Distributed self-adjusting trees in demand-aware reconfigurable networks

Report to be submitted in partial fulfillment of the requirements for the degree

of

Bachelors of Technology

in

Computer Science and Engineering

by

Bhargav Kulkarni
18CS02003

Under the supervision of

Dr. Joy Chandra Mukherjee

SCHOOL OF ELECTRICAL SCIENCES

INDIAN INSTITUTE OF TECHNOLOGY BHUBANESWAR

May 2022

Abstract

Distributed self-adjusting trees in demand-aware reconfigurable networks

by
Bhargav Kulkarni

We consider the problem of designing distributed, locally self-adjusting tree networks that adapt dynamically to the incoming traffic they serve. Due to emergence of technologies that are able to dynamically relocate links between pairs of **top-of-rack (ToR) switches**, there has been a surge in study and design of such technologies. Such networks offer great potential in application of data structures that are bounded in degree and reconfigure themselves in the face of demand patterns they serve. In the following sections, we consider **SplayNet**, a fully distributed version of the self-adjusting binary search tree (BST) data structure called **splay tree**, proposed by Sleator and Tarjan to attempt to minimize the routing cost between arbitrary pairs of nodes communicating in the network. In understanding SplayNet, we will also explore splay trees.

Finally, we will look at the scope of expanding the problem of designing a distributed data structure for reconfigurable network. Since all the network models can be modelled as bounded-degree graphs, there is a possibility of applying various self-adjusting data structures for the same.

List of Figures

Chapter 1

Introduction

Data centre networks are traditionally designed in such a manner that decision must be made in advance by the administrator(s) regarding the ToR switches that must be connected to each other and the capacity that must be provisioned to each such link based on the heuristics of expected demand so that the network may smoothly respond to the queries of communication between arbitrary ToR nodes. There are fundamental limitations to this model. Firstly, the administrator(s) are required to know demand heuristics to design such networks. While handling requests of decentralized nature, it is often difficult to predict the behaviour of static networks under such statistics. Secondly, such networks perform badly while handling requests with high temporal locality, which is a commonplace observation in real-world network demands[2]. Re-configurable networks are emerging which enable network topology to dynamically change over time[3]. With optical links being the communication medium between ToRs, the optical links from a ToR can be switched to connect other ToRs.

These dynamically re-configurable networks are similar in spirit to self-adjusting data structures. Sleator and Tarjon proposed **splay tree**[4], a data structure that moves frequently accessed elements to root to reduce amortized access time of elements. A major difference between these data structures and re-configurable networks is that requests in Splay Trees arrive at root, which route it to the element concerned, whereas communication request in networks arrive directly at the source node of the request. Hence there is a need to model these networks as distributed data structures where decisions are made locally and re-configurations are local to improve performance.

One such modelling of re-configurable networks is by utilizing self-adjusting binary trees such as splay trees[1]. Using splay trees, it is possible to construct a fully distributed, self-adjusting splay tree network that adapts the topology to the workload automatically and in an online manner (i.e., without knowledge of future demand), handling requests sequentially. We can evaluate the self-adjusting network on two metrics: (1) The amortized work, which is similar to the performance measures used in the context of self-adjusting data structures. It measures the cost of routing on and adjusting the network, and (2) The more important makespan or time cost, which measures the time it takes to serve a set of communication requests, we will argue over optimizations possible on top of SplayNet.

1.1 Problem statement

In the context of data centres, our focus is to design a distributed network model which supports link re-configurations, handles communication requests in an online manner and optimizes on parameters of work cost and time cost. It is preferred to have a completely decentralized model where all decisions regarding reconfigurations and routing are made locally.

Chapter 2

Background

2.1 Network model

The network consists of a set $V = \{v_1, v_2, ..., v_n\}$ of n nodes (ie., top-of-rack switches). The input to the network design problem is a traffic demand, given as a sequence $\sigma = (\sigma_1, \sigma_2, ..., \sigma_m)$ of m communication requests $\sigma_i = (s_i, d_i)$ occurring over time, with source s_i and destination d_i ; m can be infinite as the the network deals with requests in an online manner. We use b_i to denote the time when a request $\sigma_i = (s_i, d_i) \in V \times V$ is generated, and e_i to denote the time in which it is completed. The times between successive arrivals between requests is assumed to be at least one. An external agent that redirects requests to SplayTree will ensure this. Moreover, the sequence σ can be arbitrary: we consider a worst-case scenario, where σ is chosen adversarially, in order to maximize the cost of a given distributed algorithm. When serving these communication requests, the network can adjust, and we denote the sequence of network topologies over time by $G_1, G_2, \ldots$. However, we require that each G_i belongs to some "desirable" graph family G. In particular, for scalability reasons and due to the fact that a system can only directly connect to a maximum number of systems, the networks should be of constant degree.

We describe our cost model as follows. In order to minimize reconfiguration costs and adjust the topology smoothly over time, the tree is reconfigured locally through local rotations that preserve the BST properties. We assume that the work cost of each link change is a unit and that the communication costs one unit per link. We neglect the communication costs as it is of the order of reconfiguration cost.

For an initial BST T_0 and for a sequence of requests σ, **work cost** can be described as the total number of reconfiguration steps performed in satisfying σ.

Another important parameter to measure the service of network requests is time. Model of time considered is trivial, analogous to real time, considering the timestamps when the requests arrive and when they are served. We define **makespan** to be the total time taken to serve all requests (assuming that the network never sits idle and is engaged in serving the requests). Since synchronous algorithms execute in **rounds**, we consider here makespan to be the total number of rounds taken to serve all the requests.
Makespan $T(T_0, \sigma) = \max_{1 \leq i \leq m} e_i - \min_{1 \leq i \leq m} b_i$;

Also, an additional assumption considered is that the network never sits idle if it has more requests to serve. Since we are studying the behaviour of the algorithm over arbitrary request sequences, we make theoretical amortized analysis of the algorithm. Informally, in amortized analysis, we calculate average cost over that set of communication requests σ and that initial BST configuration T_0 that yields us the worst (or highest) cost.
Amortized cost $= \max_{\sigma, T_0} \frac{1}{m}(\sum_{i=1}^{m} \sigma_i)$

2.2 Splay trees

The splay tree is a self-adjusting form of binary search tree. Splay trees support the operations of insertion, access and deletion. When an item is to be accessed from splay tree, the data structure self-adjusts to move the item to the root if it is present, while preserving the binary search tree properties, and then reports the element. The motivation for self-adjusting data structures in general to perform re-configurations is to reduce the cost that future requests have to incur. In typical applications of self-adjusting data structures, the goal is to minimize the total time taken to serve a sequence of requests and not to optimize on an individual operation. This goal of "amortized efficiency" motivates us to consider self-adjusting data structures like splay trees, which self-adjusts to improve the efficiency of future requests.
The basic idea on which splay trees are based on is as follows: employ a simple way of restructuring the tree so that after each access, the accessed item and the items along the path are moved closer to root under the plausible assumption that the item

is likely to be accessed again. In other words, under skewed request patterns which have high temporal locality, splay tree significantly improves access time as after first accessing an element, the probability it will be accessed again in near future is high. To move an accessed element up to root, splay tree employs certain restructuring steps repeatedly on the element until it reaches root of the binary search tree. In each step, the element is moved two levels higher than its current level (if the element is a child of root, it will only be moved one level higher to reach root).

This restructuring step of moving a node up to possibly two levels higher at a time is called a **splay** step. There are three types of splaying operations. These are − **zig** (a single rotation), **zig-zig** (double rotation with staggered node-parent-grandparent relationship) and **zig-zag** (double rotation with monotonous node-parent-grandparent relationship).

In the below figures, the node to be accessed is x, and is splayed accordingly.

Figure 2.1: Zig operation

Figure 2.2: Zig-zag operation

Figure 2.3: Zig-zig operation

The left and right rotations of a BST are applied in the splay operations. The splay operation to be applied on a node is decided upon the level of the node and its relationship with its parent and grandparent. The adequate splay operations are repeatedly applied on the accessed node until it reaches root. While moving the element to root, splay operations also roughly halve the depths of elements present along the access paths, as can be observed in the below figures. Producing this effect has important consequences in dealing with such request patterns where spatial locality (accessing elements around the currently accessed element in near future) is present.

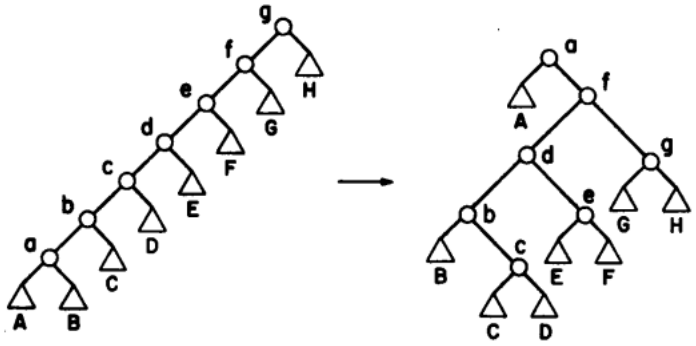

Figure 2.4: All zig-zig steps

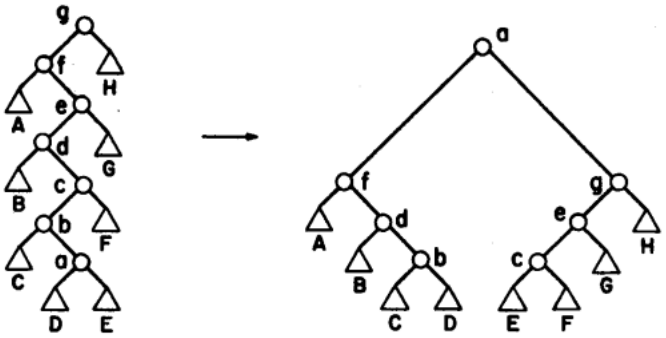

Figure 2.5: All zig-zag steps

Another property of splay trees which interests us is its performance against the static optimal BST. Static optimal BST for a set of access requests ρ is a static BST that has the lowest total access cost among all static BSTs (An offline dynamic programming algorithm exists that computes such a BST). Let the cost be $\zeta(\rho)$. If

for an initial BST T_0, $COST(Splay, T_0, \rho)$ represents the amortized access cost in satisfying ρ, then it has been proven that:
$COST(Splay, T_0, \rho) = O(\zeta(\rho))$.
Stating it informally, splay tree handles requests in an online manner at the expense of cost which of the order of minimal cost that can be invoked by a static BST. This result will be of importance to us.

2.3 Distributed algorithm for handling requests in re-configurable network

We attempt to utilize the splay tree data structure in devising a distributed algorithm for communication between requested source and destination nodes in a reconfigurable network. In the network, the communicating nodes need to splay, in a manner similar to splay trees. Since a source node need only communicate with the destination, the communicating nodes do not splay to the root, but to their **lowest common ancestor (LCA)**. In a binary search tree T, for two nodes u and v, their lowest common ancestor $\alpha_T(u, v)$ is defined as the node which is an ancestor of both u and v that is located farthest away from root.
The algorithm can simply be listed as follows:

Distributed Algorithm

---

1. for requests (u, v) in σ:
2. while $u \neq \alpha_T(u, v)$:
3. $T =$ **splay u**
4. while $parent_T(v) \neq u$:
5. $T =$ **splay v**

In the algorithm, the source nodes are required to be able to detect if they are LCA of themselves and the destination node. To enable this discovery, each node in the binary tree, in addition to storing the node IDs of its parent, left and right children, store the IDs of smallest and the largest elements in its subtree. For a node u, we will denote its parent as $u.parent$, its right child as $u.right$, left child as $u.left$, the smallest element in its subtree as $u.smallest$ and the largest element as $u.largest$.

A node u can locally verify if it is the LCA of two nodes, itself and a node v if the following holds: $v \geq u.smallest$ && $v \leq u.largest$. Message passing procedures can be made use of where a node, if not the LCA, will redirect a request to its parent querying if it is the required LCA. These decisions helps a node to decide on which splaying operations to perform (zig, zig-zig or zig-zag).

The model of the distributed algorithm can be envisioned as following:

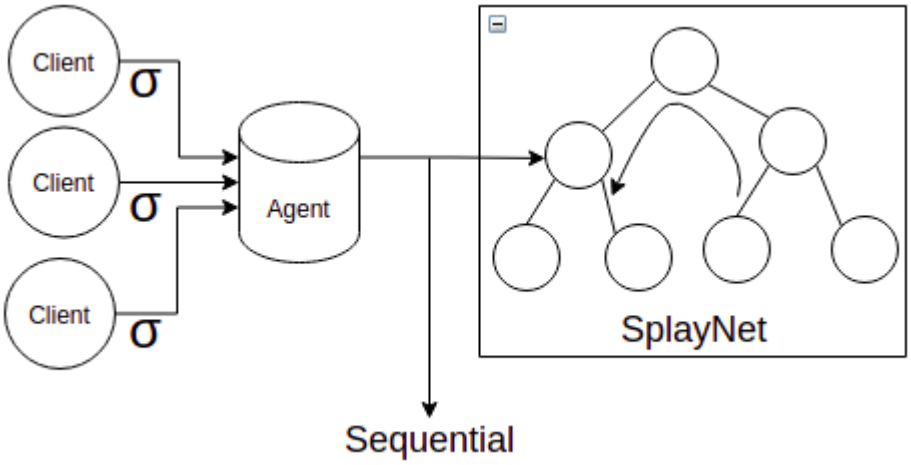

Figure 2.6: SplayNet model

Since the distributed algorithm handles all the incoming requests sequentially (does not concurrently accept requests), an 'agent' is required that accepts all client requests and sequentially feeds them to the network.
The algorithm can be better understood with the help of an example. Consider an initial configuration of the BST network T_0. A request $\sigma_1 = (25, 35)$ arrives.

The LCA Discovery algorithm initiated by node 25 (being the source) to find the lowest common ancestor of itself and the node 48 will return 35. Hence, 25 will splay to the location of the lowest common ancestor. In the next step, the destination node (node 48) will have to splay to the child of the source node, which is node 38 in this case, as shown in Fig. 2.6. After the two splay operations, the source and destination nodes are adjacent and hence can communicate with each other, as shown in Fig. 2.7. The end of their communication will mark the end of the request σ_1 and the SplayNet

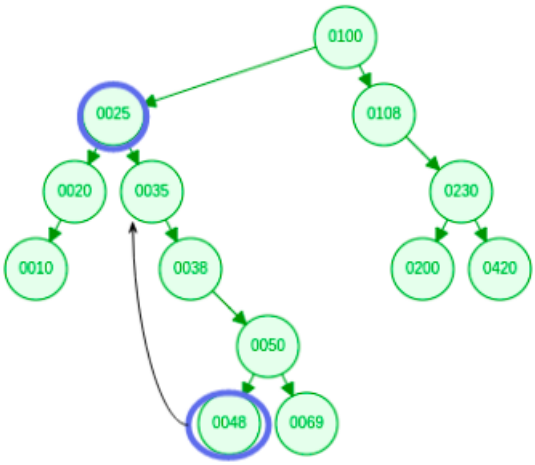

Figure 2.7: Intial configuration with request arrival

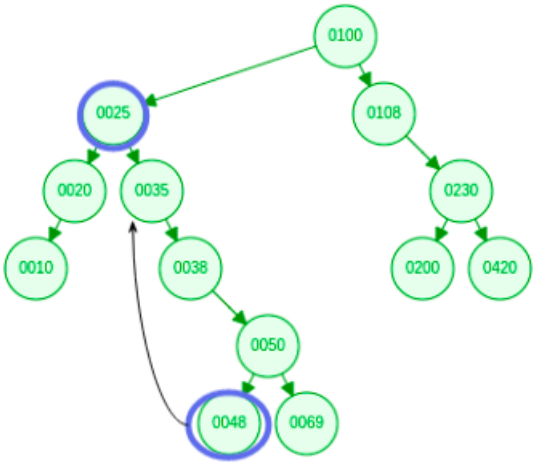

Figure 2.8: Destination node splayed to child of source

will become available for another request.

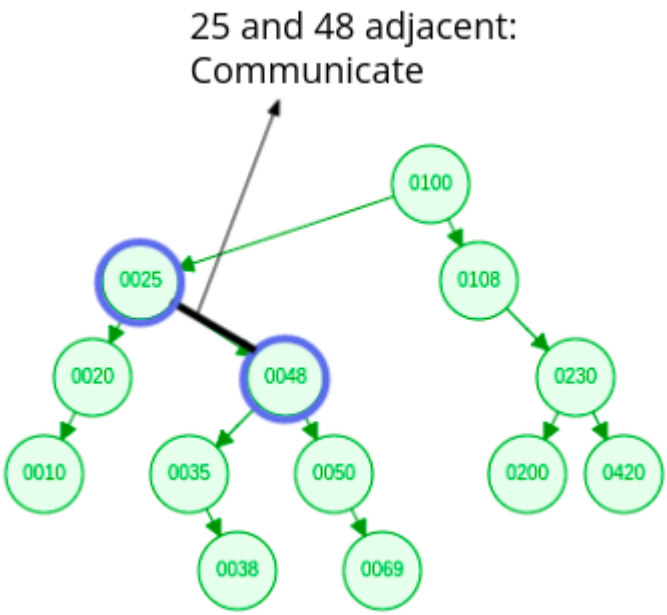

Figure 2.9: Source and destination nodes are adjacent

Chapter 3

Implementation and results

The algorithm has been implemented on Sinalgo network simulator, which is written in Java. The algorithm is completely event driven by messages, with each message triggering an action that a node in the network has to take. A 'client' node stores all the requests that the network should satisfy and feeds the request to the source sequentially after previous request has been satisfied. The nodes are initially connected in a binary tree fashion, each assigned with a unique identifier. Without loss of generality, a node is designated to be the root of the tree. Root initiates a **Membership protocol**, which is a broadcast-convergecast based algorithm for each node to identify its parent, left child, right child, smallest and the largest node within its subtree.

After the membership protocol terminates, all nodes would have set their parent, left and right children, and smallest and largest descendent correctly. The client node then sends out communication requests to source nodes. Source nodes, upon receiving request, perform splay operations repeatedly till it reaches the LCA of itself and the destination node. It then sends out request to the destination node to splay to its child. The destination node, after splaying, sends out an acknowledgement to the source node, marking the end of the communication.
Since a splay operation involves consecutive rotations, we specify the implementation of a rotation operation. A right rotation is performed as shown in the figure below.

In order to achieve the rotation, appropriate links have to be changed. This is done by passing messages to concerned nodes, where appropriate information about which links to reconfigure and to which node to connect is present.

Algorithm 1 Membership protocol

```
if isRoot() then
  broadcast(MembershipMsg)
end if
** Invoked when a message is received **
if receivedMsg(msg) then
  if msg instanceof MembershipMsg then
    setParent(msg.src)
    for all node ∈ Ngbr \ msg.src do
      sendMessage(node, MembershipMsg)
    end for
    if |Ngbr| = 1 then
      ** Leaf node **
      sendMessage(msg.src, MembershipAck)
    end if
  end if
  if msg instanceof MembershipAck then
    if msg.src.ID < this.ID then
      setLeftChild(msg.src.ID)
    else
      setRightChild(msg.src.ID)
    end if
    setSmallestDesc(min(this.smallestDesc, msg.src.smallestDesc))
    setLargestDesc(max(this.largestDesc, msg.src.largestDesc))
    if numAcksRecv = numChildren then
      sendMessage(this.parent, MembershipAck)
    end if
  end if
end if
=0
```

Algorithm 2 Right rotate

```
sendMessage(this.grandParent, ReconfigureNeighbourMsg(RightChild, this))
sendMessage(rightChild, ReconfigureNeighbourMsg(Parent, parent))
sendMessage(parent, ReconfigureNeighbourMsg(LeftChild, rightChild))
sendMessage(parent, ChangeSmallestDesc(rightChild.getSmallestDesc()))
this.changeLargestDesc(parent.getLargestDesc())
=0
```

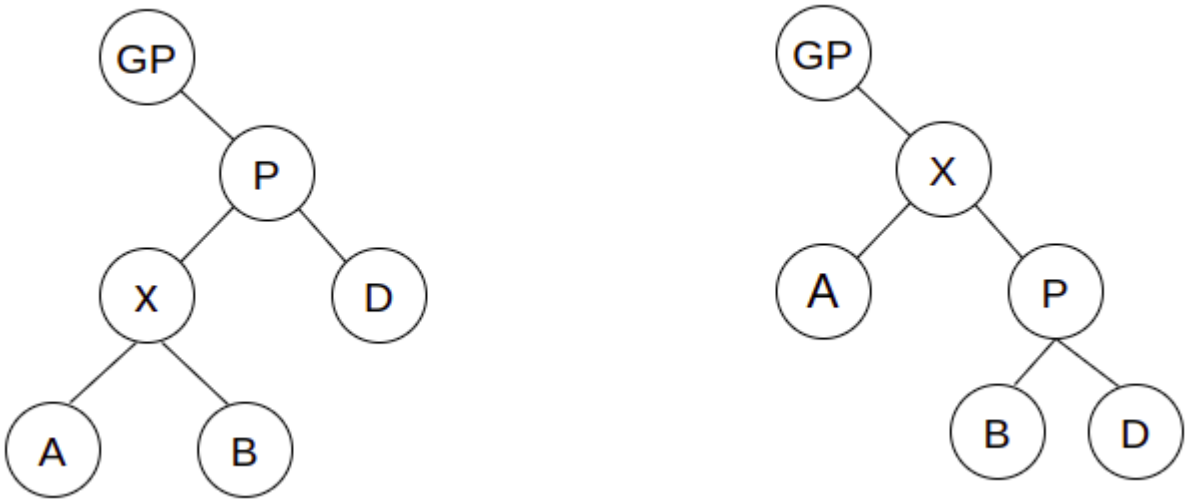

Figure 3.1: Right rotation of node X

Left rotation too will have its implementation similarly defined. All the changes will take place in the round following the one where the function is called. Source will splay to the required lowest common ancestor. After that, it sends to destination the request to splay, where the destination will splay to the child of the source.

Algorithm 3 Source splay($dest$)

```
if dest ≥ this.smallestDesc && dest ≤ this.largestDesc then
  sendMessage(dest, SplayToChild)
else if dest ≥ parent.smallestDesc && dest ≤ parent.largestDesc then
  this.performZig()
  Source splay(dest)
else if this.ID < parent.ID && parent.ID < grandParent.ID then
  this.performZigZig()
  Soure splay(dest)
else
  this.performZigZag()
  Soure splay(dest)
end if=0
```

The distributed algorithm has been run on a tree with 255 nodes serving 100 requests. The behaviour of the algorithm has been studied against temporal locality of requests. In particular, the parameter considered for generating requests is − for each request, how probable it is to match one of the previous 10 requests issued. This

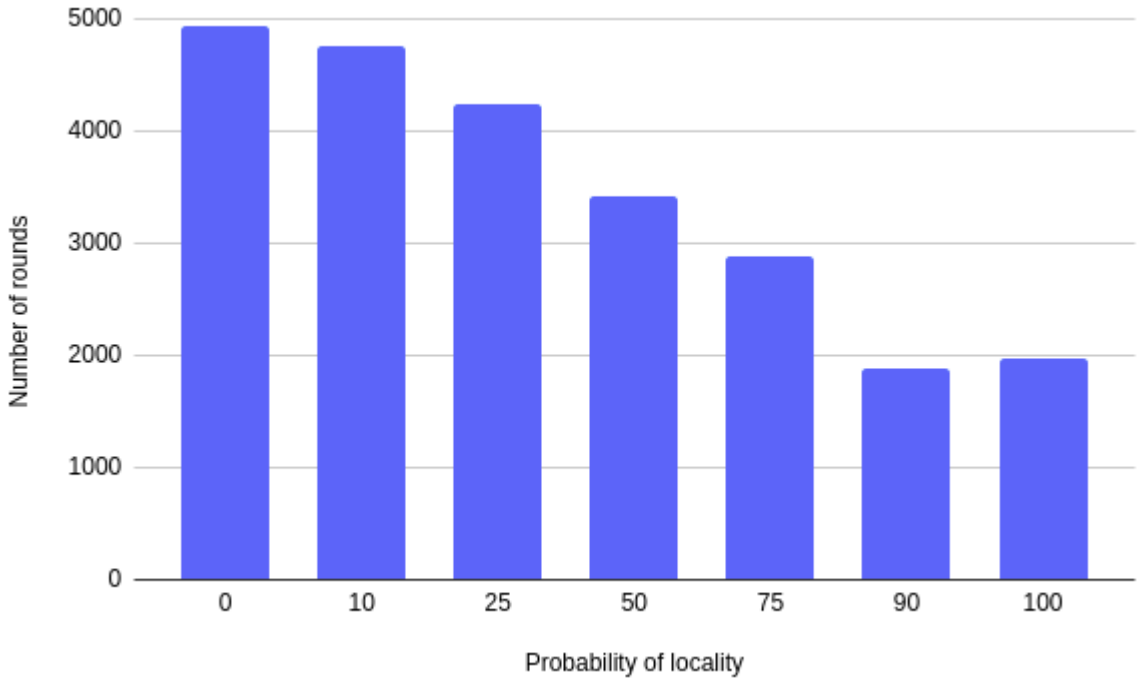

Figure 3.2: Makespan v/s probability of locality of requests X

probability has been varied and the number of rounds the algorithm took to serve all the requests has been studied. A probability of 0 indicates that the requests generated are i.i.d. An average of 3 findings is considered for our readings. The following results were generated.

The results confirm our assessment of the distributed algorithm over self-adjusting tree that with increase in temporal locality of requests, the algorithm performs better in serving the sequence of requests.

Chapter 4

Scope

The self-adjusting model described based on splay trees, despite being distributed, handles all requests sequentially. This is done to avoid any clashes in reconfigurations while splaying the nodes. Consider the below example of handling two requests concurrently:

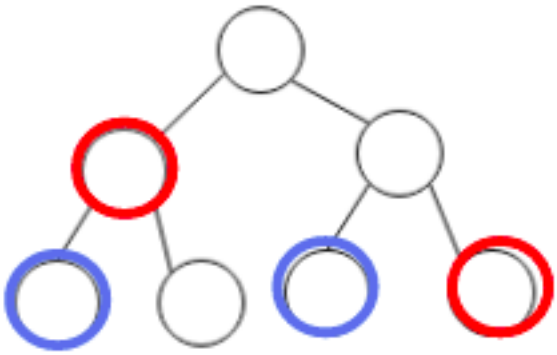

Figure 4.1: Clashing requests

The blue nodes form a $(source, destination)$ request and the red nodes form another such request. Both the requests have root of the BST as their LCA and so, while re-configuring the links, the requests must be considered sequentially. But consider another request as shown in Fig. 3.2.
The link re-configurations in this case would not clash. But the SplayNet would still consider these requests sequentially. These requests can be concurrently served[2]. The work cost will not improved but makespan can be significantly reduced by considering such '$disjoint$' requests concurrently. A concurrent, distributed model would enhance throughput and significantly reduce the total time to serve the set of requests.

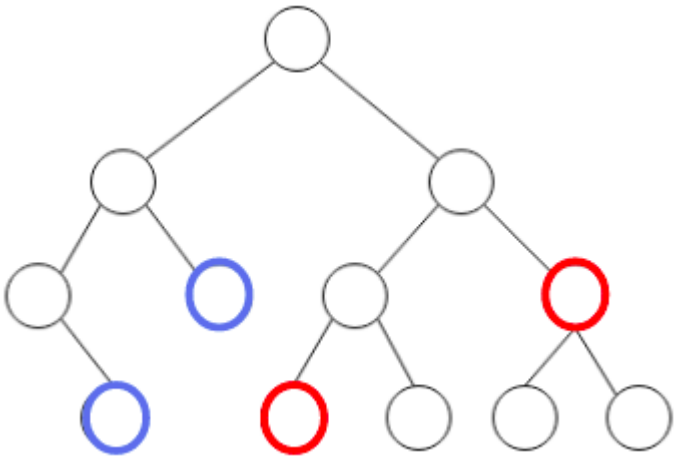

Figure 4.2: Example request with disjoint link changes

We have incorporated splay trees as our self-adjusting data structures. The primary reason for choosing splay trees is its simplicity. Splay trees have been proposed by Sleator and Tarjon in 1985. Since then, many other self-adjusting graph structures have also been proposed: link/cut tree, tango tree, top tree, CB (counting based) binary search tree, dynamic skip-graph, among others. The problem of demand-aware re-configurable network can be explored in the context of these data structures for optimized work. There is also a possibility to look into a more practical asynchronous model and extend the data structure along those directions.

Chapter 5

Conclusion

Demand aware re-configurable networks is an emerging area of research work that presents us with great opportunities to build distributed algorithms on top of them to improve on average latency. Designing a distributed data structure based on splay trees is a great starting point to explore other alternatives and cost models as the requirements fit. The simplistic nature of splay trees itself provide a groundwork in modifying it to optimize along different constraints. As the request sequences generated in actual data centers are far from i.i.d, the skewed nature of requests due to temporal and spatial locality can be exploited to serve requests more efficiently. We have made an attempt to explore such an aspect and showed that there is indeed a possibility where we can design networks and algorithms specific to certain nature of request sequences (temporal locality here).

Bibliography

[1] Monia Ghobadi et al. "ProjecToR: Agile Reconfigurable Data Center Interconnect". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: Association for Computing Machinery, 2016, pp. 216–229. ISBN: 9781450341936. DOI: 10.1145/2934872.2934911. URL: https://doi.org/10.1145/2934872.2934911.

[2] Bruna Peres et al. "Distributed Self-Adjusting Tree Networks". In: *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*. 2019, pp. 145–153. DOI: 10.1109/INFOCOM.2019.8737417.

[3] Stefan Schmid et al. "SplayNet: Towards Locally Self-Adjusting Networks". In: *IEEE/ACM Transactions on Networking* 24.3 (2016), pp. 1421–1433. DOI: 10.1109/TNET.2015.2410313.

[4] Daniel Dominic Sleator and Robert Endre Tarjan. "Self-Adjusting Binary Search Trees". In: *J. ACM* 32.3 (July 1985), pp. 652–686. ISSN: 0004-5411. DOI: 10.1145/3828.3835. URL: https://doi.org/10.1145/3828.3835.