном направлении. Корень, получив ответ ***Yes***, помещает путь в параметры сообщения ***Request***, и оно двигается по пути в прямом направлении до конечного узла. Конечный узел переписывает путь в параметры сообщения ***Result***, которое двигается по этому пути в обратном направлении до корня. Однако такое решение увеличивает размер сообщений ***Question***, ***Yes***, ***Request*** и ***Result***, в наихудшем случае на величину, определяемую эксцентриситетом начального узла, ограниченном сверху диаметром графа. Сообщения ***Request*** и ***Result*** не ограничены по длине, и для них такое увеличение можно считать допустимым. Однако сообщения ***Question*** и ***Yes*** желательно делать короткими и фиксированной длины.

Вместо этого мы предлагаем запоминать путь $v_0...v_l$ в его узлах $v_0,...,v_l$. В узле v_i запоминается идентификатор его родителя v_{i-1} в переменной $parent = v_{i-1}$. Это *обратные* ссылки по дереву в направлении от листьев к корню. Когда сообщение ***Question*** пересылается от родителя v_{i-1} его сыну v_i, родитель v_{i-1} помещает в сообщение свой идентификатор как параметр, а сын v_i переписывает его в свою переменную *parent*. Сообщение ***Yes*** двигается как раз по обратным ссылкам *parent* от сына к родителю. При этом, когда сообщение ***Yes*** пересылается от сына v_i его родителю v_{i-1}, сын v_i помещает в сообщение свой идентификатор как параметр, а родитель v_{i-1} переписывает его в свою переменную *child*. Это *прямые* ссылки по дереву в направлении от корня к конечному узлу. При таком решении сообщения ***Question*** и ***Yes*** имеют небольшую фиксированную длину.

Вопрос 2: Как долго хранить в узлах информацию, связанную с обработкой данного запроса услуги, в частности, ссылки *parent* и *child*? Параллельно в сети может обрабатываться много запросов услуг от одного или разных начальных узлов. Поэтому желательно, чтобы динамические переменные, которые узел создаёт для обработки данного запроса, хранились в нём как можно меньший интервал времени. Работу узла на этом интервале назовём *сеансом*, а соответствующие динамические переменные – *переменными сеанса*. В корне v_0 сеанс открывается, когда отправляется первое сообщение ***Question*** и закрывается получением сообщения ***Result***. В конечном узле v_l сеанс открывается при получении сообщения ***Question*** и закрывается с отправкой сообщения ***Result***. Интервал времени между сообщениями такой пары нельзя уменьшить, и он не ограничен сверху, поскольку зависит от времени оказания услуги в конечном узле.

В промежуточном узле v_i, $i = 1..l - 1$, пути $v_0...v_l$ можно было бы также открывать сеанс при получении сообщения ***Question*** и закрывать при получении и отправке дальше сообщения ***Result***. Однако в этом случае длительность сеанса в промежуточном узле (а таких узлов может быть несколько) также будет зависеть от неограниченного сверху времени оказания услуги в конечном узле.

К счастью, есть другое решение: закрывать сеанс в промежуточном узле при получении и отправке дальше сообщения ***Request***. В этом случае длительность сеанса в промежуточном узле зависит только от скорости передачи сообщения по ребру графа и эксцентриситета начального узла, ограниченного сверху диаметром графа. Но для этого нужно, чтобы путь запоминался в сообщении ***Request*** как его параметр *path*: при передаче этого сообщения по ребру v_iv_{i+1} узел v_i добавляет в конец параметра *path* прямую ссылку $child = v_{i+1}$. Конечный узел v_l получает в сообщении ***Request*** полный путь $v_0...v_l$, запоминает его, а после оказания услуги помещает в сообщение ***Result*** как его параметр *path*. Тем самым длина сообщений ***Request*** и ***Result*** увеличивается на длину пути (максимум на эксцентриситет начального узла, ограниченного сверху диаметром графа), но эти сообщения и так считаются длинными.

Вопрос 3: Что делать, если нет ответа ***Yes***? Когда нелистовой узел v_i рассылает сообщения ***Question*** всем узлам, связанным с ним ребром и находящимся на расстоянии r от корня, он ожидает ответа ***Yes***. Но что делать, если такого ответа не будет?

Одним из вариантов решения этой проблемы было бы установление таймаута. Но этот таймаут должен быть разным в разных нелистовых узлах: если в пограничном узле v_{r-1} таймаут равен τ, то в нелистовом узле v_i на пути $v_0 ... v_{r-2}$ (для $r > 2$) нужно устанавливать таймаут $(r - i)\tau$. Кроме того, работа с таймаутами приводит к неоправданному увеличению времени каждого сеанса и каждого раунда, на которых не приходит ответ ***Yes***.

Мы предлагаем вместо таймаута использовать отрицательный ответ от узлов, которые не могут оказать требуемую услугу: такой узел посылает в ответ на сообщение ***Question*** сообщение ***No***. В таком листовом узле не нужно хранить какие-то динамические переменные, и поэтому сеанс в нём не открывается. В нелистовом узле устанавливается счётчик *counter* как переменная сеанса, при открытии сеанса равный числу сыновей узла в пограничном узле или числу сыновей-шарниров в других внутренних узлах, который уменьшается на 1 при получении ответа ***Yes*** или ***No***. Когда счётчик *counter* становится равным 0, это означает, что все ответы получены. В то же время при получении ответа ***Yes*** сразу выполняются нужные действия: внутренний узел дерева $T(v_0, r)$ пересылает этот ответ ***Yes*** своему родителю, а корень посылает сообщение ***Request*** для запроса услуги у найденного конечного узла. Если счётчик *counter* обнулился, а ответа ***Yes*** не было, то внутренний узел посылает своему родителю ответ ***No***, а в корне заканчивается данный раунд r и начинается раунд $r + 1$, то есть опрос узлов на расстоянии $r + 1$ от корня или, если r достиг эксцентриситета начального узла, выносится вердикт: в сети нет требуемой услуги.

Вводится статус сеанса для нелистового узла, который хранится в переменной сеанса *status*: *no* – не получен ответ ***Yes***, *yes* – получен ответ ***Yes*** и не получено сообщение ***Request*** (для внутреннего узла) или ***Result*** (для корня), *end* – получено сообщение ***Request*** (для внутреннего узла) или ***Result*** (для корня).

Каждый сеанс закрывается освобождением памяти из-под переменных сеанса. В листе это происходит, когда услуга оказана и отправлено сообщение ***Result***. В нелистовом узле – когда счётчик *counter* обнулился и достигнут статус *end*.

Нужно отметить, что в отличие от решения с таймаутом предлагаемое решение со счётчиком ответов опирается на предположение, что сообщения не теряются. Однако такая потеря может быть неизбежной, когда выходит из строя либо соседний узел, которому направлялось сообщение ***Question*** или от которого должно было отправиться сообщение ***Yes***, либо канал связи с этим соседним узлом, то есть ребро графа. В этом случае ожидание ответа может стать бесконечным. Для того чтобы избежать такого бесконечного ожидания, мы предполагаем, что в сети имеется специальная служба, отслеживающая изменения топологии сети и при необходимости оповещающая об этом узлы сети, соседние с удаляемым из графа сети узлом или инцидентные удаляемому ребру. В настоящей статье мы не рассматриваем работу этой службы.

Вопрос 4: Что делать, если узел мог бы оказать запрашиваемую услугу, но сейчас «занят» и готов оказать услугу не немедленно, а через какое-то время, когда освободится?

Одно решение такое: узел в ответ на сообщение ***Question*** посылает сообщение ***Yes***. Однако в этом случае может оказаться, что именно этому узлу придёт сообщение ***Request***, и этому сообщению придётся ждать освобождения узла, тогда как в сети, возможно, есть другие узлы, готовые оказать услугу немедленно.

Другое решение: узел посылает ответ ***No***. Однако тогда может оказаться, что все узлы ответили сообщением ***No***, и эту ситуацию нельзя отличить от ситуации, когда в сети вообще нет узлов, которые могли бы (немедленно или позже) оказать требуемую услугу.

Мы предлагаем ввести сообщение ***Wait*** как ответ на сообщение ***Question***. В этом ответе как параметр указывается предполагаемое время ожидания *time*. Это не таймаут, назначение этого параметра *time* другое: узел, пославший ответ ***Wait***, сообщает начальному узлу, что он готов оказать требуемую услугу, но не немедленно, а через время, указанное в параметре *time*. Если ни один узел не послал ответ ***Yes***, но несколько узлов послали ответ ***Wait***, начальный узел должен выбрать один из узлов, пославших ответ ***Wait***, и направить ему сообщение ***Request***. Естественно, выбирается тот узел, который указал минимальное время *time*.

Узел, получивший ответ ***Wait***, обрабатывает его также как ответ ***No***, но отмечает, что был ответ ***Wait***, и хранит в переменной сеанса время *time* как минимальное среди всех полученных ответов ***Wait***. Соответственно, у статуса сеанса появляется ещё одно значение: *wait* – не получен ответ ***Yes***, но получен ответ ***Wait***. В статусе *wait*, кроме переменной *time*, имеется переменная *child*, в которой сохраняется прямая ссылка на сына, приславшего ответ ***Wait*** с минимальным временем ожидания. Это аналогично прямой ссылке *child* для ответа ***Yes***, а отличие от ответа ***Yes*** в том, что ответ ***Yes*** немедленно после получения пересылается родителю и, в конечном счёте, приходит в корень, а ответ ***Wait*** ожидает обнуления счётчика *counter*. Кроме того, при получении ответа ***Yes*** в статусе *wait*, устанавливается статус *yes*, поэтому дальнейшие ответы ***Wait*** ничего не сохраняют в переменных сеанса. Это значит, что ответ ***Yes*** приоритетнее ответа ***Wait***.

Когда счётчик *counter* обнуляется, а статус сеанса равен *wait*, внутренний узел посылает своему родителю (по обратной ссылке *parent*) сообщение ***Wait*** с сохранённым минимальным временем *time* как параметром. Когда счётчик *counter* обнулится в корне, а статус равен *wait*, корень заканчивает данный раунд r и начинает раунд $r + 1$, то есть опрос узлов на расстоянии $r + 1$ от корня в надежде получить сообщение ***Yes***, которое означает возможность немедленного оказания услуги. Если же r достиг эксцентриситета начального узла, а статус остался *wait*, корень посылает сообщение ***Request*** (по прямым ссылкам *child*) тому узлу, который прислал сообщение ***Wait*** с минимальным временем ожидания. В этом узле запрос ожидает освобождения узла от текущей работы и перехода к оказанию услуги.

Вопрос 5: Что делать, если приходит несколько ответов ***Yes*** и/или ***Wait***? Понятно, что лучше всего взять в обработку первый ответ ***Yes***, но что делать с повторными ответами ***Yes***, приходящими от других узлов? Понятно, что среди ответов ***Wait*** нужно выбрать (сохранить время ожидания и прямую ссылку на сына, приславшего ответов ***Wait***) то, в котором минимальное время ожидания, но что делать с другими ответами ***Wait***, пришедшими ранее или приходящими позже от других узлов?

Можно было бы просто игнорировать «лишние» ответы ***Yes*** и ***Wait***. Однако проблема в том, что листовой узел, приславший ответ ***Yes*** или ***Wait***, рассчитывает на то, что может стать конечным узлом, то есть получить сообщение ***Request***. Сообщение ***Question*** как бы «бронирует» оказание услуги в таком листовом узле, эта «бронь», возможно, предполагает резервирование каких-то ресурсов в листовом узле и в любом случае влияет на то, какие ответы будет посылать листовой узел на последующие сообщения ***Question***.

Поэтому предлагается при получении «лишнего» ответа ***Yes*** или ***Wait*** посылать (по прямым ссылкам *child*) отмену «брони»: сообщение ***Cancel***.

Вопрос 6: Как обеспечить параллельную работу с несколькими запросами от одного или разных узлов? Проблема здесь в том, что с каждым сеансом связаны переменные сеанса, а при такой параллельной работе в одном узле может быть открыто сразу несколько сеансов, которые нужно как-то идентифицировать для различения их между собой. В частности, необходимо различать сеансы, чтобы было понятно, к какому сеансу относится тот или иной ответ (***Yes***, ***No***, ***Wait***). При открытии сеанса под описание сеанса, содержащее параметры сеанса, выделяется динамическая память. Сеанс идентифицируется указателем на эту память.

Вместе с обратной (*parent*) или обратной (*child*) ссылкой на узел в описании сеанса или в параметрах сообщений задаётся также указатель на описание сеанса в этом узле.

Важным достоинством предлагаемой модели является то, что суммарный размер переменных сеанса является константой небольшой величины, зависящей от типа узла в данном запросе услуги: корень v_0, внутренний узел или лист дерева $T(v_0, r)$. Только в конечном узле на время оказания услуги (между получением сообщения ***Request*** и отправкой сообщения ***Result***) дополнительно нужно учитывать размер описания пути от родителя конечного узла до корня, которое хранится в динамически выделяемой памяти, указатель на которую помещается в описание сеанса.

Вопрос 7: Как узнать, что номер раунда r достиг значения эксцентриситета начального узла? Можно было бы хранить эксцентриситет узла в статической (не меняющейся при обмене сообщениями) переменной каждого узла (каждый узел может быть начальным узлом). Однако в этом случае любое изменение топологии сети, приводящее к изменению эксцентриситетов узлов, в том числе диаметра графа, потребовало бы глобальной перенастройки, а именно изменения этого параметра во многих или даже всех узлах сети.

Вместо этого предлагается на каждом раунде r динамически определять, имеет ли смысл начинать следующий раунд $r + 1$, если корень на раунде r не получил ответа ***Yes***. Критерий в этом случае достаточно простой: следующий раунд $r + 1$ нужно начинать, если в текущем раунде r среди листьев дерева $T(v_0, r)$ на расстоянии r от корня есть хотя бы один шарнир. Для этого узел должен знать своё расстояние от корня, а точнее разность между раундом и этим расстоянием. Если разность равна 0, узел находится на расстоянии r в раунде r. Эта разность является параметром *rounddistance* сообщения ***Question***. Корень помещает в него значение раунда r, а при каждой пересылке сообщения ***Question*** параметр *rounddistance* уменьшается на 1. Когда узел, получив сообщение ***Question***, обнаруживает, что он на раунде r находится на расстоянии r от корня (*rounddistance* = 0), то есть это лист 1-го типа, узел, посылая ответ ***No*** или ***Wait***, помещает в него как параметр булевский признак *cutnodeflag*[2], равный ***true***, если узел шарнир, и ***false*** в противном случае. Лист 2-го типа всегда помещает в ответ ***No*** или ***Wait*** параметр *cutnodeflag* = ***false***. Внутренний узел дерева $T(v_0, r)$ пересылает родителю ответ ***No*** или ***Wait*** при обнулении счётчика *counter* с параметром *cutnodeflag* равным дизъюнкции таких параметров по всех полученных ответах ***No*** и ***Wait***. Корень также вычисляет дизъюнкцию параметров *cutnodeflag*, а при обнулении счётчика *counter* открывает следующий раунд, если дизъюнкция получилась истинной. Таким образом, эксцентриситет начального узла, который нужно знать, если ни один узел не послал ответ ***Yes***, определяется динамически в процессе опроса. Эксцентриситет узла не нужно хранить в памяти узла и, тем самым, его не нужно корректировать при изменении топологии сети.

Вопрос 8: Какая статическая информация должна храниться в узлах сети? Минимально необходимая информация задаёт топологию кластеризованного графа. В каждом узле v_i хранится локальная информация, касающаяся только этого узла: идентификатор узла в переменной *Self* и множество кластеров, в которые входит данный узел, в переменной *Clusters*. Каждый кластер задаётся двумя множествами: множеством узлов кластера, кроме самого узла v_i, то есть множеством соседей узла (если в графе удалены «лишние» рёбра) в переменной *neighbors*, и его подмножеством шарниров в переменной *cutnodes*. Узел является шарниром тогда и только тогда, когда он принадлежит более чем одному кластеру.

Резюме. Последовательность передачи сообщений между соседними узлами наглядно изображена на рис. 5. Общая схема передачи сообщений показана на рис. 6. Правила коммутации сообщений в узлах сети показаны в табл. 1. В табл. 2 приведены примерные

[2] В Приложении мы используем в идентификаторах наименование «cutnode» вместо «cut vertex», поскольку в основном тексте статьи вместо «вершина» (vertex) графа используем наименование «узел» (node) графа.

размеры типов и суммарные размеры переменных сеанса и параметров сообщений. В Приложении приведены алгоритмы обработки сообщений.

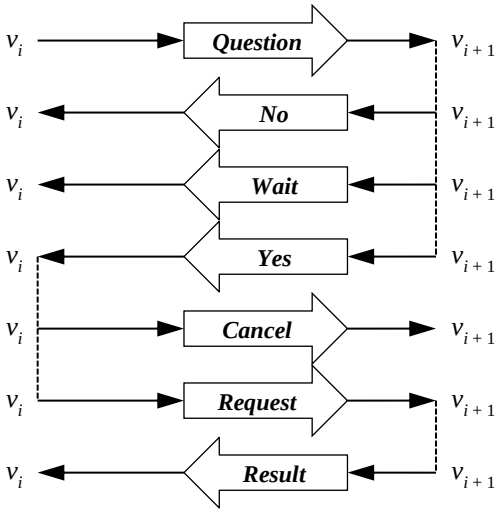

Рис. 5. Последовательность передачи сообщений между соседними узлами.
Fig. 5. Message transfer between neighboring nodes.

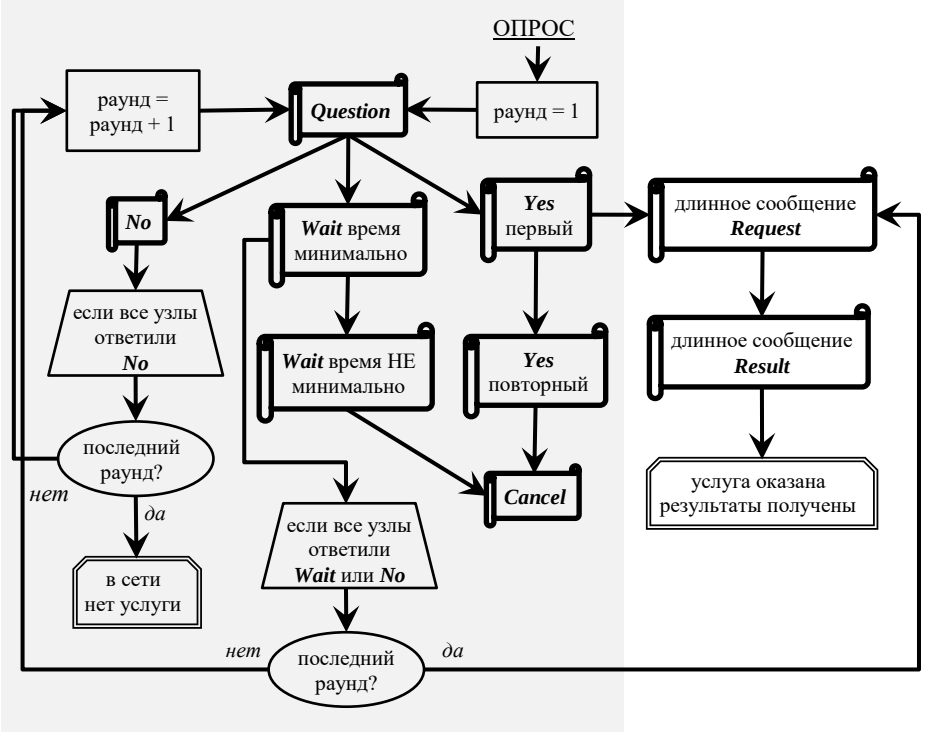

Рис. 6. Схема передачи сообщений.
Fig. 6. Message flow diagram.

Табл. 1. Правила коммутации в узлах сети.
Table 1. Switching rules in network nodes.

Root		internal				leaf
v_1	...	v_{i-1} родитель	v_i получатель	v_{i+1} сын	...	v_l

сообщение	параметры сообщения	информация в получателе		кому пересылается сообщение
		статика	на сеанс	
Question →	*rounddistance* раунд минус расстояние *parent* родитель	*Clusters* описание кластеров, в которые входит получатель	–	Если получатель шарнир, для ∀ кластера из *Clusters*, в который не входит *parent*: если *rounddistance* = 1, всем узлам в этих кластерах, если *rounddistance* > 1, всем шарнирам в этих кластерах.
← ***No***	*s* указатель на сеанс в получателе	–	*s*→*parent* родитель	родителю *s*→*parent*
← ***Wait***		–		
← ***Yes***		–		
Cancel →		–	*s*→*child* сын	сыну *s*→*child*
Request →		–		
← ***Result***	*path* путь от родителя до корня	–	–	родителю *path*(|*path*|)

Табл. 2. Размеры типов и суммарные размеры переменных сеанса и параметров сообщений.
Table 2. Type size and total size of session variables and messages parameters.

тип	размер в битах	комментарий
bool	1	
unsigned	32	
node идентификатор узла	128	размер IP-адреса в IPv6
service имя услуги	160	рекомендуемая длина имени не более 20 символов
указатель	64	для 64-х разрядной машины
leaf описание сеанса в листе	130	переменные сеанса
intern описание сеанса во внутреннем узле	453	
root описание сеанса в корне	581	
Question	387	короткие сообщения: суммарный размер параметров + имя сообщения (3 бита)
No	68	
Wait	292	
Yes	259	
Cancel	67	
Request ... – размер параметров	288 + (≤ 640) + ... 928 + ...≤	длинные сообщения суммарный размер параметров (путь длиной до 5 для «шести рукопожатий», то есть диаметр графа ≤ 6)
Result ... – размер ответных параметров	67 + (≤ 640) + ... ≤ 707 + ...	

4. Заключение

В статье предложена модель кластеризованной распределённой сети с опросом. В такой сети каждый узел может запросить ту или иную услугу, которая может быть оказана каким-то

другим узлом сети[3]. Узлов, которые могут оказать требуемую услугу, в сети может быть несколько, и, как правило, неизвестно, какие узлы годятся. Требуется найти такой узел, и желательно, чтобы это был ближайший из подходящих узлов. Для этого осуществляется опрос сети, который выполняется в несколько этапов (раундов): сначала опрашиваются узлы на расстоянии 1, если нужный узел не найден, опрашиваются узлы на расстоянии 2 и так далее. Когда узел найден, сообщение запроса услуги двигается к нему по кратчайшему пути, а после оказания услуги по тому же пути, но в обратном направлении двигается ответное сообщение с результатами оказания услуги.

Предложенная модель обеспечивает достижимость узлов, отсутствие дублирования сообщений (когда один узел получает несколько однотипных сообщений), нацелена на предотвращение перегрузки сети и ориентирована на минимизацию как времени выполнения опроса, так и размера требуемой для этого памяти в узлах и размеров сообщений.

Кластеризованность распределённой сети означает, что сеть состоит из множества кластеров, а каждый кластер – это подмножество узлов сети и соединяющих их рёбер. Рёбра графа, которые не принадлежат какому-нибудь кластеру, не используются для передачи сообщений, а переход из кластера в кластер осуществляется через общие узлы кластеров (множества их узлов могут пересекаться). Для того чтобы не было дублирования сообщений, требуется, чтобы граф кластеров был деревом.

В данной статье показано, что этим требованиям удовлетворяет блоковый граф, если блок (компоненту двусвязности) считать кластером. В блоковом графе каждый блок является кликой, то есть подграфом диаметра 1. В первую очередь узел опрашивает узлы тех кластеров, в которые он входит, потом более удалённые.

Известно, что многие реальные сети действительно кластеризованы похожим способом, поэтому данная работа имеет не только теоретическое, но и практическое значение. В то же время нужны дополнительные исследования, поскольку требование «кластер – это клика» слишком сильное и может не всегда выполняться. Во многих кластеризованных сетях кластер – это «почти» клика. Что значит это «почти», и как должна быть модифицирована модель, как раз и составляет тему возможных будущих исследований.

Список литературы / References

[1]. Бурдонов И.Б., Евтушенко Н.В., Косачев А.С., Пономаренко В.Н. Модель распределённой сети, в которой хосты могут выполнять функцию коммутации сообщений. Труды ИСП РАН, том 37, вып. 4, часть 1, 2025 г., стр. 7-30. DOI: 10.15514/ISPRAS–2025–37(4-1)–1.

[2]. Бурдонов И.Б., Евтушенко Н.В., Косачев А.С., Пономаренко В.Н. Кластеризация услуг распределённой сети, в которой хосты могут выполнять функцию коммутации сообщений. Труды ИСП РАН, том 37, вып. 5, 2025 г., стр. 7-32. DOI: 10.15514/ISPRAS–2025–37(5)–1.

[3]. Евин И.А. Введение в теорию сложных сетей. Компьютерные исследования и моделирование, 2010, т. 2, вып. 2, стр. 121-141.

[4]. Мир тесен (граф). Доступно по ссылке: https://ru.wikipedia.org/wiki/Мир_тесен_(граф), обращение: 15.04.2026.

[5]. Milgram S. The Small World Problem. Psychology Today, 1967.

[6]. Мир тесен. Доступно по ссылке: https://ru.wikipedia.org/wiki/Мир_тесен, обращение: 15.04.2026.

[7]. Карпов Д.В. Теория графов. МЦНМО, 2022. ISBN 978-5-4439-1690-3. Доступно по ссылке: https://www.klex.ru/2ad3, обращение: 15.04.2026.

[8]. Vušković K. Even-hole-free graphs: A survey // Applicable Analysis and Discrete Mathematics, 2010, vol. 4, no. 2, pp. 219-240. doi:10.2298/AADM100812027V. Доступно по ссылке: https://doiserbia.nb.rs/img/doi/1452-8630/2010/1452-86301000027V.pdf, обращение: 15.04.2026.

[9]. Блоковый граф. Доступно по ссылке: https://ru.wikipedia.org/wiki/Блоковый_граф, обращение: 15.04.2026.

[3] Это соответствует архитектуре SOA – service oriented architecture.

[10]. Hamelink R.C. A partial characterization of clique graphs. Journal of Combinatorial Theory, 1968, vol. 5, pp. 192-197, https://www.sciencedirect.com/science/article/pii/S0021980068800559?via%3Dihub, accessed: 15.04.2026. DOI: 10.1016/S0021-9800(68)80055-9.
[11]. Кликовый граф. Доступно по ссылке: https://ru.wikipedia.org/wiki/Кликовый_граф#cite_note-_4fa7c03ed9057c22-1, обращение: 15.04.2026.
[12]. Howorka E. On metric properties of certain clique graphs. Journal of Combinatorial Theory, Series B, 1979, vol. 27, no. 1, pp. 67-74. Available at: https://www.sciencedirect.com/science/article/pii/0095895679900698?via%3Dihub, accessed: 15.04.2026. DOI: 10.1016/0095-8956(79)90069-8.

Информация об авторах / Information about authors

Игорь Борисович БУРДОНОВ – доктор физико-математических наук, главный научный сотрудник ИСП РАН. Научные интересы: формальные спецификации, генерация тестов, технология компиляции, системы реального времени, операционные системы, объектно-ориентированное программирование, сетевые протоколы, процессы разработки программного обеспечения.

Igor Borisovich BURDONOV – Dr. Sci. (Phys.-Math.), a Leading Researcher of ISP RAS. Research interests: formal specifications, test generation, compilation technology, real-time systems, operating systems, object-oriented programming, network protocols, software development processes.

Нина Владимировна ЕВТУШЕНКО, доктор технических наук, профессор, главный научный сотрудник ИСП РАН, до 1991 года работала научным сотрудником в Сибирском физико-техническом институте. С 1991 г. работала в ТГУ профессором, зав. кафедрой, зав. лабораторией по компьютерным наукам. Её исследовательские интересы включают формальные методы, теорию автоматов, распределённые системы, протоколы и тестирование программного обеспечения.

Nina Vladimirovna YEVTUSHENKO, Dr. Sci. (Tech.), Professor, a Leading Researcher of ISP RAS, worked at the Siberian Scientific Institute of Physics and Technology as a researcher up to 1991. In 1991, she joined Tomsk State University as a professor and then worked as the chair head and the head of Computer Science laboratory. Her research interests include formal methods, automata theory, distributed systems, protocol and software testing.

Александр Сергеевич КОСАЧЕВ – кандидат физико-математических наук, ведущий научный сотрудник ИСП РАН. Научные интересы: формальные спецификации, генерация тестов, технология компиляции, системы реального времени, операционные системы, объектно-ориентированное программирование, сетевые протоколы, процессы разработки программного обеспечения.

Alexander Sergeevitch KOSSATCHEV – Cand. Sci. (Phys.-Math.), a Leading Researcher of ISP RAS. Research interests: formal specifications, test generation, compilation technology, real-time systems, operating systems, object-oriented programming, network protocols, software development processes.

Вера Николаевна ПОНОМАРЕНКО – кандидат физико-математических наук, старший научный сотрудник ИСП РАН. Научные интересы: формальные спецификации, генерация тестов, системы реального времени, операционные системы, объектно-ориентированное программирование, сетевые протоколы, процессы разработки программного обеспечения.

Vera Nikolaevna PONOMARENKO – Cand. Sci. (Phys.-Math.), a Senior Researcher of ISP RAS. Research interests: formal specifications, test generation, real-time systems, operating systems, object-oriented programming, network protocols, software development processes.

Приложение

Все имена записаны курсивом; мы различаем прописные и строчные буквы, а также курсив и полужирный курсив. Имена типов и вспомогательных процедур, реализация которых зависит от операционной среды, записаны ***строчными буквами полужирным курсивом***. Имена основных процедур и константы типов (кроме ***bool*** и ***unsigned***) ***начинаются с прописной буквы и записаны полужирным курсивом***. Имена глобальных переменных *начинаются с прописной буквы и записаны не полужирным курсивом*. Имена локальных переменных, полей структур и параметров процедур *начинаются со строчной буквы и записаны не полужирным курсивом*.

Обозначения: v_0 – начальный узел = корень дерева опросов,
$T(v_0)$ – дерево опросов с корнем в узле v_0 (высота дерева = эксцентриситет корня v_0),
$T(v_0, r)$ – поддерево дерева $T(v_0)$ с корнем в узле v_0 на раунде r (поддерево высотой r),
$T(v_i, r)$ – поддерево дерева $T(v_0, r)$ с корнем в узле v_i.

Узел дерева $T(v_0, r)$ на расстоянии i от корня обозначается как v_i. Для сообщения, пересылаемого по ребру ab, узел a будем называть отправителем, а узел b – получателем. Если получатель сообщения узел v_i, то отправитель узел v_{i-1} ($i \geq 1$) или v_{i+1} ($i \leq r$).

П1. Типы переменных и параметров процедур в узлах сети

bool – булевский тип,
unsigned – тип целого неотрицательного числа,
node – идентификатор узла сети,
service – имя услуги,
set(*type*) – множество элементов типа *type*,
list(*type*) – список элементов типа *type*,
parameters – параметры сообщений произвольного типа.

```
typedef struct                  // описание кластера в узле vi, принадлежащем кластеру
{   set ( node ) neighbors;     // все соседи узла vi в кластере, кроме узла vi
|   set ( node ) cutnodes;      // все соседние шарниры узла vi в кластере, кроме узла vi
} cluster;

enum nodetype       // тип узла дерева T(v0, r):
{   leaf,           // лист vl
|   internal,       // внутренний узел vi
|   root           // корень v0
};

enum status         // статус сеанса в нелистовом узле:
{   no,             // не получен ответ Yes и не получен ответ Wait
|   wait,           // не получен ответ Yes и получен ответ Wait
|   yes,            // внутренний узел: получен ответ Yes и не получено сообщение Request или Cancel
|                   // корень: получен ответ Yes и не получено сообщение Result
|   end              // внутренний узел: получено сообщение Request или Cancel
|                   // корень: получено сообщение Result
};

typedef void (* result)  // указатель на функцию обработки результата в корне v0
(   root,           // указатель на описание сеанса в корне v0
|   bool,           // false – в сети нет требуемой услуги, true – услуга оказана
|   parameters       // ответные параметры, если услуга оказана
);
```

```
typedef struct                  // описание сеанса в листе v_l дерева T(v_0, r), l = 1..r
{   nodetype   nodetype;        // тип сеанса leaf
|   union
|   {   internal *  parentsession;  // во время опроса: указатель на описание сеанса в родителе v_{l-1}
|   |   struct                      // во время оказания услуги:
|   |   {   root      *  rootsession;  // указатель на описание сеанса в корне v_0
|   |   |   list(node)  *  path;        // указатель на описание пути v_0...v_{l-1} от корня до родителя
}   }   } leaf;

typedef struct // описание сеанса во внутреннем узле v_i дерева T(v_0, r), i = 1..l - 1
{   nodetype      nodetype;     // тип сеанса internal
|   status        status;       // статус: no, yes, wait, end
|   unsigned      counter;      // число ожидаемых ответов
|   bool          cutnodeflag;  // true, если в дереве T(v_i, r) есть листовой шарнир v_r
|   unsigned      time;         // минимальное время ожидания для статуса wait
|   node          child;        // идентификатор сына v_{i+1} для статуса yes или wait
|   internal *    childsession; // указатель на описание сеанса в сыне v_{i+1} для статуса yes или wait
|   node          parent;       // идентификатор родителя v_{i-1}
|   internal *    parentsession; // указатель на описание сеанса в родителе v_{i-1}
} internal;

typedef struct                  // описание сеанса в корне v_0 дерева T(v_0, r)
{   nodetype      nodetype;     // тип сеанса root
|   status        status;       // статус: no, yes, wait, end
|   unsigned      counter;      // число ожидаемых ответов
|   bool          cutnodeflag;  // true, если в дереве T(v_i, r) есть листовой шарнир v_r
|   unsigned      time;         // минимальное время ожидания для статуса wait
|   node          child;        // идентификатор сына v_{i+1} для статуса yes или wait
|   internal *    childsession; // указатель на описание сеанса в сыне v_{i+1} для статуса yes или wait
|   unsigned      round;        // номер раунда
|   service       service;      // имя услуги
|   parameters *  inparameters; // указатель на параметры запроса услуги
|   result        result;       // указатель на функцию обработки результата
} root;

enum answer                     // ответ листа v_l:
{   no,                         // в этом узле нет запрашиваемой услуги
|   wait,                       // услуга может быть оказана через указанное время
|   yes                        // услуга может быть оказана немедленно
};

typedef struct                  // ответ листа v_l и время ожидания (для ответа wait)
{   answer answer;              // ответ листа: no, yes или wait
|   unsigned   time;            // время ожидания для ответа wait
} answertime;
```

П2. Константы типов (кроме *bool* и *unsigned*)

void * NULL – нулевой указатель,

node Notnode – идентификатор несуществующего узла сети.

П3. Статические глобальные переменные в узле сети

В каждом узле сети должна быть статическая информация о топологии сети, которая не меняется при передаче сообщений. Она может меняться только при изменении топологии сети: удалении/добавлении узла/кластера. Эта статическая информация описывает соседей узла, то есть узлы, которые соединены с данным узлом ребром. Статическая информация узла v_i состоит из идентификатора узла v_i и множества описаний кластеров, в которые входит узел v_i. Описание кластера узла v_i состоит из множества идентификаторов всех узлов кластера, кроме v_i, и подмножества идентификаторов шарниров. Шарнир принадлежит нескольким кластерам, узел, не являющийся шарниром, принадлежит ровно одному кластеру.

```
node Self;              // идентификатор данного узла v_i
set (cluster) Clusters; // множество кластеров, в которые входит узел v_i
```

П4. Вспомогательные процедуры

```
void * alloc (unsigned size) {...}; // выделение памяти указанного размера
void   free (void *   ptr)   {...}; // освобождение ранее выделенной памяти по указателю
```

П5. Процедуры в начальном узле v_0 (корне дерева $T(v_0)$)

```
root * // возвращается указатель на описание сеанса в корне v_0
RootCall // вызов запроса услуги в корне v_0 дерева T(v_0, r)
(   service      service,      // имя услуги
|   parameters * inparameters, // указатель на параметры запроса услуги
|   result       result       // указатель на функцию обработки результата
)
{   root * s = alloc(sizeof(root)); // создание сеанса в корне v_0
|   s→nodetype = root;
|   s→status = no;
|   s→round = 1;
|   s→service = service;
|   s→inparameters = inparameters;
|   s→result = result;
|   Broadcast(s, Notnode, service, 1); // рассылка Question, v_0 не изолированный узел
|   return s;
};
```

П6. Процедуры в листе v_l дерева $T(v_0, r)$

```
answertime LeafAnswer // ответ и время ожидания для ответа wait при получении сообщения Question
(   service      service      // имя услуги
) {...};
void LeafCancel       // отказ от услуги при получении сообщения Cancel
(   leaf *       s            // указатель на описание сеанса в листе v_l
) {...};
void LeafCall         // оказание услуги при получении сообщения Request
(   leaf *       s,           // указатель на описание сеанса в листе v_l
|   service      service,     // имя услуги
|   parameters   inparameters // параметры
) {...};
void LeafResult       // получение результата после вызова процедуры LeafCall в листе v_l
(   leaf *       s,           // указатель на описание сеанса в листе v_l
|   parameters   outparameters // ответные параметры
) // путь, запомненный в описании сеанса в листе v_l: s→path = v_0...v_l-1
```

```
{   path = s→path; // посылаем сообщение Result родителю v_{l-1}
|   SEND(Result(s→rootsession, path[1..|path| - 1], outparameters), path(|path|);
|   free (s); // освобождаем память из-под описания сеанса в листе v_l
};
```

П7. Процедура рассылки сообщения *Question* в нелистовом узле

```
Broadcast // рассылка сообщений Question из нелистового узла v_i на расстоянии от корня v_0 < раунда r
(   internal * s,              // указатель на описание сеанса в узле v_i
|   node       parent,        // для i = 1..l родитель v_{i-1}, для i = 0 (для корня v_0) Notnode
|   service    service,       // имя услуги
|   unsigned   rounddistance  // разность (r - i) между раундом и расстоянием узла v_i от корня v_0
)
{   s→counter = 0;
|   s→cutnodeflag = false;
|   if (rounddistance == 1) // v_i пограничный узел дерева T(v_0, r), то есть i = r - 1
|   {   for (∀ cluster ∈ Clusters) // для всех кластеров, в которые входит узел v_i
|   |   {   if (parent ∉ cluster.neighbors) { // и не входит узел parent,
|   |   |   {   s→counter = s→counter + |cluster.neighbors|; // счётчик ожидаемых ответов
|   |   |   |   for (∀ neighbor ∈ cluster.neighbors) // для всех соседних с v_i листьев в кластере
|   |   |   |   {   SEND(Question(service, rounddistance - 1, Self, s), neighbor); // опрос листьев
|   |   |   |   |   return true;
|   }   }   }   }
|   else // v_i не пограничный узел дерева T(v_0, r), то есть i < r - 1
|   {   for ( ∀ cluster ∈ Clusters ) // для всех кластеров, в которые входит узел v_i
|   |   {   if (parent ∉ cluster.neighbors) // и не входит узел parent,
|   |   |   {   s→counter = s→counter + |cluster.cutnodes|; // счётчик ожидаемых ответов
|   |   |   |   for ( ∀ cutnode ∈ cluster.cutnodes) // для всех соседних с v_i шарниров в кластере
|   |   |   |   {   SEND(Question(service, rounddistance - 1, Self, s), cutnode); // опрос шаниров
}   }   }   }   };
```

П8. Процедуры обработки сообщений в узле

```
void   Question // опрос услуги, отправитель v_{i-1}, получатель v_i, i = 1..l, l ≤ r
(   service    service,        // имя услуги
|   unsigned   rounddistance,  // разность между раундом и расстоянием от корня v_0 для узла v_i
|   node       sender,         // идентификатор отправителя v_{i-1}
|   internal * sendersession   // указатель на описание сеанса в отправителе v_{i-1}
)
{   if (rounddistance > 0 & |Clusters| > 1) // узел v_i шарнир и расстояние узла v_i от корня v_0 < раунда r
|   {   if (rounddistance == 1 ∨ ∃ cluster ∈ Clusters parent ∉ cluster.neighbors & cluster.cutnodes ≠ ∅)
|   |   |   // узел v_i внутренний узел дерева T(v_0, r)
|   |   {   internal * s = alloc(sizeof(internal)); // создаём сеанс в узле v_i
|   |   |   s→nodetype = internal;
|   |   |   s→status = no;
|   |   |   s→parent = sender;
|   |   |   s→parentsession = sendersession;
|   |   |   Broadcast(s, sender, service, rounddistance); // рассылка Question
|   |   }
```

```
    else // узел v_i лист дерева T(v_0, r)
    {   SEND(No(sendersession, false), sender); // нечего опрашивать в дереве T(v_i, r)
    } }
  else // узел v_i не шарнир или расстояние узла v_i от корня v_0 равно раунду r; лист v_r дерева T(v_0, r)
  {   cutnodeflag = (|Clusters| > 1); // узел v_i шарнир, если принадлежит нескольким кластерам
      answertime answertime = LeafAnswer(service) // ответ и время ожидания для ответа wait
      switch (answertime.answer) // посылаем ответ родительскому узлу v_{i-1}
      {   case no:   SEND(No(sendersession, cutnodeflag), sender); break;
          case wait: leaf * s = alloc(sizeof(leaf)); // создаём сеанс в листе v_l
                     s→nodetype= leaf;
                     s→parentsession = sendersession;
                     SEND(Wait(sendersession, Self, s, cutnodeflag, answertime.time), sender);
                     break;
          case yes:  leaf * s = alloc(sizeof(leaf)); // создаём сеанс в листе v_i
                     s→nodetype= leaf;
                     s→parentsession = sendersession;
                     SEND(Yes(sendersession, Self, s), sender);
} } };
void No // отрицательный ответ на опрос, отправитель v_{i+1}, получатель v_i, i = 0..l - 1, l ≤ r
(   internal  *  s,            // указатель на описание сеанса в получателе v_i
    bool         cutnodeflag   // true, если в дереве T(v_{i+1}, r) есть листовой шарнир v_r
)
{ s→cutnodeflag = s→cutnodeflag ∨ cutnodeflag; // true, если в дереве T(v_i, r) есть листовой шарнир v_r
  s→counter = s→counter - 1; // уменьшаем счётчик ожидаемых ответов
  if (s→counter == 0) // получены все ответы:
  {   if (s→nodetype == internal)    // узел v_i это внутренний узел дерева T(v_0, r), то есть i > 0:
      {   switch (s→status) // посылаем ответ родителю v_{i-1}, если статус no или wait
          {   case no:   SEND(No(s→parentsession, s→cutnodeflag), s→parent);
                         free(s); // освобождаем память из-под описания сеанса
                         break;
              case wait: SEND(Wait(s→parentsession, Self, s, s→cutnodeflag, s→time), s→parent);
                         break;
              case end:  free(s); // освобождаем память из-под описания сеанса
          } }
      else // узел v_i это корень v_0 дерева T(v_0, r), то есть i = 0:
      {   roots = (root *) s;
          switch (roots→status) // решаем, что делать дальше
          {   case no:   if (roots→cutnodeflag) // раунд r меньше высоты дерева опроса T(v_0):
                         {   roots→round = roots→round + 1; // увеличиваем номер раунда
                             Broadcast(s, Notnode, roots→service, roots→round); // рассылка Question
                         }
                         else // раунд r равен высоте дерева опроса T(v_0), опросили всё дерево T(v_0):
                         {   roots→result(roots, false, 0); // результат: услуги нет в сети
                             free(roots); // освобождаем память сеанса
                         }
                         break;
              case wait: if (roots→cutnodeflag) // раунд r меньше высоты дерева опроса T(v_0):
```

```
|   |   |   |   |           {   roots→round = roots→round + 1; // увеличиваем номер раунда
|   |   |   |   |           |   Broadcast(s, Notnode, roots→service, roots→round); // рассылка Question
|   |   |   |   |           }
|   |   |   |   |           else // раунд r равен высоте дерева опроса T(v0), опросили всё дерево T(v0):
|   |   |   |   |           |   SEND(Request(roots→childsession, roots, roots→service, Self,
|   |   |   |   |           |                roots→inparameters), roots→child); // посылаем запрос услуги
|   |   |   |   |           } break;
|   |   |   |   case end:       free(s); // освобождаем память из-под описания сеанса
}   }   }   };
void Wait // ответ на опрос: «жди», отправитель v(i+1), получатель vi, i = 0..l - 1, l ≤ r
(   internal *  s,              // указатель на описание сеанса в получателе vi
|   node        sender,         // идентификатор отправителя v(i+1)
|   internal *  sendersession,  // указатель на сеанс в отправителе v(i+1)
|   bool        cutnodeflag,    // true, если в дереве T(v(i+1), r) есть листовой шарнир vr
|   unsigned    time            // предполагаемое время ожидания
)
{   s→cutnodeflag = s→cutnodeflag ∨ cutnodeflag; // true, если в дереве T(vi, r) есть листовой шарнир vr
|   s→counter = s→counter - 1;      // уменьшаем счётчик ожидаемых ответов
|   switch (s→status)               // меняем статус и время ожидания, если нужно
|   {   case no:    s→status = wait;   // меняем статус no → wait
|   |   |           s→time = time;     // запоминаем время ожидания
|   |   |           s→child = sender; // запоминаем сына v(i+1)
|   |   |           s→childsession = sendersession; // и указатель на описание сеанса в сыне v(i+1)
|   |   |           break;
|   |   case wait:  if (time < s→time) // полученное время ожидания меньше запомненного
|   |   |           {   SEND(Cancel(s→childsession), s→child); // отменяем предыдущий ответ Wait
|   |   |           |   s→time = time;     // запоминаем время ожидания
|   |   |           |   s→child = sender;  // запоминаем сына v(i+1)
|   |   |           |   s→childsession = sendersession; // и указатель на описание сеанса в сыне v(i+1)
|   |   |           }
|   }
|   if (s→counter == 0)             // получены все ответы:
|   {   if (s→nodetype == internal) // узел vi это внутренний узел дерева T(v0, r), то есть i > 0:
|   |   {   switch (s→status)       // решаем, что делать дальше
|   |   |   {   case wait:  SEND(Wait(s→parentsession, Self, s, s→cutnodeflag, s→time), s→parent);
|   |   |   |   |           break;
|   |   |   |   case end:   free(s);     // освобождаем память из-под описания сеанса
|   |   |   }   }
|   |   else                        // узел vi это корень v0 дерева T(v0, r), то есть i = 0:
|   |   {   roots = (root *) s;
|   |   |   switch (roots→status)   // решаем, что делать дальше
|   |   |   {   case wait:  if (roots→cutnodeflag) // раунд r меньше высоты дерева опроса T(v0):
|   |   |   |   |           {   roots→round = roots→round + 1; // увеличиваем номер раунда
|   |   |   |   |           |   Broadcast(s, Self, roots→service, roots→round); // рассылка Question
|   |   |   |   |           }
|   |   |   |   |           else // раунд r равен высоте опроса дерева T(v0), опросили всё дерево T(v0):
|   |   |   |   |           {   SEND(Request(roots→childsession, roots, roots→service, Self,
|   |   |   |   |           |                roots→inparameters), roots→child); // посылаем запрос услуги
```

```
|   |   |   |                   }   break;
|   |   |   |   case end:   free(s); // освобождаем память из-под описания сеанса
}   }   }   };
void Yes // положительный ответ на опрос, отправитель v_{i+1}, получатель v_i, i = 0..l - 1, l ≤ r
(   internal *  s,              // указатель на описание сеанса в получателе v_i
|   node        sender,         // идентификатор отправителя v_{i+1}
|   internal *  sendersession   // указатель на сеанс в отправителе v_{i+1}
)
{   s→counter = s→counter - 1; // уменьшаем счётчик ожидаемых ответов
|   switch (s→status) // меняем статус и время ожидания, если нужно
|   {   case wait:  SEND(Cancel(s→childsession), s→child); // отменяем предыдущий ответ Wait
|   |   case no:    s→status = yes; // меняем статус no → yes
|   |   |           s→child = sender; // запоминаем сына v_{i+1}
|   |   |           s→childsession = sendersession; // и указатель на описание сеанса в сыне v_{i+1}
|   |   |           if (s→nodetype == internal) // v_i внутренний узел дерева T(v_0, r), то есть i > 0:
|   |   |           {   SEND(Yes(s→parentsession, Self, s) s→parent); // ответ Yes родителю v_{i-1}
|   |   |           }
|   |   |           else // v_i корень v_0 дерева T(v_0, r), то есть i = 0:
|   |   |           {   roots = (root *) s;
|   |   |           |   SEND(Request( // посылаем сообщение Request
|   |   |           |           sendersession, roots, roots→service, Self, roots→inparameters), sender);
|   |   |           } break;
|   |   case yes:   SEND(Cancel(sendersession), sender); // отменяем повторный ответ Yes
|   |   |           break;
|   |   case end:   if (s→counter == 0)     // если получены все ответы
|   |   |           {   free(s);            // освобождаем память из-под описания сеанса
|   |   |           }
}   };
void    Cancel // отмена ответа Wait или Yes, отправитель v_{i-1}, получатель v_i, i = 1..l, l ≤ r
(   internal *  s   // указатель на описание сеанса в получателе v_i
)
{   if (s→nodetype == internal)   // узел v_i это внутренний узел дерева T(v_0, r):
|   {   if (s→counter == 0)       // получены все ответы:
|   |   {   free(s);              // освобождаем память из-под описания сеанса
|   |   }   else                  // не все ответы получены
|   |   {   s→status = end;       // будем ждать получения всех ответов
|   |   |   SEND(Cancel(s→childsession), child); // отменяем ответ Wait или Yes от сына
|   }   }
|   else // узел v_i это лист v_l дерева T(v_0, r):
|   {   LeafCancel((leaf *) s);
|   |   free(s); // освобождаем память из-под описания сеанса
}   };
void Request // запрос услуги, отправитель v_{i-1}, получатель v_i, i = 1..l, l ≤ r
(   void *          s,              // указатель на описание сеанса в получателе v_i
|   root *          rootsession,    // указатель на описание сеанса в корне v_0
|   service         service,        // имя услуги
|   list(node)      path,           // путь v_0...v_{i-1} от корня v_0 до отправителя v_{i-1}
```

```
|   parameters    inparameters   // параметры
)
{   if (s→nodetype == internal) // узел v_i это внутренний узел: посылаем сообщение Request узлу v_{i+1}
|   {   SEND(Request(s→childsession, rootsession, service, path + Self, inparameters), s→child);
|   |   if (s→counter ≠ 0) // получены не все ответы:
|   |   {   s→status == end; // будем ждать получения всех ответов
|   |   else // получены все ответы
|   |   {   free(s); // освобождаем память из-под описания сеанса
|   |   }
|   else // узел v_i это лист v_l дерева T(v_0, r):
|   {   leafs = (leaf *) s;
|   |   leafs→rootsession= rootsession;
|   |   leafs→path = (list(node) * )alloc(|path|);
|   |   * leafs→path = path;
|   |   LeafCall(leafs, service, inparameters); // вызываем процедуру оказания услуги
}   };
void Result = { // ответ на запрос, отправитель v_{i+1}, получатель v_i, i = 0..l - 1, l ≤ r
(   root *          rootsession,        // указатель на описание сеанса в корне
|   list(node)      path,               // путь v_0...v_{i-1} для i = 1..l - 1, пустой путь для i = 0
|   parameters      outparameters       // ответные параметры
)
{   if (|path| ≠ 0) // узел v_i это внутренний узел дерева T(v_0, r): пересылаем сообщение Result узлу v_{i-1}
|   {   SEND(Result(rootsession, path[1..|path| - 1], outparameters), path(|path|);
|   }
|   else // узел v_i это корень v_0 дерева T(v_0, r): передаём ответные параметры в обработку результата
|   {   rootsession→result(rootsession, true, outparameters);
|   |   if (rootsession→counter ≠ 0) // получены не все ответы:
|   |   {   rootsession→status== end; // будем ждать получения всех ответов
|   |   else // получены все ответы
|   |   {   free(rootsession); // освобождаем память из-под описания сеанса
}   }   };
```