DOI: 10.15514/ISPRAS-2025-37(4-1)-1

Модель распределённой сети, в которой хосты могут выполнять функцию коммутации сообщений

[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 работе строится абстрактная модель распределенной сети, содержащей только хосты и коммутаторы, которая позволяет оценить классы задач, которые необходимо решать в такой сети, принимая во внимание, в том числе, нефункциональные параметры. Предполагается, что хосты предлагают пакеты определенных услуг (сервисов) и сообщения (запросы) между хостами пересылаются через промежуточные узлы по правилам коммутации. Правило определяет, каким соседним узлам пересылается принятое узлом сообщение в зависимости от того, откуда оно пришло, и от вектора параметров в его заголовке. Соответственно, настройка узлов определяет множество путей от хоста к хосту, по которым будут пересылаться пакеты. Ситуация моделируется с использованием графа, вершинами которого являются хосты и коммутаторы, а ребра соответствуют физическим связям между ними. Обычно предполагается, что в такой сети хосты только принимают, обрабатывают и посылают информацию другим хостам, но не занимаются коммутацией сообщений, эта функция возлагается на другие узлы – коммутаторы, но мы предполагаем, что при современных технологиях хост также может выполнять функции коммутации сообщений, то есть такой хост (как и коммутатор) содержит систему правил коммутации, указывающих, куда отправляется полученное сообщение, если почему-либо данный хост не может обработать данный запрос/сообщение. Предлагается модель сети, в которой функцию коммутации сообщений выполняет не только каждый коммутатор, но и каждый хост. Обсуждаются проблемы, связанные с нефункциональными параметрами распределенной сети, а именно, *достижимость*/*недостижимость* хостов, *зацикливание* сообщений, *перегрузка* сети сообщениями, *немасштабируемость*. Обсуждается возможность оптимизации рассматриваемых параметров сети на основе использования информации об услугах/сервисах, представляемых каждым из хостов, и алгоритмы самонастройки распределенной сети, оптимизирующие параметры сети, передачу сообщений по настроенной сети и инкрементальную (повторную частичную) настройку сети, не нарушающую функционирование сети.

Ключевые слова: распределенная сеть; хосты; коммутаторы; достижимость; зацикливание; перегрузка; масштабируемость.

Для цитирования: 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.

A distributed network model in which hosts can perform message switching functions

[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 the paper, an abstract model of a distributed network containing only hosts and switches is developed; this model allows us to estimate some problems that need to be solved in such a network, taking into account, among other things, non-functional parameters. It is assumed that hosts offer packages of certain services, and messages (requests) between hosts are forwarded via intermediate nodes according to switching rules. A rule determines which neighboring nodes a message received by a node is forwarded to, depending on where the message came from and the parameter vector in its header. Accordingly, the configuration of nodes determines the set of paths from host to host along which packets will be forwarded. The situation is modeled using a graph of physical connections, the vertices of which are hosts and switches, and the edges correspond to the physical connections between them. It is usually assumed that only hosts in such a network receive, process, and send information to other hosts, but do not switch messages. This function is assigned to other nodes, switches. However, we assume that with modern technologies, in some cases a host can perform the switching functions, i.e. such a host (like a switch) contains a system of switching rules that specify where the received message is sent if for some reason the host cannot process the obtained request/message. A network model is proposed in which the message switching function is performed not only by each switch, but also by each host. The paper discusses the problems associated with non-functional parameters of a distributed network, namely, reachability/unreachability of hosts, message looping, network overload with messages, and non-scalability. Optimization issues of the network parameters are discussed based on the set of services provided by each host. In addition, we discuss self-tuning algorithms for a distributed network that optimize network parameters, message transmission over the configured network and incremental (repeated partial) self-tuning algorithms when changing network parameters, in particular, its topology that does not disrupt network operation.

Keywords: distributed network; hosts; switches; reachability; looping; overload; scalability.

For citation: Burdonov I.B., Yevtushenko N.V., Kossatchev A.S., Ponomarenko V. N. A distributed network model in which hosts can perform message switching functions. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 4, part 1, 2025, pp. 7-30 (in Russian). DOI: 10.15514/ISPRAS-2025-37(4-1)-1.

1. Введение

В распределенных сетях (distributed networks) отсутствует центральное управление для пересылки и обработки информации, т.е. каждый элемент такой сети обладает полной автономией [1]. В этой области есть большое количество публикаций, и в большинстве работ [см. например, 1, 2] рассматриваются сети, используемые для передачи данных в специальных условиях и для специальных целей, содержащие ряд различных специальных устройств. В некоторых случаях узлы современной сети делятся на два класса: хосты и коммутаторы, и в качестве примера таких сетей можно рассматривать программно-конфигурируемые сети (*software-defined networking*, SDN) [3-8]. В настоящей работе мы строим абстрактную модель распределенной сети, содержащей только хосты и коммутаторы, которая позволяет оценить классы задач, которые необходимо решать в такой сети, принимая во внимание, в том числе, нефункциональные параметры. Мы предполагаем, что хосты предлагают пакеты определенных услуг (сервисов) и сообщения (запросы) между хостами

пересылаются через промежуточные узлы (коммутаторы). В данной работе, так же, как и в большинстве программно-конфигурируемых сетей коммутатор работает по правилам коммутации. Правило определяет, каким соседним узлам пересылается принятое коммутатором сообщение в зависимости от того, откуда оно пришло, и от вектора параметров в его заголовке [6-8]. Соответственно, настройка коммутаторов определяет множество путей от хоста к хосту, по которым будут пересылаться пакеты. Ситуация моделируется с использованием графа физических связей, вершинами которого являются хосты и коммутаторы, а ребра соответствуют физическим связям между ними. Иногда предполагается, что каждый хост может быть соединен только с одним коммутатором [6-8]. В общем случае хосты в такой сети принимают, обрабатывают и посылают информацию другим хостам, но мы предполагаем, что при современных технологиях хост может выполнять также функции коммутации сообщений, т.е. такой хост (как и коммутатор) содержит систему правил коммутации, указывающих, куда отправить полученный запрос/сообщение, если почему-либо данный хост не может его обработать. В программно-конфигурируемых сетях настройка правил коммутации обычно осуществляется специальными компонентами сети, например, SDN-контроллерами, но в ряде случаев возможна и самонастройка коммутаторов, в зависимости от передаваемых запросов, а также инкрементальная (повторная и частичная) настройка при изменении параметров сети, в частности, её топологии.

Структура работы следующая. В разделе 2 обсуждаются проблемы, связанные с нефункциональными параметрами распределенной сети: *достижимость*/*недостижимость* хостов, *зацикливание* сообщений, *перегрузка* сети сообщениями, *немасштабируемость*. В разделе 3 описывается предлагаемая модель распределённой сети, а в разделе 4 рассматривается передача сообщений по сети известным хостам-получателям. В разделе 5 обсуждается возможность оптимизации рассматриваемых параметров сети на основе использования информации об услугах/сервисах, предоставляемых каждым из хостов. В разделе 6 рассматривается настройка распределенной сети, а в разделе 7 – проблемы инкрементальной (повторной и частичной) настройки функционирующей сети (когда в ней продолжают циркулировать сообщения) при различных изменениях параметров сети. В разделах 3-7 описана идея предлагаемых алгоритмов; сами алгоритмы вынесены в приложение.

2. Четыре проблемы распределённых сетей

В качестве модели распределённой сети (далее, просто сети) рассматривается граф, в котором вершины – это хосты и коммутаторы, а рёбра – каналы связи, по которым передаются сообщения. В настоящей статье мы будем считать каналы связи дуплексными, т.е. сообщения могут передавать по ребру в обоих направлениях, поэтому граф считается неориентированным. Ребру $\{a, b\}$ соответствуют две ориентированные дуги ab и ba, вершины a и b называются *соседними*. Считается, что в графе нет кратных рёбер и петель. В этом случае путь в графе однозначно определяется как последовательность вершин. Если граф не связный, то сеть, фактически, распадается на независимые подсети, соответствующие компонентам связности, которые функционируют независимо друг от друга. В дальнейшем будем считать, что граф связный. Мы будем использовать слова «вершина» (графа) и «узел» (сети) как синонимы.

Под хостом понимается узел сети, который может обрабатывать информацию, например, выполнять вычислительные функции. Именно хост генерирует сообщения, передаваемые далее по сети, и именно хост является конечным получателем сообщений.

Коммутаторы выполняют функцию коммутации сообщений: приняв сообщение от соседней вершины a, коммутатор s пересылает его другому (или тому же самому) своему соседу b. В этом случае будем называть соседа a *предшественником*, а соседа b *преемником*

коммутатора s. Мы предполагаем, что само сообщение при этом не меняется. Коммутатор работает по правилам коммутации, которые хранятся в его памяти. Каждое правило коммутатора s в общем случае имеет вид: $(p: a, s, b)$, где a – сосед-предшественник, b – сосед-преемник, p – параметры коммутации, т.е. та часть параметров сообщения, которая определяет выбор того или иного правила. Если сообщение принято коммутатором s от предшественника a и параметры сообщения соответствуют параметрам коммутации p в правиле вида $(p: a, s, b)$, то это правило срабатывает, и сообщение будет направлено соседу b.

Если есть несколько правил, отличающихся друг от друга только преемником; предполагается, что все такие правила срабатывают. Тем самым, если срабатывают несколько правил, то сообщение *клонируется*, и каждый клон посылается соответствующему преемнику. При клонировании сообщение (его клоны) могут получить несколько хостов, поэтому, если сообщение предназначено не для всех этих хостов, например, для одного хоста или определённой группы хостов, то остальные хосты, получив такое сообщение, должны его игнорировать.

Замечание. В правиле коммутации $(p: a, s, b)$ указаны вершины графа, точнее, идентификаторы этих вершин. В реализации можно было бы обойтись без идентификаторов вершин, если считать, что все рёбра, инцидентные вершине s, линейно упорядочены, например, пронумерованы от 1 до $deg(s)$, где $deg(s)$ степень вершины s (число её соседей). Когда вершина s получает сообщение от предшественника a, ей должен автоматически сообщаться номер $i(a, s)$ её ребра $\{a, s\}$, по которому получено сообщение. Когда вершина s посылает сообщение преемнику b, она указывает номер $i(s, b)$ её ребра $\{s, b\}$, по которому посылается сообщение. Правило коммутации вместо $(p: a, s, b)$ имело бы вид $(p: i(a, s), s, i(s, b))$. С другой стороны, идентификатор вершины-хоста необходим как параметр сообщений, которые посылаются конкретному хосту-получателю. Поэтому для удобства изложения мы будем считать, что заданы идентификаторы вершин, как описано выше, а правила коммутации имеют вид $(p: a, s, b)$. В то же время, учитывая такую реализацию с номерами рёбер, будем считать, что когда вершина s получает сообщение от предшественника a, идентификатор предшественника a является не параметром сообщения, а ответным параметром оператора приёма сообщения в s. Когда вершина s посылает сообщение преемнику b, идентификатор преемника b является не параметром сообщения, а параметром оператора посылки сообщения.

Правила коммутации порождают пути в графе, по которым двигаются сообщения. Путь $a_1, a_2, ..., a_n$, где a_1 и a_n хосты, а остальные вершины коммутаторы, порождается правилами $(p: a_1, a_2, a_3)$, $(p: a_2, a_3, a_4)$, ..., $(p: a_{n-2}, a_{n-1}, a_n)$.

С такой распределённой сетью связан ряд проблем: *недостижимость* хостов, *зацикливание* сообщений, *перегрузка* сети сообщениями, *немасштабируемость*.

Достижимость хостов означает, что для каждой упорядоченной пары хостов (a, b) правила коммутации позволяют переслать сообщение из хоста a в хост b. Соответственно, проблема недостижимости возникает, когда это требование не выполняется.

Зацикливание сообщений возникает тогда, когда правила коммутации определяют путь, проходящий дважды по одной и той же дуге. В этом случае отрезок пути, начинающийся и заканчивающийся этой дугой, является ориентированным циклом, по которому сообщение будет двигаться бесконечно. Путь, не проходящий дважды по одной дуге, называется *простым по дугам*. Таким образом, для того чтобы не было зацикливания, все порождаемые пути должны быть простыми по дугам.

Перегрузка сети может возникнуть из-за клонирования сообщений, когда в сети циркулирует слишком много сообщений. Наилучшим решением было бы вообще отказаться от излишнего клонирования, когда число клонов больше числа хостов, которым сообщение предназначено.

Масштабируемость означает, что суммарное (во всех коммутаторах) число правил коммутации не более, чем линейно, зависит от числа вершин графа. Иными словами, число

правил в одном коммутаторе ограничено числом, не зависящим от числа вершин графа. Например, при отсутствии параметров коммутации, когда правило коммутации имеет вид $(:a, s, b)$, число правил в коммутаторе s не превосходит $deg(s)^2$, и не зависит от общего числа вершин в графе. В идеале в масштабируемой распределённой сети добавление в граф новых вершин и рёбер требует изменения/добавления правил только в тех «старых» коммутаторах, которым инцидентны новые рёбра. На практике это обычно не так, но в целом число изменяемых правил не должно зависеть от общего числа вершин графа.

На самом деле, проблемы, описанные выше, тесно связаны между собой, причём некоторые решения одних проблем усугубляют другие проблемы. Проиллюстрируем это простейшими примерами решения этих проблем.

Достижимость можно обеспечить просто клонированием сообщения в коммутаторе так, чтобы оно пересылалось всем соседям, кроме того, от которого получено. Сообщение с гарантией достигнет нужного хоста-получателя (поскольку граф связен). Однако, во-первых, это может привести к зацикливанию, если в графе есть цикл, а, во-вторых, и как следствие, к перегрузке сети.

Вместо этого можно посылать сообщение из хоста-отправителя по остовному дереву, ориентированному от корня, которым этот отправитель и будет. Сообщение будет клонироваться в точках разветвления дерева. Зацикливания не будет, однако проблема перегрузки всё равно остаётся: вместо одного сообщения будет число сообщений, равное числу листовых вершин остовного дерева. Кроме того, в этом случае нам нужно различать сообщения, сгенерированные разными хостами-отправителями, т.е. идентификатор хоста-отправителя будет частью параметров коммутации, что приведёт к немасштабируемости сети.

Можно посылать сообщение по остовному дереву, ориентированному к корню, которым будет хост-получатель. Зацикливания не будет, и сообщение не будет клонироваться, и поэтому не будет перегрузки. Но проблема немасштабируемости остаётся: идентификатор хоста-получателя будет частью параметров коммутации.

Наилучшим (из простейших) решений было бы выделение в графе ориентированного простого по дугам цикла, который содержал бы все хосты. Сообщение пересылается по этому циклу до нужного хоста. Цикл $a_1, a_2, ..., a_n, a_1$ порождается правилами коммутации $(:a_1, a_2, a_3)$, $(:a_2, a_3, a_4)$, ..., $(:a_{n-2}, a_{n-1}, a_n)$, $(:a_{n-1}, a_n, a_1)$, т.е. параметры коммутации отсутствуют. В этом случае можно решить все четыре проблемы.

Но для этого нужно, чтобы хосты могли выполнять, кроме вычислительных функций, также и функцию коммутации. Во многих распределённых сетях, например, в программно-конфигурируемых сетях (SDN), как правило, эти функции не совмещаются: вершина графа – либо хост, либо коммутатор, но не то и другое вместе. Более того, часто требуется, чтобы хост был терминальной вершиной графа, т.е. был подключён только к одному коммутатору.

Это приводит к целому ряду проблем. Например, если в графе есть мост, с каждой стороны которого есть несколько хостов, то невозможно одновременно решить проблемы недостижимости, перегрузки и немасштабируемости. Для того чтобы обеспечить достижимость, либо сообщения, проходящие по мосту в одном направлении, должны клонироваться после моста, чтобы попасть во все хосты с другой стороны моста, либо мы должны различать в правилах коммутации сообщения, направляемые разных хостам. В первом случае у нас будет перегрузка: создаются «лишние» клоны сообщений для хостов, которым они не предназначены. А во втором случае будет немасштабируемость: идентификатор хоста-получателя становится частью параметров коммутации. Эти проблемы пытаются решить, допуская клонирование, но в ограниченном масштабе, когда проблема перегрузки решается лишь частично.

3. Модель распределённой сети

Авторами предлагается модель распределённой сети, в которой не только коммутаторы, но и хосты выполняют функцию коммутации сообщений. Как будет показано далее, это помогает решить описанные выше проблемы вполне удовлетворительно. В дальнейшем будем считать, что все вершины графа делятся на хосты, которые выполняют как вычислительные функции, так и функцию коммутации сообщений, и коммутаторы, которые выполняют только функцию коммутации.

В этом и следующем разделах мы рассмотрим функционирование такой сети, когда выполнена её настройка, т.е., прежде всего, установлены нужные правила коммутации в вершинах графа, а также инициализированы другие переменные вершин, необходимые для передачи сообщений или для инкрементальной (дополнительной частичной) настройки. При этом мы будем иметь в виду, что в процессе такого функционирования граф может изменяться и сеть может перенастраиваться. В разделе 6 мы рассмотрим самонастройку сети, а в разделе 7 – различные виды изменений сети и требуемую при этих изменений перенастройку сети.

Будем считать, что каждое сообщение в сети имеет тип (имя) и набор параметров. Мы будем опираться на следующее требование неизменности сообщения: при пересылке сообщения по сети меняться может только тип сообщения, параметры остаются неизменными. При этом число типов сообщения ограничено и фиксировано; в предлагаемых ниже алгоритмах используется 9 типов сообщений.

Прежде всего, заметим, что ориентированный простой по дугам цикл, содержащий все хосты, о котором шла речь в предыдущем разделе, всегда существует в связном графе. Таким циклом является цикл обхода остовного дерева графа, например, по алгоритму Тэрри [9], который проходит каждое ребро два раза, но по одному разу в каждом направлении, т.е. по одному разу по каждой дуге (на рис. 1.*a* обход остовного дерева показан синим цветом).

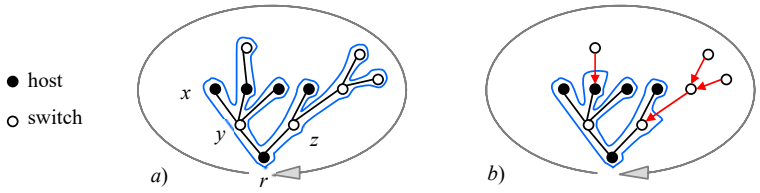

Рис. 1. a) Обход остовного дерева, b) Обход дерева хостов.
Fig. 1. a) Spanning tree traversal, b) Host tree traversal.

Для данного корневого дерева его обход, начиная с корня, по алгоритму Тэрри однозначно определяется линейным порядком соседей вершины, заданным для каждой вершины: соседи вершины проходятся алгоритмом в этом порядке. *Родителем некорневой вершины x* будем называть такую вершину y, что дуга xy последняя на пути от корня до вершины x; в алгоритме Тэрри по этой дуге мы первый раз попадаем в вершину x. *Родителем корня r* будем условно называть его последнего соседа z в линейном порядке соседей корня; в алгоритме Тэрри дуга zr проходится последней.

В дальнейшем будем считать, что корень остовного дерева является хостом. Если остовное дерево содержит листовые коммутаторы (не хосты), то они, очевидно, являются «лишними» при коммутации сообщений, и их можно удалить. Минимальное поддерево остовного дерева с тем же корнем, которое содержит все хосты, будем для краткости называть *деревом хостов*, нужно только иметь в виду, что кроме хостов оно содержит также коммутаторы, необходимые для связности дерева. Эквивалентное определение: *деревом хостов* называется поддерево остовного дерева с тем же корнем, которое содержит все хосты, и все его листья

являются хостами. Соответственно, вместо цикла обхода остовного дерева будем использовать цикл обхода дерева хостов, который для краткости будем называть *циклом хостов*. Там, где это не приводит к двусмысленности, корень дерева хостов будем для краткости называть просто *корнем*.

Дерево хостов строится процедурой «Удаление "лишних" коммутаторов». Первоначально дерево хостов совпадает с остовным деревом. Если коммутатор является листом дерева, он удаляется вместе с единственным инцидентным ему ребром дерева. Процедура повторяется до тех пор, пока все листья дерева не будут хостами. Это и будет дерево хостов, обход которого является циклом хостов (на рис. 1.*b* обход хостов показан синим цветом).

Рассмотрим ту часть остовного дерева, которая не вошла в дерево хостов. Она представляет собой лес корневых деревьев, все вершины которых, кроме, быть может, их корней, являются коммутаторами и не лежат на дереве хостов, а их корни лежат на дереве хостов. Будем называть эти деревья *деревьями коммутаторов*. Два дерева коммутаторов не имеют общих вершин. Для передачи сообщений по сети деревья коммутаторов не нужны, так как все хосты (а только они генерируют сообщения) находятся на цикле хостов, и сгенерированные сообщения двигаются по циклу хостов и не попадают в коммутаторы, не являющиеся корнями деревьев коммутаторов. Однако лес деревьев коммутаторов может быть нужен, когда происходит изменение графа сети. Например, к сети может быть добавлен новый хост, подсоединяемый к коммутатору на дереве коммутаторов. Поэтому будем считать, что правила коммутации порождают, кроме цикла хостов, лес деревьев коммутаторов, ориентированных к своим корням (на рис. 1.*b* отмечены красным цветом).

Мы будем считать, что в настроенной сети в каждой вершине инициализированы следующие переменные: *Self* – собственный идентификатор вершины, *Host* – отметка хоста, *Rules* – правила коммутации, *Root* – отметка корня дерева хостов, где «отметка» – булевская переменная.

4. Передача сообщений известному хосту-получателю

Для отправки сообщения известному хосту-получателю используется сообщение типа ***MessageToHost*** с параметрами: *sender* – идентификатор хоста-отправителя, *recipient* – идентификатор хоста, которому предназначено сообщение, *parameters* – параметры сообщения, прозрачные для коммутации сообщений. Сообщение двигается по циклу хостов до хоста-получателя с идентификатором *recipient*. Получатель, приняв такое сообщение, обрабатывает его, не посылая дальше.

Что произойдёт, если в сети нет хоста с указанным в сообщении идентификатором получателя? Заметим, что это может произойти не только в результате ошибки в хосте-отправителе, который генерирует сообщение с таким идентификатором получателя, но также из-за изменения сети, например, когда хост удаляется из сети и после соответствующей перенастройки сети удаляется из цикла хостов. В этом случае сообщение будет бесконечно циркулировать по циклу хостов.

Для того чтобы это предотвратить, предлагается такое решение. Когда фиксируется, что сообщение прошло полный цикл хостов, не найдя получателя, оно удаляется, а отправителю посылается сообщение с отрицательным ответом. Фиксация может быть выполнена корнем, контролирующим проход сообщения через него. Однако цикл хостов является простым по дугам путём, который в общем случае не проходит дважды по одной дуге, но может несколько раз проходить через одну и ту же вершину (т.е. не обязательно является вершинно-простым путём), в том числе корень. Поэтому корень сначала фиксирует проход сообщения ***MessageToHost*** по дуге, ведущей от родителя корня в корень, и меняет его тип на ***RootMessageToHost***. Когда последнее сообщение проходит по той же дуге, ведущей от родителя корня в корень, корень удаляет сообщение и посылает отправителю сообщение ***MessageToHost*** с отрицательным ответом. Пометив штрихом параметры этого

отрицательного ответа как функцию параметров исходного сообщения, имеем: *sender`* = *root*, *recipient`* = *sender*, *parameters`* = *parameters*, где *root* идентификатор корня.

Этот отрицательный ответ также может не найти своего получателя *recipient`* = *sender* из-за того, что хост-отправитель исходного сообщения неверно указал свой идентификатор *sender* или этот отправитель удалён из сети и (после перенастройки сети) из цикла хостов. Тогда по общему правилу ответное сообщение (отправитель – корень), дважды пройдя по дуге, ведущей от родителя корня в корень, будет удалено, и корень пошлёт отрицательный ответ на этот отрицательный ответ, получателем которого будет он сам: *recipient``* = *sender`* = *root*. Получив это сообщение, предназначенное корню, корень окончательно удалит его.

Схема передачи по сети сообщения известному получателю наглядно изображена на рис. 2, где двойная стрелка соответствует передаче сообщения по циклу хостов через вершины, не меняющие тип сообщения, цветной кружок означает хост-отправитель или хост-получатель, а белый кружок означает проход по дуге от родителя корня в корень. При передаче сообщений по этой схеме может случиться неправильное поведение: сообщение (ответное сообщение с отрицательным ответом от корня) может быть удалено, хотя в сети есть получатель этого сообщения (получатель ответного сообщения, т.е. отправитель исходного сообщения). Однако это может случиться только тогда, когда меняется корень, т.е. удаляется корень, сеть перенастраивается, и корнем становится другой хост. Удаление получателей и отправителей сообщения, отличных от корня, не приводит к такому неправильному поведению.

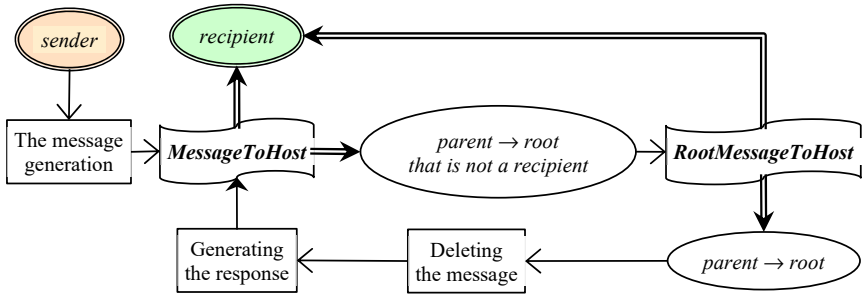

Рис. 2. Схема передачи по сети сообщения известному получателю.
Fig. 2. The network transmission diagram when sending a message to the known recipient.

Для генерации сообщения ***MessageToHost*** конкретному указанному хосту используется процедура ***SendMessageToHost*** с параметрами: *recipient*, *parameters*. Процедура возвращает ***false***, если некому послать сообщение, т.е. в хосте нет правил коммутации (сеть состоит из одной изолированной вершины). В противном случае возвращается ***true*** и посылается по циклу хостов нужное сообщение: с типом ***MessageToHost***, если данный хост не корень, или с типом ***RootMessageToHost*** в противном случае.

5. Услуги

Другой аспект функционирования распределённой сети связан с тем, что сообщение может быть предназначено не для какого-то заранее известного хоста, а «кому-нибудь», кто может это сообщение обработать. Такое сообщение можно понимать как запрос на некие вычислительные действия с параметрами, содержащимися в сообщении. Это аналогично локальному вызову той или иной процедуры внутри одного хоста с той разницей, что вызывающий процесс и вызываемая процедура находятся в разных хостах. При этом, вообще говоря, всё равно в каком хосте эта вызываемая процедура находится, лишь бы она там была. Коммутация сообщений должна позволять «найти» такой хост.

Мы предлагаем моделировать такой запрос понятием *услуги*. Хост, который может оказывать данную услугу (в хосте реализована данная услуга), будем называть *целевым хостом* для этой услуги. В сообщении запроса услуги указывается имя услуги и набор параметров, а в каждом хосте имеется множество имён услуг, которые в этом хосте реализованы, т.е. услуги, которые могут быть оказаны этим хостом. Сообщение передаётся по циклу хостов. Когда сообщение приходит в некоторый хост, то хост проверяет, может ли он оказать запрашиваемую услугу, т.е. является ли целевым хостом для этой услуги. Если не может, хост выполняет функцию коммутации, пересылая сообщение дальше согласно правилам коммутации. Если может, этот хост становится конечным получателем сообщения, сообщение дальше не пересылается, а обрабатывается в этом хосте, т.е. хост выполняет запрос, оказывая запрашиваемую услугу.

Однако такое решение имеет следующий недостаток: сообщения с данным именем услуги, отправляемые из одного и того же хоста-отправителя, будут приниматься одним и тем же хостом-получателем, а именно, ближайшим по циклу целевым хостом. Это может привести к тому, что одни хосты будут перегружены обработкой сообщений, а другие, которые тоже могли бы оказывать какие-то услуги, будут недогружены.

Для решения этой проблемы можно разрешить хосту, который мог бы оказать запрашиваемую услугу, но сейчас слишком «занят» другой работой, не принимать сообщение на обработку, а пересылать его дальше по правилам коммутации. Тем самым, сообщение будет «крутиться» по циклу до тех пор, пока не попадёт в какой-нибудь освободившийся хост, который может оказать запрашиваемую услугу.

Для того чтобы в такой распределённой сети не возникало зацикливание, нужно, чтобы каждая запрашиваемая услуга была реализована в каком-нибудь хосте. Иначе сообщение с именем этой услуги будет бесконечно двигаться по циклу, не находя нужного хоста. Для того чтобы обойти это ограничение предлагается решение аналогичное решению, рассмотренному в предыдущем разделе для сообщений указанному хосту.

Вводятся три типа сообщений: ***Message***, ***RootMessage*** и ***WaitingMessage***. Они имеют параметры: *sender* – идентификатор хоста-отправителя, сгенерировавшего сообщение, *service* – имя запрашиваемой услуги, *parameters* – параметры сообщения, прозрачные для коммутации сообщений.

Сначала сообщение, как правило, посылается с типом ***Message***. Если оно приходит в корень, который не является целевым хостом, по дуге, ведущей от родителя корня в корень, сообщение пересылается дальше по циклу с типом ***RootMessage***. Если сообщение ***Message*** или ***RootMessage*** приходит в целевой хост, но сейчас он «занят», хост пересылает сообщение дальше уже с типом ***WaitingMessage***. Корень, не являющийся целевым хостом, получая сообщение ***WaitingMessage*** по дуге, ведущей от родителя корня в корень, пересылает его дальше опять с типом ***Message***. Последнее сделано для того, что «поймать» удаление из цикла хостов того занятого целевого хоста, который сменил тип сообщения с ***Message*** на ***WaitingMessage***. Если в цикле ещё остаются целевые хосты, сообщение, проходя через целевой хост, либо принимается им, если хост «свободен», либо посылается дальше опять с типом ***WaitingMessage***, если хост сейчас «занят». Если сообщение ***RootMessage*** проходит по дуге, ведущей от родителя корня в корень, оно удаляется, а корень генерирует и посылает сообщение типа ***MessageToHost*** хосту-отправителю с отрицательным ответом.

Тем самым, в цикле может «крутиться» только сообщение с запросом услуги, для которой в цикле хостов есть целевые хосты. Но это будет не бесконечно, а до тех пор, пока какой-нибудь целевой хост не освободится и не примет это сообщение, или пока из цикла хостов не будут удалены все целевые хосты при перестройке сети. В то же время число проходов таким сообщением по циклу хостов не ограничено.

Схема передачи по сети сообщения с запросом услуги наглядно изображена на рис. 3, где двойная стрелка соответствует передаче сообщения по циклу хостов через вершины, не являющиеся целевыми хостами или корнем, цветной кружок означает хост-отправитель или

целевой хост, а белый кружок означает проход по дуге от родителя корня в корень. Многоточие показывает, что передача по сети отрицательного ответа от корня отправителю исходного сообщения выполняется, естественно, по общим правилам для сообщения типа ***MessageToHost*** (как на рис. 2).

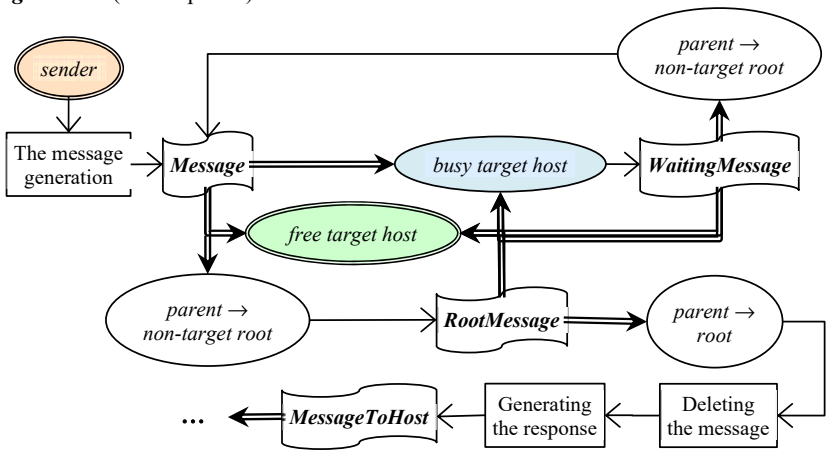

Рис. 3. Схема передачи по сети сообщения запроса услуги.
Fig. 3. The network transmission diagram when sending a service request message.

Предполагается, что когда хост принимает сообщение, оказывая запрашиваемую услугу, то после этого он может, если нужно, сам послать ответ хосту-отправителю, сгенерировавшему это сообщение с результатами оказания услуги. Заметим, что это не всегда нужно. Например, когда требуется некоторая цепочка услуг или более сложная программа, отдельные части которой понимаются как услуги, которые могут быть реализованы в разных хостах, то ответ, если нужно, будет послан хосту-отправителю только после выполнения всей программы. Для посылки ответа хосту также используется сообщение типа ***MessageToHost***, в котором указывается в качестве получателя идентификатор хоста-отправителя исходного сообщения.

Другой способ послать ответное сообщение мог бы быть основан на пути, проходимом сообщением. Если такой путь запоминать в самом сообщении как последовательность проходимых вершин, то ответ мог бы проходить этот путь просто в обратном направлении. Более того, этот путь можно было бы «спрямить», удалив циклы, т.е. в обратном направлении проходить вершинно-простой путь (не проходящий дважды через одну вершину). Однако, во-первых, такой способ требует особой динамической коммутации, происходящей не по статическим правилам коммутации, хранящимся в вершинах, а по информации из сообщения, и, во-вторых, это противоречит нашему предположению о том, что сообщения передаются по сети без изменения (кроме их типа). Поэтому далее способы передачи сообщений такого рода не рассматриваются.

Для генерации в хосте сообщения запроса удалённой услуги используется процедура ***SendMessage*** с параметрами: *service* – имя запрашиваемой услуги, *parameters* – параметры сообщения, прозрачные для коммутации сообщений. Процедура возвращает ***false***, если некому послать сообщение с запросом услуги, т.е. в хосте нет правил коммутации (хост изолированная вершина). В противном случае возвращается ***true*** и посылается нужное сообщение запроса услуги: ***Message*** – если отправитель не целевой хост и не корень, ***RootMessage*** – если отправитель не целевой корень, ***WaitingMessage*** – если отправитель целевой хост. В последнем случае предполагается, что хост делает удалённый вызов потому, что сейчас «занят».

6. Настройка сети

Под настройкой сети понимается, прежде всего, установление правил коммутации. Например, программно-конфигурируемая сеть (SDN) основана на физическом разделении плоскости данных (уровень передачи сообщений, моделируемый графом с вершинами в хостах и коммутаторах) и плоскости управления сетью. На плоскости управления находится контроллер, который и выполняет настройку сети. Он связан с каждым коммутатором, которому «спускает» на уровень плоскости данных правила коммутации. Для этого на уровне контроллера должна быть известна вся топология сети (граф физических связей).

В настоящей статье мы предлагаем алгоритмы самонастройки сети, без использования специального контроллера и с минимально необходимой информацией, заранее заданной в вершинах графа. Эта предварительно заданная информация должна быть необходима и достаточна только для задания топологии сети как упорядоченного графа (графа, в котором рёбра, инцидентные вершине, линейно упорядочены). Достаточность означает, что по такой информации однозначно восстанавливается упорядоченный граф сети, а необходимость означает, что по упорядоченному графу однозначно определяется такая информация. Кроме этого, каждый хост должен знать, какие услуги в нём реализованы.

До настройки сети в каждой вершине графа должна быть заранее задана следующая информация:

- *Self* – идентификатор этой вершины;
- список *Neighbors* идентификаторов соседних вершин;
- *Host* – признак того, является вершина хостом (*Host* = ***true***) или коммутатором (*Host* = ***false***).

Кроме того, в хосте должно быть задано множество *Services* имён реализуемых им услуг.

В предлагаемой модели под настройкой понимается установление:

1) правил коммутации в узлах сети,
2) отметки корня дерева хостов.

Поясним эти установки.

1) Правила коммутации устанавливаются в каждом узле сети: как в коммутаторе, так и в хосте, поскольку каждый узел сети выполняет функцию коммутации сообщений. В вершине инициализируется список *Rules* правил коммутации.
2) В корне дерева хостов инициализируется переменная *Root* := ***true***, в остальных хостах сети *Root* := ***false***. Как описано выше, это нужно для предотвращения бесконечной циркуляции сообщений в сети.

Алгоритмы настройки сети строят дерево хостов и цикл хостов как обход этого дерева, а также лес деревьев коммутаторов. Сами алгоритмы самонастройки, а также алгоритмы генерации и передачи сообщений по настроенной сети формально описаны в приложении.

Все правила коммутации в вершине s имеют вид $(:a, s, b)$, то есть параметры коммутации отсутствуют, а средняя вершина одна и та же – s. Поэтому в приложении для краткости в каждой вершине s правило коммутации записывается не как $(:a, s, b)$, а как (a, b).

В результате настройки в каждой вершине s цикла хостов список правил коммутации будет иметь «циклический» вид $(:a_0, a_1)$, $(:a_1, a_2)$, ..., $(:a_{n-2}, a_{n-1})$, $(:a_{n-1}, a_0)$, где a_0 родитель вершины s. Кроме того, если список правил коммутации в вершине s имеет такой вид, а сообщение в вершину s приходит от соседа b, отличного от вершин $a_0, ..., a_{n-1}$, оно пересылается вершине a_0 – родителю вершины s по подразумеваемому правилу (b, a_0). Иными словами, список (a_0, a_1), (a_1, a_2), ..., (a_{n-1}, a_0) следует понимать как сокращённую запись списка (a_0, a_1), (a_1, a_2), ..., (a_{n-1}, a_0), (b_1, a_0), (b_2, a_0), ..., (b_k, a_0), где $b_1, ..., b_k$ все соседи вершины s, кроме соседей $a_0, ..., a_{n-1}$. Такое *правило умолчания* мы применяем для упрощения

алгоритмов самонастройки сети в Приложении. Это правило умолчания используется при инкрементальной (повторной частичной) перенастройке сети, когда меняется граф сети. Для передачи сообщений по настроенной сети это правило умолчания не требуется, за исключением одного случая: мы допускаем посылку сообщения извне сети в сеть, в вершину b с указанием идентификатора предшественника a, не совпадающего с идентификаторами вершин сети, что можно понимать как посылку сообщения по дуге ab, не входящей в граф сети.

На рис. 4 показано, какие правила создаются для вершины в разных случаях. Чёрные линии изображают рёбра дерева хостов, красные – деревьев коммутаторов. Правило (a, b) в вершине s показано стрелкой, соединяющей две смежные дуги as и sb; синие стрелки соответствуют циклу хостов, а красные стрелки – деревьям коммутаторов. Показаны правила как до (рис. 4 сверху), так и после применения правила умолчания (рис. 4 внизу).

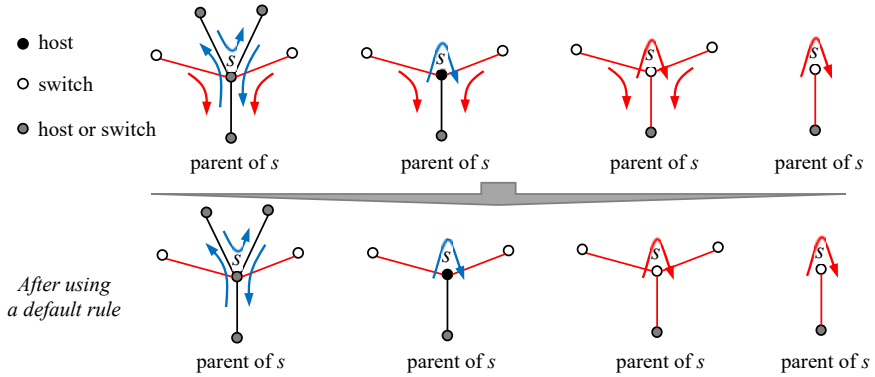

Рис. 4. Правила коммутации и правило умолчания.
Fig. 4. Switching rules and a default rule.

Будем считать, что настройка инициируется сообщением ***Start***, которое поступает в некоторый (произвольный) хост извне графа. Этот хост будем называть *инициатором*. Для удобства будем считать, что сообщение приходит по некоторому дополнительному ребру, соединяющему инициатор с внешним окружением, моделируемым дополнительной вершиной, соседней с инициатором, которую будем называть *внешним соседом* инициатора. Этому же внешнему соседу инициатор посылает ответ о выполнении этапа. В то же время будем предполагать, что внешний сосед не входит в список *Neighbors* соседей инициатора, так как этот список (по определению) содержит только вершины графа.

В предлагаемых алгоритмах во время настройки в сети циркулирует одно сообщение, которое имеет тип и не имеет параметров. При пересылке сообщения узлы сети могут менять только тип сообщения.

Замечание. Если разрешить изменяемые и неограниченные по размеру сообщения, то можно было бы с помощью сообщений сначала собрать всю информацию о сети. Для этого можно использовать сообщения, которые, обходя граф, собирали бы информацию о сети: как о её топологии, так и о распределении реализаций услуг по хостам сети. Получив такую информацию, инициатор (или его внешний сосед) мог бы произвести необходимую оптимизацию и составить описание, содержащее требуемую инициализацию переменных для каждого узла сети. После этого специальным сообщением, обходящим граф, каждому узлу сети доставлялась бы касающаяся его часть этого описания. Стоит отметить, что такой инициатор уже мало отличается от контроллера в SDN: только вместо прямой связи с каждым узлом сети ему приходится посылать сообщение по сети до этого узла. В настоящей статье мы не допускаем таких изменяемых сообщений, тем более, неограниченных по размеру.

Настройка сети может начать выполняться в любой момент времени, когда уже закончена предыдущая (если она была) настройка. Сначала неформально опишем алгоритм обхода остовного дерева по алгоритму Тэрри. Хост, получающий сообщение ***Start***, становится инициатором и корнем дерева. Далее сообщение может иметь один из трёх типов: *прямое* сообщение ***Forward*** и два *ответных сообщения* (*ответа*) ***Cancel*** и ***Back***. Сообщение ***Forward*** посылается по ещё не пройденной дуге графа. Если вершина *b* получает от соседа *a* сообщение ***Forward*** первый раз, то дуга *ab* становится дугой остовного дерева, ориентированного от корня; вершина *a* является родителем вершины *b*. На время настройки родителем корня (= инициатора) будем считать его внешнего соседа. Если вершина *b* получает от соседа *a* повторное сообщение ***Forward***, то оно приходит по хорде *ab* остовного дерева (хорда – это ребро, не лежащее на дереве, оба конца которого лежат на дереве). В этом случае по этой хорде в обратном направлении *ba* посылается ответ ***Cancel***. Иначе вершина *b* посылает сообщение ***Forward*** следующему после *a* соседу *c*, если дуга *bc* ещё не пройдена (по ней не посылалось сообщение ***Forward***) и вершина *c* отлична от родителя. То же самое делается, если вершина *b* получает ответ ***Cancel*** или ***Back*** от соседа *a*. Все выходящие из вершины *b* дуги уже пройдены в том случае, когда следующий после *a* сосед *c* является родителем для вершины *b*. Тогда вершина *b* посылает своему родителю ответ ***Back*** по обратной дуге дерева хостов. По ходу дела происходит генерация правил коммутации сообщений для цикла обхода остовного дерева. В конце корень (= инициатор) замыкает цикл обхода, заменяя два сгенерированных правила (: *внешний сосед*, *инициатор*, *b*) и (: *c*, *инициатор*, *внешний сосед*) одним правилом (: *c*, *инициатор*, *b*). После окончания настройки сосед *c* становится родителем корня.

Для того, чтобы строить не цикл обхода остовного дерева, а цикл обхода дерева хостов, т.е. цикл хостов, применяется процедура «Удаление «лишних» коммутаторов». Для этого перед посылкой вершиной *b* ответа родителю по обратной дуге остовного дерева сначала проверяется, является ли вершина *b* хостом и, если нет, то имеются ли хосты в поддереве остовного дерева с корнем в в вершине *b*. Такие хосты имеются, если либо (при использовании правила умолчания, рис. 4 внизу) в вершине *b* определено больше одного правила коммутации, либо (без использования правила умолчания, рис. 4 вверху) в этих правилах есть хотя бы два разных преемника. Если в поддереве остовного дерева с корнем в вершине *b* имеются хосты, вершина *b* посылает ответ ***Back***, иначе посылается ответ ***Cancel***.

Настройка заканчивается, когда корень (= инициатор) посылает своему внешнему соседу ответное сообщение ***Back*** (корень = инициатор является хостом, поэтому внешнему соседу корня = инициатора не может быть послан ответ ***Cancel***).

7. Инкрементальная настройка сети

Под инкрементальной настройкой сети будем понимать дополнительную, чаще всего частичную перенастройку сети, необходимую при изменении сети. Такое изменение касается либо

1) распределения реализаций услуг по хостам сети, либо
2) топологии сети (изменение графа сети).

Каждое такое изменение можно представить как последовательность элементарных изменений. Поэтому мы будем рассматривать только элементарные изменения и соответствующие им инкрементальные настройки сети.

1) При изменении распределения реализаций услуг по хостам сети элементарными изменениями являются:
   1.1 добавление реализации одной услуги в один хост,
   1.2 удаление реализации одной услуги из одного хоста.

2) При изменении топологии сети элементарными изменениями являются:
   2.1 изменение упорядочивания графа,
   2.2 добавление одного ребра,
   2.3 удаление одного ребра, не нарушающее связность графа,
   2.4 добавление одного коммутатора и одного ребра, соединяющего его со «старой» вершиной графа,
   2.5 добавление одного хоста и одного ребра, соединяющего его со «старой» вершиной графа,
   2.6 удаление одного терминального коммутатора и инцидентного ему ребра,
   2.7 удаление одного терминального хоста и инцидентного ему ребра.

Примером более сложного изменения топологии сети является удаление хоста и всех инцидентных ему рёбер. Ему соответствует последовательность удалений одного ребра (2.3) для всех инцидентных хосту рёбер, кроме одного, а затем удаление одного терминального хоста и инцидентного ему ребра (2.7).

В отличие от предыдущих разделов здесь мы предлагаем только идеи алгоритмов настройки, по которым легко можно разработать сами алгоритмы инкрементальной настройки, мы не приводим эти алгоритмы в Приложении. В некоторых случаях перенастройка сети из-за какого-то изменения может быть выполнена эффективнее, если её делать «сразу», а не как последовательность перенастроек сети по элементарным изменениям. Но это уже вопрос оптимизации (и усложнения) алгоритмов инкрементальной настройки. Кроме того, мы рассматриваем только такую инкрементальную настройку, которая необходима для поддержания функциональности сети, то есть возможности передавать сообщения от любого хоста-отправителя любому известному хосту или запрашивать любую услугу. Если перенастройка полезна для оптимизации сети (например, уменьшения длины цикла хостов), то мы только отмечаем этот факт, но не предлагаем алгоритмы такой перенастройки.

В ряде случаев во время инкрементальной настройки нужно знать, лежит ли вершина на цикле хостов или нет. У нас нет соответствующей отметки (булевской переменной) в вершинах графа, но она и не нужна. Вершина лежит на цикле хостов тогда и только тогда, когда она является хостом или это коммутатор, и в нём либо определено больше одного правила коммутации (при использовании правила умолчания, рис. 4 внизу), либо в этих правилах есть хотя бы два разных преемника (без использования правила умолчания, рис. 4 вверху).

Инкрементальная настройка сети, как правило, не нарушает функциональности сети. Если во время настройки имеется сообщение указанному хосту (***MessageToHost*** или ***RootMessageToHost***), оно будет доставлено получателю, если в сообщении верно указан его идентификатор, и получатель не удалён из сети. Иначе сообщение с отрицательным ответом (***MessageToHost*** или ***RootMessageToHost***), будет доставлено отправителю исходного сообщения, если в сообщении верно указан идентификатор отправителя, и отправитель не удалён из сети. Если во время настройки имеется сообщение с запросом услуги (***Message***, ***RootMessage*** или ***WaitingMessage***), оно будет доставлено целевому хосту, если в сообщении верно указано имя услуги и в сети есть целевые хосты для этой услуги. Иначе сообщение с отрицательным ответом (***MessageToHost*** или ***RootMessageToHost***), будет доставлено отправителю исходного сообщения, если в сообщении верно указан идентификатор отправителя, и отправитель не удалён из сети. Исключением является случай, когда меняется корень дерева хостов: «старый» корень удаляется, и корнем становится другой хост. В этом случае сообщение может не дойти до адресата, даже если такой адресат есть в сети.

7.1 Изменение распределения реализаций услуг по хостам

7.1.1 Добавление реализации одной услуги в один хост

При добавлении реализации услуги с именем *service* в некоторый хост нужно выполнить в этом хосте *Services* := *Services* ∪ {*service*}. Больше никаких действий не требуется.

7.1.2 Удаление реализации одной услуги из одного хоста

При удалении реализации услуги с именем *service* из некоторого хоста нужно выполнить в этом хосте *Services* := *Services* \ {*service*}. Больше никаких действий не требуется. Заметим, что хост, в котором не остаётся реализованных услуг, отличается от коммутатора, поскольку, хотя он не может оказывать услуг, но может генерировать сообщения с запросом услуг, и, кроме того, может быть получателем сообщений указанному хосту (***MessageToHost*** или ***RootMessageToHost***).

7.2 Изменение топологии сети

7.2.1 Изменение упорядочивания графа

Другое упорядочивание графа, т.е. определение других линейных порядков соседей каждой вершины, приводит к построению другого остовного дерева при настройке сети и, как следствие, других дерева и цикла хостов и леса деревьев коммутаторов. Для того чтобы задать новое упорядочивание графа, нужно в каждой вершине, в которой изменился линейный порядок соседей, установить новый список *Neighbors* идентификаторов соседних вершин. Однако для сохранения функциональности сети перенастройка не требуется. В то же время полная перенастройка сети могла бы быть полезна в целях оптимизации: уменьшения длины цикла хостов.

7.2.2 Добавление одного ребра

Для сохранения функциональности сети перенастройка не требуется. В то же время перенастройка сети могла бы быть полезна в целях оптимизации: уменьшения длины цикла хостов. Для подготовки к следующей настройке при добавлении ребра $\{a, b\}$ в список *Neighbors* в вершине *a* добавляется идентификатор соседа *b*, а в список *Neighbors* в вершине *b* добавляется идентификатор соседа *a*.

7.2.3 Удаление одного ребра, не нарушающее связность графа

Для подготовки к следующей настройке при удалении ребра $\{a, b\}$ из списка *Neighbors* в вершине *a* удаляется идентификатор соседа *b*, а из списка *Neighbors* в вершине *b* удаляется идентификатор соседа *a*.

Если удаляемое ребро является хордой остовного дерева, перенастройка не требуется (даже для оптимизации).

Если удаляемое ребро лежит на дереве коммутаторов, его удаление нарушает связность этого дерева (рис. 5). Если это ребро $\{a, b\}$, где вершина *a* является родителем для вершины *b*, то одна из этих компонент связности является поддеревом остовного дерева с корнем в вершине *b*, а другая компонента содержит цикл хостов. Поскольку граф остаётся связным, должно быть ребро $\{c, d\}$, соединяющее вершину *c* из первой компоненты с вершиной *d* из второй компоненты. Требуется изменить правила коммутации в вершинах пути по (неориентированному) дереву коммутаторов из вершины *b* в вершину *c*, а также в вершинах *a* и *d*. Однако с учётом правила умолчания правила в вершинах *a* и *d* можно не менять, так как в них не изменились родители по соответствующим деревьям (дерева хостов для вершины *a* и дерева коммутаторов для вершины *d*).

Если удаляемое ребро лежит на дереве хостов, его удаление нарушает связность этого дерева (рис. 6). Если это ребро $\{a, b\}$, где вершина *a* является родителем для вершины *b*, то одна из этих компонент связности является поддеревом остовного дерева с корнем в вершине *b*, а другая компонента содержит корень дерева хостов. Поскольку граф остаётся связным, должно быть ребро $\{c, d\}$, соединяющее вершину *c* из первой компоненты с вершиной *d* из второй компоненты. Требуется изменить правила коммутации в вершинах пути по (неориентированному) остовному дереву из вершины *b* в вершину *c* и в вершинах пути по (неориентированному) остовному дереву из вершины *d* до цикла хостов, а также в вершине *a*.

7.2.4 Добавление одного коммутатора и одного ребра, соединяющего его со «старой» вершиной графа

Для подготовки к следующей настройке при добавлении коммутатора *a* и ребра $\{a, b\}$ создаётся список *Neighbors* = $\{b\}$ в коммутаторе *a* и в список *Neighbors* в вершине *b* добавляется идентификатор соседа *a*. Для сохранения функциональности сети перенастройка не требуется. Однако при дальнейших изменениях топологии сети во время соответствующей перенастройки сети нужно учитывать добавляемые коммутатор и ребро. Поскольку при добавлении коммутатора *a* только с одним инцидентным ему ребром $\{a, b\}$, коммутатор *a* будет терминальной вершиной, он может входить только в дерево коммутаторов. Правила коммутации устанавливаются в добавляемом коммутаторе *a* и меняются в другом конце *b* добавляемого ребра (рис. 7). Однако с учётом правила умолчания для коммутации сообщений достаточно только в вершине *a* установить правило коммутации $\{: b, a, b\}$.

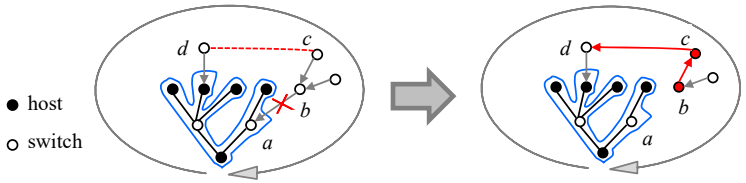

Рис. 5. Удаление ребра на дереве коммутаторов.
Fig. 5. Deleting an edge of a switch tree.

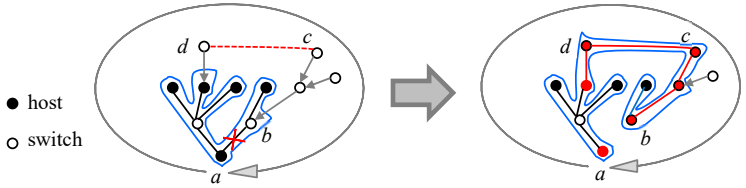

Рис. 6. Удаление ребра на дереве хостов.
Fig. 6. Deleting an edge of a host tree.

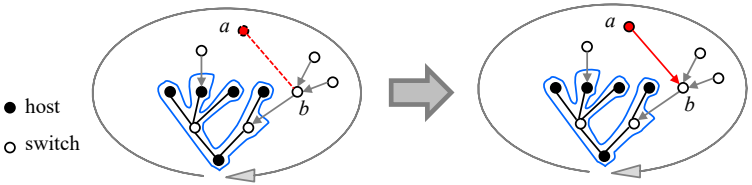

Рис. 7. Добавление одного коммутатора и одного ребра, соединяющего его со «старой» вершиной графа.
Fig. 7. Adding one host and one edge connecting that connects this host with the "old" graph node.

7.2.5 Добавление одного хоста и одного ребра, соединяющего его со «старой» вершиной графа

Когда добавляется хост a и ребро $\{a, b\}$, соединяющее его со «старой» вершиной b, нужно создать список $Neighbors = \{b\}$ в хосте a, в список $Neighbors$ в вершине b добавить идентификатор соседа a, и добавить хост a в цикл хостов. Правила коммутации устанавливаются в добавляемом хосте a и меняются в вершине b; кроме того, если вершина b не лежит на цикле хостов, то она является некорневой вершиной некоторого дерева коммутаторов, и нужно поменять правила коммутации на всём пути от вершины b до корня этого дерева (рис. 8).

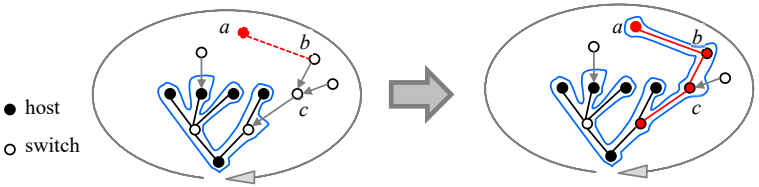

Рис. 8. Добавление одного хоста и одного ребра, соединяющего его со «старой» вершиной графа.
Fig. 8. Adding one switch and one edge connecting that connects this switch with the "old" graph node.

7.2.6 Удаление одного терминального коммутатора и инцидентного ему ребра

Связность графа не нарушается, перенастройка не требуется при использовании правила умолчания. Если правило умолчания не используется, при удалении терминального коммутатора a и ребра $\{a, b\}$, нужно в вершине b удалить правило (: a, b, c). Для подготовки к следующей настройке из списка $Neighbors$ в вершине b удаляется идентификатор соседа a.

7.2.7 Удаление одного терминального хоста и инцидентного ему ребра

При удалении терминального хоста a и (единственного) инцидентного ему ребра $\{a, b\}$ связность графа не нарушается. Для подготовки к следующей настройке из списка $Neighbors$ в вершине b удаляется идентификатор соседа a. Удаляемый хост нужно удалить из цикла хостов. Поскольку хост терминальный, он является корнем или листовой вершиной дерева хостов. С учётом правила умолчания достаточно изменить правила коммутации в вершине b (рис. 9). Кроме того, если удаляется корень дерева хостов, нужно отметить в качестве корня другой (всё равно какой) хост.

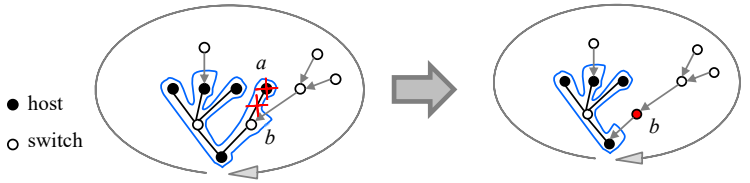

Рис. 9. Удаление одного терминального хоста и инцидентного ему ребра.
Fig. 9. Deleting a terminal host with an incident edge.

8. Заключение

У решения, предложенного в данной статье, есть один существенный недостаток: путь, который должно пройти сообщение, прежде чем «найдёт» нужный хост, может оказаться слишком длинным. Например, если в сети для некоторой услуги есть только один целевой хост a (в котором реализована эта услуга), то сообщение с запросом этой услуги, сгенерированное хостом b, следующим после хоста a в цикле хостов, пройдёт весь цикл хостов, кроме коммутаторов между a и b.

Для того чтобы устранить этот недостаток, можно было бы строить не общий цикл всех хостов, а циклы целевых хостов – по одному для каждой услуги в предположении, что таких циклов существенно меньше. Соответственно, имя услуги было бы параметром коммутации. Но такое решение плохо справляется с проблемой масштабируемости, если число услуг можно считать достаточно большим и, более того, если это число имеет тенденцию к росту. Для масштабируемого решения этой проблемы можно воспользоваться подходящей кластеризацией услуг. Предполагается, что этому будет посвящена наша следующая статья на эту тему.

Список литературы / References

[1]. М.С.Э. Танненбаум. Распределенные системы, Москва, ДМК Пресс, 2021.
[2]. Г.И. Радченко. Распределенные вычислительные системы. Челябинск, 2012.
[3]. Sezer. S, Scott-Hayward. S, Chouhan P.K., Fraser B., Lake D., Finnegan J., Viljoen N., Miller M. and Rao N. Are we ready for sdn? Implementation challenges for software-defined networks IEEE Communications Magazine, 2013, 51 (7), pp. 36-43.
[4]. Mohammed, A. H., Khaleefah, R. M., k. Hussein, M., and Amjad Abdulateef, I. A review software defined networking for internet of things. In 2020 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA), 2020, pp. 1–8.
[5]. OpenNetworkingFoundation (2012). Software-defined networking: The new norm for networks. ONF White Paper. 2012.
[6]. Burdonov, I.; Kossachev, A.; Yevtushenko, N.; López, J.; Kushik, N. and Zeghlache, D. (2021). Preventive Model-based Verification and Repairing for SDN Requests. In Proceedings of the 16th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE, ISBN 978-989-758-508-1 ISSN 2184-4895, pp. 421-428. DOI: 10.5220/0010494504210428.
[7]. Igor Burdonov, Nina Yevtushenko and Alexander Kossatchev. Implementing a virtual network on the SDN data plane. Proceedings 2020 IEEE East-West Design & Test Symposium (EWDTS). 2020, pp. 279- 283.
[8]. Бурдонов И.Б., Евтушенко Н.В., Косачев А.С. Реализация распределенных и параллельных вычислений в сети SDN. Труды института системного программирования. 2022, т. 34, № 3, с. 159- 172.
[9]. Оре О. Теория графов. М.: Наука, 1968, с. 59, 68.

ПРИЛОЖЕНИЕ

Типы глобальных переменных и параметров процедур записаны ***строчными буквами полужирным курсивом***.

Имена процедур ***Начинаются с прописной буквы и записаны полужирным курсивом***.

Имена глобальных переменных *Начинаются с прописной буквы и записаны не полужирным курсивом*.

Имена параметров процедур и локальных переменных *начинаются со строчной буквы и записаны не полужирным курсивом*.

П1. Типы глобальных переменных и параметров процедур в хостах и коммутаторах

bool – булевский тип,

vertex – идентификатор вершины графа,

service – имя услуги,

set(*type*) – множество элементов типа *type*,

list(*type*) – список элементов типа *type*,

(*type*$_1$, *type*$_2$) – пара (элемент типа *type*$_1$, элемент типа *type*$_2$).

П2. Глобальные переменные в вершине – хосте или коммутаторе

vertex *Self* = ...; /* инициализировано, не меняется: собственный идентификатор вершины */
list(***vertex***) *Neighbors* = ...; /* инициализировано, не меняется: список соседних вершин */
bool *Repeat* = ***false***; /* инициализировано первоначально как ***false***; */
/* во время настройки получает значение ***true***, когда вершина получает первое сообщение */
list(***vertex***, ***vertex***) *Rules*; /* список правил коммутации как список пар (предшественник, преемник) */
bool *Host* = ...; /* инициализировано, не меняется: признак того, что вершина является хостом */

П3. Глобальные переменные в хосте

set(***service***) *Services* = ...; /* инициализировано: множество имён услуг, реализуемых хостом */
Bool *Root*; /* признак того, что хост является корнем дерева хостов */

П4. Использование глобальных переменных

Глобальные переменные, используемые на этапе самонастройки сети: *Neighbors*, *Repeat*, *Rules*, *Host*; *Root* (все, кроме *Self*, *Services*).

Глобальные переменные, используемые при передаче сообщений по настроенной сети: *Self*, *Rules*, *Host*, *Services*, *Root* (все, кроме *Neighbors*, *Repeat*); при генерации сообщений: *Self*, *Rules*, *Root*.

П5. Правила коммутации

Правила коммутации *Rules* представлены как список длиной n пар (идентификатор соседа-предшественника, идентификатор соседа-преемника): (a_0, a_1), (a_1, a_2), ..., (a_{n-1}, a_n). Мы будем использовать следующую обычную нотацию:

Rules[$i..j$] = (a_{i-1}, a_i), ..., (a_{j-1}, a_j), если $i = 1..n$, $j = 1..n$ и $i \le j$; иначе = ();
Rules[i][1] = a_{i-1} для $i = 1..n$;
Rules[i][2] = a_i для $i = 1..n$.

Правило умолчания: при получении вершиной сообщения от вершины b, отличной от вершин a_0, ..., a_{n-1}, оно пересылается родителю a_0 (предшественнику из первого правила) по подразумеваемому правилу (b, a_0).

П6. Вспомогательные процедуры

```
vertex NextNeighbor(vertex x) { /* следующий после x сосед в циклическом списке b1, ..., bk */
    n = |Neighbors|; i := 1;
    while i ≤ n & Neighbors[i] ≠ x do { i := i + 1; }
    if i = n { i := 0; } /* если x = bi и i < n, то bi+1, если x = bk, то b1 */
    return Neighbors[i + 1]; }

vertex Successor(list(vertex, vertex) rules, vertex x) {
/* по предшественнику x вычисление преемника b в списке правил rules = (a0, a1), ..., (an-1, an) */
    n = |rules|; i := 1;
    while i ≤ n & rules[i][1] ≠ x do { i := i + 1; }
    if i ≤ n { return rules[i][2]; }
    else return rules[1][1]; } /* если для каждого b в rules нет правила (x, b), возвращаем a0 */
```

П7. Процедуры обработки сообщений

Сигнатура процедуры обработки сообщения имеет вид ***Type***(***vertex*** *x*, *параметры сообщения*), а оператор посылки сообщения имеет вид ***SEND***(***Type***(*параметры сообщения*), *y*), где ***Type*** тип сообщения, *x* сосед, от которого принято сообщение, *y* сосед, которому посылается сообщение.

П7.1 Самонастройка сети

Построение цикла хостов (цикла обхода дерева хостов с корнем в инициаторе) по алгоритму Тэрри, а также леса деревьев коммутаторов.

vertex *x* – идентификатор соседа, от которого получено сообщение,

```
Start(vertex x) { /* старт настройки, сообщение пришло в хост-инициатор от его внешнего соседа x */
    Rules := (); /* удаляем все правила коммутации в вершине */
    Root := true; /* инициатор становится корнем дерева хостов */
    if Neighbors = () { /* если корень изолированная вершина (нет соседей), то */
        SEND(Back(), x); } /* ответ внешнему соседу, конец настройки */
    else { /* у корня есть соседи */
        Repeat := true; /* отмечаем, что корень принял первое сообщение настройки */
        Rules := ((x, Neighbors[1])); /* создаём правило (внешний сосед, a1), a1 = первый сосед */
        SEND(Forward(), Neighbors[1])); } } /* прямое сообщение соседу a1 */

Forward(vertex x) { /* прямое сообщение послано по ещё не пройденной дуге */
    if Repeat = false { /* если это первое сообщение настройки, принятое вершиной, то */
        Repeat := true; /* отмечаем, что вершина приняла первое сообщение настройки */
        Rules:= ((x, NextNeighbor(x))); /* создаём правило (x, a1) от родителя a0 = x к соседу a1 */
        if Host = true { Root := false; } /* /* если это хост, то отмечаем, что хост не корень */
        if NextNeighbor(x) ≠ x { /* если вершина не листовая в остовном дереве, a1 ≠ a0 = x, то */
            SEND(Forward(), NextNeighbor(x)); } /* прямое сообщение следующему соседу a1 */
        else { /* если вершина листовая в остовном дереве, a1 = a0 = x, то */
            Repeat := false; /* очистка для следующей настройки */
            if Host = true { /* если хост, то */
                SEND(Back(), x); } /* положительный ответ родителю a0 = x */
            else { /* если не хост, то «Удаление «лишних» коммутаторов» */
                SEND(Cancel(), x); } } } /* отрицательный ответ родителю a0 = x */
    else { /* повторное прямое сообщение, пришло по хорде остовного дерева */
        SEND(Cancel(), x); } } /* отрицательный ответ соседу x */
```

```
Cancel(vertex x) { /* отрицательный ответ от вершины к её родителю или по хорде остовного дерева */
    /* в этот момент правила коммутации Rules = (a0, a1), (a1, a2), ..., (an - 2, an - 1), (an - 1, an = x) */
    n : = |Rules|; /* число правил коммутации */
    y: = NextNeighbor(x) /* сосед y следующий после соседа x в циклическом списке соседей */
    Rules[n][2] := y; /* меняем последнее правило (an - 1, an = x) → (an - 1, y) */
    if y ≠ Rules[1][1] /* если y ≠ a0 (не родитель), то не все соседи, отличные от родителя, пройдены*/
        SEND(Forward(), y); } /* прямое сообщение не пройденному соседу y */
    else { /* если все соседи, отличные от родителя, пройдены, то */
        Repeat := false; /* очистка для следующей настройки */
        if Host = true or n > 1 { /* если вершина хост или не листовая в дереве хостов, то */
            SEND(Back(), y); } /* положительный ответ родителю y = a0 */
        else {   /* если вершина листовой коммутатор, то «Удаление «лишних» коммутаторов» */
            SEND(Cancel(), y); } /* отрицательный ответ родителю y = a0 */
        if Host = true & Root = true { /* если обход завершён, то конец настройки */
            /* замыкание цикла хостов: (внешний, a1), ..., (an - 1, внешний) → (an - 1, a1), ..., (an- 2, an - 1) */
            Rules[1][1] := Rules[n][1]; /* первое правило: (внешний, a1) → (an - 1, a1) */
            Rules := Rules[1..n - 1]; } } } /* удаляем последнее правило (an - 1, внешний) */

Back(vertex x) { /* положительный ответ от вершины к её родителю в дереве хостов */
    /* в этот момент правила коммутации Rules = (a0, a1), (a1, a2), ..., (an - 2, an - 1), (an - 1, an = x) */
    y: = NextNeighbor(x) /* сосед y следующий после соседа x в циклическом списке соседей */
    Rules := Rules^(x, y); /* добавляем правило (an = x, an + 1 = y) */
    if y ≠ Rules[1][1]) /* если y ≠ a0 (не родитель), то не все соседи, отличные от родителя, пройдены*/
        SEND(Forward(), y); } /* прямое сообщение не пройденному соседу an + 1 = y */
    else { /* если все соседи, отличные от родителя, пройдены, то */
        Repeat := false; /* очистка для следующей настройки */
        SEND(Back(), y); /* положительный ответ родителю y = a0 */
        if Host = true & Root = true { /* если обход завершён, то конец настройки */
            n : = |Rules)|;    /* число правил коммутации */
            /* замыкание цикла хостов: (внешний, a1), ..., (an - 1, внешний) → (an - 1, a1), ..., (an- 2, an - 1) */
            Rules[1][1] := Rules[n][1]; /* первое правило: (внешний, a1) → (an - 1, a1) */
            Rules := Rules[1..n - 1]; } } } /* удаляем последнее правило (an - 1, внешний) */
```

П7.2 Передача сообщений по настроенной сети

vertex *x* – идентификатор соседа, от которого получено сообщение,
vertex *sender* – идентификатор хоста-отправителя, применяется для посылки ответа отправителю,
service *service* – имя запрашиваемой услуги,
parameters – параметры сообщения, прозрачные для коммутации сообщений.

```
MessageToHost(vertex x, vertex sender, vertex recipient, parameters) {
    /* сообщение хосту-получателю, не прошедшее дугу от родителя корня в корень */
    y := Successor(x); /* сосед-преемник y соседа-предшественника x */
    if Self ≠ recipient { /* если вершина не получатель, то*/
        if x ≠ Rules[1][1] ∨ Host ≠ true ∨ Root = false { /* если пришли не в корень от его родителя, то */
            SEND(MessageToHost(sender, recipient, parameters), y); } /* дальше по циклу хостов */
        else { /* если пришли в корень от его родителя, то */
            SEND(RootMessageToHost(sender, recipient, parameters), y);}} /* дальше по циклу хостов */
    else { /* если вершина получатель, то */
        MessageToHostProcessing(parameters); } } /* локальная обработка принятого сообщения */
```



```
RootMessageToHost(vertex x, vertex sender, vertex recipient, parameters) {
    /* сообщение хосту-получателю, прошедшее дугу от родителя корня в корень */
    y := Successor(Rules, x); /* сосед-преемник y соседа-предшественника x */
    if Self ≠ recipient { /* если вершина не получатель, то*/
        if x ≠ Rules[1][1] ∨ Host ≠ true ∨ Root = false {
        /* если пришли не в корень или в корень, но не от его родителя, то */
            SEND(RootMessageToHost(sender, recipient, parameters), y); } /* дальше по циклу хостов */
        else { /* если пришли по дуге от родителя корня в корень, то */
            /* отрицательный ответ отправителю сообщения, не нашедшего получателя */
            SEND(MessageToHost(Self, sender, parameters), y); } }
    else { /* если вершина хост recipient, то */
        MessageToHostProcessing(parameters); } } /* локальная обработка принятого сообщения */

Message(vertex x, vertex sender, service service, parameters) {
    /* сообщение запроса услуги, не прошедшее дугу от родителя корня в корень *
    y := Successor(Rules, x); /* сосед-преемник y соседа-предшественника x */
    if Host ≠ true ∨ service ∉ Services { /* если в вершине не реализована запрашиваемая услуга, то */
        if x ≠ Rules[1][1] ∨ Host ≠ true ∨ Root = false {
            /* если пришли не в корень или в корень, но не от его родителя, то */
            SEND(Message(sender, service, parameters), y); } /* дальше по циклу хостов */
        else { /* если пришли по дуге от родителя корня в корень, то */
            SEND(RootMessage(sender, service, parameters), y); } } /* дальше по циклу хостов */
    else { /* если вершина хост, реализующий запрашиваемую услугу, то */
        if HostReadyToExecuteService(service, parameters) = false { /* если хост «занят», то */
            SEND(WaitingMessage(sender, service, parameters), y); } /* дальше по циклу хостов */
        else { /* если хост «свободен», то */
            service(sender, parameters); } } } /* локальный вызов запрашиваемой услуги */

RootMessage(vertex x, vertex sender, service service, parameters) {
    /* сообщение запроса услуги, прошедшее дугу от родителя корня в корень*/
    y := Successor(Rules, x); /* сосед-преемник y соседа-предшественника x */
    if Host ≠ true ∨ service ∉ Services { /* если в вершине не реализована запрашиваемая услуга, то */
        if x ≠ Rules[1][1] ∨ Host ≠ true ∨ Root = false {
            /* если пришли не в корень или в корень, но не от его родителя, то */
            SEND(RootMessage(sender, service, parameters), y); } /* дальше по циклу хостов */
        else { /* если пришли по дуге от родителя корня в корень, то */
            /* отрицательный ответ отправителю сообщения, не нашедшего получателя */
            SEND(MessageToHost(Self, sender, parameters), Rules[1][2]); } }
    else { /* если вершина хост, реализующий запрашиваемую услугу, то */
        if HostReadyToExecuteService(service, parameters) = false { /* если хост «занят», то */
            SEND(WaitingMessage(sender, service, parameters), y); } /* дальше по циклу хостов */
        else { /* если хост «свободен», то */
            service(sender, parameters); } } } /* локальный вызов запрашиваемой услуги */
```

```
WaitingMessage(vertex x, vertex sender, service service, parameters) {
  /* сообщение запроса услуги, ожидающее освобождения хоста, который может оказать услугу */
  y := Successor(Rules, x); /* сосед-преемник y соседа-предшественника x */
  if Host ≠ true ∨ service ∉ Services { /* если в вершине не реализована запрашиваемая услуга, то */
    if x ≠ Rules[1][1] ∨ Host ≠ true ∨ Root = false {
      /* если пришли не в корень или в корень, но не от его родителя, то */
      SEND(WaitingMessage(sender, service, parameters), y); } /* дальше по циклу хостов */
    else { /* если пришли по дуге от родителя корня в корень, то */
      SEND(Message(sender, service, parameters), y); } } /* дальше по циклу хостов */
  else { /* если вершина хост, реализующий запрашиваемую услугу, то */
    if HostReadyToExecuteService(service, parameters) = false { /* если хост «занят», то */
      SEND(WaitingMessage(sender, service, parameters), y); } /* дальше по циклу хостов */
    else { /* если хост «свободен», то */
      service(sender, parameters); } } } /* локальный вызов запрашиваемой услуги */
```

П8. Генерация сообщения хосту и вызов удалённой услуги

vertex *recipient* – идентификатор хоста, которому предназначено сообщение.
service *service* – имя запрашиваемой услуги,
parameters – параметры сообщения, прозрачные для коммутации сообщений,

```
bool SendMessageToHost(vertex recipient, parameters); {
  /* генерация сообщения известному хосту-получателю */
  if Rules = () { /* если нет правил, то */
    return false; }
  if Root = false { /* если вершина не корень, то */
    /* посылаем сообщение по циклу хостов преемнику a1 первого правила (a0, a1) */
    SEND(MessageToHost(Self, recipient, parameters), Rules[1][2]); }
  else { /* если вершина корень, то */
    /* посылаем сообщение по циклу хостов преемнику a1 первого правила (a0, a1) */
    SEND(RootMessageToHost(Self, recipient, parameters), Rules[1][2]); }
  return true; }

bool SendMessage(service service, parameters); { /* вызов из хоста удалённой услуги */
  if Rules = () { /* если нет правил, то */
    return false; }
  y := Rules[1][2]; /* посылаем следующему соседу a1 */
  if service ∉ Services { /* если в хосте не реализована запрашиваемая услуга, то */
    if Root = true { /* если хост корень, то */
      SEND(RootMessage(Self, service, parameters), y); } /* по циклу хостов */
    else { /* если хост не корень, то */
      SEND(Message(Self, service, parameters), y); } } /* по циклу хостов */
  else { /* если в хосте реализована запрашиваемая услуга (но хост сейчас занят), то */
    SEND(WaitingMessage(Self, service, parameters), y); } /* по циклу хостов */
  return true; }
```


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.