DOI: 10.15514/ISPRAS-2025-37(5)-1

Кластеризация услуг 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. Статья является продолжением предыдущей статьи авторов, в которой строится абстрактная модель распределенной сети, содержащей только хосты и коммутаторы. Хосты предлагают пользователям пакеты определенных услуг (сервисов), сообщения (запросы) между хостами пересылаются через промежуточные узлы по правилам коммутации, и настройка узлов 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, причем каждый хост (как и коммутатор) содержит систему правил коммутации. Обсуждается возможность повышения эффективности работы сети на основе использования информации о классах услуг, на которые разбивается множество всех услуг, предоставляемых хостами сети. На основе информации о классах услуг рассматриваются задачи передачи сообщений, настройка, в том числе инкрементальная, распределенной сети при различных изменениях параметров сети.

Ключевые слова: распределенная сеть; хосты; коммутаторы; классы услуг; (инкрементальная) настройка сети.

Для цитирования: 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.

Clustering services of distributed networks 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. The paper continues the previous work of the authors where an abstract model of a distributed network containing only hosts and switches has been developed. Hosts offer certain services to users; messages (requests) between hosts are forwarded through intermediate nodes according to switching rules, and the node configuration determines a set of paths from host to host along which packets are forwarded. The situation is modeled using a graph of physical connections where the graph nodes are hosts and switches, and each host (like a switch) contains a system of switching rules. The possibility of increasing the efficiency of the network is based on using of information about service classes, into which the set of all services provided by network hosts is divided. Based on the information about service classes, the tasks of message transmission, (incremental) node configuration are considered depending on various changes of network parameters.

Keywords: distributed network; hosts, switches; service classes; (incremental) network configuration.

For citation: Burdonov I.B., Yevtushenko N.V., Kossatchev A.S., Ponomarenko V.N. Clustering services of distributed networks in which hosts can perform message switching functions. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 5, 2025, pp. 7-32 (in Russian). DOI: 10.15514/ISPRAS-2025-37(5)-1.

1. Введение

Данная статья является продолжением статьи [1], в которой авторы строят абстрактную модель распределенной сети. Сеть содержит только хосты, которые предлагают пакеты определенных услуг (сервисов), и сообщения (запросы) между которыми пересылаются через промежуточные узлы (коммутаторы). Подобно тому, как это происходит в большинстве программно-конфигурируемых сетей [2-7] коммутатор работает по правилам коммутации, которые определяют, каким соседним узлам пересылается принятое коммутатором сообщение в зависимости от того, откуда оно пришло, и от вектора параметров в его заголовке. Как следствие, множество путей от хоста к хосту, по которым будут пересылаться пакеты, определяется настройкой коммутаторов. В качестве модели такой сети используется граф физических связей, вершинами которого являются хосты и коммутаторы, а ребра соответствуют физическим связям между ними. В общем случае хосты в такой сети принимают, обрабатывают и посылают информацию другим хостам, но в [1] мы предполагаем, что хост может выполнять также функции коммутации сообщений, т.е. содержит систему правил коммутации для пересылки полученного запроса/сообщения, если данный хост не может его обработать по каким-либо причинам. В программно-конфигурируемых сетях настройка правил коммутации обычно осуществляется специальными компонентами сети, например, SDN-контроллерами, однако в наших работах мы рассматриваем возможность самонастройки коммутаторов и хостов, в зависимости от передаваемых запросов, а также обсуждаем инкрементальную настройку при изменении параметров сети, в частности, её топологии.

В работе [1] обсуждаются проблемы, связанные с нефункциональными параметрами такой распределенной сети, а именно, *достижимость*/*недостижимость* хостов, *зацикливание* сообщений, *перегрузка* сети сообщениями, *немасштабируемость*, и возможности

оптимизации рассматриваемых параметров сети на основе использования информации об услугах/сервисах, ~~представляемых~~ каждым из хостов.

Данная статья является развитием идей и алгоритмов, предложенных в [1]. В частности, обсуждается возможность повышения эффективности работы сети на основе использования информации о классах услуг, на которые разбивается множество всех услуг, предоставляемых хостами сети. На основе информации о классах услуг рассматриваются задачи передачи сообщений, настройка, в том числе инкрементальная (повторная и частичная) настройка распределенной сети при различных изменениях параметров сети.

Структура работы следующая. В разделах 2 и 3 приводятся необходимые сведения из [1], касающиеся предлагаемой модели распределённой сети (раздел 2) и передачи по сети сообщений двух видов: сообщений указанному хосту-получателю и сообщений запроса услуг/сервисов, предоставляемых хостами с автоматическим выбором хоста, способного оказать запрашиваемую услугу (раздел 3). В последующих разделах обсуждается, как можно повысить эффективность работы сети с использования информации о классах услуг (раздел 4). Передача сообщений в сети с учётом классов услуг рассматривается в разделе 5, в разделе 6 обсуждается настройка распределенной сети, и соответственно, в разделе 7 — проблемы инкрементальной настройки функционирующей сети. В разделах 4-7 описаны идеи предлагаемых алгоритмов; сами алгоритмы (кроме тех, что приведены в [1] и сохраняются без изменений) вынесены в приложение.

2. Модель распределённой сети

В качестве модели распределённой сети (далее, просто сети) рассматривается связный неориентированный граф без кратных рёбер и петель, в котором вершины — это хосты и коммутаторы, а рёбра — каналы связи, по которым передаются сообщения. Ребру $\{a, b\}$ соответствуют две ориентированные дуги ab и ba, вершины a и b называются *соседними*.

В [1] предложена модель распределённой сети, в которой функцию коммутации сообщений выполняют как коммутаторы, так и хосты. Сообщение, принятое вершиной графа от соседней вершины, пересылается другому (или тому же самому) соседу вершины. Это определяется правилами коммутации, каждое из которых имеет вид $(p: a, s, b)$, где a, s и b вершины, причём a и b соседние с s вершины (в графе есть дуги as и sb), p — параметры коммутации как та часть параметров сообщения, которая определяет выбор того или иного правила. Если сообщение принято вершиной s от соседа a и параметры сообщения соответствуют параметрам коммутации p в правиле вида $(p: a, s, b)$, то это правило срабатывает, и сообщение пересылается соседу b. Будем называть соседа a *предшественником*, а соседа b *преемником*. Мы 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. Также когда вершина s посылает сообщение преемнику b, идентификатор преемника b является не параметром сообщения, а параметром оператора посылки сообщения.

В данной статье мы предполагаем, что сообщение не клонируется, то есть может сработать только одно правило — нет двух правил, отличающихся только получателем b.

Коммутатор выполняет только функцию коммутации сообщений, тогда как хост, кроме этого, генерирует сообщения, передаваемые далее по сети, и именно хост является конечным получателем сообщений.

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

Сообщение может быть предназначено либо одному указанному хосту, либо «какому-нибудь» хосту, который может принять сообщение и обработать запрос, содержащийся в сообщении. В первом случае указывается идентификатор хоста-получателя, а во втором

случае — имя *услуги*, которую должен оказать хост-получатель хосту-отправителю. В первом случае сообщение принимается на обработку хостом, идентификатор которого указан в сообщении как идентификатор получателя, а во втором случае — каким-нибудь хостом, который может оказать запрашиваемую услугу, имя которой является параметром сообщения, такой хост в [1] назывался *целевым* хостом для данной услуги. Для этого в каждом хосте должно храниться множество имён услуг, которые этот хост может оказывать, т.е. реализованных в нём услуг. Если вершина, получившая сообщение, является коммутатором или хостом, который не может оказать запрашиваемую услугу, сообщение пересылается дальше согласно правилам коммутации.

Каждое сообщение содержит тип (имя) сообщения и набор параметров сообщения. Мы будем предполагать, что при передаче сообщения по сети может меняться только тип сообщения, а параметры сообщения передаются неизменными.

В [1] предложена установка таких правил коммутации в вершинах графа сети, которые порождают один (ориентированный) цикл, содержащий все хосты и называемый *циклом хостов*. Этот цикл строится как обход *дерева хостов*, которое определяется как поддерево остовного дерева графа с выделенным хостом-корнем (далее там, где это не приводит к двусмысленности, просто корнем), которое содержит все хосты, и все листовые вершины которого являются хостами (рис. 1). Заметим, что дерево хостов может содержать, кроме хостов, коммутаторы, необходимые для связности дерева. 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. Полученное дерево и будет деревом хостов.

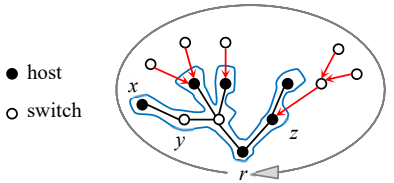

Рис. 1. Обход дерева хостов и лес деревьев коммутаторов.
Fig. 1. Host tree traversal and a forest of switch trees.

При обходе дерева хостов каждое ребро дерева хостов проходится ровно два раза, по одному разу в каждом направлении, т.е. цикл обхода является простым по дугам путём (не проходит дважды по одной дуге). Именно это свойство обеспечивает отсутствие клонирования сообщений. Для данного корневого дерева его обход, начиная с корня, 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; при обходе 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; при обходе дуга zr проходится последней.

За пределами дерева хостов оказывается часть дерева хостов, представляющая собой лес корневых поддеревьев остовного дерева, корни которых лежат на цикле хостов, а все остальные вершины являются коммутаторами. Будем называть эти поддеревья *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, может добавляться новый хост, подсоединяемый к коммутатору на дереве коммутаторов. Поэтому будет считать, что правила коммутации порождают, кроме цикла хостов, лес деревьев коммутаторов, ориентированных к своим корням (на рис. 1 отмечены красным цветом).

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.

3. Передача сообщений по циклу хостов

Сообщение с указанием идентификатора хоста-получателя имеет тип ***MessageToHost*** или ***RootMessageToHost***. Параметры сообщения: *sender* — идентификатор хоста-отправителя, *recipient* — идентификатор хоста-получателя, *parameters* — параметры сообщения, прозрачные для коммутации сообщений. Сообщение такого типа двигается по циклу хостов до хоста *recipient*, который и принимает сообщение на обработку, не пересылая его дальше. Для предотвращения бесконечного зацикливания сообщение, которое гарантированно прошло полный цикл хостов, удаляется. Поскольку цикл хост — это простой по дугам путь, каждая его дуга (но не обязательно вершина!) проходится один раз. Если сообщение ***MessageToHost*** проходит дугу от родителя корня в корень, не являющийся получателем сообщения, сообщение посылается дальше уже с типом ***RootMessageToHost***, а если по этой дуге проходит сообщение ***RootMessageToHost***, оно удаляется. Далее корень генерирует сообщение типа ***MessageToHost*** с отрицательным ответом хосту-отправителю. Последнее происходит, когда в сети нет хоста с требуемым идентификатором получателя. Тем самым, предотвращается бесконечная циркуляция по циклу хостов сообщений указанному хосту.

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-отправителя 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 некорректному поведению.

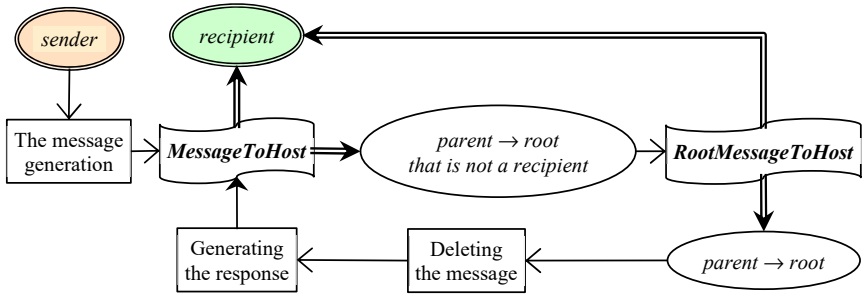

Рис. 2. Схема передачи по сети сообщения известному получателю.
Fig. 2. The network transmission diagram when sending a message to the known recipient.

Для генерации сообщения ***MessageToHost*** (***RootMessageToHost***) конкретному указанному хосту используется процедура ***SendMessageToHost*** с параметром *recipient* — идентификатор хоста, которому предназначено сообщение. Эта процедура посылает по циклу хостов сообщение ***MessageToHost***, если данный хост не корень дерева хостов, или сообщение ***RootMessageToHost*** в противном случае.

Сообщение с указанием имени запрашиваемой услуги может быть трёх типов: ***Message***, ***RootMessage*** или ***WaitingMessage***. Параметры сообщения: *sender* — идентификатор хоста-отправителя, *service* — имя запрашиваемой услуги, *parameters* — параметры сообщения, прозрачные для коммутации сообщений.

Сначала сообщение, как правило, посылается с типом ***Message***. Если оно проходит по дуге от родителя корня в корень, который не является целевым хостом, 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 по дуге, ведущей от родителя корня в корень, сообщение ***WaitingMessage***, пересылает его дальше опять с типом ***Message***. Последнее сделано для того, что «ловить» 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 «крутится» 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. 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).

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. В любом случае сообщение-ответ посылается с типом ***MessageToHost*** или (из корня) с типом ***RootMessageToHost***. В сообщении в качестве получателя указывается отправитель исходного сообщения, ответ на которое посылается.

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 является целевым хостом или корнем, ***RootMessage***, если отправитель является корнем, но не является целевым хостом,

WaitingMessage, если отправитель является целевым хостом. В последнем случае предполагается, что хост делает удалённый вызов потому, что сейчас «занят».

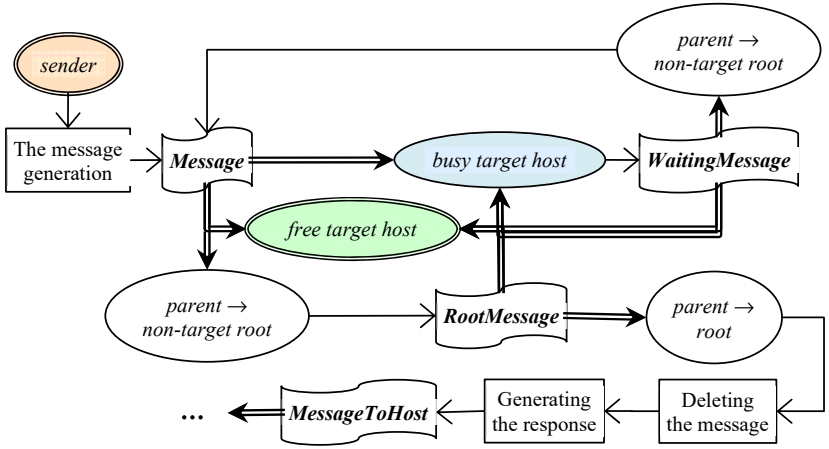

Рис. 3. Схема передачи по сети сообщения запроса услуги.
Fig. 3. The network transmission diagram when sending a service request message.

4. Классы услуг

У решения, предложенного в [1] и кратко описанного в предыдущем разделе, есть один существенный недостаток: путь, который должно пройти сообщение, прежде чем найдёт нужный хост, может оказаться слишком длинным. Например, если в сети запрашиваемая услуга реализована только в одном хосте *a*, то сообщение с этим именем 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 для каждой услуги обход не дерева хостов, а минимального дерева, содержащего, быть может, не все хосты, но все целевые хосты, т.е. хосты, в которых эта услуга реализована. Сообщение с запросом данной услуги двигалось бы не циклу хостов, а по циклу обхода такого минимального дерева.

Однако в этом случае имя услуги становится частью параметров коммутации. Поэтому всё зависит от того, сколь велико число услуг, реализуемых сетью, и имеет ли оно тенденцию к росту. Существуют сети с фиксированным множеством реализуемых услуг, однако могут быть и другие сети, которые предоставляют пользователям, вообще говоря, нефиксированный набор услуг, который к тому же может расти вместе с развитием сети. Например, облачные вычисления имеют такую характеристику, как эластичность — услуги могут быть предоставлены, расширены, сужены в любой момент времени. Для таких сетей предлагаемое решение с циклом для каждой услуги приведёт к слишком большому числу правил коммутации в каждом узле сети, к тому же имеющим тенденцию к росту при росте числа услуг, что можно понимать как немасштабируемость сети, зависимой от такого параметра как число услуг.

Для решения такой проблемы немасштабируемости мы предлагаем воспользоваться некоторой кластеризацией услуг. В этом случае множество всех услуг разбивается на подмножества, которые будем называть *классами услуг*. В сообщении указывается не только имя услуги, но также имя её класса, вычисляемого хостом-отправителем по заданному отображению имени услуги в имя её класса. Коммутация сообщения происходит по имени

класса услуги, т.е. имя класса становится параметром коммутации, правило коммутации сообщений с запросом услуг имеет вид (*class*: *a*, *b*, *c*). Хост принимает сообщение на обработку по имени услуги, если она в нём реализована и он сейчас не «занят». Предполагается, что число классов услуг мало, а число услуг велико, новые классы услуг появляются редко, тогда как новые услуги в классах могут появляться чаще, а кроме того, в сети число хостов, реализующих услуги данного класса, как правило, ощутимо меньше общего числа хостов.

Примером могут служить различные приложения, к которым можно получить онлайн-доступ через интернет. Класс арифметических вычислений (функции сложения, вычитания, умножения, деления и т.п.) выполняют различные онлайн-калькуляторы, например, *https://calculator888.ru/* или *https://okcalc.com/ru/*. Последний калькулятор может вычислять также логарифм, но эту функцию может выполнять и специальный калькулятор логарифма *https://umath.ru/calc/vychislenie-logarifma-chisla-onlajn/*. Предел функции вычисляется приложением *https://mathdf.com/lim/ru/*, но также входит в число функций, реализуемых приложением *https://mathsolver.microsoft.com/ru/algebra-calculator*, который, кроме этого, умеет решать линейные и квадратные уравнения, неравенства, вычислять производные и интегралы. Последние вычисляются и специальным калькулятором интегралов *https://www.integral-calculator.ru/*. График функции можно построить с помощью приложения *https://yotx.ru/*, это умеет делать и графический калькулятор Desmos (*https://www.desmos.com/?lang=ru*), но у него гораздо больше возможностей (реализованных функций). И так далее.

Мы привели примеры услуг и их группировки по разным приложениям. В то же время такие приложения — не то же самое, что наши классы услуг. Группировка услуг в приложениях выполняется по самым разным критериям, отличающимся в разных приложениях. Кроме того, такая группировка — это не разбиение (как у нас на классы), а покрытие множества услуг, поскольку одна и та же услуга может быть реализована в разных приложениях. Наше разбиение услуг на классы можно сделать устойчивым к изменениям множества услуг и, тем более, к их реализации и группировке в различных приложениях. Например, класс услуг по вычислению элементарных функций может включать арифметические действия, степенную, показательную, логарифмическую функции и тригонометрические и обратные тригонометрические функции, и не включать другие функции. Этот класс услуг не зависит от того, в каких приложениях реализованы элементарные функции (услуги) и есть ли приложение, реализующее в точности этот набор функций (услуг).

Далее будем считать, что разбиение множества услуг на классы задано. Теперь *целевой хост* будем определять не для услуги, а для класса услуг — это хост, реализующий хотя бы одну услугу этого класса. Мы будем строить *дерево целевых хостов* и *цикл целевых хостов* для каждого класса услуг. Дерево целевых хостов определяется как поддерево дерева хостов с корнем в целевом хосте, содержащее все целевые хосты, все листовые вершины которого являются целевыми хостами. Заметим, что дерево целевых хостов может содержать, кроме целевых хостов, коммутаторы и нецелевые хосты, необходимые для связности дерева. Цикл целевых хостов определяется как цикл обхода дерева целевых хостов, этот цикл порождается правилами коммутации. Дерево целевых хостов строится процедурой «Удаление «лишних» коммутаторов и нецелевых хостов», аналогичной процедуре «Удаление «лишних» коммутаторов» при построении дерева хостов, только вместо «лишних» коммутаторов в ней будут «лишние» коммутаторы и нецелевые хосты.

За пределами цикла целевых хостов для данного класса услуг могут остаться «лишние» нецелевые хосты. Если такой хост генерирует сообщение, запрашивающее услугу данного класса, то возникает вопрос, как сообщение попадёт на цикл целевых хостов? Для этого достаточно, чтобы правила коммутации порождали простые по дугам пути, которые начинаются в нецелевых хостах и заканчиваются в вершинах цикла целевых хостов, и вместе

с циклом целевых хостов суммарно содержат все хосты. Для того чтобы не было клонирования, нужно, чтобы эти пути не разветвлялись после прохода по общей дуге.

В [1] за пределами цикла хостов могли быть только коммутаторы, и на них строился лес деревьев коммутаторов, корни которых лежали на дереве хостов (рис. 1). Аналогично теперь для каждого класса услуг будем строить лес *деревьев нецелевых хостов*, который получается из остовного дерева удалением рёбер дерева целевых хостов и образующихся после такого удаления изолированных вершин. Дерево нецелевых хостов ориентировано к корню, лежащему на дереве целевых хостов, и все его некорневые вершины являются коммутаторами или нецелевыми хостами (рис. 4). В дерево нецелевых хостов входят коммутаторы, которые нужны, прежде всего, для связности дерева, но, кроме того, мы помещаем в дерево нецелевых хостов коммутаторы, не лежащие на путях по дереву из нецелевых хостов к корню, а именно, коммутаторы на деревьях коммутаторов (в том числе, терминальные коммутаторы), которые «лишние» для передачи сообщений по сети. Это делается по той же причине, по которой в [1] строились деревья коммутаторов, в которых вообще нет хостов. Такие деревья нужны для инкрементальной настройки сети [1], когда меняется топология сети (граф). Например, в сеть добавляется новый хост, соединяемый новыми рёбрами со «старыми» вершинами. Такими «старыми» вершинами могут быть эти «лишние» коммутаторы (некорневые коммутаторы деревьев коммутаторов), и они перестают быть «лишними».

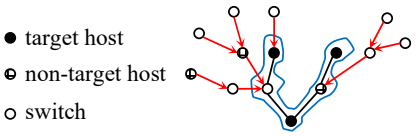

Рис. 4. Обход дерева целевых хостов и лес деревьев нецелевых хостов.
Fig. 4. Target host tree traversal and a forest of non-target host trees.

Для каждого класса услуг *class* в каждой вершине создаются свои «копии» переменных *Rules* и *Root*: *ClassToRules*(*class*) — правила коммутации для класса *class*, *ClassToRoot*(*class*) — отметка корня дерева целевых хостов для класса *class*.

5. Передача сообщений при наличии классов услуг

Сообщения с указанием идентификатора хоста-получателя типа ***MessageToHost*** или ***RootMessageToHost*** имеют те же параметры и передаются так же, как описано в разделе 3 и изображено на рис. 2, — по циклу хостов.

Сообщения с запросом услуги имеют те же три типа: ***Message***, ***RootMessage*** или ***WaitingMessage***. В параметры сообщения добавляется имя класса услуги: *sender* — идентификатор хоста-отправителя, *class* — имя класса услуги, *service* — имя запрашиваемой услуги, *parameters* — параметры сообщения, прозрачные для коммутации сообщений. Коммутация сообщений происходит по имени класса услуги, т.е. имя класса входит в параметры коммутации, правило коммутации сообщений с запросом услуг имеет вид (*class*: *a*, *b*, *c*).

Если сообщение с запросом услуги не находится на цикле целевых хостов (генерируется в нецелевом хосте на дереве нецелевых хостов и отлично от его корня), то сначала оно двигается по дереву нецелевых хостов к его корню, лежащему на цикле целевых хостов. Далее сообщение двигается (возможно, меняя свой тип) точно так же, как описано в разделе 3 и изображено на рис. 3, со следующими отличиями: 1) сообщение двигается не по циклу хостов, а по циклу целевых хостов для указанного класса услуг, 2) не всякий «свободный» целевой хост становится получателем сообщения, а только тот, в котором реализована

запрашиваемая услуга, 3) не всякий «занятый» целевой хост передаёт сообщение дальше под именем ***WaitingMessage***, а только тот, в ком реализована запрашиваемая услуга.

Для генерации сообщения запроса услуги используется процедура ***SendMessage***, как описано в разделе 3, с теми же параметрами. Отличие в том, что в сообщение добавляется имя класса запрашиваемой услуги, которое процедура вычисляет по имени услуги. Процедура возвращает ***false***, если некому послать сообщение с запросом услуги, т.е. в хосте нет правил коммутации для данного класса услуг: *ClassToRules*(*class*) = (). В противном случае возвращается ***true*** и посылается нужное сообщение запроса услуги с указанием её класса: ***Message*** — если в хосте не реализована запрашиваемая услуга, и хост не является корнем дерева целевых хостов, ***RootMessage*** — если в хосте не реализована запрашиваемая услуга, и хост является корнем дерева целевых хостов, ***WaitingMessage*** — если в хосте реализована запрашиваемая услуга, но сейчас хост «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, как и в [1], мы предлагаем алгоритмы самонастройки сети, без использования специального контроллера и с минимально необходимой информацией, заранее заданной в вершинах графа. Настройка разделяется на *первичную настройку* и *настройку для данного класса услуг*.

6.1 Первичная настройка сети

Первичная настройка, описанная в [1], определяет правила коммутации, порождающие цикл хостов и лес деревьев коммутаторов, а также отмечает корень дерева хостов. В результате первичной настройки в каждой вершине инициализируется список *Rules* правил коммутации и переменная *Root* := ***true*** в корне дерева хостов, *Root* := ***false*** в остальных хостах сети.

Все правила коммутации в вершине *s* имеют вид (: *a*, *s*, *b*), т.е. параметры коммутации отсутствуют, а средняя вершина одна и та же — *s*. Поэтому в алгоритмах для краткости в каждой вершине *s* правило коммутации записывается не как (: *a*, *s*, *b*), а как (*a*, *b*).

В каждой вершине *s* цикла хостов список *Rules* имеет «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*.

Мы также вводим *правило умолчания*: если 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). По сути, это *правило умолчания*: список (a_0, a_1), (a_1, a_2), ..., (a_{n-1}, a_0) понимается как сокращённая 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. Оно используется в трёх случаях: 1) при инкрементальной (повторной частичной) перенастройке сети, когда меняется граф сети; 2) при передаче сообщений по дереву нецелевых хостов в настроенной сети; 3) при посылке сообщения в сеть извне сети, т.е. при посылке сообщения 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. 5 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. 5 сверху), так и после применения правила умолчания (рис. 5 внизу).

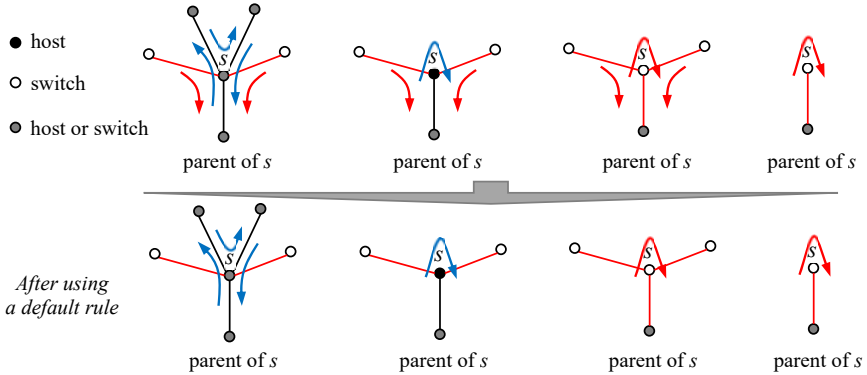

Рис. 5. Правила коммутации и правило умолчания.
Fig. 5. Switching rules and a default rule.

6.2 Настройка сети для заданного класса услуг

В данной статье мы вводим классы услуг, и для каждого класса услуг требуется своя дополнительная настройка сети, которая выполняется в предположении, что ранее была выполнена первичная настройка. Настройки для разных классов услуг выполняются независимо друг от друга, в том числе, они могут выполняться параллельно.

Настройка для заданного класса услуг определяет правила коммутации для этого класса услуг, порождающие цикл целевых хостов и лес деревьев нецелевых хостов.

В результате настройки для класса услуг *class* в каждой вершине инициализируется список *ClassToRules*(*class*) правил коммутации для этого класса услуг и переменная *ClassToRoot*(*class*) := ***true*** в корне дерева целевых хостов для этого класса услуг, *ClassToRoot*(*class*) := ***false*** в остальных хостах сети. Кроме того, каждой вершине сообщается множество *serviceSet* имён услуг данного класса услуг *class*, что позволяет инициализировать отображение *ServiceToClass* имени услуги в её класс для данного класса услуг: $\forall$ *service* $\in$ *serviceSet* *ServiceToClass*(*service*) := *class*.

Правило умолчания применяется и для правил коммутации для каждого класса услуг. Если на рис. 5 чёрный кружок понимать как целевой хост, белый кружок как коммутатор или нецелевой хост, а серый кружок как коммутатор или (любой) хост, то изображения на рисунке иллюстрируют правила умолчания для класса услуг.

Алгоритм настройки для данного класса услуг аналогичен алгоритму первичной настройки. Отличие в том, что при первичной настройке обходится весь граф, используя заранее инициализированный список соседей в каждой вершине, и строятся цикл хостов как обход дерева хостов и лес деревьев коммутаторов, а при настройке для заданного класса услуг список соседей не используется. В этом случае с использованием цикла хостов выполняется обход уже построенного дерева хостов, что определяется правилами коммутации, и ~~строится~~ цикл целевых хостов как обход дерева целевых хостов и лес деревьев нецелевых хостов.

Настройка для данного класса услуг инициируется сообщением ***ClassStart***, которое поступает в некоторый (произвольный) хост извне графа. Этот хост будем называть *инициатором* для данного класса услуг. 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 настройки. Во время настройки в сети циркулирует одно сообщение, которое имеет тип (первоначально ***ClassStart***) и параметры: *class* — имя класса услуг, *serviceSet* — множество имён услуг этого класса.

Настройка проходит в три этапа: ***A***, ***B*** и ***C***.

На этапе ***A*** хост, получающий сообщение ***ClassStart***, становится инициатором, и организует поиск целевого хоста. Для этого сообщение с типом ***SearchA*** двигается по циклу хостов до целевого хоста или, если такого хоста нет, возвращается в инициатор. По ходу дела происходит инициализация в каждой вершине переменной *ClassToRules*(*class*) := () (пока целевой хост не найден, цикла целевых хостов нет), и инициализация в каждом хосте переменных *ClassToRoot*(*class*) := ***false*** и *ServiceToClass*(*class*) как описано выше. Если в сети нет целевого хоста, так будут проинициализированы переменные во всех вершинах. Тогда инициатор посылает своему внешнему соседу отрицательный ответ ***CancelC***. Если целевой хост найден, он становится корнем дерева целевых хостов (*ClassToRoot*(*class*) := ***true***) и начинается этап ***B***.

На этапе ***B*** выполняется обход дерева хостов с выполнением процедуры «Удаление «лишних» коммутаторов и нецелевых хостов», описанной в разделе 4. Тем самым строится цикл целевых хостов как обход дерева целевых хостов и лес деревьев нецелевых хостов. Заметим, что часть этого леса, а именно деревья коммутаторов, построена при первичной настройке и при настройке для данного класса услуг не меняется. Попутно происходит инициализация в каждой вершине переменной *ClassToRules*(*class*) := *Rules* (первоначально цикл целевых хостов совпадает с циклом хостов) и в каждом хосте переменных *ClassToRoot*(*class*) := ***false*** и *ServiceToClass*(*class*) как описано выше. Последние две инициализации, если целевой хост найден, могут быть уже выполнены в некоторых хостах на этапе ***A***, но не во всех хостах.

На этапе ***B*** сообщение может иметь один из трёх типов: *прямое* сообщение ***ForwardB*** и два *ответных сообщения* (*ответа*) ***CancelB*** и ***BackB***. Сообщение ***ForwardB*** посылается по дуге дерева хостов, ориентированного от корня. Когда это сообщение первый раз попадает в некоторую вершину *s*, пройдя по дуге *xs*, в вершине *s* сначала устанавливаются правила коммутации для данного класса такие же, как для цикла хостов, однако начало отсчёта циклического списка правил может стать другим, поскольку для дерева целевых хостов родителем вершины *s* считается вершина *x*. Если для цикла хостов в вершине *s* был список правил *Rules* = $(a_0, a_1), (a_1, a_2), ..., (a_{n-2}, a_{n-1}), (a_{n-1}, a_0)$, где a_0 родитель вершины *s* в дереве хостов, то для $x = a_i$ в дереве целевых хостов сначала будет *ClassToRules*(*class*) = $(a_i, a_{i+1}), (a_{i+1}, a_{i+2}), ..., (a_{n-2}, a_{n-1}), (a_{n-1}, a_0), (a_0, a_1), (a_1, a_2), ..., (a_{i-1}, a_i)$, где $x = a_i$ родитель вершины *s* в дереве целевых хостов. Это объясняется тем, что в общем случае корень дерева хостов может не быть целевым хостом для данного класса услуг. Смена корня при переходе от дерева хостов к дереву целевых хостов приводит, прежде всего, к изменению ориентации рёбер, когда деревья ориентированы от их корней. Это показано на рис. 6, где чёрные стрелки изображают совпадающую ориентацию рёбер в дереве хостов и дереве целевых хостов с корнем в вершине *x*, а красные стрелки показывают ориентацию рёбер в дереве целевых хостов, которая противоположна их ориентации в дереве хостов; оба дерева ориентированы от их корней. Там, где ориентация рёбер меняется, у вершины меняется её вершина-родитель: для дуги *ab* вершина *a* является родителем вершины *b*, поэтому у циклического списка правил в вершине выбирается, быть может, другое начальное правило. В алгоритмах в Приложении это изменение делается для каждой вершины *s*, хотя его достаточно сделать для вершин на пути по дереву хостов от «старого» до нового корня *x*, поскольку в остальных вершинах начальное правило списка правил не меняется, что видно по рис. 6.

Когда все выходящие из вершины *b* дуги дерева хостов, кроме дуги, ведущей к её родителю *a*, уже пройдены, вершина *b* посылает своему родителю *a* ответное сообщение. Это ответное

сообщение имеет тип ***BackB***, если поддерево дерева хостов с корнем в вершине *b* содержит целевой хост, или, в противном случае, тип ***CancelB***. В этом поддереве есть целевые хосты, если вершина *b* является целевым хостом, а также, если либо в вершине *b* определено больше одного правила коммутации (при использовании правила умолчания, рис. 5 внизу), либо в этих правилах есть хотя бы два разных преемника (без использования правила умолчания, рис. 5 вверху). Если посылается ответ ***BackB***, вершина *b* войдёт в дерево целевых хостов, а если посылается ответ ***CancelB***, вершина *b* войдёт в дерево нецелевых хостов. Тем самым, будет выполняться процедура «Удаление «лишних» коммутаторов и нецелевых хостов». Соответствующим образом корректируются правила коммутации для данного класса услуг. Этап ***B*** заканчивается, когда пройден весь цикл хостов, тогда начинается этап ***C***.

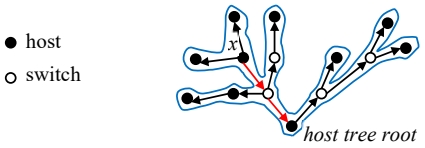

Рис. 6. Для дерева, ориентированного от корня, смена ориентации рёбер при смене корня.
Fig. 6. Changing the edge orientation when the root is changed for a root-oriented tree.

На этапе ***C*** происходит возврат в инициатор, сообщение двигается по циклу хостов, пока не окажется в инициаторе. Инициатор посылает своему внешнему соседу положительный ответ на настройку типа ***BackC***. Напомним, что отрицательный ответ посылается в том случае, когда в сети нет целевых хостов, что обнаруживается на этапе ***A***.

7. Инкрементальная настройка сети

Под инкрементальной настройкой сети так же, как в [1], будем понимать повторную и частичную перенастройку сети, необходимую при изменении сети. В [1] рассматривалась инкрементальная настройка как частичное изменение первичной настройки, т.е. без класса услуг. Основные идеи этой инкрементальной настройки применимы и после введения классов услуг. Нужно только понимать, что изменения сети могут касаться не только цикла хостов и леса деревьев коммутаторов, но также цикла целевых хостов и леса деревьев нецелевых хостов для того или иного класса услуг. В то же время, правила коммутации без класса услуг и для данного класса услуг, а также для разных классов услуг не пересекаются, так как определяются параметрами коммутации: пустым параметром — правила вида (: *a*, *s*, *b*), или классом услуг *class* — правила вида (*class*: *a*, *s*, *b*). Поэтому в данной статье мы не рассматриваем инкрементальную настройку без класса услуг (она такая же, как в [1]) и ограничиваемся рассмотрением инкрементальной настройки для *одного* класса услуг. При некоторых изменениях сети такую инкрементальную настройку нужно выполнить для каждого класса услуг, который затрагивается этим изменением.

Изменение сети касается либо 1) распределения реализаций услуг по хостам сети, либо 2) топологии сети (изменение графа сети), либо 3) распределения услуг по классам. 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 или нецелевого хоста и одного ребра, соединяющего его со «старой» вершиной графа, 2.5) добавление одного целевого хоста и одного ребра, соединяющего его со «старой» вершиной графа, 2.6) удаление одного терминального коммутатора или нецелевого хоста и инцидентного ему ребра, 2.7) удаление одного терминального целевого хоста и инцидентного ему ребра.
3) При изменении распределения услуг по классам элементарными изменениями являются: 3.1) удаление одной услуги из класса, 3.2) добавление одной услуги в класс, 3.3) удаление пустого класса, 3.4) добавление пустого класса.

Примером более сложного изменения сети является удаление целевого 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 целевого хоста и инцидентного ему ребра (2.7). Другой пример — удаление класса услуг, которому соответствует последовательность удалений услуг из класса (3.1), пока он не станет пустым, а затем удаление пустого класса (3.3).

В отличие от предыдущих разделов здесь мы будет давать 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 выполнять «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, т.е. возможности передавать сообщения от любого хоста-отправителя любому известному хосту или с запросом любой услуги. 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 целевых хостов или нет. У нас нет соответствующей отметки (булевской переменной) в вершинах графа, но без неё можно обойтись. Вершина лежит на цикле целевых хостов тогда и только тогда, когда она является целевым хостом или это коммутатор или нецелевой хост, и в нём либо определено больше одного правила коммутации (при использовании правила умолчания, рис. 5 внизу), либо в его правилах есть хотя бы два разных преемника (без использования правила умолчания, рис. 5 вверху). Напомним, что для цикла целевых хостов на рис. 5 чёрный кружок нужно понимать как целевой хост, белый кружок как коммутатор или нецелевой хост, а серый кружок как коммутатор или (любой) хост.

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 нужному хосту, если в сообщении верно указаны имя услуги и имя её класса, и в сети есть хосты, в которых реализована эта услуга. 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 (целевых) 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 a нужно выполнить в хосте a *Services* := *Services* $\cup$ {*service*}. Если в хосте a была реализована другая услуга того же класса, никаких действий не требуется. В противном случае хост a становится новым целевым хостом для этой услуги, и его нужно добавить в цикл целевых хостов. Для этого на пути по дереву нецелевых хостов от хоста a до цикла целевых хостов нужно поменять правила коммутации в вершинах пути, кроме вершины a, у которой не меняется родитель, и поэтому с учётом правила умолчания в ней изменения не нужны (рис. 7).

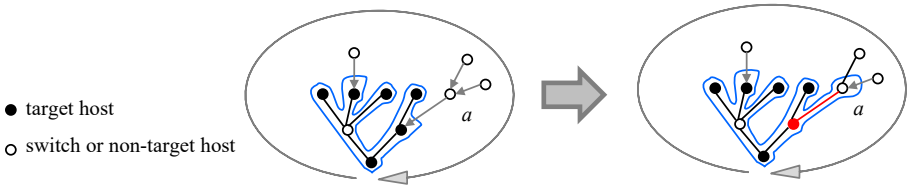

Рис. 7. Нецелевой хост становится целевым хостом.
Fig. 7. A non-target host becomes a target host.

7.1.2 Удаление реализации одной услуги из одного хоста

При удалении реализации услуги с именем *service* из некоторый хоста a нужно выполнить в хосте a *Services* := *Services* \ {*service*}. Если в хосте a осталась реализация другой услуги того же класса, никаких действий не требуется. В противном случае хост a перестаёт быть целевым хостом для этой услуги, и его можно было бы удалить из цикла целевых хостов. Однако это нужно только в целях оптимизации (уменьшения длины цикла целевых хостов), поскольку наличие нецелевых хостов в цикле целевых хостов не влияет на функциональность сети. Относительно просто такую оптимизацию можно выполнить, когда хост a является листовой вершиной дерева целевых хостов, т.е. ему инцидентно в дереве единственное ребро $\{a, b\}$: меняются правила коммутации в вершине b (рис. 8).

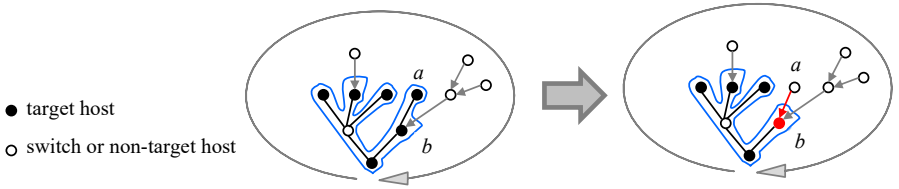

Рис. 8. Листовой целевой хост становится нецелевым хостом.
Fig. 8. A leaf target host becomes a non-target host.

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 целевых хостов и леса деревьев нецелевых хостов. Для того чтобы задать новое упорядочивание графа, нужно в каждой вершине, в которой изменился линейный порядок соседей, установить новый список идентификаторов соседних вершин

(отличающийся от старого порядком 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 циклов. Тогда нужно выполнить первичную настройку и далее настройку для каждого класса услуг.

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 целевых хостов. Для подготовки к следующей настройке при добавлении ребра $\{a, b\}$ в список соседей вершины a добавляется идентификатор соседа b, а в список соседей вершины 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 соседей вершины a удаляется идентификатор соседа b, а из списка соседей вершины b удаляется идентификатор соседа a.

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

Если удаляемое ребро лежит на дереве нецелевых хостов, его удаление нарушает связность этого дерева (рис. 9). 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 нецелевых хостов 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 целевых хостов, его удаление нарушает связность этого дерева (рис. 10). 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 целевых 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 или нецелевого хоста и одного ребра, соединяющего его со «старой» вершиной графа

Для подготовки к следующей настройке при добавлении коммутатора или нецелевого хоста a и ребра $\{a, b\}$ создаётся список (b) соседей вершины a и в список соседей вершины b добавляется идентификатор соседа a. Если добавляется коммутатор, для сохранения функциональности сети перенастройка не требуется. Если добавляется нецелевой хост, его нужно включить в дерево нецелевых хостов, а также сообщить ему имена услуг класса (для отображения услуг в класс). В любом случае при дальнейших изменениях топологии сети во время соответствующей перенастройки сети нужно учитывать добавляемые вершину и ребро. Поскольку при добавлении коммутатора или нецелевого хоста a только с одним инцидентным ему ребром $\{a, b\}$, вершина a будет терминальной, она может входить только в дерево нецелевых хостов. Правила коммутации устанавливаются в добавляемой вершине a

и меняются в другом конце b добавляемого ребра (рис. 11). Однако с учётом правила умолчания достаточно только в вершине a установить правило коммутации $\{: b, a, b\}$.

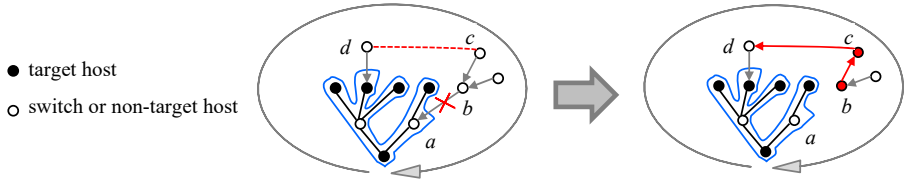

Рис. 9. Удаление ребра на дереве нецелевых хостов.
Fig. 9. Deleting an edge of the non-target host tree.

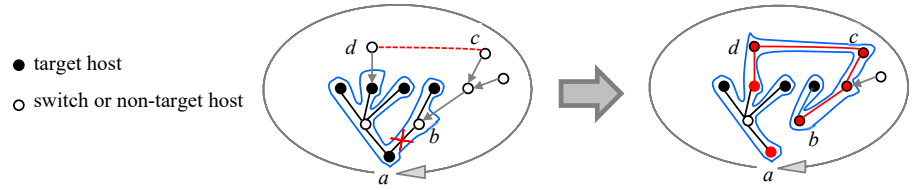

Рис. 10. Удаление ребра на дереве целевых хостов.
Fig. 10. Deleting an edge of the target host tree.

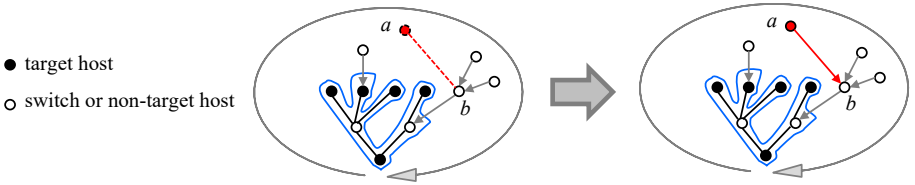

Рис. 11. Добавление одного коммутатора или нецелевого хоста и одного ребра, соединяющего его со «старой» вершиной графа.
Fig. 11. Adding a switch or a non-target host and an edge for its connection with the "old" node of the graph.

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

Когда добавляется целевой хост a и ребро $\{a, b\}$, соединяющее его со «старой» вершиной b, нужно создать список (b) соседей хоста a, в список соседей вершины b добавить идентификатор соседа a и добавить хост a в цикл целевых хостов. Кроме того, нужно сообщить хосту a имена услуг класса (для отображения услуг в класс). Правила коммутации устанавливаются в добавляемом хосте a, и меняются в вершине b; кроме того, если вершина b не лежит на цикле целевых хостов, то она является некорневой вершиной некоторого дерева нецелевых хостов, и нужно поменять правила коммутации на всём пути от вершины b до корня этого дерева (рис. 12).

7.2.6 Удаление одного терминального коммутатора или нецелевого хоста и инцидентного ему ребра

Связность графа не нарушается, перенастройка не требуется. Для подготовки к следующей настройке при удалении терминального коммутатора или нецелевого хоста a и ребра $\{a, b\}$ из списка соседей вершины b удаляется идентификатор соседа a.

7.2.7 Удаление одного терминального целевого хоста и инцидентного ему ребра

При удалении терминального целевого хоста a и (единственного) инцидентного ему ребра $\{a, b\}$ связность графа не нарушается. Для подготовки к следующей настройке из списка соседей вершины b удаляется идентификатор соседа a. Удаляемый хост нужно удалить из цикла целевых хостов. Поскольку хост терминальный, он является корнем или листовой вершиной дерева целевых хостов. С учётом правила умолчания достаточно изменить правила коммутации в вершине b (рис. 13). Кроме того, если удаляется корень дерева целевых хостов, нужно отметить в качестве корня другой (всё равно какой) целевой хост.

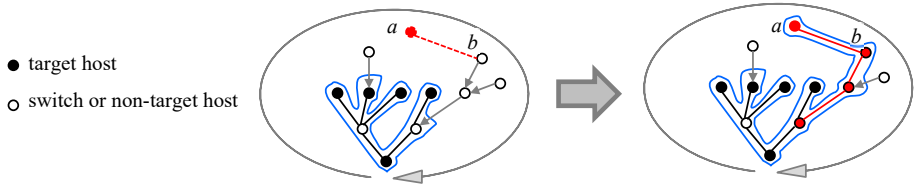

Рис. 12. Добавление одного целевого хоста и одного ребра, соединяющего его со «старой» вершиной графа.
Fig. 12. Adding a target host and an edge connecting that connects this host with the "old" graph node.

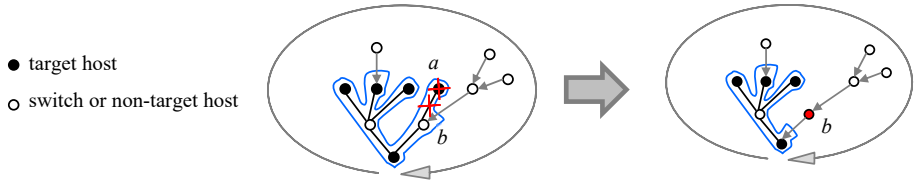

Рис. 13. Удаление одного терминального целевого хоста и инцидентного ему ребра.
Fig. 13. Deleting a terminal target host and its incident edge.

7.3 Изменение распределения услуг по классам

7.3.1 Удаление одной услуги из класса

При удалении услуги из класса нужно сообщить об этом всем хостам, чтобы скорректировать отображение услуг в классы: удалить отображение этой услуги в её класс. Для сохранения функциональности сети перенастройка сети не требуется. Однако в результате такого удаления некоторые хосты, которые были целевыми для данного класса, могут стать нецелевыми (если в них была реализована единственная услуга, удаляемая из класса). Поэтому перенастройка может быть полезна в целях оптимизации: уменьшения длины цикла целевых хостов.

7.3.2 Добавление одной услуги в класс

При добавлении услуги в класс нужно сообщить об этом всем хостам, чтобы скорректировать отображение услуг в классы: добавить отображение этой услуги в её класс. Кроме того, в результате такого добавления некоторые хосты, которые не были целевыми для данного класса, могут стать целевыми (в них реализована добавляемая услуга). Каждый такой хост должен быть включён в цикл хостов, что делается аналогично инкрементальной настройке при добавлении реализации одной услуги в один хост (рис. 7).

7.3.3 Удаление или добавление пустого класса

Никаких действий не требуется.

8. Заключение

Можно указать следующие направления исследований в рамках модели распределённой сети, предложенной в [1] и в данной статье:

1) Мы не исследовали детально (инкрементальную) настройку сети, если она не нужна для сохранения функциональности сети, но полезна для повышения эффективности работы сети, а только указывали на возможную оптимизирующую настройку. Было бы интересно исследовать влияние на эффективность работы сети таких факторов, как распределение услуг по классам и распределение реализаций услуг по хостам в зависимости от топологии сети.
2) Для улучшения эффективности (скорости передачи сообщений) и надёжности сети полезно учитывать соответствующие нефункциональные параметры. Например, можно рассматривать взвешенные графы, в которых рёбрам/вершинам приписаны веса, моделирующие время прохождения сообщения через ребро/вершину (эффективность) или вероятность сбоя при прохождении сообщения через ребро/вершину (надёжность). Соответственно, при построении путей эти параметры желательно учитывать. Для предлагаемых алгоритмов это означает выбор наилучшего дерева хостов, т.е. наилучшего упорядочивания графа и наилучшего корня дерева хостов, выбор наилучшего дерева целевых хостов, т.е. наилучшего корня дерева целевых хостов для данного класса услуг при заданном упорядочивании графа, а также учёт указанных параметров в тех инкрементальных настройках, которые рассмотрены в [1] и этой статье. Также интересно исследовать возможность самонастройки сети: по всей видимости, какие-то решения проблем оптимизации позволяют использовать самонастройку сети, а какие-то нет, требуя предварительного глобального анализа сети (не в процессе самонастройки).

Список литературы / References

[1]. И. Б. Бурдонов, Н. В. Евтушенко, А. С. Косачев, В. Н. Пономаренко. Модель распределённой сети, в которой хосты могут выполнять функцию коммутации сообщений. Труды института системного программирования. 2025, т. 37. № 4, с. 159-172.

[2]. 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.

[3]. 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.

[4]. OpenNetworkingFoundation (2012). Software-defined networking: The new norm for networks. ONF White Paper. 2012.

[5]. 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, pages 421-428. DOI: 10.5220/0010494504210428.

[6]. 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.

[7]. 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.

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,
class — имя класса услуг,
set(*type*) — множество элементов типа *type*,
list(*type*) — список элементов типа *type*,
(*type*$_1$, *type*$_2$) — пара (элемент типа *type*$_1$, элемент типа *type*$_2$),
type$_1$→*type*$_2$ — отображение элемента типа *type*$_1$ в элемент типа *type*$_2$, эквивалентно ***set***((*type*$_1$, *type*$_2$)) с ограничением: любое значение встречается в левых частях пар не более одного раза.

П2. Глобальные переменные в вершине — хосте или коммутаторе

vertex *Self* = ...; /* инициализировано, не меняется: собственный идентификатор вершины */
list(***vertex***, ***vertex***) *Rules*; /* список правил коммутации как список пар (предшественник, преемник) */
/* без класса услуг (для цикла хостов и леса деревьев коммутаторов */
class → ***list***(***vertex***, ***vertex***) *ClassToRules*; /* список правил коммутации для данного класса услуг */
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 */
class → ***vertex*** *ClassToExternal*; /* идентификатор внешнего соседа инициатора */
/* для данного класса услуг */
class → ***Bool*** *ClassToInitiator*; /* признак инициатора для данного класса услуг */
class → ***Bool*** *ClassToRoot*; /* признак корня дерева целевых хостов для данного класса услуг */
service → ***class*** *ServiceToClass*; /* отображение имени услуги в имя её класса */

П4. Использование глобальных переменных

Глобальные переменные, используемые на этапе самонастройки сети для данного класса услуг: *Rules*, *ClassToRules*, *Host*, *Services*, *ClassToExternal*, *ClassToInitiator*, *ClassToRoot*, *ServiceToClass* (все, кроме *Self*).

Глобальные переменные, используемые при передаче сообщений запроса услуг по настроенной сети: *Self*, *Rules*, *ClassToRules*, *Host*, *Services*, *ClassToRoot* (все, кроме *ClassToExternal*, *ClassToInitiator*, *ServiceToClass*); при генерации сообщений запроса услуг: *ClassToRules*, *ClassToRoot*, *ServiceToClass*.

П5. Правила коммутации

Правила коммутации *rules* = *Rules* без класса услуг или *rules* = *ClassToRules*(*class*) для класса услуг *class* 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 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*[$i..j$] = ();

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 Successor(list(vertex, vertex) rules, vertex x) { /* эта процедура такая же, как в [1] */
    /* 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 */
list(vertex, vertex) RulesReordering(vertex x) /* изменение начала циклического списка правил */
    n = |Rules|; i : = 1; /* первым правилом становится правило вида (x, b) */
    while i ≤ n & Rules[i][1] ≠ x do { i := i + 1; }
    return rules[i][n]^rules[1][i - 1];
```

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-предшественника, от которого принято сообщение, *y* идентификатор соседа-преемника, которому посылается сообщение.

Предполагается, что выполнена первичная настройка: построены цикл хостов и лес деревьев коммутаторов, в каждой вершине инициализированы переменные *Self*, *Rules*, *Host*, *Root*, *Services*. Алгоритмы первичной настройки приведены в [9], там же приведены процедуры обработки сообщений указанному хосту ***MessageToHost*** и ***RootMessageToHost*** и процедура генерации таких сообщений ***SendMessageToHost***. Процедуры запроса услуг ***Message***, ***RootMessage*** и ***WaitingMessage***, а также процедура ***SendMessage*** генерации сообщения запроса услуг, приведённые ниже, отличаются от тех, что даны в [1].

П7.1 Самонастройка сети для данного класса услуг

vertex *x* — идентификатор соседа, от которого получено сообщение,
class *class* — имя класса услуг,
set(***service***) *serviceSet* — множество имён услуг этого класса.

П7.1.1 Этап A: поиск целевого хоста

```
ClassStart(vertex x, class class, set(service) serviceSet) {
    /* старт настройки, сообщение пришло в хост-инициатор от его внешнего соседа x */
    for ∀ service ∈ serviceSet do { /* отображение услуг в их класс */
        ServiceToClass(service) : = class; }
    if Rules = () { /* если инициатор изолированная вершина, то */
        ClassToRules(class) := (); /* правил нет */
        if Services ∩ serviceSet ≠ ∅ { /* если инициатор целевой хост, то */
            ClassToRoot(class) := true; /* инициатор становится корнем дерева целевых хостов */
            SEND(BackC(class, serviceSet), x); } /* положительный ответ на настройку */
        else { /* если инициатор не целевой хост, то */
            ClassToRoot(class) := false; /* инициатор не является корнем дерева целевых хостов */
            SEND(CancelC(class, serviceSet), x); } } /* отрицательный ответ на настройку */
    else { /* если инициатор не изолированная вершина, то */
        ClassToInitiator(class) := true; /* хост становится инициатором */
        ClassToExternal(class) := x; /* запоминаем идентификатор x внешнего соседа инициатора */
        if Services ∩ serviceSet ≠ ∅ { /* если инициатор целевой хост, то */
            ClassToRules(class) := Rules; /* сначала цикл целевых хостов совпадает с циклом хостов */
            ClassToRoot(class) := true; /* инициатор становится корнем дерева целевых хостов */
```



```
            /* прямое сообщение этапа B соседу a1 */
            SEND(ForwardB(class, serviceSet), Rules[1][2]); } }
        else { /* если инициатор не целевой хост, то */
            ClassToRules(class) := (); /* сначала цикла целевых хостов нет */
            ClassToRoot(class) := false; /* инициатор не является корнем дерева целевых хостов */
            /* поиск целевого хоста, начиная с соседа a1 */
            SEND(SearchA(class, serviceSet), Rules[1][2]); } } }
SearchA(vertex x, class class, set(service) serviceSet) { /* этап A, поиск целевого хоста */
    y := Successor(Rules, x); /* сосед-преемник y соседа-предшественника x в цикле хостов */
    if x = Rules[1][1] { /* если пришли в вершину от её родителя a0 = x */
        if Host = false ∨ ClassToInitiator(class) = false {
            /* если вершина не инициатор, то первый раз на этапе A пришли в вершину, тогда */
            if Host = true { /* если вершина хост, то */
                for ∀ service ∈ serviceSet do { /* отображение услуг в их класс */
                    ServiceToClass(service) : = class; }
                if Host = true & Services ∩ serviceSet ≠ ∅ { /* если вершина целевой хост, то */
                    ClassToRules(class) := Rules; /* сначала цикл целевых хостов равен циклу хостов */
                    ClassToRoot(class) := true; /* вершина становится корнем дерева целевых хостов */
                    /* прямое сообщение этапа B вершине y = a1 */
                    SEND(ForwardB(class, serviceSet), Rules[1][2]); }
                else { /* если вершина не целевой хост, то */
                    ClassToRules(class) := (); /* сначала цикла целевых хостов нет */
                    if Host = true { /* если вершина хост, то */
                        ClassToRoot(class) := false; /* очистка признака корня целевых хостов */
                        SEND(SearchA(class, serviceSet), y); } } } /* идём дальше по циклу хостов */
            else { /* если пришли в инициатор, то цикл хостов пройден и целевых хостов нет */
                ClassToInitiator(class) := false; /* конец настройки, очистка для следующей настройки */
                SEND(CancelC(class, serviceSet), External(class)); } /* отрицательный ответ на настройку */
        else { /* если цикл хостов не пройден, то */
            SEND(SearchA(class, serviceSet), y); } } /* идём дальше по циклу хостов */
```

П7.1.2 Этап B: построение цикла целевых хостов и деревьев коммутаторов

```
ForwardB(vertex x, class class, set(service) serviceSet) { /* этап B, прямое сообщение */
    /* первое сообщение, получаемое вершиной на этапе B, новым родителем становится вершина x */
    ClassToRules(class) := RulesReordering(x); /* меняем начало циклического списка правил (a0 = x) */
    if Host = true { /* если вершина хост, то запоминаем отображение услуг в их класс */
        for ∀ service ∈ serviceSet do { ServiceToClass(service) : = class; }
        ClassToRoot(class) := false; } /* очистка признака корня целевых хостов */
    if |ClassToRules(class)| = 1 { /* если это листовая вершина дерева хостов, то */
        if Host = true & Services ∩ serviceSet ≠ ∅ { /* если это целевой хост, то */
            SEND(BackB(class, serviceSet), x); } /* положительный ответ родителю a0 = x */
        else { /* если это не целевой хост, то */
            SEND(CancelB(class, serviceSet), x); } } /* отрицательный ответ родителю a0 = x */
    else { /* если это не листовая вершина дерева хостов, то */
        /* прямое сообщение соседу a1 */
        SEND(ForwardB(class, serviceSet), ClassToRules(class)[1][2]); } }
```

```
CancelB(vertex x, class class, set(service) serviceSet) { /* этап B, отрицательный ответ */
    y := Successor(ClassToRules(class), x); /* преемник y предшественника x в цикле правил */
    /* Корректировка правил коммутации для данного класса услуг */
    n := |ClassToRules(class)|; i := 1; /* определяем индекс правила, в котором вершина x преемник */
    while i ≤ n & ClassToRules(class)[i][2] ≠ x do { i := i + 1; }
    if i < n { /* x = a_i, i < n, текущая вершина не корень */
        /* меняем правило (a_{i-1}, a_i) → (a_{i-1}, a_{i+1}) */
        ClassToRules(class)[i][2] := ClassToRules(class)[i + 1][2]; {
        /* удаляем правило (a_i, a_{i+1}) */
        ClassToRules(class) := ClassToRules(class)[1..i]^ClassToRules(class)[i + 2..n]; }
    else { /* если x = a_n = a_0, то текущая вершина корень, тогда */
        ClassToRules(class)[1][1] := ClassToRules(class)[n][1]; /* меняем правило (a_0, a_1) → (a_{n-1}, a_1) */
        ClassToRules(class) := ClassToRules(class)[1..n - 1]; } /* удаляем правило (a_{n-1}, a_0) */
    CancelBackB(y, class, serviceSet); } /* окончание в процедуре CancelBackB */
BackB(vertex x, class class, set(service) serviceSet) { /* этап B, положительный ответ */
    y := Successor(ClassToRules(class), x); /* преемник y предшественника x в цикле правил */
    CancelBackB(y, class, serviceSet); } /* окончание в процедуре CancelBackB */
CancelBackB(vertex y, class class, set(service) serviceSet) { /* окончание процедур CancelB и BackB */
    if y = ClassToRules(class)[1][2] { /* если y = a_1, то прошли цикл хостов и пришли в корень */
        if ClassToInitiator(class) = true { /* если инициатор, то */
            ClassToInitiator(class) := false; /* конец настройки, очистка для следующей настройки */
            /* положительный ответ внешнему соседу */
            SEND(BackC(class, serviceSet), ClassToExternal(class)); }
        else { /* если не инициатор, то положительный ответ посылается инициатору */
            SEND(BackC(class, serviceSet), Rules[1][2]); } } /* по циклу хостов соседу a_1 */
    else { /* если не прошли цикл хостов, то */
        if y = ClassToRules(class)[1][1] {
            /* если y = a_0 (родитель), то прошли поддерево с корнем в этой вершине, тогда */
            if Host = true & Services ∩ serviceSet ≠ ∅ ∨ |ClassToRules(class)| > 1 {
            /* если в поддереве с корнем в данной вершине есть целевой хост то, */
                SEND(BackB(class, serviceSet), y); } /* положительный ответ родителю y = a_0 */
            else { /* если в поддереве с корнем в данной вершине нет целевого хоста, то */
                SEND(CancelB(class, serviceSet), y); } } /* отрицательный ответ родителю y = a_0 */
        else { /* если y ≠ a_0 (не родитель), то не прошли поддерево с корнем в этой вершине */
            /* прямое сообщение этапа B дальше по циклу целевых хостов */
            SEND(ForwardB(class, serviceSet), y); } } }
```

П7.1.3 Этап C: положительный ответ на настройку сети

Отрицательный ответ ***CancelC*** посылается во внешнюю вершину в процедурах ***ClassStart*** и ***SearchA***.

```
BackC(vertex x, class class, set(service) serviceSet) { /* этап C, положительный ответ на настройку */
    y := Successor(Rules, x); /* сосед-преемник y соседа-предшественника x в цикле хостов */
    if Host = true & ClassToInitiator(class) = true { /* если вершина инициатор, то */
        ClassToInitiator(class) := false; /* конец настройки, очистка для следующей настройки */
        /* положительный ответ внешнему соседу */
        SEND(BackC(class, serviceSet), ClassToExternal(class)); }
    else { /* если вершина не инициатор, то */
        SEND(BackC(class, serviceSet), y); } } /* положительный ответ по циклу хостов инициатору */
```


П7.2 Передача сообщений запроса услуг по настроенной сети

Обработка сообщений запроса услуги отличаются от обработки одноимённых сообщений в [1] только добавлением в число параметров параметра ***class*** *class*, а также именами переменных: *ClassToRules*(*class*) вместо *Rules* и *ClassToRoot*(*class*) вместо *Root*.

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

vertex *sender* — идентификатор хоста-отправителя, применяется для посылки ответа отправителю,

class *class* — имя класса запрашиваемой услуги,

service *service* — имя запрашиваемой услуги,

parameters — параметры сообщения, прозрачные для коммутации сообщений.

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

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

WaitingMessage(vertex x, vertex sender, class class, service service, parameters) {
    /* сообщение запроса услуги, ожидающее освобождения хоста, который может оказать услугу */
    y := Successor(ClassToRules(class), x); /* сосед-преемник y соседа-предшественника x */
```

```
    if Host ≠ true ∨ service ∉ Services { /* если вершина не реализует запрашиваемую услугу, то */
        if x ≠ ClassToRules(class)[1][1] ∨ Host ≠ true ∨ Root = false {
            /* если пришли не по дуге от родителя корня в корень, то */
            /* посылаем сообщение дальше по циклу целевых хостов */
            SEND(WaitingMessage(sender, class, service, parameters), y); }
        else { /* если пришли по дуге от родителя корня в корень, то */
            /* посылаем сообщение дальше по циклу целевых хостов */
            SEND(Message(sender, class, service, parameters), y); } }
    else { /* если вершина хост, реализующий запрашиваемую услугу, то */
        if HostReadyToExecuteService(service, parameters) = false { /* если хост «занят», то */
            /* посылаем сообщение дальше по циклу целевых хостов */
            SEND(WaitingMessage(sender, class, service, parameters), y); }
        else { /* если хост «свободен», то */
            service(sender, parameters); } } } /* локальный вызов запрашиваемой услуги */
```

П8. Вызов удалённой услуги из хоста

service *service* — имя запрашиваемой услуги,
parameters — параметры сообщения, прозрачные для коммутации сообщений.

```
bool SendMessage(service service, parameters); { /* вызов из хоста удалённой услуги */
    class : = ServiceToClass(service); /* определяем класс услуг */
    if ClassToRules(class) = () { /* если нет правил, то */
        return false; }
    y := ClassToRules(class)[1][2]; /* посылаем следующему соседу a1 */
    if service ∉ Services { /* если в хосте не реализована запрашиваемая услуга, то */
        if ClassToRoot(class) = true { /* если хост корень дерева целевых хостов, то */
            SEND(RootMessage(Self, class, service, parameters), y); } /* по циклу целевых хостов */
        else { /* если хост не корень цикла целевых хостов, то */
            /* посылаем сообщение по пути до цикла целевых хостов и далее по циклу */
        SEND(Message(Self, class, service, parameters), y); }
    else { /* если в хосте реализована запрашиваемая услуга (но хост сейчас занят), то */
            /* посылаем сообщение по циклу целевых хостов */
        SEND(WaitingMessage(Self, class, service, parameters), y); }
    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.