ARTICLE IN PRESS

Computer Networks xxx (2013) xxx–xxx

Contents lists available at SciVerse ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Incrementally upgradable data center architecture using hyperbolic tessellations ☆

Márton Csernai [a,*], András Gulyás [a,b], Attila Kőrösi [a,b], Balázs Sonkoly [a,c,d], Gergely Biczók [e]

[a] Dept. of Telecommunications and Media Informatics, Budapest University of Technology and Economics, 1117 Budapest, Hungary
[b] Hungarian Academy of Science (MTA), Information System Research Group Széchenyi István tér 9., 1051 Budapest, Hungary
[c] MTA-BME Future Internet Research Group Széchenyi István tér 9., 1051 Budapest, Hungary
[d] Inter-University Centre for Telecommunications and Informatics, Kassai út 26., 4028 Debrecen, Hungary
[e] Dept. of Telematics, Norwegian University of Science and Technology, N-7491 Trondheim, Norway

A R T I C L E I N F O

Article history:
Received 6 April 2012
Received in revised form 5 December 2012
Accepted 11 December 2012
Available online xxxx

Keywords:
Cloud computing
Data center
Incremental upgrade
Hyperbolic tessellation
Greedy routing
Distributed operation

A B S T R A C T

Current trends in cloud computing suggest that both large, public clouds and small, private clouds will proliferate in the near future. Operational requirements, such as high bandwidth, dependability and smooth manageability, are similar for both types of clouds and their underlying data center architecture. Such requirements can be satisfied with utilizing fully distributed, low-overhead mechanisms at the algorithm level, and an efficient layer 2 implementation at the practical level. On the other hand, owners of evolving private data centers are in dire need of an incrementally upgradeable architecture which supports a small roll-out and continuous expansion in small quanta. In order to satisfy both requirements, we propose *Poincaré*, a data center architecture inspired by hyperbolic tessellations, which utilizes low-overhead, greedy routing. On one hand, *Poincaré* scales to support large data centers with low diameter, high bisection bandwidth, inherent multipath and multicast capabilities, and efficient error recovery. On the other hand, *Poincaré* supports incremental plug & play upgradability with regard to both servers and switches. We evaluate *Poincaré* using analysis, extensive simulations and a prototype implementation.

1. Introduction

Cloud computing has been emerging to be the dominant operation model of present and future networked services. In order to provide the underlying pervasive networking functions, data centers have to scale up to previously unseen proportions. Tens of thousands of servers is already the norm for large providers, and this number is predicted to grow significantly over the next few years, as services, storage, as well as enterprises and users are increasingly relying on the cloud. Factor in virtualization and we are easily in the range of a million virtual end points. In parallel to migrating to huge, public clouds, another trend is gaining momentum: more and more organizations decide to consolidate their computing resources into small- or medium-scale private clouds [2]. There are multiple reasons behind the surge of private clouds. First, security and privacy issues using a public infrastructure can be prohibitive for certain organizations, such as governments [3]. Second, private cloud operation policies and management procedures can be tailor-made to the owner's liking [2]. Finally, of course, cost is always a deciding factor; surprisingly, operating a private cloud could be advantageous in the long(er) run [4]. As a consequence, the increasing proliferation of both small and large data centers are highly likely.

There are two orthogonal requirements for a data center design to suit both the needs of small-and-starting

☆ A shorter version [1] of this paper appeared in DCPerf 2012.
* Corresponding author. Tel.: +36 14632187.
E-mail addresses: csernai@tmit.bme.hu (M. Csernai), gulyas@tmit.bme.hu (A. Gulyás), korosi@tmit.bme.hu (A. Kőrösi), sonkoly@tmit.bme.hu (B. Sonkoly), gbiczok@item.ntnu.no (G. Biczók).

1389-1286/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.comnet.2012.12.004

and large-and-evolving data centers. First, such a design should be *structurally upgradable*: a small company should be able to start a private data center quickly and with a limited budget, and build it out incrementally in small quanta [5]. Note that even if servers are co-located at a data center provider, small and medium enterprises (SMEs) still have to face increasing costs as the number of servers increases. On the other hand, upgrades are also frequently needed in large data centers, triggered by a growing user base (private, e.g., Facebook) or deployment of more demanding cloud applications and getting more corporate customers (public, e.g., Amazon). As a well-known real-world example, Facebook has upgraded its data center facilities frequently and step-by-step [6], resulting in a doubling of servers over the course of 7 months in 2010 [7]. With most designs providing mechanisms only for large-scale expansion [8] [9], little work has been already done regarding incremental upgradability in data centers. LEGUP [10] leverages hardware heterogeneity to reduce the cost of upgrades in fat-tree based data centers. Jellyfish [11] and REWIRE [12] propose a non-symmetric topology to facilitate upgradability of data centers and employ shortest-path based MPTCP [13].

Second, the respective data center design should be *operationally scalable* providing performance, dependability (meaning path diversity and good failure tolerance) and manageability (i.e. simple configuration of the layer-2 protocol) for tens or hundreds of thousands of network nodes. On the network algorithm level, this implies distributed, low-overhead mechanisms both for routing and failure handling. On the implementation level, due to both virtualization and network management reasons, data centers are often managed as a single logical layer 2 fabric. On the other hand, traditional LAN technologies do not scale well to the size of large data centers. Network layer techniques, such as large forwarding tables, loop-free routing, quick failure detection and propagation, etc. are needed to be realized utilizing only switch hardware. Recently proposed solutions, such as TRILL [14], SEATTLE [15] and Portland [16], address these issues to a certain extent, but they come with a cost of significant complexity at the switch/fabric control plane. Other architectures, like BCube [8] and VL2 [17], rely on servers routing to overcome this limitation. While certainly a worthwhile approach, with the advent of computation-intensive cloud services and the migration of vast online storages into the cloud, coupled with the cloud providers' economic incentives to run their servers close to their full capacity [18], we argue that servers may face resource constraints if being the main responsible also for routing.

In this paper we propose *Poincaré*, a data center architecture which is by design structurally upgradeable and operationally scalable. The topology of *Poincaré* is inspired by hyperbolic tessellations, providing incremental expandability and favorable performance characteristics. *Poincaré* uses a fully distributed, low-overhead, greedy routing algorithm efficiently utilizing the features of the topology. Such a lightweight routing mechanism can be fully implemented in layer 2, while keeping both the control and the forwarding plane simple at the same time, enabling a data center built out of commodity off the shelf (COTS) network equipment. The main benefits of *Poincaré* are threefold. First, *Poincaré*'s hyperbolic structure provides analytically provable low network diameter and high performance greedy routing, multiple short paths between arbitrary nodes and high bisection bandwidth. Second, *Poincaré*'s greedy routing mechanism harness the aforementioned structural characteristics while allowing for natural local failure handling within failure detection time and low-overhead multipath and multicast routing. Finally, *Poincaré* supports incremental plug & play upgradability with regard to both servers and switches, while enabling a small initial rollout and a flexible, budget-conscious way of data center expansion. We justify our design with analytical proofs and extensive simulations augmented by a prototype implementation and testbed experiments.

The rest of this paper is structured as follows. Section 2 introduces the *Poincaré* data center structure based on hyperbolic tessellations. In Section 3 we present our greedy geographic routing algorithm along with its multipath, multicast and error recovery extensions. The incremental structural and performance upgrade process is carefully described in Section 4. In Section 5 we thoroughly evaluate the performance of *Poincaré* via simulation. Section 6 introduces the *Poincaré* prototype implementation and provides testbed experiment results. Section 7 presents practical considerations on cabling, initial rollout and expansion costs. Finally, related work is described briefly in Section 8 and the paper is concluded in Section 9.

2. Structure: a trip to the hyperbolic space

A tree is a very cost effective interconnection structure when routing has to be solved on a population of network nodes. A k-ary tree can provide low diameter (low delay), low average degree (low cost), easy loop-detection and simple routing decision in the nodes [19]. Such compelling properties qualify trees to be utilized in an array of routing protocols (STP, OSPF, ISIS) and, more recently, in data centers. On the negative side, trees cannot ensure path diversity and high throughput: two key requirements for data center architectures. Recently, several augmentations of trees have been proposed to overcome these limitations by densification (e.g., Clos networks) providing multiple paths, large bisection bandwidth and no single point of failure [20] [17]. A common drawback of such approaches is that when expanding the DC these "embedded" interconnection structures has to be carefully maintained and sometimes completely replaced and rewired[1] to keep up with the number of servers. Such complete rewiring in fact mean the building of a new DC from scratch. LEGUP [10] addresses this issue by allowing for heterogeneous switches; yet, it is only suited for large upgrades and relies on higher layer mechanisms for performance.

Since we require *Poincaré* to be incrementally upgradeable, total rewiring is unacceptable. Thus we must need an architecture which can *preserve its inherent structure* regardless of the number of servers and still provide path diversity and high throughput. In the following we design

[1] When the arity of the tree has to be increased.

a topology inspired by tessellations embedded in the hyperbolic plane exhibiting structural similarity with trees [21]. Intuitively a tessellation can be interpreted as an augmentation of the tree structure in which the branches are connected and can exchange traffic without affecting the core.

2.1. The basic topology of Poincaré: hyperbolic tessellations

During the construction of the topology we use the Poincare disk model of the hyperbolic plane. The Poincare disk model represents the hyperbolic plane with a unit circle in the Euclidean plane. In this model the hyperbolic lines are arcs or lines that are perpendicular to the unit circle. The model has the property that while converting a shape form the hyperbolic plane to the Euclidean plane the angles of the shape are preserved. The distance in the Poincare model can be easily calculated from the Euclidean coordinates (in the rest of the paper we refer to these simply as coordinates) of the points with a formula shown later in Section 3.1. The reflection about a line in the hyperbolic plane is a reflection about the line or an inversion respect to the circle in the Euclidean plane.[2] These properties make the Poincare model well suitable to construct hyperbolic tessellations and represent them in the Euclidean plane. The (n,k) regular tessellation uses regular n-gons from which k meet in a given vertex to fill the hyperbolic plane with no overlaps and no gaps.[3]

If we consider the vertices of the polygons as nodes, and the sides as links, we have a regular topology embedded into the hyperbolic plane. We define the *Poincaré* topology to contain a regular tessellation of the hyperbolic plane as a subgraph. This subgraph will serve as a "base" topology which is always present in a Poincaré DC and over which we can install upgrades over time (Section 2.2). This base topology immediately ensures that greedy routing is 100% successful, which we prove later in Section 3.

The coordinates of the nodes for a given (n,k) tessellation can be easily computed by implementing simple geometric mappings in the hyperbolic plane. Start from an initial regular n-gon. The coordinates of the first node of the initial polygon are $(0, \cos(\pi/k + \pi/n)/\sqrt{\cos(\pi/k)^2 - \sin(\pi/n)^2})$. We rotate this point with $2\pi/n$ around the origin for $n-1$ times to get the first polygon. Reflect the vertices of the polygon about each possible side of the polygon for getting some other vertices and sides. Now reflect all points about all possible sides and do this procedure recursively to construct all nodes up to a defined number. The first steps of this process is shown on Fig. 1. Note that the hyperbolic tessellations are well studied geometrical objects; for a more detailed description consult [22]. A source code in Java can be found at [23].

This construction allows us to generate a tessellation of arbitrary size. The derived coordinates will point to the logical places of network elements (servers or switches) and the sides of the polygons points the location of the links. See the right side of Fig. 2 for an example. For a starter, imagine the topology of *Poincaré* like we place switches and servers (in this order) to the unoccupied node positions starting from the inner part of the tessellation.

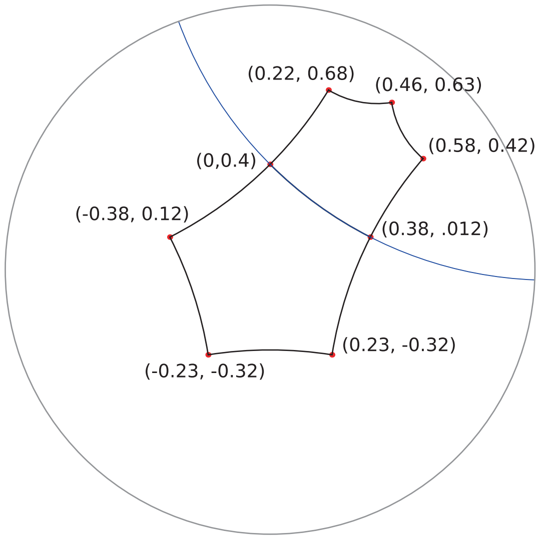

Fig. 1. ows the initial polygon of a (5,4) hyperbolic tessellation. The vertices of the initial polygon are reflected about one side of the polygon (blue line) to get one of the polygons belonging to the next level of the tessellation. The coordinates of the tessellation are shown next to the nodes. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Fig. 2 shows a fat-tree topology and a *(5,4)* tessellation having similar topological parameters. In th figure we plotted a traffic heat map of an all-to-all traffic scenario (every node sends one unit of traffic to all other nodes) for a 15 server fat-tree topology and a basic *Poincaré* topology with similar topological parameters (average degree, average path length). It can be seen that *Poincaré*'s routing can effectively utilize the "side" links to exchange traffic between "regions" of a tessellation without affecting the core. Basically this property will enable us to start building from a fixed sized core and implement incremental server and performance upgrades (as opposed to fat-tree's core which size varies when upgrading to larger core switches).

The basic topology immediately possesses compelling properties. In the sequel we prove that the diameter of such a network grows logarithmically with the network size while the degree of the nodes is bounded by k.

Theorem 1. *The diameter of the (n,k) tessellation grows logarithmically with the number of nodes.*

Proof. Denote the number of polygons of the mth level with p_m. Let v_m be the number of vertices that belong to level m and do not belong to level $m-1$ (see Fig. 3, where $v_1 = 5$, $v_2 = 25$, etc). For any $n > 3$, p_{m+1} and v_{m+1} can be calculated from p_m and v_m according to the following recursion.

[2] It depends on what the hyperbolic line's image is in the Euclidean plane.

[3] As opposed to Euclidean tiling there exist a (n,k) tiling on the hyperbolic plane if $\frac{1}{n} + \frac{1}{k} < \frac{1}{2}$ because on the hyperbolic plane the angles of a regular n-gon could be arbitrary small.

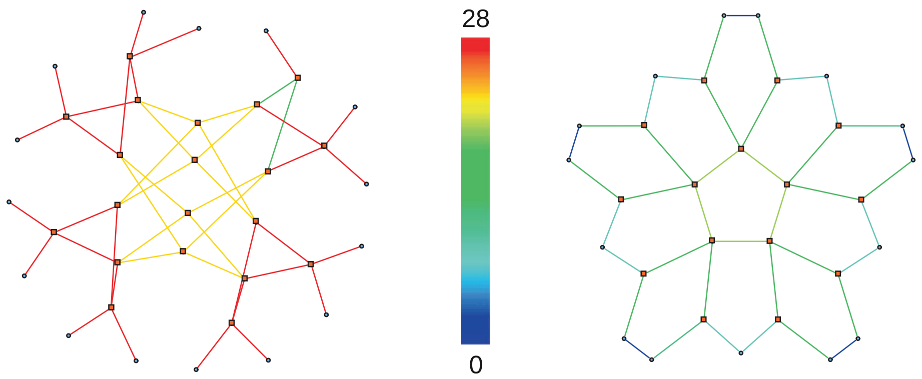

Fig. 2. Link traffic distribution in case of all-to-all traffic in (left) fat-tree and (right) tessellation structures. The squares are switches and the circles are severs.

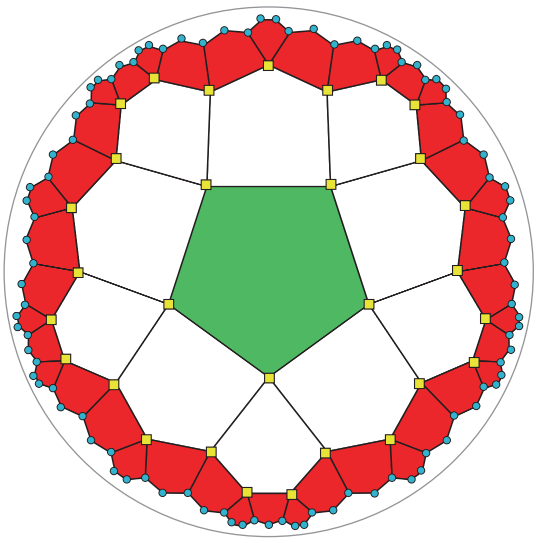

Fig. 3. The levels of the recursion is shown with different colors in case of a (5,4) tessellation. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

$$v_{m+1} = (n-3)v_m + (k-3)(n-2)v_m - (n-2)p_m$$
$$p_{m+1} = v_m + (k-3)v_m - p_m$$

Solving the recursion we get:

$$\begin{pmatrix} v_m \\ p_m \end{pmatrix} = \begin{pmatrix} (k-2)(n-2)-1 & -(n-2) \\ k-2 & -1 \end{pmatrix}^m \begin{pmatrix} v_0 \\ p_0 \end{pmatrix}$$

Let M be the matrix of the recursion, then M^m can be written as $M^m = Q\Lambda^m Q^{-1}$, where Q is the matrix of the eigenvectors of M and Λ is a diagonal matrix whose nonzero elements are the corresponding eigenvalues. In our case

$$Q = \begin{pmatrix} -\frac{\sqrt{(Z-2)^2-4}-Z}{2(k-2)} & -\frac{-Z-\sqrt{(Z-2)^2-4}}{2(k-2)} \\ 1 & 1 \end{pmatrix}$$

$$\Lambda = \begin{pmatrix} \frac{Z-2-\sqrt{(Z-2)^2-4}}{2} & 0 \\ 0 & \frac{Z-2+\sqrt{(Z-2)^2-4}}{2} \end{pmatrix}$$

where $Z = (n-2)(k-2)$. Hence by using $v_0 = n$ and $p_0 = 0$ we get

$$v_m = n\frac{(Z-2-Y)^m(Y-Z)-(Z-2+Y)^m(Y+Z)}{2^{1+m}Y}$$
$$\approx n(Z-1)^m = n((n-2)(k-2)-1)^m$$

where $Y = \sqrt{(Z-2)^2-4}$ and the last step is due to $Z \approx Y$.

For the special case $n = 3$

$$v_{m+1} = (k-4)v_m - v_{m-1}.$$

Let $\lambda_{1,2}$ be the solutions of equation $r^2 - (k-4)r + 1 = 0$, then v_m can be written in the following form:

$$v_m = a\lambda_1^m + b\lambda_2^m,$$

where a and b can be calculated from $v_1 = 3$ and $v_2 = 3(k-3)$.

Since the number of vertices grows exponentially with the number of levels, the diameter is at most $(2m+1)\frac{n}{2}$ which completes the proof. □

Although our model permits an infinite number of different tessellations to use, there can be major differences between them. First of all, k, i.e. the number of polygons to place next to each other in the tiling determines the minimal degree in the topology. Tessellations with higher k produce higher minimal degrees in the resulting topologies. On the other hand, the diameter of the network is linearly dependent on n. For example, a (3,16) tessellation produces small diameter but the minimal degree is 16 in the center of the topology. Note that we always build a finite topology so at the sides of the system (where the servers will reside) the degree is lower (see Fig. 3).

When a *Poincaré* system is designed, it can be chosen, how many switches to use constructing a DC topology for a given number of servers. We measured various fat-tree topologies and found that usual value of the switch to server ratio is around 0.16. We use this ratio as well. For example, we install 640 switches in case of a 4000-server system. Then the coordinate list is generated for 4640 nodes. The switches are placed to the innermost possible node positions (closest to the center of the disk we tile), and the servers are placed to the rest of the locations.

2.2. Constructing high-performance topologies

To improve performance we can add more links between our nodes. For *analysis and evaluation* purposes we now present a simple heuristic algorithm which produces more dense topologies by systematically adding more links on top of the basic topology. In Section 4 we also give more realistic algorithm which will adapt to the number and capabilities (port number, interface speed) of switches and servers we have, so as to make the best out of a given set of devices.

We can add links heuristically as follows: take an (n,k) tessellation as a basic topology and a radius r, and connect the nodes whose distance in the hyperbolic space is less than r. The Poincaré distance $d_{(n,k)}$ between neighbor nodes is different for different (n,k) tessellations. If r is smaller than $d_{(n.k)}$, then there will be no extra links in the graph other than links in basic topology. If we increase r more and more links are added. To meet the realistic constraint that servers can have a very limited number of ports, we use different connection radius r_{sw} and r_{se} in case of switches and servers respectively, thus connection radius can be parametrized for switches and servers separately (see Fig. 4 for an example). We connect two switches if their distance is less than r_{sw}. In case of server–server and server–switch links, r_{se} is used as a connection radius.

Using this methodology, link locality is preserved hereby moderating cabling costs. Also note that due to its heavy-tailed degree distribution *Poincaré* uses only a small number of large switches as it can be seen on Fig. 5.

Table 1 presents how the characteristics of the topology is enhanced by adding more and more links on top of diverse basic 4000-server *Poincaré* topologies. In all cases, we use 640 switches. The different basic topologies are densified with parameters r_{sw} and r_{se} to achieve high bisection bandwidth. It can be seen from the results, that all tessellations can lead to high performance. However *low n, high k* tessellations, such as $(3,16)$ achieves this with high max degree $(\hat{k}_{sw})$ and low diameter (D). For *high n, low k* tessellations, the system presents with lower degrees but higher diameter. There is a middle ground in case of $(5,6)$ tessellation, although we note that the bisection bandwidth is slightly lower than in case of the other two tessellation. This means that $(5,6)$ needs more densification to achieve the same performance, which increases its max degree to 72.

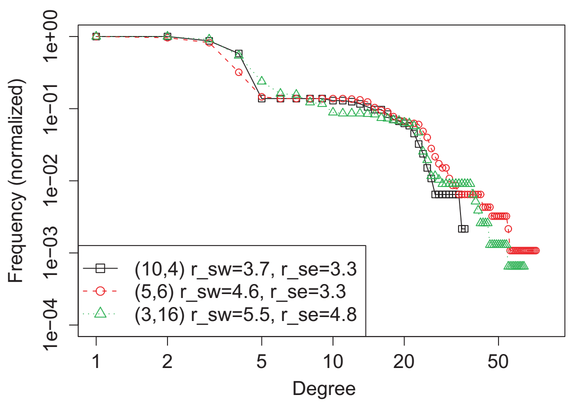

Fig. 5. Cumulative degree distribution of the different tessellation topologies.

Bisection bandwidth is calculated as follows. We define a random cut on the graph, i.e. we divide the nodes randomly into two equal sized parts. We need to count the total bandwidth between the two parts. For this we consider the weight of each link of the graph as 1. We add two additional nodes s and t to the graph. We connect one of them with all the nodes contained in the first part of the cut. Also, we connect the other node to every node in the other part. We set the weight of these new links to 2000 so these links cannot be bottlenecks. The maximal flow is calculated between s and t. Since there is only a slight variation in the results, we repeat this process 10 times, and consider the average value of the maximal flows as the bisection bandwidth of the graph.

As our results show there are many aspects of choosing the best tessellation for a specified DC architecture given its network size, port number restrictions, budget, etc. We note that our simple algorithm increases the average degree by large quanta, but we emphasize that link additions can be done with arbitrary granularity and using diverse algorithms to make the best out of a fixed budget (see Section 4 for a realistic algorithm). Various finer-grained optimization methods (in terms of throughput, cost, etc.) would further enhance *Poincaré* overall performance, however such techniques are not in the scope of this paper. In the next section we compare *Poincaré* to current DC topologies in terms of performance and costs.

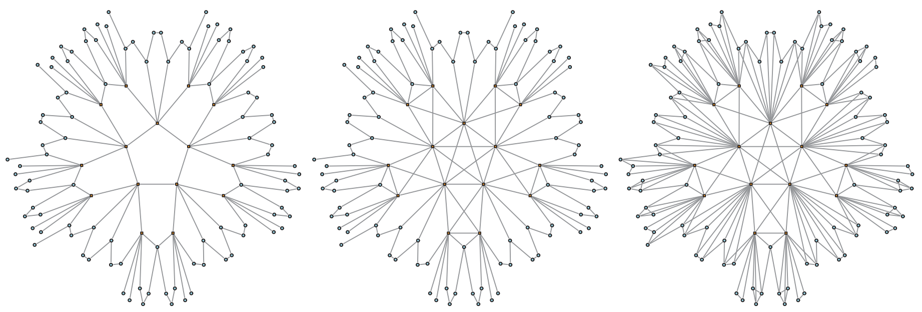

Fig. 4. A basic (left) and performance topologies by increasing r_{sw} (middle) and both r_{sw} and r_{se} (right).

Table 1
Topological parameters of different (n,k) tessellations in case of 4000-server topologies. The columns from left to right: tessellation type, basic tessellation distance, connection radius for switches and servers, number of links, total number of switch ports, total number of server ports, switch max. degree, server max. degree, avg. path length, diameter and bisection bandwidth.

(n,k)	$d_{(n,k)}$	r_{sw}	r_{se}	$\|E\|$	$\|p_{sw}\|$	$\|p_{se}\|$	$\hat{k}_{sw}$	$\hat{k}_{se}$	$\bar{l}$	D	B
(3,16)	3.28	3.3	3.3	9568	6700	12,436	16	9	6.23	7	4781
		5.5	3.3	11,212	9988	12,436	64	9	5.20	7	5610
		5.5	4.8	12,743	11,252	14,234	64	9	5.06	7	6434
(5,6)	2.19	2.2	2.2	5870	3785	7955	6	5	9.59	12	2847
		4.4	2.2	8500	9045	7955	48	5	6.09	9	4208
		4.4	3.3	11,740	11,595	11,885	48	5	5.32	9	5855
		4.6	3.3	12,745	13,605	11,885	72	5	5.04	9	6334
(10,4)	1.67	1.7	1.7	4700	2520	6880	4	4	12.79	17	2397
		3.7	1.7	7345	7810	6880	36	4	7.52	11	3655
		3.7	3.3	12,685	11,910	13,460	36	4	5.91	11	6354

Table 2
Topological comparison of different DC topologies, number of servers is 4000. The columns are: DC type, num. of switches, num. of links, num. of 1 Gbps ports, num. of 10 Gbps ports, switching costs in K$, maximal switch degree, maximal server degree, diameter, bisection bandwidth.

DC type	$\|Sw\|$	$\|E\|$	$\|p_{1G}\|$	$\|p_{10G}\|$	$K	$\hat{k}_{sw}$	$\hat{k}_{se}$	D	B
BCube ($n = 16, k = 2$)	762	12,000	24,000	0	2400	16	3	8	6036
BCube ($n = 8, k = 3$)	2028	16,000	32,000	0	3200	8	4	10	7866
Fat-Tree ($n = 28$)	980	14,976	29,952	0	2995	28	1	6	7529
Poincaré (4,10), $r_{sw} = 3.7$	640	12,685	25,370	0	2537	36	4	11	6359
Poincaré (4,10), $r_{sw} = 3.3$	640	11,870	21,740	2000	2674	24	4	11	10,672

2.3. Topological comparison of DC topologies

In the following the topological parameters of *Poincaré* is compared to fat-tree and BCube topologies. We demonstrate that our system is comparable to these currently used DC systems in terms of bisection bandwidth for the same amount of switching costs. The costs of the systems are estimated by counting all ports in the system. Current price trends indicate that the cost of switches can be estimated for $100 per 1 Gbit/s ports. Also, a DC specific 1 Gbit/s server port is also estimated as $100. Table 2 presents the properties of 4000 server *Poincaré* topologies with corresponding fat-tree and BCube structures. A 4000 server DC topology can be built with the BCube fabric two ways. We can either build it with two layers of 16-port switches or three layers of 8-port switches. We see a difference between the two systems in terms of costs and performance. A same size DC topology with the fat-tree fabric is achievable only with 28-port switches. This topology is between the two BCube variants in terms of costs and topological performance, although we note that fat-tree has lower and fixed diameter. On the other hand, BCube offers link disjoint multipath capabilities for the residing servers, while regular fat-trees do not. One can see that *Poincaré* is comparable with fat-tree and BCube topologies in terms of costs and bisection bandwidth while using less switches ($|Sw|$). Current technological trends allow us to depend on larger switches,[4] as the price of such switches grows only linearly[5] with the number of their ports [24]. For example, Arista's latest offerings even allow for 384 ports per switch [25].

In the last row of Table 2 we demonstrate that *Poincaré* can be easily augmented with high capacity links (10 Gbps). We put such high capacity links between the most inner core switches of the topology. We estimate such ports to be $250. We can see that for about the same switching costs, we can enhance the bisection bandwidth of the topology with the use of high capacity links while lowering the total number of links between switches. While this comparison looks unfair towards fat-tree and BCube, we note that those topologies cannot benefit from such performance enhancements out of the box. BCube routing is highly specialized for the homogeneous hypercube structure. Fat-tree needs other Layer 2 and Layer 3 extensions to take advantage of such enhancements [17]. In Section 5 the effect of high capacity core links is analyzed through traffic flow simulations.

3. Routing: greedy geographic routing

Poincaré routes greedily on the geographic coordinates of nodes in the tessellation. By using greedy routing the forwarding and control plane becomes very simple as opposed to current DC architectures. This is because all traditional control plane issues (path computation, guaranteeing loop-freeness, failure localization, reconvergence from failures, avoidance of route oscillations, etc.) usually implemented as a routing protocol can be completely omitted when greedy routing is used. In this section we present the basic routing mechanism and prove its performance on *Poincaré*'s structure in an analytical manner. Simple yet efficient multipath and multicast algorithms are also

[4] VL2 for example uses at least 40-port switches for a 4000-server DC topology.

[5] The price per port might even decrease with switch degrees up to a point.

proposed, which both leverage and enhance the greedy routing paradigm, and sustain the low overhead operation.

3.1. Basic routing mechanism

In greedy geographic routing, by default, the routing mechanism does not require routing states to be kept in the switching fabric. The routing decision is solely based on the metric distances between the link neighbor nodes $v(v_1, v_2)$ of u and the destination node $t(t_1, t_2)$. An intermediate node on the forwarding path always forwards a packet to its neighbor closest to the destination. In *Poincaré* we use the distances between the nodes on the hyperbolic *Poincaré* disk as the metric for greedy routing:

$$d(v,t) = \operatorname{arccosh}\left(1 + 2\frac{(v_1 - t_1)^2 + (v_2 - t_2)^2}{(1 - v_1^2 - v_2^2)(1 - t_1^2 - t_2^2)}\right)$$

If u does not have any neighbor v for which $d(v,t) < d(u,t)$ then greedy routing is in a local minimum and fails. Note that for example BCube's single path routing algorithm[6] also suffers from such phenomena. To overcome this, BCube's routing falls back to BSF search to be able to route when network failures are present. We will see that *Poincaré*'s structure ensures such events to happen only in case of massive network failures and with extremely small probability. The basic greedy routing algorithm is described in Algorithm 1.

Algorithm 1. Greedy routing algorithm for packets with given destination node ID.

```
GreedyRouting (current ID $u$, destination ID $d$):
$dist\_min = Inf$
**for all** $i$ link neighbors of $u$ **do**
  $dist$ = CalculatePoincareDistance ($i$, $d$)
  **if** $dist < dist\_min$ **then**
    $dist\_min = dist$
    $next\_hop = i$
  **end if**
**end for**
ForwardPacket ($next\_hop$)
```

In the following it is proven that greedy routing is always successful on a (n,k) tessellation, in case there are no link failures in the graph.

Theorem 2. *Greedy routing always finds paths between arbitrary node pairs in an (n,k) tessellation, assuming there are no link failures in the graph.*

Proof. We prove the theorem indirectly. Let P and Q be nodes of the tessellation, so that Q cannot be reached by a greedy path from node P. Without loss of generality assume that greedy routing fails at node A. First, we determine the neighbor of A that is closest to Q (if there are more neighbors equidistant to Q, we choose one from them). Let us assume that this neighbor is D, as shown on Fig. 6. Then Q must be in the angle range at A bounded

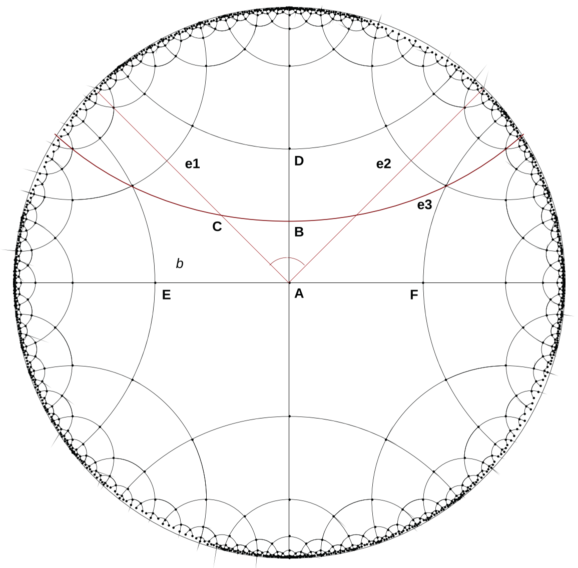

Fig. 6. Greedy routing always finds a path between all node pairs.

by e_1 and e_2, where e_1 and e_2 are the angle bisectors of $EAD\angle$and $DAF\angle$. Let us consider that part of the angle range, to where there are no greedy paths from A through D. This is the set of points that are closer to A than D. Let e_3 be the bisector of $\overline{AD}$. The set of points will be in the area bounded by e_1, e_2 and e_3.

We can assume that Q is closer to e_1 than e_2. Since the lines e_1 and e_3 are symmetry axes of the polygon b, their intersection C must be the midpoint of b. B is a midpoint of $\overline{AD}$, so Q must be contained by $ABC\triangle$. Trivially, since $\overline{AB}$ lies on the side of b and C is a midpoint of b, this contradicts our graph generation process, where every node is a vertex of the tessellation ($Q = A$).

During the routing steps the next hop is always strictly closer to the destination according to the greedy routing rule. This means that a node can be only once on the path and the path is in the circle with center Q and radius $\overline{PQ}$, and there are only a finite number of nodes in this region, the path must end at Q. □

The following theorem shows that greedy routing always finds the shortest paths in arbitrary hyperbolic tessellations.

Theorem 3. *Assuming no network failures, greedy routing always finds shortest paths between arbitrary node pairs in an (n,k) tessellation.*

Proof. We prove this statement in an indirect manner. Assume that there exist node pairs between which the length of the shortest path is smaller than the greedy path. Among these pick the pair (u,v) for which $h(u,v)$ is minimal, where $h(u,v)$ refers to the minimal hopcount between u and v in the tessellation. Fig. 7 shows the tessellation from the point of u. Since $h(u,v)$ is minimal the greedy path must deviate from the shortest path at u. The red and the blue lines show the shortest and the greedy paths on which the first node is s and g respectively.

[6] A special kind of greedy routing in high dimensional space.

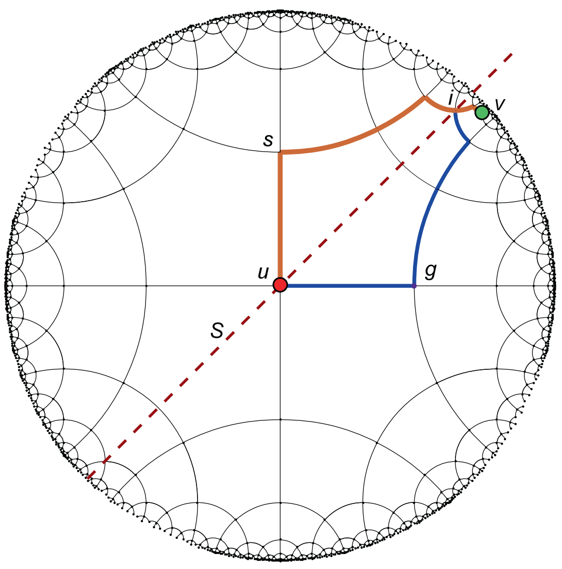

Fig. 7. The next-hop on the shortest path is also a greedy next-hop.

Due to the reflection symmetry of the tessellation there is an axis of symmetry S which maps s to g and also separates[7] s and v. Hence there exists at least one intersection i of the shortest path and S. By symmetry i can coincide with a node or can be a midpoint of an edge. Let us reflect the shortest path between u and i with respect to axis S. The mirror image and the part from i to v of the original shortest path is also a shortest path between u and v. Since g resides in the mirror image it is also a part of a shortest path. This is in contradiction with our assumption that $h(u, v)$ is minimal. □

Although the proof above holds for an infinite hyperbolic tessellation only, our simulations readily showed that greedy routing can effectively find the shortest paths on finite *Poincaré* topologies, thus eventuating quasi the same values for average distance and average greedy distance for the topologies shown in Table 1.

One can note that the simplicity of the control plane is traded off for the introduced computational requirement in the data plane. We argue that this is a justified choice in the design. Since arccosh () is monotone this operation can be left out of the calculation for boosting forwarding performance. Hence the required computation at each forwarding decision is reduced to about a dozen of simple arithmetic operations which consumes reasonably few CPU cycles. By default, nodes are required to calculate the next-hop distances for each packet to be forwarded. To further improve forwarding performance, routes can be cached by intermediate switches, and flow labels can be used for per-flow forwarding decisions. This trade-off is shown in Section 6 in a working greedy routing environment. We emphasize that *Poincaré* retains all advantages of greedy routing thus there is no link state propagation protocol prevalently used in DC architectures. This routing mechanism does not require carefully adjusted routing tables and implements routing with in essence zero messaging overhead.

[7] This is simply because g is closer to v than s also in Euclidean distance, while u is in the center.

3.2. Low overhead greedy routing extensions

DC specific routing requires many features such as multicast (to support MapReduce induced traffic load and distributed storage systems) and multipath routing (for multipath TCP, error tolerance, VM migration). We extend *Poincaré*'s default greedy routing to support the above mentioned features in a way that maintains the low overhead and distributed routing operation.

Multicast. For *small multicast groups*, *Poincaré* implements explicit multicasting like e.g. Xcast (explicit multi-unicast) [26]. Xcast is a very simple multicast mechanism which works as follows: let l be a list of destinations a packet should be delivered to and let this list be contained in the packet header. When such a multicast packet arrives to a node, l is extracted from the header and the next hops on the shortest paths $\text{next}(x)$, $\forall x \in l$ are calculated. If for some $u_1, u_2, \ldots, u_k \in l$, $\text{next}(u_1) = \text{next}(u_2) = \cdots = \text{next}(u_k) = n$, instead of submitting k packets to n, it is enough to send just one multicast packet having a modified list $l' = \{u_1, u_2, \ldots, u_k\}$ in its header. Note that $\text{next}(x)$ is uniquely determined by shortest path routing.

In case of *Poincaré* the situation is a bit different and gives more space for optimization. This is because in *Poincaré* we can use any of the available greedy next hops. Let c be the node which receives a multicast packet containing list l. In this case $\text{next}(x)$ turns into a set of next hops given by $\text{next}(x) = \{u | (d(u,x) < d(c,x)\}$. Putting it differently, as opposed to Xcast where $\text{next}(x)$ is a unique node, in *Poincaré* for each $x \in l$ we have a set of next hops. So node c at this point has the sets $\text{next}(x)$, $\forall x \in l$. Trivially, if node c calculates the minimal set of next hops $n_1, n_2, \ldots, n_k$ covering[8] all sets in $\text{next}(x)$, $\forall x \in l$ then the number of outgoing packets is clearly minimized.

Since solving the optimal set cover problem is NP hard, our implementation employs a greedy heuristic to approximate its solution. This means that for a packet with list l in its header, from the neighbors of c we pick node $u = \text{argmax}_v(|\{x \in l | v \in \text{next}(x)\}|)$. Now update the list $l = l \setminus \{x \in l | u \in \text{next}(x)\}$ and repeat the process until l becomes empty. Fig. 8 shows the different path selection results for a source and a two-member destination group in case of Xcast (left) and greedy multicast (right). In case of regular Xcast, the forwarding path is split after one hop, because the packets are sent on the shortest path from the first hop to the two destinations. It can be seen that in case of greedy multicast the packet is sent on common path to both destinations as long as there is a common greedy routable link (i.e. getting closer to the destination), hence saving precious link capacity. To speed up multicast forwarding in a working DC environment, the solution can make use of the recent implementation of the greedy heuristic function in NetFPGA architectures [27]. The descrip-

[8] A node n covers a set s if $n \in s$.

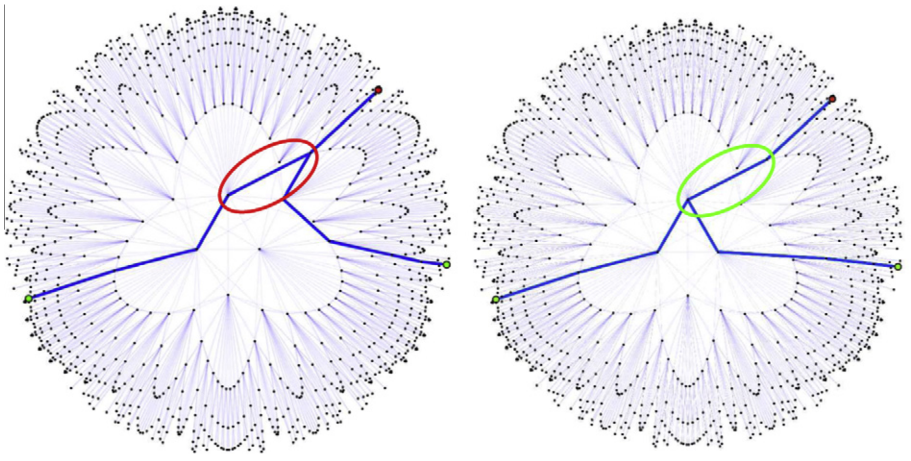

Fig. 8. Multicast modes: Xcast (left) and our greedy multicast algorithm (right). The plot shows links in blue which are used for forwarding multicast traffic. Xcast splits the common path earlier since it uses shortest paths. Greedy multicast take advantage of all paths that take the packet closer to the destination.

tion of our greedy multicast algorithm is presented in Algorithm 2.

Algorithm 2. Greedy multicast routing algorithm.

```
$l$ = list of destinations, $forwardingSet$={}
while $l$! = {} do
    Calculate $\text{next}(x), \forall x \in l$
    $u = \text{argmax}_v(|\{x \in l | v \in \text{next}(x)\}|)$
    $forwardingSet = forwardingSet \cup u$
    $l = l \backslash \{x \in l | u \in \text{next}(x)\}$
end while
ForwardPacket ($forwardingSet$)
```

In current data center architectures *large multicast groups* are usually supported by introducing state to the switching fabric. Note that the space-embedded structure of *Poincaré* permits the addressing of larger groups by defining a *destination* and an *offset* in the hyperbolic space. This type of addressing can designate servers whose distance from the destination is less than the offset value, hereby succinctly addressing larger network segments. We leave the implementation of such a multicast solution for future work.

Multipath. For multipath purposes *Poincaré* uses the following simple distributed algorithm: for a new incoming flow, choose the least-loaded outgoing link through which the packets can reach the destination on a greedy path. Such a multipath algorithm relies strongly on the number of edge disjoint paths that can be used by greedy routing. To measure how many such paths exist between pairs of nodes in a *Poincaré* topology G, we generate a directed subgraph G_d from G for every d destination. In G_d a link pointing from u to v exists only if u and v are connected in G and $d(u,d) > d(v,d)$. All link capacities are set to 1 and the maximal flow on G_d is calculated. By applying the Max-flow min-cut theorem [28] we get the exact number of link-disjoint greedy routable multiple paths between s and d. Table 3 shows the outcome of this process averaged for all source–destination pairs in 1000-node *Poincaré* topologies. After only moderate topology upgrades on average 2, maximally four greedy routable link disjoint paths are present in our simulated topologies. For easier positioning the results we note that the corresponding values for fat-tree, dual-homed fat-tree (DHFT) [13], and BCube are 1, 2 and $k+1$ respectively, where k stands for the number of BCube levels.

3.3. Failure tolerance

Greedy routing provides a fairly natural way of tolerating failures. Consider the scenario in Fig. 9 where PC2 sends traffic to PC1. From the coordinates we can compute the greedy path as PC2–Switch1–Switch2–PC1. If for example the link between Switches 1 and 2 goes down as indicated in the figure, Switch 1 notices the failure and for the next packet received from PC2 the greedy calculation gives Switch 4 as the next hop, hereby avoiding the failed link without any global failure propagation and route recomputation and requiring time only for the detection of the failure.

In case of link failures it can occur that greedy routing fails (stuck in a local minimum) however there would be available non-greedy paths. We will show in Section 5 that the probability of such events to happen in a *Poincaré*

Table 3
Link disjoint paths accessible by greedy routing.

Topology	Server ports		Greedy disjoint paths	
	Avg.	Max.	Avg.	Max.
$(4,10)$, $r_{sw} = 3.3$, $r_{se} = 3.3$	3.371	4	2.138	4
$(4,10)$, $r_{sw} = 3.7$, $r_{se} = 3.3$	3.663	4	2.569	4
Fat-tree	1	1	–	–
Dual-homed fat-tree	2	2	–	–
BCube	$k+1$	$k+1$	–	–

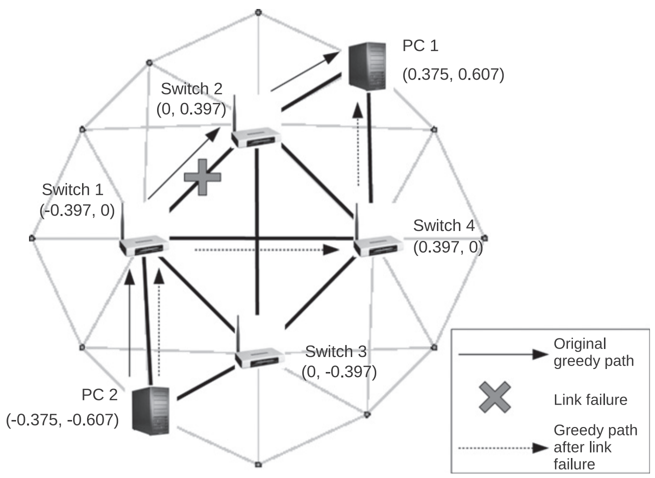

Fig. 9. (8,4) tessellation with coordinates and our testbed topology.

topology is very low compared to e.g. the disconnection probability of servers from the fat-tree topology in case of link failures. Moreover we can exploit the path diversity of *Poincaré* to reduce the probability of greedy fails by considering the following algorithm.

When a source node fails to find its destination with default greedy routing it can assign a random trajectory "bias" (α) to the next packet and retry the transmission. Intermediate nodes use this bias to assign weights d_i^α to their neighbors based on their distance d_i to the destination and pick a neighbor with greater probability that has larger weight. We can easily set up such a probability distribution and pick a neighbor with probability $p(j) = d_j^\alpha / \sum_i^{\#neighbors} d_i^\alpha$. In our simulations in Section 5 setting the number of retries to 10 gives a fair performance. The failure handling algorithm is described in Algorithm 3.

The effect of the different α parameter values on the greedy trajectories can be seen in Fig. 10. By setting α to a very low value the algorithm always favors the shortest greedy path, while if set to a higher positive value, the traversed routes will be distributed among the many greedy routable paths hereby avoiding the local minimum. To completely eliminate the possibility of getting stuck in local minima one can use recent improvements of greedy routing [29] [30], however such techniques notably increase routing complexity.

Algorithm 3. Greedy failure handling algorithm.

```
FailureHandlingGR (current node $u$, destination node
  $t$, $\alpha$):
$C$ = link neighbors $i$ of node $u$, where $d(i,t) < d(u,t)$
**for all** $j \in C$ **do**
  $w[j] = d(j,t)^\alpha$
**end for**
$next\_hop = Random\left(p(j) = \frac{w[j]}{\sum_{i \in C} w[i]}\right)$
ForwardPacket ($next\_hop$)
```

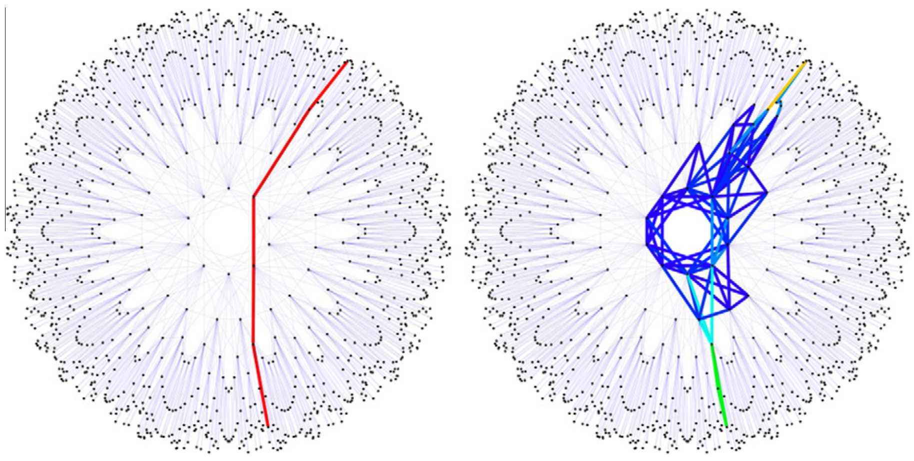

Fig. 10. Path distribution for a source–destination pair in case of single path where $\alpha = -100$ (left) and multipath where $\alpha = 1$ (right).

3.4. Addressing

Poincaré uses the hyperbolic coordinates as the addresses of servers and switches. We need to make sure that two arbitrary nodes can be surely distinguished by their coordinates. The coordinates are real numbers which can be represented in the floating point number representation. We computed according to the IEEE Standard for Floating-Point Arithmetic [31], how many nodes can be safely assigned distinct coordinates in a given tessellation structure. Table 4 shows the sizes of the different tessellation topologies that can be surely accommodated by the common number representations. For example, if we use the binary64 format to store one coordinate, that means we use 64 bits. From these 64 bits, 52 is used to represent the significant bits of the number. This lets us to use 5.42×10^{26} nodes with distinct coordinates in the (10,4) tessellation topology.

During the design of our architecture, we tried to keep the control plane as simple as possible. By using greedy routing most of the control plane functions can be omitted. What we still have to consider is the distribution and maintenance of node addresses. Addressing in *Poincaré*, i.e. mapping hostnames to hyperbolic coordinates, is proposed as the following. In case of small data centers, a central directory could hold the hostname-to-coordinate mappings for all servers, from which servers could look up and cache the coordinates of the destinations. In case of large data centers, where a central directory would be unfeasible, the directory service could be implemented in a DHT shared among the servers. Whenever a server is moved to a new coordinate, it can update its address in the DHT. Servers could also cache the currently used destination addresses. Similarly to our solution, both DCell [32] and BCube [8] use 32-bit specific host addresses based on the position of a server in their respective structure, which are mapped to IP addresses for applications. Portland [16] uses a central Fabric Manager that assigns pseudo MAC addresses for hosts based on their locations in the structure, and forwards packets based on these layer-2 addresses. VL2 [17] also uses a centralized and replicated directory service which resolves application specific addresses used by services to location specific addresses that are used for routing.

We emphasize that the control plane in *Poincaré* does not require topology information to be distributed and constantly updated, nodes only have to know the coordinates of their direct neighbors. Also note that all traditional control plane issues (path computation, guaranteeing loop-freeness, failure localization, re-convergence from failures, avoidance of route oscillations etc.) usually implemented as a routing protocol can be completely omitted when greedy routing is used. We strongly believe that getting rid of such complex tasks has made the control plane much simpler.

Table 4
Size of the topology supported by the different sized floating point number representation for the different tessellation structures.

Type	Significant	(10,4)	(5,6)	(3,16)
Binary32	23	506,000	805,000	85,700
Binary64	52	5.77×10^{12}	1.57×10^{13}	4.14×10^{11}
Binary128	112	4.99×10^{25}	5.42×10^{26}	9.64×10^{24}

4. Structural growth and performance upgrades

One important, albeit under-researched aspect of data center networks is incremental upgradability, i.e., adding servers and capacity to the data center on-demand [11] [10]. Upgrades can be triggered by a growing user base or deployment of more demanding cloud applications. Incremental roll-out is also a logical strategy supported by industry experts [5] and it is standard in practice. Microsoft's "Generation 4" data center concept builds heavily on incremental deployment [33], and Facebook's robust capacity expansion was also realized through incremental roll-out [6]. In the former sections of the paper, we used a heuristic model for constructing the topology of *Poincaré*, which was advantageous for the analysis of our system. One may argue that the heuristic model presented in Section 2 is not well adaptable to real world situations. In reality, there are fixed port switches and the servers have limited port capabilities. Here we show the process how we can build the structure taking these real world aspects of DC equipments into consideration. This section also discusses how *Poincaré* satisfies the requirement of incremental upgradability and describes various methods for capacity provisioning in a cost optimized manner. The upgrade process affects only the immediate vicinity of the newly connected servers or switches, i.e., without impacting the data center core and main operation. From a very high level point of view the upgrade process will be the following: when the topology of a *Poincaré* architecture is gradually increased, and a server runs out of free ports and we do not want to or cannot increase its port count, then this server needs to be shifted towards the perimeter, and it needs to be replaced by a router with higher port count.

As we have described before, there are various kinds of tessellations. The topologies based on the different tessellations have differing topological parameters, such as diameter and maximal degree. Along the presented analysis of the topological properties one can choose the suitable tessellation, which meets the requirements of the desired DC network. Then the building of the DC can be started from an arbitrary number of servers and switches, and can be gradually built out by adding more servers to the perimeter. The coordinates of the devices can be computed in advance and stored in a directory service to be used at new server or switch installment. An automatic coordinate assignment protocol could also be implemented, which would tell newly installed devices their new coordinates based on the coordinates of their neighboring nodes. Besides this coordinate assignment no extra configuration is needed to get *Poincaré* up and running.

4.1. Constructing the base topology

For building a proper Poincaré topology we have to take care of two things. First, to exploit the advantageous properties of a tessellation the main task when growing a *Poincaré* structure is to carefully maintain the base topology. This is formalized in Rule 1:

Rule 1. After adding a new node to the topology, the links of the base tessellation topology must be present.

Rule 1 basically means that if there is a tessellation link between nodes v_1 and v_2, then the devices that are on the coordinates of v_1 and v_2 must be connected.

Secondly, we need to ensure the correct allocation of new coordinates in the system:

Rule 2. The position where a new server can be installed is an unoccupied position having the minimal distance from the center (0,0). (Ties are broken randomly.)

Note that because of Rule 2 the topology will be balanced.

Now we can formalize the upgrade process as follows. We assume that the possible coordinates (x_1,y_1), (x_2,y_2), $\ldots$, (x_N,y_N) for the (n,k) tessellation is computed and stored in a list denoted by L. For the compliance with Rule 2, the coordinate list L is sorted according to the distance of the location (x_i,y_i) from the center of the disk (0,0) in increasing order.

Let $G(V,E)$ be a graph representing the current topology of our DC. Initially G is empty. Let v_{ns} be a new server node, which we want to install into G. We define $Pop(L)$ be function which returns the first element (x,y) of the ordered list L and removes (x,y) from L. (x,y) will be the coordinate of the new node v_{ns}. Also, let us define a function $E_{T,ns} = F(n,k,x,y)$, which returns the set of tessellation links $\{v_{ns},v_1\},\{v_{ns},v_2\},\ldots,\{v_{ns},v_j\}$ which must be included in G to meet the requirement of Rule 1. Let p_v denote the number of available ports in server v. The procedure of growing the *Poincaré* structure is described in Algorithm 4.

Algorithm 4. Structure growth.

```
AddServer ($G,v_{ns}$):
Insert new server $v_{ns}$ to coordinates $(x,y) = Pop(L)$ to
    meet Rule 2
Try installing links $E_{T,ns}$ to meet Rule 1
if $\exists w$, s.t. creating links $E_{T,ns}$ would increase the port
    requirement of $w$ above $p_w$ then
    substitute $w$ with a switch
    AddServer ($G,w$)
end if
connect links $E_{T,ns}$ to meet Rule 1
```

Let $|v_{sw}|$ be the number of switches in G after inserting $|v_{se}|$ servers. One can see that the switches will be on positions

$$\underbrace{(Pop(L), Pop(L), \ldots, Pop(L))}_{|v_{sw}|\text{times}},$$

meaning that they are placed to the $|v_{sw}|$ innermost coordinates of the disk. The severs are placed on the subsequent

$$\underbrace{(Pop(L), Pop(L), \ldots, Pop(L))}_{|v_{se}|\text{times}}$$

coordinates.

We demonstrate the build out method in the following simple example. Let's say, we build a DC structure based on the (5,4) tessellation. We generate the coordinate list with the method described in Section 2.1, which tells us the possible logical places of the devices. Fig. 11a shows the possible locations of devices for the 4-pentagon tessellation based structure with black points.

Now we start assigning physical devices to the logical places. When a device is installed to a place, the device gets the coordinates of the location. Suppose for simplicity that we have the most common DC servers with two ports. We add one server at a time employing Rule 2, paying attention to install the base topology links to meet the requirements of Rule 1. Since the servers' port count is 2, we can deploy at most five servers on the first level. As a result, we have a DC structure with five servers, as shown on Fig. 11b. Note that the topology on Fig. 11b is consistent with Rule 1.

The first non-trivial situation arises, if we want to install one more server. It is clearly seen, where we can deploy the next new server according to Rule 2, shown with red dots on Fig. 11b. Recall that Rule 2 breaks ties randomly. However, if we installed one more server anywhere among the possible places, the required port count of the neighboring server would increase to 3, which contradicts our assumption of the server port counts. Let's take out this neighboring server from the structure, replace it with a switch that has a higher port count. Let the port count of the switch be 4 that is equal to parameter k, which satisfies the trivial requirement that the port number of a switch must be greater than k (otherwise the tessellation cannot be constructed). Also, put the removed server back to the structure according to Rule 2. This process is shown on Fig. 11c. If there are still servers, whose port count would have to go higher than 2, then repeat this procedure until all servers are installed into the topology and they all need only two ports. After the insertion of four more servers, the resulting topology can be seen on Fig. 11d. Note if we permit the servers to have four ports then in case of a (10,4) tessellation, we can build the entire structure with servers. Now we show how to improve the throughput of the DC by putting more advanced switches into the inner part of the topology.

4.2. Performance upgrades

As resource demand is getting higher over time, the performance of *Poincaré* can be structurally upgraded by adding more switching equipment as well. Larger switches can replace smaller switches in a plug-and-play manner always resulting in better structural benchmarks. Clearly, adding links to the inner part of the topology can have larger impact. With a simple performance upgrade rule we can assure that a new bigger switch is deployed in a easy manner:

Rule 3. If a switch is replaced by a new switch with higher port count k_{nsw}, the new switch has to be connected to those k_{nsw} closest nodes, which have available free ports. Note that unplugging the old switch makes free ports for the tessellation links, which makes such upgrade conformant to Rule 1.

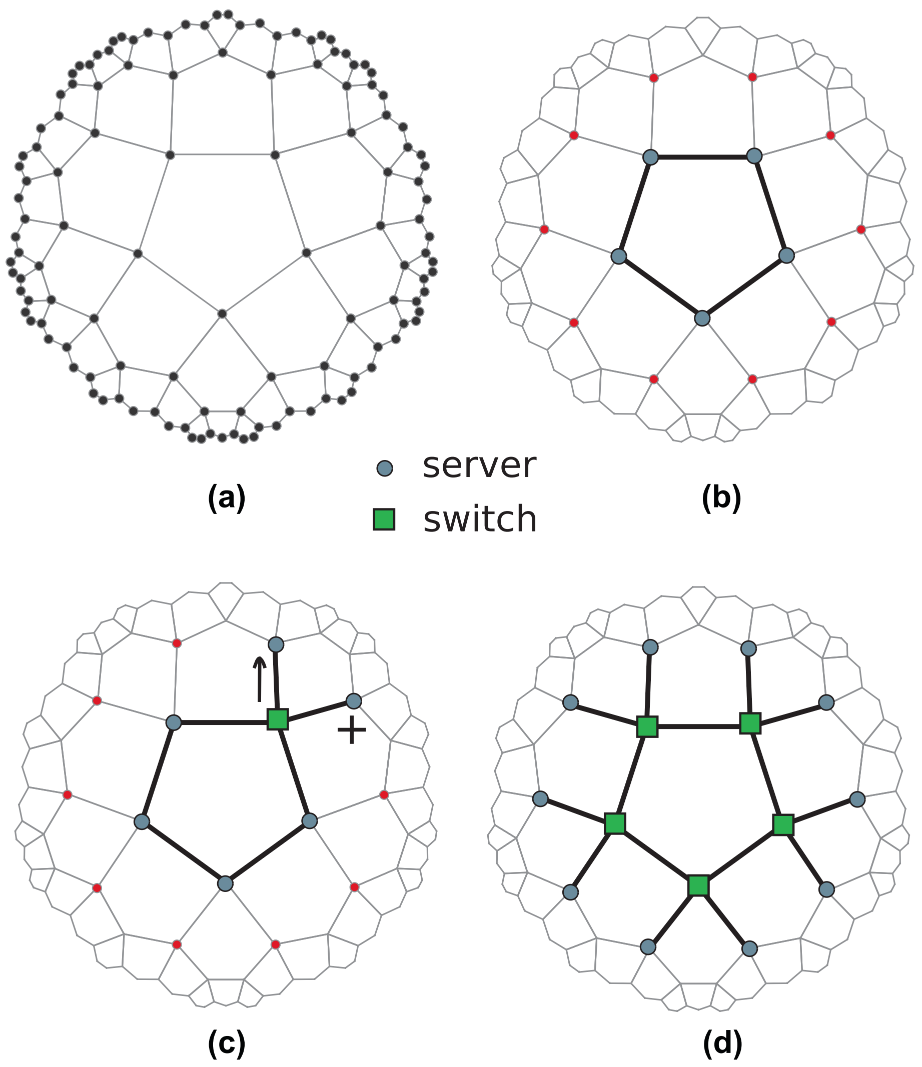

Fig. 11. Figure (a) shows the possible locations of devices in a (5,4) tessellation based structure with black points. Figure (b) shows an initial topology with five severs, and the possible next server locations with red dots. Figure (c) depicts when a server is moved to an outer location on the tessellation and gets replaced by a switch with higher port count, so the structure can accommodate the new server denoted with the "+" sign. Figure (d) shows the snapshot of the growth process with five switches and 10 servers. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

There is a possibility that the structure is in a state, when a new bigger switch cannot be connected to other switches following Rule 3, see Fig. 12a. In the worst case, it can even happen that we cannot connect the new switch to any other device because there are no available ports that could be used. This step is then regarded as a necessary intermediary step during the upgrade process. However, the next switch upgrade step will be surely successful, and new switch-to-switch links can be installed, as it can be seen on Fig. 12b. We note that the smaller switch, which was replaced can be reused at a outer location of the structure.

The links that are shown with orange color on Fig. 12 are not tessellation links. These links can be removed whenever a shorter link could be used as the result of employing Rule 3. When there are ties between distances between upgraded devices, these non-tessellation links can be arbitrarily added/removed following Rule 3 to optimize the overall performance.

We note that if we want to further enhance the connectivity of a given server, the fabric permits this. The only limiting factor is the available ports in devices in the proximity of the server. Also, usual DC servers come with 2/4 Ethernet ports built-in, and they can be expanded with port extension cards. Current technological trends show that a server can have as many ports as 10–12. For example, the two original server ports can be increased to 12 with two 5-port extension NIC.

The growing algorithm can be further optimized (in terms of throughput, cost, etc.) by considering many physical constraints (server room size, rack placement, energy availability, A/C) to enhance *Poincaré* overall performance.

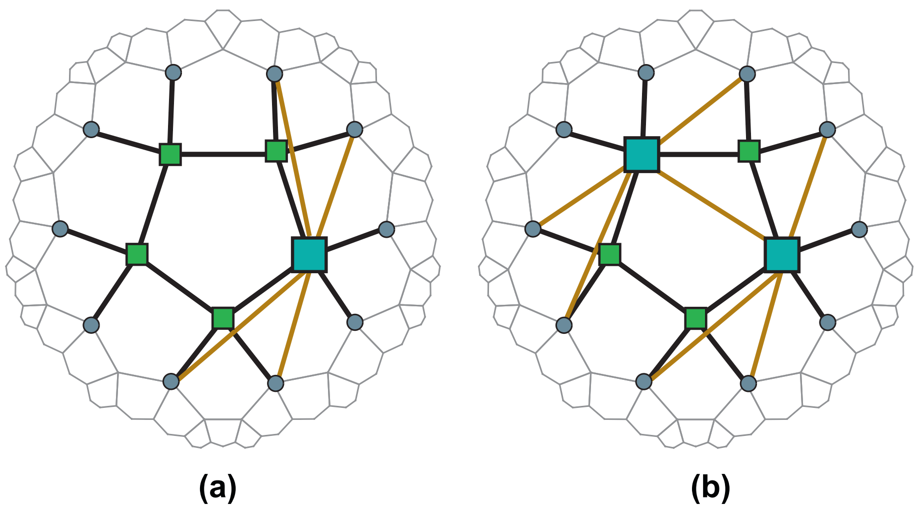

Fig. 12. Figure (a) shows how a 4-port switch is replaced by a 8-port switch to enhance the connectivity and thus the performance of the structure. Since there are no other free switch ports in the system, it is reasonable to install another 8-port switch as shown on Figure (b) to introduce additional ports to the system. The orange links are non-tessellation links, these can be easily managed following Rule 3.

These further extensions to the model are considered as future work.

5. Performance evaluation

We now turn to analyze *Poincaré*'s performance according to diverse metrics via simulation. First, we describe the simulation environment and our general traffic scenario. Next, the results of multipath and multicast performance simulations are provided and we analyze the inherent fault tolerance of the architecture.

5.1. Throughput

To evaluate *Poincaré*'s throughput, we have implemented a simple flow-level traffic simulator in C++, that simulates fat-tree and *Poincaré* routing on the generated topologies. All topologies contain 4000 servers with varying number of switches and the results are averaged over 10 simulation runs. We adapt a permutation traffic matrix from [13] where every host sends 10 MB data to another host, and no host receives more than one flow. This is implemented in the simulator first by generating the traffic matrix. The matrix contains the flows (4000 in this case), source and destination addresses. The routing function computes the path that the flow will use. This is done for all flows, and then it is summed, how many flows use one link. The topology file contains the capacity of each link, so it can be determined, how much capacity each flow will use, if we assume fair sharing of bandwidth resources among the concurrent flows on the same link. The simulator is paced to advance the data transfer in milliseconds. A flow is active, until the amount of its data is not fully transferred. We define the aggregate throughput of the systems as the total shuffled (transferred) data divided by the completion time of flows.

Table 5 shows the results of the traffic simulation. It can be seen that there is a trade-off between the number of switch ports and the incurred aggregate throughput. More dense the topology is, the higher aggregate throughput it can achieve. Furthermore the performance of the system is enhanced by high capacity links in the core. The table shows three cases for *Poincaré*; the first is a budget-friendly topology with few switch ports and few high capacity links. The second row shows a configuration equivalent to a similarly sized fat-tree DC in terms of throughput performance. Finally, we show an enhanced configuration with a higher switch port count and slightly more high capacity links. We also indicate the throughput results for different fat-tree configurations. Although the latter two configurations represent a notably different cost (see Section 7), they perform similarly to each other. We compare the switching cost of the structures similarly as we did in Section 2.2. *Poincaré* achieves the same performance as fat-tree for only a slightly higher switching costs.

Traffic distribution. In Fig. 13 we show the distribution of traffic flowing through links for a 1000-node tessellation. The figures show that if the topology is more dense, then traffic in the middle is distributed evenly across all core links.

Forwarding burden of servers. In *Poincaré* server nodes are also actively participating in routing to ensure greedy connectedness throughout the system. The definition of the forwarding burden of a server is the total number of flows that are routed through a server other than the flows that are initiated or terminated at a given server. Table 6 demonstrates how many external flows need to be routed through the end hosts in a all-to-all communication pattern. The numbers in the table are the average of this metric for all servers in the given network. In a fat-tree system server do not participate in the routing fabric. BCube heavily uses servers switching resources for overall routing. The forwarding burden metric was calculated for a

Table 5
Throughput comparison of 4000-server *Poincaré* and fat-tree DCs with different topology parameters. The columns from left to right: Topology, num. of switches, num. of 10 Gbps switch ports, num. of 1 Gbps switch ports, num. of 1 Gbps server ports, total switching costs in K$, total aggregate throughput (Gbps), avg. per server throughput (Gbps), runtime of the total data transfer in milliseconds.

Topology	$\|Sw\|$	$\|p_{sw,10G}\|$	$\|p_{sw,1G}\|$	$\|p_{se,1G}\|$	K$	$\sum T$	$\overline{T}_{se}$	t (ms)
$(4,10)$, $r_{sw} = 3.3$, $r_{se} = 3.3$	640	2400	8048	13,510	2755.8	239.91	0.143	1334.67
$(4,10)$, $r_{sw} = 4.3$, $r_{se} = 3.3$	640	4700	9898	13,510	3515.8	494.47	0.280	649.33
$(4,10)$, $r_{sw} = 4.6$, $r_{se} = 3.3$	640	7000	8878	13,510	3988.8	558.77	0.364	573.17
Fat-tree ($n = 28$)	980	0	25,952	4000	2995.2	471.91	0.265	680.5
Fat-tree ($n = 32$)	1280	0	36,768	4000	4076.8	473.97	0.265	680.17

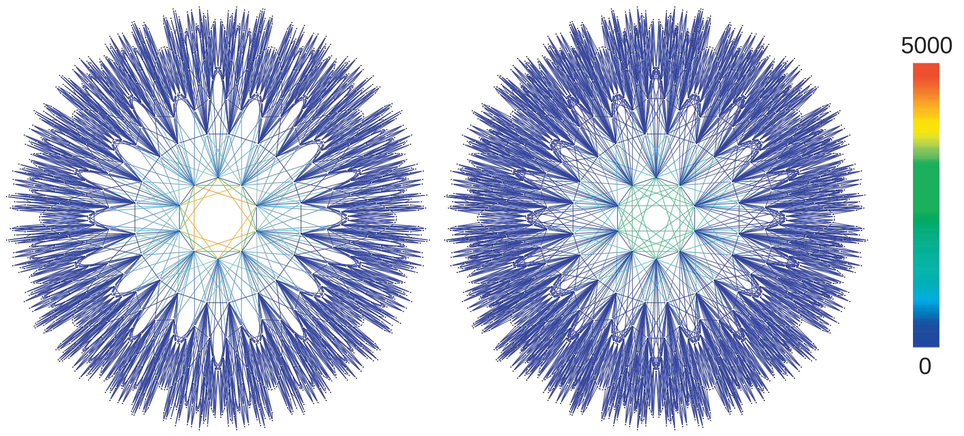

Fig. 13. Link traffic distribution in sparse ($r_{sw} = 3.3$, $r_{se} = 3.3$) and medium ($r_{sw} = 3.5$, $r_{se} = 3.5$) 4-decagon topologies.

Table 6
Average forwarding burden on servers for the all-to-all communication pattern on a 4096 server topology.

DC type	Avg. # of external flows on servers
Fat-tree	0
BCube	13,299
Poincaré (4-decagon, $r_{sw} = 4.3$, $r_{se} = 3.3$)	1386.42

4096 server BCube system. Contrarily to the server-based routing in Bcube, *Poincaré* imposes only a small forwarding burden which can be easily managed by hosts without additional resources.

5.2. Load-balancing and multicast performance

Load-balancing. The simple greedy multipath extension algorithm is not always able to generate edge disjoint paths on-demand. This can be regarded as a trade-off for the minimal overhead requirement of the routing algorithm (and budget friendliness). We demonstrate that the algorithm proves to be a practical and efficient load balancing approach. Table 7 shows the throughput simulation results of the multipath routing algorithm. We use the same *Poincaré* topology as in the second row of Table 5. We choose random pairs of servers in 4000-server topologies, and send 100 different flows of 1 MB data from the same source to the same destination. Results are averaged over 1000 different source–destination pairs. In contrast to fat-tree, where the bottleneck is the access link speed of the server, *Poincaré* leverages the multiple disjoint paths between the source and destination servers. We note that *Poincaré* routing could be also augmented with a multipath congestion control algorithm for further improvement. Smart flow management issues constitute important future work for us.

Multicast performance. The efficiency of the greedy multicast algorithm is compared to Xcast in Fig. 14. The plot shows the ratio of overall link loads when routing flows to the same set of destinations in case of Xcast and greedy multicast working modes. A factor of 1.1 means that 10% less link capacity is used by greedy multicast when sending to the same destinations.

5.3. Effects of structural failures on greedy search

Here we show how *Poincaré* copes with random link and node failures. Fig. 15 shows that the resilience of the architecture is remarkable in case of realistic link failure rates [17]. We simulate random link failure events in the topology (4000 servers, $r_{sw} = 4.3$, $r_{se} = 3.3$) and measured the overall success ratio of greedy routing for 50,000 source–destination pairs. The plot also shows the failure handling feature of greedy search, retrying to find a path for a maximum of 10 times, which improves routing success. To compare the results to fat-tree based topologies, we also plot the probability of host pairs remaining

Table 7
Comparison of multipath throughput capability of 4000 server fat-tree and *Poincaré* systems (Mbps).

Topology	Avg. multi. throughput	Var	Min.	Max.
Poincaré fat-tree equiv.	1527.41	406.43	1000	2898.55
Fat-tree ($n = 32$)	1000	0	1000	1000

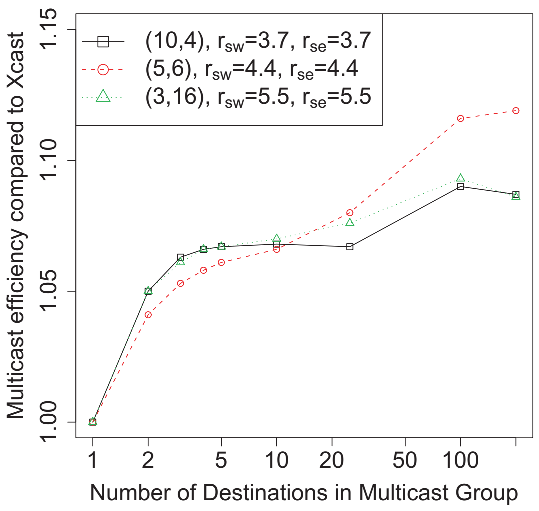

Fig. 14. Multicast algorithm efficiency compared to Xcast.

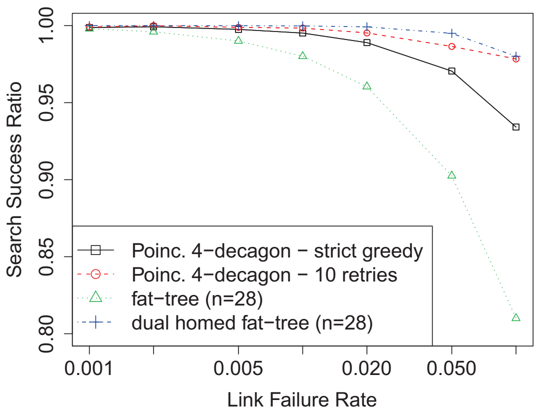

Fig. 15. Greedy search and fat-tree search success ratio versus random link failure rates.

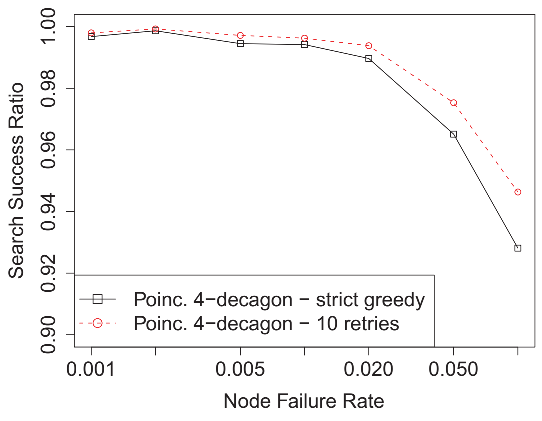

Fig. 16. Greedy search search success ratio versus random node failure rates.

connected in these structures at the presence of link failures which is an upper bound on the success rate.

In Fig. 16 the greedy search success ratio is plotted for various rates of random node failure in the topology. It can be seen on the plots that *Poincaré* copes with random link and node failures through graceful degradation.

Note that the results are generated with the strictest version of greedy search, which fails in a local minimum. Recently numerous improvements have been proposed to enhance greedy routing [29] [30], which could be readily applied in *Poincaré* to further improve failure resilience. Also, redundant storage and application-level solutions can mitigate the effects of failures [34].

6. Prototype implementation and measurements

In order to demonstrate and further evaluate *Poincaré*, we have implemented a prototype in OpenFlow and carried out performance measurements.

The OpenFlow specification [35] aims at enabling network innovation by strictly separating forwarding and control logic. Forwarding mechanisms including vendor-specific, proprietary solutions are dedicated to network devices which can be controlled through an open interface by separate controller entities. Network devices (OpenFlow switches) forward packets based on matching certain fields of packet headers with flow table entries while flow tables are maintained by controllers. In this framework, we have implemented greedy routing as a novel forwarding mechanism added to the OpenFlow reference software switch v1.0 [36] working as follows.[9] The flow table of a switch consists of one entry for all ports storing the coordinates of the corresponding neighbor and an additional one describing its own position. We use the destination MAC field in the packet header for storing coordinates and the forwarding mechanism takes this single field into account. As the OpenFlow reference switch requires, we store greedy rules in the linear flow table (wildcarded rules). Instead of standard matching, greedy forwarding calculates the distance between destination node (coordinates from incoming packet) and all neighbors (coordinates from flow entries), and finally, the entry with the minimum distance is chosen (match) and its action is executed. This means that the packet is forwarded to the closest neighbor or dropped when no closer node can be found. We emphasize that this forward-

[9] Sea also a downloadable OpenFlow implementation of greedy routing in [37].

ing mechanism results in a *fixed size flow table* bounded by the port number plus one. We note that the applied operations realizing greedy forwarding are computationally feasible in real switches. In addition, we use NOX v0.9.0 [38] as OpenFlow controller which plays a role only in the bootstrap phase; a special greedy application has been implemented in order to add flow entries to switches based on topology information.

The basic greedy forwarding mechanism can be enhanced by caching active flows in a hash table. This improvement yields faster forwarding (after the first packet, the standard exact matching is applied) at the cost of increased flow tables and memory usage. Both greedy forwarding methods, basic and enhanced, have been ported into OpenWRT [39] firmware (trunk version, bleeding edge, r26936) operating on TP-Link TL-WR841ND commodity switches.[10] In order to handle link failures, a lightweight greedy daemon has been implemented detecting link up/down events.[11] In case of link failure, the corresponding greedy flow entry is deleted by the `dpctl` tool through the local control interface of the switch, while link up event causes the restoration of the entry.

Our testbed environment consisted of the aforementioned TP-Link switches with 4 + 1 ports, PCs (Intel Core i3-530 CPUs at 2.93 GHz, 2 GB of RAM, running Debian GNU/Linux Squeeze with kernel 2.6.32-5) and a NOX v0.9.0 OpenFlow controller operating in out-of-band control mode on a separate management network.

Fig. 17 shows the results of simple performance measurements between two hosts connected to a single TP-Link device running different versions of OpenFlow switch. Throughput has been measured by `iperf` for both UDP and TCP (Cubic) traffic. We analyzed how much the calculation involved in greedy forwarding degrades performance compared to standard flow based techniques. For positioning the results of Fig. 17 we note that we use OpenFlow reference switch (software switch) in OpenWRT. Actually, we have software switches running on TP-Link devices. This software switch is running in user space. Furthermore, this device has a low-performance, quite "slow" CPU. Due to these facts, it is important to analyze the performance of the standard reference switch in this environment. We use the switch application of the NOX controller in cooperation with this standard switch and the throughput performance of the corresponding scenario is shown in Fig. 17 with label 'NOX OF switch'. This result can be considered as a reference, and the performance of our greedy forwarding implementation is compared to this value.

On one hand, performance of the enhanced version of greedy forwarding (with caching) is very similar to the standard switch application implemented in NOX. In both cases only the first packet suffers from increased delay. In case of the standard OpenFlow switch, the controller-

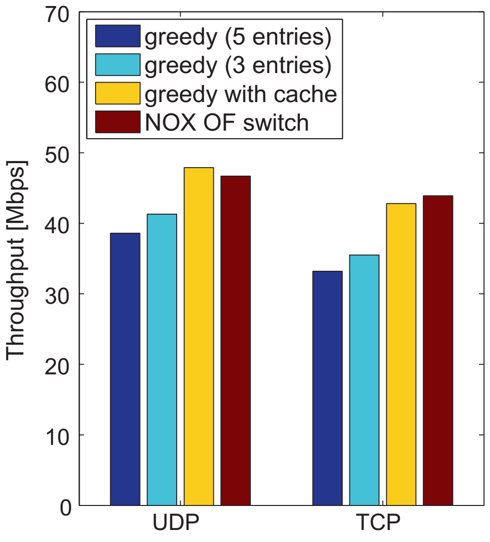

Fig. 17. Performance on a single switch.

to-switch communication (packet in, flow mod) causes the delay, while in case of greedy forwarding, the greedy next-hop calculation induces additional operations for the first packet. This shows that leveraging some of the control plane functionality into the data plane is supported by our measurements.

On the other hand, the basic greedy algorithm, which calculates distances for all packets, shows 10–20% performance degradation depending on the number of flow entries. The basic greedy implementation calculates the distances for all neighbors, thus, the performance is affected by the current number of neighboring nodes (which of course determines the number of flow entries). Here, we investigate the performance for two and four neighbors (three and five flow entries), respectively. On Fig. 17 the throughput of a switch with three and five entries are plotted. It should be emphasized that the switch is able to operate with fixed size flow tables at the cost of only a moderate performance degradation, even in case of our low-end COTS device with very limited computational power. Note that this implements a trade-off between the flow table memory requirement of switches and the performance of forwarding.

Furthermore, we have built (a relevant subset of) a *Poincaré* topology. The testbed topology consisting of four switches and two hosts (and a NOX controller) is shown in Fig. 9. In this network environment, the link failure recovery mechanism of *Poincaré* is demonstrated. PC2 generates UDP traffic to PC1 at a constant bit rate and the incoming packets with timestamps are logged at the receiver and plotted in Fig. 18. At 7.1 s the indicated link on the original greedy path is unplugged (link failure), thus PC1 cannot receive packets. The greedy daemons running on Switches 1 and 2 detect the link-down event and Switch 1 updates its flow table to the alternative path via Switch 4. This strictly local failure recovery mechanism achieves on-demand flow rerouting within 1 s *with zero communication overhead*: flow restoration only depends on the speed of link event detection.

[10] This TP-Link model requires some modification in the firmware, more exactly the MAC learning function has to be disabled in the kernel driver of the switch (ag71xx_ar7240.c), and an extension is also necessary for correct port status detection.

[11] Current version of OpenFlow uses LLDP for link state detection which is implemented in the controller. Therefore, this approach can be used only for small networks, and according to our experiments, the minimum timeframe is around 4–5 s even in very simple testbeds.

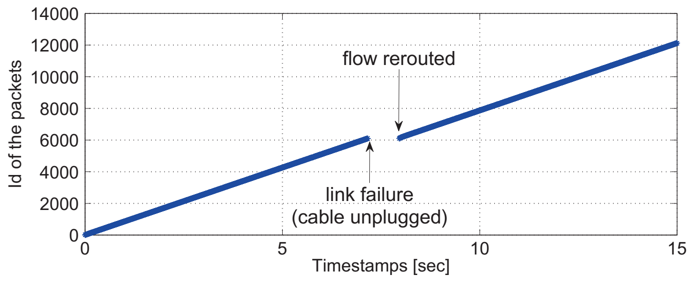

Fig. 18. Failure recovery.

7. Cost considerations: cabling, initial rollout and expansion

7.1. Cabling

A practical and economical requirement of data center designs is to exhibit reasonable cabling complexity. *Poincaré* achieves this by implementing the locality of the edges since it connects nodes that are close in the hyperbolic space. If we use an (n,k) tessellation as a minimum topology, the following arrangement hints at low cabling complexity. Form k rows of racks each containing one top level t_i switch and the servers and switches to whom t_i is the closest among the k top level switches. In this case most of the cables reside in one row since it can be shown that only $O(\log n)$ edges are "cross-row" edges, and hence easier to implement [40]. Similarly racking servers and switches based on space proximity should result in a moderate cabling burden.

7.2. Initial rollout and incremental expansion

As it was previously shown in Section 4 *Poincaré* is well-suited for starting small and upgrading incrementally. As for an initial build-out, a *Poincaré*-based data center can be constructed of any reasonable number of servers and switches of arbitrary port count. This flexibility enables start-ups and other budget-conscious companies to restrict their initial investment (low CAPEX). To put *Poincaré* in a cost perspective, we performed a cost comparison with the dual homed fat-tree design. Our cost model is based on [24] using regular ($100) and high capacity ($250) switch ports, and also server ports ($100) as our main cost elements. The cost of cabling is also taken into account. Based on the number of servers, we can estimate how many racks we need for a specific system. We consider 10% of all cables as inter-rack cables ($50 each) and the rest as intra-rack cables ($10 each). Also we consider the labor cost of cabling, and add $300 for every first cable between racks. We find that cabling costs are marginal in compliance with [24]. Suppose that a start-up wants to build a private DC based on a fat-tree structure, starting with 500 servers. We considered two scenarios: (a) minimum entry cost and (b) easily upgradable system for the foreseeable future with a larger initial CAPEX. We assume that upgrades are *incremental and continuous* over time. Fig. 19 shows the total cost of the company's DC while expanding from 500 to 8200 servers. It is shown that by starting with a small sized fat-tree topology ($n = 16$) the incurred entry cost is low. However, when reaching maximal tree size (1024 servers), the DC requires a structural reconfiguration of changing every switch in the system. This means that the next upgrade could cost the same as the whole system did until this point. On the other hand, a larger initial fat-tree could pose prohibitive entry costs for a small company, as high as double or triple of the smaller, but equal performance tree. Moreover, this larger tree would also reach its ceiling (note the jump after 8000 servers).

We compare the two systems such that we generated fat-tree topologies with $500, 1000, 2000, \ldots, 8000, 8200$ servers. The throughput was measured on these topologies as described in Section 5. Then *Poincaré* topologies with the same number of servers were generated and the parameters of the system were manually adjusted in every network size instance, so that the throughput would be the same as in case of the corresponding sized fat-tree topologies. The cost of the two systems were calculated and interpolated. The results can be seen on Fig. 19.

The two lines corresponding to *Poincaré* show (a) the minimum amount of money needed to build a working system and more importantly and (b) a system that is equivalent to a fat-tree system in terms of throughput. In the first case the initial DC has a very low entry cost, and it can be smoothly upgraded in small increments. In the second case, the *Poincaré* DC, that achieves the same throughput as fat-tree, can be built with lower entry cost and stays cheaper than or at worst comparable to its fat-tree counterpart.

8. Related work

Recently hyperbolic geometry is gaining more focus when it comes to explaining the intricate structure of real world complex networks [21] [41] and designing networks with extreme navigational properties [42] [43] [30]. Moreover, the hyperbolic embedding of the AS-level Internet topology has been computed to illustrate the effectiveness of the hyperbolic space in real routing situations [44]. However, current proposals for generating networks embedded in the hyperbolic space cannot be directly used in data centers. Poincare's structure is inherently symmetric as opposed to the above mentioned hyperbolic growing models. The symmetric topology lends itself to easier con-

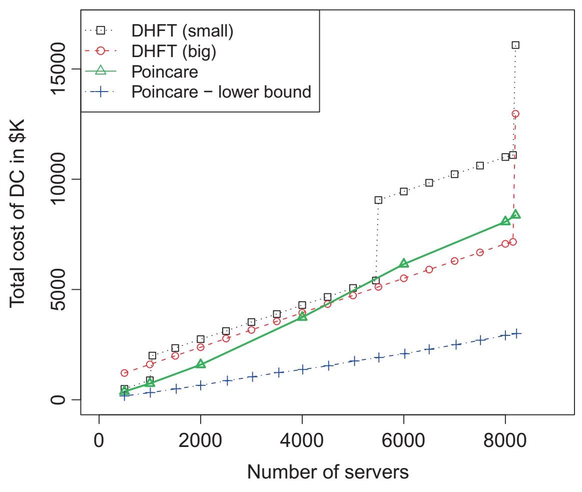

Fig. 19. Comparison of total cost for fat-tree and *Poincaré* DCs while adding servers incrementally.

figurations not just in terms of cabling simplicity, but also plug & play configuration; neighbors are able to calculate the coordinates of a newly joined node just from seeing which interface the newbie is connected to. Putting it differently, the operating crew of the DC can focus on the simple and symmetric topology the rest is handled by Poincare. To compare this with other hyperbolic embeddings, in those cases the newcomer must propagate its randomly chosen coordinates to the rest of the system and the crew of the DC must carefully check, which other nodes they should connect the newly attached node. This means that the crew has to be aware of the complete algorithm running behind the DC's topology.

Poincare's topology can operate and provide 100% greedy success rate by using switches with limited number of ports. In Theorem 2, we prove that using a tessellation 100% greedy success is analytically guaranteed. If we consider for example the 4-decagon topology, this comes with switches having a port limit of 4. The existing hyperbolic embeddings are to the contrary. For example, to generate a topology of [21] with a low maximal node degree, the γ parameter needs to be set relatively high. However, greedy routing performs worse, in terms of success ratio, on networks with higher γ values. More importantly, Poincare provides 100% greedy success rate by using only strict greedy search. This means that no additional mechanisms are used for search besides the simple greedy routing procedure. The topology proposed in [21] can guarantee 100% success rate by applying non-trivial tricks (gravity-pressure routing, face routing) to find a path, when greedy routing fails in a local minimum. Note that Poincare can also be augmented with these routing tricks. By doing so, we can get a much more robust DC system. However, from the point of analysis, strict greedy routing seems to be the only setting for which analytical proofs can be derived.

Several data center architectures have been proposed recently. Data centers based on fat-tree topologies [20,17,16] are built using commodity switches arranged in three layers, namely core, medium (aggregation), and server (edge) layers. The structure is also known as Clos topology. Portland [16] is a scalable, fault tolerant, but complex layer-2 data center fabric built on multi-rooted tree topologies which supports easy migration of virtual machines. VL2 [17] is an agile data center fabric based on end-host routing with properties like performance isolation between services, load balancing, and flat addressing. The topology of BCube [8] is generated using a recursive algorithm. BCube is intended to be used in container based, modular data centers, with a few thousand servers. MDCube proposes a method to interconnect these containers to create a mega-data center [9]. An initial design of an asymmetric, scale-free network inspired data center structure was proposed in [45].

The above designs are not geared towards incremental upgradability. The LEGUP proposal tackles the problem of expanding some existing data centers, adding heterogeneity to Clos networks [10]; however, its performance for small upgrades is unknown, as is the impact of the added heterogeneity on routing performance. Jellyfish suggests to use random networks as underlying topology [11]. Due to the random structure, incremental expandability of a Jellyfish DC is adequate. However, since Jellyfish uses a random graph based topology, it needs a sophisticated routing protocol to keep the system running in a fairly efficient way. Jellyfish's k-shortest-path routing has to monitor the topology continuously, converge swiftly to a changed state in case of failures, avoid oscillation and loops, etc. For this end, an up-to-date, global database has to be kept; this results in an unfavorable and possibly intolerable overhead in case of a fairly large data center. Poincare's strength in this context is that it both (a) has a symmetric, easy to manage topology and (b) uses only local decisions in routing and failure handling resulting in zero messaging overhead. To sum it up, Poincare's control plane is significantly simpler and scales better than Jellyfish's. A similar argument holds in the case of REWIRE [12], as well.

9. Conclusion

In this paper we introduced *Poincaré*, a data center architecture embedded into the hyperbolic plane. *Poincaré* uses greedy routing to forward packets between any pair of servers and due to its topological properties always routes along optimal paths. Moreover *Poincaré*'s routing relies only on local decisions and does not require complicated routing tables to be stored and sweaty routing protocols to be run. We showed that in the hyperbolic space greedy routing effectively exploit network redundancy which bards *Poincaré* with excellent failure tolerance and recovery. Our extensions of greedy routing provides multipath and multicast possibilities hereby boosting one-to-one and one-to-many communication performance. Satisfying our basic requirement *Poincaré*'s structure and performance is easy to upgrade and can be done with arbitrary granularity, servers can be added one-by-one and every single link improves performance without additional configuration. Our feasibility analysis indicates that *Poincaré* provides comparable throughput in comparison with fat-tree topologies and its flexibility enables to start from a very cheap but working DC and incrementally upgrade to any desired performance level, hereby significantly lowering entering costs.

Acknowledgements

The authors would like to thank the anonymous reviewers for their insightful comments which greatly helped to improve the paper. This work was partially performed in the High Speed Networks Laboratory at BME-TMIT. This work was supported by the OTKA-KTIA grant CNK77802. The publication was supported by the TÁMOP-4.2.2.C-11/1/KONV-2012-0001 project. This work was partially supported by the European Union and the European Social Fund through project FuturICT.hu (grant no.: TAMOP-4.2.2.C-11/1/KONV-2012-0013).

References

[1] M. Csernai, A. Gulyás, A. Kőrösi, B. Sonkoly, G. Biczók, Poincaré: a hyperbolic data center architecture, in: Proceedings of the 2nd International Workshop on Data Center Performance (DCPerf 2012) in Conjuction with ICDCS 2012, IEEE, 2012.
[2] Forrester Research, Market Overview: Private Cloud Solutions, Q2 2011 <http://goo.gl/RyTWd>.
[3] Defense Systems, Dod tackles information security in the cloud <http://goo.gl/Zcrnx>.
[4] Advanced Systems Group, Private vs. Public Cloud Financial Analysis <http://goo.gl/DzGCZ>.
[5] A. Licis, HP, Data Center Planning, Design and Optimization: A Global Perspective, June 2008 <http://goo.gl/Sfydq>.
[6] R. Miller, Facebook now has 30,000 servers, November 2009 <http://goo.gl/EGD2D>.
[7] R. Miller, Facebook server count: 60,000 or more, June 2010 <http://goo.gl/79J4>.
[8] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, S. Lu, BCube: a high performance, server-centric network architecture for modular data centers, ACM SIGCOMM Computer Communication Review 39 (4) (2009) 63–74.
[9] H. Wu, G. Lu, D. Li, C. Guo, Y. Zhang, MDCube: a high performance network structure for modular data center interconnection, in: CoNEXT '09, ACM, New York, NY, USA, 2009, pp. 25–36. doi:http://doi.acm.org/10.1145/1658939.1658943.
[10] A.R. Curtis, S. Keshav, A. Lopez-Ortiz, LEGUP: using heterogeneity to reduce the cost of data center network upgrades, in: Co-NEXT '10, ACM, New York, NY, USA, 2010, pp. 14:1–14:12. doi:http://doi.acm.org/10.1145/1921168.1921187.
[11] A. Singla, C. Hong, L. Popa, P. Godfrey, Jellyfish: Networking Data Centers, Randomly, in: Proceedings of USENIX NSDI'12, 2012.
[12] A. Curtis, T. Carpenter, M. Elsheikh, A. Lopez-Ortiz, S. Keshav, REWIRE: an optimization-based framework for data center network design, in: Proceedings of IEEE International Conference on Computer Communications (INFOCOM), 2012.
[13] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, M. Handley, Improving datacenter performance and robustness with multipath TCP, ACM SIGCOMM '11.
[14] R. Perlman, D. Eastlake, 3rd., D. Dutt, S. Gai, A. Ghanwani, Routing Bridges (RBridges): Base Protocol Specification, RFC 6325, July 2011.
[15] C. Kim, M. Caesar, J. Rexford, Floodless in seattle: a scalable ethernet architecture for large enterprises, in: ACM SIGCOMM 2008, ACM, New York, NY, USA, 2008, pp. 3–14.
[16] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, A. Vahdat, PortLand: a scalable fault-tolerant layer 2 data center network fabric, in: SIGCOMM '09, ACM, New York, NY, USA, 2009, pp. 39–50. doi:http://doi.acm.org/10.1145/1592568.1592575.
[17] A. Greenberg, J.R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D.A. Maltz, P. Patel, S. Sengupta, VL2: a scalable and flexible data center network, in: SIGCOMM '09, ACM, New York, NY, USA, 2009, pp. 51–62. doi:http://doi.acm.org/10.1145/1592568.1592576.
[18] DataCenterDynamics, Increasing infrastructure utilization rates is key to cloud providers' success <http://goo.gl/ey334>.
[19] P. Fraigniaud, C. Gavoille, Routing in trees, in: ICALP '01, 2001, pp. 757–772.
[20] M. Al-Fares, A. Loukissas, A. Vahdat, A scalable commodity data center network architecture, in: SIGCOMM '08, ACM, New York, NY, USA, 2008, pp. 63–74. doi:http://doi.acm.org/10.1145/1402958.1402967.
[21] D. Krioukov, F. Papadopoulos, M. Kitsak, A. Vahdat, M. Boguñá, Hyperbolic geometry of complex networks, Physical Review E 82 (3) (2010) 036106.
[22] Worldmaking Blog, Hyperbolic Geometry: Poincaré Disc Tessellation How-to <http://goo.gl/J0Se3>.
[23] David E. Joyce, Hyperbolic Tessellations <http://goo.gl/YlhWR>.
[24] L. Popa, S. Ratnasamy, G. Iannaccone, A. Krishnamurthy, I. Stoica, A cost comparison of datacenter network architectures, in: Proceedings of the 6th International COnference on emerging Networking EXperiments and Technologies (CoNEXT), ACM, 2010, p. 16.
[25] Arista <http://www.aristanetworks.com>.
[26] O. Paridaens, Y. Imai, R. Boivie, W. Livens, D. Ooms, Explicit multicast (xcast) concepts and options, RFC 5058 IETF.
[27] A. Aloisio, V. Izzo, S. Rampone, FPGA implementation of a Greedy algorithm for set covering, in: Real Time Conference, 2005, 14th IEEE-NPSS, IEEE, 2005, p. 5.
[28] P. Elias, A. Feinstein, C. Shannon, A note on the maximum flow through a network, IRE Transactions on Information Theory 2 (4) (1956) 117–119.
[29] H. Frey, I. Stojmenovic, On delivery guarantees of face and combined greedy-face routing in ad hoc and sensor networks, in: Proceedings of the 12th Annual International Conference on Mobile Computing and Networking, ACM, 2006, pp. 390–401.
[30] A. Cvetkovski, M. Crovella, Hyperbolic embedding and routing for dynamic graphs, in: INFOCOM, IEEE, 2009, pp. 1647–1655.
[31] IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2008 (2008) 1–58, http://dx.doi.org/10.1109/IEEESTD.2008.4610935.
[32] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, S. Lu, Dcell: a scalable and fault-tolerant network structure for data centers, in: SIGCOMM '08, ACM, New York, NY, USA, 2008, pp. 75–86. doi:http://doi.acm.org/10.1145/1402958.1402968.
[33] Microsoft, Utilities embracing new technology concepts like cloud, software plus services and modular data centers, December 2008 <http://goo.gl/WKsG6>.
[34] P. Gill, N. Jain, N. Nagappan, Understanding network failures in data centers: measurement, analysis, and implications, in: Proceedings of SIGCOMM, 2011.
[35] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, OpenFlow: enabling innovation in campus networks, SIGCOMM Computational Communication Review 38 (2) (2008) 69–74.
[36] OpenFlow Switch, <http://openflow.org/>.
[37] F. Németh, A. Stipkovits, B. Sonkoly, A. Gulyás, Towards SmartFlow: Case studies on enhanced programmable forwarding in OpenFlow switches, in: ACM SIGCOMM, Helsinki, Finland, 2012, pp. 85–86.
[38] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, S. Shenker, NOX: towards an operating system for networks, SIGCOMM Computational Communication Review 38 (3) (2008) 105–110.
[39] OpenWrt <http://openwrt.org/>.
[40] J. Mudigonda, P. Yalagandula, J.C. Mogul, Taming the flying cable monster: a topology design and optimization framework for data-center networks, in: Proceedings of USENIX ATC'11, USENIX, 2011.
[41] M. Boguna, D. Krioukov, K.C. Claffy, Navigability of complex networks, Nature Physics 5 (1) (2009) 74–80. doi:http://dx.doi.org/10.1038/nphys1130.
[42] R. Kleinberg, Geographic routing using hyperbolic space, in: INFOCOM 2007, 26th IEEE International Conference on Computer Communications, IEEE, 2007, pp. 1902–1909.
[43] F. Papadopoulos, D. Krioukov, M. Bogua, A. Vahdat, Greedy forwarding in dynamic scale-free networks embedded in hyperbolic metric spaces, in: INFOCOM Proceedings, IEEE, 2010, pp. 1–9.
[44] M. Boguna, F. Papadopoulos, D. Krioukov, Sustaining the internet with hyperbolic mapping, Nature Communications 1 (6) (2010) 1–8.
[45] L. Gyarmati, T. Trinh, Scafida: A scale-free network inspired data center architecture, ACM SIGCOMM Computer Communication Review 40 (5) (2010) 4–12.

Márton Csernai is a Ph.D. student at the Department of Telecommunications and Media Informatics, Budapest University of Technology and Economics, Hungary (BME-TMIT). He received his MSc degree in Electrical Engineering from BME in next-generation communication networks. He studied one semester at Universität Karlsruhe (TH), Germany in 2006–2007 with DAAD scholarship. He has participated in various national and EU research projects in social networks inspired future routing architectures and energy efficient wireless communications. His main research fields are scalable routing technologies, next-generation communication networks, and complex networks.

András Gulyás received M.Sc. and Ph.D. degree from Budapest University of Technology and Economics (BME), Hungary in 2002 and 2008 respectively. Currently he is an assistant professor at the Department of Telecommunications and Media Informatics, BME. His current research areas spans over novel routing mechanisms, social and complex networks, and self-organizing networks.

Attila Kőrösi is an engineer at the Budapest University of Technology and Economics, Hungary working at the High Speed Network Laboratory. He received a M.S. degree in Mathematics from the Budapest University of Technology and Economics in 2007. His main research interest are FIB compression, multimedia services and their stochastic models. His prior research focused on priority queuing systems.

Balázs Sonkoly is an assistant professor at the Department of Telecommunications and Media Informatics, Budapest University of Technology and Economics, Hungary (BME). He received his Ph.D. (2010) and M.Sc. (2002) degrees in Computer Science from the BME. He has participated in several national projects and involved in EU projects as well. His research activity focuses on OpenFlow, novel routing mechanisms, congestion control, traffic modeling and intelligent transportation systems.

Gergely Biczók is a postdoctoral fellow at the Norwegian University of Science and Technology. He earned his Ph.D. in Computer Science at Budapest University of Technology and Economics in 2010. He spent one year at Northwestern University as a Fulbright visiting researcher from 2007 to 2008. He was also a research fellow at Ericsson Research from 2003 to 2007. His current research focuses on data centers, novel forwarding mechanisms and network economics.