

HELSINKI UNIVERSITY OF TECHNOLOGY
Faculty of Electronics, Communications and Automation

Jegadish Devadoss

DATA DISTRIBUTION OVER AN OVERLAY NETWORK

The thesis is submitted for examination for the degree of Master of Science in
Technology

Espoo 15.1.2009

Thesis supervisor:

Prof. Jörg Ott

Thesis instructor:

Igor Curcio

Author: Jegadish Devadoss

Title: Data distribution over an overlay network

Date: 15.1.2009

Language: English

Number of pages: 5+55

Faculty: Faculty of Electronics, Communications and Automation

Professorship: Networking technology

Code: S-38

Supervisor: Prof. Jörg Ott

Instructor: Igor Curcio

The Client-Server model based data distribution is inefficient for sessions with a large number of participants interested in receiving the same content at the same instant. Examples of such applications are live audio/video streaming, weather updates, stock tickers etc. The lack of global multicast infrastructure has made the research community to consider 'Overlay networks' as alternatives. Overlay networks require effective mechanisms for bootstrapping, constructing, maintaining and repairing the overlay. The effectiveness of these mechanisms influences the quality of the service experienced using the overlay network. In this thesis, we propose solutions that can be used by the overlay network to construct, maintain and repair the overlay. More precisely, the solutions that we propose can construct a minimum spanning tree for data distribution and identify capable (nodes with extra outbound degree) nodes using a decentralised design. Overlay networks can be classified into different types depending on the nature of the participants and the type of data distribution mechanism (tree, mesh). In the thesis, our focus is only on the overlay networks that uses tree based data distribution mechanism.

Keywords: Overlay networks, Algorithms

Preface

This work started with the thoughts on the algorithms that can be efficient for constructing a data distribution tree in an overlay network. Personally, This work has been a transforming journey that has given me directions on how to understand and design a system.

I am very grateful to Professor. Jörg Ott who agreed to supervise my thesis. The freedom that he gave to explore and his valuable suggestions/pointers are very significant in shaping this thesis. I also would like to thank my instructor Igor Curcio for his valuable suggestions. Finally, I thank my lab colleagues, who had given me, many valuable suggestions and were very encouraging.

Otaniemi, 15.1.2009

Jegadish Devadoss

Contents

Abstract	ii
Preface	iii
Contents	iv
1 Introduction	1
1.1 IP Multicast	1
1.2 Overlay Networks	2
1.3 Overlay based data distribution	3
2 Scope, Problem statement and Related work	5
2.1 Scope	5
2.2 Problem Definition	7
2.3 Related Work	8
2.4 Summary	9
3 Distributed Tree Routing	10
3.1 Overview of the algorithm	10
3.1.1 Local Correction-1	10
3.1.2 Local Correction-2	11
3.1.3 Local Correction-3	12
3.1.4 States maintained by the nodes	12
3.2 Locks for DTR execution	13
3.3 An Example	15
3.4 Metrics for evaluation	17
3.5 Evaluating DTR performance using simulation	21
3.6 Analysing the efficiency	23
3.6.1 Operation of DTR in degree constrained environment	23
3.7 Applications for this algorithm	26
3.8 Conclusion	26
4 Co-Operative Resource Identification Protocol	28
4.1 Operation of CORIP	28

4.1.1	Operation of R_{up}	28
4.1.2	Operation of R_{down}	33
4.2	Applications for the protocol	36
4.2.1	Pro-Active Optimisation	36
4.2.2	Reactive Repair	40
4.3	Future Work and Conclusion	40
5	Overlay network simulator design	41
5.1	Event scheduling mechanism in Ns2	41
5.2	Design and implementation details of an overlay node	42
5.2.1	Design overview	42
5.2.2	Implementation Details	43
5.3	Design and implementation overview of DTR	48
5.3.1	Local correction-1:	49
5.3.2	Local correction-2 and 3:	50
5.4	Summary	51
6	Conclusion and Future Work	52
	References	53

1 Introduction

The Client-Server model has been successfully applied by many Internet applications. Examples of such applications are web, mail, file transfer etc. These applications have been the catalyst for the growth of the Internet. In Client-Server based systems, a participating entity can either be a client or a server. Servers are used for storing the content that needs to be distributed. Each client makes an individual connection to access the content hosted by the server.

The content distributed in the Internet can be either stored or live data. In live data distribution, the data is periodically generated and sent to the interested receivers. In this form of distribution, the generated data has a validity period associated with it. The data is not usefull, if it reaches the receiver after the expiry of its validity period. Examples of live data distribution applications are live media streaming, weather updates etc.

Applications like media streaming consume significant bandwidth. Typically, streaming video consumes few hundred Kbps to few Mbps. The share of video traffic in the total Internet traffic, is getting more significant. In the case of live events like football match, music concert and election, there are many viewers who are interested in watching them live. In such scenarios, the percentage of duplicate traffic in the underlying network links, is very high. For example, let x be the bit rate of the streamed video and n be the number of participants viewing it simultaneously. For the case of client-server model, the server need to have outbound bandwidth that is at least $x \times n$. For $x = 400$ Kbps and $n = 1000$, the outbound bandwidth required for the server is 400 Mbps. In addition, if a significant number of users are behind a set of common links, then the percentage of duplicate traffic in those common links shall be very high. This can result in congestion and can degrade the performance to both the viewers and non-viewers of the event.

Applications like weather/news/stocks updates and real time updates from sensor equipments, continuously send updates on the current situation. If every participant where to connect as a client, then we see the same issues discussed in the previous paragraph. The increased use of the above applications and the inefficiency of the client-server model to support these type of applications necessitate the need for one-to-many data distribution model.

The high outbound bandwidth requirements and inefficient network utilisation motivates us to consider IP multicast and overlay-network based data distribution as alternative solutions. In the subsequent subsections, we discuss on their applicability and effectiveness in today's scenario.

1.1 IP Multicast

IP multicast operates over IP and the class 'D' (224.0.0.0 to 239.255.255.255) addresses are reserved for the use of IP multicast applications. It requires the source to send only one copy of the data and also at any instant, only one copy of the data

is carried in the intermediate network links. The above two points make it the most efficient one-to-many data delivery mechanism.

Operating mechanism: A receiver that wants to join an IP multicast session, sends an Internet Group Management Protocol (IGMP) message to its router, informing it about its intentions to join a multicast session. The router responds by joining the corresponding data distribution tree. The typical protocol used by the router for this purpose is Protocol Independent Multicast (PIM). If a receiver decides to leave the session, it sends a IGMP leave request to its router. Also, When a router observes that it does not have any active receivers under its network, it detaches from the data distribution tree.

Deployment: Even though IP multicast is an efficient many-to-many data distribution mechanism, currently its global availability is very less. Organisations like universities and industries are increasingly using IP multicast within their local network. But, we do not have IP multicast sessions that are globally available. Issues like address management – choosing multicast address that does not conflict – and reluctance from the ISPs – economic viability – in enabling the service are some of the important bottlenecks for global deployment. Also, an IP multicast application programmer need to make sure that his application supports dynamic IP multicast addressing, session management, heterogeneous receivers, security issues etc [1].

1.2 Overlay Networks

Overlay networks are networks that are built on top of another network. Nodes in the overlay are interconnected using virtual or logical links. These virtual or logical links can be considered as a path consisting of one or more physical links. The overlay network has no control on how the packets/messages are routed in the underlying network. But it can control the sequence in which a packet/message flow through the overlay nodes. In other words, the order in which the overlay nodes are connected is configurable.

Operating mechanism: There are different types of overlay networks and their classification are discussed in the second chapter. Overlay networks can be used to provide different types of services like, lookup service (DHT overlay), application layer multicast (ALM) etc. Here, we consider an overlay network used for ALM and explain its operating mechanism. In this form of overlay network, every participating node has a parent node from which it receives the data and it maintains a list of child nodes to which it forwards the copy of data. The participating nodes arrange themselves to form a data distribution tree. The important differences with respect to the IP multicast is illustrated in the figure 1. In the figure 1, 'S' represents the data source, 'R1, R2 and R3' represent the router and 'r1 and r2' represent the receivers. The data flow in both the mechanisms is illustrated in the figure 1. In

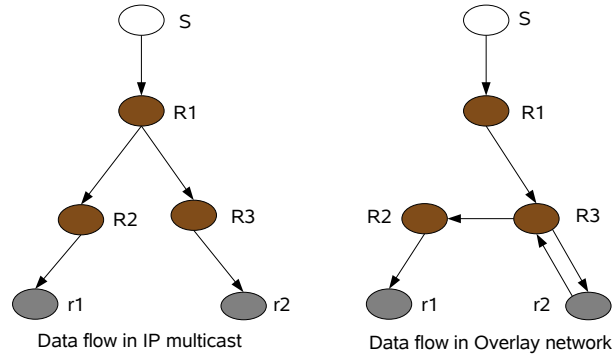


Figure 1: Data flow in IP multicast and overlay networks

the IP multicast, the data distribution tree is constructed in cooperation with the routers. In the overlay networks, the data distribution tree is constructed by the participants in the overlay network.

We see (figure 1) that in the case of overlay-network, the access network of the receiver 'r2' carries two copies of the data. So, With regard to efficiency of the solution, IP multicast is more efficient than the overlay-network based solution.

Effectiveness of Overlay networks: Overlay networks does not have the deployment issues discussed in the IP multicast section. Deploying overlay network does not require any change to the underlying network infrastructure. By using efficient tree construction mechanism in overlay-network based data distribution, we can effectively reduce the amount of duplicate traffic in the underlying network links. There are real-world solutions that are based on overlay networks. The examples include Akamai's [2] overlay-network that provides effective content delivery mechanisms. Akamai's solution maintains an overlay network of servers that are used to distribute data more efficiently. There are also many proposed solutions from the academic research projects [11, 12, 13], in this regard.

From the above analysis, we observe that there is a need one-to-many data distribution systems and overlay based approach is an effective mechanism for the current situation.

1.3 Overlay based data distribution

Designing an overlay based solution requires efficient and effective algorithms for data distribution and in handling the dynamics of the overlay network. These algorithms need to be distributed so as to scale for large number of participants. In this thesis, we propose distributed algorithms that can be used in designing a scalable overlay based data distribution solution.

In the second section, we classify the overlay networks and discuss on the different types of operations that are needed for the effective working of the overlay. In

the third section, we propose and analyse a data distribution tree construction algorithm. In the fourth section, we propose and analyse a resource identification protocol. We evaluated our solutions by implementing them as part of an overlay node in a discrete event simulator. In the fifth section, we discuss on the design details and the learnings from the process of implementing an overlay node in a discrete-event simulation environment. Finally, In the sixth section, We conclude and discuss on the future work.

2 Scope, Problem statement and Related work

In this section, we discuss about the scope, the problem statement and related work done by the research community.

2.1 Scope

Overlay networks can be used for different purposes like data distribution, content search etc. In this thesis, when we refer to overlay networks, we refer to the overlays used for one-to-many live data distribution. In live data distribution, every data packet has a threshold for the reception timestamp. If the time at which the receiver receives the data exceeds the threshold, then the data is discarded. To put our scope more precisely, we need to classify the overlay networks used for one-to-many live data distribution. The overlay networks can be classified depending on the mechanism of live data distribution and the nature of the nodes participating in the overlay[10].

Live data distribution mechanisms: Depending on how the data is delivered, the data distribution architecture can be classified as either Mesh-Pull or Tree-Push. A general overview of the mesh and tree based system are illustrated in the figure 2 and 3 respectively.

The Mesh-Pull[9] architecture is similar to the mechanism used in BitTorrent, but has been adapted to be used in live data distribution. Participating nodes periodically exchange buffer maps with peers and retrieve the missing content. One important difference from the BitTorrent is the use of peer selection algorithms that considers the time constraints imposed by the live data. Many deployed overlay based IPTV services like PPLive[4, 3], SopCast[5], CoolStreaming[7, 8], TVAnts[6] etc., use this approach.

The advantages of Mesh-Pull architecture is that it is robust to churn and the implementation complexity is less. The disadvantages are high end-to-end delay in receiving the data, high delay in channel switching time (moving from one data channel to other) and causes more stress to the underlying network.

In Tree-Push architecture, a data distribution tree is constructed based on the application criteria (like end-to-end delay). Every participating node receives data from its parent node and passes on the data to its child nodes. The operations involved in a tree push architecture can be classified as

1. *Bootstrapping:* This mechanism is used by a new node to join the overlay.
2. *Proactive tree construction/maintenance:* This mechanism is required to optimise/strengthen the data distribution tree.
3. *Tree repair mechanism:* When a participating node loses a parent node, it uses this mechanism to re-join the overlay.

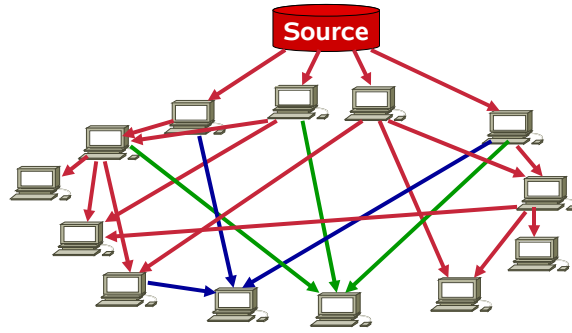


Figure 2: Mesh based systems

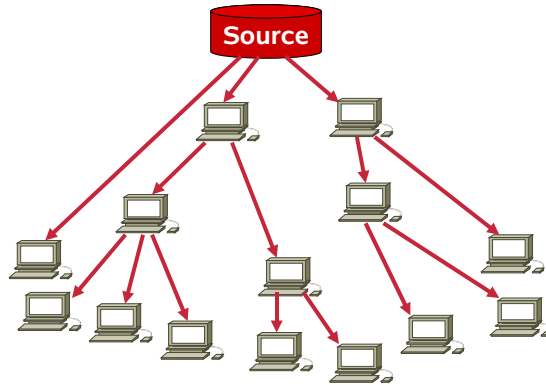


Figure 3: Tree based systems

The advantages of Tree-Push architecture are low end-to-end delay in receiving the data, low time lag between peers in receiving the data and use of an efficient tree construction mechanism can considerably reduce the stress to the underlying network. But the Tree-Push is not inherently as robust as Mesh-Pull in handling churn.

Classifying overlay nodes: Depending on the stability of the participating nodes, the nodes can be classified either user nodes or infrastructure nodes. Infrastructure nodes are highly stable nodes that are made part of the overlay-network to improve performance and stability. Depending on the type of nodes participating in the overlay-network, the overlay-network can be classified as,

1. Overlay network with only user nodes
2. Overlay network with both user nodes and infrastructure nodes
3. Overlay network with only infrastructure nodes.

Focus: For one-to-many live data distribution, the data need to reach the receivers with minimum end-to-end delay. If the delay in receiving the data from the data source is high, then it leads to bad user experience and high start-up delay. Also, the operation of overlay network should not stress the underlying network i.e. avoid the redundant data carried over a network link. In the earlier paragraphs, we had discussed the advantages and disadvantages of the different data distribution mechanisms. After analysing the discussion, we see that Tree-Push architecture can meet these requirements, provided the overlay is optimised for these requirements. Regarding the type of overlay nodes, we consider a network that may have only user nodes or both user and infrastructure nodes. In the subsequent sections, we present the problems involved in optimising the overlay-network for minimum end-to-end delay and minimum network stress.

2.2 Problem Definition

The problem is to construct an efficient data distribution tree. The constructed tree has to minimise the cumulative end-to-end delay of all nodes, in receiving the data. The tree also has to minimise the network stress. To put the term 'network stress' more clearly, we analyse the scenarios illustrated in the figure 4. In both the scenarios given in figure 4, the cumulative end-to-end delay in receiving the data is same. But in the second scenario, an efficient data distribution tree is used. So, the stress to the underlying network is minimum.

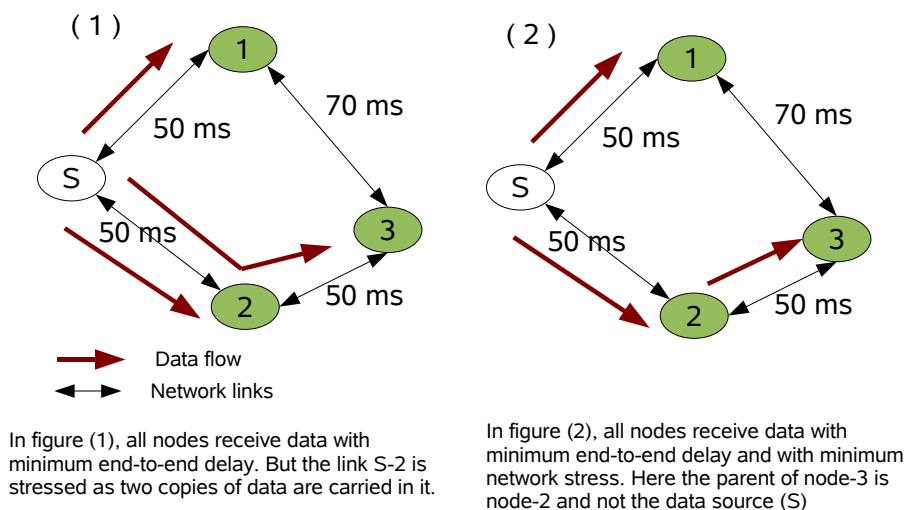


Figure 4: Stress to the underlying network

The solution to the problem should also have features discussed below.

Decentralised and Self-Organising The efficiency of data distribution needs to evolve (improve with time) by making incremental changes in a distributed manner.

In a centralised solution, all nodes report the collected metrics to a designated node. And this designated node constructs the distribution tree based on the reported metrics. This approach is not scalable to many number of participants. So, we require the solution to be distributed and self-organising over a period of time.

Adapting to network dynamics: In an overlay network, new nodes arrive and this can introduce alternate paths that were not available before. The solution should be able to adapt and construct a better distribution tree in such scenarios. Also, as a node departs, the delivery of data to its child nodes are disrupted. So, the solution need to have mechanisms that identifies potential alternate parent nodes in advance.

The problem can be concisely put as: Construct a adaptive (to network dynamics) data distribution tree that minimises the end-to-end delay in receiving the data and stress to the underlying network in a distributed manner.

2.3 Related Work

In this section, we present the related-works in the field of overlay-based data distribution. We also analyse the applicability of existing distributed minimum spanning tree algorithms.

NICE [12] is a distributed tree building protocol. In NICE, every member is assigned to a particular layer (hierarchical level). The members of a particular layer are partitioned into a set of clusters. Every cluster chooses a cluster leader which in turn joins the higher layer. An important function of the NICE protocol is to maintain the clustering and layering operations. It is specifically designed for low bandwidth data streams with large receiver sets. NICE protocol operates with a state space complexity of $O(\log N)$ (where N is the total number of participants).

Narada [11] is a protocol that adds multicast support to the end systems. Narada constructs a distribution tree using a two step process. As a first step, it constructs a mesh-network (richly connected graph) of participating members. In the second step, Narada constructs a spanning tree using well known routing algorithms. It has been specifically designed for small group members. In Narada, every participating node maintains state about all other participating nodes $O(N)$.

OMNI (Overlay Multicast Network Infrastructure) [13] is an overlay-based solution for networks deployed with the help of infrastructure nodes. It uses a two-tier infrastructure to implement large scale media-streaming applications. The OMNI infrastructure consists of a set of Multicast Service Nodes (MSNs - Infrastructure nodes) distributed in the network and is used for delivering content to the end hosts. The MSNs run a distributed protocol to organise themselves into a multicast data delivery overlay. In this solution each MSN has to maintain state about *degree* + $\log N$ other MSNs.

Our work is related to constructing an efficient data distribution tree. In this work,

we attempt to reduce the state space complexity and improve the efficiency of the data distribution tree. We also propose an idea of identifying potential parent nodes using a cooperative mechanism. Our proposed solution can construct an MST with minimal cumulative end-to-end delay in receiving the data and also minimises the network stress. Also, In our case, the state space complexity is independent of the number of participants ($O(x^2)$, where x is the degree).

Many algorithms have been proposed for building minimum spanning tree (MST) in distributed manner. A summary of such proposed solutions are presented in [16]. As our problem involves building efficient distribution tree, we evaluated their applicability in overlay networks. For the following reasons, the existing algorithms cannot be applied directly.

1. Distributed MST algorithms assume that nodes usually know how to distinguish their neighbours based on the incoming links. In Overlay networks, there is no concrete means to know who their neighbour is?
2. In the case of failure of one or more nodes, entire MST has to be recomputed. In overlay networks, with many number of participants, this is not desirable. As the tree is used for distributing data, during the time elapsed in recomputing MST, there will be loss of data.
3. The algorithms do not have mechanism that allow incremental evolution (building) of data distribution tree i.e, during the process of computing the MST, there should not be disruption in the flow of data.
4. The algorithms compute MST to reach minimal sum of weights (link costs). But our problem is to compute MST to minimise the cumulative end-to-end delay in receiving the data from the data source.

The solution that we propose in the thesis can construct MST by taking the above constraints into consideration.

2.4 Summary

We see that tree-push architecture can reduce the end-to-end delay in receiving the data and can minimise the network stress by using an efficient tree construction mechanism. The task of designing an efficient data distribution tree is the core problem. Also, the solution to be designed need to be decentralised and adapt to network dynamics.

3 Distributed Tree Routing

The initial bootstrapped position of the nodes are determined by the bootstrapping algorithm. This initial position – decided by the bootstrapping algorithm – may not be the optimal one. This problem can lead to high end-to-end delay in receiving the data and high network stress. So, we need efficient algorithms for continuous optimisation of the data distribution tree. Minimising stress in the underlying network and end-to-end delay need to be an important consideration in deciding the quality of such algorithms. The problem can be modeled as: Given a weighted graph with 'n' vertices and 'l' links, construct a minimum spanning tree (MST) with the vertex 'S' as the root. The solution for such problems require searching the graph, so as to build a minimum spanning tree. In the field of computer networks, it becomes a necessity that such search operation need to be distributed so as to scale for large number of network elements.

In this chapter, we propose a distributed algorithm that can build an MST in spite of the dynamics of the overlay network. The MST constructed by the algorithm is optimised for minimising the cumulative end-to-end delay in receiving the data.

3.1 Overview of the algorithm

The algorithm has three node re-positioning steps executed concurrently in each of the participating node. By re-positioning, we refer to the change of the parent node from which a node receives the data. These three mechanisms are further referred as *local correction-1*, *local correction-2* and *local correction-3* respectively. They are collectively further referred to as 'Distributed Tree Routing (DTR)'.

3.1.1 Local Correction-1

In this step, every node calculates the cost of the virtual link with each of its sibling nodes. The collected metrics are reported to their parent node. Also, every parent node calculates the link cost with each of its child nodes. These metrics are periodically collected and maintained in a table by the parent node. In Every T_0 seconds, the parent runs Dijkstra algorithm over the collected metrics to calculate the least cost path to reach all of its child nodes. If the newly calculated paths are different from the current paths, then a route update message is sent to all the child nodes. The route update message can initiate two kinds of actions (i) change of parent and (ii) addition of new child node.

Here, we explain using an example on how the *local correction-1* works, by taking link delay as weights. In figure 5, there are three child nodes that receive the data from a common parent. At the parent node, a path cost table is maintained as in table 1. Dijkstra algorithm is used to calculate the least cost path to reach all of the nodes (starting from the parent node). The new distribution path calculated using Dijkstra algorithm is updated to all the child nodes. The new distribution

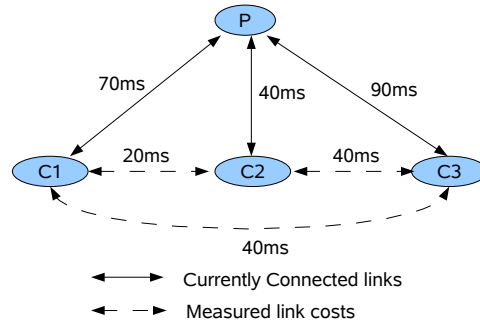


Figure 5: Sample Topology for local-correction-1 explanation

Table 1: Tabulating path costs

From	To	Cost(In ms)
P	C1	70
P	C2	40
P	C3	90
C1	C2	20
C1	C3	40
C2	C1	20
C2	C3	40
C3	C1	40
C3	C2	40

path shall look like, as in figure 6.

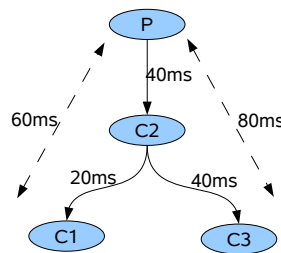


Figure 6: Topology after local-correction-1 execution

3.1.2 Local Correction-2

In this step, a participating node measures link costs with its child and grandchild nodes. Let 'X' be a node, 'Y' be the child node of 'X' and 'Z' be the child node of 'Y'. It checks for the two conditions 1 and 2. The condition 1 checks if the cost of reaching the node 'Z' directly from node the 'X' is less than the cost of reaching node 'Z' via node 'Y'. The condition 2 checks if the cost of reaching the node 'Y' via the node 'Z' is less than or equal to the cost of reaching node 'Y' directly from

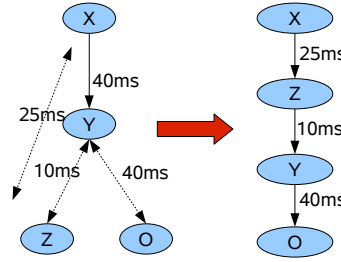


Figure 7: Sample Topology for local-correction-2 explanation

the node 'X'.

$$COST(X - Z) < (COST(X - Y) + COST(Y - Z)) \quad (1)$$

$$COST(X - Z) + COST(Z - Y) \leq COST(X - Y) \quad (2)$$

If both the conditions are satisfied, then the position of 'Y' and 'Z' are changed in the data distribution tree. An example showing how the topology gets transformed is given in the figure 7.

3.1.3 Local Correction-3

In this step, similar to the local-correction-2, a node measures the link costs with its child and grandchild nodes. Let 'X' be a node, 'Y' be the child node of 'X' and 'Z' be the child node of 'Y'. It checks for the two conditions 3 and 4. The condition 3 checks if the cost of reaching the node 'Z' directly from node the 'X' is less than the cost of reaching node 'Z' via node 'Y'. The condition 4 checks if the cost of reaching the node 'Y' via the node 'Z' is greater than the cost of reaching node 'Y' directly from the node 'X'.

$$COST(X - Z) < (COST(X - Y) + COST(Y - Z)) \quad (3)$$

$$COST(X - Z) + COST(Z - Y) > COST(X - Y) \quad (4)$$

If both the conditions are satisfied, then the position of 'Z' is changed to become a child node of 'X'. An example showing how the topology gets transformed is given in the figure 8.

3.1.4 States maintained by the nodes

For the above repositioning mechanisms to work, each node needs to maintain state information about a set of nodes. In a tree structure with a constant degree of 'x',

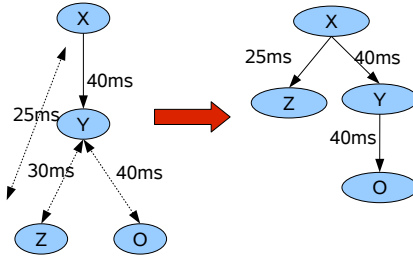


Figure 8: Sample Topology for local-correction-3 explanation

each node need to maintain state about

$$x(\text{children}) + x \times x(\text{grandchildren}) + (x - 1)(\text{siblings}) + 1(\text{parent}) + 1(\text{grandparent}) \tag{5}$$

$$\text{Consolidating}(5) = x^2 + 2x + 1 \tag{6}$$

So, the state space complexity of the algorithm is proportional to square of the degree of the tree ($O(x^2)$, where x is the degree). In a graph, with 'n' nodes, every participating node maintains only its local information and amount of information maintained is independent of the number of participants (n). This is an important feature that makes the algorithm scale for distributed systems. The Figure 9 depicts the state space complexity for a participating node 'X' in a distribution tree with a constant degree of 2.

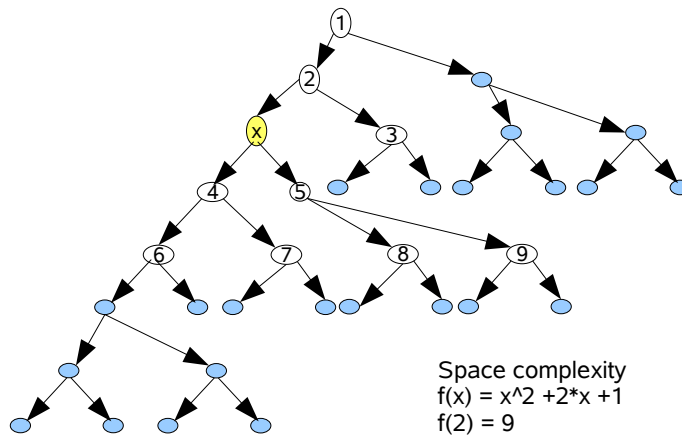
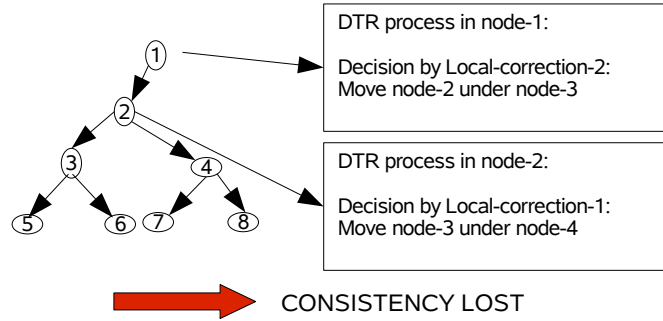


Figure 9: State complexity

3.2 Locks for DTR execution

Every participating node except the data source, executes the same DTR process, so it can be classified as a form of semi-uniform distributed algorithm. In the DTR process, *local correction-1* can move the nodes down from their current hierarchy and

the *local-correction-2* and *3* can move the nodes up from their current hierarchy. So, we see two forces, one trying to move a node *UP* and other trying to move *DOWN* (in terms of hop count). These two operations need to be mutually excluded to maintain consistency and success of the DTR process. In figure 10, we illustrate by example where consistency would be broken.



What to do: Ensure Mutual Exclusion of conflicting process

Figure 10: Concurrency issues in DTR

Every solution in the field of distributed systems has to ensure consistency due to concurrent operations. In our case, the DTR process needs to have the required mechanism that can mutually exclude operations on a sub-tree of the main distribution tree. Here, we ensure consistency by using locks in each node and the locks are acquired by messaging between nodes. The central point is that when the DTR process in a parent node wants to perform *local-correction-2*, it needs to acquire the lock of its immediate child node whose position is to be changed. This prevents that child node from doing any executions related to the DTR process. In the figures (11), we explain how the locking mechanism mutually exclude operations on a sub-tree region.

With respect to the locking mechanism, the node can be in one of the three states. The set of events that are possible for the node with respect to the locking mechanism are listed in the event set.

States = {*AVAILABLE*, *LOCKED_BY_SELF*, *LOCKED_BY_PARENT* }

Events = {*LOCK_ACQ*, *LOCK_OK*, *LOCK_NOT_OK*, *ROUTE_UPDATE*, *LOCK_TIMEOUT*}

We look at two sample scenarios (figure 11) where the lock mechanism prevents the loss of consistency for the problem represented in figure 10. There are many other possible scenarios where the lock prevents the loss of consistency, the figure 10 presents two of such scenarios.

In distributed systems, the locking mechanism also need to consider packet losses. Here we explain a sample scenario where the locking mechanism helps to maintain consistency in an operating environment with packet losses. Let us consider a scenario where a node 'X' sends *LOCK_ACQ* message to its child node 'Y'. And the node

'Y' sends *LOCK_OK* message to its parent node 'X'. If this message (*LOCK_OK*) is lost, then node 'X' does not make any repositioning decision related to node 'Y'. But the lock of node 'Y' need to be released. To recover from such scenarios, every node has a timer event associated with its lock. When a node's lock is acquired, a timeout event (*LOCK_TIMEOUT*) is scheduled. After the expiry of the timer (t seconds), the node 'Y' automatically releases the lock. The value chosen for 't' need to be chosen depending on the path delays observed in the operating environment. A large value for 't' means more consistency and stability, but the optimising process of the distribution tree would be slow.

In scenarios where a node involved in a locking transaction departs the overlay, the locking mechanism fails.

3.3 An Example

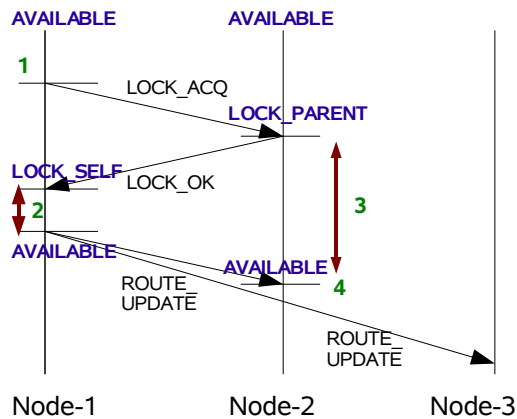
The figure 12 represents a graph with 15 nodes and 18 edges. For subsequent analysis, we shall be using this graph as reference.

We use four different scenarios to verify the solution. The first scenario is explained in detail using figures that depict the detailed node transformations. For other three scenarios, the convergence is shown using summary charts. In the scenario-1, we put the source at node-1. The other nodes join (get active) the overlay to receive the data. We simulate a condition where farther nodes join the overlay sooner than the nodes closer to the source. In our sample topology, node-15 joins the overlay first, then node-14 and so on. The DTR process in each node repositions the nodes in the overlay to reach a MST. The DTR process repositions nodes depending on the metrics reported by the nodes about which it maintains state. There is an interval for which DTR process collects the required metrics and then makes the repositioning decisions. In our example, we use this interval as 10 seconds, we refer this interval as $dtr_{timeout}$ further in this document. The algorithm used for bootstrapping will be presented in chapter 5.

In this scenario, the distribution tree converges at time 73.20 seconds. The detailed view of how the convergence happens in a distributed fashion is illustrated using figures 13 and 14. The figure 14 illustrates the timeline summary of the node arrivals and the repositioning decisions. From figure 14, we see that the repositioning algorithms converge to an MST. The table in figure 14 presents the minimum possible distance to reach all nodes from the data source (node-1). The algorithm positions the nodes in a way that they get the data with minimum possible delay and also with minimum stress to the underlying network. From this example, we see that the algorithm converges to an MST by iteratively repositioning the participating nodes. The algorithm is able to handle to new node arrivals and the resulting change in the topology (arrival of new links and nodes).

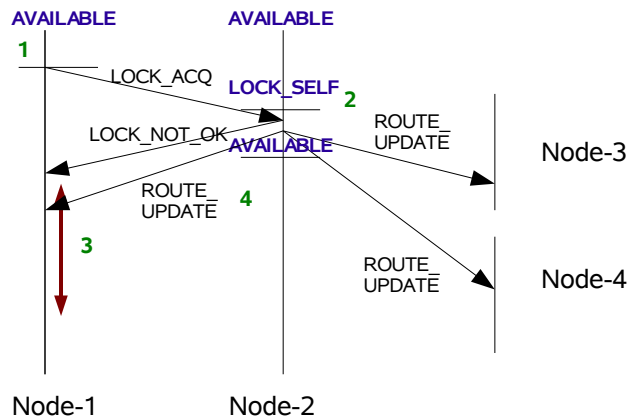
In scenario-2, the source is at the location node-1 and a preferable node arrival pattern is used. The nodes closer to the source join the overlay sooner than the nodes farther away. In this type of scenario, We expect the number of DTR transformations

A possible scenario: Node-1 acts before Node-2



- 1 -> DTR process in node-1 detects that node-3 needs to be moved under node-1 and node-2 to be moved under node-3.
- 2 -> In this period, the DTR process checks if the decision on repositioning node-2 and node-3 still holds
- 3 -> In this period, node-2 puts its DTR process in hold
- 4 -> Change of parent resets the lock acquired by the parent node

Another possible scenario: Node-2 acts before Node-1



- 1 -> DTR process in node-1 detects that node-3 needs to be moved under node-1 and node-2 to be moved under node-3.
 - 2 -> DTR process in node-2 decides to move node-3 under node-4. (acquires its lock)
 - 3 -> DTR process defers the re-positioning decision till its next timeout.
 - 4 -> It informs node-1 that, node-3 need to be removed from its grandchild list
- Important Design Decision: The DTR process attempts repositioning mechanisms once in say 'x' seconds. Let the time taken for ROUTE_UPDATE message (number 4) to reach from node-2 to node-1 be 'y'. Then, for successful operation 'x' need to be at-least greater then 'y'.

Figure 11: State transitions of DTR lock process

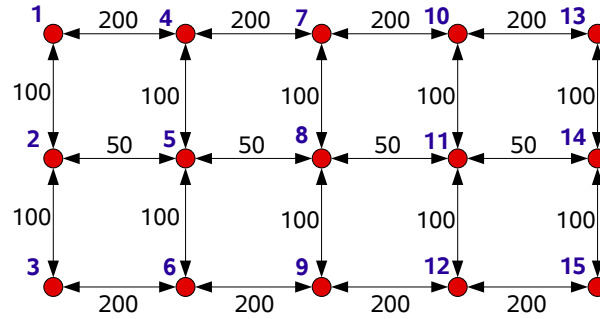


Figure 12: A sample network graph used for evaluation

to be less, as the initial bootstrapped location is itself a near optimal one. The timeline that summarises scenario-2 is represented in figure 15. The time taken for the network to converge is 60.15 seconds (one iteration less than scenario-1).

In scenario-3, the source is at the location node-8 and the node arrival pattern is same as the one used for scenario-1. The timeline that summarises scenario-3 is represented in figure 15. The time taken for the network to converge is 60.15 seconds.

In the scenario-4, the source is at the location node-8 and the node arrival pattern is similar to the one used for scenario-2. The timeline that summarises scenario-4 is represented in figure 15. The time taken for the network to converge is 68.30 seconds.

The four scenarios that were presented above were chosen so as to illustrate their relevance to the two metrics - Number of transformations required (proportional to the time taken to converge) and end-to-end delay in receiving the media. In scenario-2 and 4 (favourable arrival pattern), the number of iterations required were less than the scenario-1 and 3. Also, the position of the source significantly influences the end-to-end delay in receiving the data. In scenario-3 and 4, the cumulative end-to-end delay in receiving the data is less than the other two scenarios.

3.4 Metrics for evaluation

In case of large topologies, it may not be clearly visible on how efficient the algorithm is, in minimising the end-to-end delay and stress to the underlying network. The mathematical form of the problem of minimising the end-to-end delay and the network stress can be represented as,

S = Source (data source)

N = Set of all participating nodes (other than the source)

N_i = A participating node belonging to the set N

For our example: DTR process in each node times out once in 10s and performs the required re-positioning. So, we present the graph for every 10 seconds. The weights given for each links represent the cost to reach the node from the source.

-  Local Correction-2
-  Local Correction-3
-  Local Correction-1

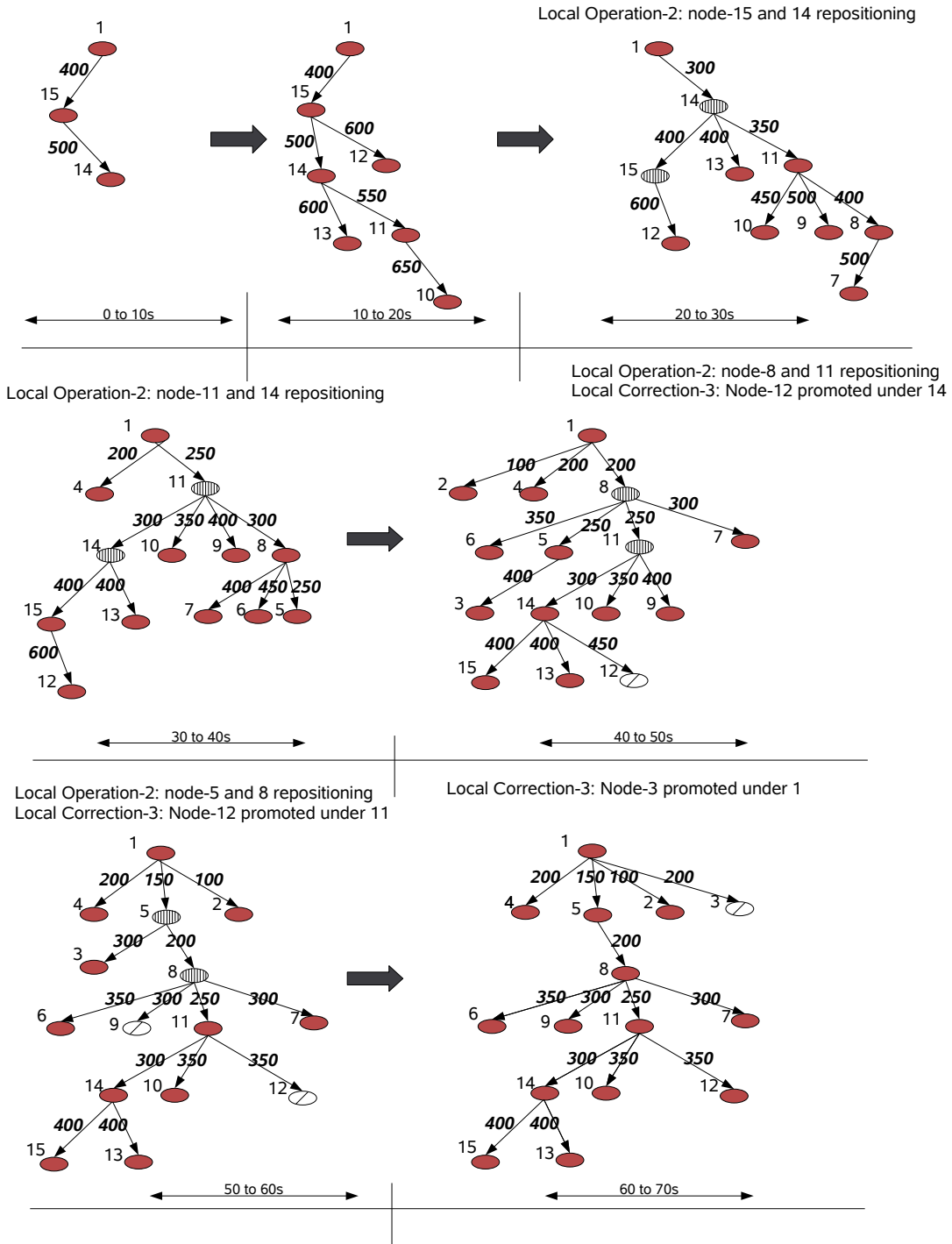


Figure 13: Scenario-1: Transformation steps

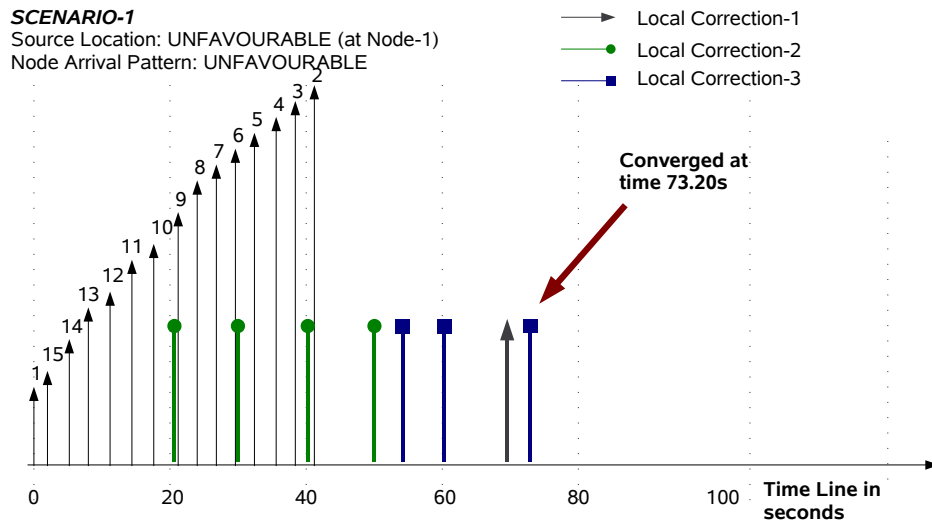
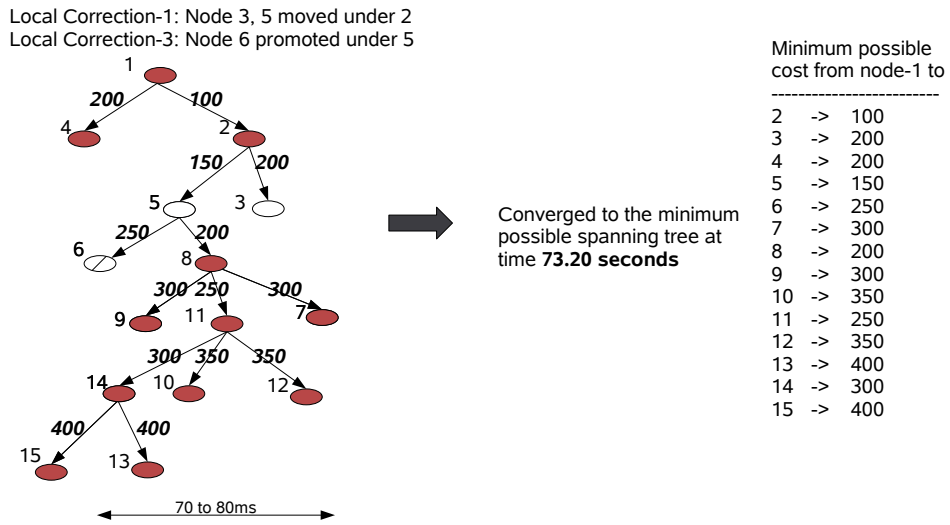


Figure 14: Scenario-1: Transformation steps

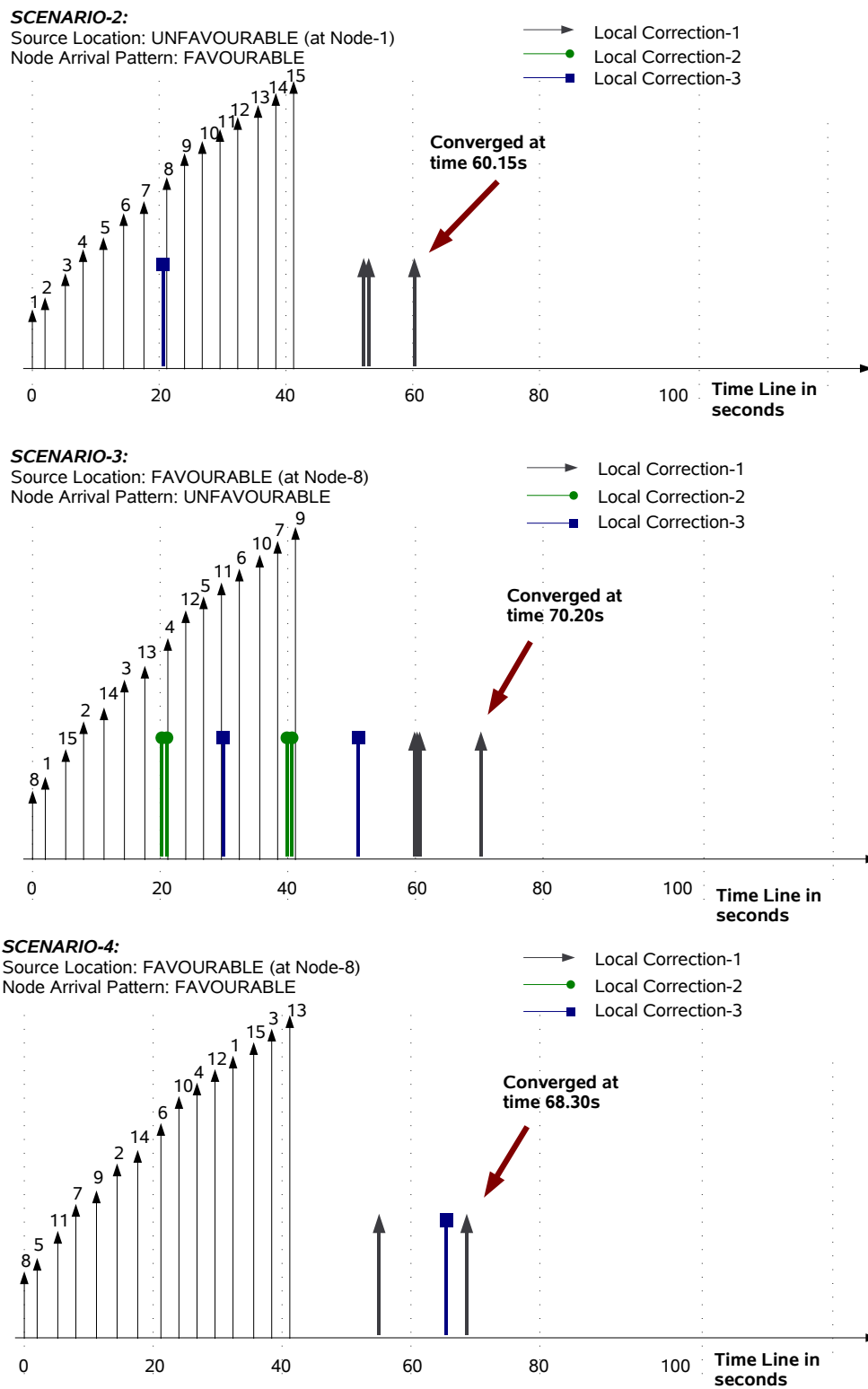


Figure 15: Scenario 2, 3 and 4: Summary

P^{N_i} = Parent of the participating node N_i

$L_{P^{N_i}N_i}$ = Path connecting (set of links) P^{N_i} and N_i

$C_{P^{N_i}N_i}$ = Cost of the path $L_{P^{N_i}N_i}$

$\sum_{i \in N} C_{P^{N_i}N_i}$ = Cost of Data Distribution Tree (C_{tree})

C_{StoN_i} = Cost of the data in reaching from source (S) to N_i

$\sum_{i \in N} C_{StoN_i}$ = Cost of the Data in reaching all participating nodes (C_{data})

The goal of the algorithm is to continuously minimise C_{data} and C_{tree} , in spite of new node arrivals and the resulting change in topology. At scenarios where the requirements of C_{data} and C_{tree} contradict each other, DTR will give priority to C_{data} . The figure 16, represents a sample topology and the possible data distribution tree that can be constructed (considering 'S' as data source). From the figure 16, one can differentiate and understand the contradicting requirements of C_{data} and C_{tree} .

To analyse the working of the solution in large topologies, we periodically collect and plot the metrics of C_{tree} and C_{data} .

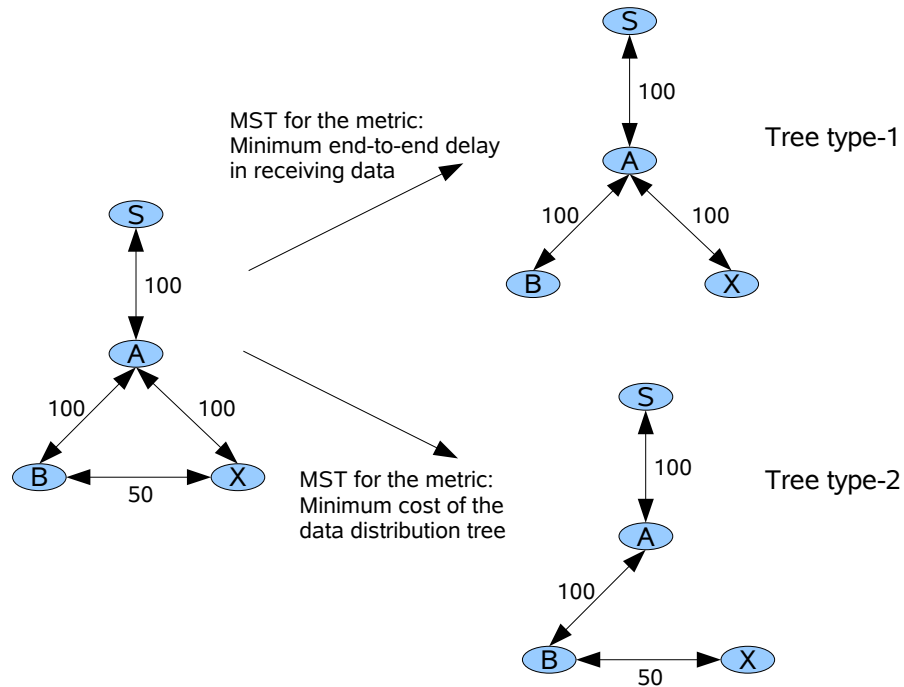
3.5 Evaluating DTR performance using simulation

In this section, we evaluate the performance of DTR using simulation. Two sample networks with 1020 and 100 nodes respectively are chosen for evaluation. The topologies are generated using the GT-ITM network topology generator[15]. The results of the simulation are discussed in this section.

Joining Mechanism We present two scenarios with 100 (scenario-1) and 1020 (scenario-2) participating nodes. In both scenarios, a new node arrives for every 3 seconds. So, at time 300 (scenario-1) and 3060 (scenario-2) seconds, all node arrivals are complete. For a node ($Node_{new}$) to join the overlay, it has to contact the bootstrap server. The bootstrap server chooses 5 random addresses from the existing overlay participants list and returns it to the $Node_{new}$. $Node_{new}$ sends *Join Request* all the 5 existing overlay members. The node whose reply reaches first is taken as the parent node. This mechanism helps the $Node_{new}$ to choose a parent (among the 5 node addresses received) that has less end-to-end delay.

Simulation Results The cost of the data distribution tree (C_{data} and C_{tree}) are calculated for every 10 seconds. The collected metrics are plotted as graphs and presented in the figures 17 and 18. The graph shows the quality of the data

Understanding the metrics: C(data) and C(tree)



(Let the cost of the links be represented in milliseconds)

For tree type-1:

- Node 'X' receives the data in 200 ms
- \sum (E2E delay in receiving media) = cost(SA) + cost(SB) + cost(SC) = **500 ms**
= (C(data))
- \sum (delay in receiving the media from parent) = cost(SA) + cost(AB) + cost(AX)
= **300 ms** = cost of the data distribution tree = C(tree)

For tree type-2:

- (i) Node 'X' receives the data in 250 milliseconds
- (ii) \sum (E2E delay in receiving media) = cost(SA) + cost(SB) + cost(SC) = **550 ms**
= (C(data))
- (iii) \sum (delay in receiving the media from parent) = cost(SA) + cost(AB) + cost(BX)
= **250 ms** = cost of the data distribution tree = C(tree)

From the above example, we can see that C(data) and C(tree) can be contradictory. In such scenario, DTR gives higher priority to C(data).

Figure 16: Understanding C_{tree} and C_{data}

distribution path with and without DTR. The graph shows that the performance is continuously optimised in spite of the change in network dynamics (new node arrivals) and the performance improvement due to the use of DTR is also significant (figure 17 and 18).

3.6 Analysing the efficiency

From the evaluations using simulation, we see that the algorithm can adapt to the arrival of new nodes and can converge to an MST. So, we see that the algorithm works in constructing an MST. The time taken to converge and the ability to scale in a distributed way are taken as parameters to decide the efficiency of the algorithm. The state space complexity of the system is independent of the number of nodes, so this shows that the algorithm can scale and evolve in a distributed way. Regarding the time taken to converge, it depends on the following parameters.

Arrival pattern and bootstrapping mechanism: From Scenario 1 (figure 14) and 2 (figure 15), we see that the number of transformation required to converge is less, if the arrival pattern of the nodes are favourable. The bootstrapping mechanism has an important part to play in deciding the initial position of the node. The better the initial position, the quicker is the convergence.

Number of nodes and network links: The number of participants, the density of the links interconnecting the nodes and the weights of the network links form an important factor in the time taken to converge.

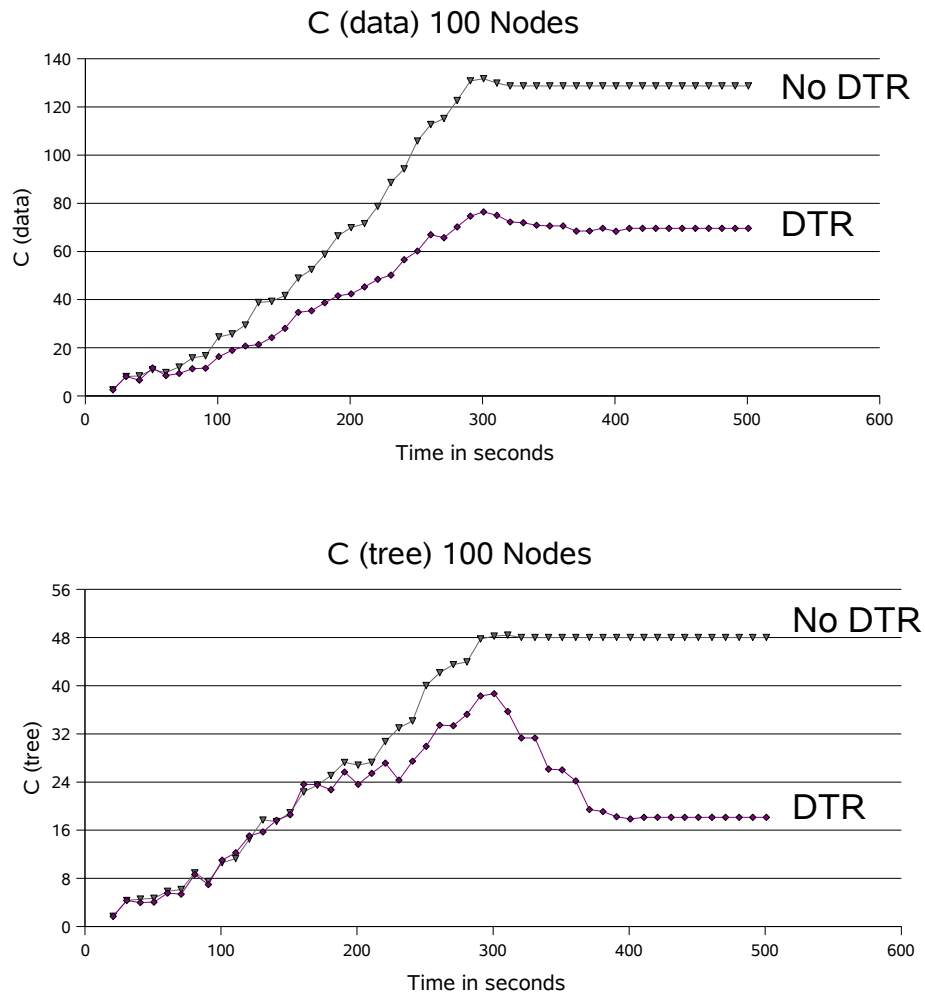
dtr_{timeout}: The DTR process in each node executes the local correction algorithms once in every *dtr_{timeout}* seconds. This parameter is a design decision that can be modified depending on the operating environment of the algorithm. In our example scenarios, we used a value of 10 seconds. Every node maintains data about 'x' other nodes. The data about 'x' other nodes is periodically (say every 'y' seconds) refreshed. The *dtr_{timeout}* value depends on the value chosen for 'y'.

Packet loss percentage: Loss of messages that carry route updates (repositioning decisions) can lead to increased convergence time. Other engineering decisions such as number of retransmissions, timeout to detect a lost 'route update packet' etc. also have a role to play in deciding the convergence time.

3.6.1 Operation of DTR in degree constrained environment

To converge as an MST, the nodes must accept the repositioning decisions made by the DTR processes. From scenario-1 (figure 14) execution, we see that at some point in the middle of transition to an MST, a particular node was having four child

Simulation results (for 100 nodes)



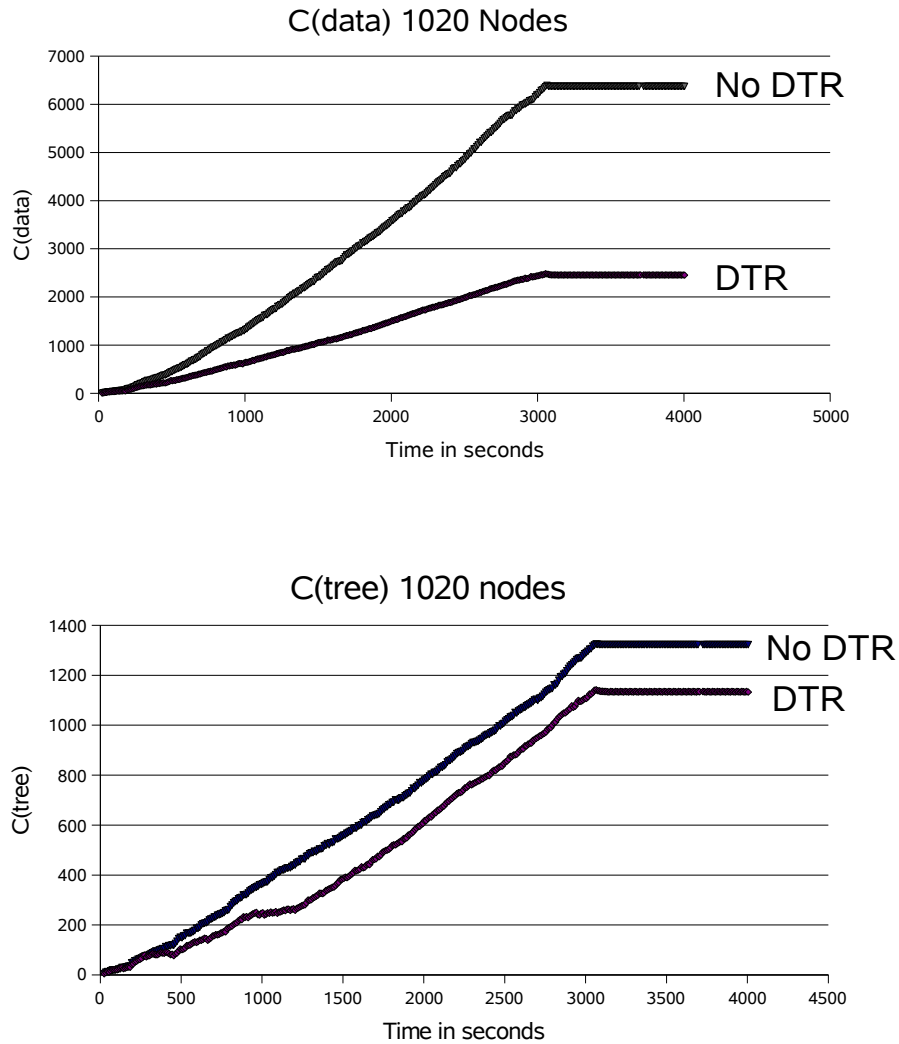
Improvement in using DTR:

C (data) = With DTR, the performance is 84.94 % better

C (tree) = Without DTR, the performance is 164.57 % better

Figure 17: DTR performance (100 Nodes)

Simulation results (for 1020 nodes)



Improvement in using DTR:

C (data) = With DTR, the performance is 159.74 % better

C (tree) = Without DTR, the performance is 16.78 % better

Figure 18: DTR performance (1020 Nodes)

nodes connected to it. If a node cannot perform some transformation for reasons like lack of bandwidth (degree restrictions) or policy decisions, the algorithm cannot converge as an MST (minimal C_{data}). But by including these capacity restrictions into the algorithm, we can still form a distribution tree with reduced C_{data} and C_{tree}

To show the operation of DTR in a network with capacity restrictions, we take the figure 12 as the reference network and all the nodes are considered to have a constant degree restriction of two. The DTR algorithms are modified to consider the degree restrictions of the nodes before making repositioning decisions. The initial position of the nodes are chosen so as to represent a worst case condition. Figure 19 shows the transformation that are possible even with degree restrictions. From figure 19, we see that even with a worst initial bootstrapped position and a strict degree constraints, five repositioning (for a overlay with 15 participants) are made to reduce the network stress and E2E delay in data path.

3.7 Applications for this algorithm




One important motivation behind the design of this algorithm is to find a solution that can build an MST in a distributed way, taking into account of new node arrivals and the resulting formation of better paths to deliver data (that form as a result of new node arrivals). In this section, we give examples where this algorithm can have application. Examples of overlay based data distributions are weather updates, stock tickers, media streaming etc. All these applications require a bootstrapping mechanism, a distribution tree construction mechanism, pro-active repair (or tree optimisation) and reactive repair mechanisms (loss of a participating node). Our distributed algorithm proposes effective solution for the tree construction and the pro-active tree optimisation problems. So, all applications that require tree construction and pro-active optimisation mechanisms can use our proposed approach.

3.8 Conclusion

In this chapter, we have proposed a solution that can construct a data distribution tree in a distributed way. The state space complexity is independent of the number of participants in the session ($O(x^2)$, where 'x' is the degree) and the distributed nature of the solution can make it scale to large number of participants.

When the nodes do not have enough capacity to perform the DTR transformations, then the distribution tree does not converge as a completely optimal one. In such scenarios, as a next step, we shall see how to make the distribution near optimal using CORIP.

For our example: DTR process in each node times out once in 10s and performs the required re-positioning. So, we present the graph for every 10 seconds. The initial position of the nodes are chosen so as to represent a worst case condition.

-  → Local Correction-2
-  → Local Correction-3
-  → Local Correction-1

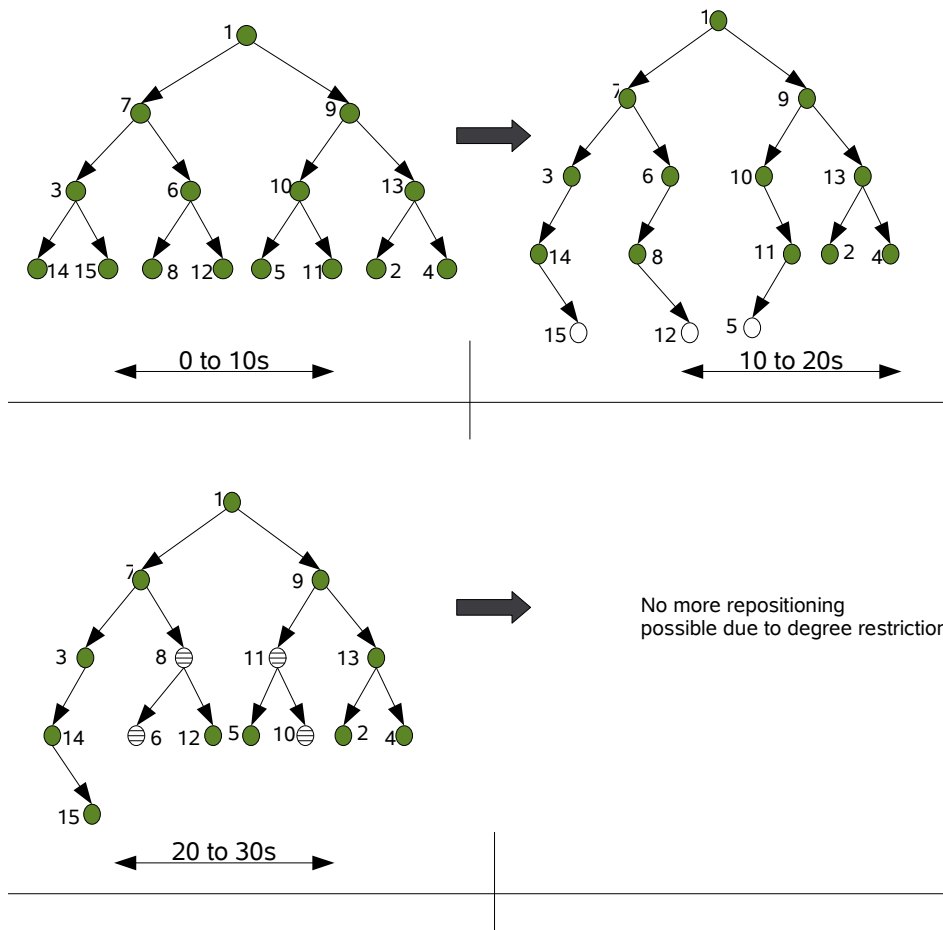


Figure 19: Degree restricted scenario

4 Co-Operative Resource Identification Protocol

Co-Operative Resource Identification Protocol (CORIP) is a protocol, using which a node participating in a P2P-overlay session – with a tree based data distribution – can identify resources. In the context of P2P-Overlay session, resources represent capacity to accommodate a new child node. CORIP has been designed by taking the following requirements into consideration.

1. Not all participating nodes may have enough capacity to support many child nodes.
2. A node that joins the overlay session may not always be bootstrapped to the best possible location.
3. To supplement DTR in sessions with many degree constrained nodes.
4. To construct/maintain a list of potential nodes that can be used by reactive repair mechanisms (in case of loss of parent node).

Using CORIP, the participating nodes identify resources co-operatively. The identified resources are shared among the participating nodes. It is a distributed approach, where the task of identifying resources are shared among the participating nodes.

4.1 Operation of CORIP

Using CORIP, a participating node can identify resources that are distributed throughout the distribution tree. CORIP is a combination of two algorithms: R_{up} and R_{down} . Using R_{up} , the participating nodes operating with a hop count value of x (from the source) can know about the resources with hop count value of 'less than or equal to x '. Using R_{down} , the participating nodes operating with a hop count value of x (from the source) can know about the resources with the hop count value of 'greater than x '.

4.1.1 Operation of R_{up}

To explain the working of R_{up} , we start by establishing the definition of the state variables involved and the message types used for communicating between nodes.

State variables involved: The following are the variables that are read/updated by the algorithm.

n = Represents a node that is participating in the session

S_{nodes}^n = Set of sibling nodes for the node n

P_n = Parent of the node n

C_n = Set of child nodes for the node n

$TI1_{nodes}^n$ = Set of nodes that are to be identified by node n

$TI2_{nodes}^n$ = Set of nodes that are to be identified by C_n

IR_{nodes}^n = Set of nodes that are identified as resource

IR_{nodes}^n : This set is built not by n alone, but in co-operation with all ancestors of node n . $TI1_{nodes}^n$ is a set of nodes that is passed on to the node n by its parent node (P_n). $TI2_{nodes}^n$ is a set of nodes that is passed on to the C_n by the node n .

Message classification: R_{up} uses three types of messages. These messages operate (read/update) over the variables discussed above.

Message Types = QUERY, QUERY_RESULT, SHARE

QUERY : Using this message, one node (x) queries another node (y) to know about its (y 's) capabilities and its child node information.

QUERY_RESULT : Using this message, a node informs the querier about its capability and its child node information.

SHARE : Using this message, a node shares with its child nodes two types of information. Type-1 refers to set of nodes that are identified as resources. Type-2 refers to set of nodes that are yet to be identified as resource.

Algorithm: Every node participating in the session (other than source) has four events. They are

1. Q_{event} : Arrival of the message *QUERY*
2. QR_{event} : Arrival of the message *QUERY_RESULT*
3. S_{event} : Arrival of the message *SHARE*
4. T_{event} : Periodic interval (Timeout event) to send *QUERY* and *SHARE* messages

Here, we explain how the events are handled and how the state variables are affected by the events.

Algorithm 1 Handle Event : Q_{event}

```

queryReply = {}
n ← currentNode
if n has extra degree (bandwidth) then
    queryReply+ = n
end if
queryReply+ = Cn
send QRevent(queryReply)

```

Algorithm 2 Handle Event : QR_{event}

```

n ← currentNode
Qnode ← getEventSourceDetails(QRevent)
ToBeI ← getToBeIdentifiedNodes(QRevent)
if Qnode has extra degree (bandwidth) then
    IRnnodes+ = Qnode
end if
TI2nnodes+ = ToBeI

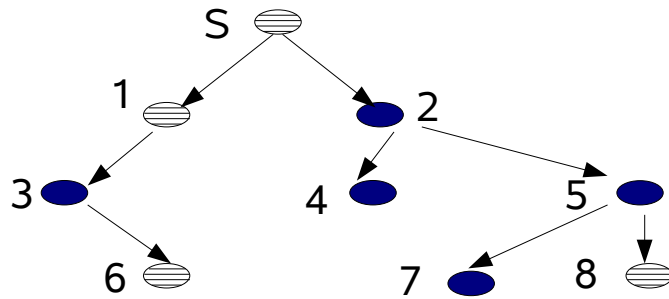
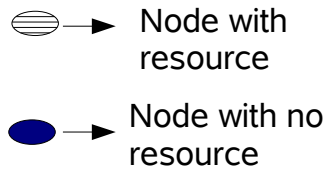
```

The event handlers (Functions 1, 2, 3, 4) periodically update the state variables using the defined messages. The nodes maintained in the set IR_{nodes}^n are the products of the R_{up} algorithm. Depending on the requirements of the application, the T_{event} can be split into two events, one as a trigger for sending $QUERY$ message and the other for sending $SHARE$ message. Also, one more timeout event can be added to the protocol to cleanup obsolete records in the state variables. The value for the T_{event} timer, need to be decided based on the requirements of the application using the protocol.

The figure 20 and 21 explain the working of R_{up} , using a sample topology. In this Example, the execution starts with node 1 and 2 querying each other and updating their state variables. Then, they prepare the $SHARE$ message and pass it to their child nodes. After node 3, 4 and 5 complete their queries, they update the state variables and send the $SHARE$ message to its child nodes. At the last hop, the leaf nodes contain the list of nodes that have resources to contribute (i.e extra degree available). This example explains the working using sequential flow of execution, just to have better visualisation for the readers. But in a real-world scenario, every node shall be updating the state variables concurrently. From figure 20 and 21, we see that all nodes successfully identify the resources that are operating with a hop count value that is lower than their hop count value.

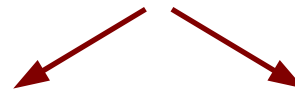
Scaling to large topologies: In sessions with many participants, the length of the $SHARE$ message grows exponentially. The $SHARE$ message is a container holding $TI2_{nodes}^n$ (set of 'to be identified nodes') and IR_{nodes}^n (identified resources). And the $SHARE$ message is sent by nodes for every 'x' seconds. To make the solution scale to large number of participants, the size of the $SHARE$ message

A Sample Data Distribution Tree



(1)	
n	= 1
S^1_{nodes}	= { 2 }
P^1	= S
C^1	= { 3 }
$TI1^1_{nodes}$	= { }
$TI2^1_{nodes}$	= { 4, 5 }
IR^1_{nodes}	= { S }

(2)	
n	= 2
S^2_{nodes}	= { 1 }
P^2	= S
C^2	= { 4, 5 }
$TI1^2_{nodes}$	= { }
$TI2^2_{nodes}$	= { 3 }
IR^2_{nodes}	= { S, 1 }



(3)	
n	= 3
S^3_{nodes}	= { }
P^3	= 1
C^3	= { 6 }
$TI1^3_{nodes}$	= { 4, 5 }
$TI2^3_{nodes}$	= { 7, 8 }
IR^3_{nodes}	= { S, 1 }

(4)	
n	= 4
S^4_{nodes}	= { 5 }
P^4	= 2
C^4	= { }
$TI1^4_{nodes}$	= { 3 }
$TI2^4_{nodes}$	= { 6 }
IR^4_{nodes}	= { S, 1 }

(5)	
n	= 5
S^5_{nodes}	= { 4 }
P^5	= 2
C^5	= { 7, 8 }
$TI1^5_{nodes}$	= { 3 }
$TI2^5_{nodes}$	= { 6 }
IR^5_{nodes}	= { S, 1 }

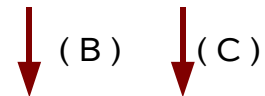
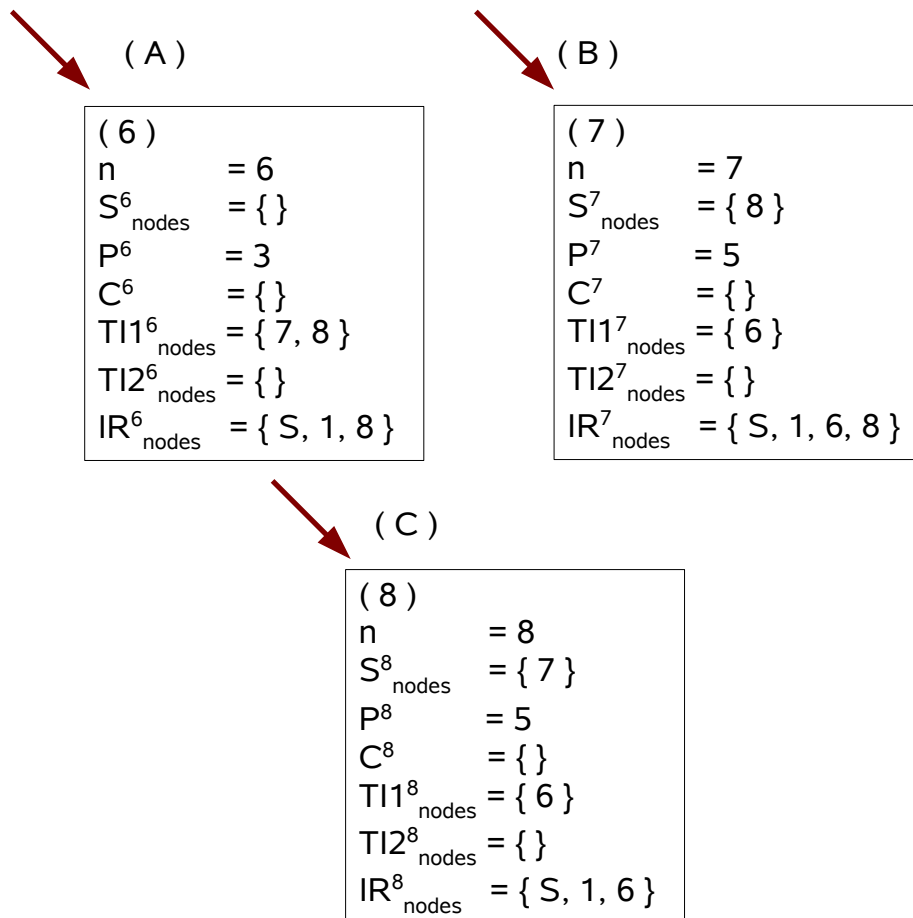


Figure 20: Working of R_{up}



Using R_{up} , a node operating at hop count 'x', is able to identify all resource nodes that are operating at hop count $\leq x$ (less than or equal to 'x').

The Result of the algorithm is the list of nodes stored in the set IR_{nodes}^n .

Figure 21: Working of R_{up}

Algorithm 3 Handle Event : S_{event}

```

 $n \leftarrow currentNode$ 
 $I_{nodes} \leftarrow getIdentifiedNodes(S_{event})$ 
 $ToBe_I \leftarrow getToBeIdentifiedNodes(S_{event})$ 
 $IR_{nodes}^n + = I_{nodes}$ 
 $TI1_{nodes}^n + = ToBe_I$ 

```

Algorithm 4 Handle Event : T_{event}

```

 $n \leftarrow currentNode$ 
 $I_{tmp} \leftarrow S_{nodes}^n + TI1_{nodes}^n$ 
 $S_{msg} \leftarrow IR_{nodes}^n + TI2_{nodes}^n$ 
for all every node in  $I_{tmp}$  do
  send  $Q_{event}$ 
end for
for all every node in  $C_n$  do
  send  $S_{event}$ 
end for

```

need to be either fixed or made independent of the number of participants. This can be achieved by maintaining a threshold on the maximum number of nodes that can be packed into the *SHARE* message, from the $TI2_{nodes}^n$ and IR_{nodes}^n . In scenarios, where the number of nodes available in the $TI2_{nodes}^n$ and IR_{nodes}^n exceeds the threshold, the required number of nodes can be chosen by random selection, based on the capability reported, based on history of the session etc. By using such approaches, we shall be able to scale to large number of participants.

4.1.2 Operation of R_{down}

R_{down} can be considered as an extension of R_{up} . To explain the operation of R_{down} , we start by establishing the definition of the state variables and the messages that are additionally needed.

State variables involved: The R_{down} uses all the state variables that are used by the R_{up} . In addition, it uses the variable $IR3_{nodes}^n$. Also, it considers IR_{nodes}^n as a combination of two sets $IR1_{nodes}^n$ and $IR2_{nodes}^n$.

$$IR_{nodes}^n = IR1_{nodes}^n + IR2_{nodes}^n$$

$IR1_{nodes}^n$ = Set of nodes identified as resources by the ancestors.

$IR2_{nodes}^n$ = Set of nodes identified as resources by the current node.

$IR3_{nodes}^n$ = Set of nodes identified as resources by the child nodes.

Message classification: It uses all the messages that are used by the R_{up} . In addition, it requires one more message.

$PASS_IR$: Using this message, a node informs its parent node about the identified resources. Here by identified resources, we refer to nodes that are having a hop count value higher than the parent node.

Algorithm: We add two more events to the events listed in R_{up} . They are

1. PR_{event} : Arrival of the message $PASS_IR$.
2. TP_{event} : Periodic interval (Timeout event) to send $PASS_IR$ message.

Here, we explain how the events are handled and how the state variables are affected by the events.

Algorithm 5 Event Handler: TP_{event}

```

n ← currentNode
PRmsg ← IR2nnodes + IR3nnodes
if n is resource then
    PRmsg ← PRmsg + n
end if
if n NOT data source then
    send PRevent(PRmsg) to Pn
end if

```

Algorithm 6 Event Handler: PR_{event}

```

n ← currentNode
Itmpnodes = getIRNodes(PRevent)
IR3nnodes + = Itmpnodes

```

From the function 6, we see that every node is able to receive the list of identified resources and from the function 5, we see that this information is recursively spread till it reaches the data source.

The figure 22 explains the working of R_{down} , using a sample topology. In this Example, the execution starts at node 6, 7 and 8. They send PR_{event} to their parent node. The parent node in turn sends PR_{event} to its parent node. This recursion stops, when the data source is reached. From the figure 22, we see that any node operating with a hop count (from data source) value of ' x ' is able to know about resources that are having hop count value ' $> x$ '. The exception to the above rule is, if a node is operating with a hop count value of ' x ' and is a leaf node, then it cannot know about the resources that are having hop count value ' $> x$ '.

In this example, we start explaining the working from the stage when the leaf nodes have already identified the resources (using R_{up}). This is done to help the task of explaining the solution

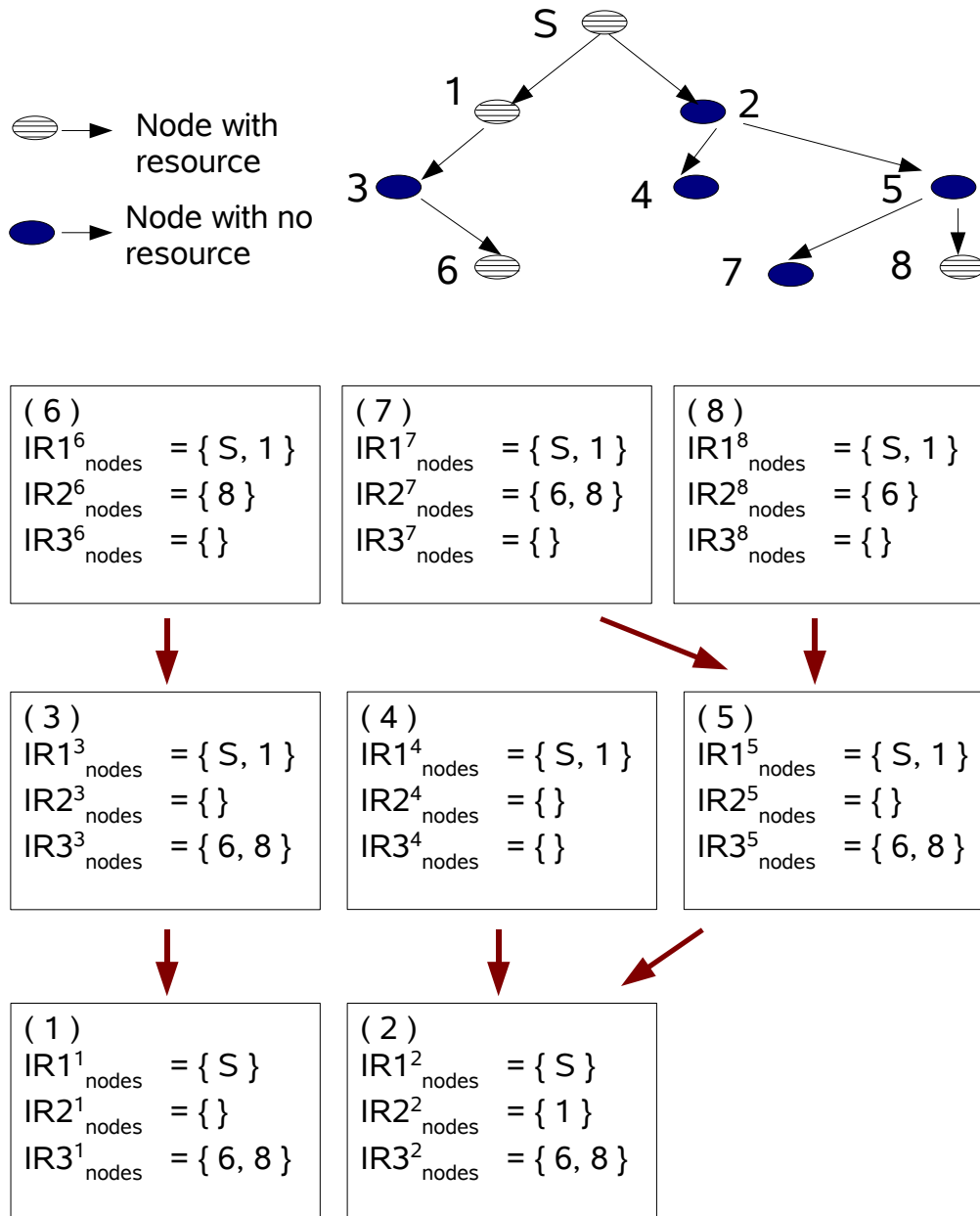


Figure 22: Working of R_{down}

Scaling to large topologies: In case of sessions with many number of participants, the length of the *PASS_IR* message grows exponentially. So, to scale for large number of participants, the length of the *PASS_IR* message need to be maintained within a threshold. The approach that was discussed for limiting the length of *SHARE* message, is applicable for the *PASS_IR*, too.

4.2 Applications for the protocol

4.2.1 Pro-Active Optimisation

One important requirement for the effective operation of P2P-Overlay sessions is its ability to self-organise the data distribution tree over a period of time. By self-organising, we refer to reducing the end-to-end delay in receiving the data and reducing the stress to the underlying network. DTR is one such mechanism which attempts to self-organise the data distribution tree using node repositioning techniques. As discussed earlier, DTR has its limitations when operating in a session with many degree constrained nodes (limited outbound bandwidth). So, participating nodes need to have a mechanism that can identify potential parent nodes in the data distribution tree. CORIP is one such mechanism using which nodes can identify a set of nodes that have extra degree to contribute. Using CORIP, a node can identify resources ($IR1_{nodes}^n$, $IR2_{nodes}^n$ and $IR3_{nodes}^n$). The identified resources can be periodically probed for better alternate parent nodes. Had it not been for the CORIP or a similar mechanism, a node cannot know which nodes can be potential parent nodes. So, Using CORIP, We can successfully give a scope to the search for a better parent node.

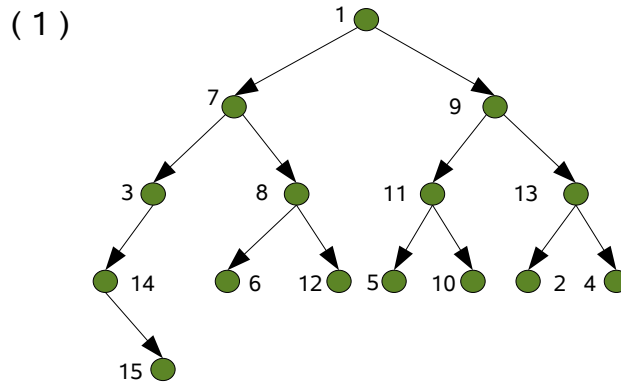
In DTR chapter, we discussed about the performance of DTR in a session with many degree constrained nodes. We found that DTR is able to optimise but not completely optimise the data distribution tree in such sessions. Here, We take a graph where DTR is not able to do any more repositioning due to degree restrictions and see how CORIP is able to optimise the distribution tree. Figures 23, 24 and 25 show the transformations in the node positions that were possible due to CORIP.

C_n^s = End-to-End delay for the data to reach the node n from source

D_n^p = Delay for data to reach from node p to node n

Let n represent the current node and X represent the node that is to investigated for its suitability as a parent node. A node moves from the current position to the new position if,

$$C_X^s + D_n^X < C_n^s \quad (7)$$



C_2^1	900 ms	P_2	300 ms
C_3^1	600 ms	P_3	300 ms
C_4^1	950 ms	P_4	350 ms
C_5^1	550 ms	P_5	100 ms
C_6^1	550 ms	P_6	150 ms
C_7^1	300 ms	P_7	300 ms
C_8^1	400 ms	P_8	100 ms
C_9^1	300 ms	P_9	300 ms
C_{10}^1	550 ms	P_{10}	100 ms
C_{11}^1	450 ms	P_{11}	150 ms
C_{12}^1	550 ms	P_{12}	150 ms
C_{13}^1	600 ms	P_{13}	300 ms
C_{14}^1	900 ms	P_{14}	300 ms
C_{15}^1	1000 ms	P_{15}	100 ms
$\Sigma C_n^s = C_{data}$	8600 ms	$\Sigma P_n = C_{tree}$	3000 ms

In this graph, every node can contribute to only 2 child nodes. Using CORIP, nodes identify the resources (i.e nodes that have an unfilled position)

CORIP identifies resource nodes as

$$IR = \{ 3, 14, 6, 12, 5, 10, 2, 4 \}$$

This set is available in all the participating nodes.

From here, the graph can make many possible transformation. Here, we show just one possible instance of it.



(A)

Figure 23: CORIP Application: Self-Organising Overlay

From the figures 23, 24 and 25, we see that 4 nodes (out of a total of 15 participants) were able to identify better node positions. Also the values of C_{data} and C_{tree} have been reduced by 11.63 % and 30.43 % respectively. From these results, we see that CORIP is able to add the self-organising nature to the data distribution tree. Also, Its operation together with DTR can be used to build data distribution tree with minimum end-to-end delay and network stress.

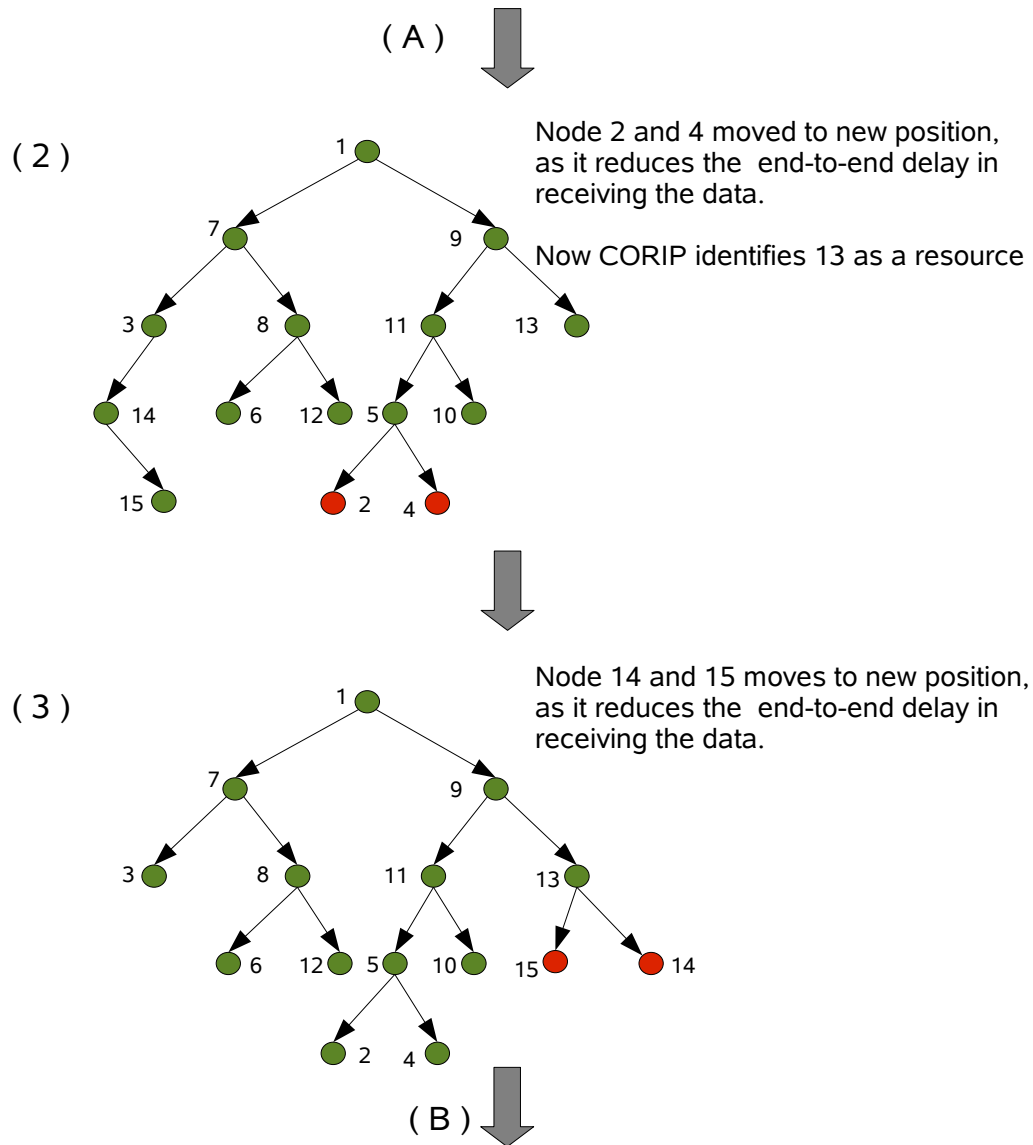
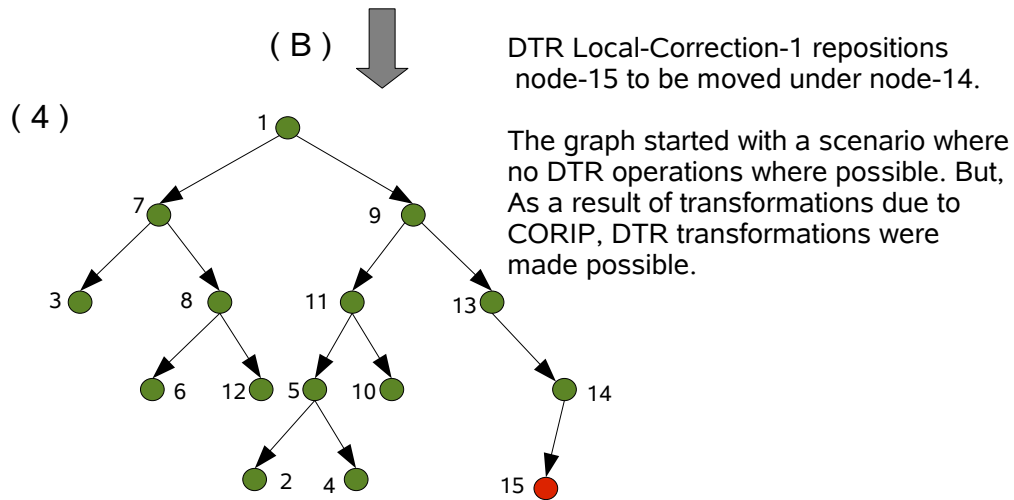


Figure 24: CORIP Application: Self-Organising Overlay



c_{2}^1	600 ms	P_2	50 ms
c_{3}^1	600 ms	P_3	300 ms
c_{4}^1	650 ms	P_4	100 ms
c_{5}^1	550 ms	P_5	100 ms
c_{6}^1	550 ms	P_6	150 ms
c_{7}^1	300 ms	P_7	300 ms
c_{8}^1	400 ms	P_8	100 ms
c_{9}^1	300 ms	P_9	300 ms
c_{10}^1	550 ms	P_{10}	100 ms
c_{11}^1	450 ms	P_{11}	150 ms
c_{12}^1	550 ms	P_{12}	150 ms
c_{13}^1	600 ms	P_{13}	300 ms
c_{14}^1	700 ms	P_{14}	100 ms
c_{15}^1	800 ms	P_{15}	100 ms
$\Sigma C_n^s = C_{data}$	7600 ms	$\Sigma P_n = C_{tree}$	2300 ms

Comparing the metric with starting position

C_{data} and C_{tree} metrics have improved by 11.63 % and 30.43 % respectively

Figure 25: CORIP Application: Self-Organising Overlay

4.2.2 Reactive Repair

One important issue with the tree-based data distribution mechanism is that the loss of a node results in disruption of service to all the descendant nodes. In such scenarios, the nodes need to find a new parent node to rejoin the data distribution tree. To find a new parent node, first they need to know contact addresses of the other (capable) nodes in the session. If the nodes have not maintained any information about the nodes which can be potential parent nodes, then the time taken to rejoin the overlay will be high. In the worst case, they need to contact the bootstrap server to retrieve a list of potential parent nodes. Contacting bootstrap server to make re-join attempts has a high cost associated with it (delay and loss of data). CORIP constructs and periodically updates information about capable nodes. As this information is a selectively filtered set of capable nodes, It can reduce the time taken by a node to rejoin the data distribution tree.

4.3 Future Work and Conclusion

In this chapter, We presented CORIP and analysed its working and applicability. From the initial results, We see that CORIP can be used in proactive tree optimisation and reactive tree repair mechanisms.

By its design nature, CORIP distributes the load of identifying resources among all the participants. And, it spreads the identified resource information to all the participants. This feature of the CORIP can make it scale for large topologies. To validate this statement, we need to evaluate its applicability in the overlay simulator. Currently, the CORIP functionality has not been implemented in our overlay simulator. So, this evaluation is planned as a future task.

5 Overlay network simulator design

In this chapter, we explain the design and implementation details of an overlay node in a discrete-event network simulator. Also, We describe the implementation details of DTR in an overlay node. This chapter is structured as follows: The overlay simulation environment that we built was by extending Network-simulator-2 [17]. So, we start by explaining the fundamentals of ns2 scheduling mechanism. Then we explain the design details involved in implementing an overlay node. Finally, we explain the implementation details of DTR.

5.1 Event scheduling mechanism in Ns2

Ns2 is a discrete event simulator. In discrete event simulation, the operation of the system is represented as a chronological sequence of events. Ns2 has a scheduler that maintains the chronological sequence. An explanation of the ns2 scheduling mechanism (taken from [18]) is as follows:

Ns2 is a single threaded discrete event simulator. The ns-2 scheduler maintains an internal virtual clock. The simulator objects use this virtual clock as a time reference. The scheduler also maintains a timely-ordered list of events and processes them one by one. It takes the next earliest event from the list, advances the virtual clock till the firing time of the event, and executes it till completion. Then the control returns back to the scheduler to execute the next event. There are two basic scheduler categories that differ in the method used to advance the virtual clock non real-time and real-time . In a non real-time scheduler, the virtual clock simply jumps between firing times of consecutive events. The real-time scheduler in contrast tries to execute events in the actual moments in real-time. It uses the physical clock of the machine as a real-time reference. If the firing moment of a next earliest event is in the future, the scheduler waits until that moment in time.

Overlay node can be considered as a system, which in turn is a collection of modules. Each module implements protocol for operations like bootstrapping, joining, routing etc. A protocol implementation can be modelled as a program that handles incoming events and does specific tasks at specific times. The type of task can depend on the ‘state variables’ of the protocol instance. In Ns2, protocol implementations are referred as agents or ns2 objects. All agents have functions for handling incoming packets and has timer objects than can schedule timeout events to the scheduler. The overview on how the agents and the scheduler interacts is illustrated in the figure 26. In the subsequent section, we explain the design and implementation details of the different modules of the overlay node. Each module of the overlay node may have one or more timer objects associated with them. So, the understanding of figure 26 can help in visualising the implementation of the overlay node.

Ns2: Scheduler And Protocol Agent Interaction - HOW?

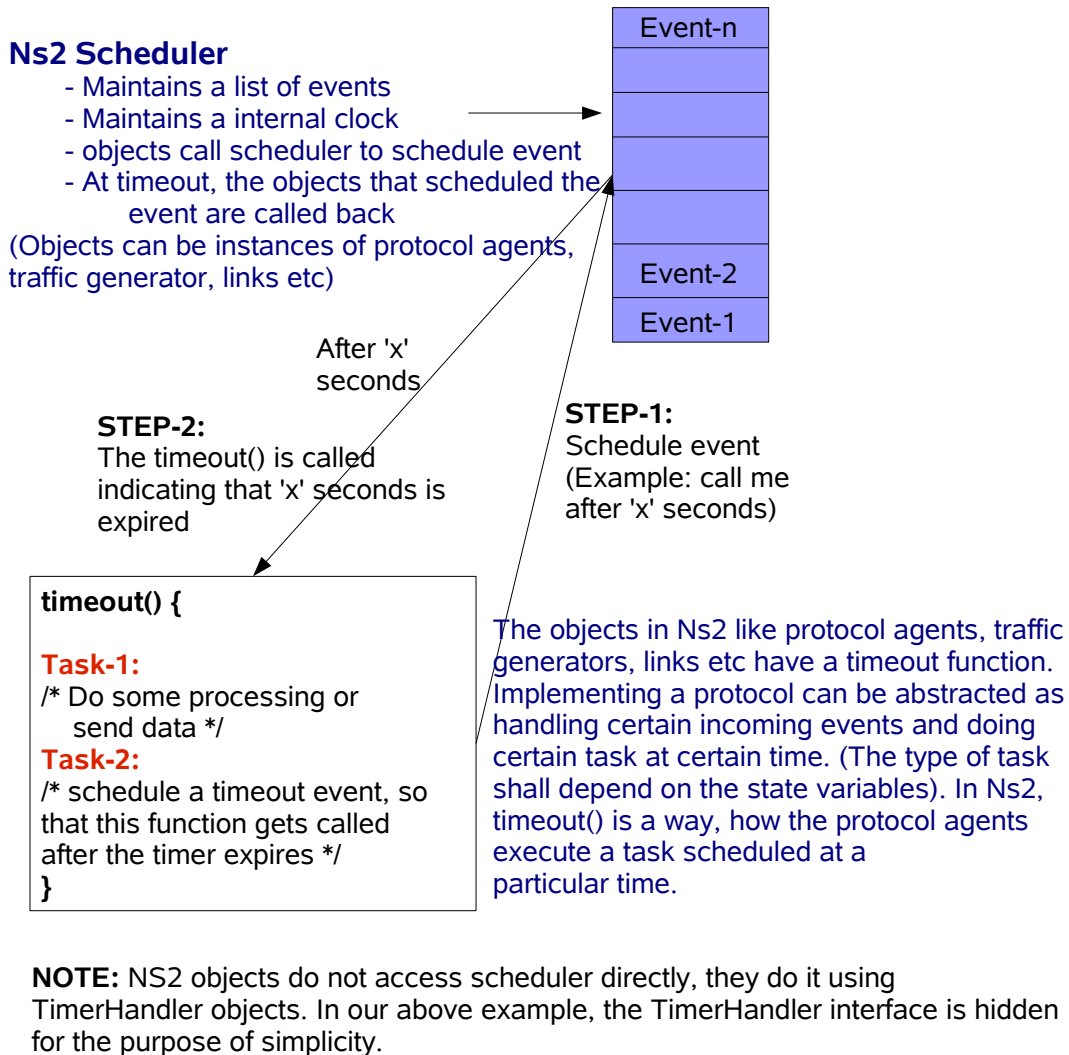


Figure 26: ns2: Scheduler and agents interaction

5.2 Design and implementation details of an overlay node

5.2.1 Design overview

In this section, we classify the overlay nodes and present the various functionalities that constitute the overlay nodes. Also, here we focus on the core components involved in implementing an overlay node. The implementation details related to DTR is explained separately in the next section. This differentiation is made, so that the components can be explained incrementally.

For an overlay-based data distribution network, there are three types of nodes, they are participating node, data source node and bootstrap server node. The functionality needed in an overlay node are presented below.

1. *bootstrap client*: Used for connecting to the bootstrap server to retrieve potential parent node addresses and to periodically inform the *bootstrap server* about the node's presence.
2. *bootstrap server*: It maintains a list of active overlay participants and responds to bootstrap client with the contact addresses of the potential parent nodes.
3. *overlay client*: It is used for connecting to the overlay, by making join requests to the addresses retrieved by the bootstrap client.
4. *overlay server*: It handles incoming join requests from newly joining (or re-joining) nodes. It is also responsible for detecting loss of child nodes.
5. *data source*: It acts as a data source and sends the generated data to the child nodes. *data forwarder*: It handles the data received from the parent node and forwards the received data to the child nodes. It also detects the loss of the parent node.

In the table 2, we represent, which set of modules constitute a participating node, data source node and bootstrap server node. Further in this section, when we refer to *bootstrap server* and *data source*, we are referring to the module that is part of the bootstrap server node and data source node respectively.

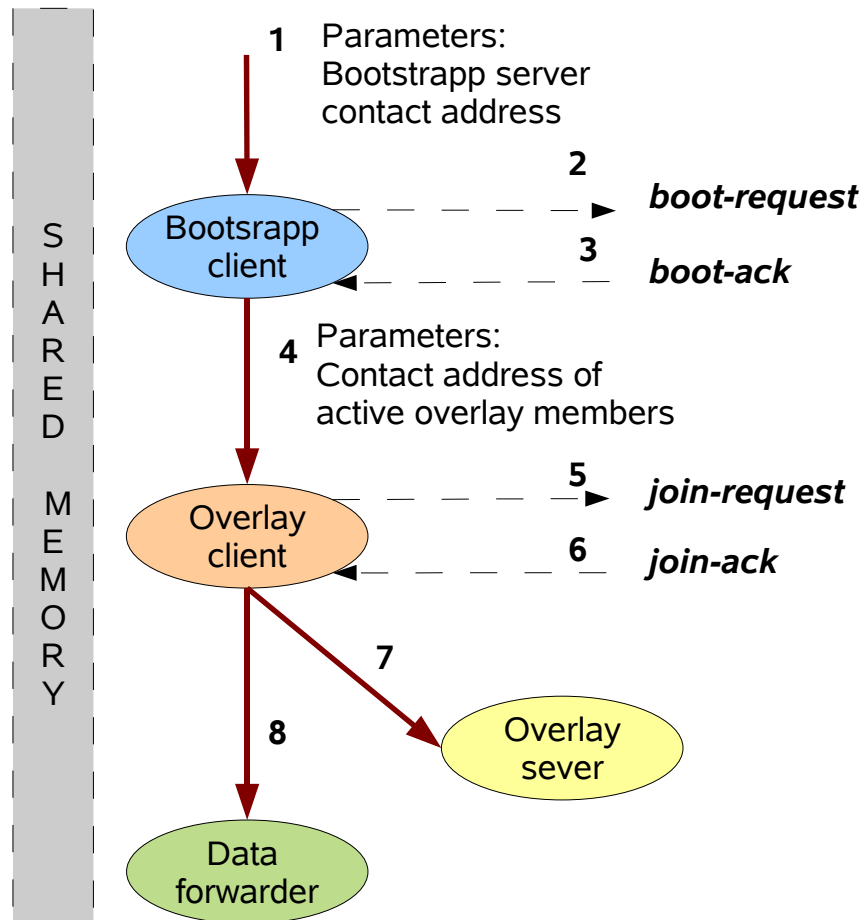
Modules	participating node	data source	bootstrap server node
bootstrap server	×	×	✓
bootstrap client	✓	✓	×
overlay server	✓	✓	×
overlay client	✓	×	×
data source	×	✓	×
data forwarder	✓	×	×

Table 2: Overlay node types and their modules

5.2.2 Implementation Details

In this section, we provide detailed description on how the modules are implemented. The figure 27 is provided to support the process of understanding the implementation details. Further in this chapter, when we refer to 'expiry of timer event', it means that the corresponding module's *timeout()* function is called by the ns2 scheduler. In real-world implementations, it can be done using the timer related functions provided by the the operating system.

Scenario: A node join successfully to the overlay network. The sequence in which the modules are initialized is presented below.



- Step-1: Node is initialized with the bootstrap server address as input.
- Step-2: *boot-request* is sent to bootstrap server node.
- Step-3: Receives *boot-ack* from bootstrap server node.
- Step-4: Initializes the *overlay client* with the contact addresses received in the *boot-ack* message
- Step-5: Sends *join-request* to an active overlay member.
- Step-6: Receives *join-ack* from the requested member.
- Step-7 & 8: Initializes *overlay sever* and *data forwarder* modules.

The overlay node has a shared memory module, to which all the modules have access. The shared memory module has functions to access the shared state of the node.

Figure 27: Initialisation sequence of the modules

1. Bootstrapp server: *Bootstrapp server* handles two message events (*boot-request* and *keep-alive*) and one timer event (*clean-up*). This module maintains a sorted list of active overlay members. The list is sorted for the capability value reported by the active nodes, using *keep-alive* messages. The operating overview of the *bootstrap server* mechanism is depicted in the function 7.

boot-request: *boot-request* messages are sent by *bootstrap client*. The response to this event is a *boot-reply* message that consists of '*Num_records*' (count) contact addresses of active overlay members that can be potential parent nodes.

keep-alive: *keep-alive* messages are periodically sent by active overlay members and they carry the capability information of the reporting node. By capability information, we refer to the count of the number of child nodes that the reporting node can accommodate. Absence of *keep-alive* message for a duration *Active_check*, results in removal of the node's record from the active overlay member list.

clean-up: The *bootstrap server* maintains the list of active overlay members. Over a period of time, this list accumulates obsolete records i.e, nodes that are not currently participating in the session. So, for every *Active_check*, the obsolete records are erased from the list.

2. Bootstrapp client: *Bootstrapp client* handles one message event (*boot-reply*) and one timer event (*boot-timer*). When initializing the *bootstrap client*, the address of the bootstrap server node is passed as the parameter. At start, the *bootstrap client* sends *boot-request* and schedules the *boot-timer* event to be triggered after *boot_timeout* seconds.

boot-reply: On receiving the *boot-reply* message, this module initialises the *overlay client* module. Also, it cancels the *boot-timer* event. The addresses returned by the *bootstrap server* are passed as parameters to overlay client during initialisation.

boot-timer: This timeout means that there was no reply to the *boot-request*. On this timeout, the *boot-request* is sent again and *boot-timer* is rescheduled to be triggered after *boot_timeout* seconds, provided *boot_attempts* is less than *MaxBoot_attempts*.

3. Overlay server *Overlay server* handles three message events (*join-request*, *leave-msg* and *child-report*) and one timer event (*child-timer*). *Overlay server* is initialised when an overlay node joins the overlay-network.

Algorithm 7 Bootstrapp server mechanism

```

nodeList = {}
eventTypes = {boot-request, keep-alive, clean-up}
repeat
  event ← recvdEvent
  currTime ← event.currTime
  if event = boot-request then
    returnList ← {}
    for i = 1to5 do
      if nodeList.iisempty then
        break
      end if
      returnList ← returnList + nodeList.i
    end for
    return returnList
  else if event = keep-alive then
    avlBw ← event.avlBw
    nodeAddr ← event.nodeAddr
    if nodeAddr in nodeList then
      nodeRecord ← nodeList.nodeAddr
      nodeRecord.avlBw ← avlBw
      nodeRecord.time ← currTime
    else
      newNodeRecord ← 0
      newNodeRecord.nodeAddr ← nodeAddr
      newNodeRecord.avlBw ← avlBw
      newNodeRecord.time ← currTime
      nodeList ← nodeList + newNodeRecord
    end if
    sort nodeList
  else if event = clean-up then
    for i = 1 to nodeList.length do
      nodeRecord ← nodeList.i
      if currTime - nodeRecord.time > 10 then
        nodeList ← nodeList - nodeRecord
      end if
    end for
  end if
until wait for an event

```

join-request: On receiving this event, the *overlay server* checks if it has capability to accept this request. If capacity is available, then *join-ack* is sent, else the request is ignored.

leave-msg: *overlay server* module maintains the set of child nodes to which it is forwarding the data stream. On receiving the *leave-msg*, the node that sent the *leave-msg* message is removed from the child node set.

child-report: Every node has to periodically send messages to its parent node. These messages (*child-report*) can serve as keep-alive messages and can also carry the metrics related to the quality of the received data. On receiving this message, *overlay server* module updates the timestamp that represents the activeness of the child node that sent the report.

child-timer: On this event, all the child nodes are verified for their activeness. If any child node is found to be inactive for a continuous period of $child_{timeout}$, then it is removed from the child node set.

4. Overlay client *Overlay client* handles one message event (*join-ack*) and two timer events (*join-timer* and *report-timer*). When initialising the *Overlay client*, it is given the contact addresses of the potential parent nodes as input. At start, the *Overlay client* sends *join-request* to the potential parent nodes passed to it during initialisation and also schedules the *join-timer* to be triggered after $join_{timeout}$ seconds.

join-ack: The reception of this message indicates that the node has connected successfully to the overlay network. On receiving this message, the *join-timer* event is cancelled and the *report-timer* is scheduled to be triggered after $report_{timeout}$ seconds.

join-timer: This timeout means that there was no reply to the *join-request*. On this timeout, the *join-request* is sent again and *join-timer* timer is rescheduled to be triggered after $join_{timeout}$ seconds, provided $join_{attempts}$ is less than $MaxJoin_{attempts}$.

report – timer: On the expiry of this timer, a *child-report* is sent to the parent node and the timer is rescheduled. The *child-report* message can carry metrics on the quality of the data received by this node.

5. Data source *Data source* handles one timer event (*data-timer*). When the module is initialised, the *data-timer* is scheduled to be triggered after $data_{timeout}$

seconds. The value for the $data_{timeout}$ depends on the packet size and the bit rate of the data being distributed.

data-timer: On expiry of this event, the *data-msg* is sent to all the child nodes and the timer is rescheduled to be triggered after $data_{timeout}$ seconds.

6. Data forwarder *Data forwarder* handles one message (*data-msg*) event. On receiving this message, it forward the message to all the child nodes connected to it. Data forwarder can also detect loss of parent node by detecting the absence of *data-msg* for a prolonged time.

Engineering decisions: In designing a distributed system, we need to choose effective values for the variables involved in the system. Below, we discuss on the values assigned to the variables of different overlay modules.

Bootstrapp server: For the variable $Num_{records}$, '5' was set as its value. Experimentally, we find that it worked well for our scenario. These parameters may require tuning depending on the arrival pattern of the nodes, churn and the characteristics of the application.

Bootstrapp client: For the variable $MaxBoot_{attempts}$, '3' was set as its value and for the $boot_{timeout}$ variable, '1.5' seconds was set as its value. As we did not perform simulations with different packet loss values, we cannot decide on the effectiveness of these values. Simulations with different packet loss percentages and diverse topology structure (influences end-to-end delay) are needed to find the effective values for these variables.

Overlay server: For the variable $child_{timeout}$, '5' was set as its value. The value chosen for this variable is directly linked with the inter-reporting interval of *child – report* message from the *overlay client*.

Overlay client: For the $join_{timeout}$, '3' seconds was set as its value and for $report_{timeout}$, '1.5' seconds was set as its value. The value of $report_{timeout}$ is directly linked to the value used for $child_{timeout}$ (by *overlay server*). The value used by us for $child_{timeout}$ is based on the following design decision: A parent node can consider a child node to be lost, if three consecutive reports from the child node are lost.

5.3 Design and implementation overview of DTR

DTR is the combination of three node repositioning algorithms (*local correction-1*, *local correction-2* and *local correction-3*). In this section, we specifically discuss on how the required state space for the operation of the DTR are constructed by the participating nodes.

Type-1		Type-2	
From	To	From	To
P	C1	C2	C1
P	C2	C2	C3
P	C3	C3	C1
C1	C2	C3	C2
C1	C3	-	-

Table 3: Metrics required for *Local correction-1*

5.3.1 Local correction-1:

Local correction-1 is a repositioning algorithm involving nodes that have parent-child relationship. The repositioning decision is made by the parent node based on the collected metrics. Let 'P' be the parent node of three child nodes C1, C2 and C3. The *Local correction-1* requires the knowledge of the (virtual) link costs tabulated in table 3, to make the repositioning decisions. From table 3, we see that the links can be either 'parent-to-child' or 'child-to-child' (siblings).

Calculating *parent-to-child* link delay: The parent node includes the 'sending timestamp' in the data packets that are forwarded to the child nodes. On receiving the data packet, the child node calculates the delay between the parent node and itself. The *child-report* (refer 5.2.2) message is used to report the calculated delay metric to the parent node.

Calculating *child-to-child* link delay: The parent node periodically reports (*sibling-data*) to its child nodes. The (*sibling-data*) report carries information about all the child nodes operating under a parent node. These periodic reports enable the child nodes to know about their sibling nodes. Using this state information, the child nodes periodically probe and calculate the link delay metrics with their sibling nodes. The *child-report* message is used to report the calculated metric to the parent node.

Repositioning decisions: The parent node periodically ($dtr_{timeout}$) evaluates the collected metrics and decision is made on node repositioning. If a need for repositioning is identified, then the child nodes are informed about the change in distribution path using a *route-update* message.

Engineering decisions: In our simulations, '10' seconds was used as the value for $dtr_{timeout}$. The effectiveness of the value chosen for $dtr_{timeout}$ directly depends on the inter-reporting interval of *child-report* and *sibling-data* messages. In our simulations, '5' seconds was used as the inter-reporting interval for both the messages (*child-report* and *sibling-data*) and we experimentally find that these values are

Type-1		Type-2		Type-3		Type-4	
From	To	From	To	From	To	From	To
P	C	P	G1	C	G1	G1	C
-	-	P	G2	C	G2	G2	C

Table 4: Metrics required for *Local correction-2 and 3*

providing the required results. Here, one important design decision is to choose ‘reporting interval’ that is less than the value chosen for $dtr_{timeout}$. This ensures that for every $dtr_{timeout}$ seconds, the metrics values are refreshed.

The *route-update* message carries repositioning decisions, So it is a critical message (loss of this message disrupts the distribution path). To increase the reliability of this message, two parameters ($routeAck_{timeout}$ and $routeUp_{attempts}$) need to be configured. The $routeAck_{timeout}$ represents the duration till which the DTR module can wait for *route-ack* (acknowledgement for *route-update*) before retransmitting the *route-update*. The $routeUp_{attempts}$ represents the number of attempts that can be made for retransmitting the *route-update* message. The simulator at this point does not have functionality to retransmit the *route-update* message. But we see that the effective values ($routeAck_{timeout}$, $routeUp_{attempts}$ and inter-reporting interval of *child-report* and *sibling-data* messages) for these parameters will directly depend on the application using the overlay, the operating environment (packet loss ratio, network topology etc) and the timeout value used for detecting the loss of parent node.

5.3.2 Local correction-2 and 3:

Local correction-2 and 3 are repositioning algorithms involving nodes having parent-child and parent-grandchild relationship. The repositioning decision is made by the parent node based on the collected metrics. Let ‘P’ be the parent node of a child node ‘C’ and let ‘C’ have two child nodes (G1 and G2). The *Local correction-2 and 3* requires the knowledge of the (virtual) link costs tabulated in table 4, to make the repositioning decisions. From table 4, we see that the links can be ‘parent-to-child’ or ‘parent-to-grandchild’ or ‘child-to-grandchild’ or ‘grandchild-to-child’.

The metric collection mechanism for ‘parent-to-child’ (P to C) has already been discussed in the previous section. In the case of ‘child-to-grandchild’ (C to G1 and C to G2), it is also a form of ‘parent-to-child’ metric, but this metric need to be reported to the ‘P’ node. So, Every child node periodically reports (*child - gc*) to the parent node about the ‘child-to-grandchild’ (i.e metrics of (C to G1) and (C to G2)) metrics.

Calculating *parent-to-grandchild* link delay: The child node periodically reports (*child - data*) to its parent node. The (*child - data*) report carries information about all the child nodes operating under the reporting node. These periodic re-

ports, enable the parent nodes to know about their grandchild nodes. Using this state information, the parent node periodically probes and calculate the link delay metrics with their grandchild nodes.

Calculating *grandchild-to-child* link delay: It is a form of ‘child-to-parent’ metric which need to be reported to the grandparent node. Every child node periodically measures link delay with its parent node and reports (*gChild – data*) it to the grandparent node.

Repositioning and Engineering decisions: The same approach as discussed in the section 5.3.1 is used to inform the affected nodes about the repositioning decisions. Regarding the engineering decisions, ‘5’ seconds was used as the inter-reporting interval for both the messages (*child – data*) and (*gChild – data*). From our experiments, we see that the chosen values gave expected results. Here, one important design decision is to choose reporting interval that is less than the value chosen for $dtr_{timeout}$. This ensures that for every $dtr_{timeout}$ seconds, the metrics are refreshed. This reasoning also applies to the reporting interval value chosen for *child – report* and *sibling – data*.

5.4 Summary

In this chapter, we have seen the design and implementation details of an overlay node. Designing an overlay node requires proper configuration of different parameters like timeout value, number of retransmission etc. The design overview discussed here is also applicable for a real-world overlay-node design.

6 Conclusion and Future Work

In this thesis, we have provided solutions that construct an efficient data distribution tree (DTR) and an effective protocol (CORIP) for identifying capable nodes (nodes that have extra contributing resources) in an overlay-network. We have also presented the design and implementation details on the building of an overlay node, in an discrete-event simulator.

DTR constructs minimum spanning tree for data distribution in overlay networks. DTR is completely decentralised, self-organising and adapts to network dynamics. From the simulation results, we see that the solution is very efficient in reducing the end-to-end delay in receiving the data and the stress to the underlying network.

CORIP proposes a new approach of identifying capable nodes in an overlay network. In this approach, the load of identifying capable nodes are shared among all the participating nodes. We have presented initial results on, how CORIP together with DTR can optimise the data distribution path in overlay networks with many degree constrained nodes.

We presented the details and issues involved in implementing an overlay node in a discrete-event simulator. We also presented the engineering decisions that need to be made in the design and implementation of the overlay nodes. The details and issues discussed in this regard are also valid for a real-world overlay node implementation.

Our bigger objective is to build a solution that covers all aspects of distributing data in an overlay network. On this direction, the following are the tasks that requires further research and analysis.

1. Need to implement CORIP in the overlay simulator. Need to study the effectiveness of CORIP and DTR in optimising the data distribution path in resource constrained overlay networks. It requires large scale simulations with diverse parameter sets.
2. In implementing the overlay node, we had seen many variables which need to be configured appropriately (example: inter-reporting interval, timeout values etc). Need to study the correlation of these variables with each other and with the operating environment. Based on the study, we plan to propose rules for the design of an effective overlay node.
3. Need to study the existing reactive repair mechanisms and draw conclusions on their effectiveness. If needed, we plan to look for alternate solutions in this regard. Here by reactive repair, we refer to the mechanisms used by a node that lost the parent node, to re-join the overlay with minimal or no loss of data. The resource list built using CORIP is already a step in this direction.

References

- [1] B. Quinn, K. Almeroth *RFC 3170: IP Multicast Applications: Challenges and Solutions*.
- [2] Akamai Technologies <http://www.akamai.com>
- [3] Hei, X. and Liang, C. and Liang, J. and Liu, Y. and Ross, K. W.: *Insights into PPLive: A Measurement Study of a Large-Scale P2P IPTV System*. In Proc. of IPTV Workshop, International World Wide Web Conference, July 2006.
- [4] <http://http://www.pplive.com/>
- [5] <http://www.sopcast.com/>
- [6] <http://www.tvants.com/>
- [7] X. Zhang, J. Liu, B. Li, and T.-S. P. Yum, *DONet/CoolStreaming: A Data-driven Overlay Network for Peer-to-Peer Live Media Streaming*. In IEEE INFOCOM, vol. 3, Mar. 2005, pp. 2102–2111.
- [8] Susu Xie; Bo Li; Keung, G.Y.; Xinyan Zhang, *Coolstreaming: Design, Theory, and Practice*. Multimedia, IEEE Transactions on, Volume 9, Issue 8, Dec. 2007 Page(s):1661 - 1671.
- [9] Hei, Xiaojun and Liu, Yong and Ross, K. W.: *IPTV over P2P streaming networks: the mesh-pull approach*. Communications Magazine, IEEE, 2008.
- [10] Jiangchuan Liu and Sanjay G. Rao and Bo Li and Hui Zhang: *Opportunities and Challenges of Peer-to-Peer Internet Video Broadcast*. 2007.
- [11] Yang-hua Chu, Sanjay G. Rao, Srinivasan Seshan & Hui Zhang: *A Case for End System Multicast*. In ACM Sigmetrics, June 2000.
- [12] Suman Banerjee, Bobby Bhattacharjee & Christopher Kommareddy: *Scalable Application Layer Multicast*. In ACM Sigcomm, August 2002.
- [13] S. Banerjee and C. Kommareddy and K. Kar and B. Bhattacharjee and S. Khuller: *Construction of an efficient overlay multicast infrastructure for real-time applications*. INFOCOM, 2003.
- [14] <http://www.bittorrent.com/>
- [15] Ellen W. Zegura and others: <http://www.cc.gatech.edu/projects/gtitm> *GT-ITM Topology Modelling*. Last updated 26 July, 2000.
- [16] Felix C. Gärtner: *A Survey of Self-Stabilizing Spanning-Tree Construction Algorithms*. June, 2003.
- [17] <http://www.isi.edu/nsnam/ns/>

- [18] Daniel Mahrenholz and Svilen Ivanov: *Real-Time Network Emulation with ns-2*. 2004.