ISSN 1064-2307, Journal of Computer and Systems Sciences International, 2018, Vol. 57, No. 3, pp. 407–419. © Pleiades Publishing, Ltd., 2018.
Original Russian Text © I.S. Petrov, R.L. Smeliansky, 2018, published in Izvestiya Akademii Nauk, Teoriya i Sistemy Upravleniya, 2018, No. 3, pp. 64–75.

Minimization of Multicast Traffic and Ensuring Its Fault Tolerance in Software-Defined Networks

I. S. Petrov[a],* and R. L. Smeliansky[a],**

[a]*Moscow State University, Moscow, 119991 Russia*

**e-mail: ipetrov@cs.msu.ru*

***e-mail: smel@cs.msu.ru*

Received November 27, 2017; in final form, January 12, 2018

Abstract—In recent years, the widespread TCP/IP computer network model has been replaced by the software-defined network model, where the control plane is separated from the data plane and is logically centralized. The new model requires a revision of traditional network control protocols. One group of such revised protocols consists of multicast routing protocols. In this paper, the multicast routing protocols used in traditional TCP/IP networks are analyzed, and their basic disadvantages and difficulties in their application in software-defined networks are revealed. Multicast routing algorithms that use the capabilities of software-defined networks and ensure the optimization and reliability of routes in multicast routing are described. These algorithms are exempt from the drawbacks of traditional networks. The proposed algorithms are implemented as an application for the RunOS controller. This experimental study shows that the delays due to the use of the proposed algorithms for restructuring routes satisfy the requirements of telecommunications operators of large regional networks.

DOI: 10.1134/S1064230718030085

INTRODUCTION

Multicast routing is an efficient means of transmitting data from a single server to a group of clients. In multicast routing, the packets sent to different clients are duplicated by certain network devices in the process of transmission rather than by the sender, as is the case in unicast routing. This approach allows us to reduce the traffic in the network compared with unicast routing when the server has to send the same packets to each client. There is a variety of applications in which such a mechanism is required. For example, these are online broadcasting, internet TV, video conferences, and distribution of large files.

Presently, the architecture of computer networks is changing. The traditional TCP/IP architecture is replaced by the software-defined network (SDN) architecture [1]. In the opinion of the leading network hardware manufactures [2–4], nowadays SDNs form one of the most promising directions of the network industry. In the context of the current paper, the most important aspect of SDNs is their centralized network control, which provides a global view of the topology and state of the network. In turn, this opens new opportunities for routing with regard to constructing optimal routing trees taking into account the network load.

The existing solutions for multicast routing in traditional networks use routing trees produced by protocols as such Routing Information Protocol (RIP) [5] or Open Shortest Path First (OSPF) [6] on the network layer (L3) and STP (Spanning Tree Protocol) [7] on the data-link layer (L2). The application of certain route optimization criteria on one layer (e.g., the network layer) can contradict the criteria used by the protocol on another layer, e.g., the data-link layer. In the TCP/IP architecture, routing on the network and data-link layers is separated from each other and not coordinated with each other. It may turn out that the minimum weight tree on the layer L3 cannot be fully implemented because the STP closed the required ports on the L2 layer.

The SDN technology solves this problem by enabling the SDN controller to configure the routing tree on layers L3 and L2 consistently. The SDN does not use devices such as routers. The controller “sees” the network topology directly on layer L2. This makes it possible to overcome the routing problem indicated above, which is inherent in traditional networks by using a specific route optimization criterion for each group.

Since the SDN controller is aware of all changes in the state of the network devices, communication links, or ports, new opportunities for designing fault-tolerance mechanisms open up. Such a mechanism makes it possible to use various algorithms for reconstructing multicast routing trees when the group state changes (new clients joining the group or leaving it) or when the network topology changes.

In this paper, we describe algorithms for multicast routing in SDNs that ensure the optimization and fault tolerance of multicast traffic. These algorithms make it possible to use different optimization criteria for different groups. To this end, we analyze the existing multicast routing algorithms used both in traditional networks and in SDNs and discuss their drawbacks. We consider the following optimization criteria of multicast routing in SDNs: minimization of the distance from the server to the clients, maximization of capacity, and minimization of the network link load. In this paper, we describe an algorithm ensuring the fault tolerance of multicast routing, i.e., an algorithm for updating the group tree while preserving its optimality under the changing group state and network topology. We also present an algorithm for minimizing the total number of rules installed on switches.

The proposed algorithms were implemented in an application for the RunOS controller [8]. The experimental study showed that the delays due to route reconstruction satisfy the requirements of telecommunications operators of large regional networks.

The paper is organized as follows. Section 1 contains the analysis of the existing algorithms of multicast routing in the traditional and software-defined networks. In Section 2, the problem is formally stated. In Section 3, we consider the construction of multicast routing trees. In Section 4, an implementation of the proposed algorithm for the RunOS SDN controller is described. Section 5 contains the results of testing this implementation.

1. MULTICAST ROUTING

1.1. Multicast Routing in Traditional Networks

One of the standards of multicast routing on the network layer is the Protocol Independent Multicast—Sparse Mode (PIM-SM) [9]; in this protocol, the routing tree is the shortest path tree from the server to the clients. The routing tree is a rooted tree in which the root is at the server, the leaves are at the clients, all other vertices correspond to routers, and each edge corresponds to a physical network link.

A disadvantage of PIM-SM is that it does not build routing trees that are optimal in terms of the number of edges [9]. The routing trees produced by this protocol can have a larger number of edges. As a result, the traffic will use a greater number of physical network links and take away a part of the network capacity.

The number of links in multicast routing is minimized by constructing the Steiner tree [10]. This problem is formulated as follows: given a weighted graph $G = G(V, E)$ and a set of vertices $M \subseteq V$, construct the connected minimum cost acyclic subgraph H of G containing all the vertices of M. The cost of the subgraph is defined as the sum of the weights of its edges. If the weights of all edges are identical, then the Steiner tree is the tree with the minimum number of edges.

It is known that the problem of the Steiner tree construction is NP-hard [11]. In other words, the state of the art of the computational complexity theory does not raise hopes of designing a polynomial-time algorithm for solving this problem. For this reason, an approximate solution is typically sought [12]. Approximation algorithms are often complicated for the implementation in traditional networks due to a large number of data exchanges between routers. For this reason, algorithms for the Steiner tree construction are rarely used in the existing multicast protocols for traditional networks [13].

Another disadvantage of PIM-SM is the loss of packets when the routing tree is updated due to failures of network communication links or communication devices (switches or routers). In traditional networks, updating the routing tree takes too much time [14].

In traditional networks, routing on the data-link layer is based on the STP protocol [7] and Internet Group Management Protocol (IGMP) snooping [15], which provides the means for tracking the IGMP-traffic on the data-link layer. STP is a protocol for building a spanning tree in the ethernet network to prevent traffic loops on the data-link layer.

IGMP snooping prevents the transmission of redundant multicast traffic on the data-link layer that appears because the switch by default sends the multicast traffic to all ports (the broadcast mode); this is due to the fact that the switch uses the MAC (Media Access Control) address of the client, which in this case is the group address. IGMP snooping makes it possible to track the ports connected to the clients, thus preventing the group traffic broadcasting and reducing the network load.

It is seen from the reasoning given above that the STP tree does not satisfy various route optimization criteria, such as the link workload and path length from the source to the clients. Moreover, updating the spanning tree using the STP in the case when the network topology changes is time consuming [16].

1.2. Multicast Routing in SDNs

The SDN technology allows us to build routing trees using more complicated metrics and to drastically reduce the time needed to build routing trees due to the centralization of the network control.

In [17], it is proposed to use an application for the SDN controller called NOX; this application processes the IGMP messages from clients and creates routes of the multicast traffic in the network. The routes from the server to the clients are found using Dijkstra's algorithm [18]. The distances between network nodes (the number of segments in routes) are used as the edge weights.

In [19], an algorithm for multicast routing in data centers called the Avalanche Routing Algorithm (AvRA) was proposed. This algorithm minimizes the number of edges in the routing tree. It was implemented as the application for the OpenDaylight SDN controller [20].

AvRA is a polynomial randomized algorithm that builds the routing tree by connecting each new client to the nearest vertex of the routing tree. AvRA does not produce the optimal solution for networks with an arbitrary topology. However, it was proved in [19] that, for some popular topologies, such as Tree and FatTree, this algorithm finds the optimal solution with a probability close to one. If AvRA cannot find the optimal tree, it uses the same algorithm used in PIM-SM [19].

In [21], by contrast with the routing tree, a tree called the Branch-aware Steiner Tree (BST) is proposed. The distinction of the BST from the Steiner tree is that not only the number of used edges but also the number of branch nodes is minimized when this tree is constructed. The authors of [21] propose a k-approximation algorithm for constructing the routing tree, where k is the number of clients. This algorithm is called the Branch Aware Edge Reduction Algorithm (BAERA); by a k-approximation algorithm we mean an algorithm that produces an approximate solution at a cost that does not differ by more than a factor k from the cost of the optimal solution.

It is proved in [21] that the BST construction is NP-hard. Moreover, while there are approximation algorithms for the Steiner tree that find a solution with the cost not exceeding the cost of the Steiner tree by more than a factor of 1.55; there are no polynomial approximation algorithms for a BST that can find a solution that differs from the optimal one by a factor $k^{1-\varepsilon}$ for any $\varepsilon > 0$ (k is the number of clients) [21]. In other words, in this problem it is difficult to obtain an approximate solution that is close to the optimal one.

None of the multicast routing algorithms for SDNs allow us to select the optimality criteria for multicast routing trees for each individual group. However, this option is needed because different groups can have different requirements related to the service quality. For example, constraints on delays in the traffic to group members can be imposed for the groups responsible for video conferencing. In IP-television, the delay is not as important; however, the minimization of the network load is the key factor. In addition, the existing algorithms cannot update the routing tree in the case when the network topology changes due to failures in network links or switches.

2. STATEMENT OF THE PROBLEM

We represent the network by the graph $G = G(V;E)$, where V is the set of vertices corresponding to switches and E is the set of edges represented by the pairs (v,u) if the switches v and u are connected by a physical link. Denote by n the number of vertices and by m the number of edges in G; $M = M(g)$ is the set of members of the group g, where $M \subseteq V$. Each switch connected to the clients of the group g is a member of this group. The group members may join the group or leave the group. There is a dedicated vertex s in M called the group server, where the server is the source of traffic of this group. Suppose that a function $c(v,u) : E \to \mathbb{N}_0$ determining the capacity of the edge between the vertices v and u is defined on all edges $(v,u) \in E$.

2.1. Multicast Routing Problem

In the notation defined above, the multicast routing problem in the network can be formulated as follows: for every group g, find a subgraph $T = T(g)$ of the graph G such that T is a tree and T contains all the vertices of M. The tree T is called the routing tree for the group g.

2.2. Optimization Problem

In this paper, we consider three different optimization criteria for the tree $T(g)$ (it is denoted simply by T below):

(1) Hop,

(2) Port Speed,

(3) Load.

The meaning of these criteria is explained below. Let us give the following definitions.

The *length of the path* $P : v = p_0, p_1, \ldots, p_k = u$ is defined as the number of edges in P. It is denoted by $l(P)$.

The *shortest path* between the vertices v and u in the graph G is the path $P : v = p_0, p_1, \ldots, p_k = u$ such that $l(P) \le l(P') \; \forall P' : v = p_0', p_1', \ldots, p_k' = u$.

The tree T is optimal with respect to the *Hop* criterion if, for every vertex $v \in M$, the path from v to s in T is the shortest path between v and s in G (i.e., the solution to the multicast routing problem is provided by the shortest path tree).

The *maxmin* path with respect to the criterion c between the vertices v and u in the graph G, where $c : E \to \mathbb{N}_0$, is the path $P : v = p_0, p_1, \ldots, p_k = u$ such that the function $m(P) = \min_{i \in [1,k]} c(p_{i-1}, p_i)$ is maximal.

The tree T is optimal with respect to the *Port Speed* criterion if, for every vertex $v \in M$, the path from v to s in the tree T is the *maxmin* path with respect to the criterion c between v and s in G, where $c = c(v,u)$ is the capacity of the edge (v,u) (i.e., the paths between the group members are chosen to maximize the minimum capacity of all the edges in the path).

The *cost of the tree* T is the quantity $w(T) = \sum_{(v,u) \in T} I(v,u)$, where $I(v,u)$ is the membership indicator of the edge (v,u) in the tree T (i.e., $I(v,u) = 1$ if the edge (v,u) belongs to T and $I(v,u) = 0$ otherwise).

The tree T is optimal with respect to the *Load* criterion if T is the minimum cost tree and contains all vertices $v \in M$.

The minimum cost $w(T)$ is achieved on the tree that has the minimum number of edges. In other words, when the routing is done in a tree that is optimal with respect to the *Load* criterion, the minimum number of network links is used.

The problem of building a tree that is optimal with respect to the *Load* criterion is the Steiner problem on graphs [18]. The vertices $t \in V \backslash M$ are called the Steiner vertices. It is proved in [18] that this problem is NP-hard, and an exponential number of trees must be examined for finding the exact optimal solution. Therefore, to solve the multicast routing problem with respect to the *Load* criterion, approximation algorithms should be used.

2.3. Fault Tolerance Problem

The fault tolerance can be ensured by updating the routing tree T when the network topology changes. The network state changes as follows:

(1) a client joins (connects to) a group or leaves a group;

(2) the communication link between two switches is broken or recovered;

(3) a switch is disconnected or reconnected.

The algorithm must update the tree T in such a way that the updated tree T' is optimal with respect to the criterion used to optimize the preceding tree T.

3. MULTICAST ROUTING ALGORITHM

Recall that, due to the centralized control, the convergence time in SDNs is practically zero and the difference between the routing on the L2 and L3 layers disappears. In traditional networks, the convergence time is nonzero because the routers need time to exchange data about their current states and form a consistent representation of the network topology and its state in each router. Since the SDN controller has the complete information about the topology and state of the network, the routing trees in SDN can be constructed using more complex metrics. Moreover, a specific optimality metric may be chosen for each group that meets the service quality requirements for the group.

The SDN controller can directly track all changes in the network topology. This property can be used to develop an algorithm for ensuring fault tolerance, i.e., an algorithm for updating the multicast routing trees when clients join a group or leave it or when physical links or switches fail.

The multicast routing algorithm proposed in this paper is based on modifications of the existing algorithms designed for building the optimal routing trees described in this section. For each optimality criterion mentioned above, an algorithm building the corresponding multicast routing tree is chosen and an algorithm for updating this tree in the case when the network topology changes is described.

3.1. The Hop Criterion

To build the routing tree that is optimal with respect to the *Hop* criterion, we should construct a tree of the shortest paths from s to each vertex of the set M. To this end, the spanning tree of the shortest paths in G is first constructed, and then the minimum subtree containing all vertices of M is found. The minimum subtree is constructed by pruning the branches of the spanning tree.

The spanning tree is constructed using the breadth first search (BFS) algorithm [22]. When executing the BFS, the algorithm stores the list of leaf vertices (the leaves of the spanning tree), which is then used to remove the subtrees not containing the vertices of M. It is proved in [22] that the complexity of the BFS is $O(n+m)$.The complexity of pruning the branches of the BFS tree is $O(n)$. Therefore, the total complexity of the algorithm is $O(n+m)$.

In traditional networks, the shortest paths are constructed using distributed (parallel) algorithms of the same time complexity but with a greater number $O(n^3)$ of message exchanges [6].

Proposition 1. A tree built using the above algorithm is optimal with respect to the *Hop* criterion.

Proof. It is proved in [22] that the BFS tree is a tree of the shortest paths, i.e., the path from the root s to any vertex v of the tree is the shortest (in terms of the number of vertices) path in G. Therefore, $\forall w \in M$—the paths to s are also the shortest ones. Hence, the resulting tree is *Hop*-optimal.

When a client v joins a group g, the branch from the BFS tree containing v is added to the routing tree. When a client leaves a group, the branch containing v is pruned. The branch is pruned up to a vertex u that has more than one descendant.

3.2. The Port Speed Criterion

To construct the routing tree that is optimal with respect to the *Port Speed* criterion, we should build the *maxmin* tree T rooted at the vertex s: $\forall v \in M \Rightarrow v \in T$. As in Subsection 3.1, a spanning tree satisfying this criterion is constructed, and then the minimum subtree containing all the vertices of M is found.

The algorithm for constructing the *maxmin tree* [22] is Dijkstra's modified algorithm for finding the shortest paths from a graph vertex to all the other vertices. The modification is that when a vertex is added to a path, the condition $Min(d[w], c(v,w)) > d[v]$ is checked instead of the condition $d[w] + c(v,w) < d[v]$ in the original Dijkstra algorithm ($d[w]$ is the capacity of the found path to the vertex w and $c(v,w)$ is the capacity of the edge vw). The capacity of the path to w passing through v is $d[v] = Min(d[w], c(v,w))$. The order of examining the vertices is also determined by the function $Max(d[w] \mid w \in F)$ rather than by the function $Min(d[w] \mid w \in F)$, where F is the set of unexamined vertices.

It is proved in [22] that the complexity of Dijkstra's algorithm is $O(n^2)$; therefore, the resulting complexity of the algorithm for constructing the tree that is optimal with respect to the *Port Speed* criterion is $O(n^2)$. As in the preceding algorithm, if a client leaves a group, the branch containing this client is pruned.

Table 1. KMB heuristic

```
Input   Connected graph $G$
Output  Tree $T$ approximating Steiner tree
1       Construct complete graph $K(M, E)$ with set of vertices $M$
2       Weight of edges $(i, j)$, where $i, j \in M$ equals weight of minimal path from $i$ to $j$ in $G$
3       Find minimum spanning tree $T'$ in $K$
4       Replace each edge $(i, j) \in T'$ by path in $G$ corresponding to this edge. Denote resulting subgraph of $G$ by $G'$
5       Find minimum spanning tree $T$ in $G'$
6       repeat
7         r := false
8         if $\exists w \in T : w \notin M, w \in leaf(T)$ then
9           $T := T \backslash w$
10          r := true
11      until r = false
```

Proposition 2. The tree built using Dijkstra's algorithm modified as described above is optimal with respect to the *Port Speed* criterion.

The proof of this proposition is similar to the proof of Proposition 1.

3.3. The Load Criterion

To construct the optimal routing tree with respect to the *Load* criterion, the Steiner tree connecting all vertices of M should be constructed. In contrast to the preceding subsections, where the spanning tree was first constructed and then paths to the clients were selected from it, in the case of the *Load* criterion the Steiner tree must be constructed individually for each set M. In other words, when clients are connected or disconnected, the optimal tree can be completely different from the preceding tree. The problem in which clients can connect and disconnect is called the *dynamic Steiner tree problem* [10].

It was proved in [10] that no approximation algorithm giving an approximate solution with a coefficient less than $0.5\log k$, where $k = |M|$, exists. For this reason, we propose to use heuristic algorithms for solving the dynamic Steiner tree problem in real networks.

We will use the greedy heuristic algorithm for constructing the Steiner tree described in [23]. The algorithm works as follows. When a new client appears, the closest point in the available Steiner tree is found and the client is connected to this point—this is the heuristic rule. The shortest path can be found using Dijkstra's algorithm the complexity of which is $O(n^2)$ [22]. Initially, when the group has no clients, the tree consists of the single vertex $s \in M$, which is the server of this group.

A disadvantage of this algorithm is that it does not take into account the disconnections of clients from the group. There are examples of topologies and sequences of connections and disconnections of clients in them for which this heuristic produces nonoptimal trees [23]. However, the reconstruction of the tree at each change of the group content (the set M) can cause delays and loss of traffic for the clients remaining in the group due to updating the multicast traffic paths. We need an algorithm for updating the tree and a criterion to determine whether or not the tree needs to be updated.

To estimate the degree of nonoptimality of the tree produced by the algorithm described above, we should compare its cost with the cost of the Steiner tree (the optimal solution); if the difference exceeds a certain threshold Δ, then the tree should be updated. Since finding the optimal tree is an NP-hard problem, we will compare the available tree with the tree produced using the KMB (Kou, Markovski, Berman) heuristic [24], which is described in Table 1. It was proved in [24] that this heuristic gives a solution whose cost exceeds the cost of the optimal solution by a factor of $2 - 2/k$, where $k = |M|$.

The outline of this heuristic is as follows. In the weighted graph G, the shortest paths between all pairs of vertices in M are found. Using these paths, the new graph K is constructed in which the edges are the paths found in G and the weights of these edges are equal to the weights of the corresponding paths in G. Next, the minimum spanning tree T' in K is found. Upon replacing the edges of T' by the paths in G, we obtain the subgraph G'. Next, we find the minimum spanning tree T and remove the branches not containing the vertices of M. The resulting tree T is an approximation of the Steiner tree in G.

Note that the subgraph in row 4 of Table 1 is not necessarily a tree because the paths in G corresponding to the edges $(i, j) \in T'$ can intersect. Therefore, the graph G' can contain cycles. In row 8, $leaf(T)$ denotes the set of leaves of the tree T. In rows 6–11, the branches of T not containing the vertices of M are removed.

The minimum cost paths between all pairs of vertices in M can be found using the Floyd–Warshall algorithm [22], the complexity of which is $O(n^3)$. The minimum spanning trees can be found using the Kruskal algorithm [22] with the complexity $O(m\log n)$. Thus, the resulting complexity of constructing the tree using the KMB heuristic is $O(n^3)$.

3.4. Updating the Routing Tree

To solve the fault tolerance problem, we need an algorithm that can update the routing tree when the network topology changes. The SDN technology makes it possible to track such changes because each SDN switch sends the controller information about the physical links connected to this switch.

We describe the procedure of updating the routing tree in the case of the following events:

(1) failure or recovery of the link between two switches;

(2) disconnection or reconnection of a switch.

Since the disconnection and reconnection of a switch can be modeled by the failure and recovery of the links connected to this switch, we will consider only the first case below.

When a link fails, the edge e corresponding to this link is removed and the clients connected through this edge should be reconnected using another path not passing through the removed edge. When an edge is removed, the graph's connectivity can be violated and the graph can be decomposed into a number of connected components of which one will contain the vertex s. In this case, the clients in the connected components not containing s will not receive the multicast traffic. Therefore, the clients with which the communication was lost must be reconnected. The reconnection should be performed in such a way that the other clients not affected by the link's failure do not experience a traffic delay or loss due to updating the routing tree.

Below we assume that the root of the routing tree is at the vertex s. Then, each vertex $w \in V(T)\backslash\{s\}$ has a parent vertex v.

When an edge $e = (v, w)$, where v is the parent of w, is removed, a new tree T' that is optimal with respect to the *Hop* or *Port Speed* criteria will be constructed in the graph $G' = G\backslash e$. The tree $T'\backslash\hat{T}(v)$ coincides with $T\backslash\hat{T}(v)$, where $\hat{T}(v)$ is the set of vertices of the subtree rooted at v in the rooted tree T in G, because the removal of an edge $e \notin E(T\backslash\hat{T}(v))$ does not affect the shortest *maxmin*-route from s to the vertex $x \in T\backslash\hat{T}(v)$.

Thus, the tree must be changed only in the vertices in $\hat{T}(v)$. Therefore, the changes in the traffic routes will not affect the clients in the set $T\backslash\hat{T}(v)$.

If a new edge $e = (v, w)$ appears, two cases should be considered.

1. The new edge is an edge that was earlier removed from the graph G. Then, the routes in the new tree will be changed only for the clients affected by the previous removal of this edge, while the routes to the clients not affected by this removal will not change and they will not experience delays and loss of packets due to updating the tree.

2. The new edge is indeed new or it existed earlier but the network topology changed during the time this edge was removed. In this case, it cannot be guaranteed that the new edge will not change the routes to all clients. Such edges can appear in the network if its physical configuration changes, e.g., new hardware is installed. In this case, the tree updating is a reasonable procedure because it can optimize the distribution of the multicast traffic.

For the *Load* criterion, the clients disconnected from the group will be reconnected one-by-one to the subtree of G containing the vertex s by the greedy Steiner tree construction algorithm described in Subsection. 3.3. The subtrees not containing s will be removed.

If new physical links appear in the network, then the routing tree will be updated only if the tree cost exceeds the cost of the tree constructed using the KMB heuristic more than by the threshold Δ.

4. IMPLEMENTATION

The proposed multicast routing algorithms were implemented in an application for the SDN controller RunOS [25]. The RunOS controller controls switches using the OpenFlow protocol [26]. This application stores information about each group represented in the network. For each group, the collection of pairs (switch, port) determining the location of servers that represent the group in the network is specified. Here by the server we mean a physical server connected to the port and the router that receives group traffic through L3 protocols, such as PIM and DVMRP.

To join or leave a group, network users send IGMP messages *Join Group* and *Leave Group*. For this reason, the application installs on the SDN switches the routing rules that redirect all IGMP packets to the controller. When the first client joins a group, the IGMP message *Join Group* is sent to the server of this group to make the server initiate sending the multicast traffic. When the last client leaves this group, the IGMP message *Leave Group* is sent to the server of this group to make it stop sending the traffic.

The messages *Join* and *Leave Group* are sent from the switch port connected to the server by sending the message Packet-Out to the OpenFlow switch.

4.1. Multicast Routing

As the controller starts, the application receives information about the network topology and the state of the network devices from the application *topology.* Next, for each group, depending on the optimality criterion for the routing tree of this group, a spanning tree is constructed in which the paths connecting the server with the clients will be sought. The spanning trees are constructed using the algorithms described in the preceding sections.

At the start of the controller, the rule that redirects all IGMP messages to the controller is installed on each SDN switch. Such messages are sent to the controller by the OpenFlow protocol using *Packet-In* messages. The application installed on the controller processes IGMP messages. When the controller receives the *Join Group* IGMP message, it seeks a path from the server to the client that sent the message for joining the group. The information about the port number of the switch that received the message is available for the application from the message *Packet-In*.

After the path from the server to the client has been found, the application installs the corresponding rules on the switches lying on the route implementing this path. If this is the first client that requested the group traffic, then the IGMP message *Join Group* is sent to the port to which the group server is connected. If the client is not the first one requesting the group traffic, then the path to this client is sought from the switch corresponding to the nearest vertex in the routing tree rather than from the switch connected to the server.

On each switch, the *Flow* rules are installed in which the *match* field has the *in_port* entry equal to the port number through which this switch is connected to the preceding switch in the route from the server to the client and the *ip_dst* entry contains the group address. Thus, this rule processes only the packets arriving from the server and related to the corresponding group.

The field *action* contains the number of the OpenFlow group rule corresponding to this group. This rule is the routing rule that duplicates the traffic to different ports of the switch. The ports to which the traffic is duplicated are described by the so-called Bucket records. The group rules are as follows.

The group is an All-group; i.e., packets are duplicated to each Bucket record, where the set of these records is determined by the current structure of the routing tree. If the vertex corresponding to the switch in the spanning tree of this group has multiple descendant vertices, then a Bucket record is created for each such vertex, and this record sends traffic to the port connecting this switch with the switch corresponding to its descendant vertex. Thus, the traffic is propagated through the routing tree. Below, the pair of rules (*Flow, Group*) will be called the multicast routing rules.

4.2. Minimization of Open Flow Rules in Multicast Routing

If the number of groups in the network is large, a large number of OpenFlow rules is needed, which can negatively affect the performance of OpenFlow switches. For this reason, we need an algorithm for minimizing the number of installed group rules while preserving the logic of routing the multicast traffic for all active groups.

In this paper, we propose an algorithm for minimizing the number of group rules installed on switches. This algorithm replaces the group rules of the same type by a single rule.

Each rule that the application installs on switches goes through an aggregation procedure. The rule aggregation procedure joins the group OpenFlow rules with the identical set of Bucket records into a single rule and updates the field *dst _ group* to the new group in the corresponding Flow rules.

The group routing rules are aggregated in two steps.

In the first step, the joined rules are created from the set of group routing rules created for each group. For each switch, the application stores the list of the original (not joined) rules and the list of actual rules installed on the switches.

As a request for creating a new multicast routing rule on a switch appears, it is checked if such a rule already exists on this switch (the check is performed based on the identifiers of the group rules):

—if such a group is found, then the Flow rule is rewritten to point at an existing group rule;

– if no such rule exists, then the procedure of joining the new rule with the existing rules is initiated; for this purpose, a rule with the identical set of Bucket records is sought among the rules installed on the switch;

—if no rule with the identical set of Bucket records is found, then the new rule is installed on the switch without any changes.

For each aggregate rule, there is a counter of multicast routing rules associated with it. A rule is removed from the switch only when all the rules associated with it are removed, i.e., when the counter becomes zero.

The second step of the aggregation procedure is the update of the switch states. The switch states are updated in response to the events that change the configuration of the multicast routing rules in the network, such as the events of leaving or joining a group, failures and recovery of communication links, or failures and connection of switches. These events affect all groups; therefore, the states of switches are updated only after the rule aggregation procedure is executed for all groups.

In the course of rule aggregation, the application compares the lists of the existing rules with the lists of the updated rules. If there are rules that are included only in one list, then they are either removed from the switch (if they do not appear in the updated list) or they are installed (if they are not installed on the switch but appear in the updated list). The corresponding Flow rules are also updated.

To minimize the number of OpenFlow messages (if there are groups to be removed and groups to be added), the pair of messages (delete and add) is replaced by a single OpenFlow message that changes the set of Bucket records in the removed rule for the set corresponding to the new rule.

Thus, the rule aggregation allows us to reduce the number of rules installed on the switches if there exist at least two group OpenFlow rules with the identical set of Bucket records.

4.3. Backing up the Application State

The RunOS controller supports the active and standby modes to ensure the fault tolerance of the SDN control loop that uses standby controllers. The standby controllers are connected to the OpenFlow switches as *slaves,* while the main controller is connected as the *master.* If the main controller fails, a standby controller is chosen and switched to the *master* role.

To ensure the coordinated work of controllers, the last state of the applications is regularly saved to a distributed database. The application also saves its state (the groups, connected clients, topology of the routing trees for each group, and the configuration of the application operation). Thus, when the network control is passed to a standby controller, the application can continue its work.

5. TEST RESULTS

5.1. Description of Test Scenarios

In this section, we describe the results of an experimental study of the developed application.

The experimental study was performed in the virtual SDN built using the network emulator *Mininet* version 2.2.2 [27]. The *OpenVSwitch* version 2.7.0 switches [28] were used as the software SDN switches, and the group traffic was generated by the *iperf* utility, version 2.0.5 [29].

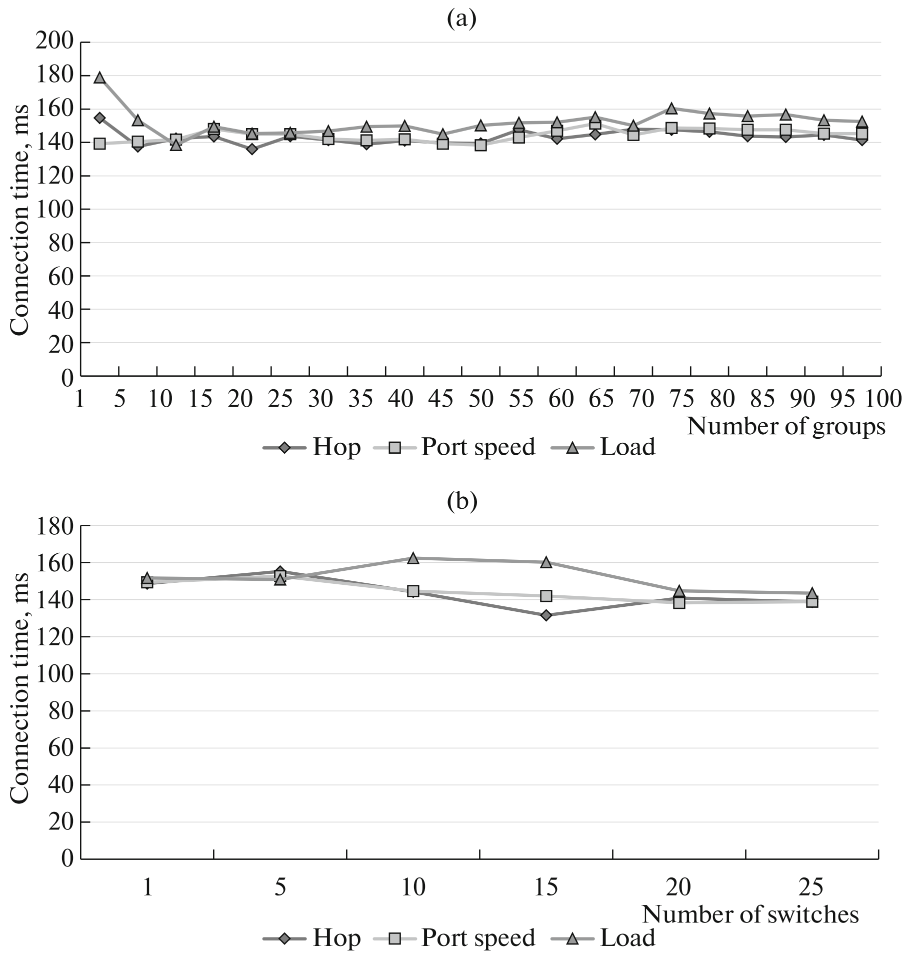

Fig. 1. Dependence of connection time on number of groups (a) and number of switches (b).

The application developed in this study was installed on the *RunOS* controller [25]. The controller was run on the virtual machine on a server with the following characteristics:

The CPU was Intel® Core® i7 9xx (Nehalem Class Core i7), two cores clocked at 2.4 GHz, RAM 2 GB DDR3 1333 MHz, and network interface 1 Gb ethernet.

The virtual network with software switches ran on a server with the following characteristics:

The CPU was Intel® Xeon® CPU E3-1240 V2, eight cores clocked at 3.40GHz, RAM 32 GB DDR3 1333 MHz, and network interface 1 Gb ethernet.

As a test topology, we generated a random network topology N for each test constructed using the Barabási–Albert algorithm [30]; this algorithm is used to generate random networks and it uses the preferential attachment principle. This principle assumes that the likelihood of attaching new nodes increases with the node's degree. The random networks generated using this model are connected and describe real-life computer networks used by the telecommunication operators.

For each multicast routing algorithm, two types of tests were performed:

(1) $\mathcal{T}_1$ tests the time needed to connect clients to a group;

(2) $\mathcal{T}_2$ tests the reconnection time after a communication link fails.

The client connection time is the time Δt passing from the instant t_1 when the client sent the IGMP request for joining a group to the instant t_2 when the traffic arrives at this group. The client reconnection time $\Delta\overline{t}$ passing from the time $\overline{t_1}$ of the failure of a communication link and when the traffic stopped being received by a client to the time $\overline{t_2}$ when this client begins receiving the traffic again.

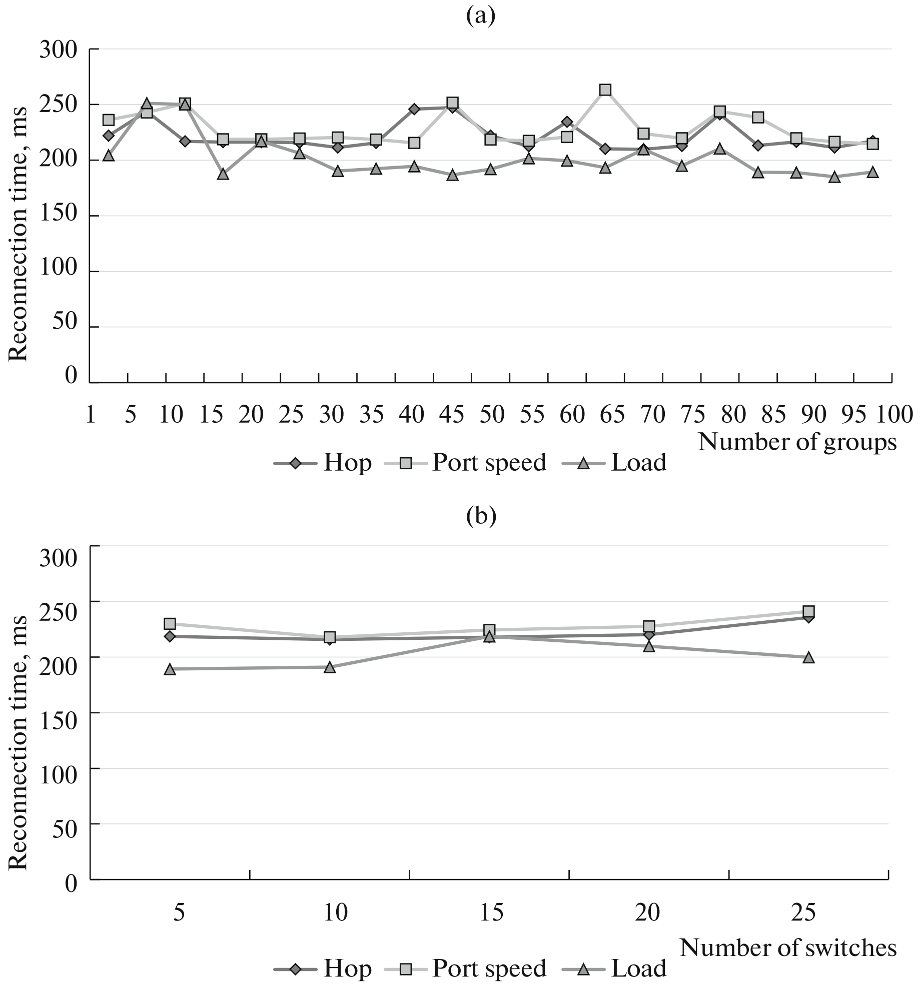

Fig. 2. Dependence of reconnection time on number of groups (a) and number of switches (b).

For each type of tests T_i, where $i \in \overline{1,2}$, a set of random network topologies N_{ij} with various number of network switches $j \in \overline{1,25}$ and a random set of clients C_{ij} was generated. For each topology N_{ij}, a set of tests T_{ijk}, where $k \in \overline{1,100}$, was executed; in these tests, clients joined the groups $\{g_1,\dots,g_k\}$. Depending on the test type i, either the mean connection time needed to connect the clients to these groups or the mean time needed to update the routing trees for each group together with the reconnection of clients was measured. For executing the tests of type $\mathcal{T}_2$, ten random communication links were sequentially cut off and the routing trees were updated.

We also note that the reconnection time was not measured for the topology of size one because it does not allow building reserve routes for multicast traffic.

5.2. Results

Figure 1a shows the mean time needed for a client to join a group as a function of the number of groups. The test results show that the mean time is almost independent of the criterion used to build the tree and of the number of groups.

Figure 1b shows the mean time needed for a client to join a group as a function of the number of switches in the network. The results for the same set of tests considered as a function of the number of groups and the number of switches illustrated in Fig.1 show that the performance of the algorithms is almost the same for all three criteria.

Figure 2a shows the mean time of client reconnection after the communication link failure as a function of the number of groups.

Figure 2a shows that the client reconnection time after the communication link failure is also almost independent of the criterion used to update the routing tree. Figure 2b shows the mean time of client reconnection after the communication link failure as a function of the number of switches in the network.

Testing showed that the tree building algorithms using three different criteria are equally efficient. In other words, both the routing tree construction time and the time needed to update this tree are independent of the tree type but depend only on the performance of the test platform. This can be explained by the fact that the greater part of the computation time is taken by installing the routing rules on the switches rather than by the tree construction algorithms.

The client connection time and the time needed to update the routing tree is between 130 and 260 ms. This is an acceptable time period in telecommunication networks [31]. Thus, the implemented algorithms are sufficiently efficient and can be used for practical purposes.

CONCLUSIONS

In this paper, we described multicast routing algorithms for SDNs that optimize multicast traffic routes and ensure their fault tolerance. These algorithms make it possible to use different criteria for the optimization of routing trees for different groups. The results of the analysis of the existing multicast routing algorithms used both in traditional networks and in software-defined networks are presented, and their main drawbacks are discussed. These algorithms were implemented in an application for the RunOS [25] controller both in the case of a single SDN controller and for the distributed network control plane. The application processes IGMP messages arriving from clients and installs routing rules that redirect the multicast traffic to the clients.

The algorithms discussed in this paper optimize multicast traffic routes and ensure their fault tolerance under changes in the group contents and network topology due to various reasons (communication link, switch, or its port failures). Three different routing tree optimality criteria were considered. For each criterion, an algorithm for finding the optimal routing tree was chosen. For the optimality criterion based on the number of communication links, a combination of heuristics for updating the routing tree was proposed. In addition, an algorithm for reducing the total number of multicast routing rules installed on switches was described.

The implementation of the multicast routing algorithms was tested on various randomly generated network topologies consisting of SDN switches. The test results showed that the algorithms using the three optimality criteria are equally efficient. In other words, both the routing tree construction time and the time needed to update this tree are independent of the tree type and, therefore, depend only on the performance of the test platform. Delays due to updating the routing tree turned out to be acceptable in the networks of telecommunication operators.

ACKNOWLEDGMENTS

This work was supported by the Skolkovo Foundation, project no. 79 of June 2, 2012.

REFERENCES

1. R. L. Smelyanskii, “Software-configurable networks,” Otkryt. Sist., No. 9 (2012).
2. N. McKeown, T. Anderson, H. Balakrishnan, et al., “OpenFlow: enabling innovation in campus networks,” ACM SIGCOMM Comput. Commun. Rev. **38**, 69–74 (2008).
3. A. Tootoonchian and Y. Ganjali, “HyperFlow: a distributed control plane for OpenFlow,” in *Proceedings of the Internet Network Management Conference on Research on Enterprise Networking* (San Jose, USA, 2010), pp. 3–9.
4. R. Wang, D. Butnariu, and J. Rexford, “OpenFlow-based server load balancing gone wild,” in *Proceedings of the Hot-ICE* (Boston, 2011), Vol. 11, pp. 12–18.
5. Routing Information Protocol (RIP) specification. www.ietf.org/rfc/rfc1058.txt.
6. Open Shortest Path First (OSPF) specification. www.ietf.org/rfc/rfc2328.txt.
7. Spanning Tree Protocol (STP) Application of the Inter-Chassis Communication Protocol (ICCP). https://tools.ietf.org/html/rfc7727.
8. A. Shalimov, D. Morkovnik, R. Smeliansky, et al., “The RunOS openFlow controller,” in *Proceedings of the 4th European Workshop on Software Defined Networks* (Bilbao, Spain, 2015), pp. 103–104.

9. D. Farinacci, C. Liu, and S. Deering, "Protocol independent multicast-sparse mode (PIM-SM)," Protocol Specification (1998).
10. M. Imase and B. Waxman, "Dynamic steiner tree problem," SIAM Discrete Math. **4**, 369–384 (1991).
11. P. Winter, "Steiner problem in networks: a survey," Networks **17**, 129–167 (1987).
12. F. K. Hwang and D. S. Richards, "Steiner tree problems," Networks **22**, 55–89 (1992).
13. C. A. S. Oliveira and P. M. Pardalos, "A survey of combinatorial optimization problems in multicast routing," Comput. Operat. Res. **32**, 1953–1981 (2005).
14. X. Wang, C. Yu, H. Schulzrinne, et al., "IP multicast fault recovery in PIM over OSPF," in *Proceedings of the International Conference on Network Protocols* (Vancouver, Canada, 2000), pp. 116–125.
15. Considerations for Internet Group Management Protocol (IGMP) and Multicast Listener Discovery (MLD) Snooping Switches. https://tools.ietf.org/html/rfc4541.
16. R. Pallos, J. Farkas, I. Moldovan, et al., "Performance of rapid spanning tree protocol in access and metro networks," in *Proceedings of the International Conference on Access Nets* (Ottawa, Canada, 2007), pp. 1–8.
17. L. Bondan, L. F. Muller, and M. Kist, "Multiflow: multicast clean-slate with anticipated route calculation on OpenFlow programmable networks," Appl. Comput. Res. **2** (2), 68–74 (2013).
18. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness* (W. H. Freeman, New York, 1979), Vol. 29, p. 270.
19. A. Iyer, P. Kumar, and V. Mann, "Avalanche: data center multicast using software defined networking," in *Proceedings of the 6th International Conference on Communication Systems and Networks COMSNETS* (Santa Barbara, USA, 2014), pp. 1–8.
20. J. Medved, R. Varga, A. Tkacik, et al., "OpenDaylight: towards a model-driven SDN controller architecture," in *Proceedings of the Conference on A World of Wireless, Mobile and Multimedia Networks WoWMoM* (Boston, USA, 2014), pp. 1–6.
21. L. Huang, L. Huang, H. Hung, C. Lin, et al., "Scalable steiner tree for multicast communications in software-defined networking," arXiv:1404.3453 (2014).
22. T. Cormen, C. Leiserson, R. Rivest, et al., *Introduction to Algorithms* (MIT Press, Cambridge, 2001), Vol. 6.
23. B. M. Waxman, "Routing of multipoint connections," IEEE J. Sel. Areas Commun. **6**, 1617–1622 (1988).
24. L. Kou, G. Markowsky, and L. Berman, "A fast algorithm for Steiner trees," Acta Inform. **15**, 141–145 (1981).
25. RunOS OpenFlow Controller. https://github.com/ARCCN/runos.
26. OpenFlow Switch Specification Version 1.3.2, Open Networking Foundation, CA, USA, 2013.
27. Mininet: Emulator for Rapid Prototyping of Software Defined Networks. https://github.com/mininet/mininet.
28. Open vSwitch. https://github.com/openvswitch/ovs.
29. Iperf: A TCP, UDP, and SCTP Network Bandwidth Measurement Tool. https://github.com/esnet/iperf.
30. R. Albert and A. L. Barabási, "Statistical mechanics of complex networks," Rev. Mod. Phys. **74**, 47 (2002).
31. Enterprise Campus 3.0 Architecture: Overview and Framework. http://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/Campus/campover.html.

Translated by A. Klimontovich