# APPCAST - A NEW APPLICATION LAYER MULTICAST SYSTEM

Thesis submitted for the award of
Doctor of Philosophy

V. RADHA

Department of Computer & Information Sciences
School of Mathematics and Computer/ Information Sciences
University of Hyderabad

NOVEMBER 2004

**Department & Computer & Information Sciences School of Mathematics and Computer/ Information Sciences**
**University of Hyderabad**
**Hyderabad - 500 046**

This is to certify that 1, **V. Radha,** have carried out the research embodied in the present thesis for the full period prescribed under the Ph. D. ordinances of the University.

I declare to the best of my knowledge that, no part of this thesis was earlier submitted for the award of any research degree of any University.

[illegible]
**V. Radha**

[illegible] 8/11/09
**Prof. A. K. Pujari**
Supervisor & Head of the Department
University of Hyderabad

[illegible]
**Dr. V. P. Gulati**
Supervisor
IDRBT

[illegible]
**Prof. Rajat Tandon**
Dean of the School

## ACKNOWLEDGMENTS

*This work would not have been possible without the active help of many well-wishers, who chipped in at various stages to ensure that I succeeded in this endeavour.*

*My supervisors Prof V. P. Gulati and Prof A. K. Pujari gave me an opportunity to work on this doctoral thesis and fine-tuned the project idea. They inspired me to take on this work and remain focused. I will always remember their zeal and encouragement. My sincere gratitude to both of them.*

*I am deeply grateful to the Faculty of the Institute for Development and Research in Banking Technology and the Department of Computer and Information Sciences, University of Hyderabad, for their valuable advices during this project. They helped me not only in the planning phase but were also readily available for discussions, whenever required.*

*I sincerely thank the staff at the IDRBT and the Librarian for readily making available a lot of data and guiding me to zero in on the right resources. Without them, this task would have been much more difficult and time-consuming.*

*Thanks are also due to my family. Each one of them shared my responsibilities and took off all the pressures, thereby allowing me to concentrate on this work. My warmest thanks to my family. I also thank my friends who were a source of constant encouragement.*

*November 2004* *V. Radha*

# TABLE OF CONTENTS

## LIST OF FIGURES

## LIST OF TABLES

## Abstract

This thesis presents a new *Application Layer Multicast* **protocol.** To reduce the load on servers and increase the network efficiency, *Application Layer Multicast* protocols construct overlay networks and distribute data efficiently. E-mail distribution and USENET news distribution are early Internet applications that fall into the broad general category of *Application Layer Multicast. Application Layer Multicast* protocols do not change the network infrastructure. These protocols implement multicast forwarding mechanism at the end hosts. Many researchers are finding ways to push the multicast routing capabilities onto hosts than onto routers. The idea here is to construct an overlay network of hosts, on top of routers. Routers do normal unicast routing, while the hosts attached to them will take care of multicast group management and routing among themselves. While *IP Multicast* protocol works at network layer, that requires applications to be aware of multicast and programs be written specifically using *IP Multicast* API [76]; protocols like *End-system Multicast* [1-2], *Scattercast* [3-4] and *Overcast* [5] require an application-layer overlay infrastructure to be laid of. All these protocols, share the common goal of providing the benefits of multicast without requiring direct router support or the presence of a physical broadcast medium [4].

The main contribution of this thesis is to give a DNS kind of infrastructure based solution for multicast services. We emphasize that multicast can better be provided at application level rather than at router level. To provide multicast at application level we present four major contributions named 1 .Construction of Appcast Overlay. 2. Exploiting Broadcast Media Property. 3. Exploiting Query Redundancy and 4. A Generic Multicast Application Development Framework.

**1. Construction of Appcast Overlay:** We propose a new application layer multicast that exploits the topology information. Any application layer multicast protocol that docs not exploit topology may in fact decrease the efficiency in terms of bandwidth and increase delay. We develop new algorithm 'Appcast' that creates a multicast overlay topology, which we further optimize by two more extensions to it taking delay and processing power as parameters.

**2. Exploiting Broadcast Media Property: The application developers are not able to** take advantage of the media nature like broadcast for satellite networks. We show ways of how the broadcast nature can best utilized by developers.

3. **Exploiting Query Redundancy:** Hosts **are** organized as an overlay in application layer multicast. Overlays disseminate information in a hierarchical and incremental fashion. Also, not all members in an overlay or multicast group need exactly same information. In this case, if we can filter the information at different levels of overlay, while the information is distributed, we can minimize the bandwidth and processing requirement. Each member's interests are represented by Xpath [78] query and the information to be distributed is in the form of XML [80] document **as** XML is gaining importance as the standard for information exchange. We look at the XML document-processing problem particularly in the context of application layer multicast and propose a new Xpath processing algorithm called YALXP - Yet Another Light weight Xpath [78] processor. YALXP algorithm processes multiple Xpath queries over a XML document in single document traversal i.e. in one pass it can answer multiple queries.

**4. A Generic Multicast Application Development Framework:** We design generic application architecture and develop three different kinds of applications. We take 'scheduled file pushing' application as one-way multicast application, 'database replication' as event and rule based one-way multicast application and 'auctions' as interactive multicast application and implement all three.

Chapter 1

# Introduction

This dissertation presents a new *Application Layer Multicast* Protocol, called Appcast for providing multicast services over Internet. Multicast protocols aim at eliminating the redundant movement of data packets in a network. *Application Layer Multicast* protocols, construct overlay network for efficient data transmission over large networks like Internet. In *Application Layer Multicast* protocols, hosts take over the job of multicast routing, which is in contrast to IP Multicast [23] wherein routers do the additional job of multicast routing. The first part of this dissertation is about *Application Layer Multicast* protocols and Appcast. *Application Layer Multicast* protocols construct and manage overlays. The first protocol in this series is ESM [1-2] and is considered very inefficient from control and management perspective i.e. it has a very high control overhead of $O(n^2)$. While protocols like CAN[18-19], DT Protocolp 1] do not consider the topology information at all, others like NICE[15-17], HMTP[13] etc could not exploit topology information to maximum. Our protocol constructs overlay in alignment of the underlying network topology and also keeps the control overhead to $O(n)$. The second part deals with how to architect efficient multicast applications over Appcast. The third part is concerned about how TCP/IP can be improved to leverage upon the broadcast nature of the media for multicast applications. While *Application Layer Multicast* protocols are efficient in reducing the redundant data packet movements, they introduce slight delay compared to native IP Multicast. So, the fourth part of this dissertation shows how this delay can be minimized using efficient XPath[78] stream processing. New Internet standards like XML[80], SOAP[47-49] etc. have given us opportunities to architect multicast services over Appcast as an end-to-end solution.

## 1.1 State of **the Art** Technologies

Internet is popular for its efficiency in distributing information: mail, news, web pages (HTML), files of all types like jpg, mpg etc, chat channels, DNS records, audio and video broadcast and so on. While majority of the information that flows across Internet is for mass distribution of static content, the dynamic and two way interactive applications could not scale up to Internet. This is quite evident from DNS that used centralized system ultimately moved to a distributed environment by arranging the DNS servers in **a** hierarchical fashion. The zone transfers replicate DNS information on to many servers, to make the two-way interactive query-response system of DNS work efficiently by keeping the responses nearer to users who make queries.

The 1999 Victoria's secret webcast with 1.5 million hits [105], ultimately could not serve any request. The 1998 distribution of Starr Report is another example. Servers like Netscape and major content distributors are forced to look into alternatives like distributed networks of servers, simply to carry the load. Akamai [106] is one such network, with almost 4200 servers across the globe located in 50 countries. To reduce the load on servers and increase the network efficiency, many schemes like *Content Distribution Networks, Active Networks, IP Multicasting, Overlay Networks* etc are proposed. In this section, we briefly describe each area of technology that greatly increases the utilization of information, bandwidth and computing resources on the Internet,

### 1.1.1 Load Balancing

Though, in recent years both network and server capacity have improved, response time continues to challenge researchers, as centralized web systems are not able to scale. Improving the power of a single server does not solve the web scalability problem in a foreseeable future. Another solution is to deploy a distributed web system [70] composed of multiple server nodes and distributing the load reaching this web site evenly among the server nodes, so as to improve system performance. Therefore, any distributed web system must include some component (under the control of the service provider) that routes client requests among the servers with the goal of maximizing load sharing. This component can be deployed at various levels of TCP/IP stack like - NAT at the MAC and IP Level [108]; *round robin DNS* at DNS level [109] and *proxies* at application level. Adding more nodes, complete with processors, storage and bandwidth, expands the system capabilities.

### 1.1.2 Active Networks

The design of inter-network protocols abstracts away how messages are routed or forwarded through out the network, keeping the applications away from the complexities of underlying network. This design makes it hard for the applications to exploit detailed knowledge of the network in order to enhance their performance and reduce network load. This abstraction is actually degrading the performance of both the application and network [66]. *Active Networks'* prime concern is to make the network dynamic and more intelligent by allowing new protocols and application code to be downloaded on to any network device - like routers, hosts, clients etc. The code modifies the behavior of router for particular type of packets specific to application needs. ANTS[107] is the most successful toolkit. Deployment problems and serious security issues are discouraging the ISPs to deploy *Active Networks.* A variation of *Active Networks* is *Active Services Framework* that recommends programmable service architecture. With this architecture, the core of the network remains unchanged, but it allows applications and users to download and execute code at strategic locations for the application i.e. code is executed on hosts instead of on routers.

### 1.1.3 Overlay Networks

*Overlay Networks* inherently use the concept of *Active Networks* [66]. In overlay networks, hosts form an overlay of unicast connections to cooperate and communicate over the general Internet infrastructure.

### 1.1.4 Content Distribution Systems

IP *Content Distribution Networks* (CDN) are special purpose overlay networks that provide scalability by distributing many servers across the Internet close to consumers. Consumers obtain content from these edge servers directly rather than from the origin server. One of the major techniques used for content distribution is *Caching. Caching* [68, 69] technique's main aim is to relieve the source from servicing many connections. *Caching* allows storing the initial service responses in intermediary servers like *proxy/cache* servers, which will serve subsequent requests on behalf of the actual source server. The *caching* concept can be extended in hierarchical manner. Content distribution involves pushing the content onto these cache servers. PRISM's content naming, management, discovery and redirection mechanisms [71] support high quality streaming media services in an IP based *Content Distribution Network. Internet Backplane Protocol* [72], or IBP supports logistical networking to allow applications to control the movement and storage of data between nodes. A number of service **providers (e.g., Adero,** Akamai and Digital Island) operate content distribution networks, but in-depth information describing their internals is not public information.

### 1.1.5 Peer-to-Peer Network

As the computer and communication technology is advancing, the distinction between server and client is diminishing in terms of computing power and they become a network of peers. CDNs' main purpose is to distribute content and is designed from the perspective of a distributor who establishes infrastructure and controls the same across Internet. In contrast, *Peer-to-Peer* network emphasizes on communal sharing of computer resources [73,74], without any controlling authority. *Peer-to-Peer* computing opposes traditional client-server computing. In *Peer-to-Peer,* an application is split into components that act as equals. Gnutella [110] has become the most successful development, though it is inefficient. Gnutella broadcasts thousands of messages per request. *Peer-to-Peer* systems provide limited support for applications that go beyond file sharing. It some how, could not go beyond this fundamental application, probably due to its very objective of file sharing and unreliability of the protocols as peers come and go at unpredictable times.

### 1.1.6 Grid Computing

Networks' communication speed is outpacing computers' processing speed making communication essentially free. The doubling of network performance relative to computer speed every 18 months [75] has made the researchers think about many collaborative applications from different areas like life sciences - genomics; engineering - air craft control; physical science - astronomy, particle, nuclear; business – fraud detection, anti money laundering etc that are resource and process intensive. A grid is a kind of *Peer-to-Peer* network, in which an application can draw computational power equivalent to super computer, transparently from the grid. The grid allows sharing of resources like memory, processor etc on loosely coupled systems on Internet.

## 1.2 Multicast

Though much communication over Internet is for group distribution (inherently) in the form of replications, caching etc, every distribution application on the Internet runs over the unicast infrastructure and derives its distribution functionality from application specific mechanisms. For instance, RFC 822 mail headers have mechanisms for detecting loops among mail forwarders[111]; NNTP has the NEWNEWS command [112] to manage flooding of news articles and HTTP has redirection and mechanisms for handling caches [113] and so on.

All the above applications are written independently and use the transport layer API and achieved some sort of multicast capability by sending the same information multiple times but to different destinations, though crossing the same routers at times, as no multicast infrastructure available over Internet.

The goal of multicast or broadcast mechanism is to eliminate redundant packet replication in a network, when a group of computers participate in a communication. Multicast can no more be just a choice but it is requirement as it can greatly improve the efficiency of Internet, by reducing unnecessary traffic movement. Multicast is an efficient mechanism that transmits only one copy of data to the receivers by replicating it as necessary while traveling through the Internet till it reaches the receivers. This is in contrast to unicast, where in multiple copies are sent by the source to the receivers, requiring much processing capacity at source and more bandwidth on network links.

### 1.2.1 Global Wide Area Multicast Applications

Internet RFC 3170 has listed many multicast applications[114]. One can define multicast communication as one that has parallel/simultaneous communication from/to a group of entities like hosts, people etc. Based on this definition we can arrive at three general categories of multicast applications:

**One-to-Many (1-to-M):** A single host communicates to two or more (n) receivers in this kind of multicast application.

**Many-to-Many** (M-to-M): In this model, any number of hosts can send information to the same group, as well as receive from it.

**Many-to-One (M-to-1):** Any number of hosts can send information to a single host in this type of multicast application.

#### 1.2.1.1 One-to-Many (1-to-M) Applications

**a) Scheduled audio/video (a/v) distribution:** Lectures, presentations, meetings, or any other type of scheduled event like news (i.e. television and radio "broadcasts") etc fall into this category of applications.

b) **Push media:** News headlines, weather updates, sports scores etc applications represent this kind of multicast where in information is pushed onto users' appliances like TV, phone, computer etc. Relatively low-bandwidth data characterize these applications.

c) **File Distribution and Caching:** Applications like distributing web site content, executable binaries and other file-based updates to end-user or replication/caching sites etc use multicast.

d) **Announcements:** Network time, multicast session schedules, random numbers, keys, configuration updates etc applications can exploit multicast. Their bandwidth demands can vary, but generally they are very low bandwidth.

e) **Monitoring:** Stock prices, sensor equipment (seismic activity, telemetry, and meteorological or oceanic readings), security systems, manufacturing or other types of real-time information can reach users effectively using multicast. Bandwidth demands vary from constant-bit-rate to bursty (if event driven) traffic.

#### 1.2.1.2 Many-to-Many (M-to-M) Applications

In many-to-many (M-to-M) applications two or more of the receivers also act as senders. In other words, M-to-M applications are characterized by two-way multicast communications.

The M-to-M capabilities of multicast enable the most unique and powerful applications. Each host running an M-to-M application may receive data from multiple senders while it also sends data to all of them. As a result, M-to-M applications often present complex coordination and management challenges.

a) **Multimedia Conferencing:** Audio/Video and whiteboard comprise the classic conference application. Having multiple data streams with different priorities characterizes this type of application. Co-ordination issues, such as determining who gets to talk when, complicate their development and usability. There are common heuristics and "rules of play", but no standards exist for managing conference group dynamics.

**b) Synchronized Resources:** Shared distributed databases of any type exchange information on schedules, directories etc in M-to-M way.

c) **Concurrent Processing:** Distributed parallel processing is an M-to-M process where in multiple processors collaborate with each other.

d) **Collaboration:** In shared document editing many users can edit the same document simultaneously.

f) **Chat Groups:** Chat group applications allow many users exchange information.

g) **Distributed Interactive Simulations [DIS]:** In this kind of applications, each object in simulation multicasts descriptive information (e.g., telemetry). So all other objects can render the object, and interact as necessary. The bandwidth demands for these can be tremendous, as the number of objects and the resolution of descriptive information increases.

h) **Multi-player Games:** Many multi-player games are simply distributed interactive simulations and may include chat group capabilities. Bandwidth usage can vary widely, although today's first-generation multi-player games attempt to minimize bandwidth usage to increase the target audience (many of whom still use dial-up modems).

#### 1.2.1.3 Many-to-One (M-to-1) Applications

Unlike the one-to-many and many-to-many application categories, the many-to-one (M-to-1) category does not represent a communications mechanism at the IP layer. M-to-1 applications have multiple senders and one (or a few) receiver(s), as defined by the application layer. M-to-1 applications have many scaling issues. Too many simultaneous senders can potentially overwhelm receiver(s), a condition characterized as an "implosion problem.

a) **Resource Discovery:** Service location, for example, leverages IP Multicast to enable something like a "host anycasting service" capability [AnyCast in RFC 1546]: A multicast receiver to send a query to a group address, to elicit responses from the closest host so that they can satisfy the request. The responses might also contain information that allows the receiver to determine the most appropriate (e.g., closest) service provider to use.

b) **Data Collection:** This is the converse of a 1-to-M "monitoring" application described earlier. In this case there may be any number of distributed "sensors" that send data to a data collection host. The sensors might send updates in response to a request from the data collector, or send continuously at regular intervals, or send spontaneously when a pre-defined event occurs.

c) **Auctions:** The "auctioneer" starts the bidding by describing whatever it is for sale (product or service or whatever), and receivers send their bids privately or publicly (i.e., to a unicast or multicast address) in auction applications.

d) **Polling:** In this kind of applications, the "pollster" sends out a question, and the "pollees" respond with answers.

### 1.2.2 IP **Multicast**

*IP Multicast* service [8, 23-31, 33] is proposed as an efficient multi-packet delivery mechanism. IP *Multicast* is a bandwidth conserving technology that reduces traffic by simultaneously delivering a single stream of information to thousands of corporate recipients and homes. To reduce the complexity and address the shortcomings of *IP Multicast,* SSM [32] - *Source Specific Multicast* has been developed. *IP Multicast* is designed at a very low level as a network primitive and is already deployed over major portions of Internet and is an inherent part of IPv6. *IP Multicast* delivers datagrams to logical addresses known as multicast groups identified by Class-D addresses. IGMP[115] protocol manages the group management and routers construct distribution trees for each group by using multicast routing protocol like DVMRP[116]. Though, majority of the routers on Internet are *IP Multicast* enabled, very few applications take advantage of the same, as not many developers are aware of multicast API and are conversant with network technology.

### 1.2.3 Application Layer Multicast

Application layer multicast protocols, construct overlay network for efficient data transmission. E-mail distribution and USENET news distribution are early Internet applications that fall into the broad general category of *Application Layer Multicast. Application Layer Multicast* protocols do not change the network infrastructure. These protocols implement multicast forwarding mechanism at the end hosts. These kinds of protocols are being designed extensively for content distribution networks. Unlike native multicast, where data packets are replicated at routers inside the network, in application layer multicast, data packets are replicated at **end** hosts. Many researchers are finding ways to push the multicast routing capabilities onto hosts than onto routers. The idea here is to construct an overlay network of hosts, on top of routers. Routers do normal unicast routing, while the hosts attached to them will take care of multicast group management and routing among themselves. While *IP Multicast* protocol works at network layer, that requires applications to be aware of multicast and programs [76] be written specifically using *IP Multicast* API; protocols like *End-system Multicast* [1-2], *Scattercast* [3-4] and *Overcast* [5] require an application-layer overlay infrastructure to be laid of. All these protocols, share the common goal of providing the benefits of multicast without requiring direct router support or the presence of a physical broadcast medium [4].

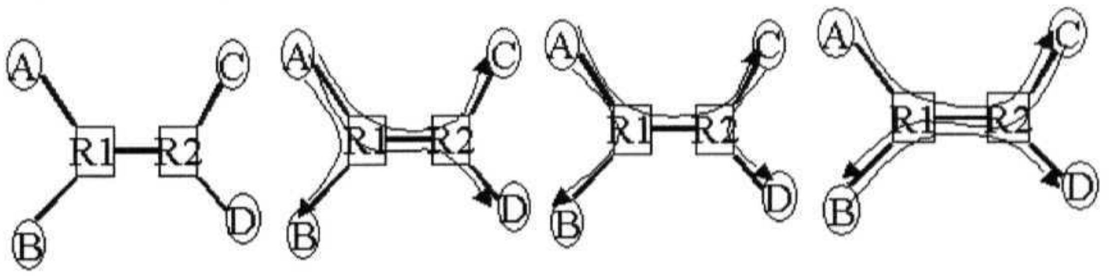
## 1.3 Problem Definition

We aim at improving the network performance on the existing inter network infrastructure. We mainly look at improving performance and simplifying the multicast application development by taking four different tasks.

### 1.3.1 Exploiting the Topology

It is quite evident that when a host has to send information to multiple stations, the packets travel over common links. The figure 1.1 shows unicast and overlay multicast topologies.

**Figure 1. 1: Overlay Network Illustration**



**la.** Network 1b. Unicast 1c.Overlay1 1d .Overlay2

A->B; A->C; A->D A->B; A->C; C->D A->C;C->B;B->D

The goal of multicast is to reduce the redundant packet movement. While there are many overlay multicast protocols like [ 11 ]*DT Protocol*, CAN[18-19] etc that do not consider underlying network topology, we create an overlay in alignment with the underlying topology, such that the over all link utilization is optimum. For instance, in figure-1, the **link** R1-R2 is very costly, and if the overlay2 is created **as** A->C; C->B and B->D then **this** overlay becomes more inefficient compared to unicast. This is because, in unicast, the R1-R2 link is utilized twice, whereas in overlay it is used thrice.

We propose to build an overlay such that the over **all** link utilization in the overlay is kept optimal i.e., from the network perspective, the constructed overlay must ensure that redundant transmission on physical links is kept minimal. In overlay multicast, hosts take over the routing functionality of normal network routers and forward the packets among the participating hosts. The topology-building algorithm has to consider each joining hosts' capacity and each host can specify to how many hosts it can forward the data in the overlay. As the overlay grows, the delay increases for the data to reach from the source to the destination, as the number of intermediary routing hosts increase. So, each host can specify how much delay it can sustain. The protocol must allow for the overlay to incrementally evolve into a better structure **as** more information becomes available.

### 1.3.2 Exploiting Media Characteristics

The Transmission Control Protocol/Internet Protocol (TCP/IP) standard is the dominating protocol in the communication world. TCP/IP standard has achieved this dominance, and become a de facto standard for any network, because its superiority in handling high-speed data traffic has been proved over a long period in the largest global network: the Internet.

In principle, broadcasting via satellite is simpler than for terrestrial networks, which are not naturally broadcast networks. However, TCP/IP has shielded this nature. TCP/IP follows a layered approach in which the lower level protocol details are shielded by high-level protocols. Broadcast nature of the media, is a physical line property, which falls into physical layer in OSI layered model that naturally gets shielded by TCP - transport layer i.e., 1000 hosts, which are connected by the same satellite media, though can communicate to all by just transferring the data once, due to lack of availability of this specific media property to the application developers, applications are written to transfer the data 1000 times as if the hosts are connected through 1000 different network links.

Geo-stationary satellite systems have a fixed **round** trip delay of 600ms that cannot be avoided. The problem with TCP/IP over satellite links is that the protocol will not let data be sent, beyond a certain limit or window, without an acknowledgement from the receive end. Thus if the traffic is affected by latency, the system has to permit the data flow to increase, although unacknowledged, and allow the receiver to control it. This control is done by using a receive buffer whose parameters are sent to the transmit end during the set up phase so that it can adjust the traffic flow accordingly. The buffer size is reduced each time data is transmitted, so that when it is empty no more data can be sent without a TCP acknowledgement. The window data rate limit is set because the TCP/IP only has 16 header bits available to describe the packet size, which sets a maximum throughput of 216 bytes, corresponding to 840kbit/s. Satellite vendors using a technique called 'TCP Spoofing' have solved this latency problem of satellites communication.

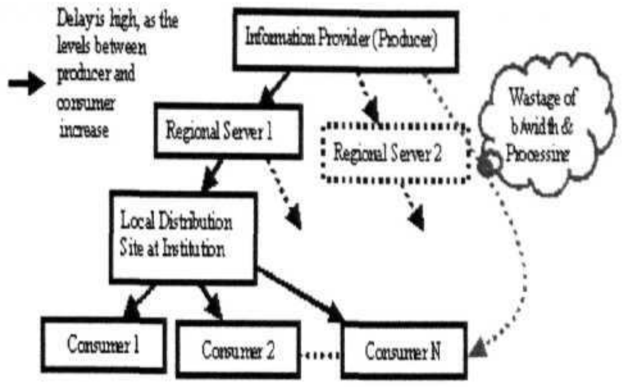
While there are many vendors improving TCP efficiency over satellite for only specific applications like video broadcast etc, there is no way for a general application developer to exploit the broadcast property in his application. We propose a method 'Multicast Spoofing', using which we pass on the broadcast benefit to any application.

### 1.3.3 Exploiting Query Redundancy

Search tools till date have insufficient capabilities to keep pace with the information generated. Particularly getting right and relevant information has become a nightmare. In this scenario, SDI - *Selective Dissemination of Information*, a popular concept used by libraries is gaining importance. In SDI, users register with servers that are nearer to them with their interests, which are stored as profiles. Based on the profiles, the servers filter right information and push the same to the user. A centralized server cannot scale up to the requirements of large number of users spread all over Internet.

Overlays can help meet this requirement, however, we have to devise ways to reduce the latency/delay introduced by overlays. Figure 1.2 shows an example of overlay. Stream processing allows data to be processed by intermediate servers as data streams in thereby not adding any delay at processor side. Using overlays, we can register profiles with the servers nearer to the users (consumers) and these servers in turn send the combined profiles to the servers above them. This profile grouping and sending can further go up the tree till the producer server.

**Figure 1. 2: Information Flow in a Typical Overlay**


Delay is high, as the levels between producer and consumer increase
Information Provider (Producer)
Regional Server 1
Regional Server 2
Wastage of b/width & Processing
Local Distribution Site at Institution
Consumer 1
Consumer 2
Consumer N

Whenever the producer sends information, this has to be filtered as per their profiles at each server at lower levels of the tree, till it reaches the consumer. Data is exchanged in the form of XML[80] in SDI systems and each profile is represented in the form of an XPath[78] query and we filter the information based on multiple profiles (queries), registered at each server.

When thousands of profiles are grouped together, naturally there will be lots of commonality in the kind of information in which the users are interested. Our objective is to exploit this redundancy such that while filtering the information as per the profiles, we need not traverse the same information for thousand times and rather we should filter the information for all users in just one pass.

### 1.3.4 Generic Application Development Framework

Our aim is to provide broadcast/multicast advantage to any kind of application that sends same information to many receivers, without writing separate applications - one for broadcast and the other for unicast. Unlike point-to-point communication, where sender and receiver only co-operate, in broadcast communication sender and many receivers must co-operate. Naturally, not all receivers are in homogeneous environment. Lot of following for general applications protocols like SMTP, HTTP, FTP, Telnet etc made easy for deployment on any platform because almost every vendor has a product for it that can interact with other similar products just because all of them follow protocol specifications and API. But the same is not the case for other business specific applications. Applications written for one platform are now not able to communicate with other applications on other platforms due to many reasons like dependency on specific - language, communication protocol, database etc. Therefore we looked for a platform independent distributed computing environment. We rely on standard SOAP [47-49] - *Simple Object Access Protocol* specification that is fast becoming the standard middleware to provide a generic simple to use architecture for multicast applications.

## 1.4 Contributions

The main contribution of this thesis is to give a simple DNS kind of infrastructure based solution for multicast services. We emphasize that multicast can better be provided at application level rather than at router level.

**Application Layer Multicast (Appcast):** We propose a new application layer multicast that exploits the topology information. Any application layer multicast protocol that does not exploit topology may in fact decrease the efficiency in terms of bandwidth and increase delay. We develop new algorithm Appcast that creates a multicast overlay topology, which we further optimize by two more extensions to it taking delay and processing power as parameters.

**Advantage Broadcast Media:** The application developers are not able to take advantage of the media nature like broadcast for satellite networks. We show ways of how the broadcast nature can best utilized by developers.

**A Generic Application Architecture for Multicast Services:** We design generic application architecture and develop three different kinds of applications. We take 'scheduled file pushing' application as one-way multicast application, 'database replication' as event and rule based one-way multicast application and 'auctions' as interactive multicast application and implement all three.

**Xpath Query Processing (YALXP):** Hosts are organized as an overlay in application layer multicast. Overlays disseminate information in a hierarchical and incremental fashion. Also, not all members in an overlay or multicast group need exactly same information. In this case, if **we** can filter the information at different levels **of overlay, while the** information is distributed, we can minimize the bandwidth and processing requirement. Each member's interests are represented by XPath[78] query and the information to be distributed is in the form of XML[80] document as XML is gaining importance as the standard for information exchange. We look at the XML document-processing problem particularly in the context of application layer multicast and propose a new XPath processing algorithm called YALXP. YALXP algorithm processes multiple XPath queries over a XML document in single document traversal i.e. in one pass it can answer multiple queries.

## 1.5 Thesis Outline

We organize the thesis as follows. Chapter 2 does a survey on already proposed application layer multicast protocols. It classifies them based on criteria like type of algorithm - centralized, distributed; type of topology created - tree, mesh etc; and analyses their merits and demerits. In Chapter 3 we propose a new algorithm Appcast. We simulate and compare the performance of Appcast and other closely related application layer multicast protocols like TAG [22], NICE [15-17] and HMTP [13]. Also, Chapter 3 studies various ways of enhancing the TCP/IP over broadcast networks and details the approach we proposed i.e., 'multicast spoofing'. Chapter 4 deals with generic multicast application architecture design, development and implementation. In Chapter 5, we propose new XPath[78] processing algorithm YALXP. In this chapter we surveyed the existing XPath processors, simulated algorithms like XAOS [79] and compared them with our algorithm. We conclude the thesis in Chapter 6 with discussion on future work,

# Chapter 2

# Application Layer Multicast Protocols' Survey

In this chapter, we discuss various existing *Application Layer Multicast* protocols. Many protocols have been proposed in the recent past, with ESM - End System Multicast [1-2] being the first. Every *Application Layer Multicast* protocol constructs an overlay on basic network, manages this overlay and transmits information on this overlay. The various protocols that have been proposed differ in three basic criteria - a) the way they create the overlay i.e., topology building algorithm; b) the exact overlay topology they create i.e., mesh, tree, hierarchy; and c) their knowledge about the underlying network topology i.e., fully aware or partially aware or not at all aware of the network. This chapter surveys *Application Layer Multicast* protocols based on these three criteria and groups them into different categories. Section 1 introduces the *Application Layer Multicast* with its definition and properties. Section 2 deals with how we classified *Application Layer Multicast* protocols into various groups and section 3 details specific *Application Layer Multicast* protocols from each category.

## 2.1 Introduction

Multicast applications are implemented as multiple point-to-point applications i.e. an application logically requiring multicast must send individually addressed packets to each recipient. This has two drawbacks – 1. The source should know the addresses of all recipients and 2. Transmitting multiple copies of the same packet results in inefficient usage of sender's resources and network bandwidth. Unfortunately, TCP/IP shields the underlying network topology that facilitates multi-destination delivery. For example, local area bus, ring, radio networks and even satellite based wide area networks provide multicast facility directly.

Unicast is completely impractical due to its redundant use of link bandwidth, when thousands of receivers have to receive the same data. The benefits of multicast in terms of bandwidth efficiency are quiet often outweighed by the control complexity associated with group setup and maintenance. *Application Layer Multicast* protocols normally follow two steps to achieve multicast capability. 1. Arrange receivers into an overlay network of unicast connections and 2. Construct efficient data distribution trees

over this overlay network. *Application Layer Multicast* differs from *IP Multicast* in that the multicast support is done at host systems instead of core routers. Host systems implement all multicast functionalities like membership management, packet replication and data distribution. *Multicast* in host group model has been defined in [8] as "a host group is a set of network entities sharing a common identifying multicast address, all receiving any data packet addressed to this multicast address by senders that may or may not be members of the same group and have no knowledge of the group's membership"
This definition characterizes the following points for the term *multicast.*

- There must be a unique address to represent the multicast group members.
- All members receive all packets.
- Any member can join the group openly, even without the knowledge of the source.
- Any host can send data to the group irrespective of its membership in the group i.e., non-members also can send data.

*Application Layer Multicast* differs from traditional *multicast* in many respects as given below.

- Hosts take over the role of routers for multicasting.
- Within an overlay topology, the underlying physical topology is completely hidden.
- In traditional multicast, the membership knowledge is distributed in the multicast routers. In *Application Layer Multicast,* group members are known either by a rendezvous point (RP), the source, or every body or distributed among members.
- The application layer topology is potentially under control.
- The group membership can be under control.
- The precise topology relationship among hosts can control who receives data from whom, and thereby eliminating DOS attacks, which cannot be controlled in traditional *IP Multicast* by its own definition.

Figure 2.1 shows the distinction between *Unicast, IP Multicast* and *Application Layer Multicast.*

Figure 2. 3:

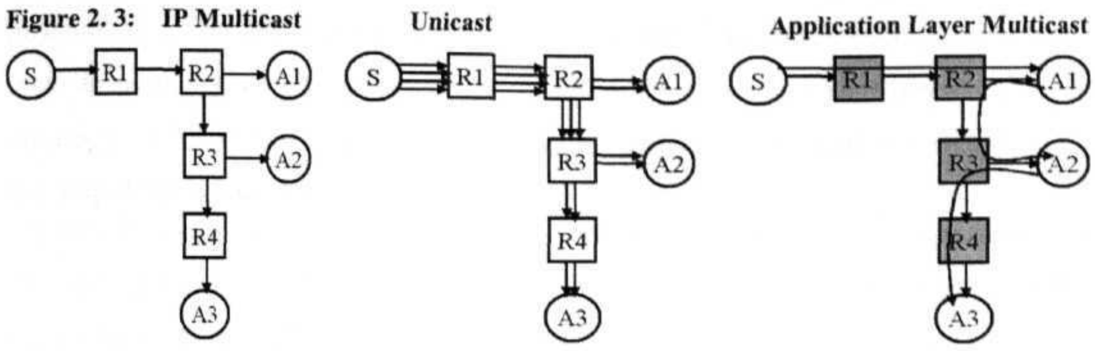


Figure 2.1 clearly shows that in case of IP Multicast, packets from source (S) to all hosts (Al, A2, A3), traveled only once on any link. In case of unicast links and routers nearer to source experienced redundant packet movement. In case of Application Layer Multicast, though it does not achieve the performance like IP Multicast, it shows it is better than unicast.

The performance of an *Application Layer Multicast* can be based on two parameters - *stress* and *stretch* [15-17]. *Stress* is defined per host or per link as 'the number of duplicate packets that a host sends or duplicate packets that travel over a link in an overlay'. *Stretch,* also called as '*relative delay penalty'* is defined per pair of hosts i.e., source and destination as the 'ratio between the number of hops a packet travels from source to destination in unicast mode to the number of hops the packet travels in multicast'. In case of IP Multicast, all links experience unit stress and every pair of hosts experiences unit stretch. In case of unicast, in figure 2.1, the maximum stress is 4 on links S-R1 and R1-R2. Stretch in unicast is always 1. In case of *Application Layer Multicast,* from the figure 2.1, maximum stress is 2 on links R2-A1 and R3-A2; and maximum stretch is 9/4 i.e., 2.5 for the pair S and A3. While unicast shows maximum redundant packet movements on links nearer to source, *Application Layer Multicast* shows redundant packet movements on links that are nearer to destinations.

**Classification of Proposed Application Layer Multicast**

The existing *Application Layer Multicast* protocols can be classified based on three criteria. 1. Knowledge of underlying network topology. This criterion is about the protocol's awareness of topology to align the physical network topology with overlay topology. 2. Topology building algorithm e.g., distributed or centralized. In a centralized algorithm, a master host takes decisions about the topology construction, by keeping the physical topology information with itself. In distributed algorithm, every joining host will take its own decision of joining other hosts by its own knowledge about the physical topology. 3. Topology created by the protocol e.g., shared tree, mesh and tree etc.

## 2.2 Topology Awareness

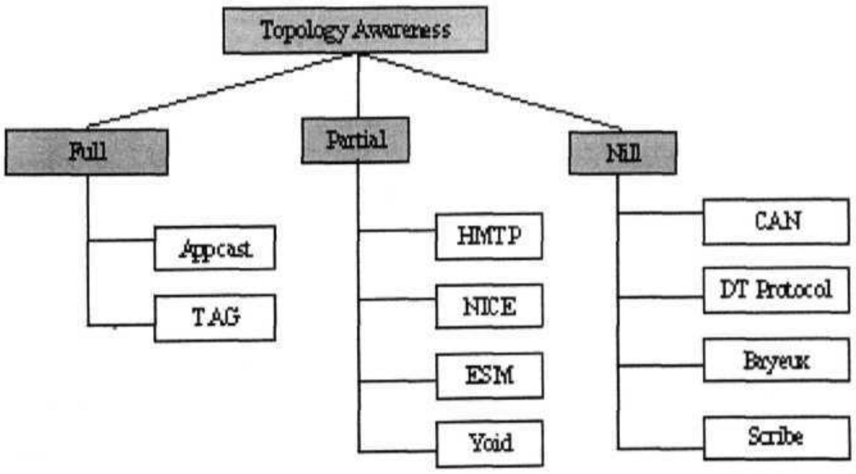
*Application Layer Multicast* is performed at end hosts, automatically by application code. Performing multicast at end hosts incurs performance penalty, as end hosts do not have the routing information available unlike routers. Unless the overlay topology matches the underlying network topology, the performance would be so degraded that unicast will out perform the gains of multicast. To map overlay topology to the underlying network topology, many *application layer multicast* protocols rely upon end-to-end measurements to infer network metrics upon which the overlay multicast tree is built. Many a times, the network metrics are based upon rtt (round trip time), traceroute (to get the path), ping (TTL scoped packets) etc. Though, these methods can give some information about network topology, the *application layer protocol* cannot get the holistic view of the entire network topology.

Based on topology awareness criteria, the proposed *application layer multicast* protocols are classified as a) *Fully aware* b) *Partially aware* and c) *Nill* or *not aware.* An *application layer multicast* protocol is '*fully aware'* of the underlying network, if it is implemented on network controlled by a single authority, e.g., corporate networks, content distribution networks etc. Over Internet, projects like Topology Server [9] are being implemented that collect entire network information which can be used by an *Application layer multicast* to align its overlay with the underlying network. An *Application layer multicast* protocol is '*partially aware*' of the underlying network if it just guesses the physical connectivity of the routers based on methods like 'rtt' – round trip time; 'ping' - packet internet groper; 'traceroute' - tracing the path between hosts; 'TTL' - time to live scoped messages etc. For example, NICE [15-17] sends TTL scoped messages among the multicast group members to infer the multicast distances among them and uses this information to create 'parent - child' relationships among the members. An *Application layer multicast* protocol is '*not aware'* of underlying network topology, if it does not consider any network property like hops, delay, bandwidth etc.,

**but rather** considers information like *public key* to create an unique member identity like IP address, for each multicast member. This unique member identity is used to create the overlay topology and distribute the information on this overlay. It may happen that two members who are actually connected by the same router, get such different identities that the information exchange between the two will actually take more time on the overlay topology compared to the time taken when transmitted on physical network topology.

TAG[22] and Appcast (explained in detail in chapter 3) are *fully aware* of network topology as they depend upon Topology Server[9] that uses network management kind of solutions like polling the routers. HMTP [13], NICE [15-17], ESM [1-2], Yoid [20-21] are *partially aware* of the topology by depending on methods like trace route, ping etc. These methods construct topology information on their own unlike Appcast and TAG [22] which depend on Topology Server [9]. CAN [18-19], DT Protocol[11], Scribe[14] and Bayeux [12] are *not aware* of topology at all, as they basically construct overlay topology based on unique member identities that are created randomly. For example, Scribe and Bayeux give separate identity numbers to hosts, based on the hosts' unique parameters like IP address, Public Key etc. CAN [18-19] and DT Protocol [11] map the hosts into a geometric space by assigning unique random numbers to each member.

**Figure 2.4: Classification Based on Topology Awareness**



## 2.3 Topology Building Algorithm

At the heart of *Application Layer Multicast* protocols, is the overlay topology they create. The topology is created as members/hosts join the multicast group. *Application Layer Multicast* topology building algorithms define a definite relationship among the participating members and thereby create topologies like tree, mesh, hierarchy etc. The relationship can be parent-child, host-neighbors, cluster member - cluster leader etc. The topology building algorithms differ in the way they build the topology and can be classified as 1 .Centralised and 2.Distributed.

The overlay topology is created as the members join the multicast overlay. There has to be a mechanism also called as *'rendezvous mechanism'* using which the joining member can contact the members of the overlay and become part of the overlay. This *'rendezvous mechanism'* can be in three ways. 1. Broadcast Mechanism: joining member broadcasts a message containing its willingness to join the overlay and interested overlay members respond to the message. Even for overlay members, when their relationship with immediate parent breaks due to member leave or link failure, to select a new parent, they follow broadcast mechanism. 2. Buddy List: joining members keep a list of 'likely' members of the overlay and contact members from this list. In case of link failures or member leave, the affected overlay members use this list to join the overlay. 3. Well known server: joining members contact well known server to learn about overlay members and join the overlay. *Application Layer Multicast* protocols that follow the 'broadcast mechanism' or 'buddy list mechanism also termed as limited broadcast' can be termed as 'distributed application layer multicast' protocols and those who follow the 'well known server mechanism' can be termed as 'centralised application layer multicast' protocols. While distributed protocols utilize more bandwidth due to their broadcast mechanisms, centralized protocols suffer from the single point failure of the central server.

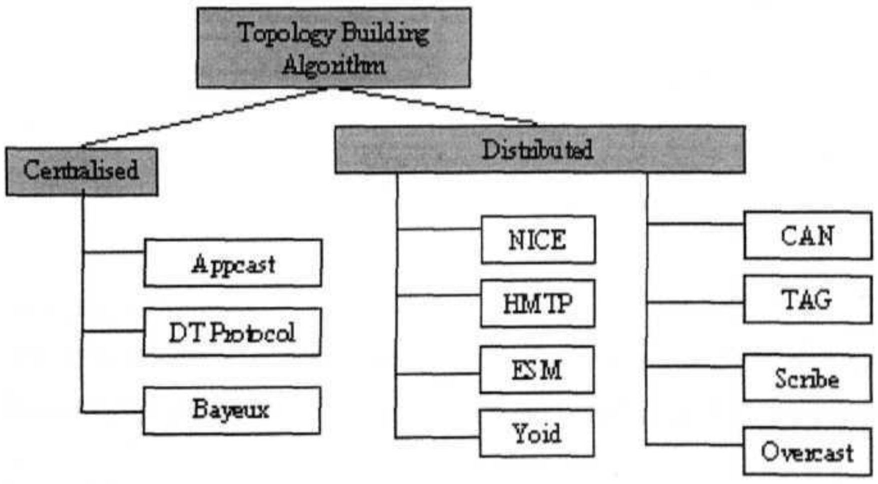
### 2.3.1 Centralised Algorithm

In centralized algorithms, a central server decides the place of a joining host in the topology. A new joining host should contact the central server to know about its own position and its relationship like parent-children etc., with its immediate neighbors. Appcast (explained in chapter 3), DT Protocol [11] and Bayeux [12] fall into this category. In DT Protocol, new hosts join the overlay by sending a request to 'DT Server'. The server responds with the logical and physical addresses of some host that is already in the overlay network. The new host then sends a message to the host identified by the server and thus establishes its relationship with the overlay. In Bayeux, the joining host sends a 'JOIN' message to 'Root' i.e., central server. 'Root' replies with a 'TREE' message. While the 'TREE' message is traveled towards the joining host, all in between hosts include the joining host's ID in their list of hosts for whom they have to route the information, creating a routing path from the root to the joining host.

## 2.3.2 Distributed Algorithm

In distributed algorithms, all joined hosts guide the new joining host, to know its place in the topology and relationships. The onus of contacting the already joined members and knowing its own position is on the joining host only. NICE[15-17], Yoid[20-21], HMTP[13] etc. fall into this category of protocol as shown in figure 2.3. In Narada, every host contacts a common host (called RP - Rendezvous Point) before joining the group. This common host returns addresses of a set of already joined hosts. New host contacts them randomly to find whether any one will accept it to become its *neighbor.* The one who accepts it first will be given first priority. In NICE, 'root' periodically sends a heartbeat packet to all overlay members, from which each member infers their distance to 'root'. Similarly, each parent sends messages to its children from which children infer their distance to parent. Using this information, members reorganize themselves in the overlay.

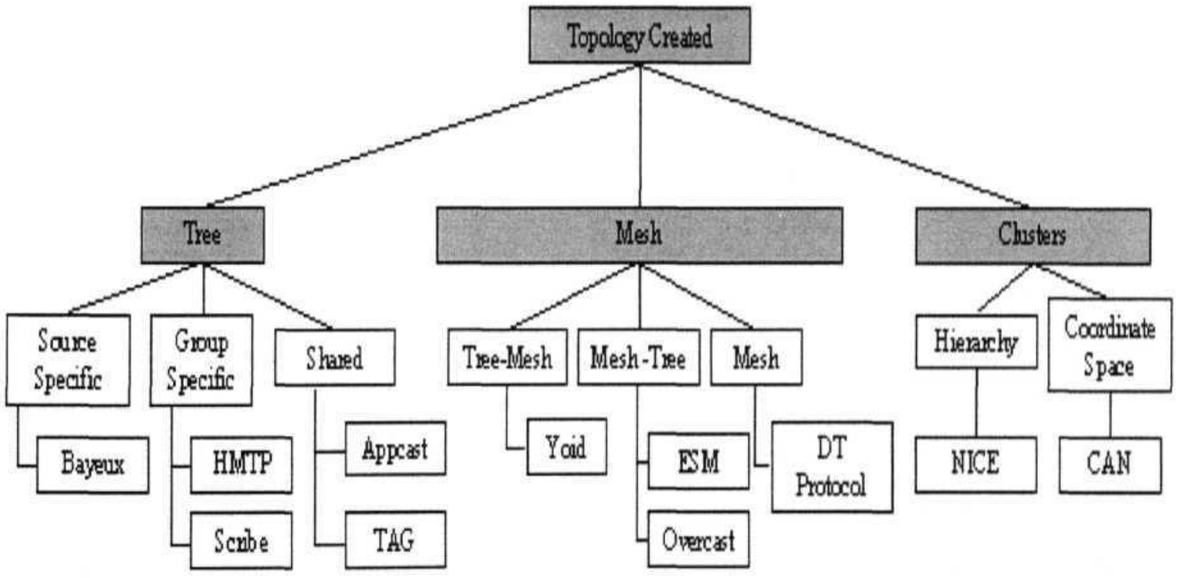
Figure 2.5: Classification Based on Topology Building Algorithm



## 2.4 Topology Created

*Application layer multicast* protocols create and manage the overlay topologies to efficiently route the data packets. Based on the topology they create, the proposed protocols can be classified as those who create a. Tree, b. Mesh or c. Clusters. All *Application Layer Multicast* protocols that create 'Tree' topology ensure that there be no loop in the path between any pair of participating nodes. In contrast, the protocols that create 'Mesh' topology look for alternate paths between nodes. In case of 'cluster' topology, the members are grouped in clusters and one of these members acts as a cluster leader. The cluster leaders are grouped again into clusters at higher level and one of them acts as the cluster leader at the level. The cluster grouping into levels continues, till only one member becomes the leader of the entire cluster hierarchy. The general purpose of creating a topology is 1 .To distribute the data packets and 2.To send control information to manage the topology. Protocols like HMTP [13], TAG [22], Appcast use the same topology for both the purposes, while others like Narada [1-2], Yoid [20-21] use separate topologies like tree and mesh.

Figure 2. 6: Classification Based on Topology Created



### 2.4.1 Mesh topology

End System Multicast (ESM)/Narada [1-2], Scattercast [3-4] and Yoid [20-21] fall into this category. Data topology is tree in all these systems. ESM constructs mesh first as the

**basic overlay** network and then constructs separate trees for each source. Narada(ESM) and Scattercast construct trees in a two-step process. First they construct efficient meshes among participating members and then construct spanning trees of the mesh using well-**known** routing algorithms. Yoid constructs tree first and then constructs mesh to support **this tree.** Unlike ESM, Yoid uses the single shared tree for all sources for data dissemination. DT protocol [11] uses mesh for both data and control topologies.

### 2.4.2 Tree

Protocols like Overcast [5], TAG–Topology aware group communication [22], ALMI–Application Level Multicast Infrastructure [10] and HMTP-Host Based Multicast [13] create shared tree. Single shared trees are not optimized for the individual source and are prone to failures as a single point of failure can break the tree into major partitions. In case of mesh, a link failure can with stand due to alternate paths. Only multiple failures can cause the mesh to partition. The single shared tree approach, justifies its method from the fact that the underlying network topology already takes care of multiple paths. So one should consider node failures than the link failures. An efficient way is to maintain multiple trees for each source, but that requires multiple overlay topologies, which is complex. The application layer multicast protocols that create trees as their basic topologies differ in the number of trees they manage per source, per group or a single shared tree.

### 2.4.3 Clusters

Protocols like CAN and NICE group the members into clusters. Each cluster is assigned a cluster leader. The cluster leaders in turn are grouped and one of the cluster leaders is assigned as leader of this group. This process repeated till there exists only one leader on top of the grouped cluster leaders, generating a hierarchy topology. NICE arranges set of end hosts into a hierarchy. The hierarchy implicitly defines data path. Each member maintains soft state information about other hierarchically nearer members and has only limited knowledge about other members. In NICE, all members belong to Layer 0. Members are grouped into clusters with size K and 3K where K>1. For each cluster, one of the cluster members acts as a leader and enters into higher layers.

## 2.5 Well-known Application Layer Multicast Protocols

In this section, we describe certain well-known protocols in detail, to support our classification discussed in previous section.

### 2.5.1 ESM - End System Multicast / Narada

ESM [1-2] is one of the first application layer multicast protocol. Narada constructs an overlay structure among participating end systems in a *self-organizing* and *fully distributed* manner. Narada creates an overlay mesh topology on top of router topology connecting all hosts that participate in a multicast session. Once the mesh is created, any one from the group can send messages to every one else in the group. For this, Narada uses existing multicast protocols like DVMRP – Distance Vector Multicast Routing Protocol. Narada has protocols for host join, mesh partition repairing, adding/deleting mesh links for better performance.

**Member Join:** When a member wishes to join a group, Narada assumes that the member is able to get a list of group members by an out-of-band bootstrap mechanism. The list needs neither be complete nor accurate, but must contain at least one currently active member. New host contacts members in list, randomly to find whether any one will accept it to become its *neighbor*. A host is a *neighbor* to another host, if they have a direct connection on the overlay. It repeats this process till some host accepts it as a neighbor. The one who accepts it first will be given first priority. Having joined the group, the new member exchanges refresh messages with neighbors who in turn exchange with their neighbors. This process keeps on going till every one in the group is aware of the new member join.

**Member Leave:** When a member leaves a group, it notifies its neighbors and this information is propagated to the rest of the group members along the mesh.

**Control Messages:** The group membership information keeps flowing across all members of the group periodically. Every host keeps state information about every other host.

The complexity of Narada algorithm is $O(N^2)$, where N is the number of hosts. Narada cannot scale for larger groups. As the group's size increases, the number of overlay hops between any pair of members' increases and hence the delay between them potentially increases.

### 2.5.2 Yoid - Yet Other Internet Distribution Protocol

Yoid [20-21] generates two topologies per group - a shared tree topology and a mesh topology. The tree topology is for data distribution and the mesh topology is to support the tree from not breaking. While Narada creates mesh first and then tree; yoid creates tree first and then mesh. Yoid has many protocols on top of TCP and UDP like YIDP - Yoid Identification Protocol, The Yoid Transport Layer - yTCP, yRTP, yMTCP etc., and Yoid Distribution Protocol - YDP.

**Yoid Tree:** In Yoid, every host is responsible to find its parent. A new host gets a list of potential parents from a common host (RP). A new host selects one as its parent if the parent-child link does not cause a loop. If a new host finds many parents, it will select best based on some metric of interest like number of hops from parent to child or number of children the parent already has etc. If new host could not get any parent, it declares itself as parent and informs RP. RP arbitrates and merges the tree partitions into a single tree.

**Yoid Mesh:** After constructing the tree, Yoid proposes to have multiple paths among members such that they will act as back up. So each host randomly selects few more members in the group, which are not its immediate neighbors and constructs links in a different path. These additional paths will give strength to Yoid in case of failures.
Yoid has many disadvantages like Yoid is too complex, the specifications are too descriptive and lengthy, too many protocols and APIs and all these issues will actually burden the whole set up.

### 2.5.3 HMTP - Host Multicast

HMTP [13] creates group specific tree topology as the multicast overlay topology. In HTMP, each multicast group requires a *Host Multicast Rendezvous* point that acts as a contact point for new members to join the group. HMTP Clusters nearby members together. Members choose their parent closer to them. Following steps describe the HMTP protocol.

1. New member sets the root as potential parent (PP) and contacts PP.
2. Query PP to discover all its children and measure its nearness to PP and PP's children.

3. Find the nearest member among the PP and PP's children except those marked as invalid. If all of them are marked as invalid, pop the top element from stack, set it as PP and return to step 2.
4. If the nearest member is not current PP, push current PP onto stack; set the nearest member as new PP and return to step 3.
5. Otherwise send PP join request. If PP accepts it as a child, it becomes child of the PP; if rejected mark PP as invalid and return to step 3 (PP may not accept it as its child due to many reasons like - out degree); otherwise parent found and so establish unicast path.

HMTP proposed member leave, link failure and improvement algorithms also. In HMTP [13], every member keeps track of every other member that falls in the path of member and root.

### 2.5.4 NICE - Nice Internet Communication Environment

NICE [15-17] claims, relatively small control overhead. Its motivation is actually from key distribution in a secure group communication. NICE arranges set of end hosts into a hierarchy. The hierarchy implicitly defines data path. Each member maintains soft state information about other hierarchically nearer members and has only limited knowledge about other members. In NICE, all members belong to Layer 0. Members are grouped into clusters with size K and 3K where $K>1$. For each cluster, one of the cluster members acts as a leader and enters into higher layers. A member is part of L,, if it is leader in all $L_{i-1}$ levels. A cluster leader has minimum maximum distance from all of its members. A host belongs to only a single cluster at any layer. If a host is not present in layer $L_i$, it cannot be present in any layer $L_j$, where $j>i$. For a group size N, and cluster size K, there can be at most $\log_K N$ layers. Each member maintains information about every other member of its own cluster in all of its layers.

NICE [15-17] constructs an overlay tree, before it clusters the group members and arranges them into a hierarchy. NICE constructs an overlay tree based on the underlying network topology. Next, it uses a clustering protocol to group the members into clusters of size K to 3K-1, where K is a constant by traversing the overlay tree bottom up. This clustering is basically to reduce the depth of the tree and to keep control overhead cost to be constant. As the cluster size increases, unicast with in the cluster may increase. NICE [15-17] does not give flexibility to the joining member, to choose its leader.

**Member Join:** When a new host joins the multicast group, it must be mapped to some cluster in layer First, it contacts the RP with its join query. The RP responds with the hosts that are present in the highest layer of the hierarchy. The joining host then contacts all members in the highest layer to identify the member closest to it. Once a nearest host ($CL_i$) is found in a layer, the joining host contacts all child members of CL, in (i-1) layer, to identify the member nearest to it. This process is repeated till it gets a cluster to join as a member.

**Member Leave:** When a host leaves the multicast group, it sends a *Remove* message to all clusters to which it is joined. This is a graceful-leave. However, if host H fails without being able to send out this message, all cluster peers of H detects this departure through non-receipt of the periodic *HeartBeat* message from H.

**Cluster Split and Merge:** A cluster-leader periodically checks the size of its cluster, and appropriately splits or merges the cluster when it detects a size bound violation. A cluster that just exceeds the cluster size upper bound, 3k-1 is a split into two equal-sized cluster. Subsequently, an equal-sized split would create two clusters of size k each. However, a single departure from any of these new clusters would violate the size lower bound and require a cluster merge operation to be performed.

### 2.5.5 CAN - Content Addressable Network

CAN[18-19] is designed for large groups. While other methods are towards data dissemination initiated from a single source, this method also tries to achieve interactive group games. CAN[18-19] maintains a logical d-dimensional co-ordinate space, with no relation to physical co-ordinate system or network topology. The entire co-ordinate space is dynamically partitioned among all the nodes. Every node, owns its individual unique zone with in the overall space. The new node must find a node already in the CAN[18-19]. Using the CAN[18-19] routing mechanism, it must find a node, whose zone will be split. The neighbors of the split zone must be notified so that routing can include the new node.

**Member Join:** To find a zone, new node randomly chooses a point (x,y). It sends *join* request destined for point (x,y). This message is sent into the CAN via any existing

CAN node. Each CAN node then uses the CAN routing mechanism to forward the message, until it reaches the node in whose zone (x, y) lies. This current occupant node then splits its zone in half and assigns one half to the new node. Having obtained its zone, the new node learns the IP addresses of its coordinate neighbor set from the previous occupant. This set is a subset of the previous occupant's neighbors, plus that occupant itself. Similarly, the previous occupant updates its neighbor set to eliminate those nodes that are no longer neighbors. Finally, both the new and old nodes' neighbors must be informed of this reallocation of space. Every node in the system sends an immediate update message, followed by periodic refreshes, with its currently assigned zone to all its neighbors. These soft-state style updates ensure that all of their neighbors will quickly learn about the change and will update their own neighbor sets accordingly.

Member Leave: When nodes leave a CAN, the zones they occupied are taken over by the remaining nodes. The normal procedure for this zone take over is for a node to explicitly hand over its zone and the associated (key, value) database to one of its neighbors. If the zone of one of the neighbors can be merged with the departing node's zone to produce a valid single zone, then this is done. If not, then the zone is handed to the neighbor whose current zone is smallest, and that node will then temporarily handle both zones.

## 2.5.6 Bayeux

Bayeux [12] is an efficient, source specific, explicit join application level multicast system. It uses Tapestry [12], an application level routing protocol. Each Tapestry node has names independent of their location and semantic properties, in the form of random fixed length bit sequences represented by a common base (e.g., 40 hex digits represented by 160 bits). Bayeux uses four types of control messages in building a distribution tree - JOIN, LEAVE, TREE and PRUNE. A member joins the multicast session by sending a JOIN message towards the 'Root'. 'Root' replies with a TREE message. The actual paths taken by JOIN and TREE are different due to asymmetric nature of Tapestry unicast routing. When a router receives TREE message, it adds the new member node ID to the list of receiver node Ids that is responsible for, and updates its forwarding table. LEAVE message from an existing member triggers a PRUNE message from the root, which trims from the distribution tree any routers, whose forwarding state become empty after LEAVE operation.

**Member Join:** When a node with Id 1250 tries to join multicast session where node 7876 is the root, a JOIN message from node 1250 traverses nodes xxx6, xx76, x876, and 7876 via Tapestry unicast routing, where xxx6 denotes some node that ends with 6. The root 7876 then sends a TREE message towards the new member traversing the nodes xxxO, xx50, x250 and 1250, which sets up the forwarding state at intermediate application-level routers. Note that while both control messages are delivered by unicasting over the Tapestry overlay network, the JOIN and TREE paths might be different, due to the asymmetric nature of Tapestry unicast routing. When a node receives a TREE message, it adds the new member node ID to the list of receiver node IDs that it is responsible for, and updates its forwarding table. For example, consider node xx50 on the path from the 'root node to node 1250. Upon receiving the TREE message from the root, node xx50 will add 1250 into its receiver ID list, and will duplicate and forward future packets for the multicast session to node x250.

**Member Leave:** A node that leaves the overlay, informs the root by sending a LEAVE message. A LEAVE message from an existing member triggers a PRUNE message from the root, which trims from the distribution tree any routers whose forwarding states become empty after the leave operation.

### 2.5.7 TAG - Topology Aware Group Communication

TAG [22] uses information about path overlap among group members to construct the overlay tree. In TAG each new member of multicast group, determines the path from the root to itself and finds out its parent and children by partially traversing the overlay tree. TAG proposed complete path matching algorithm, where in a new node selects one as its parent, which shares the maximum common path with it. Each TAG node maintains a *Family Table*, with information about its parent and children. The path-matching algorithm traverses the overlay tree from root down the children, matching the paths from the 'root to new node' with the path from 'root to TAG node'. It considers three mutually exclusive cases. Let N be a new member wishing to join and C be the node being examined. Then the three cases are 1. There exists a child A of C, whose path is a prefix for the path N, with the condition that the path length of N > A > C. In this case N chooses node A, and traverses the sub-tree rooted at A. 2. There exist children $A_i$ of C, who have the path of N as the prefix, in their path. In this case, N becomes child of C, with all $A_i$ as its children. 3. In case, there's no child of C satisfying the cases 1 or 2, N becomes the child of C. As an optimization method, TAG [22] proposed partial path matching algorithm, where in, instead of matching the complete path of a new member, a predefined number of elements in the path are matched. This helps reduce the depth of the tree.

**Member Join:** A new member joining a session sends a JOIN message to the main sender S of the multicast session (the root of the tree). Upon the receipt of a JOIN, S computes the 'spath' (shortest path), to the new member, and executes the path-matching algorithm. If the new member becomes a child of S, the FT (family tree) of S is updated accordingly. Otherwise, S propagates a FIND message to its child that shares the longest 'spath' prefix with the new member 'spath'. The FIND message carries the IP address and the 'spath' of the new member, and is processed by executing path matching and either updating the FT, or propagating the FIND. The propagation of FIND messages continues until the new member finds a parent.

**Member Leave:** A member can leave the session by sending a LEAVE message to its parent. A LEAVE message includes the FT of the leaving member. Upon receiving LEAVE from a child, its parent removes the child from its FT and adds FT entries for the children of leaving child.

## 2.6 Conclusions

In this chapter we have looked at different *Application Layer Multicast* protocols. We analyzed these protocols and could classify them based on properties like a) the topology they create – tree, mesh, hierarchy etc., b) kind of algorithm they use like centralized, distributed etc., and c) based on their awareness of network topology. While we could classify the protocols and explain them in this chapter, we simulated these protocols and compared their performances and time complexities of the algorithms along with our proposed protocol 'Appcast' in chapter 3.

# Chapter 3

# Appcast - An Application Layer Multicast Protocol

In this chapter, we propose a new *Application Layer Multicast Protocol* called Appcast. The existing protocols we described in chapter 2 follow two steps to achieve multicast capability. 1. Arrange receivers into an overlay network of uni-cast connections and 2. Construct efficient data distribution trees over this overlay network to distribute data. At the heart of *Application Layer Multicast* protocols is the overlay topology they create. The topology is created as members/hosts join the multicast group. *Application Level Multicast* topology building algorithms define a definite relationship among the participating members and thereby create topologies like tree, mesh, hierarchy etc. The relationship can be parent-child, host-neighbors, cluster member – cluster leader etc. While many *Application Layer Multicast Protocols* are not topology aware as explained in chapter 2, in Appcast we exploit the topology information to better align the overlay topology with the underlying network topology. We also consider exploiting the broadcast nature of the media, which is shielded by TCP/IP. In Appcast, we propose to exploit this broadcast property by re-engineering the TCP behavior, by a method called 'multicast spoofing', which is an extension of 'TCP Spoofing'. Section 1 of this chapter introduces *Application Layer Multicast Protocol.* Section 2 describes the basic conditions and method that Appcast follows to create overlay topology and distribute data over it. Section 3 gives algorithms that create Appcast overlay satisfying the conditions laid down in section 2. Section 4 is about optimizing the basic Appcast overlay creation and gives the optimized algorithms. In section 5, we simulate Appcast overlay protocol and other protocols and compare them. Section 6 describes the broadcast media and problems of TCP over such media. Section 7 describes our approach 'multicast spoofing'. Section 8 gives performance improvement results and implementation details of 'multicast spoofing'. Section 9 concludes the chapter.

## 3.1 Introduction

Each application level multicast protocol differs in how they arrange the hosts into an overlay, manage it and distribute the data over it. To manage the tree, hosts in the overlay keep exchanging information about their health at regular intervals. The overlay topology considers this overhead of exchanging information as control overhead and every protocol tries to minimize it. The major goal of *Application Layer Multicast* is to reduce the redundant movements of data on network links through the routers. The number of links a packet crosses to reach all receivers measures whether this goal is achieved by a protocol. The lesser the number of links a packet travels, the more efficient the protocol is in terms of resource utilization like bandwidth, router processing and memory. The overall efficiency of the overlay protocol is measured by two factors – 1.Stress and 2.Stretch. *Stress* is defined per host or per link as 'the number of duplicate packets that a host sends or duplicate packets that travel over a link in an overlay'. *Stretch*, also called as '*relative delay penalty'* is defined per pair of hosts i.e., source and destination as the 'ratio between the number of hops a packet travels from source to destination in unicast mode to the number of hops the packet travels in multicast'. In case of IP Multicast, all links experience unit stress and every pair of hosts experiences unit stretch (Chapter 2 gives more details on stress and stretch). Though *Application Layer Multicast* protocols are not capable of achieving unit stress and unit stretch, they try to balance stress and stretch. Protocols like Bayeux [12] and Scribe [14] are motivated by Peer-to-Peer networks and arrange the hosts into overlay by logically assigning unique number to each host, irrespective of their proximity relations in real underlying network topology. Similarly CAN [18-19] and DTProtocol [11] arrange the hosts by assigning each host a place in a geometric space. The performance of these protocols is heavily taxed by their lack of awareness of underlying topology. The protocol NICE [15-17] is motivated by secure group communication. It first arranges the hosts into a tree topology and groups them into clusters by traversing the entire tree. This grouping is basically to reduce the depth of the tree so that it can contain the control overhead as cluster leaders only exchange information instead of all hosts exchanging information with every one else in the overlay. By doing so, it actually loses the multicast advantage, as communication within a cluster is unicast i.e., larger groups increase the over all links a packet travels to reach all receivers and smaller groups increase the tree depth and hence the control overhead. So an efficient Application *Layer Multicast* protocols is one, that takes few overall links to disseminate data to all group members, balances stress and stretch and manages the overlay with less control overhead.

## 3.2 Appcast Overlay Topology

In this section, we describe our proposed 'Appcast protocol' as an *Application Layer Mutlicast Protocol.* Appcast creates an overlay topology that is common for all multicast groups. The overlay topology is a tree topology with centralized algorithm to construct the tree. Appcast depends on the topology information to create the tree as it exploits the path redundancy between nodes while creating the tree. One of the nodes, preferably the one owned by the network service provider acts a root node for the multicast tree. Unlike other application layer multicast protocols, wherein every node acts as a routing node also, we clearly demarcate the functionalities of the nodes. In our scheme, we have two kinds of nodes, 1. *End hosts* and 2. *Proxies.* Proxies are the ones, who can actually route the applications, where as the end-hosts are the ones which can be either source or destination. In other words, in Appcast tree, *proxies* can have children, but *end-hosts* cannot have the children. In all the proposed topology creation algorithms, we considered nodes as *proxy* nodes only. The *end-host* nodes information need not proliferate through out the topology and is local to each *proxy*.

### 3.2.1 Overlay Topology Creation

Every new *proxy* that has to join the overlay, contacts the *root* first. The *root* determines to which *proxy,* this new member should get hooked based on the distance metric i.e., number of hops. The algorithms can work for other metrics also, like bandwidth, cost etc. *Root* selects a *proxy* (p), as the parent of the new member (m) meeting the following conditions. If *such proxy* is not found, *root* itself becomes parent to the new member.

1.D(R,m)>D(p,m)

The distance between *root* and new member is greater than the distance between *proxy* and new member.

2. For all *proxies* $P_i$ that satisfy condition 1, D(p,m)<=D($P_i$,m)

There can be more than *one proxy* satisfying the condition 1, and if so, choose the ones which are nearer to the new member.

3. Path(Parent(p),m) contains Path(p,m)

The path between *proxy* and member is a prefix to the path between *proxy*'s parent and member. The path need not be shortest path.

When **a new** member is about to join the group satisfying the above conditions will create three cases. 1. A new member's arrival does **not** affect the overlay structure. 2. A new member takes over its parent by reordering the parent-child relationship. 3. A new member takes over its siblings as their parent.

**Case1: A new member's arrival does not affect the overlay structure**

In this case the new joining *proxy* Pn gets a P, which is very near to it. Introducing this *proxy* into overlay does not affect rest of the tree structure. Figures 3.1a and 3.1b depict this scenario. The figures 3.1a and 3.1b show a network with 8 routers and 3 proxies, in which S acts as the root node and P1 and P2 joined the overlay in that order. In this scenario, the algorithm selects S *(root)* as the parent of P1 and when P2 also selects S as its parent, the overlay relation of S and P1 remain unchanged.

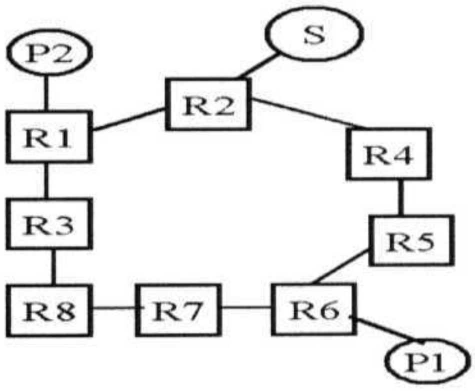


Figure 3. la Appcast Topology - Casel

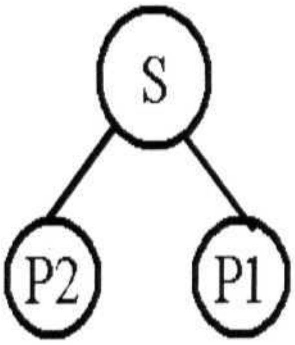
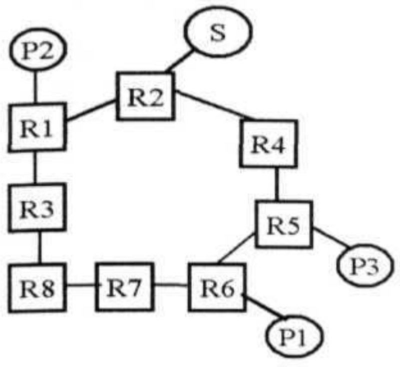
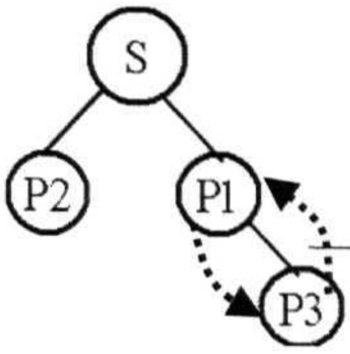


Figure **3.1b Appcast** Overlay

**Case2:** A **new member takes over its parent by reordering parent-child relationship**

The new joining *proxy* $P_n$ in this case selects a P such that $P_n$ is in the path of P's parent and P. In this case as in figure 3.2 series, the algorithm first selects P1 as the parent of P3 as per condition 1, but to meet the condition 3, it has to reorder the parent-child relationship, i.e., P3 becomes parent of P1 and child to S.





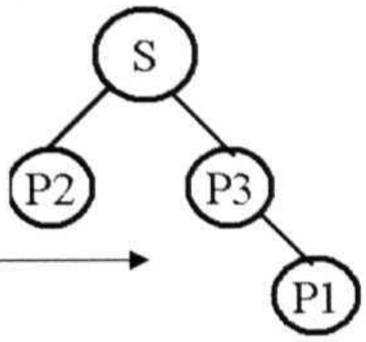


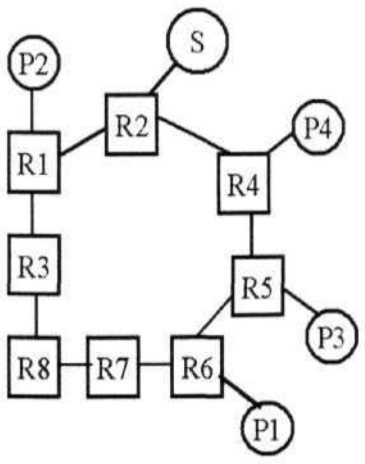
Figure 3. 2a: Appcast Topology - Case2 Figure 3.2b: Before reorder Figure 3.2c: After reorder

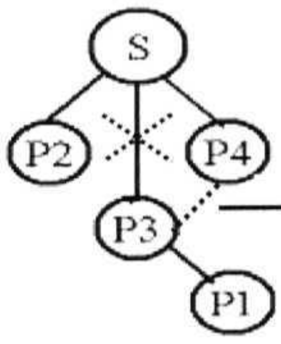
**Case3: A new member takes over its siblings as their parent**

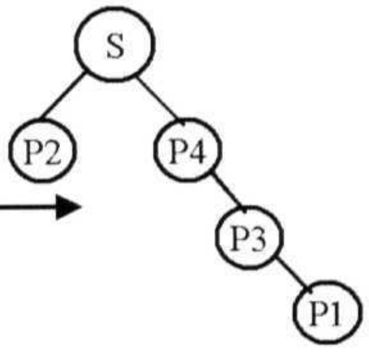
In this case, the new *proxy* $P_n$ selects a parent P such that $P_n$ is in the path of P and some of the children *proxies* of P. In figure 3.3 series, P4 selects S as its parent and since P4 is the child of S and is in the path of S and P3; P4 takes over as parent to P3.

Figure 3. 3a: Appcast Topology: Case3 Figure 3.3b: Before Taking over sibling

Figure 3.3c: After Taking over Sibling







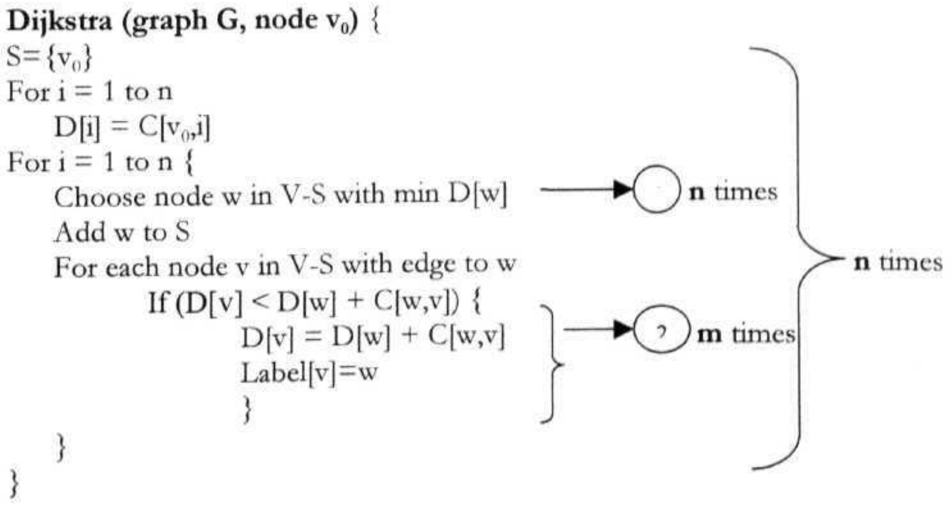
### 3.2.2 Appcast Tolpology Creation - A Low Stress and High Stretch Overlay Protocol

In this section, we describe the algorithms that create the overlay topology satisfying the conditions and cases discussed in section 3.2.1. Every new *proxy* willing to join the group sends a join message to the *root*. The *root* invokes the function "FindNearestProxy" which returns a *proxy* that is closest to the node. *Root* then calls "FindRelations" to fix the relationship of the new joining *proxy* and others in the overlay tree satisfying the conditions laid in section 3.2.1.

The well-known Dijkstra's algorithm [117] finds the shortest paths from a source to a destination (vertices) or all destinations in a graph. Dijkstra's algorithm keeps two sets of vertices. 1. The set of finished vertices and 2. Set of active vertices (active set). A vertex is active if it has a temporary label and a finite distance but the investigation of the vertex have not yet been finished. The algorithm terminates once the required destination joins the first set, in case it has to find path between source and destination or once the 2nd set becomes null, in case it has to find paths between source and all destinations.

**Dijkstra's Algorithm**

Let **V** be set of **n** vertices in graph G representing the given network. S is the set of vertices whose labels are permanent. Let V-S be the set of active vertices with temporary labels. **C[i,j]** is the cost of edge between vertex **i** and vertex **j. D[i]** is the distance from source node to the destination node **i. Label|i|** denotes the predecessor node to reach node i. The pseudo code algorithm for finding shortest paths from a single source $v_0$ to all nodes in a network represented by graph G is as given below.



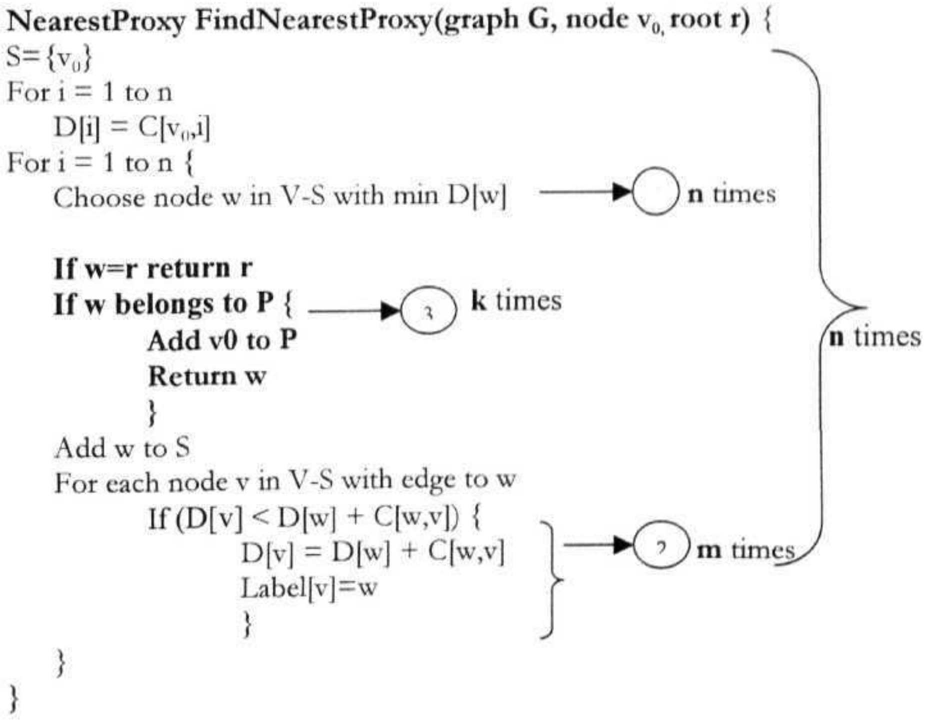
```
Dijkstra (graph G, node v0) {
S={v0}
For i = 1 to n
    D[i] = C[v0,i]
For i = 1 to n {
    Choose node w in V-S with min D[w]
    Add w to S
    For each node v in V-S with edge to w
            If (D[v] < D[w] + C[w,v]) {
                    D[v] = D[w] + C[w,v]
                    Label[v]=w
                    }
    }
}
```

**Complexity of Dijkstra's Algorithm**

The complexity of Dijkstra's algorithm [118] is based on two operations: 1. Finding a node with minimum distance. This has a complexity of $O(N)$ and 2. Changing the labels with complexity $O(m)$. These two steps are performed N times and so the total complexity of the algorithm is $O(N^2)+O(mN)$.

**Appcast - FindNearestProxy Algorithm**

This algorithm is written by modifying the above Dijkstra's Algorithm. The base Dijkstra's algorithm finds shortest path between a source and destination. The algorithm starts from the source node and keeps fixing the labels for intermediary nodes to reach the destination node and terminates the algorithm once it reaches the destination. In our case, the source node is the 'joining proxy' and destination node is the 'root'. The problem of finding the nearest proxy turns out to be finding an intermediary node (this node has to be proxy) in Dijkstra's algorithm while traversing from source to destination via shortest path i.e., 'joining proxy' to 'root'. Apart from the two sets that Dijkstra's algorithm keeps; we keep one more set of nodes i.e., set of all proxies. We change the algorithm such that the algorithm terminates once it reaches any node that belongs to this set. The pseudo code for this algorithm is given below.



```
NearestProxy FindNearestProxy(graph G, node $v_0$, root r) {
S={$v_0$}
For i = 1 to n
    D[i] = C[$v_0$,i]
For i = 1 to n {
    Choose node w in V-S with min D[w]

    If w=r return r
    If w belongs to P {
            Add v0 to P
            Return w
            }
    Add w to S
    For each node v in V-S with edge to w
            If (D[v] < D[w] + C[w,v]) {
                    D[v] = D[w] + C[w,v]
                    Label[v]=w
                    }
    }
}
```

n times
k times
n times
m times

**Complexity of FindNearestProxy Algorithm**

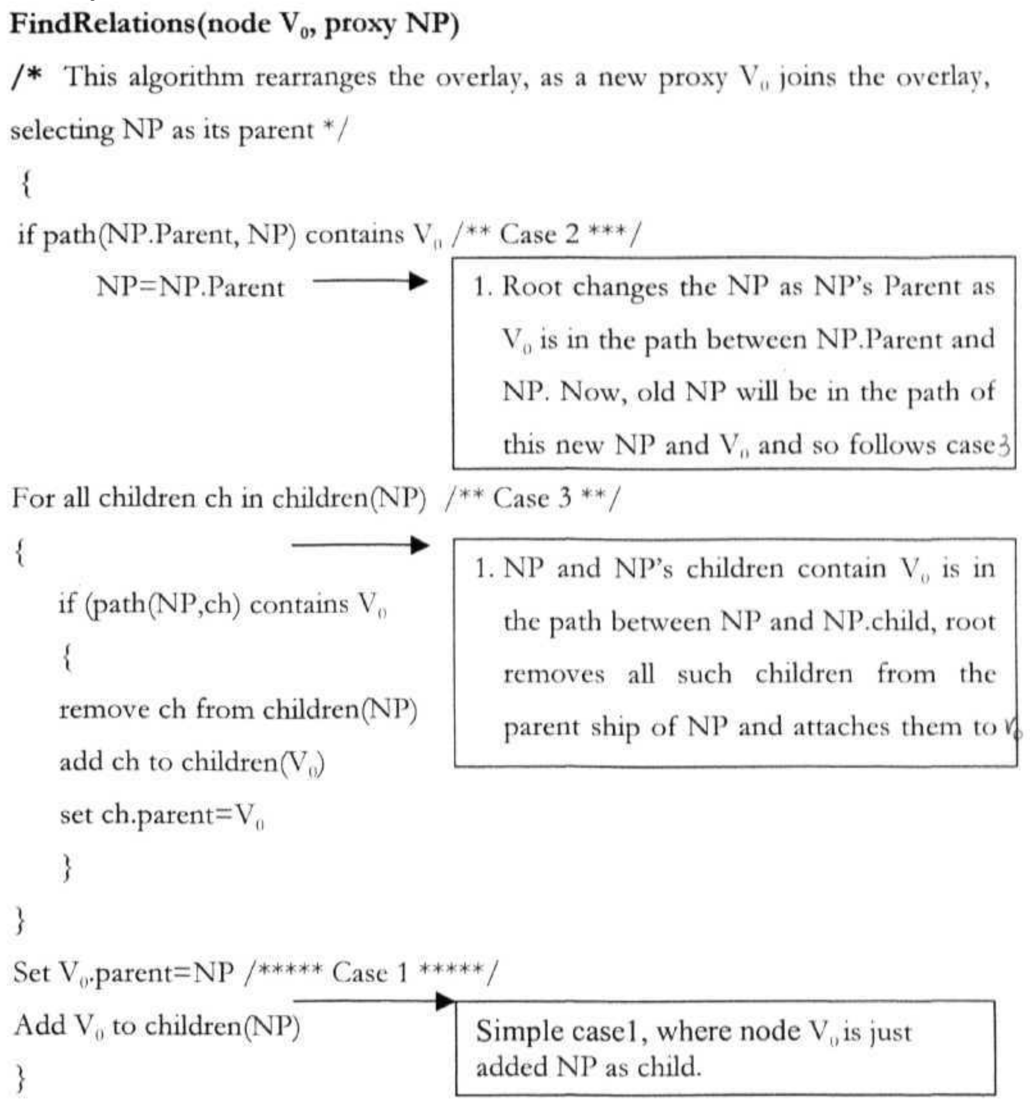
As said earlier, the complexity of Dijkstra's algorithm is based on two operations: 1. Finding a node with minimum distance. This has a complexity of O(N) and 2. Changing the labels with complexity O(m). These two steps are performed N times and so the total complexity of the algorithm is $O(N^2)+O(mN)$. In addition to this, in our algorithm, we have to find whether the selected node with minimum index belongs to the set of proxies. This has a complexity of O(k). These three steps are perfomied N times and total complexity is $O(N^2)+O(mN)+O(kN)$.

### 3.2.3 Appcast Topology Management

In this section, we describe how Appcast, manages the overlay that it created keeping the relationships of members intact, i.e., satisfying the conditions laid down in section 3.2.1. The topology management has to take care of members leaving and links down, members down etc., scenarios. The *root* keeps information of all proxies. Every *proxy* keeps information about its parent and its children. Also, every *proxy* keeps track (heart beat) of its children and parent. If any *proxy* is down, immediately, its children contact the *root* and try to hook to the parent of the downed *proxy*. It is the *root*, which tells the children about their new parent, keeping all constraints satisfied. Whenever, a new *proxy* joins or leaves, few other proxies also will be informed by *root* to change their relationships, so that the constraints are satisfied.

**Appcast - FindRelations Algorithm**

Given below is the algorithm that takes care of all three cases whenever a new proxy joins the overlay. This algorithm is invoked by 'root' after the algorithm 'FindNearestProxy'.



```
FindRelations(node $V_0$, proxy NP)
/* This algorithm rearranges the overlay, as a new proxy $V_0$ joins the overlay,
selecting NP as its parent */
{
 if path(NP.Parent, NP) contains $V_0$ /** Case 2 ***/
        NP=NP.Parent
For all children ch in children(NP)  /** Case 3 **/
{
     if (path(NP,ch) contains $V_0$
     {
     remove ch from children(NP)
     add ch to children($V_0$)
     set ch.parent=$V_0$
     }
}
Set $V_0$.parent=NP /***** Case 1 *****/
Add $V_0$ to children(NP)
}
```

1. Root changes the NP as NP's Parent as $V_0$ is in the path between NP.Parent and NP. Now, old NP will be in the path of this new NP and $V_0$ and so follows case3

1. NP and NP's children contain $V_0$ is in the path between NP and NP.child, root removes all such children from the parent ship of NP and attaches them to $V_0$

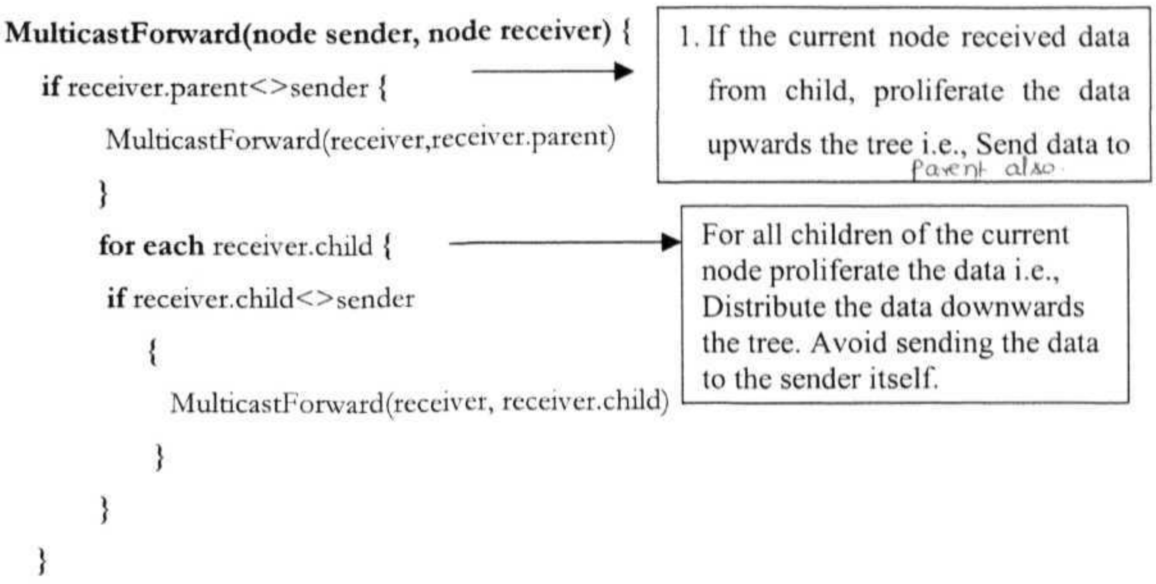
Simple case1, where node $V_0$ is just added NP as child.

**Appcast Data Movement**

In Appcast, data can be flowed bottom-up and top-down across the Appcast topology tree. To avoid loops, each node checks from which it received the data and accordingly forwards to selective children and parent. Whenever a *proxy* receives data, it checks from whom it received. If it received from it's parent, then it forwards packets to all its children. If it received packet from one of its children (i.e., not parent), it forwards the packet to **its** own parent and to all its children except the child from whom it received.

**Appcast - MulticastForward Algorithm**

Every proxy in the overlay calls the following algorithm as and when it receives a piece of information on overlay.



```
MulticastForward(node sender, node receiver) {
   if receiver.parent<>sender {
        MulticastForward(receiver,receiver.parent)
        }
        for each receiver.child {
         if receiver.child<>sender
             {
               MulticastForward(receiver, receiver.child)
             }
        }
  }
```

1. If the current node received data from child, proliferate the data upwards the tree i.e., Send data to parent also.

For all children of the current node proliferate the data i.e., Distribute the data downwards the tree. Avoid sending the data to the sender itself.

## 3.3 Appcast Algorithms

In this section, we give all the algorithms described above as a series. Following are the notations used in the Appcast Algorithms.

**G** - connected graph representing the network
**$v_0$** - is the *proxy* joining to multicast group
**r** – root node
**V** - is the set of all vertices in the graph G
**S** - set of all vertices with permanent labels
**P** – Set of all *proxy* nodes on G
**n** - number of vertices in G
**D** - set of distances to v0
**C** - set of edges in G
**NP** – Nearest Proxy found from the algorithm "FindNearestProxy"

**Distance(w, $v_0$)** is the unicast distance from node w to node $v_0$
**C_Distance(w, r)** is the cumulative (application level) distance from node w to root r
**Acpt_children (w)** is the number of children node w can accept
**Cur_children(w)** is the number of children node w has
D– Set of Distributing Proxies
O – Set of Publisher Proxies

**Algorithm 1:**

```
1.  NearestProxy FindNearestProxy (graph G, node $v_0$, node r, Proxies P)
/*  Returns the 'NearestProxy' from the set of proxies 'P', to which 'v0' can
be a child. If no such proxy is found, v0 is attached to root r */
{
        S={$v_0$}
        For i = 1 to n
                D[i] = C[$v_0$,i]
        For i = 1 to n-1
                {
                Choose node w in (V-S) with min D[w]
                If node w belongs to P
                      {
                              add $v_0$ to P
                              return w
                      }
              Add w to S
              For each node v in (V-S)
                      D[v] = min(D[v], D[w] + C[w,v])
                }
}
```

**Algorithm 2:**

```
2. FindRelations(node $V_0$, proxy NP)
/*  This algorithm rearranges the overlay, as a new proxy $V_0$ joins the overlay,
selecting NP as its parent */
 {
    if path(NP.Parent, NP) contains $V_0$ /** Case 2 ***/
              NP=NP.Parent
    For all children ch in children(NP)  /** Case 3 **/
    {
        if (path(NP,ch) contains $V_0$
                  {
                  remove ch from children(NP)
                  add ch to children($V_0$)
                  set ch.parent=$V_0$
                  }
    }
    /***** Case 1 *****/
    Set $V_0$.parent=NP
    Add $V_0$ to children(NP)
}
```

**Algorithm 3:**

```
3. MulticastForward(node sender, node receiver)
/***** A recursive algorithm called by every proxy in the overlay to distribute
data over the Appcast Overlay  *********/
  {
        if receiver.parent<>sender /**Distribute data upwards ***/
        {
             MulticastForward(receiver,receiver.parent)
        }
        for each receiver.child /*** Distribute data downwards ***/
           {
              if receiver.child<>sender /*** Avoid loops by not sending data
                                          to the sender again ***/
                 {
                    MulticastForward(receiver, receiver.child)
                  }
           }
  }
```

### 3.4 Appcast Optimization - A Balanced Multicast Overlay Protocol

In this section, we propose optimization to Appcast overlay creation and data distribution by using two approaches. In the first approach, the performance criteria of any *Application Layer Multicast* protocol like *Stress* and *Stretch* have been considered and the protocol tries to optimize these criteria. In second approach, a publisher and distributor approach is considered wherein a publisher is the one who can own and originate and distribute the data, while the distributor is the one who just can keep distributing the data without any right to originate the data. We describe both the approaches in the following sections.

#### 3.4.1 Appcast Optimization using Stress and Stretch Criteria

In Appcast described in section 3.3, a *proxy* joining the multicast group selects the very first *proxy* that it comes across while finding the path from itself to the *root.* This approach definitely ensures that the path length from the 'joining *proxy'* to the 'parent *proxy*' is lesser than the path length from the 'joining*proxy*' to the '*root*'. However, if we take into consideration the actual path length from *root* to this new *proxy* (along the proxies), the path would be lengthier.

The performance results as in section 5, clearly show that Appcast uses very few over all links/hops. At the same time, it also show the maximum application level path lengths i.e., maximum stretch. To keep the stretch and stress at an optimum level, the Appcast_optl algorithm is proposed. In this algorithm, a 'joining*proxy*' can specify how many children (stress) it can accept and how much stretch (delay) it can bear. So the algorithms described in section 3.2 are modified to accommodate the above optimization for 'FindNearestProxy' and 'FindRelations' algorithms, while the 'Multi cast Forward' algorithm remains same. Both the algorithms are modified to verify the '*stress*'capacity of the 'proxy' chosen by a joining node and *'stretch '* criteria chosen by the joining node itself. The algorithm 'MulticastForward' does not require any modifications, as all proxies are arranged into an overlay only after verifying their capacities and so at the time of distributing the data, they need not be verified for the capacity.

Algorithm - **FindNearestProxy_Opt1**

NearestProxy **FindNearestProxy_opt1(graph** G, node $v_0$, node r, float **acpt_stretch)**

/* This algorithm finds a 'NearestProxy' that satisfies the stretch criteria imposed by node $v_0$, and the stress criteria imposed by 'NearestProxy' */

```
{   S={v_0}
    for i =1 to n  { D[i] = C[v_0,i] }
    for i = 1 to n-1 {
        Choose node w in V-S with min D[w]
        if node w belongs to P { ----> Evenif a proxy is found, it is verified for its capacity.
            if (((Distance(w,v_0)+ C_Distance(w,r)) / Distance(v_0,r) <
            acpt_stretch) && acpt_children(w)>cur_children(w)) {
                add v_0 to P
                return w }     }
        if node w = r, return r
        Add w to S
        for each node v in V-S
        D[v] = min(D[v], D[w] + C[w,v])
        }
}
```

**Algorithm FindRelations_opt1(node $V_0$, node NP)**

Following the optimization criteria set by each joining proxy, even this algorithm is modified from FindRelations algorithm of section 3.2.

```
FindRelations_opt1(node $V_0$, node NP) {
        if path(NP.Parent, NP) contains $V_0$  /** Case 2 **/
         NP=NP.Parent      →     Stretch need not be verified,
                                 as it is always less from
                                 NP.Parent to $V_0$ compared
                                 to NP to $V_0$
        for all children ch in children(NP) { /** Case 3 **/
          if (path(NP,ch) contains V0 && Acpt_children($V_0$)> Cur_children($V_0$)){
                remove ch from children(NP)
                add ch to children($V_0$)   →   Relationship is modified
                set ch.parent=$V_0$ } }         only if the node $V_0$ has
                                                capacity.
        /** Case 1 **/   →
        Set $V_0$.parent=NP        Capacity need not be verified as its
        Add $V_0$ to children(NP)   already taken care in
        Add $V_0$ to P              FindNearestProxy_Opt1 algorithm in this
                                    case.
        }
```

### 3.4.2 Appcast optimization using Publisher and Distributor proxies

In addition to the optimization given using the criteria like stress and stretch specified by each *proxy*, a '*proxy*' can also specify whether it is a *distributor* or a *publisher* at the time of joining. A *distributor* is the one who can only distribute the data and a *publisher* is the one who can originate as well as *distribute* data. The algorithm ensures that all publishers are nearby and can proliferate the data faster onto every node of the tree. Otherwise, if the publisher happens to be at the end of the tree, the data originated from it will take long time to pass over all the nodes of the tree. Also, a *distributor* cannot send the data to a *publisher*, whereas publishers can send data to any one, including other publishers and distributors. Opt2 series of algorithms use this concept.

```
NearestProxy FindNearestProxy_opt2(graph G, node $V_0$, node r, float acpt_stretch){
    S={$V_0$}
    if $V_0$ is a publisher
    {
        Add $V_0$ to O
        P=O
    }
```

If the joining node is a publisher, add it to the set of publishers 'O'. Also, a publisher cannot take a distributor as its parent. So the set of proxies 'P' whom it can consider, as its potential parents is just the set of publishers.

```
    else
    {
        Add v0 to D
        P=O+D
    }
```

If the joining node is a distributor, add it to the set of distributors 'D'. Also, a distributor can take any distributor or publisher as its parent. So, the set of proxies 'P' whom it can consider, as its potential parents is the set of publishers as well as distributors.

```
    for i = 1 to n
        D[i] = C[v0,i]

    for i = 1 to n-1
    {
    Choose node w in (V-S) with min D[w]
     if node w belongs to P
      {
      if (((Distance(w,v0)+ C_Distance(w,r)) / Distance(v0,r) < acpt_stretch) &&
                                        acpt_children(w)>cur_children(w))
      return w
      }
    if node w = r
            return r
    Add w to S
    for each node v in V-S
            D[v] = min(D[v], D[w] + C[w,v])
    }
}
```

Optimization using stress and stretch

**Appcast –FindRelations_Opt2 Algorithm**

```
FindRelations_opt2(node v0, node NP) {
 if path(NP.Parent, NP) contains v0  /** Case 2 **/
   NP=NP.Parent
   for all children ch in children(NP) /** Case 3 **/
   {
       if (path(NP,ch) contains v0 && Acpt_children(v0)> Cur_children(v0))
       {
               if ch is a Distributor or v0 is a Publisher
               {
                       remove ch from children(NP)
                       add ch to children(v0)
                       set ch.parent=v0
               }
        }
   }
       /** Case 1 **/
   Set V0.parent=NP
   Add V0 to children(NP)
}
```

If Vo is a publisher, algorithm FindNearest Proxy ensures NP and NP.Parent also to be publishers.

1. If Vo is a publisher, all children of NP can as well be Vo's children irrespective of whether they are publisher or distributor. Similarly, if child is a distributor it can be a child to any proxy

**Data Distribution**

Data can be flowed bottom-up and top-down across the Appcast topology tree. Whenever a *proxy* receives data, it checks from whom it received. If it received from it's parent, then it forwards packets to all its children. If it received packet from one of its children (i.e., not parent), it forwards the packet to its own parent and to all its children except the child from whom it received. Since we differentiate proxies as distributors and publishers, if receiver proxy is not a *publisher,* it cannot send data to its parent. Distributors can distribute the data downwards, not upwards, as a *distributor proxy* can never originate the data. With these differences the algorithm 'MulticastForward' is changed as 'MulticastForward_opt2' algorithm.

```
MulticastForward_opt2(node sender, node receiver) {
  if receiver.parent<>sender  && receiver is a Publisher
            {
            MulticastForward(receiver,receiver.parent)
            }
      for each receiver.child {
            if receiver.child<>sender {
                  MulticastForward(receiver, receiver.child)
            } }}
```

Proliferate data upwards only if the current node's parent happens to be a publisher. No child can originate the data unless it is publisher. It also means, only publishers can send data upwards.

### 3.4.3 Appcast Optimization Algorithms

In this section we give all algorithms described in section 3.4.2 as a series of algorithms.

**Algorithm 4:**

```
4. NearestProxy FindNearestProxy_opt1(graph G, node $v_0$, node r, float acpt_stretch)
/* This algorithm finds a 'NearestProxy' that satisfies the stretch criteria
imposed by node $v_0$, and the stress criteria imposed by 'NearestProxy' */
  {
      S={$v_0$}
      for i = 1 to n
            D[i] = C[$v_0$,i]
      for i = 1 to n-1
      {
            Choose node w in V-S with min D[w]
            if node w belongs to P
                  {
                  if (((Distance(w,$v_0$)+ C_Distance(w,r)) / Distance($v_0$,r) <
                  acpt_stretch) && acpt_children(w)>cur_children(w))
                        {
                        add $v_0$ to P
                        return w
                        }
                  }
            if node w = r, return r
            Add w to S
            for each node v in V-S
            D[v] = min(D[v], D[w] + C[w,v])
      }
}
```

**Algorithm 5:**

```
5. FindRelations_opt1(node $V_0$,  node NP)
{
        if path(NP.Parent, NP) contains $V_0$  /** Case 2 **/
           NP=NP.Parent
        for all children ch in children(NP) { /** Case 3 **/
            if (path(NP,ch) contains V0 && Acpt_children($V_0$)> Cur_children($V_0$)){
                    remove ch from children(NP)
                    add ch to children($V_0$)
                    set ch.parent=$V_0$
                    }
          }
        /** Case 1 **/
        Set $V_0$.parent=NP
        Add $V_0$ to children(NP)
        Add $V_0$ to P
}
```

**Algorithm 6:**

```
6. NearestProxy FindNearestProxy_opt2(graph G, node v0, node r, float acpt_stretch)
{
        S={v0}
        if v0 is a sender {
              Add v0 to O
              P=O }
        else {
             Add v0 to D
             P=O+D }
        for i = 1 to n
             D[i] = C[v0,i]
        for i = 1 to n-1 {
        Choose node w in (V-S) with min D[w]
        if node w belongs to P {
        if (((Distance(w,v0)+ C_Distance(w,r)) / Distance(v0,r) < acpt_stretch)
                                              && acpt_children(w)>cur_children(w))
            return w  }
        if node w = r
                return r
        Add w to S
        for each node v in V-S
               D[v] = min(D[v], D[w] + C[w,v])
         }
}
```

**Algorithm 7:**

```
7. FindRelations_opt2(node v0,  node NP) {
if path(NP.Parent, NP) contains v0  /** Case 2 **/
      NP=NP.Parent  /* if Vo is a publisher, alg 6 ensures NP & NP.Parent  also
                        to      be senders */
      for all children ch in children(NP) { /** Case 3 **/
      if (path(NP,ch) contains v0 && Acpt_children(v0)> Cur_children(v0)) {
            if ch is a Distributor or v0 is a Publisher {
                        remove ch from children(NP)
                        add ch to children(v0)
                        set ch.parent=v0 }
                  }
            }
      /** Case 1 **/
      Set V0.parent=NP
      Add V0 to children(NP)
}
```

**Algorithm 8:**

```
8. MulticastForward_opt2(node sender, node receiver)
{
      if receiver.parent<>sender   && receiver is a Publisher /* Sender is
                                                       receiver's child */
            {
            MulticastForward(receiver,receiver.parent)
            }
      for each receiver.child {
             if receiver.child<>sender {
                   MulticastForward(receiver, receiver.child)
             }
        }
}
```

## 3.5 Comparative study

The protocols (CAN [18-19], Bayeux [12], DTProtocol [11] etc.,) that have no knowledge of the underlying topology suffer poor performance and can help only in sharing and distributing the load of the source across the members. The mesh based protocols like ESM [1-2] and Yoid [20-21] suffer from control overhead $O(N^2)$ and are not suitable for large groups. The tree and hierarchical topologies like HMTP [13], TAG

[22] and NICE [15 –17] are able to contain the control overhead and at the same time performing well. Following table shows the intuitive comparison metrics.

Table 3.1 Application Layer Multicast Protocol Comparison

| Criteria / Protocol | Max Path Length | Max Tree Degree | Avg. Control Overhead |
|---|---|---|---|
| Appcast | Parameterized | Parameterized | O (Max. Degree) |
| ESM | Unbounded | Unbounded | O ($N^2$) |
| Yoid | Unbounded | O (Max. degree) | O (Max. degree) |
| HMTP | Unbounded | O (Max. degree) | O (Max. degree) |
| CAN | O ($dN^{1/d}$) | Constant | O (log N) |
| Bayeux | O (log N) | O (log N) | O (log N) |
| TAG | Unbounded | Unbounded | O (Max. tree degree) |
| NICE | O (log N) | Constant | Constant |

## 3.5.1 Simulation and results

For comparison purposes we considered only TAG [22], NICE [15-17] and HMTP [13] as our proposed protocol Appcast also is a tree based protocol. The figures 3.5, 3.6 and 3.7 show the overlay topologies created by them, when taken the network shown in figure 3.4. Figure 3.8 shows the overlay created by Appcast. Rl,R2,R3....R10 represent routers and S,A1,A2...A5 nodes in figure 3.4. The order of joins of the nodes is A3, A4, A5, Al and A2. Since TAG chooses its parent based on the longest path match over shortest path from node to root, A3, selects A2 as parent, though Al is nearer to it. Order of joins, matter a lot for the performance of HMTP. Since 'A1' joined in the last, it just took the one as its parent, which is nearer to it ie A3, without checking whether it is on the way between S and A3. NICE groups nearby members into clusters and arranges these clusters into a hierarchy.

**Figure 3. 4: Sample Network**

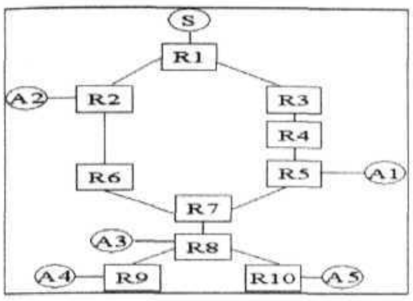


**Figure 3. 5: TAG Overlay**

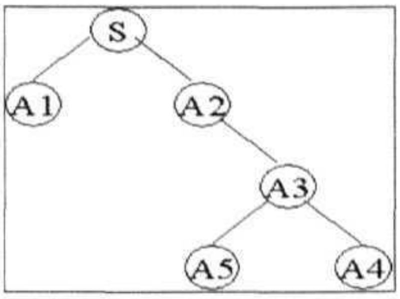


Figure 3. 6: **NICE**

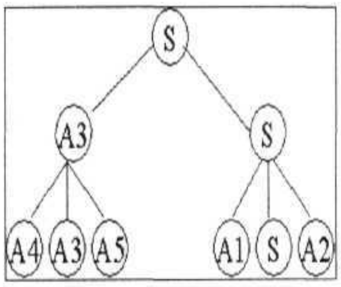


Figure 3. 7: HMTP

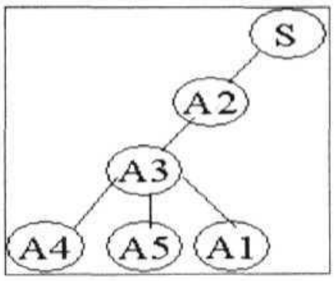
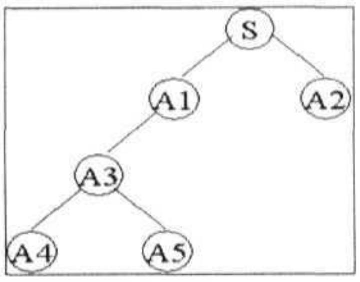


Figure 3. 8: **Appcast**



We used Boston University's Network Topology generator - BRITE [77] to simulate our experiments. BRITE generates different kinds of network topologies based on the models - Flat Router Level models (Router Waxman, Router BarbasiAlbert); Flat AS Level models (AS-Waxman, AS-BarbasiAlbert) and Hierarchical models (Transit-stub, tiers). First, we generated 100 nodes in AS model and assigned 20 hosts to these nodes. In this experiment, HMTP [13] showed higher stress and lower stretch. TAG [22] showed even more higher stress and less/no stretch. NICE, with cluster members fixed to 3, almost showed similar result like HMTP. Similar experiments have been conducted on network topology with 1000 nodes and with varying group memberships of hosts. Figure 3.9 shows hop comparison, figure 3.10 shows application level hops comparison; figure 3.11 shows stretch comparison and figure 3.12 shows the stress comparison of unicast, NICE, Appcast, HMTP and TAG.

Appcast used overall less hops followed by HMTP and TAG used almost similar hops like unicast. This is because - TAG node will select a parent, which has maximum overlapping shortest path with it. In other words, TAG does not look into alternative paths. This makes most of the nodes select the source itself as their parent i.e., very few nodes get nodes other than source as parent.

Figure 3. 9: Hops Comparison

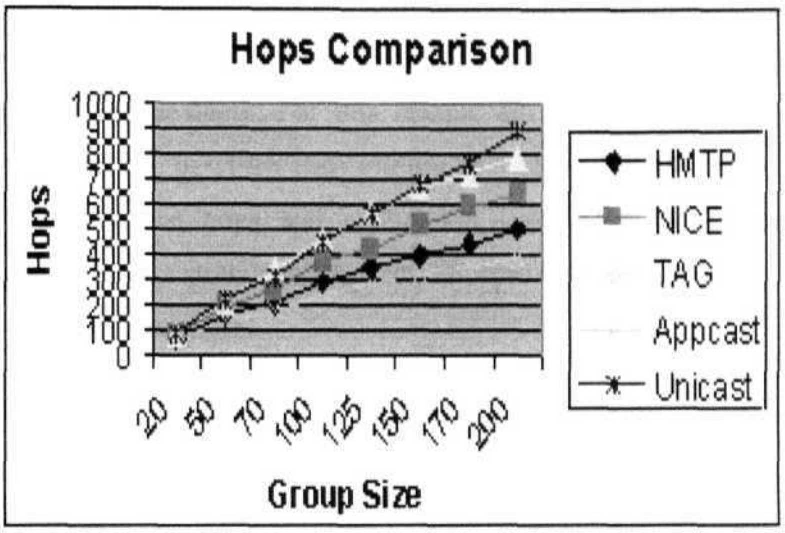
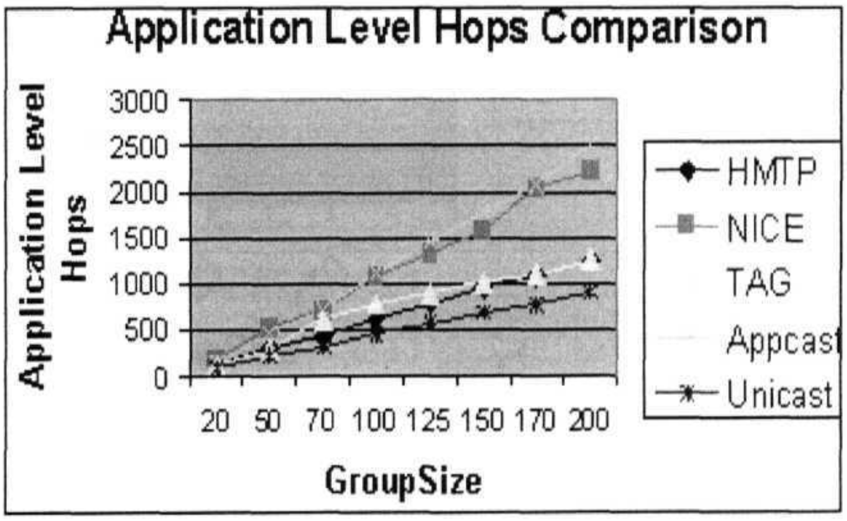


Figure 3. 10: Application Level Hops Comparison



TAG showed application level hops almost similar to unicast, as TAG does not look into alternative paths. NICE [15-17], while showing less over all hops compared to TAG, showed the higher application level hops compared to TAG and HMTP. This is because, with in clusters, NICE uses normal unicast among the cluster members and clusters leaders. As the group size increases, application level hops increase tremendously for NICE. Appcast is the one, which used the less number of hops. However, it is the one, which used maximum application level hops. This is because; it does not use any mechanism to control the tree depth. For this reason, an optimized version of Appcast protocol has been proposed, in which each joining host can specify the stretch parameter - the ratio between unicast hops and application level hops. As far as stretch is concerned, TAG showed less stretch and Appcast showed high stretch. NICE showed less stress and TAG showed high stress.

**Figure 3. 11: Stretch Comparison**

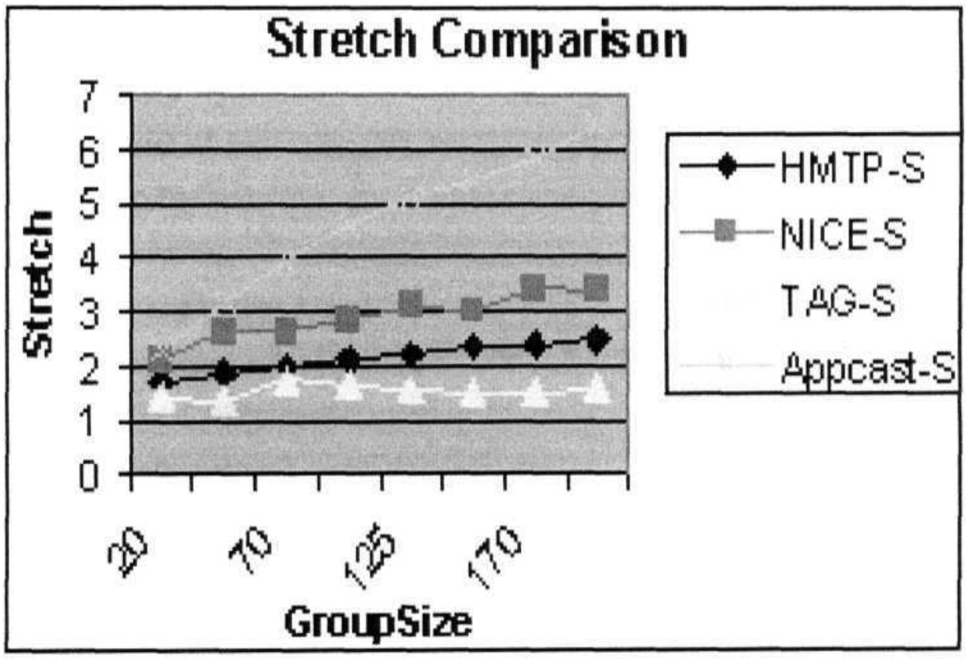
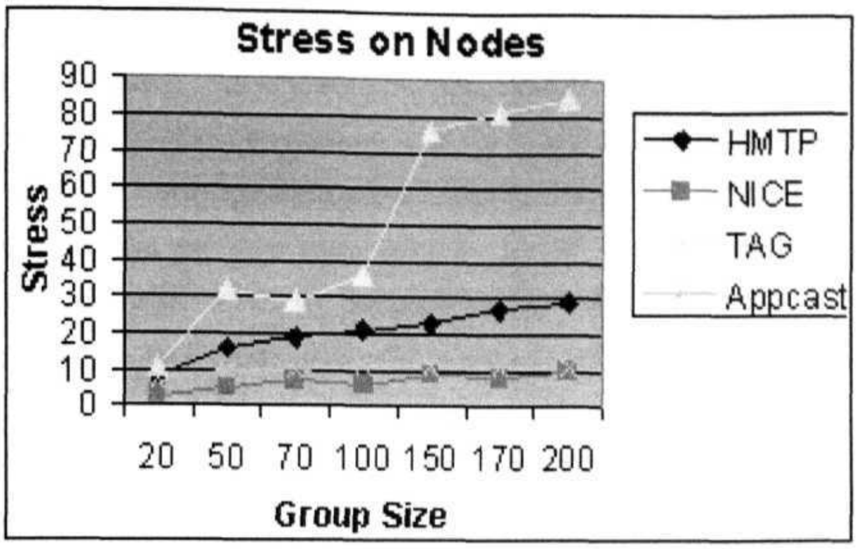


Figure 3. 12: Stress Comparison



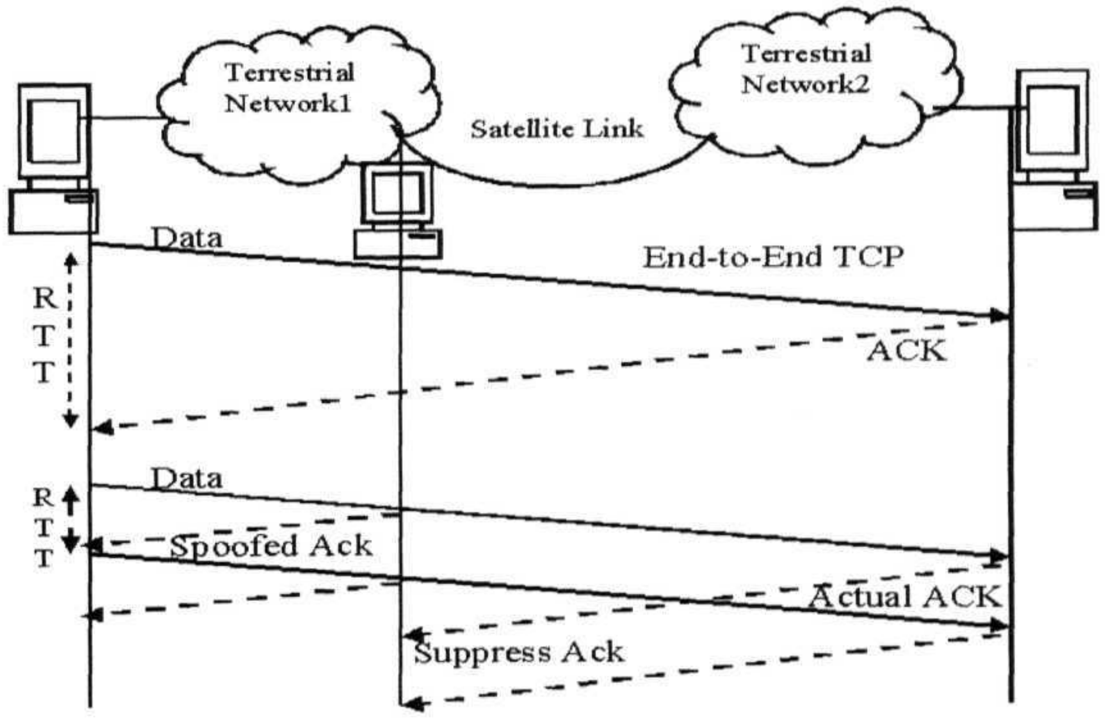
### 3.6 TCP/IP Over Satellite

The paper [53] described issues and pitfalls of satellite-based systems. Very Small Aperture Terminal (VSAT) networks are well known for delivering broadcast mode traffic. By nature broadcasting via satellite is simpler than for terrestrial networks, which are point-to-point networks. The wide range of content delivery supported by satellite clearly illustrates the versatility of this medium. The delivery of high-speed and interactive data via satellite has its own unique problems [53]. Geo-stationary satellite systems have a fixed round trip delay of 600ms that cannot be avoided. TCP/IP allows certain amount of data limited by window size, without an immediate acknowledgement from the receive end [57]. For a satellite link with round-trip delay of 0.8 seconds and bandwidth capacity of 1.54 Mbps, the theoretical optimal window size is 154 Kbytes, which is far higher than the maximum allowed TCP window size of 64K. TCP requires a three-way handshake between sender and receiver, before actual data start passing the network [55] i.e., three control packets should be exchanged between sender and receiver. For a satellite network with 600ms round-trip time, 1.8 sec is required for every communication session before the first data packet is exchanged.

To maximize bandwidth utilization in a satellite network, TCP needs a much larger window size. A new TCP extension, or TCP-LW for "large-window" [59], has been defined to increase the maximum window size from $2^{16}$ to $2^{32}$, allowing better utilization of links with large bandwidth delay products (bandwidth x delay). To obtain good TCP performance over satellite links, both sender and receiver use a version of TCP that implements TCP-LW. Transaction TCP, or T/TCP [55,63], is an extension to TCP designed to make 3-way handshake more efficient. T/TCP does this by bypassing the three-way handshake and slow start, using the cached state information from previous connections. Although T/TCP is designed mainly for short client-server interaction applications, it can be used to reduce the impact of latency on the beginning of a TCP connection.

TCP spoofing is another mechanism devised to increase the satellite performance using TCP/IP. In TCP spoofing as shown in figure 3.13, an intermediate gateway (usually at the satellite uplink) prematurely acknowledges a TCP segment without waiting for the actual acknowledgment from the receiver [53]. This gives the sender the illusion of a low-latency network so the TCP slow start phase can progress more rapidly. The intermediate gateway buffers segments in transit. When the actual acknowledgment from the receiver arrives at the gateway, it is suppressed to prevent duplicate acknowledgments from reaching the sender. When receiver's acknowledgment never arrives and the gateway times out, it retransmits the lost segment from its local buffer.

Figure 3. 13: TCP Spoofing



## 3.7 Multicast Spoofing

Satellite communication has a property of more inward capacity and less upward capacity at remotes coupled with high latency, i.e. a VSAT can receive more data than it can send. Sometimes even sending an acknowledgment can prove costly. Since Satellite communication is reliable, a mechanism called "TCP Spoofing" [55-59] (also explained in section 2.2.3.2) has been devised such that the sender keeps sending data without waiting for the actual acknowledgments arrive.

In TCP spoofing, an intermediate gateway (usually at the satellite uplink) prematurely acknowledges a TCP segment without waiting for the actual acknowledgment from the receiver. This gives the sender the illusion of a low-latency network so that TCP slow start phase can progress more rapidly. The intermediate gateway buffers segments in transit. When the actual acknowledgment from the receiver arrives at the gateway, it is suppressed to prevent duplicate acknowledgments from reaching the sender. We extend the 'TCP spoofing' behavior to the needs of multicasting as described in next paragraph.

**The Approach**

Applications that use IP Multicast assign a common IP multicast address and port number at which all clients (receivers) keep listening for the sender (server) to send information. This requires client application software to be aware of the multicast capability and write the code specifically with multicast API i.e., the software on clients in multicast mode and on clients in unicast mode has to be different. Also, many application developers are not aware of the multicast techniques (a network specialization) and seldom show any interest to develop such applications. Commercial high-speed multicast applications confined to general applications like audio and video distribution and are controlled mainly by satellite communication vendors like Hughes.

Our aim is to develop a methodology that simplifies the multicast application development exploiting the broadcast nature of the media. In next chapter we show exact applications like a) 'file push service' which pushes a static file to all recipients, b) 'database replication service' which pushes a XML file that contains both data and events to recipient databases for the databases to take action against the data sent based on the events and c) 'auction service' which pushes the bid quotes to all recipients in broadcast mode and receives responses in unicast mode. Information is pushed on to the recipients in either multicast or unicast mode based on the media through which recipients are connected to sender. Recipients need not use separate software, however they inform the sender of their media. All these applications are designed on standard Internet Protocols like HTTP, SMTP etc with SOAP [47-49] as wire protocol.

We modified the concept of "TCP Spoofing" to suit our requirement such that the gateway behavior is implemented at sender and receiver. We allocate a multicast address common to all the receivers. We implement multicast over normal TCP using "TCP Spoofing". The sender keeps sending information to the multicast address in unicast mode only. At the sender, the gateway itself generates acknowledgement packets with the source IP address of the receiver, without actually receiving them over wire as shown in figure 3.14.

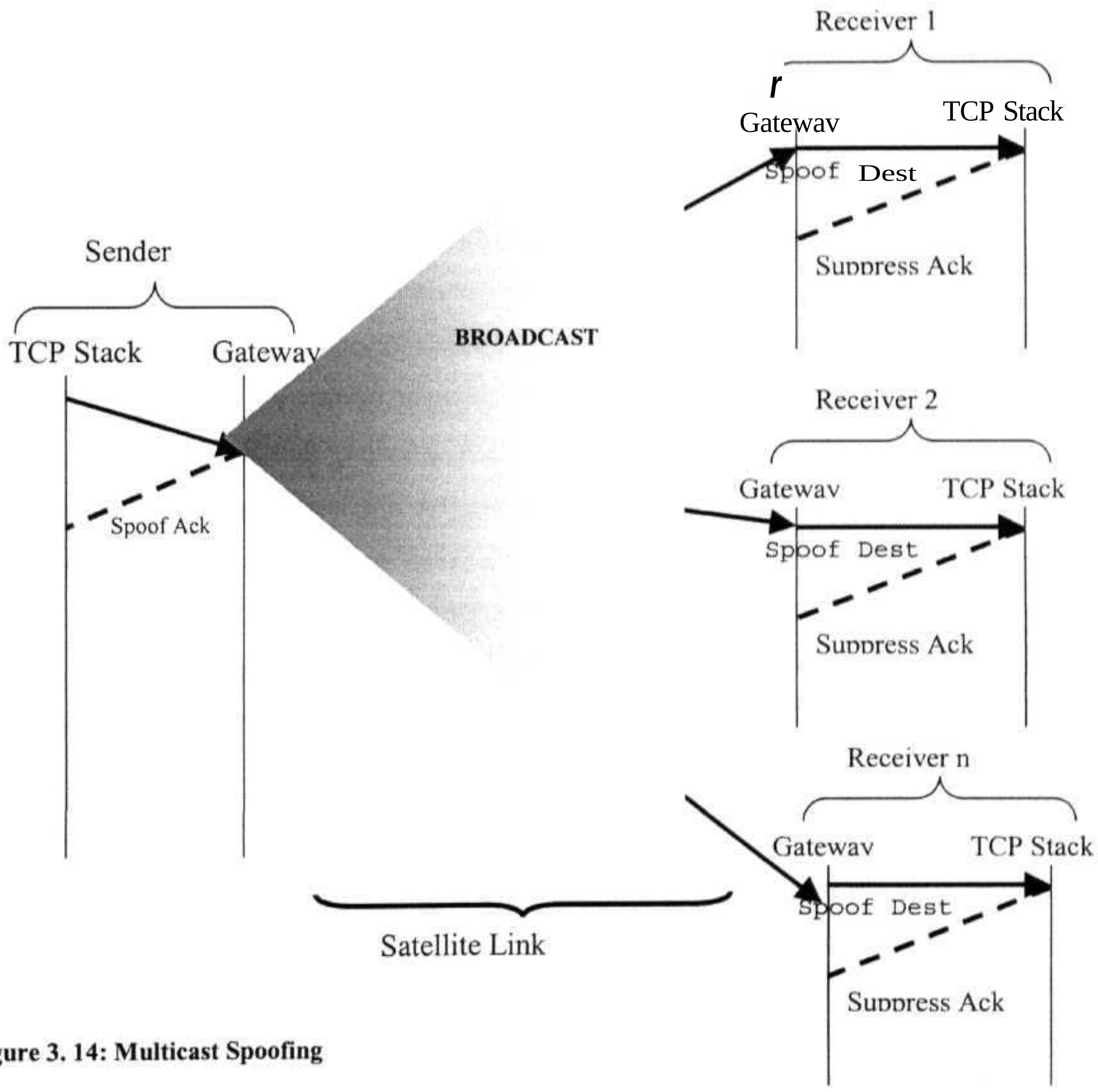


**Figure 3. 14: Multicast Spoofing**

At the receiver, the gateway rewrites (spoofs) the destination multicast address as the receiver's address before handing it over to the TCP layer. It also suppresses the acknowledgements. This way the sender keeps on pumping data packets and its assumed that all packets reach the receiver. In future this can be modified, such that one of the receivers only acknowledges the packets, to ensure that all packets reach the receivers.

The steps required at the sender side and client side to receive data in multicast mode are as given below.

**Server:** At server the following events take place to send data to receivers.

a. The sending application sends the information at URL http:\\service.com\filepush.
b. The URL is converted to an IP address whose incoming packets can be received by all recipients connected through the broadcast media.
c. The TCP stack at sender sends the packets as usual.
d. The gateway software generates a spoofed acknowledgement of the packet sent and sends back to the TCP stack.
e. TCP stack after receiving the acknowledgement releases the next packet.
f. The above process is repeated till all the packets are sent.

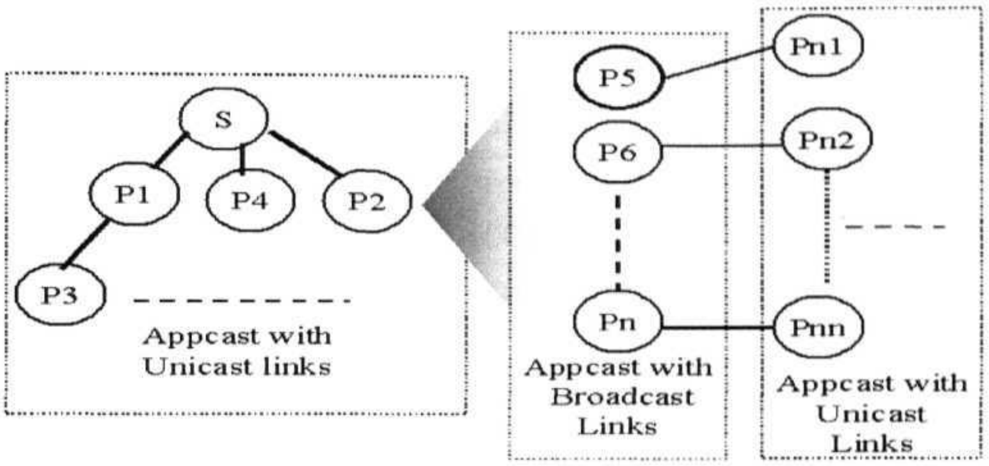
Receiver: At receiver the following events take place to receive data.

a. Receiver sets up a web service at the URL http:\\service.com\filepush on its own machine. This webservice can receive the information sent by the server.
b. Receiver gateway rewrites the packets destination IP address with its own IP address and hands over the packet to the TCP/IP stack of the receiver.
c. The service at receiver behaves as if the information has arrived in unicast mode.
d. The TCP/IP stack at receiver replies with an acknowledgement to the packets it received.
e. The gateway software at receiver suppresses those acknowledgments.

Both sender and receiver follow the above steps. While the gateway software takes care of rewriting the packets, generating and suppressing the acknowledgments; the application software is just not aware of the fact that they are **in** multicast mode i.e., application developers need not worry about the network specialization.

With the above multicast spoofing method available at proxies connected through broadcast medium, our proposed Appcast protocol creates topologies with both unicast and broadcast links as shown in figure 3.15.

Figure 3. **15:** Appcast overlay with unicast and broadcast links


S
P1
P4
P2
P3
Appcast with Unicast links
P5
P6
Pn
Appcast with Broadcast Links
Pn1
Pn2
Pnn
Appcast with Unicast Links

**3.8 Simulation and Performance Results of 'Multicast Spoofing'**

We simulated the satellite environment on an Ethernet LAN with 3 computers out of which one acted as sender and two as receivers. We used ndis3NT software written by Dan Lanciani, on windows 2000, which allows altering the packets before and after TCP/IP stack. Below we describe the crucial API provided by this software.

1) **nd_send_to_tcp(int unit, char *data, int len, int hlen)**

This function sends a frame to MSTCP just as if it had been received from the network. The len is the total length of the frame and hlen is the hardware header portion (e.g., 14 for Ethernet and emulations thereof)-

2) **nd_send_as_tcp(int unit, char *data, int len)**

This function is identical to nd_send_pkt except that ndis3pkt's tcp/ip multiplexor treats the frame as if it had been generated by MSTCP.

The ACCESS_FLAG_TUNNEL flag is applied to nd_access_type in the typelen argument. It indicates that the handle is to receive any packets that MSTCP sends on the interface. Only one handle on each interface can be opened in TUNNEL mode. As long

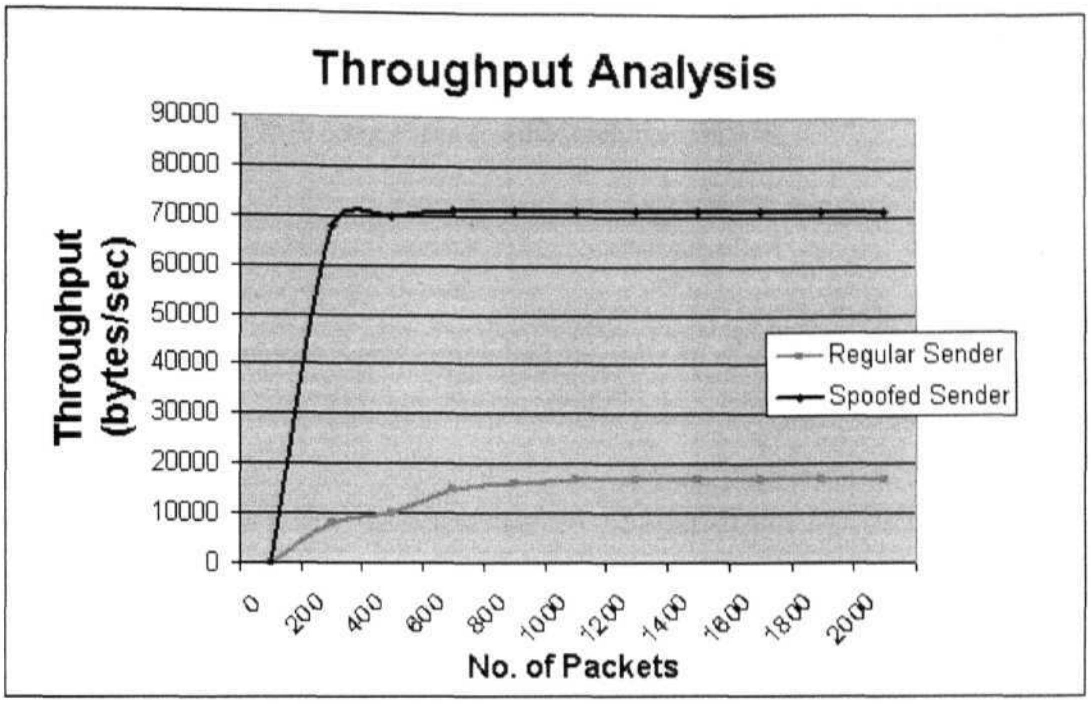
**as such** a handle is open, MSTCP's normal reception of packets is blocked (i.e., it receives packets only from nd_send_to_tcp) and all packets mat it sends are intercepted and rerouted to the handle. When the handle is released, MSTCP operation returns to normal.

The ACCESSJFLAGASMSTCP flag is applied to nd_access_type in the typelen argument. It indicates that, for purposes of ndis3pkt's tcp/ip multiplexor, received frames are treated as if they were destined for MSTCP.

A typical Win32 intermediate filter would create two handles, one with ACCESS_FLAG_TUNNEL and one with ACCESS_FLAG_ASMSTCP. It would read from the first handle, apply any desired changes to the output packets, and then forward them to the network with ndsendastcp. It would read from the second handle, apply and desired changes to the input packets, and then forward them to MSCTP with nd_send_to_tcp. Entire packets can, of course, be added or deleted from the stream.

We kept the delay as 250ms for regular sender to receive the acknowledgement, just to emulate the satellite environment i.e., receiver sends acknowledgement only after 250ms of receiving a packet from sender. We calculated throughput (bytes transferred per sec) in two cases - 1) with regular sender and 2) with spoofed sender. At the time of spoofing, we generated spoofed acknowledgements at sender and suppressed the actual acknowledgements at receiver itself. As the spoofing is at TCP stack level, packets can be sent transparently by any application. We sent packets of size 1500bytes using FTP protocol. We calculated the actual throughput, when we sent 200, 400 up to 2000 packets. The graph shows that throughput is less when few packets have been sent and is stabilized as number of packets grew in case of 'Regular Sender'. This is due to TCP's 3-way handshake mechanism. But in case of 'Spoofed Sender', throughput remained constant almost.

**Figure 3. 16: Throughput Analysis of Multicast Spoofing**



## 3.9 Conclusions

The proposed application level multicast protocols basically differ in the overlay topology creation and distribution of data over the same. While studying the existing protocols, it has been found that mesh based systems are complex to maintain and tree based systems give good performance and less control overhead. In both the tree-based systems i.e. TAG [22] and HMTP [13], new joining node traverses the tree from root, down the children. While TAG is relying on shortest path, HMTP relies on shortest distance. These features some times may lead to overlapping links. We proposed a new method that allows the joining node to select a parent, which is on its way to the source.

This chapter also described about problems faced using TCP in a satellite environment. TCP indeed deteriorates the actual capacity of a satellite link because of its window size, 3-way handshake etc. Failure of TCP over high bandwidth capacity with long latency clearly shows that TCP was not designed for a reliable and high-speed network. We found that spoofing is indeed beneficial for large file transfers. Spoofing's benefit to web servers and other content providers may be significant. In chapter 4, we show how we designed applications on Appcast that can cater information both in unicast and multicast mode, without any special effort from the developer to develop multicast applications. On the proposed new topology-building algorithm, we use SOAP [47-49] as application level transport mechanism and implement applications like Mass Information Push, Database Replication and E-Auctions. Chapter 4 gives details of these applications and application architecture with comparison to other architectures.

# Chapter 4

# Efficient Multicast E-Services over Appcast

In this chapter we show **the** design **and** development of multicast applications on the overlay created using the Appcast protocol in chapter 3. While chapter 3 focused on overlay creation, this chapter focuses on multicast application development. We show three kinds of applications: 1. Mass File Push is a static file transfer application **that** pushes files at regular intervals to the members of the overlay. 2. Database Replication is **an XML** file transfer application, which carries events **and** actions **in the file** and **the** members after receiving the file take actions at their end on their databases. 3. E-Auctions is an auction application that is interactive with the quotes and products information pushed on to member terminals in multicast mode and the individual bids for products by each member is carried back to e-auction server in unicast mode. We propose to use HTTP, SMTP and FTP as transport protocols; SOAP [18-19] as middleware and XML data exchange language in our design and development of these applications. Section 1 introduces the multicast application development and the middleware. Section 2 gives the generic multicast application development architecture over Appcast. Section 3 describes how the three applications listed earlier are implemented. Section 4 concludes the chapter.

## 4.1 Introduction

Applications are not able to exploit broadcast media property and reduce redundant packets' movement over network as the applications are written in unicast mode due to much complexity of multicast application programming support. In the paper titled 'Broadcast News' the author John **Hunt** explained how to write multicast applications. Below is the method of writing multicast applications in which a multicast server is set up that keeps sending data at regular intervals and clients **are** set up to receive the data sent by server. Both server and clients are set up using native TCP/IP socket programming in Java [76].

**The MulticastServer**

A multicast server can send data to a group of clients. Following steps show how to set up a multicast server.

1. Get an **I net** Ad dress for the group.
2. Create a **MulticastSocket** object on a specified local port.
3. Create a **DatagramPacket** object containing the data to be sent, the host and port.
4. Send the packet to the multicast socket to broadcast it.
5. Loop back to step 3.

**The ClientListener**

The client listens for receiving the multicast server data. Following steps show how a multicast client can be set up.

1. Get an **InetAddress** for the group.
2. Create a MulticastSocket object on a specified local port.
3. Join the multicast group.
4. Create a DatagramPacket containing a byte array into which the client will receive the data broadcast.
5. Call receive() to wait for a packet.
6. Query received datagram packet for the data sent.
7. Loop back to step 5.

The above-explained IP multicast is a very powerful and efficient way of broadcast information quickly and effectively to a large number of listeners (which may be changing dynamically). It does not tie the server to the clients or the clients to the server. Indeed the server could change and the clients need never know. Also, the clients can broadcast data, which cannot be controlled. It has its limitations, including being not available in every environment.

We are aiming at removing the programming complexities. For example, developer need not be aware of the multicast programming. He just develops the application as a normal unicast application. However, at the time of deployment the application can take advantage of the multicast based on the infrastructure available.

### 4.1.1 SOAP

To develop applications on any network or system, the developers require some kind of API - Application Programming Interface. Traditionally, sockets have been the interface between transport layer (TCP/IP) of underneath network and the application above the network. The Application Layer Multicast protocols discussed in chapter 2 built their own programming interfaces on top of TCP/IP using sockets. As the protocols like HTTP, SMTP and FTP on top of TCP/IP becoming the ubiquitous standard applications across enterprise, we look for a programming interface that uses these protocols i.e., instead of using low-level TCP/IP sockets, the high level protocol API of HTTP, SMTP etc are preferred. SOAP - Simple Object Access Protocol is one such protocol, which started gaining importance. In this section we describe SOAP and its evolution as middleware.

Introduced in 1999, specifying how to use XML and HTTP as an RPC like infrastructure, SOAP provides a standard object invocation protocol built on Internet standards, using HTTP/SMTP/FTP as the transport and XML for data encoding. SOAP [18,19] evolution into middleware classifies it into the Service Oriented Middleware as described below.

a) *POM - Procedure Oriented Middleware:* Enables the logic of an application to be distributed across the network. Program logic on remote systems can be executed as simply as calling a local routine. Remote Procedure Call packages from Sun RPC[15], DCE 1.1 RPC[16], Microsoft RPC fall into this category.
b) *MOM - Message Oriented Middleware* [17]: Provides program-to-program data exchange, enabling the creation of distributed applications. MOM is analogous to email in the sense it is asynchronous and requires the recipients of messages to interpret their meaning and to take appropriate action. IBM MQSeries, Microsoft's MSMQ, JMS - are examples of this type.
c) *OOM - Object Oriented Middleware:* Enables the objects that comprise an application to be distributed and shared across heterogeneous networks. CORBA[10] from OMG,

DCOM[12] based **on** Microsoft's COM[11], Java RMI [14] **fall into this** category. A **good** comparison of all these methods from a developer perspective **can** be from [13].

d) *SOM - Service Oriented Middleware [19,20]:* Enable **the** application services interact **with** each other. SOAP is fast becoming the **standard** for **this** application service oriented architecture.

SOAP is much flexible, giving application developers to choose any application protocol **like HTTP,** FTP, SMTP as the wire protocol. Developers use SOAP along with other web services protocols like WSDL (Web Services Definition Language) and UDDI (Universal Description, Discovery and Integration).

**Web Services Definition Language (WSDL)**

WSDL standard is an XML format for describing the network services and its access information. It defines a binding mechanism used to attach a protocol, data format, an abstract message, or set of endpoints defining the location of services. Usually a service provider creates Web services by generating WSDL from its exposed business applications. A public/private registry is utilized for storing and publishing the WSDL-based information.

**Universal Description, Discovery and Integration (UDDI)**

UDDI defines the standard interfaces and mechanisms for registries intended for publishing and storing descriptions of network services in terms of XML messages. In web services model, UDDI provides the registry for Web services to function as a service broker enabling the service providers to populate the registry with service descriptions and service types and the service requestors to query the registry to find and locate the services.

## 4.2 Design and Implementation

As stated earlier our approach depends on SOAP, HTTP, FTP and SMTP, which are all standard Internet protocols. Our aim is not to create any more protocols on top of these for setting up broadcast services. We define "Producer Service" as the *Server Service* that pushes the information, actions etc and "Consumer Service" as the *Client Service* that keeps listening to receive the information, actions etc from "Producer Service". Apart from the producer server and consumer server, we use one more component called

"Registry", that acts as a directory of producers and consumers. Any interested consumer can search this "Registry" and the consumer can subscribe to the service of his interest and set up his own "Consumer Service" to consume the information that is received from the "Producer Service". The "Registry" itself is a service like UDDI service that allows producer services to advertise their service specifications. While this registry is a global service, every producer service also keeps a registry containing the consumer information.

### 4.2.1 Producer Service Registry Information

Every "Producer Service" gives the following details to the "Registry" to advertise its service. The "Registry" service assigns a unique "Service Id" to identify the service.

**Service ID:** This ID is automatically generated by "Registry" system to identify the service.

**Name:** Every service has to identify itself with user-friendly service name that can easily be searched by the consumers. In our applications the names are given like Getfile, GetAuction, GetTrans etc

**Description:** A detailed description of the service in normal plain text describing how the service can be utilized, what can be done with the information that the service delivers to consumer, how to set up the corresponding consumer service to receive the information etc.

**Started On:** The Date from which onwards the service is available.

**Schedule:** Schedule of the service to push the information at regular intervals.

**Broadcast Address:** It is the network Broadcast address of the producer service.

**URI:** The URI (Uniform Resource Identifier) with which the "Consumer Service" would be identified, i.e. consumers have to setup their service as this URI only if they want to receive information in broadcast mode.

**Request Type:** Type of the protocol using which the service pushes the information, like HTTP, SMTP, and FTP etc.

**Response Type:** Type of the protocol using which the consumer service can send response, like HTTP, SMTP, and FTP etc.

**WSDL:** A file, that contains the Service Description Language written **as** per the WSDL specifications.

**Contact Address:** Service Administrator's postal address.

**E-Mail Address:** E-mail addresses to which queries and responses can be sent.

**Other Info:** Any other information that the service likes to inform its consumers.

### 4.2.2 Consumer Service Registry Information

Consumers search for the "Producer Service" in the global "Registry" service, look at its details, set up their corresponding consumer service and subscribe for it with the registry maintained at producer. Following information for each consumer is kept at producer.

**Service UniqueID:** The Producer Service's ID, the consumer has selected.

**ConsumerUniqueID:** Generated automatically by producer system

**Name:** Name of the consumer / organization.

**Broadcast/Unicast?** Whether subscribing for Broadcast service or Unicast service?

**Unicast Address:** Unicast address to be used by "Producer Service" if the consumer has subscribed for Unicast mode or if the information is lost while sending through broadcast.

**Contact Address:** Consumer Service Administrator's contact postal address.

**E-Mail Address:** E-mail addresses for contact.

**Other Info:** Any other information that the "Producer Service" likes to know about its consumers.

### 4.2.3 The Basic Steps

Below we show the steps involved for producer server and consumer server to communicate within an Appcast environment.

1. The producer server publishes/advertises that it has set up a service - S, that pushes information in XML form at regular intervals. It also advertises the *Service Description Language* of the corresponding consumer service using which the consumers can set up their service.

2. It also advertises that it is on a broadcast medium and those who want to consume this service on broadcast should set up service with the name URI - Http://braodcastadress/servicename......
3. The clients (consumers) who want to consume that service now can set up a service - C such that it can receive the content sent by the server and in response can send a mail or call an acknowledgement service hosted at producer.
4. The clients now can register with the service - S, giving their uni-cast IP address, so that the server can resend the content in point-to-point model in case the client could not send the acknowledgment.
5. If C has many clients further down on its broadcast network, it can simply make it self as proxy service - P to S as shown below.

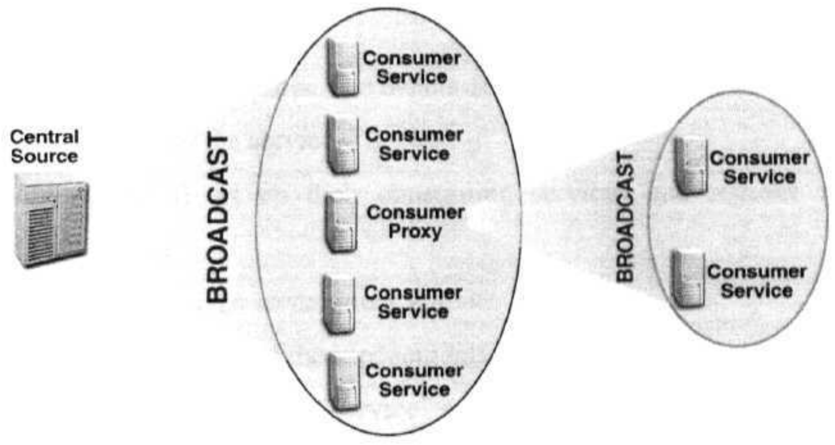


Figure 4. 1 Proxy Chaining

6. P advertises the same and clients down P can set up their own services. Like this the chain can go further down.
7. Those who are not on broadcast medium at any level of the chain can register with the service as point-to-point communications.

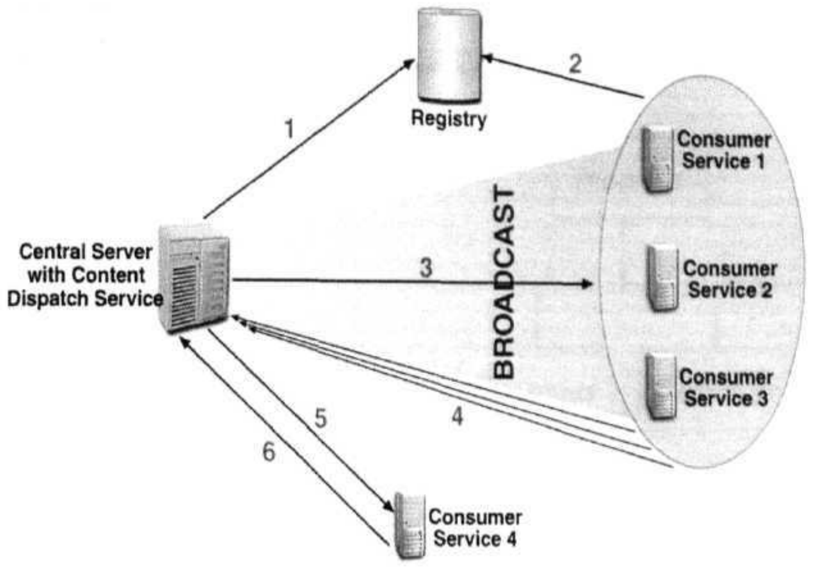


**Figure 4. 2 Appcast Application Architecture**

The figure 4.2 shows the flow of events. The events are as described below.

1. Producer advertises the service.
2. Consumers (1,2,3) set up their consuming services and register with the "Producer Service".
3. Producer broadcasts the content to 1,2,3.
4. Consumers 1,2,3 send acknowledgements as a mail using standard SMTP protocol or call "acknowledge service" hosted at producer.
5. Consumer 4 registers itself as unicast, and receives data using point-to-point communication.
6. Consumer 4 sends acknowledgement as mail using SMTP or calls "acknowledge service" hosted at producer.

## 4.3 Implementation

In all the examples of our implementation, we used the producer as the original information provider. Consumers can subscribe to producer for their own consumption or for distributing the information further down. We call these kinds of consumers as distributors. The distributors are almost equal to producers except that they themselves cannot produce any content. Once distributors receive content from **the** original producer, then only can they distribute further down. The following figure 4.3 depicts this scenario.

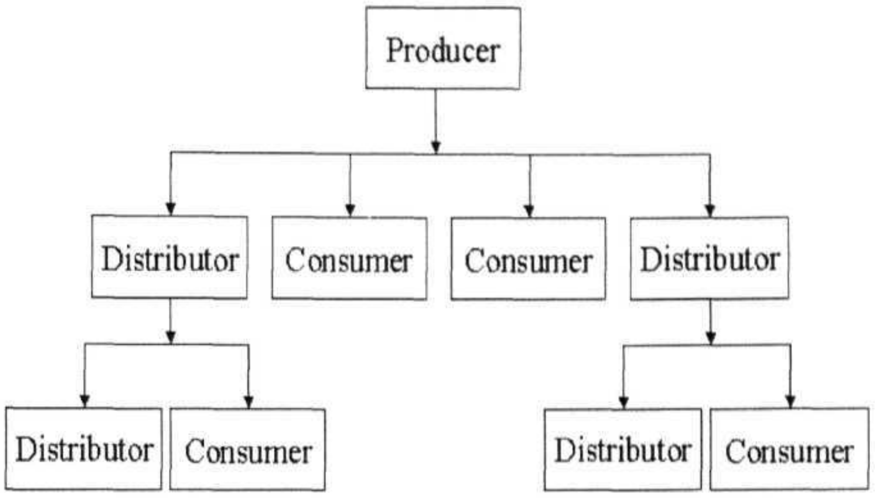
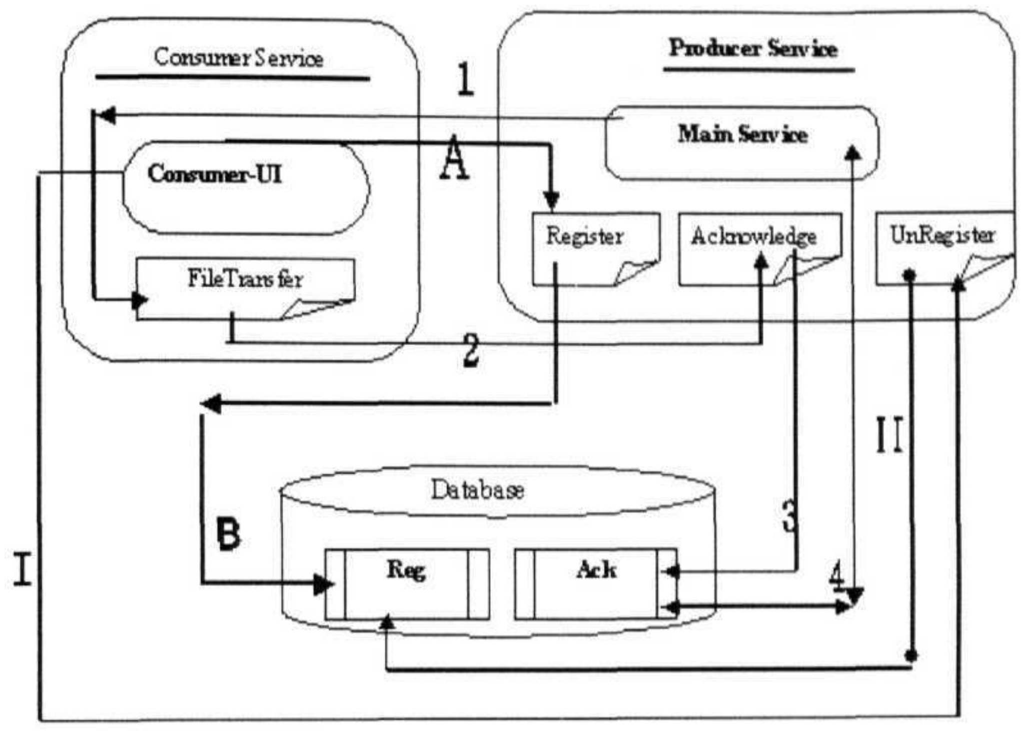


Figure 4. 3 Producer-Distributor-Consumer Hierarchy

### 4.3.1 Mass Information Push

The consumer can set up the "GetFile" consumer service from the information advertised by the "Producer Server" such as the corresponding GetFile WSDL, the broadcast address, Service's e-mail address and the UR1. The figure 4.4 depicts the design and flow of "GetFile" service. The consumer contacts the registry of the producer, and registers himself. Once it registers, a virtual directory will be created at the consumer and a dynamic web service will be created at the consumer as per the producer's specifications like the UR1, input parameters, output parameters etc and configures the web service to receive the file at scheduled intervals. It allows the consumer to modify his registration or unregistered also. The figure 4.5 shows the sequence of events for registration.

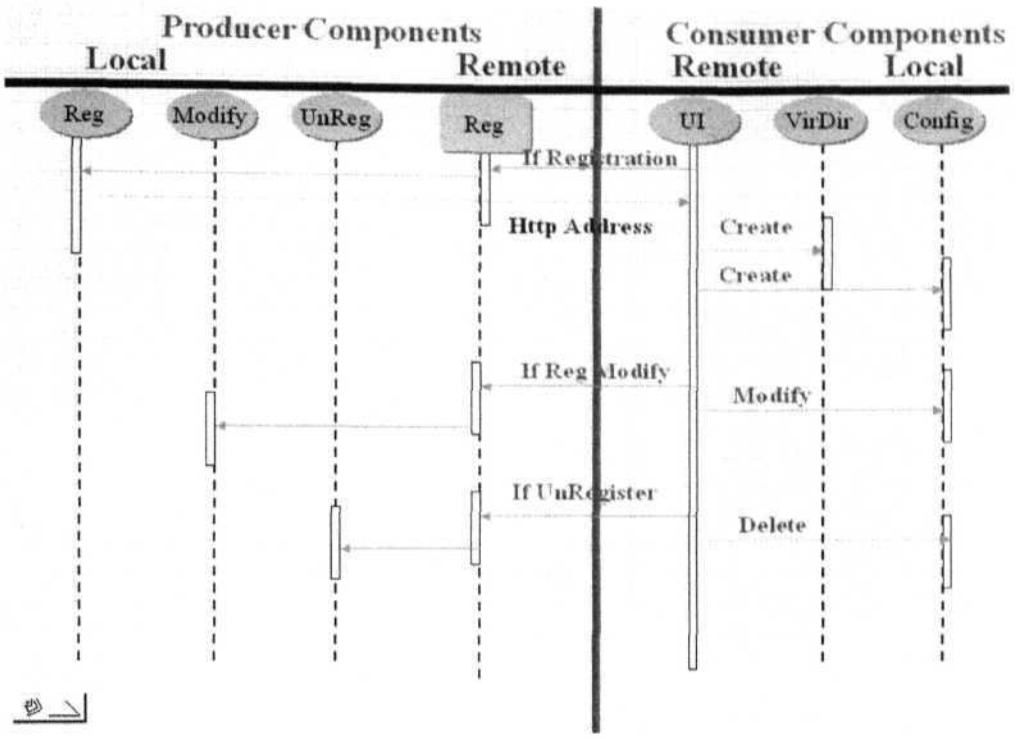
Figure 4. 4 GetFile Service



***File Transfer Series: 1→2 →3 →4***

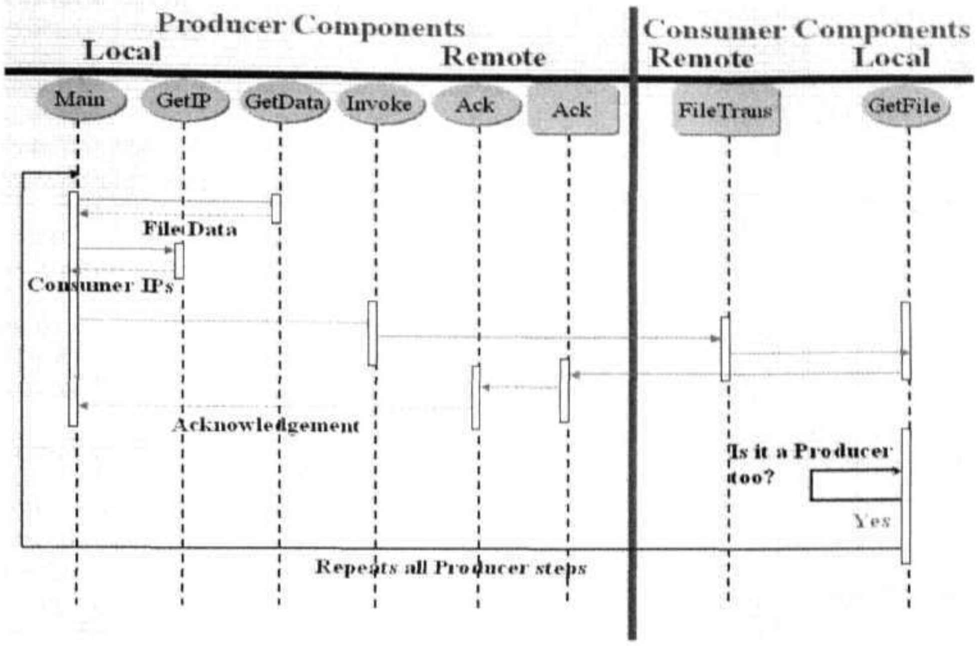
The consumer can register either as distributor or just as a consumer. In case he registers as a distributor, then the web service created at the consumer will be capable of redistributing the file to the consumers down it.

**Figure 4. 5 Consumer Registration**



The producer gets the data file that has to be pushed, by calling the local "GetData" procedure and gets all the URI of the consumers to whom it should push the data, by calling "GetIP" local procedure. Then it calls the remote "FileTrans" procedure, a web service hosted at consumer, which in turn calls its local "Getfile" procedure that takes action (like storing) on the received file. The "Getfile" procedure checks whether its distributor and if so, it repeats all the steps as the producer. The figure 4.6 shows the sequence diagram show the file push events.

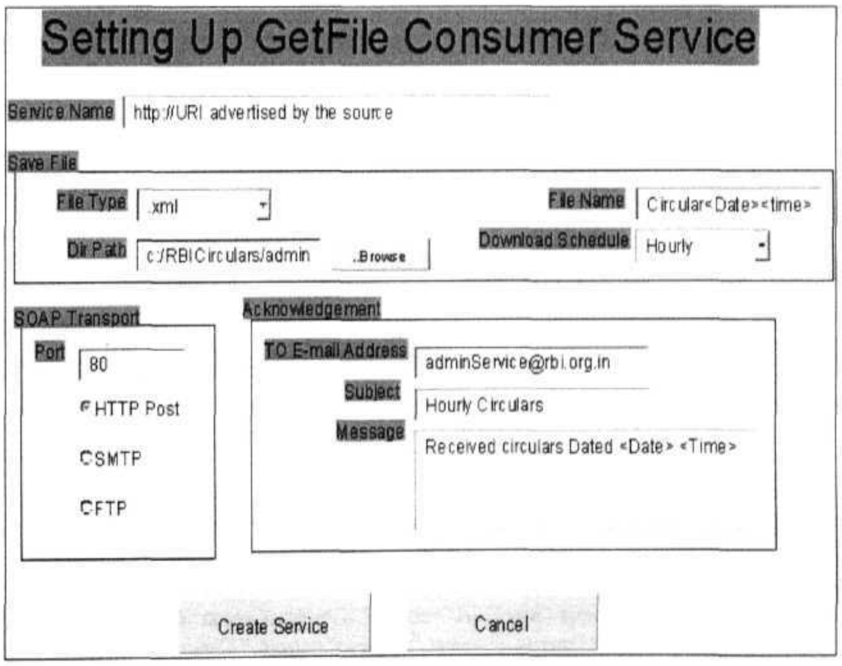
**Figure 4. 6 Sequence Diagram of GetFile**



### Setting up consumer service

To set up a consumer service, the consumer has to download the corresponding WSDL file hosted at the producer. From the WSDL file, the consumer computer can display the following wizard. Once the required fields are specified, a web service at the consumer side is created automatically.

Figure 4.7 Getfile Wizard



Setting Up GetFile Consumer Service

Service Name: http://URI advertised by the source

Save File

File Type: xml

File Name: Circular<Date><time>

Dir Path: c:/RBICirculars/admin

Download Schedule: Hourly

SOAP Transport

Port: 80

● HTTP Post

○ SMTP

○ FTP

Acknowledgement

TO E-mail Address: adminService@rbi.org.in

Subject: Hourly Circulars

Message: Received circulars Dated <Date> <Time>

**GetFile Consumer Service:**

```
<%@ WebService Language="c#" Class="GetFile" %>
using System;
using System.IO;
using System.Web;
using System.Web.Mail;
using System.Web.Services;

[WebService]
public class GetFile:WebService {

[WebMethod]
public void getFile(string fileType, string content) {
/ ** Type of the file and content are parameters to this web method **/

/** Storing the File **/
 String FILE_NAME = "C:\\WS\\upload\\getFile"+"."+fileType;
    StreamWriter sw = File.CreateText(FILE_NAME);
    sw.WriteLine (content);
    sw.Close();

/** Sending E-Mail **/
 MailMessage msg=new MailMessage();
 SmtpMail.SmtpServer="mailidrbt";
 msg.From = " consumer1@idrbt.ac.in";
 msg.Subject = "Acknowledgement:";
 msg.To= getFileservice@idrbt.ac.in;
 msg.Body=content;
 msg.BodyFormat = MailFormat.Html;
 SmtpMail.Send(msg);
 Server.ClearError();
}
}
```

The received content is stored as file.

Prepare the acknowledgement and send the mail to the server.

**GetFile WSDL file:**

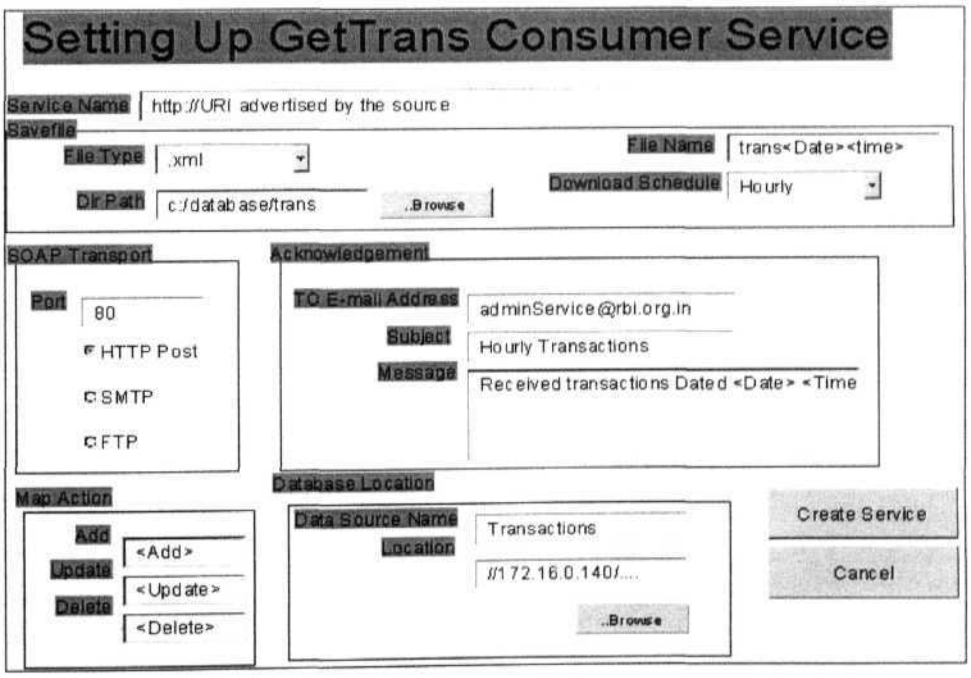
```
<?xml version="1.0" encoding="utf-8"?>
<definitions
xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:s="http://www.w3.org/2001/XMLSchema"xmlns:s0="http://tempuri.org/"xmlns:soapenc="http://sch
emas.xmlsoap.org/soap/encoding/"xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/" targetNamespace="http://tempuri.org/"
xmlns="http://schemas.xmlsoap.org/wsdl/">
<types>
<s:schema elementFormDefault="qualified" targetNamespace="http://tempuri.org/">
<s:element name="getFile"><s:complexType>
<s:sequence><s:element minOccurs="0" maxOccurs="1" name="emailID" type="s:string" />
<s:element minOccurs="0" maxOccurs="1" name="fileType" type="s:string" />    <s:element
minOccurs="0" maxOccurs="1" name="content" type="s:string" />
</s:sequence> </s:complexType></s:element>
<s:element name="getFileResponse">    <s:complexType /></s:element></s:schema>
```

```
</types>
<message name="getFileSoapIn"><part name="parameters" element="s0:getFile" />
</message><message name="getFileSoapOut">
<part name="parameters" element="s0:getFileResponse"/> </message>
<portType name="GetFileSoap"><operation name="getFile"><input message="s0:getFileSoapIn"
/><output message="s0:getFileSoapOut" /></operation>
</portType><binding name="GetFileSoap" type="s0:GetFileSoap">
<soap:bindingtransport="http://schemas.xmlsoap.org/soap/http" style="document" /><operation
name="getFile"><soap:operation soapAction="http://tempuri.org/getFile" style="document"
/><input><soap:body use="literal" /></input><output><soap:body use="literal"
/></output></operation></binding>
<service name="GetFile"> <port name="GetFileSoap"binding="s0:GetFileSoap">
<soap:addresslocation="http://localhost/ws/getfile/getfile.asmx" /></port></service>
</definitions>
```

### 6.4.2 Database Replication

Just like the previous application, in this application, the consumer can set up a "GetTrans" consumer service with the information advertised by the "Producer Server". In our example, we just used a simple XML file as database transaction file that includes database actions like 'add', 'update', and 'delete' along with data. Once the XML file with all data and actions is received, the consumer acts upon the data using the actions specified against its own database. Lot of work on standardizing the XML with database events embedded is going on [19].

Figure 4.8: Database Replication Wizard



Setting Up GetTrans Consumer Service

Service Name: http://URI advertised by the source

Savefile
- File Type: .xml
- Dir Path: c:/database/trans
- File Name: trans<Date><time>
- Download Schedule: Hourly

SOAP Transport
- Port: 80
- [✓ HTTP Post] SMTP FTP

Acknowledgement
- TO E-mail Address: adminService@rbi.org.in
- Subject: Hourly Transactions
- Message: Received transactions Dated <Date> <Time

Map Action
- Add: <Add>
- Update: <Update>
- Delete: <Delete>

Database Location
- Data Source Name: Transactions
- Location: //172.16.0.140/....

**GetTrans Consumer Service:**

```
<%@ WebService Language="c#"  Class= "Transactions" %>
using System;
using System.Data;
using System.Data.SqlClient;
using System.Web.Services.Protocols;
using System.Web;
using System.Web.Mail;
using System.IO;
using System.Web.Services;
public struct AddTrans
{
        public String AccountNumber;
        public String TransactionID;
        public String TransactionType;
        public String Amount;
}
public struct AddCust
{
        public String AccountNumber;
        public String CustomerName;
        public String Address;
        public String Balance;
}

[WebService]
public class Transactions : System.Web.Services.WebService
{
         public String strToParse;
         public AddTrans at;
         public AddCust ac;
         public String TableName;
[WebMethod]
public void GetTrans(String strSource)
{
/** Storing the File **/
 String FILE_NAME = "C:\\WS\\upload\\trans.xml";
    StreamWriter sw = File.CreateText(FILE_NAME);
    sw.WriteLine (strSource);
    sw.Close();
try{
 String strAdd;
 String strDelete;
 String transID;
 String strModify;
 int iPos; /* Initial position */
 int ePos; /* End Position */
 strToParse=strSource;
 TableName=getValue("<Table>");

/** Add operation **/
iPos=strSource.IndexOf("<Add>");
```

Store the received file for log purposes

Filter the file for 'add' operation

```
ePos=strSource.IndexOf("</Add>");
strAdd=strSource;
while(iPos!=-1)
{
        strToParse=strAdd.Substring(iPos,ePos-iPos);
        if(TableName=="TransTable")
        {                                                              Add record to
                AddTrans ad=new AddTrans();                            'Transaction'
                at.AccountNumber=getValue("<AccountNumber>");          table
                at.TransactionID=getValue("<TransactionID>");
                at.TransactionType=getValue("<TransactionType>");
                at.Amount=getValue("<Amount>");
                AddDetails();
        }
        if(TableName=="CustTable")                                     Add record to
        {                                                              'Customer'
                ac=new AddCust();                                      table
                ac.AccountNumber=getValue("<AccountNumber>");
                ac.CustomerName=getValue("<CustomerName>");
                ac.Address=getValue("<Address>");
                ac.Balance=getValue("<Balance>");
                AddDetails();
        }
        strAdd=strAdd.Substring(ePos+5,strAdd.Length-ePos-5);
        iPos=strAdd.IndexOf("<Add>");
        ePos=strAdd.IndexOf("</Add>");
}
/** Delete Operation **/
 iPos=strSource.IndexOf("<Delete>");
 ePos=strSource.IndexOf("</Delete>");
 strDelete=strSource;
 String keyToDelete;
 String str="";
 while(iPos!=-1)
{
        strToParse=strDelete.Substring(iPos,ePos-iPos);
         if(TableName=="TransTable")                               Delete record from
                keyToDelete=getValue("<TransactionID>");           'Transaction' table
        else
                keyToDelete=getValue("<AccountNumber>");
        Delete(keyToDelete);
        strDelete=strDelete.Substring(ePos+8,strDelete.Length-ePos-8);
        iPos=strDelete.IndexOf("<Delete>");
        ePos=strDelete.IndexOf("</Delete>");
}

/** Sending Mail **/
//Now send the mail.
```

**GetTrans WSDL file:**

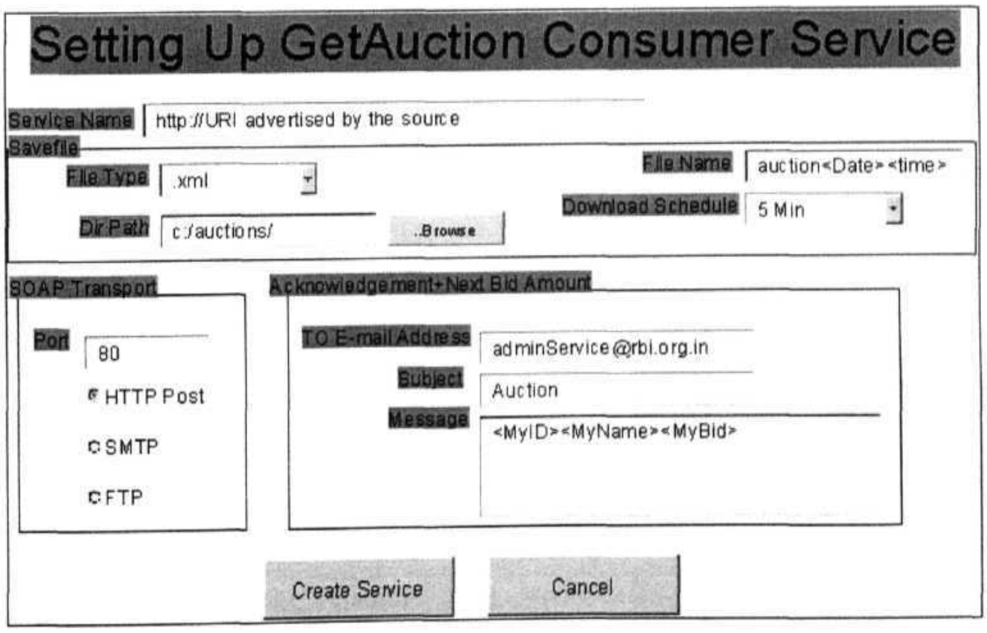
```
<?xml version="1.0" encoding="utf-8"?>
<definitionsxmlns:http="http://schemas.xmlsoap.org/wsdl/http/"xmlns:soap="http://schemas.xmlsoap.org/
wsdl/soap/"xmlns:s="http://www.w3.org/2001/XMLSchema"xmlns:s0="http://tempuri.org/"xmlns:soapenc
="http://schemas.xmlsoap.org/soap/encoding/"xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"x
mlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"targetNamespace="http://tempuri.org/"xmlns="http:/
```

/schemas.xmlsoap.org/wsdl/"><types><s:schema elementFormDefault="qualified"
targetNamespace="http://tempuri.org/"> <s:element name="GetTrans">
<s:complexType><s:sequence><s:element minOccurs="0"maxOccurs="1"name="strSource"
type="s:string" /></s:sequence> </s:complexType></s:element><s:element
name="GetTransResponse"><s:complexType>
<s:sequence><s:element minOccurs="0" maxOccurs="1" name="GetTransResult"
type="s0:ArrayOfString" /></s:sequence>
</s:complexType></s:element><s:complexType name="ArrayOfString"><s:sequence>     <s:element
minOccurs="0" maxOccurs="unbounded" name="string" nillable="true" type="s:string" /></s:sequence>
</s:complexType><s:element name="ArrayOfString" nillable="true" type="s0:ArrayOfString"
/></s:schema>
</types><message name="GetTransSoapIn">
<part name="parameters"element="s0:GetTrans" /></message><message
name="GetTransSoapOut"><part name="parameters"element="s0:GetTransResponse"
/></message><portType name="TransactionsSoap"><operation name="GetTrans"><input
message="s0:GetTransSoapIn" /><output message="s0:GetTransSoapOut" /></operation>
</portType><binding name="TransactionsSoap"type="s0:TransactionsSoap"><soap:binding
transport="http://schemas.xmlsoap.org/soap/http" style="document" /><operation
name="GetTrans"><soap:operation soapAction="http://tempuri.org/GetTrans" style="document"
/><input><soap:body use="literal" /></input><output><soap:body use="literal"
/></output></operation></binding>
<service name="Transactions"><port
ame="TransactionsSoap"binding="s0:TransactionsSoap"><soap:address
location="http://localhost/ws/transactions/transactions.asmx" /></port></service></definitions>

### 6.4.3 Open Auctions

Participating in auctions is an interactive activity. The consumer has to establish 2 services, one through which he sets up his information receiving "Consumer Service", and the other through which he views the information and sends the response.

**Figure 4.9: Auction Service Wizard**



Setting Up GetAuction Consumer Service

Service Name: http://URI advertised by the source

Savefile

File Type: .xml

Dir Path: c:/auctions/

File Name: auction<Date><time>

Download Schedule: 5 Min

SOAP Transport

Port: 80

[✓ HTTP Post]

SMTP

FTP

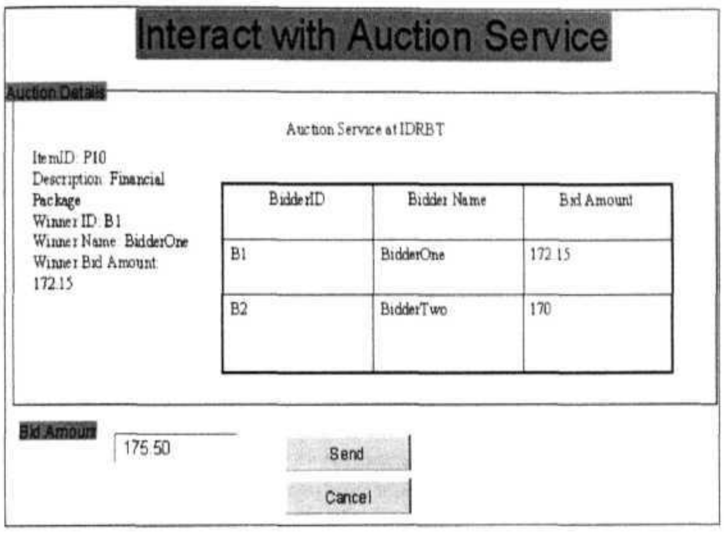
Acknowledgement+Next Bid Amount

TO E-mail Address: adminService@rbi.org.in

Subject: Auction

Message: <MyID><MyName><MyBid>

**Figure 4.10: Interact with Auction service**



Interact with Auction Service

Auction Details

Auction Service at IDRBT

ItemID: P10
Description: Financial Package
Winner ID: B1
Winner Name: BidderOne
Winner Bid Amount: 172.15

| BidderID | Bidder Name | Bid Amount |
|---|---|---|
| B1 | BidderOne | 172.15 |
| B2 | BidderTwo | 170 |

Bid Amount 175.50

Send

Cancel

**GetAuction Consumer Service:**

```
<%@ WebService Language="c#" Class="Auction" %>
using System;
using System.IO;
using System.Web;
using System.Web.Mail;
using System.Web.Services;

[WebService]
public class Auction:WebService {

[WebMethod]
public String GetAuction(string content) {

/** Storing the File **/
 String FILE_NAME = "C:\\WS\\upload\\Auction.xml";
    StreamWriter sw = File.CreateText(FILE_NAME);
sw.WriteLine (content);
sw.Close();

Display content;
try{
/** Sending E-Mail **/
 MailMessage msg=new MailMessage();
 SmtpMail.SmtpServer="mailidrbt";
 msg.From = " getaucservice@idrbt.ac.in";
 msg.Subject = "Auction Ack:";
 msg.To=consumer1 @idrbt.ac.in;
 msg.Body=content;
 msg.BodyFormat = MailFormat.Text;
 SmtpMail.Send(msg);
 Server.ClearError();
```

Store the auction.xml for audit purpose

Show the bids

Send the mail as acknowledgement

```
 return "Mail is sent";
 }catch(Exception e){
  return e.Message;
 }}}
```

**GetAuction WSDL File:**

```
 <?xml version="1.0" encoding="utf-8"?>
<definitions
xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"x
mlns:s="http://www.w3.org/2001/XMLSchema"xmlns:s0="http://tempuri.org/"xmlns:soapenc="http://sche
mas.xmlsoap.org/soap/encoding/"xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"xmlns:mime="
http://schemas.xmlsoap.org/wsdl/mime/"targetNamespace="http://tempuri.org/"xmlns="http://schemas.xmls
oap.org/wsdl/"><types><s:schema
elementFormDefault="qualified"targetNamespace="http://tempuri.org/"><s:element
name="GetAuction"><s:complexType>  <s:sequence><s:element minOccurs="0"maxOccurs="1"
name="emailID" type="s:string" /><s:element minOccurs="0" maxOccurs="1" name="content"
 type="s:string" /></s:sequence></s:complexType>   </s:element><s:element
 name="GetAuctionResponse">  <s:complexType><s:sequence><s:element
 minOccurs="0"maxOccurs="1"name="GetAuctionResult" type="s:string" /></s:sequence>
 </s:complexType></s:element><s:element name="string" nillable="true" type="s:string" />
 </s:schema></types><message name="GetAuctionSoapIn"><part name="parameters"
 element="s0:GetAuction" /> </message><message name="GetAuctionSoapOut"><part
 name="parameters"element="s0:GetAuctionResponse" /></message><portType name="AuctionSoap">
 <operation name="GetAuction"><input message="s0:GetAuctionSoapIn" /><output
 message="s0:GetAuctionSoapOut" />  </operation></portType><binding name="AuctionSoap"
 type="s0:AuctionSoap">
 <soap:bindingtransport="http://schemas.xmlsoap.org/soap/http" style="document" /><operation
 name="GetAuction"><soap:operation soapAction="http://tempuri.org/GetAuction" style="document"
 /><input><soap:body use="literal" /></input><output><soap:body use="literal"
 /></output></operation></binding>
 <service name="Auction"><port name="AuctionSoap" binding="s0:AuctionSoap"> <soap:address
 location="http://localhost/ws/Auction/Auction.asmx" /></port></service></definitions>
```

## 6.5 Advantages and Disadvantages of the model

Since our approach is at application level, we compare our method with other application level broadcast approaches. Figure 4.11 clarifies the simplicity of our approach. While protocols like Overcast [5]and Scattercast [3-4] had built some more new protocols - Gossamer, Up/Down protocols etc on top of TCP/IP; Appcast relies on its Appcast overlay on which SOAP is used as wire protocol.

√ The receivers need not write applications especially for broadcast as in traditional IP Multicast.

V No special infrastructure deployment is required as in Overcast [5] or Scattercast [3-4].

√ Customization wizards can be set up by commercial vendors such that consumers can very easily set up consumer service without need to write a code as all applications follow the standard protocols like SOAP, HTTP, SMTP etc. Using Proxy chaining is exclusively in the hands of network administrators.

√ Controlling is very easy.

√ No need to look into router level network outages etc, as all data loss and resending is controlled by application level utilities like HTTP and SMTP.

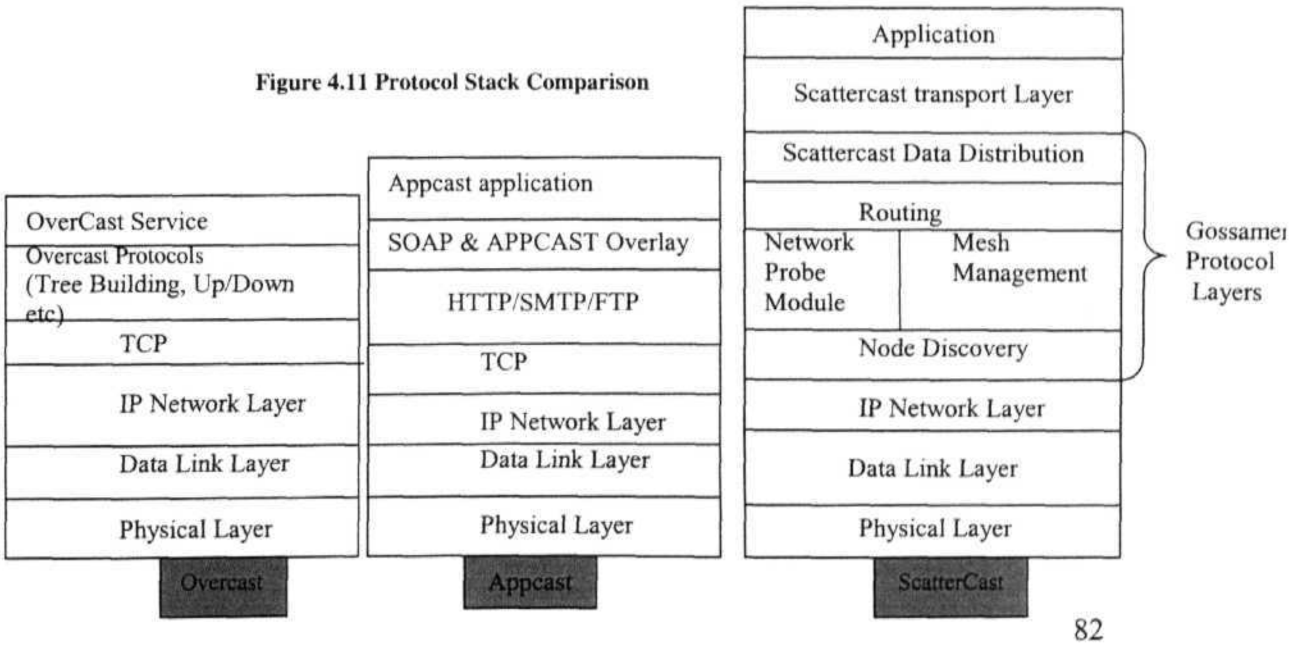
V Appcast environment can be build incrementally.

x The model is taking advantage of "TCP Spoofing" which may not be acceptable to all. But in a closed environment and between a parent and just it's next level client node, this can be enabled without any problem.

x The model assumes that the communication between sender and receiver is reliable. We are taking advantage of broadcast media between sender and receiver of same network. On LANs and Satellite based WANs communication media is reliable. If the sender and receiver are on different networks, with so many networks in between, the communication is just like point-to-point. And for this, the sender need not write application separately.

x Though the source sends the information, it receives acknowledgements from all receivers unlike in other multicast protocols. Since we use application level features, unlike others, who control information-flow at packet level and hence no application level acknowledgement.

**Figure 4.11 Protocol Stack Comparison**



## 4.4 Conclusion

Multicast push-services increase network efficiency. In our model, though we could make the same application push the data in both unicast and multicast modes, we could not actually install E-Service on consumer machine dynamically. We did the whole experiments in our LAN environment and we wrote the consumer services separately for each application as installable executables. We discuss below some issues in setting up consumer services.

**How to Set Up Consumer Service?**

Normally setting up an E-service in a web-service paradigm is to receive a request and send a response. Even the web sites behave this way. Majority of the times, the request is just like a query, which is few bytes from the initiator and response is like sending a list of results (may be huge data), which is from the service. The initiator need not be connected always to Internet/network. However, the service must be always up listening for the requests from consumers. In multicast, it is totally in contrast. The initiator (producer) pumps large amounts of data. The consumers must always be up and connected to network, listening for the service (producer) to push the data and the consumers just send an acknowledgement as response.

**Dynamic E-Service**

How the consumers can set up the listening service? What do they do with the information they receive? Unless there is some natural language, it is difficult to specify these things. In general, the WSDL specifies only the input, output parameters, the remote service names, the ports at which these services listen etc. But it cannot specify how to act upon those parameters. But the service provider is aware of how to process the input parameters and arrive at the result to send the response. So, here the service provider informs the consumers, how to call its service. In multicast, it is different in that the consumer should already be having the consumer service to which the producer can push the information. Unless, all consumers set up their receiving services in same way, the producer cannot push the information. So, he advertises the service definition in terms of input and output parameters, just like in WSDL. However, the difference is that, in normal course the consumers call the producer service and in multicast, the producer calls the consumer's service. The consumer has to set up its service dynamically by looking at the WSDL.

E-Service Languages

As discussed earlier, WSDL is just like an Interface Definition Language. It cannot describe the E-Service in totality. In a multicast environment, one needs a language that is common on all platforms, such that once described, it can be implemented on any platform. Work in this direction of describing a web service in XML and executing on any platform has been described in [52]. It details on program statements, expressions, variables etc. In our case of multicasting, this work may help if every consumer would like to use the received information in similar fashion. This can be applied in case of distributors, where in producer can program how the information be distributed. If so, the producer can place the service in XL language [52], which the distributor can download. Digital signatures can be used to authenticate the code. However, if the consumers would like to use the received information in different fashions, then this will not help. The producer can dictate what the consumer can receive, but not how to consume the same. So, we feel that, the programming languages for the web services paradigm must vary from simple to complex. For example, it should allow simple statements like below to receive the office order, store it and display or mail it.

```
{
Receive Office Order From http://www.organization.com/orders/
Store Office Order In d:\folder
Display Office Order
}
{
Receive Office Order From http://www.organization.com/orders/
Mail Office Order admin@myoffice.com
}
```

# Chapter 5

# Exploiting Query Redundancy

In this chapter, we present a new XPath [78] processor. Chapter 4 detailed how to develop multicast applications on Appcast. Appcast uses XML [80] to exchange data and SOAP [18-19] as middleware. XPath, short for XML Path language, is a querying language used to select specific parts of very large XML documents. Many algorithms have been proposed for processing multiple XPath expressions in the context of streaming XML documents such as Xfilter[82], Yfilter[83], IndexFilter[85], Xtrie[86] and CQMC[84]. But these algorithms do not handle backward axes like parent and ancestor. XAOS[79] deals with XPath expressions with forward and backward axes. But XAOS handles only one query at a time. As a result a document is parsed *q* times for a set of *q* queries. In case of large documents, this parsing time goes out of bounds. More over, research suggests that there is bound to be significant commonality among different queries in a large-scale system. By exploiting this commonality we can reduce a lot of processing and runtime memory consumption. In this chapter we present a method called YALXP (Yet Another Light Weight Xpath Processor), to evaluate multiple queries, with both forward and backward axes, in a single pass exploiting commonality among the queries. YALXP is built upon XAOS algorithm. We make the following contributions in this approach.

1. A concise representation of all XPath expressions, with out loss of information, called combined x-dag, where all backward constraints in all the queries are converted into forward constraints.
2. Procedure to create combined x-dag.
3. Data structure called combined matching structure that represents all the matches of all XPath expressions in the document.
4. Method to evaluate Xpath queries using the above structures.

Section 1 introduces XML and its related processing languages like XPath, parser like SAX [104] and DOM [91]. Section 2 surveys the algorithms proposed for efficient XPath processing. Section 3 presents YALXP – Yet Another Light Weight XPath Processor proposed by us. Section 4 simulates YALXP, XAOS [79] and compares the results. Section 5 concludes this chapter.

## 5.1 Introduction

Search tools till date have insufficient capabilities to keep pace with the information generated. Particularly getting right and relevant information has become a nightmare. In this scenario, SDI - Selective Data Information Dissemination, a popular concept used by libraries is gaining importance. In SDI, users register with servers that are nearer to them with their interests - profile. Based on the profiles, the servers filter right information and push the same to the user. Definitely, a centralized server cannot scale up to the requirements of large number of users spread all over Internet.

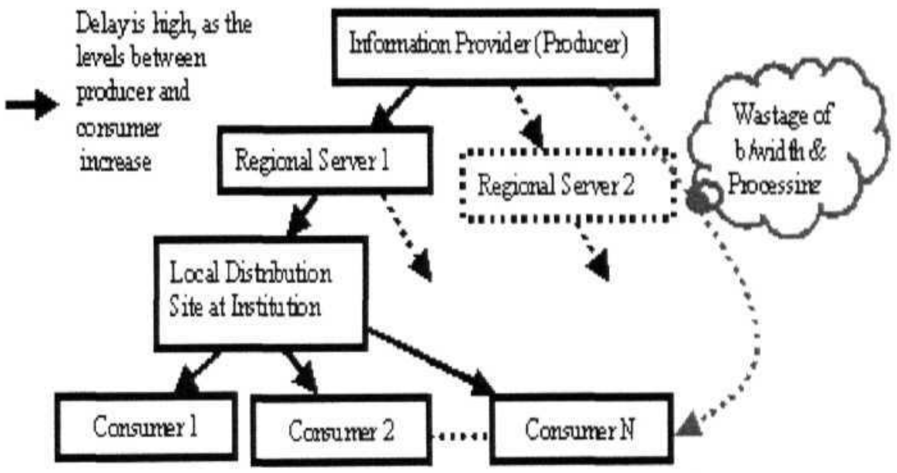

Delay is high, as the levels between producer and consumer increase
Information Provider (Producer)
Regional Server 1
Regional Server 2
Wastage of bandwidth & Processing
Local Distribution Site at Institution
Consumer 1
Consumer 2
Consumer N

**Figure 5. 1 Overlay**

Overlays can help meet the requirement of distributing information to a large user base on Internet, however, we have to devise ways to reduce the latency/delay introduced by overlays. Figure 5.1 shows as an example of overlay. Stream processing (explained below in detail) allows data to be processed as it streams in thereby not adding any delay at processor side. Using overlays, we can register profiles with the servers nearer to the consumers and these servers in turn send the combined profiles to the servers above them. This profile grouping and sending can further go up the tree till the producer. Now, when the producer sends info, this has to be filtered as per their profiles at each server at lower levels of the tree, till it reaches the consumer. Data is exchanged in the form of XML[80] in SDI systems and each profile is represented in the form of an Xpath[78] query and we filter the information based on multiple profiles (queries), registered at each server. In following sub sections we deal with the fundamental concepts, notations, functions etc of XML and its related technologies like XPath, DOM, SAX etc.

### 5.1.1 XML (Extensible Markup Language)

*XML* [80] is a simple and standard way to describe structured data. *Markup* is a way of conveying metadata (that is, information about a dataset). A collection of this markup that conforms to a defined syntax and grammar may be called a *language*. Markup languages use string literals or tags to delimit and describe data. XML is *extensible* because it involves a standard mechanism for defining new tags and their usage. The XML markup describes the contents of the document, both through explicit descriptions (the tag names and attributes) and through implicit structure (how the tags are nested within one another). XML was developed by an XML Working Group formed under the auspices of the World Wide Web Consortium (W3C) in 1996

**Nature of** XML

XML is a text-based format, making it a great solution for exchanging information across platforms. In addition, XML can be extended to meet the needs. The extension mechanism being standard, can be easily described to anyone-programmer or machine who reads the data. A well-formed XML document can be used by any XML-enabled system to provide the required information or transformations to the document.

Although XML was designed primarily for the Web, its usefulness extends beyond the confines of a browser's window or an HTML page. XML separates data from layout in a way that hides both the data source and the formatting from one another. It does not matter whether the XML came from a database or from someone typing it in Notepad; if the file is well-formed (that is, conforms to XML syntax rules), any XML parser can read it. This is not always the case with other types of data exchange. Often, a developer must deal with the back-end data structure in order to extract the data he or she wants. With XML, data can be locally consumed, created, or modified from a logical data structure that is independent of all back-end implementation. Multiple data sources may feed data into a single type of XML structure, allowing seamless integration of disparate systems. Because XML is plain text, this integration may take place through the Web via HTTP.

XML documents contain information about themselves—metadata. Well-designed tags and attributes can be read, understood and used by both people and computers. Document Type Definitions (DTDs) or XML Schemas [121] fomially declare the type of XML document. XML can be transformed using the Extensible Style Language (XSL) [119]. XSL is of two types of XSL: XSL for formatting and XSL for data transformation. XML documents may be converted into a formatted, human-readable document, or transformed into another data structure, including another XML document with a different DTD. XML documents heavily use the concept of 'name space' to uniquely refer to parts of XML documents. A namespace is a set of names in which all names are unique. The members of a namespace can be referred to without ambiguity through namespace-qualified names. Namespaces allow for shorter names and help in processing XML documents without any name collisions.

### 5.1.2 XML Parsers

Document Object Model (DOM) and SAX (Simple API for XML) are the most common parsers available in the industry.

**Document Object Model (DOM)**

The Document Object Model (DOM [91]) is a standard for programmatically accessing the structure and data contained in an XML document. The DOM is based on an in-memory tree representation of the XML document. When an XML file is loaded into the processor, it builds an in-memory tree that correctly represents the document. The DOM defines the programmatic interface (including the names of the methods and properties) that should be used to programmatically traverse an XML tree and manipulate the elements, values and attributes. When DOM is used to manipulate an XML text file, it parses the file, breaks it into individual elements, attributes and so on. It then creates a representation of the XML file as a node tree in memory. The contents of the document may then be accessed and manipulated through the node tree using the DOM interfaces.

**SAX**

The "Simple API" for XML (SAX)[104] is an event-driven, serial-access mechanism that does element-by-element processing. SAX works through callbacks: call the parser, it calls methods that are supplied. A SAX parser reads the XML document as a stream of

XML tags and genarates events such as starting elements, ending elements, text sections, etc corresponding to various components of an XML document.

**Difference between DOM and** SAX

- DOM reads the entire XML document into memory and stores it as a tree data structure where as SAX reads the XML document and sends an event for each element that it encounters.
- DOM provides "random access" into the XML document where as SAX provides only sequential access to the XML document.
- SAX is fast and requires very little memory, so it can be used for huge documents (or large numbers of documents).This makes SAX much more popular for web sites.
- Some DOM implementations have methods for changing the XML document in memory where as SAX implementations do not have such methods.
- SAX parsers generally require one to write a bit more code than the DOM interface.
- Limited API in SAX i.e., every element is processed through the same event handler. One needs to keep track of location in document and in many cases store temporary data.
- DOM is slow and requires huge amounts of memory, so it cannot be used for large XML documents.

### 5.1.3 XPath

XML does not specify how data is transmitted over the wire and it does not specify how data is stored. XML simply determines the format of the data. XPath [78], short for XML path language, has been specified by the W3C for addressing fragments of an XML document. XPath grew out of efforts to share a common syntax between XSL Transformations (XSLT) [119] and Xpointer [120]. It allows for the search and retrieval of information within an XML document structure. XPath is to XML as SQL is to relational database systems. XPath uses a compact, path-based, rather than XML element-based syntax. It operates on the abstract, logical structure of an XML document (tree of nodes) rather than its surface syntax. XPath uses a path notation (like URLs) to navigate through this hierarchical tree structure. This tree has 7 types of nodes.

*Root* - corresponds to the root of the document.

*Element* - corresponding to each element in the document.

*Attribute* - corresponding to the each attribute of an element.

*Namespace* - corresponding to the namespace that the sub-tree of an element belongs to.

*Processing Instruction* - corresponding to each processing instruction.

*Comment* - corresponding to each comment.

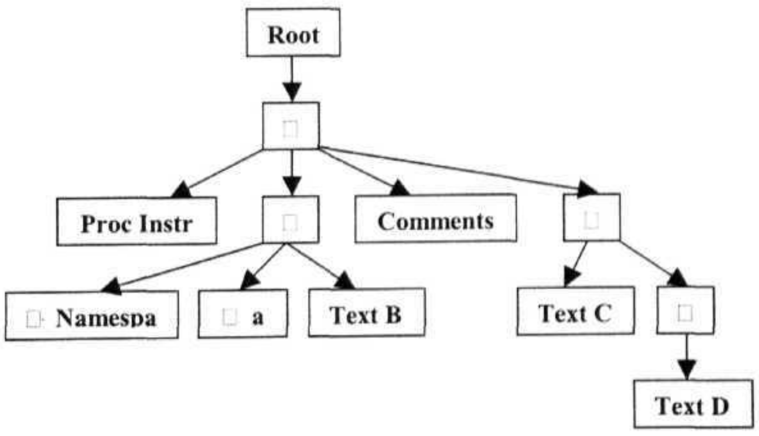
*Text* - Corresponding to parsed and unparsed character data within each element.

**Document Order**

Informally, document order is the order returned by an in-order, depth-first traversal of the hierarchical tree structure of an XML document. Within a tree, document order satisfies a set of constraints. Given below is a sample XML document

```
<A>
        <?xml-stylesheet type="text/css" href="stylesheet.css"?>
        <B xmlns:bk="http://www.books.com/nmsp" id="1">Text B</B>
        <!--...comment text...-->
        <C>Text C
            <D>Text D</D>
        </C>
</A>
```

Figure 5. 7: Tree (document-order) representation of the above XML document



**Expression vs. Location path**

An expression is the most general type of 'path' statement. Every XPath query is an *expression.* An expression can evaluate to yield an object of 4 types – 1.Node-set (unordered collection of nodes without duplicates), 2.Boolean (true/false), 3.Number (float) and 4.String (sequence of UCS chars). An expression that returns only a node-set is called a location path. Each location path in turn consists of a series of location steps. A location step looks as below.

*Location step = Axis:: node-test [optional predicate]*

Where *predicate = location path* and|or *location path*

*Location path = location step / location step....*

Location paths can be relative or absolute. Relative location paths consist of one or more location paths separated by backslashes. Absolute location paths consist of a backslash optionally followed by a relative location path. In other words, relative location paths navigate relative to the context node. Absolute paths specify the absolute position within the document. An absolute location path would then be:

/filesys/drive[@letter='C']/folder[@name='XML']

Using an absolute location path, the current context node is ignored when evaluating the XPath query, except for the fact that the path being searched exists in the same document.

**XPath Axis**

The axis component of an XPath query determines the direction of the node selection in relation to the context node. An axis can be thought of as a directional query. In XPath, there are *forward* axes that only select nodes that are after the context node in document-order and *reverse* axes that select nodes that are before the context node in document order. For every reverse axis in XPath there is a corresponding forward axis that is symmetrical to the reverse axis.

The following pairs of axes are defined in XPath:

- *Child* / *parent:* The *child* axis selects all direct children of the context node, i.e. all nodes directly contained in the context node. The *parent* axis selects the parent node of the context node.
- *Descendant* / *ancestor*: The *descendant* axis selects all nodes in the subtrees starting with the children of the context node, i.e. the children of the context node and all descendants of those children. The *ancestor* axis selects all the nodes that the context node is a descendant of, i.e. all nodes that are on the path from the context node to the root of the document tree.
- *Descendant-or-self* / *ancestor-or-self*: This pair of axes is equivalent to the *descendant* / *ancestor* pair, except that also the context node is selected.

- *Following-sibling / preceding-sibling:* The siblings of a context node are all nodes in the document tree that have the same parent node as the context node. The *following-sibling* axis selects all nodes that follow the context node in document order and that are siblings of the context node. The *preceding-sibling* axis selects all siblings that precede the context node in document order.
- *Following / preceding:* The following axis selects all nodes that follow the context node in document order starting with the first following sibling, i.e. the descendants of all following siblings of the nodes selected by the ancestor-or-self axis relative to the context node (including those siblings).
- *Self:* The *self*-axis is symmetrical to itself and selects only the context node.
- *Attribute* and *namespace:* These special axes are used for selecting the attribute nodes of the context node and the namespaces in scope at the context node.

*Following*, *following-sibling*, *preceding* and *preceding-sibling* are also called as horizontal axes because the elements corresponding to these axes are found al the same level as that of the elements corresponding to the context node.

**XPath Node-Test**

The XPath *node test* does just what its name implies, i.e., it tests nodes to determine if they meet a condition. By specifying the node name in the node test component of the XPath statement, we limit the results so that only a single node is returned. A node-test may be: a node name, "prefix:*" ,"text()" ,"node()" ,"processing-instruction()" or comment() etc.

**Predicates**

A predicate acts as a further filter on the node set selected by the axis and node test. This is optional in any location step. There is no restriction on one location step to contain more than one predicate.

**Abbreviated Syntax**

In order to simplify the representation of most frequently used syntactic fragments of XPath, short notations are introduced in Xpath, like 1) V represents child. 2) attribute:: can be abbreviated to @. 3) // is short for /descendant-or-self::node()/. 4) A location step of '.' is short for self::node(). 5) Location step of".." is short for parent: :node()

**Core Function Library**

XPath defines a core set of functions and operators that are classified into Node-set, String, Boolean and number functions depending on the type of data they operate on. Few examples are given below.

1. Count(/descendant::X) is a node-set functions that returns the no of nodes in the argument node set.

2. Number() is a number function that converts the argument to number.

3. Starts-with() is a string function that takes two string arguments and checks to see if first starts with the second argument.

4. not(boolean) is a bollean function that returns true if its argument is false, and false otherwise.

**Importance of** Reverse Axes [102]

Current evaluation methods such as the one followed in DOM are very inefficient when evaluating reverse axes such as *parent* and *ancestor*. The fundamental difficulty caused by the reverse axes is that they incur bottom-up traversal in the document tree while the tree is usually traversed in pre-order. In streaming environment, since seeking-back in the stream is usually not allowed or very expensive, this difference (between the semantics of the reverse axes and the restrict of pre-order traversal) seems to be conflicting. Though theoretically it is possible to express every XPath query with forward axes alone (not using backward axes), it would reduce the expressiveness of the XPath query language. More over, reverse axes are important for the users. Firstly, it empowers the user to specify more complex patterns. Without reverse axes, XPath queries can only specify tree patterns (where the edges are of forward axis), while with reverse axes the pattern could be a graph instead of tree.

The reverse axes are also very convenient for user to specify query that can fit data of various DTDs or schema. For example, in the scenario of information dissemination, it is very likely a query issued by the user will be applied to heterogeneous data sources. For example, the query "//book[subject="XML" or parent::pub="O'Reilly"]", which can be executed on both the following XML fragments, cannot be expressed in single query with only forward axes.

*<book>< subj* ect>XML< */subj* ect><sulejc t>....< */subj* ect></book>

*<pub>O'Reilly<book><subject>XML</subject><subject>.</subject>*
*</book></pub>*

**Approaches to handle reverse axes**

Following are three principal options how to evaluate reverse axes in a stream-based context: (proposed in [98]).

- Storing in memory sufficient information that allows to access past events when evaluating a reverse axis. This amounts to keeping in memory a (possibly pruned) DOM representation of the data.
- Evaluating an XPath expression in more than one run. With this approach, it is also necessary to store additional information to be used in successive runs. This information can be considerably smaller than what is needed in the first approach.
- Replacing XPath expressions by equivalent ones without reverse axes.

Our approach closely follows the third approach.

## 5.2 Existing XPath Engines

There are two approaches of processing XML documents namely, index-based and navigation-based. Index-based approaches such as index-filter build an index over the input XML document and probe the same for matching with queries. Most of existing XML processing engines follow the other approach, i.e., navigation-based approach. In this approach the queries are evaluated as the processor navigates through or parses the input document. Proposed YALXP follows navigation-based approach.

**Table** 5.1 Existing XPath engines and their scope

| Engine | Streaming XML | Single* | Multiple* | Backward |
|---|---|---|---|---|
| Xfilter | √ | - | √ | - |
| Yfilter | √ | - | √ | - |
| Xtrie | √ | - | √ | - |
| Mtrie | √ | - | √ | - |
| MQSPEX | √ | - | √ | √ |
| SPEX | √ | √ | - | √ |
| XSQ | √ | √ | - | √ |
| XAOS | √ | √ | - | √ |
| Xalan | - | √ | - | √ |

***Single** - Single-query processing system, **Multiple** - Multiple-query processing system. All systems invariably handle forward axes.

The table 5.1 gives a list (though not exhaustive by any means) of existing work done in the area of XPath processing. Xfilter[82], Yfilter[83], Xtrie[86] and Mtrie[87] do not focus on backward axes. SPEX[101], XSQ[97][102], MQSPEX[92] and XAOS[79] are streaming XML parsers that handle both forward and reverse axes. Apart from above systems, several algorithms have been proposed supporting different fragments of XPath. Substantial work has also been done to systematically assess time and space complexities for XPath evaluation. The paper "XPath Query Evaluation: Improving Time and Space Efficiency" [93] presents some improved time and space efficient algorithms on non-streaming XML documents. The paper "On the Memory Requirements of XPath Evaluation over XML Streams" [103] does the first formal analysis of the time and space complexities and presents the minimal space and time requirements of XPath evaluation on streaming XML documents. In this section, we give a brief introduction to different approaches to XPath processing. We give a detailed description of XAOS as our method is built upon the same.

### 5.2.1 Forward-only XPath Processors

In this section we describe XPath engines, which take care of only forward axes of XPath.

#### 5.2.1.1 Xfilter

Xfilter [82] is a benchmark for all the XML processing systems. Based on the XFilter system, several filtering engines for the selective dissemination of information (SDI) represented in XML have been proposed recently. These systems have focused on efficient filtering of (relatively small) XML messages or documents according to subscriptions expressed as XPath queries. The Xfilter system sets up the use of deterministic finite automata for filtering of XML data and proposes a novel query index optimizing state transitions of the DFAs: An incoming element label is used as key in a hash of all element labels occurring in any subscription. In a hash table the states (representing a step in the XPath expression) reachable from the current state by the associated hash key are noted. For each such state *s* in the hash table of the incoming element, all states corresponding to steps following the associated step of *s* in the subscription are added to the appropriate hash table. For the average case this leads to a very efficient selection of the state transitions in the DFAs. XFilter does not perform multi-query optimization. CQMC (Continuous Queries for Mobile Clients)[84] modifies Xfilter approach to exploit commonalities among queries.

#### 5.2.1.2 Yfilter

The optimization of multiple queries by sharing common prefixes is the principal contribution of Yfilter [83]. To enable prefix sharing, an NFA is used instead of multiple DFAs as in XFilter. Experimental evaluation shows a considerable lower processing time compared to XFilter on large number of subscriptions. Furthermore, two optimizations for the handling of predicates and nested path expressions (such as in $/a[b/c]/d$) are proposed. Selection postponement delays the evaluation of value-based predicates until a structural match is reached (thus avoiding the evaluation of predicates where no structural match is reached for the remaining expression). Both optimizations are based on the assumption that it is affordable to store possibly all nodes in a document for further processing, an assumption invalid for unbounded streams. In YFilter, the authors argue that their experimental evaluation shows that the cost for matching a subscription is no longer the dominant cost if compared to parsing and further processing and thus no further optimizations (e.g., to avoid the exponential complexity) is necessary.

#### 5.2.1.3 Xtrie

Xtrie[86] decomposes complex, tree-structured XPath expressions (XPE) into a set of simple and linear patterns (sub-strings) and indexes the sub-strings in a trie structure. This trie facilitates detection of sub-string matches in the input XML data. Each sub-string is a sequence of element names along some path in XPE tree, where each consecutive pair of nodes is related by a "/" operator (without any "*" or "//"). Information about each sub-string of each XPE is stored in a sub-string table (ST). The information in ST is used to check for partial matches. A matching algorithm using these two structures does query evaluation.

### 5.2.2 Backward-Axis XPath Processors

Two methods (other than XAOS[79]) have so far been proposed that handle backward axes on streaming XML. One is XSQ[97,102] and other is SPEX[101]. Both works have similarities in their approach. In this section we give a brief description of the two and the non-streaming backward-axis XPath processing engine Xalan.

#### 5.2.2.1 XSQ

XSQ[97,102] system uses a pushdown transducer (PDT) - an abstraction of the input query, to process the events that are generated by a SAX parser[104]. PDT is a pushdown automaton with actions defined along with the transition arcs of the automaton. In the start state, the PDT is initialized. At each step, based on the next input symbol and the symbols in the stack; it changes state and operates the stack according to the transition functions. The PDT also does an output operation, which could generate output during the transition enabling eager emission of output. PDT makes use of buffer to support these operations. XSQ was later extended for reverse-axes. The extended XSQ addresses a large segment of XPath and employed a totally new approach.

#### 5.2.2.2 SPEX

SPEX-Streaming and Progressive Evaluation of Xpath [101] employs almost the same mechanisms as followed by XSQ (forward-only), but addresses broader fragment of XPath such as nested predicates etc. SPEX also rewrites a query with reverse axes to reverse-axis free one.

*Q1 (Query):* /desc::b[desc::d or child::e]/parent::a/fsibl::c

*Q2 (Rewritten Query by SPEX):* /desc::a[child::b[desc::d or child::e]]/fsibl::c

A query graph is created for each query, where syntactical details of XPath are abstracted out. The query graph (or query plan) is implemented by a network of deterministic pushdown transducers. The collective incremental work of transducers yields the answers to the original query.

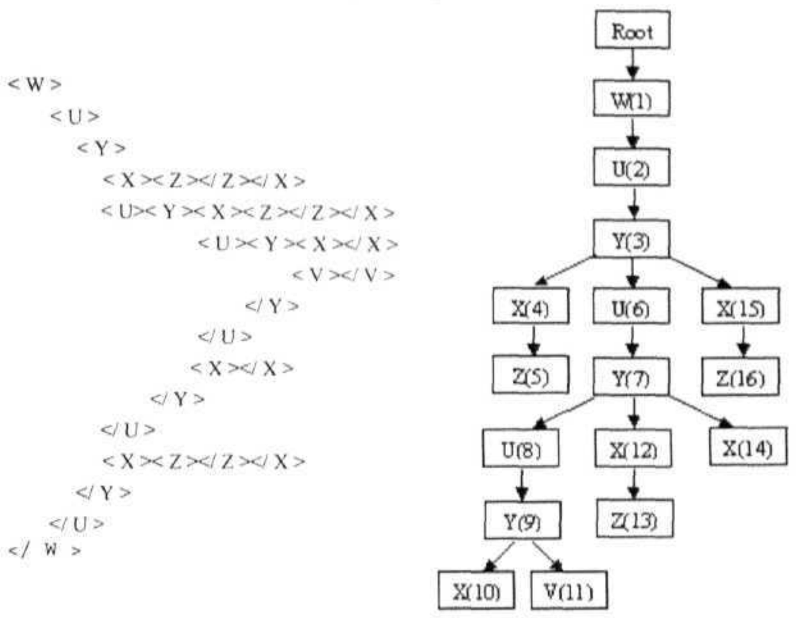
#### 5.2.2.3 MQSPEX

This method is an extension of the previous work, SPEX - Streaming and Progressive Evaluation of Xpath, for multiple queries. MQSPEX[92] finds the optimal cost query plan that allows the simultaneous evaluation of multiple queries against the same stream. MQSPEX optimizes processing for multiple queries by taking advantage of the arbitrary commonality among different queries along with common prefixes by building optimized query plans for multiple queries. Their experiments show that sharing arbitrary operators under a realistic cost function results in query plans that have consistently lower cost for reasonable sets of queries than query plans where only common prefixes are considered. MQSPEX also claims to have addressed a larger and complex fragment of XPath than many other engines.

### 5.2.2.4 Xalan

Xalan[81] is a non-streaming XPath engine where the input document is represented in the form of a tree, called DOM (Document Object Model). After the tree is built the query is evaluated by traversing the tree.

Figure 5. 8: XML document and Tree corresponding to the same



```
<W>
   <U>
      <Y>
         <X><Z></Z></X>
         <U><Y><X><Z></Z></X>
                  <U><Y><X></X>
                           <V></V>
                     </Y>
                  </U>
                  <X></X>
               </Y>
         </U>
         <X><Z></Z></X>
      </Y>
   </U>
</ W >
```

To explain the evaluation of a query in Xalan, let us consider the query

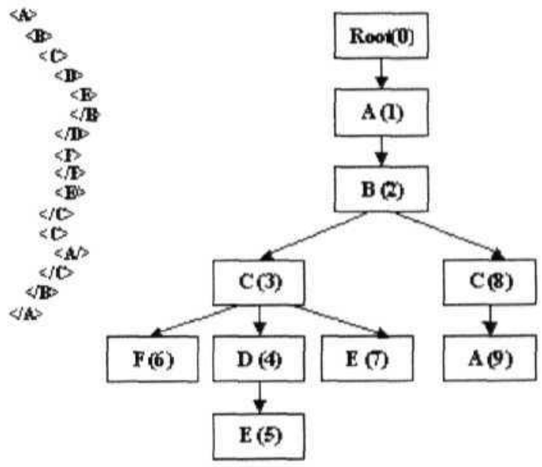
"Descendant::U/descendant::Y | descendant::V and ancestor::W/descendant::X]/descendant::Z"

on the document in first column of figure 5.3. The tree corresponding to the document looks as in 2nd column of figure 5.3. The query evaluation takes place in 3 traversals. In first traversal, elements 2,6 and 8 are selected. In second traversal, 3,7 and 9 are selected. In the next traversal, 5, 13 and 16 are selected, which are the output of the query. The performance of Xalan is worse when the query has more *descendant* and *ancestor* axes. This is because, for *descendant* of an element Y all the elements within the sub-tree of Y have to be searched and for the *ancestor* of Y all the *ancestors* of Y up to Root have to be searched.

### 5.2.2.5 XAOS

XAOS[79], pronounced as Chaos, is an algorithm to evaluate XPath expressions having both forward and backward axes on streaming XML data. Though it was originally designed for only two backward axes namely *parent* and *ancestor*, it is extensible to other backward axes. XAOS caters to only location paths of the XPath specification. This particular subset of XPath is termed as Rxp (Restricted XPath). XAOS works on two views of the input query namely x-tree and x-dag. X-tree is a rooted tree that has x-nodes corresponding to each node test in the query. Each x-node, but for the 'Root', has a unique incoming edge that is labeled with the relationship (such as *descendant/parent* etc.) it shares with the source x-node. For example, figure 5.5 depicts the x-trec for a query. There exists one output x-node in this x-tree. (XAOS is extensible for multiple output nodes). X-node labeled E(5) is the output node in figure 5.5.

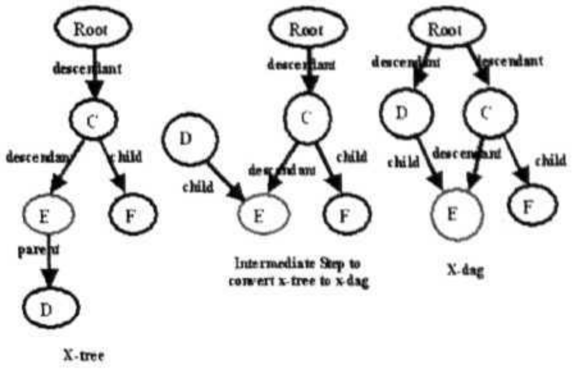
Figure 5. 9: (a) An XML Document (b) Tree representation of the same
The number in parentheses next to the tag of each element is the id of the element. Query in figure 5.5 is based on this document



```
<A>
 <B>
  <C>
   <D>
    <E>
    </E
   </D>
   <F>
   </F>
   <E>
  </C>
  <C>
   <A/>
  </C>
 </B>
</A>
```

The backward axes such as ancestor and parent in x-tree are converted to forward axes and the resulting view or data structure is called x-dag (x- directed, acyclic, graph). This conversion is done in three steps. First step reverses the direction of backward edges. The labels of theses edges are renamed with corresponding forward axes. Ancestor relation is renamed as descendant and parent relation is renamed as child. For example, in figure 5.5, the parent edge from 'E' to 'D' is changed to a child edge from 'D' to 'E'. The last step consists of adding an incoming edge with label descendant from 'Root' to each x-node (other than 'Root') that is left with no incoming edge. In figure 5.5, a descendant edge is added from 'Root' to 'D'.

**Figure 5. 10: Construction of x-tree & x-dag for the query "/descendant::C[child::F]/descendant::E[parent::D]"**



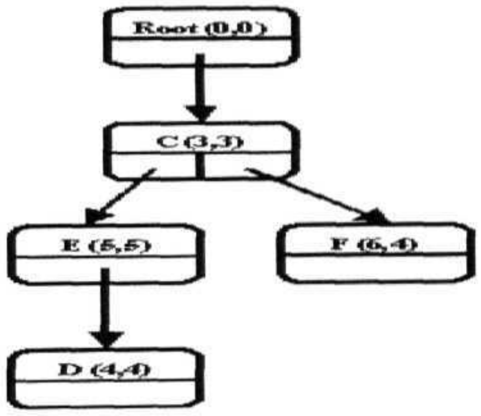
After the two views x-tree and x-dag are ready, execution of XAOS proceeds with the filtering of events. Matching structures are created for matched elements and finally the output is emitted at the end of the document. Filtering of events consists of construction of a looking-for-set at the end of each event making use of the x-dag view of the query. Looking-for-set consists of the set of relevant elements to be searched for in the next event. An element is open if we saw the start tag of the element and we did not yet see the end tag of the same. An element is relevant if this element matches with one of the elements we are looking for. For an element to be looked for in next event, it should have an open and relevant element for each of the remaining parents in x-dag.

**Table 5.2:** Walk through of evaluation of XPath on XML document of Figure 5.4. In the first column, Start/End:A(x, y) denotes the start/end element event for an element A(x,y)in XML document with x as node id and y as level. The **Looking-for set** column shows L (set of **(element, level))** at the end of processing the event.

| Event | Matches | Looking-for-set | Event | Matches | Looking-for-set |
|---|---|---|---|---|---|
| Start: Root(0,0) | Root(0,0) | {(C,*),(D,*)} | End: E(7,4) | - | {(C,*),(D,*),(F,4)} |
| Start: A(1,1) | - | {(C,*),(D,*)} | End: C(3,3) | (C,*) | {(C,*),(D,*)} |
| Start: B(2,2) | - | {(C,*),(D,*)} | Start: C(8,3) | (C,*) | {(C,*),(D,*) ,(F,4)} |
| Start: C(3,3) | (C,*) | {(C,*),(D,*),(F,4))} | Start: A(9,4) | - | {(C,*),(D,*)} |
| Start: D(4,4) | (D,*) | {(C,*),(D,*),(E,5)} | End: A(9,4) | - | {(C,*),(D,*)} |
| Start: E(5,5) | (E,5) | {(C,*),(D,*)} | End: C(8,3) | (C,*) | {(C,*),(D,*)} |
| End: E(5,5) | - | {(C,*),(D,*),(E,5)} | End: B(2,2) | - | {(C,*),(D,*)} |
| End: D(4,4) | (D,*) | {(C,*),(D,*),(F,4)} | End: A(1,1) | - | {(C,*),(D,*)} |
| Start: F(6,4) | (F,4) | {(C,*),(D,*)} | End: Root(0,0) | Root(0,0) | - |
| End: F(6,4) | - | {(C,*),(D,*),(F,4))} | | | |
| Start: E(7,4) | - | {(C,*),(D,*)} | | | |

Each item in the looking-for-set has the name of the element to be looked for and the level at which it is to be found. If the outgoing edge is "descendant", then the element we are looking for can be found at all levels above the level of current element before we see the end tag of current element. In above example, the start document event matches with 'Root' of x-dag. 'Root' of x-dag has two outgoing edges namely descendant to x-nodes 'C' and 'D'. So looking for set at the end of this event includes the items (C, *) and (D, *). Similarly if the outgoing edge is "child", then the element we are looking for can be found at a level equal to (1 + the level of the current element). In above example, the start event of C(3,3) matches with 'C' of x-dag. Since 'C' has an outgoing edge namely child to x-node F, looking for set at the end of this event includes the item (F, 4). 'C' also has an outgoing edge namely descendant to x-node E. But we did not see an open and relevant element that matches with 'D', which is a parent of 'E'. So looking for set does not include the item (E, *). By doing this, most of the irrelevant elements are discarded.

Figure 5. 11: Matching Structure after evaluation of query in figure 5.5 on document in figure 5.4



Matching structures are built using x-tree representation of the query. Each matching structure is represented by $M_{e,i}$ , where 'e' is the element name and 'i' is the id of the element. (XAOS uses another representation interchangeably with this, that is $M_{v,e}$ - where V is x-node and 'e' is element in XML document that matches. If a start event of an element is found to be relevant (i.e. matches with an x-node in the x-tree) a matching structure is created for the match. Each matching structure has the node test (name) of the element and pointers to sub-matches corresponding to each child x-node in the x-tree. At the end element event of this element, a key step called propagation is done to see if there is total matching at the matching for this element according to three conditions. If the corresponding x-node is a leaf in the x-dag, there is total match for the corresponding matching structure. In above figure 5.6, $M_{F,6}$ corresponds to a leaf in x-dag. So, at the end element event for F(6,4), this matching is propagated to its parent matching $M_{C,3}$. If the corresponding x-node is not a leaf in the x-dag and all its sub-matches are non-empty and total, there is total match at the corresponding matching structure. In figure 5.6, $M_{C,3}$ corresponds to a non-leaf in x-dag. And there is total match for its sub-matches $M_{E,5}$ and $M_{D,4}$. SO, at the end element event for C(3,3), $M_{C,3}$ is propagated to its parent matching $M_{Root,0}$. At the end of document event, the same check is done at the 'Root' matching structure.

There is a catch in this process. If the x-tree does not have ancestor and parent labels, this process is straightforward as above. Otherwise, i.e. when there is/are ancestor and parent outgoing edges from an x-node X to x-node X' for which matching occurred at an event, the propagation is optimistic at the corresponding end of element event. This special case is dealt in two steps.

1. At the start event of an element Y that matches X, if the sub-matching $M_{Y',i}$ (at an element Y' with id 'i') corresponding to the x-node X' is empty, it is optimistically propagated that $M_{Y',i}$ is total. Similarly if other sub-matches of Y are non-empty and total it means there is a total match at Y. This total match is propagated to appropriate parent-matches.
2. At the end event of Y', if it can now be determined conclusively that $M_{Y',i}$ is not total, the optimistic propagation should be recursively undone from its parent and other superior matches.

In above example, the x-node 'E' has an outgoing edge labeled 'parent'. So by the time the end of element event for E(5,5) occurs, we have not seen the end tag for D(4,4). So, at this point, logically there is no total matching at D(4,4). According to XAOS, it is assumed that $M_{D,4}$ has total matching and the same is propagated upwards to $M_{E,5}$. However, since there is total matching at $M_{D,4}$ (i.e. we saw end tag of D(4,4)), we do not need undo this optimistic propagation at end of element event of D(4,4) from $M_{E,5}$ and its superior matches.

At the end of document event, $M_{Root,0}$ has total matching since its sub-matching $M_{C,3}$ is non-empty and total. If there are one or more, then these matches should be suitably emitted by traversing the whole matching structure. The solution for the example is $\{E(5,5)\}$. And total matching at Root is $\{Root \rightarrow 0, C \rightarrow 3, F \rightarrow 6, D \rightarrow 4, E \rightarrow 5\}$.

## 5.3 YALXP - Yet Another Light Weight XPath Processor

We continue to call each XPath Expression as an *Rxp* (Restricted Xpath), since we cater to essentially that fragment of XPath which XAOS addresses. YALXP operates on a combined view of all the input XPath expressions called combined x-dag or c-dag. Combined matching structure that represents the matching elements for c-nodes in the c-dag is built as the execution of the algorithm progresses.

YALXP can be viewed as an application built on top of an event-based XML parser such as SAX. It consumes the events sent by SAX parser and deals with them appropriately. At the end of every event we maintain a *looking-for-set* that has the set of elements expected in the next event, along with 'qids' (query identifier) for which they are likely to match. For an element to be relevant it has to match with at least one of these items. This way, we can filter out irrelevant events. Relevant elements along with the links among them are stored in combined matching structure. YALXP uses combined x-dag to build combined matching structure. At the end of the algorithm, we emit the output matches for different queries with the help of individual x-tree for each query. We stick to the lazy emission of output (at the end document event), a concept originally followed by XAOS. This chapter describes the construction of combined x-dag, the structure of combined matching structure and complete process of query evaluation.

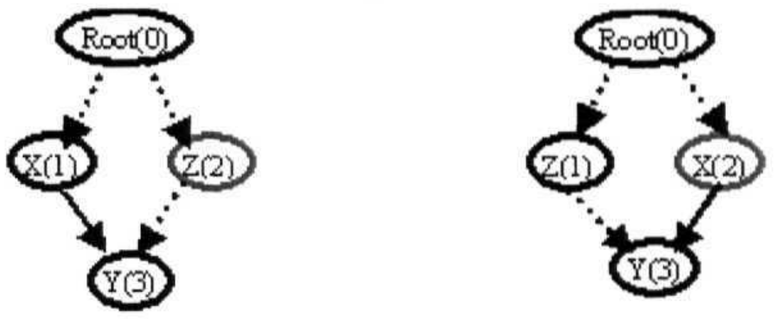
### 5.3.1 Combined X-Dag

YALXP operates on a combined representation of the input *Rxp* set that is called combined x-dag or c-dag. Combined x-dag is a directed and acyclic graph that is obtained by combining individual x-dags constructed for each *Rxp* in the input *Rxp* set. Combined x-dag is built on the premise that there exists a significant commonality among different *Rxps* in large-scale systems.

Commonality not only exists in the string form of the XPath queries. As we explain below commonality may exist in two *Rxp*'s, which need not have any common sub-string. In figure 5.7, the two *Rxp*'s do not have any commonality in the string form of the *Rxp's*. But after the x-dags are drawn, they look like mirror images of each other. This is the kind of commonality we are going to exploit by building combined x-dag for multiple *Rxp's*.

Figure 5. 12: Commonality in two queries looking different

Q1 :/descendant::Z [descendant ;Y / parent: X] Q2 :/descendant: :X [child::Y / ancestor::Z]



Each node in c-dag is called c-node. Each c-node represents a set of x-nodcs of different queries. We define *info*, an ordered pair *(qid, x-nodeid)*, which stores the information about each x-node that a c-node represents. For example, if a c-node has the *info {qid, x-nodeid)* with it, it means that this c-node corresponds to an x-node with id '*x-nodeid*' in the x-dag with id '*qid*'. Each edge in c-dag represents edges of different x-dags. At each edge in c-dag, we store the qids of x-dags in which the edge participates. Each edge also has a label associated with it, which is either *Descendent* or *Child*. The c-dag is constructed in such a manner that it is possible to reconstruct individual x-dags from the c-dag.

The construction of c-dag starts with the construction of individual x-dag for each *Rxp*. This involves construction of individual x-tree for each *Rxp* as explained in XAOS. X-dags for all *Rxp's* are built from these x-trees by translating all the backward constraints ('*ancestor*' and '*parent*') to forward constraints ('*descendant*' – represented by dotted edge and '*child*'—represented by thick edge in figure 5.7) according to the rules specified by XAOS. Nodes in red in figure 5.7 are the output nodes.

After this step, the c-dag is initialized to the first x-dag (corresponding to the first *Rxp)*. Now the c-dag has a set of c-nodes and edges that correspond to x-nodes and edges of the first x-dag. At each c-node thus created, *info* (1,*x-nodeid*),where x-nodeid is x-node that the c-node represents, is registered. qid '1'is registered at each edge in the c-dag. To the 'Root' of c-dag, *info* (*qid*,0) about Root x-node of each x-dag(*qid*) is added, since Root of c-dag matches with Root's of all the x-dags.

After this initialization process, the rest of the x-dags are added to the c-dag by traversing each x-dag in depth-first fashion. The x-node '$x_j$' whose outgoing edges and outgoing x-nodes (the x-node to which an outgoing edge leads) are going to be added to the c-dag. There are six scenarios listed as cases below in bold in which an outgoing edge and outgoing x-node can be added to c-dag. The steps with small Roman letters under each case in the list are the operations to be performed in that case. We followed the following conventions in the list. '$x_k$' is the outgoing x-node being added. *c(qid,j)* is the c-node corresponding to x-node with id '*j*' in x-dag for *qid.* We say that a c-node exists for '$x_k$' if this c-node has info matching with *qid* and *k* (id of $x_k$).

**1. If a c-node exists for $x_k$ i.e., an outgoing c-node of *c(qid,j)* and an edge with label *axis($x_j$,$x_k$)* exists between *c(qid,j)*and *c(qid,k)***

i. Register *qid* at edge between *c(qid,j)* and *c(qid,k)*;

**2. If a c-node exists for $x_k$ i.e., an outgoing c-node of *c(qid,j)* and no edge with label *axis(Xj,Xk)* exists between *c(qid,j)*and *c(qid,k)***

i. Let *prevnode* = *c(qi d, k)*;

ii. Remove *info (qid,k)* from *prevnode;*

iii. Create a new c-node for $x_k$;

iv. Add *info* to this c-node, which is *c(qid,k)* from now on;

v. Add edge with label *axis($x_j$,$x_k$)* between *c(qid,j)* and *c(qid,k)* and register *qid* at this edge;

vi. Add incoming edges of *prevnode* w.r.t. *x-dag(qid)* to *c(qid,k)* and register *qid* at the same;

vii. Remove qid from incoming edges *of prevnode;*

viii. Add outgoing edges *of prevnode* w.r.t. *qid*, to *c(qid,k)* and register *qid* at the same;

**ix.** Remove *qid* from outgoing edges of *prevnode;*

**3. If a c-node exists for $x_k$ that is not an outgoing c-node of $c(qid,j)$**

i. Add an edge with label $axis(x_j,x_k)$ between *c(qidj)* & $c(qid,k)$;

**ii.** Register *qid* at this edge;

**4. If no c-node exists for $x_k$, an outgoing c-node of $c(qid,j)$ has name equal to name of $x_k$ and the edge between $c(qid,j)$ and this outgoing c-node has label $axis(x_j,x_k)$**

i. Add *info* to this c-node;

ii. Register *qid* to edge between $c(qid,j)$ and $c(qid,k)$;

**5. If no c-node exists for $x_k$, an outgoing c-node of $c(qid,j)$ has name equal to name of $x_k$ and the edge between $c(qid,j)$ and this outgoing c-node has label not equal to $axis(x_j,x_k)$**

i. Add *info* to this c-node;

ii. Add an edge with label $axis(x_j,x_k)$ between $c(qid,j)$ & $c(qid,k)$;

iii. Register *qid* to edge between $c(qid,j)$ and $c(qid,k)$;

**6. If no c-node exists for $x_k$ and no outgoing c-node of $c(qid,j)$ has name equal to name of $x_k$**

i. Create new c-node with name equal to name of $x_k$;

ii. Add *info* to newly created c-node;

iii. Add an edge with label *axis(xj,Xk)* between $c(qid,j)$ and $c(qid,k)$;

iv. Register *qid* to edge between $c(qid,j)$ and $c(qid,k)$;

In cases 1, 2 and 3 where $x_k$ was not visited in a previous traversal of x-dag, edges and x-nodes out of $x_k$ are added in similar manner as that of $x_j$. In cases 4, 5 and 6, there is no need to add these edges and x-nodes as they would have been added to c-dag when $x_k$ was visited previously. It is understood that while un-registering *qid* from edges, if the edge is not left with any *qid* after removal, the edge itself is removed.

Similarly the remaining x-dags are also added to the c-dag. The resulting c-dag after this step has all the commonalities among different *Rxps* exploited to the optimal extent (to the extent that there is no loss of information).

## 5.3.2 Sample Construction of c-dag

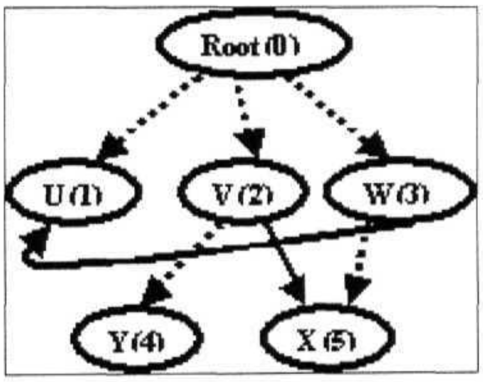
We explain the construction of c-dag for two queries in given in figure 5.8 below. In figure 5.8, the conventions followed are as follows. A c-node or x-node is represented as N(i) where 'N' is the name of the node and 'i' is the id of the node. The notation e(N(i),M(j),axis) represents edge between N(i) and M(j) with label 'axis'. For example, e(Root(0),W(2)),*descendant*) represents edge between Root(0) and W(2) with label '*descendant*'. The set such as {1,2} near an edge represents the qids in which this edge participates. The pairs such as (1,2), (2,1) near each c-node represent the x-nodes to which this c-node corresponds, where first part is qid and second part is x-node id.

**Figure 5. 13: Construction of c-dag for queries**

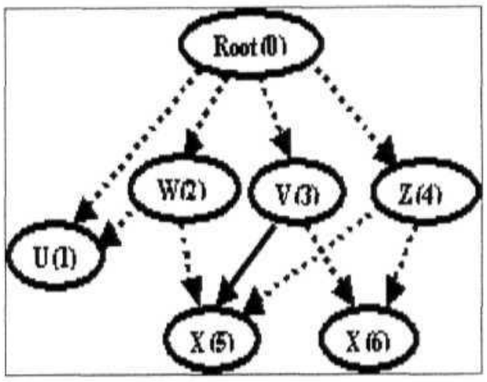
Q1:/descendant: :U | parent: :W/descendant::X| parent:: V/descendant"Y]| **and**

Q2: /descendant::U[ancestor::W/descendant::X[parent::V/descendant "• X and ancestor ::Z/descendant::X]]

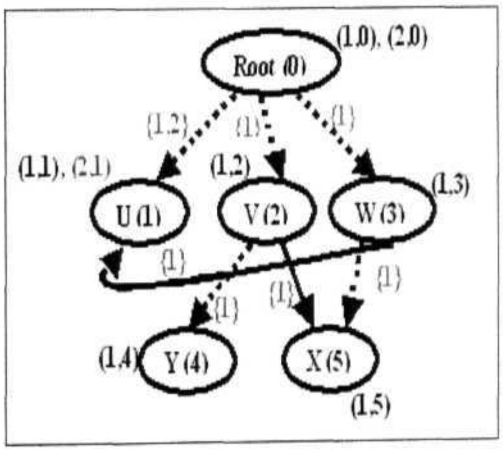
x-dag1 for Q1:



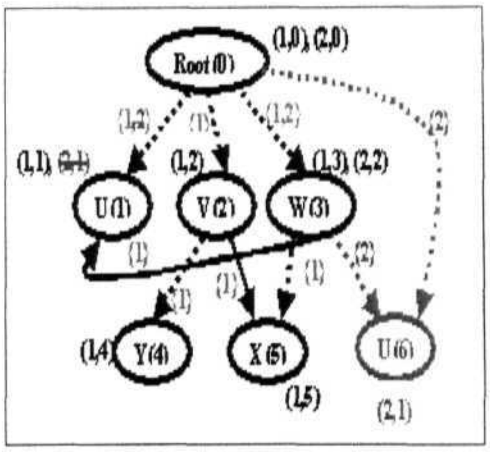
x~dag2 for Q2 :



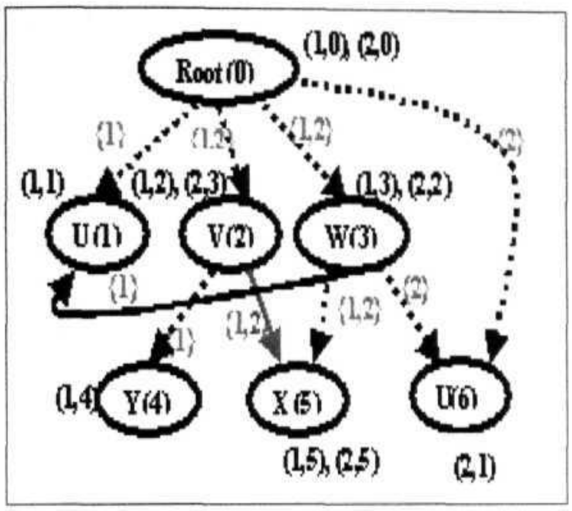
**Step1**- Case 4
Adding (U(l),e(Root(0),U(l)), *descendant)*



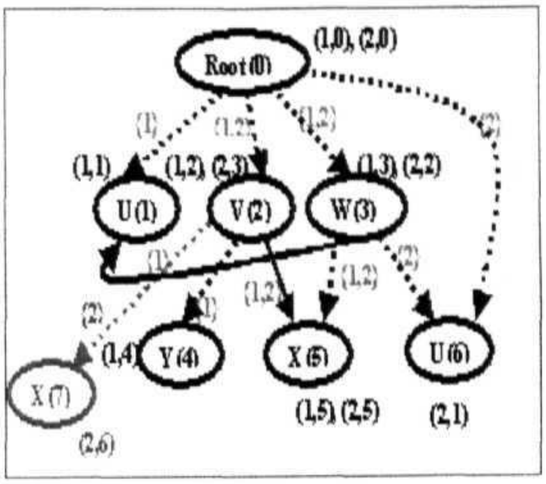
**Step3** - Case **2**
Adding (U(l ),e(W(2),U(1)), *descendant*))



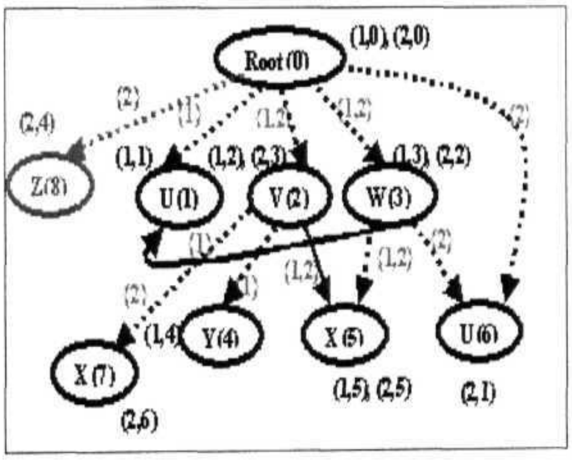
**Step6 - Case 1**
**Adding (X(5),e(V(3),X(5), child))**



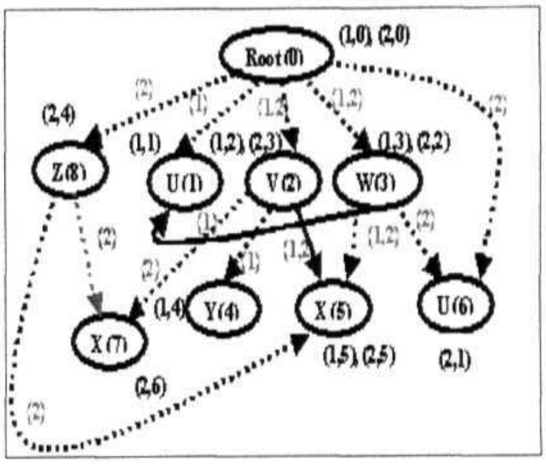
**Step7 - Case 5**
**Adding** (X(6),e(V(3),X(6), **descendant))**



**Step8 - Case 6**
**Adding** (Z(4),e(Root(0),Z(4), *descendant*))



**Step 10 - Case 3**
**Adding** (X(6),e(Z(4),X(6), ***descendant))***



**Step1:**

In this step, U(l) and e(Root(0),U(1),*descendant*) of x-dag 2 are added to Root(0) of c-dag. There is no C(2,l), i.e. c-node with *info* (2,1) in c-dag. However, Root(O) of c-dag has an outgoing c-node with name ‘U’, i.e. U(l). And Root(0) and U(l) of c-dag have a *descendant* edge between them. So this step falls into the Case 4. Hence *info*, i.e. (2,1) is added to U(l) in c-dag and qid '2' is registered at e(Root(0),U(1),*descendant*) of c-dag.

**Step2:**

In this step, W(2) and e(Root(0),W(2)),*descendant*) of x-dag 2 are added to Root(0) of c-dag. *info,* i.e. (2,2) is added to W(3) in c-dag and qid '2' is registered at e(Root(0),W(3),*descendant*)of c-dag, according to Case **4.**

**Step3:**

In this step, U(1) and e(W(2)),U(1),*descendant*) of x-dag 2 are added to W(3) of c-dag. There is C(2,l), i.e. c-node with *info* (2,1) in c-dag, i.e. U(1). And W(3) and U(1) of c-dag have a *child* edge (added while adding x-dag 1) between them. So this step falls into the Case 2. So a new c-node with name 'U' and id '6' is created and an edge with label '*descendant*' is added from W(3) to U(6) and qid '2' is registered at this edge. The incoming edge to U(l) w.r.t. x-dag 2, i.e. '*descendant*' from Root(0) is added to U(6) and qid '2' is registered at this edge.

**Step4:**

In this step, X(5) and e(W(2),X(5),*descendant*) of x-dag 2 are added to W(3) of c-dag *info,* i.e. (2,5) is added to X(5) in c-dag and qid '2' is registered at e(W(3),X(5),*descendant*) of c-dag, according to Case 4.

**Step5:**

In this step, V(3) and e(Root(0),V(3),*descendant*)of x-dag 2 are added to Root(0) of c-dag. *info,* i.e. (2,3) is added to V(2) in c-dag and qid '2' is registered at e(Root(0),V(2),*descendant*)of c-dag, according to Case 4.

**Step6:**

In this step, X(5) and e(V(3),X(5), *child)* of x-dag 2 are added to V(2) of c-dag. There is C(2,5), i.e. c-node with *info* (2,5) in c-dag, i.e. X(5); and V(2) and X(5) in c-dag have a *child* edge between them. So this step falls into the Case 1. Hence, qid '2' is registered at e(V(2),X(5),*child*) of c-dag.

**Step7:**

In this step, X(6) and e(V(3),X(6),*descendant*) of x-dag-2 are added to V(2) of c-dag. There is no C(2,6), i.e. c-node with *info* (2,6) in c-dag. However, V(2) of c-dag has an outgoing c-node with name 'X', i.e. X(5). But, V(2) and X(5) of c-dag have a *child* edge between them. So this step falls into the Case 5. So a new c-node with name 'X' and id

'7' is created, an edge with label '*descendant*' is added from V(2) **to** X(7) and qid '2' is registered at this edge.

**Step8:**

In this step, Z(4) and e(Root(0),Z(4),*descendant*) of x-dag-2 are added to Root(0) of c-dag. There is no C(2,4), i.e. c-node with *info* (2,4) in c-dag and Root(0) of c-dag has no outgoing c-node with name 'Z'. So this step falls into the Case 6. So a new c-node with name 'Z' and id '8' is created, an edge with label '*descendant*' is added from Root(O) to Z(8) and qid '2' is registered at this edge.

**Step9:**

In this step, X(5) and e(Z(4),X(5),*descendant*) of x-dag-2 are added to Z(8) of c-dag. An edge with label '*descendant*' is added from Z(8) to X(5) and qid '2' is registered at this edge, according to Case 3 (explained below).

**SteplO:**

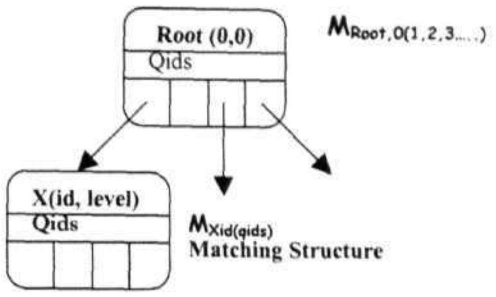
In this step, X(6) and e(Z(4),X(6),*descendant*) of x-dag 2 arc added to Z(8) of c-dag. There is C(2,6), i.e. c-node with *info* (2,6) in c-dag, i.e. X(7); and Z(8) of c-dag has no outgoing edge to X(7). So this step falls into the Case 3. Hence, an edge with label '*descendant*' is added from Z(8) to X(7) and qid '2' is registered at this edge.

### 5.3.3 Combined Matching Structure

We extend the concept of matching structure of XAOS to combined matching structure in case of multiple *Rxp's*. A combined matching structure represents the matches for all the input *Rxps* (i.e. for different c-nodes) at the *Root* of the c-dag. We store the information of an element that matches a particular c-node C in a *matching structure*[1] represented as $M_{x,i(qid's)}$ where '*x*' is name and '/" is id of the element. *qid's* are the ids of the input *Rxps* for which *x* matches. This matching structure has pointers to matching structures of elements that match with outgoing c-nodes of C.

[1] We use the term combined matching structure to denote the whole structure and matching structure to denote individual matching structure within combined matching structure.

**Figure 5. 14: Combined Matching structure**



$M_{Root,0(qid's)}$ represents the *Root* matching structure, where *qid's* are the ids of the input *Rxp*s for which there is a matching at the *Root* of the c-dag. A matching structure $M_{x,i(qid's)}$ is said to be a parent-matching of a matching structure $M_{x',i'(qid's)}$ if *x* corresponds to a c-node that is a parent of the c-node for which '*x*' has matching. $M_{x',i'(qid's)}$ is said to be the child-matching or sub-matching of $M_{x,i(qid's)}$.

Diagrammatically, there are 3 rows in a matching structure. First row has the element name along with its id and level. Second row has the qids for which the matching takes place. Third row has slots, which contain pointers to sub-matchings for the children of respective c-node in the c-dag. There is no need to store the edge information (qids at these pointers) in combined matching structure (since these pointers match for the same set of queries, id's of which are present at the destination sub-matching). As the query evaluation progresses, combined matching structure is built based on c-dag.

### 5.3.4 Query Evaluation Algorithm

The query evaluation process occurs at four events namely 'start document', 'start element', 'end element' and 'end document'. This process can be spilt into two components. One is the matching process, which occurs at all events, that filters out irrelevant events simultaneously storing relevant elements in combined matching structure. The next component is emission of the output, which occurs at 'end document' event using individual x-tree of each *Rxp.*

#### 5.3.4.1 Matching Algorithm

Following points are to be considered in the matching process.

1. An element is open if we saw the start tag of the element and we did not see the end tag of the same.
2. An element *e* is relevant only if this element matches with one of the elements we are looking for.
3. It is decided that there is a total matching at a matching structure for a particular *Rxp,* if there is at least one sub-matching, that has total matching, corresponding to each of the child c-nodes with respect to that *Rxp* in the combined x-dag.

**Filtering Events**

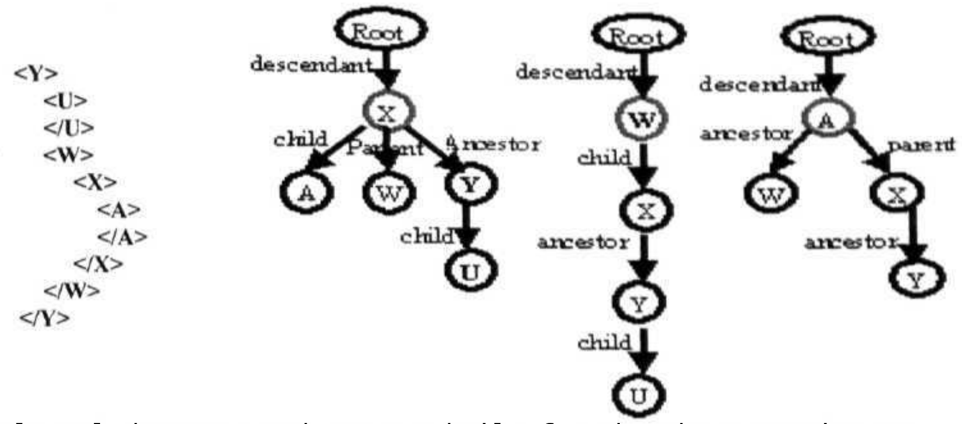
Filtering of irrelevant events is enabled by the construction of *looking-for-set.* At every event, based on c-dag, we construct *looking-for-set* that consists of elements expected in next event. Each item in the *looking-for-set*, called *lfitem*, corresponds to a c-node. An *lfitem* has the name of the element to be looked for, the level at which it is to be found and the *Rxp* ids (mentioned in table 5.3 with superscripts) with which it matches. For an element to be looked for, with respect to an *Rxp,* in next event, it should have an open and relevant element for each of the remaining parents[2] in c-dag with respect to that *Rxp*. If an element matches with a c-node in the c-dag for a particular set *of qids,* only the outgoing c-nodes in c-dag are considered to be looked for in the next event. While adding an *lfitem* to *looking-for-set,* the set of qids to be considered for this item is obtained by the intersection between the set of qids present at the outgoing edge and the qids for which current element matched. If the outgoing edge of the c-node (for which current element matched) is *"child"* for a set of qid's, the outgoing c-node will have a matching element at a *level* = \+*level* of the current element. Otherwise, if the outgoing edge is "*descendant*" for a set of qid's, an element that matches the outgoing c-node is at all levels above the *level* of current element before we see the end tag of current element.

In figure 5.11, we have shown c-dag having only the ids of *Rxp*'s of the x-nodes that a c-node represents without corresponding x-node ids, for easier visualization. This c-dag was constructed in the same manner explained in earlier section. In the c-dag, $W^{1,2,3}$ is the parent of $X^{1,2,3}$ with respect to *Rxp*'s land 2 but not with respect to *Rxp* 3 since the

[2] A c-node is said to be a parent of a c-node with respect to an *Rxp* only if there exists an edge between these two c-nodes that contains the id of the *Rxp* in question.

*child* edge between these two c-nodes has only 1 and 2 registered at it. All these cases are clearly described at one place in Step 2 of the table 5.3.

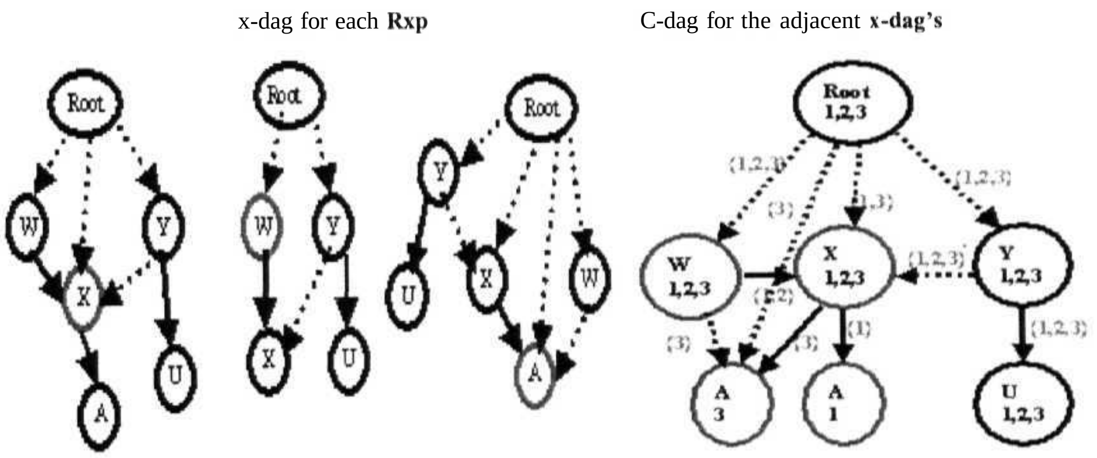
Figure 5.15: Sample xml document and x-trees built



```
<Y>
  <U>
  </U>
  <W>
    <X>
      <A>
      </A>
    </X>
  </W>
</Y>
```

Sample xml document and x-trees built for the three queries **Q1:** `descendant::X [/ancestor::Y/child::U` and `/parent::W` and `/child::A]` **Q2:** `descendant::W` [/child::X `/ancestor::Y/child::U]` **Q3:** `descendant::A [/ancestor::W` and `/parent::X/ancestor::Y/child::U]`

Figure **5.16:** x-dags for x-trees built in figure 5.10 and the e-dag built from **x-dags**



## Building Matching Structures

A new matching structure is created when we find an element matching with an item in the *looking-for-set*. This matching structure has only those qids for which the match occurred. In this way there may exist two or more sub-matches (each with different set of qid's) for a c-node in the combined matching structure.

**Table 5.3:** Walk through of evaluation of XPath queries on XML document of Figure 5.10 and 5.11. In the second column, Start (End): A(x,y) denotes the start (end) element event for an element, A(x,y). The last column shows Looking-for set at the end of processing the event. Steps from 2 to 12 are described in same format as step 1.

| **Step** | **Event** | **Matches** | **Looking-for set** |
|---|---|---|---|
| 1 | Start:Root(0,0) | $(Root^{1,2,3},0)$ | $\{(W^{1,2,3},*), (Y^{1,2,3},*)\}$ |
| **Comments:** Add $(W^{1,2,3},*)$ and $(Y^{1,2,3},*)$ to L, since Root is an open, relevant element matching Roots of all x-dags. Do not start looking for $(X^{1,2},*)$ as there is no open and relevant element matching $(W^{1,2},*)$. Do not start looking for $(X^{3},*)$ as there is no open and relevant element matching $(Y^{3},*)$. Also do not start looking for $(A^{3},*)$ as there is no open and relevant element matching $(X^{3},*)$. | | | |
| 2 | Start:Y(1,1) | $(Y^{1,2,3},*)$ | $\{(W^{1,2,3},*),(Y^{1,2,3},*),(U^{1,2,3},2), (X^{3},*)\}$ |
| Start looking for $U^{1,2,3}$ at level 2 since $U^{1,2,3}$ is connected to Y by a *child* edge having qids 1,2 and 3 in the c-dag, and Y is matched at level 1. Start looking for $(X^{3},*)$ since $X^{3}$ is connected to Y by a *descendant* edge having qids 1,2 and 3 in the c-dag. Do not start looking for $(X^{1,2},*)$ as there is no open and relevant element matching $(W^{1,2},*)$. Continue looking for $(W^{1,2,3},*)$, $(Y^{1,2,3},*)$ because any element with these tags in the sub-tree of this element will also be a candidate for matching the same. | | | |
| 3 | Start:U(2,2) | $(U^{1,2,3},2)$ | $\{(W^{1,2,3},*), (Y^{1,2,3},*) ,(X^{3},*)\}$ |
| Stop looking for $U^{1,2,3}$ until we see the end of this element. Because, the level of any element we come across is greater than 2 before we see the end of this element. | | | |
| 4 | End:U(2,2) | $(U^{1,2,3},2)$ | $\{(W^{1,2,3},*),(Y^{1,2,3},*),(X^{3},*), U^{1,2,3},2)\}$ |
| There is a total matching at U(2,2) for the three Rxp's 1,2 and 3 represented as $\mathbf{M}_{U,2(1,2,3)}$ since it is a leaf in the c-dag. This matching is propagated to the appropriate sub-matching of $\mathbf{M}_{Y,1(1,2,3)}$ , which is the only parent-matching of $\mathbf{M}_{U,2(1,2,3)}$. Looking for set is built in same manner as for step 2. | | | |
| 5 | Start:W(3,2) | $(W^{1,2,3},*)$ | $\{(W^{1,2,3},*),(Y^{1,2,3},*),(X^{3},*),(X^{1,2},3)\}$ |
| Start looking for $(X^{1,2},3)$ since $W^{1,2,3}$ has an outgoing *child* edge having qids 1 and 2 registered at it to $X^{1,2,3}$. | | | |
| 6 | Start:X(4,3) | $(X^{3},*),(X^{1,2},3)$ | $\{(W^{1,2,3},*),(Y^{1,2,3},*),(X^{3},*), (A^{1,3},4)\}$ |
| Start looking for $(A^{1,3},4)$ since $X^{1,2,3}$ has outgoing edges labeled *child* and having qids 1 and 3 respectively to $A^{1}$ & $A^{3}$. | | | |

| | | | |
|---|---|---|---|
| 7 | Start:A(5,4) | $(A^{1,3},4)$ | $\{(W^{1,2,3},*), (Y^{1,2,3},*), (X^{3},*)\}$ |
| Continue looking for $(W^{1,2,3},*)$, $(Y^{1,2,3},*)$ and $(X^{3},*)$ as these are still candidates to match with elements in the sub-tree that follows in the input document. | | | |
| 8 | End:A(5,4) | $(A^{1,3},4)$ | $\{(W^{1,2,3},*), (Y^{1,2,3},*), (A^{1,3},4), (X^{3},*)\}$ |
| There is total matching at A(5,4) for the two Rxp's 1 and 3 represented as $M_{A,5(1)}$ and $M_{A,5(3)}$. $M_{A,5(1)}$ is propagated to the appropriate submatching of $M_{W,3(1,2,3)}$. $M_{A,5(3)}$ is propagated to the appropriate submatchings of $M_{X,4(3)}$ , $M_{W,3(1,2,3)}$, $M_{Root,0(1,2,3)}$ which are parent matchings of $M_{A,5(3)}$. Looking for set is built in same manner as for step 6. | | | |
| 9 | End:X(4,3) | $(X^{3},*),(X^{1,2},3)$ | $\{(W^{1,2,3},*),(Y^{1,2,3},*),(X^{3},*),(X^{1,2},3)\}$ |
| There is total matching at X(4,3) for the three Rxp's 1,2 and 3 represented as $M_{X,4(1)}$ , $M_{X,4(2)}$ and $M_{X,4(3)}$. $M_{X,4(1)}$ has total matching for the Rxp 1 and $M_{X,4(2)}$ has total matching for the Rxp 2. Because sub-matching of $M_{X,4(1)}$ for Rxp 1 in the form of $M_{A,5(1)}$ is total. And $M_{X,4(2)}$ for Rxp 2 corresponds to a leaf in the c-dag (x-dag(2)). $M_{X,4(1)}$ and $M_{X,4(2)}$ are propagated to the appropriate sub-matchings of $M_{Root,0(1,2,3)}$, $M_{W,3(1,2,3)}$ and $M_{Y,1(1,2,3)}$, which are the parent-matchings of $M_{X,4(1)}$ and $M_{X,4(2)}$ $M_{X,4(3)}$ is propagated to the sub-matching of $M_{W,3(1,2,3)}$ and $M_{Y,1(1,2,3)}$. Looking for set is built in same manner as for step 5. | | | |
| 10 | End:W(3,2) | $(W^{1,2,3},*)$ | $\{(W^{1,2,3},*),(Y^{1,2,3},*),(U^{1,2,3},2),(X^{3},*)\}$ |
| There is total matching at W(3,2) represented as $M_{W,3(1,2,3)}$ for Rxp's 1,2 and 3. Because $M_{W,3(1,2,3)}$ has sub-matchings in the form of $M_{X,4(1)}$ , $M_{X,4(2)}$ and $M_{X,4(3)}$ which are total. This matching is propagated to the appropriate sub-matching of $M_{Root,0(1,2,3)}$, which is the only parent-matching of $M_{W,3(1,2,3)}$. Looking for set is built in same manner as for step 2 or 4. | | | |
| 11 | End:Y(1,1) | $(Y^{1,2,3},*)$ | $\{(W^{1,2,3},*), (Y^{1,2,3},*)\}$ |
| $M_{Y,1(1,2,3)}$ has total matching for the three Rxp's since all its sub-matchings $M_{X,4(1)}$, $M_{X,4(2)}$ ,$M_{X,4(3)}$ and $M_{U,2(1,2,3)}$ are total. So this matching is appropriately propagated to the sub-matching of $M_{Root,0(1,2,3)}$. Looking for set is built in same manner as for step 1. | | | |
| 12 | End:Root(0,0) | $(Root^{1,2,3},0)$ | Not Applicable |
| There are total matchings at $M_{Root,0(1,2,3)}$ for the three Rxp's 1,2&3. | | | |

In the example, the three matching structures $M_{X,4(1)}$ , $M_{x,4(2)}$ and $M_{x,4(3)}$ were created in this fashion. Also with respect to same set of *qid*'s, there may be more than one element that match with a c-node and a matching structure is created for each such match i.e., there may exist another $M_{X,i(3)}$ (element X with id *i* different from 4) as a sub-matching of $M_{Root,0(1,2,3)}$. The sub-matchings are initially empty for a newly created matching. In figure 5.11 showing combined matching structure for the example, we grouped pointers related to sub-matchings for a single c-node in one slot of the third row of the matching structure.

One of the key steps in this algorithm is propagation. At the end element event of an element *x* with id '*i*', a check is performed to see if there is a total matching at $M_{x,i(Rxp'set)}$ for each of the *Rxp*'s in *Rxp*'set. This propagation has two cases.

1. If the corresponding c-node is a leaf with respect to an *Rxp* in the combined x-dag[3], the matching at this c-node is total for that *Rxp*. We propagate this match to its parent matchings appropriately (A point to be noted here is that the set of *qid*'s of a parent matching of a sub-matching is always a superset of the qid's of sub-matching).
2. If it is not a leaf with respect to an *Rxp*, this matching represents a total matching with respect to that *Rxp* if and only if all its sub-matchings with respect to that *Rxp* are total. If we found appropriate total matching with respect to that *Rxp* for all the child sub-matchings, we propagate this matching to its parent-matchings appropriately.

In the example, the first case occurs at step 9 where $M_{x,4(2)}$ is a leaf and has total matching with respect to *qid* 2. So this is propagated to its parents $M_{W,3(1,2,3)}$. $M_{Y,1(1,2,3)}$ and $M_{Root,0(1,2,3)}$. The second case also occurs in step 9 for the *Rxp* 1. $M_{X,4(1)}$ is a non-leaf with respect to qid 1 and has total matching with respect to qid 1 for sub-matching $M_{A,5(1)}$. SO this is also propagated to its parents $M_{W,3(1,2,3)}$. $M_{Y,1(1,2,3)}$ and $M_{Root,0(1,2,3)}$.

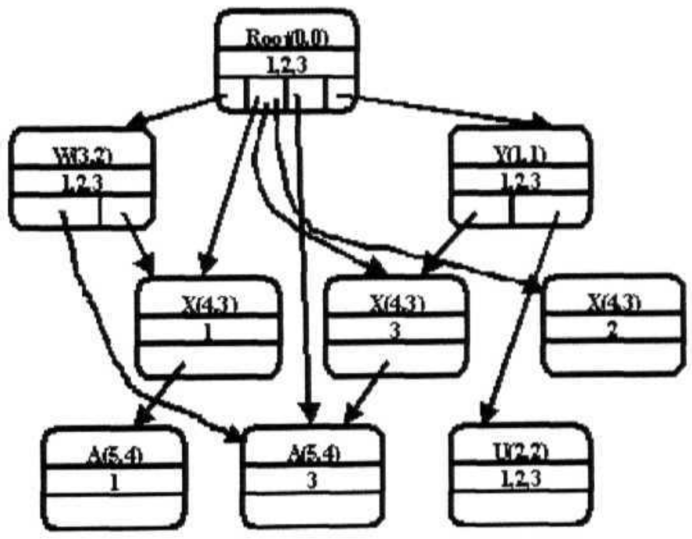
[3] A c-node can be a leaf with respect to one Rxp, i.e. has no outgoing edge with respect to that Rxp and at the same time a non-leaf with respect to another Rxp, i.e. has an outgoing edge with respect to this Rxp.

**Emission of output**

At **the end** document event, a check is **done to see if total** matching exists at 'Root' matching **with** respect to all *Rxp*'s. After this, before emitting the output, we traverse through the combined matching structure **with** respect to all individual x-trees in an iteration and determine total matching with respect to x-tree at each matching structure for all qids present at the matching structure. Then we traverse through the combined matching structure again with respect to each individual x-tree to emit the correct output when we first visit a matching structure that corresponds to the output x-node.

Consider the following scenario in emission of output. M(xl,qid) is a matching structure corresponding to x-node x1 of x-tree(qid) where x1 has an outgoing edge labeled *ancestor* or *parent* to an x-node x2. In this case, a matching structure for x2 is found as a parent matching structure of M(xl,qid). In the example, there arc total matchings for all 3 *Rxp*'s at $M_{Root,0(1,2,3)}$., which are given in figure 5.11.

**Figure 5.17: Combined matching structure at the end of executing algorithm on example**



The total matchings at Root with respect to each *Rxp* are,
{Root→0,Y→1,W→3, U→2,X→4,A→5} for 1st *Rxp*.
{Root→0,Y→1,W→3, U→2,X→4} for 2nd *Rxp*.
{Root→0,Y→1, W→3, U→2,X→4,A→5} for 3rd *Rxp*.
Elements in bold are the output.

## 5.4 Experimental Results

We performed our experiments on Windows 2000 Advanced Server with 450 MHz Intel Pentium 2 processor and 1024 MB as Main memory. We implemented XAOS and YALXP in Java 1.4.1. We used xml4j_2_0_9, developed by IBM as SAX parser. We implemented various versions of XAOS and modified Xalan 2.5 to perform our experiments. We also implemented custom XML generator that generates XML document based on the queries generated by custom XPath generator (YALXP). Below we present details of each implementation.

**XAOS (OSPCN) - One SAX Parsing Commonality Not Exploited**

This version of XAOS has arrays of structures for multiple queries. Data structures required for processing each query such as x-tree, x-dag, looking-for-set and matching structure are maintained individually for each query. Processing at each event takes place in iterations for each query individually using the respective data structures for that query.

**XAOS (MSP) - XAOS (Multiple SAX Parsing)**

XAOS executed individually for each query in a loop. For n queries, SAX parsing takes place n times.

**Xalan (ODP) - Xalan (One DOM Parsing)**

This is the version of Xalan in which only one in-memory DOM representation of the input document is built. The same will be queried individually for each query one after the other.

**Xalan (MDP) - Xalan (Multiple DOM Parsing)**

In this version of Xalan (MDP), construction of in-memory DOM takes place for each query. Hence evaluation of a query takes a lot of time, thereby increasing the total time taken for multiple queries.

**Custom Xpath Generator (YALXP)**

We customized the custom Xpath Generator used by XAOS to generate multiple queries with commonality. Custom Xpath Generator (XAOS) takes "[-c<max_children>] [-e<number of elts>] [-nr (no-recursion)] [-na (no ancestors or parents)]" as parameters and generates a single query based on these parameters. Each query has unique node tests and each node test has alphanumeric characters generated randomly.

*Sample query generated by Custom Xpath Generator (XAOS)*

"//k3[.//q3 and l0]//c2[.//a2 and .//l1 J//q2"

Custom Xpath Generator (YALXP) takes the number of queries 'n' to be generated as input in addition to above parameters and generates 'n' queries. Each query has 6 unique node tests pseudo-randomly generated from the domain {U, V, W, X, Y, Z} so that different queries have significant commonality.

*Sample queries generated by* Custom *Xpath Generator (YALXP)*

1. /descendant: :Y [ancestor: :W and /child: :X [ descendant: :V ]/descendant::U]/descendant::Z
2. /descendant::l.Vdescendant::Y[dcscendant::Z and /descendant::V]/child::W/descendant::X
3. /descendant: :V/child::X[ descendant ::W[ child: :Y]/ancestor::U]/child::Z
4. /descendant::X/child::Z[child::V/ancestor::Y and /child::U]/ancestor::W
5. /descendant::U/ancestor::Y[child::Z and /child::V]/parent::W/ancestor::X

## Custom XML Generator

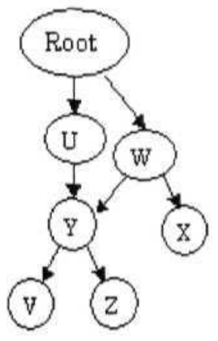
We developed a custom XML generator that generates an **XML** document based on the queries generated by Custom XPath Generator (YALXP). **For each** query a minimal XML document having elements, which conform to the relationships between different node tests within the XPath query, is generated. All the minimal documents generated in this fashion are concatenated and enclosed within a super element to generate the final XML document for multiple queries. This generated document has total matches for at least 9 out of 10 queries.

**Figure 5.18: Sample query and doc generated from the same**

***Sample query***

Descendant::U/descendant::Y[descendant::V and ancestor::W/descendant::X]/descendant::Z

***x-dag for above query***



***Sample minimal XML document generated***

```
< W >
    < U >
       < Y >
          < X >< Z ></ Z ></ X >
          < U>< Y >< X >< Z ></ Z ></ X >
                     < U >< Y >< X ></ X >
                                < V ></ V >
                          </ Y >
                     </ U >
                     < X ></ X >
                </ Y >
          </ U >
          < X >< Z ></ Z ></ X >
       </ Y >
    </ U >
</ W >
```

**Results**

We define *commonality factor* (CF) as a measure to indicate the degree of commonality among multiple queries with forward and backward axes. This factor gives some hint about the amount of reduction in the number of matching structures created and the number of traversals (both through dag and matching structures) by creating c-dag.

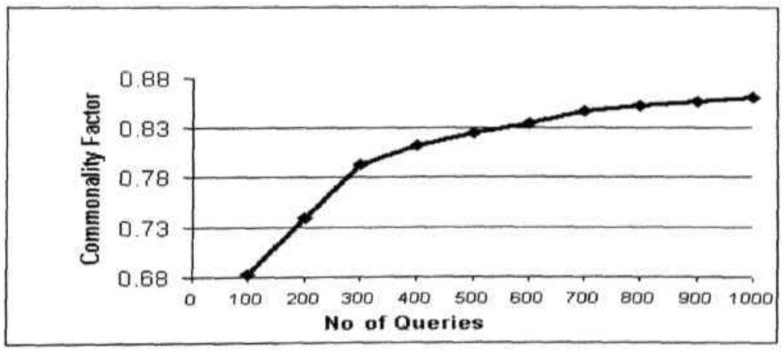
$$CF(N) = 1 - \frac{Tcn + Tce}{Txn + Txe} = 1 - \frac{Tcn + Tce}{7*N + 6*N + Tb},$$

where *N* is number of queries, *Tcn* is Total no of c-nodes in c-dag, *Tee* is Total no of edges in c-dag, *Txn* is Total No of x- nodes in *N* x-dags, *Txe* is Total no of edges in *N* x-dags and *Th* isTotal No of Backward axes in *N* x-dags.

**Table 5.4** commonality factors for different no of queries

| N | Tb | Tcn | Tee | C F(N) | Time to build **cdag** (sec) |
|---|---|---|---|---|---|
| 100 | 55 | 190 | 239 | 0.683 | <1 |
| 200 | 97 | 308 | 393 | 0.74 | 1 |
| 300 | 153 | 358 | 482 | 0.793 | 1 |
| 400 | 206 | 426 | 589 | 0.812 | 2 |
| 500 | 267 | 486 | 698 | 0.825 | 3 |
| 600 | 312 | 543 | 794 | 0.835 | 4 |
| 700 | 355 | 572 | 873 | 0.847 | 6 |
| 800 | 401 | 627 | 964 | 0.853 | 8 |
| 900 | 447 | 686 | 1046 | 0.857 | 10 |
| 1000 | 501 | 743 | 1130 | 0.861 | 12 |

**Figure 5.19: Graph for the results in Table 5.4**



**Comparisons between YALXP, XAOS (MSP), Xalan (ODP) and Xalan (MDP)**

In our experiments, we observed that Xalan (MDP) takes the highest amount of time due to construction of in-memory DOM and querying the same for each query individually. We also observed that as the number of queries grows, due to the parsing cost involved for each query and the traversal through all the matching structures for each query, XAOS (MSP) takes more time. By adapting the approach of XAOS for multiple queries, YALXP takes lesser time. This is because parsing cost comes down heavily by virtue of single document-order traversal in YALXP. However, the memory consumption of YALXP grows as the number of relevant elements in the input XML document grows due to creation of matching structures for multiple queries. But, this approach by exploiting commonalities among XPath queries with backward and forward axes, works much better in scenarios where the number of relevant elements is likely to be less and the document size is big as is evident from results. We found that Xalan (ODP) performs closer to YALXP than others in CPU time. (Refer to table 5.5)

Table 5.5 Comparison of CPU & Memory usage of various algorithms on docs of sizes given in Table 5.6

| No of queries | YALXP | | XAOS (MSP) | | Xalan (ODP) | | Xalan (MDP) | |
|---|---|---|---|---|---|---|---|---|
| | CPU | Mem | CPU | Mem | CPU | Mem | CPU | Mem |
| 100 | 29s | 26.5M | 81s | 9M | 64s | 33.7M | 201s | 28.5M |
| 200 | 115s | 73M | 411s | 12M | 256s | 58.2M | 890s | 50M |
| 300 | 426s | 148M | 1172s | 13.8M | 602s | 84M | 1978s | 68.6M |
| 400 | 723 s | 253M | 1818s | 13.5M | 1029s | 108.2M | 3544s | 92.5M |
| 500 | 1429s | 569M | 3177s | 18.15M | 1657s | 135M | 5523s | 105M |

Table 5.6 Sizes of input documents for algorithms in table 5.5

| No of queries | Doc. Size (No of elems) | MX-Stream – Size* | XAOS(MSP) -Size* |
|---|---|---|---|
| 100 | 511K (15906) | 4452 | 512 |
| 200 | 1.04 M (35066) | 13195 | 1129 |
| 300 | 1.59M(53348) | 24382 | 1899 |
| 400 | 2.12M(70669) | 36635 | 2383 |
| 500 | 2.68M(89349) | 50486 | 3084 |

*Size - Combined Matching Structure (YALXP) / Max size of Matching Structure for a query (XAOS (MSP))

Figure 5.20: Graph for the results in Table 5.5

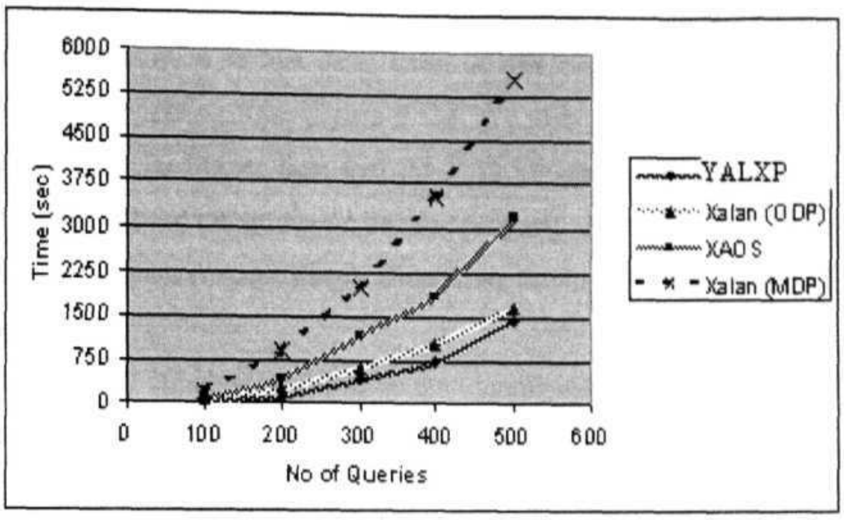
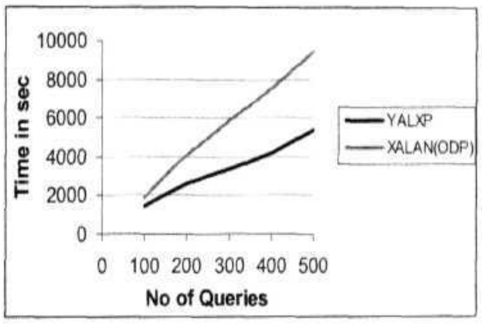
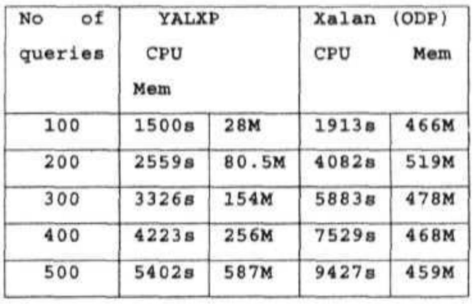


Figure 5.21 Graph and table with CPU lime and Memory taken by MX-Stream and Xalan (ODP) on a document of 15,00,000(approx) elements (The doc. sizes were 9.99,11,10,9.81 and 9.88M respectively for 100,200,300,400 and 500 Q.)





| No of queries | YALXP CPU | Mem | Xalan (ODP) CPU | Mem |
|---|---|---|---|---|
| 100 | 1500s | 28M | 1913s | 466M |
| 200 | 2559s | 80.5M | 4082s | 519M |
| 300 | 3326s | 154M | 5883s | 478M |
| 400 | 4223s | 256M | 7529s | 468M |
| 500 | 5402s | 587M | 9427s | 459M |

We also implemented XAOS (OSPCN) with One SAX Parsing and Commonalities among queries Not exploited. We observed that, though the processing time is in between that of YALXP and Xalan (ODP), the memory consumption is prohibitively large from 300 queries on. So we excluded the same from comparison in our experiments.

To see the performance of our algorithm on large documents, we compared Xalan (ODP) and YALXP (can be imagined as XAOS with one SAX parsing and commonalities exploited) on a document of size around 10M with 15,00,000 elements (elements without any attributes, text etc) for different no of queries. (This document has the same number of relevant elements as that for smaller documents but the number of irrelevant elements is large. The **number** of events fired is equal to twice the number of elements in the document in this case. Even in this case YALXP performs better than Xalan (ODP) in processing time (figure 5.15). We also found that the memory required for Xalan (ODP) is far higher than that for YALXP up to 400 queries. More over, the input document in these comparisons has the property that no irrelevant element has an *ancestor* that is relevant (except Root). Otherwise, the time taken by Xalan (ODP) would be even worse.

Xalan (ODP) failed to complete on documents with more than 52,00,000 elements (size 27M) due to thrashing of Operating System. From these experiments we can conclude that the memory consumption of YALXP depends only on the number of relevant elements and on number of queries, but not on document size. The processing time taken by YALXP increases for large documents due to the minimum processing required at each irrelevant event. We did not compare XAOS (MSP) and Xalan (MDP) on documents of 10 M for the obvious reason that processing time goes out of comparison due to multiple parsing.

In cases where the amount of memory available is less and the number of queries is high, YALXP can be run on a subset of queries at a time and it still performs much better in terms of processing time. We did not compare YALXP with other multiple XPath processing engines such as Xfilter[82], Yfilter|83], Xtrie and Mtrie etc due to the fundamental difference in the fragment of XPath queries we address. Yfilter[87], Xtrie[86] cannot be applied to queries with backward axes. Our main aim is to exploit commonality among queries with both forward and backward axes.

## 5.5 Conclusions

We presented YALXP for multiple queries that have forward and backward axes for streaming XML data with the following advantages.

1. Ability to handle backward axes on streaming XML.
2. Evaluation of multiple queries in single document-order traversal.
3. Exploiting commonality among different queries with forward and backward axes thereby sharing processing and memory among different queries.

Our experiments show that YALXP performs better than Xalan(ODP) for multiple queries with single DOM construction.

# Chapter 6

# Conclusions and Future Scope

Central servers are not able to scale up to the ever-growing demands of users on Internet. Right from Domain Name System, which moved from centralized system to distributed system in 1983 to the recent 1999 Victoria's secret web cast, which could not serve any request make this point very clear. Though, in recent years both network and server capacity have improved, response time continues to challenge researchers, as centralized web systems are not able to scale. Improving the power of a single server does not solve the web scalability problem in a foreseeable future. Technologies like clusters, load balancing, grid computing, content addressable networks etc are trying to meet the network and processing requirements. Application Layer Multicast is one such technology.

Application Layer Multicast protocols build overlay networks connecting end hosts on top of routers and route data on this overlay. Researchers have proposed many Application Layer Multicast protocols like End System Multicast [1-2], Topology Aware Group Communication [22], Host Multicast Protocol [13], Content Addressable Network [18-19] etc., in the recent past. All these protocols vary in performance mainly based on their knowledge about the underlying physical topology. For example, Content Addressable Network (CAN) performance is heavily taxed by its lack of underlying topology. An overlay can perform better only if it is in alignment with the physical router topology. We proposed a new overlay protocol (Appcast) that keeps the over all link utilization in the overlay to be optimal i.e., from the network perspective, the constructed overlay ensures that redundant transmission on physical links is kept minimal.

In overlay multicast, hosts take over the routing functionality of normal network routers and forward the packets among the participating hosts. Our topology-building algorithm considers each joining hosts' capacity and each host can specify to how many hosts it can forward the data in the overlay. As the overlay grows, the delay increases for the data to reach from the source to the destination, as the number of intermediary routing hosts increase. So, each host can specify how much delay it can tolerate. The protocol allows for the overlay to incrementally evolve into a better structure as more information becomes available. Though overlays cannot perform on par with native IP multicast, they outperform unicast and also Application Layer Overlays follow a natural producer distributor hierarchical relationship, which is definitely a big advantage that can boost e-commerce.

Network performance is not only based on the media capacity, but also based on the protocol stack that lays down the rules of communication. The Transmission Control Protocol/Internet Protocol (TCP/IP) standard is the dominating protocol in the communication world. In principle, broadcasting via satellite is simpler than for terrestrial networks, which are not naturally broadcast networks. However, TCP/IP has shielded this nature. TCP/IP follows a layered approach in which the lower level protocol details are shielded by high-level protocols. Broadcast nature of the media, is a physical line property, which falls into physical layer in OSI layered model that naturally gets shielded by TCP - transport layer.

While there are many vendors improving TCP efficiency over satellite for only specific applications like video broadcast etc, there is no way for a general application developer to exploit the broadcast property in his application. We proposed a method 'Multicast Spoofing', using which we pass on the broadcast benefit to any application.

Multicast application development support is much complex. Due to this many multicast applications are written to use multiple unicast connections. While creation of overlays can reduce the burden on routers and increase the efficiency in moving packets to destination hosts, the real advantage of multicast cannot be achieved unless and until the development support is given. Many application multicast protocols proposed their own API almost in parallel with TCP/IP. This can be seen from the proposals of Scattercast and Yoid. Even protocols like ESM proposed their own API for multicast application development. In contrast to this, we proposed same existing Internet protocol API like HTTP, SMTP and FTP for both unicast and multicast. However, the administrator of the developed application has to mention how the user would use the application, i.e., unicast or multicast. We showed the generic multicast application development using SOAP, XML etc., and web services protocols and implemented on three different kinds (one-way information push, interactive auctions and database replication) of multicast applications.

XML is sooner or later becomes de-facto standard for information exchange over Internet. Our applications also are developed using XML. Any XML document can be rendered into meaningful and validated information only by using its corresponding XSD-XML schema definition file. This makes XML document processing much slower when compared to traditional HTML parsing. When used on overlays XML can even further degrade the performance coupled with overlay delay. Overlays disseminate information in a hierarchical and incremental fashion. Also, not all members in an overlay or multicast group need exactly same information. In this case, if we can filter the information at different levels of overlay, while the information is distributed, we can minimize the bandwidth and processing requirement. Each member's interests are represented by Xpath [78] query and the information to be distributed is in the form of XML [80] document as XML is gaining importance as the standard for information exchange. We looked at the XML document-processing problem particularly in the context of application layer multicast and proposed a new Xpath processing algorithm called YALXP - Yet Another Light weight Xpath [78] processor. YALXP algorithm processes multiple Xpath queries over a XML document in single document traversal i.e. in one pass (traversing document only once) it can answer multiple queries.

## Discussions and Future Scope

Many researchers debate and doubt the need of *Application Layer Multicast* once IP V6 becomes available all over Internet. We argue that the multicast supported by IP V6 can well be utilized by network infrastructure requirements like multicasting route tables etc., and since Application Layer Multicast follows the natural producer and distributor model, it is still required even in the presence of IPV6 as a natural e-extension dealer and distributor network to e-dealer and e-distributor network.

YALXP Extensions: We proposed YALXP – Yet Another Light Weight XPath processor, to exploit the commonality among user queries to filter information efficiently and distribute the same over the overlay. YALXP handles only few axes like parent, child, ancestor and descendant. Both the algorithms XAOS, YALXP can be extended to handle horizontal axes such as *following, following-sibling, preceding, preceding-sibling* and more of XPath. Some work has already been initiated in this direction and following changes to XAOS and YALXP that are needed to make them work for horizontal axes have been identified.

Horizontal backward axes such as *preceding* and *preceding-sibling* have to be converted to corresponding horizontal forward axes such as *following-sibling.* To support horizontal axes, especially "*preceding*", the algorithm has to support the wild card operator '*' of XPath.

Figure 6. 1: Translation of **"/descendant:: A/preceding: :B"** to a reverse-axis free query

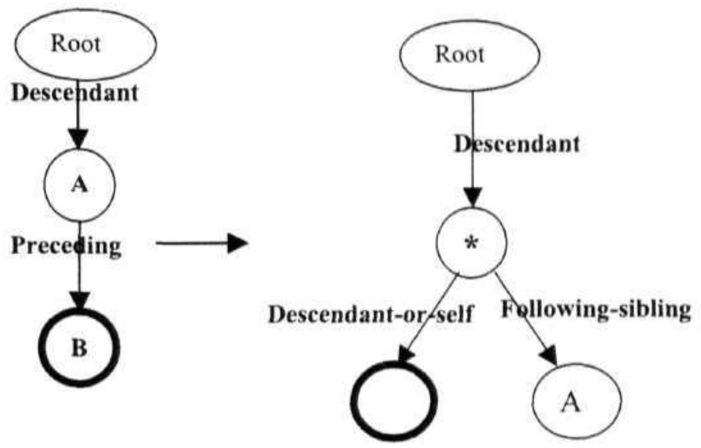
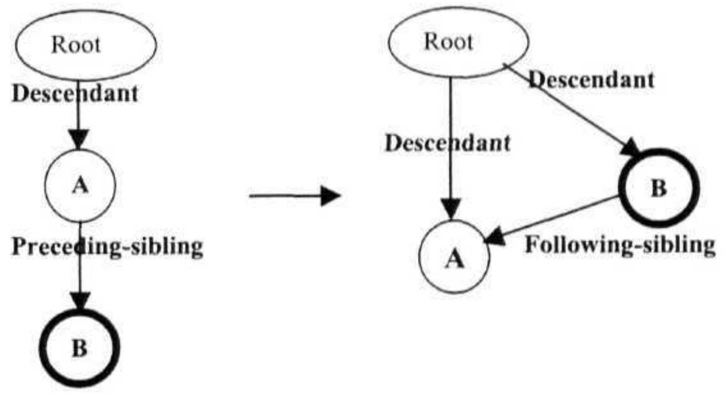


Figure 6. 2: Translation of **"/descendant::A/preceding-sibling::B" to** a reverse-axis free query

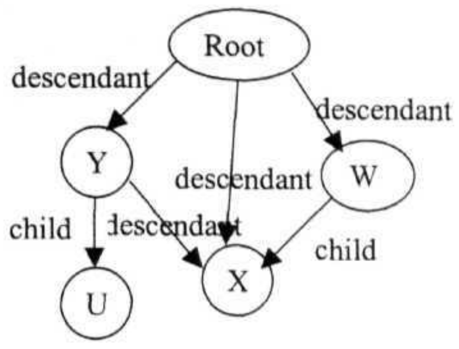


Either *Optimistic Propagation* proposed in XAOS or *Pessimistic Propagation* has to be resorted to in case of horizontal axes. This is because in case of non-horizontal axes, *descendants* and *children* of an element will be found before the end tag of the element is seen where as *following and following-sibling* can be found only after the end tag of the element is seen. Pessimistic propagation means assuming *no total matching* or *total* *matching to be false* at sub-matching-structures corresponding to child c-nodes, connected by an edge with label equal to a horizontal forward axis, as soon as the sub-matching-structures are created. If it has been ascertained, at the end event of element corresponding to the sub-matching-structure, that there exists a total matching at this element, the pessimistic propagation has to be undone recursively from the parent and ancestral matching structures. There is no real advantage of pessimistic propagation over optimistic propagation. Pessimistic propagation results in lesser processing when the number of relevant elements that do not have total matching is more than number of relevant elements that have and Optimistic propagation results in lesser processing when the number of relevant elements that have total matching is more than number of relevant elements that do not have.

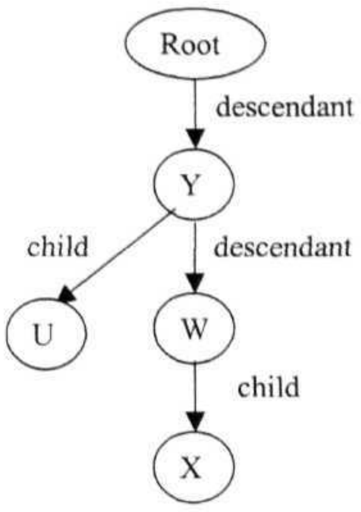
Handling horizontal axes and wild card operator '*' is much more complicated. Since it requires buffering more elements than what vertical axes require. Consider the horizontal location path "*//A/preceding::**". This requires buffering (in the form of matching structures) all the preceding-siblings of *A* along with all their descendants. More over, "*" means selecting all nodes including *text()*, *attributes*, *comments* and other components of XML. Therefore, the matching structures have to be selectively released while processing large documents.

Optimization can be done to minimize the buffering of data by storing Boolean values for total matching at matching structures, the subtrees of which do not have any output nodes, instead of whole matching structures. Also, YALXP can be optimized in the memory used especially to buffer irrelevant events even after their irrelevance is known to be certain.

The query rewriting can be optimized, even in cases where DTD is not available, to reduce the graph behavior of queries by converting them into more tree-like patterns. This can be explained by the following example. Consider the query *Q1: descendant: :X[/ancestor::Y/child::U and /parent: :W]*. One can easily infer that W has to be a descendant of Y since X can have only one parent, i.e. W and hence Y has to be an ancestor to X as well as to W. This results in a tree versus a more complicated x-dag shown in figure 6.3.



**X-dag for Ql**



**Optimized X-dag for Q1**

Figure 6. 3: Reduction in the graph behaviour of queries after optimization

There can be a better way of storing *qids* at each *c-node* and *cedge* than storing boolean values (whether particular qid *exists* at the edge/c-nodc or *not)* for all qids. This is a trade-off between memory and processing time. We gave priority to processing time over memory. As a result our c-dag size grows large for large number of queries. Optimization can be done to reduce the size of c-dag using scalable data structures such as hash tables.

YALXP exploits commonality among x-dags of queries. It can be explored whether arbitrary commonality among queries, by finding common sub expressions of different queries, can also be exploited in YALXP. MQSPEX exploits such arbitrary commonality by constructing a single query plan for all the queries.

**E-Service Propagation:** As per our model, producers can give service definition files to distributors using which distributors can set up their services and further they can give service definition files to consumers who can set up services to consume the services offered by producers and distributors. We could not actually install E-Service on distributor/consumer machine dynamically. We did the whole experiments in our LAN environment and we wrote the consumer services separately for each application as installable executables.

How the consumers can set up the listening service? What do they do with the information they receive? Unless there is some natural language, it is difficult to specify these requirements. In general, the WSDL specifies only the input, output parameters, the remote service names, the ports at which these services listen etc. But it cannot specify how to act upon those parameters. But the service provider is aware of how to process the input parameters and arrive at the result to send the response. So, here the service provider informs the consumers, how to call its service. In multicast, it is different in that the consumer should already be having the consumer service to which the producer can push the information. Unless, all consumers set up their receiving services in same way, the producer cannot push the information. So, he advertises the service definition in terms of input and output parameters, just like in WSDL. However, the difference is that, in normal course the consumers call the producer service and in multicast, the producer calls the consumer's service. The consumer has to set up its service dynamically by looking at the WSDL.

## E-Service Languages

As discussed earlier, WSDL is just like an Interface Definition Language. It cannot describe the E-Service in totality. In a multicast environment, one needs a language that is common on all platforms, such that once described, it can be implemented on any platform. Work in this direction of describing a web service in XML and executing on any platform has been described in [52]. It details on program statements, expressions, variables etc. In our case of multicasting, this work may help if every consumer would like to use the received information in similar fashion. This can be applied in case of distributors, where in producer can program how the information be distributed. If so, the producer can place the service in XL language [52], which the distributor can download. Digital signatures can be used to authenticate the code. However, if the consumers would like to use the received information in different fashions, then this will not help. The producer can dictate what the consumer can receive, but not how to consume the same. So, we feel that, the programming languages for the web services paradigm must vary from simple to complex. For example, it should allow simple statements like below to receive the office order, store it and display or mail it.

```
{
```

```
Receive Office Order From http://www.organization.com/orders/
Store Office Order In d:\folder
Display Office Order
}
{
Receive Office Order From http://www.organization.com/orders/
Mail Office Order admin@myoffice.com
}
```

## References:

[1] Hua Chu, Y., Rao, S., and Zhang, H. "A Case For End System Multicast." *Proceedings of ACM Sigmetrics '00* (Santa Clara, CA, June 2000).

[2] Hua Chu, Y., S. G. Rao, S. Seshan, and H. Zhang. "Enabling Conferencing Applications on the Internet using an Overlay Multicast Architecture". *Proceedings of ACM SIGCOMM*, August 2001.

[ 3] Chawathe, Y. "Scattercast: An Architecture For Internet Broadcast Distribution As An Infrastructure Service". PhD thesis, University of California, Berkeley, Dec. 2000.

[ 4] Chawathe, Y., Mccanne, S., And Brewer, E. A. "An architecture for Internet content distribution as an infrastructure service". http://www.cs.berkeley.edu/~yatin, 1999.

[ 5] Jannoti, J., Gifford, D.K., and Johnson, K.L. "Overcast: Reliable Multicasting With An Overlay Network". Proceedings of the *4th Symposium on Operating System Design and Implementation (OSDI)* (San Diego, CA, Oct. 2000), USENIX.

[ 6] Prabhakar Raghavan, "Beyond Web Search Services" *IEEE Internet Computing*, Mar-Apr 2001.

[ 7] Peter N. Yianilos and Sumeet Sobti, "The Evolving Field Of Distributed Storage" *IEEE Internet Computing*, Sept-Oct 2001.

[ 8] D.Cheriton and S.Deering, "Host Groups: A Multicast Extension For Datagram Internetworks", *Data Communications Symposium*, Sept. 1985, pp. 172-79.

[ 9] A. Shaikh, M. Goyal, A. Greenberg, R. Rajan, and K. K. Ramakrishnan. "An OSPF Topology Server: Design And Evaluation", 2001. http://www.cis.ohio-state.edu/mukul/research.html.

[10] D. Pendarakis et al., "ALMI: An Application Level Multicast Infrastructure," *3rd USNIX Symposium on Internet Technologies and Systems*, Mar. 2001.

[11] J. Liebeherr, M. Nahas, and W. Si, "Application-Layer Multicast with Delaunay Triangulations," *IEEE GLOBECOM '01*, also tech. rep. CS-2001-26, Nov. 2001.

[12] S. Zhuang et al., "Bayeux: An Architecture **for Scalable and Fault-Tolerant** Wide-Area Data Dissemination," *11th International. Workshop on Networks and Operating System Support for Digital Audio and Video*, June 2001.

[13] B. Zhang, S. Jamin, and L. Zhang, "Host Multicast: A Framework for Delivering Multicast to End Users," *IEEE INFOCOM '02*, New York, NY, June 2002.

[14] M. Castro et al., "Scribe: A Large-Scale and Decentralized Application-level Multicast Infrastructure," *IEEE JSAC, 2002.*

[15] S. Banerjee, B. Bhattacharjee, and C. Kommareddy, "Scalable Application Layer Multicast," *ACM SIGCOMM'02*, Pittsburgh, PA, Aug. 2002.

[16] S. Banerjee and B. Bhattacharjee. "Analysis of the NICE Application Layer Multicast Protocol". Technical report, UMIACSTR 2002-60 and CS-TR 4380, Department of Computer Science, University of Maryland, College Park, June 2002.

[17] S. Banerjee and B. Bhattacharjee. "Scalable Secure Group Communication over IP Multicast". In Proceedings *of International Conference on Network Protocols*, Nov. 2001.

[18] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. "A scalable content-addressable network". In *Proceedings of ACM Sigcomm*, August 2001.

[19] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. "Application-Level Multicast Using Content Addressable Networks". In Proceedings of *3rd International Workshop on Networked Group Communication*, Nov. 2001.

[20] P. Francis. "Yoid: Extending the Multicast Internet Architecture", 1999. White paper http://www.aciri.org/yoid/.

[21] Francis, P. "Yallcast: Extending The Internet Multicast Architecture", September 1999. http://www.yallcast.com.

[22] Minseok Kwon, Sonia Fahmy. "Topology Aware Group Communication". In proceedings of *NOSSDAV'02*, May 12-14, 2002, Miami, Florida, USA.

[23] Deering, S. And Cheriton, D. "Multicast Routing in Datagram Internetworks and Extended LANs". *ACM Transactions on Computer Systems* 8, 2 (May 1990)

[24] S. Deering. "Host Extensions For Ip Multicasting". *RFC 1112, August.*, 1989.

[25] C. Partridge, D. Waitzman, and S. Deering. "Distance Vector Multicast Routing Protocol". *RFC 1075, November.*, 1988.

[26] R. Perlman, C. Lee, T. Ballardie, J. Crowcroft, Z.Wang, T. Maufer, C. Diot, J. Thoo, and M. Green. "Simple Multicast: **A** Design **For Simple,** Low Overhead Multicast". *Internet Draft, IETF,* March 1999.

[27] D. Thaler, M. Handley, and D. Estrin. "The Internet Multicast Address Allocation Architecture". *RFC 2908, IETF,* September 2000.

[28] D. Thaler, M. Talwar, L. Vicisano, and D. Ooms. "IPv4 Automatic Multicast Without Explicit Tunnels (AMT)". *Internet Draft, IETF,* February 2001.

[29] L. Wei and D. Estrin. "A Comparison Of Multicast Trees And Algorithms". In *Proc. of IEEE INFOCOM,* June 1994.

[30] R. Finlayson. "The UDP Multicast Tunneling Protocol". *Internet Draft, IETF,* March 2001.

[31] R. Finlayson, R. Perlman, and D. Rajwan. "Accelerating The Deployment Of Multicast Using Automatic Tunneling". *Internet Draft, IETF, February* 2001.

[32] H. Holbrook and B. Cain. "Source-Specific Multicast For IP". *Internet Draft, IETF,* November 2000.

[33] L. Sahasrabuddhe and B. Mukherjee. "Multicast Routing Algorithms And Protocols: A Tutorial". *IEEE Network, pp. 90-102, January.* 2000.

[34] M. Doar and I. Leslie. "How Bad Is Naive Multicast Routing". *Proceedings of the IEEE Infocomm.,* 1993.

[35] L. Wei and D. Estrin. "The Trade-Offs Of Multicast Trees And Algorithms". *In International Conference on Computer Communications and Networks.,* 1994.

[36] P. Liefooghe. "Castgate: An Auto-Tunneling Architecture For IP Multicast". *Internet Draft, IETF,* November 2001.

[37] Microsoft Corporation. "The Component Object Model Specification", Version 0.9, October 24, 1995 [online]. http://www.microsoft.com/Com/resources/comdocs.asp.

[38] Microsoft Corporation. "Distributed Component Object Model Protocol-DCOM 1.0", draft, November 1996 [online]. http://www.microsoft.com/Com/resources/comdocs.asp (1996).

[39] Gopalan Suresh Raj, "A Detailed Comparison Of CORBA, DCOM And Java RMI" (with specific code examples)
http://my.execpc.com/~gopalan/misc/compare.html

[40] White paper from Sun Microsystems, "Java Remote Method Invocation - Distributed Computing For Java".
http://java.sun.com/marketing/collateral/javarmi.html

[41] RFP 1050, "Rpc: Remote Procedure Call Protocol Specification" http://rfc.sunsite.dk/rfc/rfc1050.html

[42] Open Group Technical Standard, "DCE 1.1: Remote Procedure Call", Document Number C706 August 1997

[43] V.Baggiolini, M.Vanden, Eynden "Message Oriented Message Oriented Middlewares (Moms)", *Middleware Works hop*-26-March 1999

[44] "SOAP Version 1.2 Part 1: Messaging Framework", *W3C Working Draft* 26 June 2002 http://www.w3.org/TR/soap12-part1/

[45] "SOAP Version 1.2" *W3C Working Draft* 9 July 2001 http://www.w3.org/TR/2001/WD-soap12-20010709/

[46] Heping Shang and Phil Shaw (USA), "Database Event Stream (Des) In XML; Change Proposal For The Replication Facilities."
http://www.sqlstandards.org/SC32/WG3/Meetings/HEL_2000_10_Helsinki_FIN/helO36.pdf

[47] James Snell, Doug Tidwell & Pavel Kulchenko, "Building Distributed Applications - Programming With SOAP", *O'Reilly, Jan 2002*

[48] Arron E. Walsh, Editor, "UDDI,SOAP And WSDL – The Web Services Specification Reference Book", *Pearson Education Asia ISBN 81-7808-669-7, 2002*

[49] Kennard Scriber, Mark C. Stiver, "Understanding SOAP", *SAMS Publishing 2000*

[50] Joseph Ishac, Mark Altaian, "On The Performance Of TCP Spoofing In Satellite Networks", http://citeseer.nj.nec.com/ishac01performance.html

[51] M.Allman, H.Kruse, and S.Ostermann, "On The History Of The Improvement Of Internet Protocols Over Satellite", *ACTS Conference 2000, May 2000*

[52] Daniela Florescu and Donald Kossmann "An XML Programming Language For Web Service Specification And Composition", Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[53] Yongguang Zhang, Dante De Lucia, Bo Ryu, Son K. Dao, "Satellite Communications in the Global Internet Issues, Pitfalls, and Potential"; Hughes Research Laboratories USA

[54] E. Amir. "A map of the Mbone" August 5th, 1996. http://www.cs.berkeley.edu/~elan/mbone.html.

[55] R. Braden. "Extending TCP for transactions-concepts". Internet Request for Comments RFC 1379, November 1992.

[56] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. "TCP Vegas: New techniques for congestion detection and avoidance". *ACM SIGCOMM 1994*, pages 24-35, May 1994.

[57] R. C. Durst, G. J. Miller, and E. J. Travis. "TCP Extensions For Space Communications". *Proceedings of the 2nd ACM Conference on Mobile Computing and Networking*, November 1996.

[58] S. Floyd and V. Jacobson. "Random Early Detection Gateways For Congestion Avoidance". *IEEE/ACM Transactions on Networking*, 1(4):397-413, August 1993.

[59] V. Jacobson, R. Braden, and D. Borman. "TCP Extensions For High Performance". Internet Request for Comments RFC 1323, May 1992.

[60] M. R. Macedonia and D. P. Brutzman. "Mbone Provides Audio And Video Across The Internet". *IEEE Computer*, 27(4):30-36, April 1994.

[61 ] S. McCanne. "NS - Lbnl Network Simulator". http://www-nrg.ee.lb-l.go-v/ns/.

[62] S. McCanne and V. Jacobson. "VIC: A Flexible Framework For Packet Video". *ACM Multimedia,* November 1995, San Francisco, California, pages 511-522, November 1995.

[63] W. R. Stevens. "TCP/IP Illustrated", *Volume 3.* Addison-Wesley Publishing Company, 1996.

[64] Y. Zhang and S. K. Dao. "HBX: High Bandwidth X For Satellite Internetworking". In *Proceedings of the 10th X Technical Conference, X Resource, Issue 17*, February 1996.

[65] Y. Zhang and S. K. Dao. "Integrating Direct Broadcast Satellites with Local Wireless Access", In *Proceedings of the First International Workshop on Satellite-Based Information Services (WOSBIS)*, Rye, New York, November 13, 1996.

[66] David L. Tennenhouse, Jonathan M.Smith, W.David Sincoskie, David J. Wetherall and Gary J. Minden. "A Survey Of Active Network Research". *IEEE Communications Magazine,* 35(1); 80-86, January 1997

[67] Ayman El-Sayed and Vincent Roca et al "A Survey of Proposals for an Alternative Group Communication Service". *IEEE Network • January/February 2003*

[68] B. D. Davison. "A web caching primer". *IEEE Internet Computing,* 5(4):38–45, July/August 2001.

[69] M. Zari, H. Saiedian, and M. Naeem. "Understanding And Reducing Web Delays". *IEEE Computer*, 34(12):30–37, Dec. 2001.

[70] Valeria Cardellini, Emiliano Casalicchio, Michele Colajanni and Philip S. Yu. "The State of the Art in Locally Distributed Web-Server Systems". *ACM Computing Surveys, Vol. 34, No. 2, 263-311, June 2002.*

[71] Charles D. Cranor, Matthew Green, Chuch Kalmanek, David Shur, sandeep sibal and Jacobus E.Van der Merwe, Cormac J.Sreenan; "Enhanced Streaming Services In A Content Distribution Network", *IEEE Internet Computing,* Mar-Apr 2001

[72] James S.Plank, Alessandro Bassi, Micah Beck and Terence Moore, D.Martin Swany and Rich Wolski, "Managing Data Storage In The Network", *IEEE Internet Computing,* Sept-Oct 2001

[73] Manoj Parameswaran, Anjana Susarla, Andrew B. Whinston. "P2P Networking:An Information-Sharing Alternative", *IEEE Computer, 31-38, July 2001*

[74] David Barkai. "Technologies for Sharing **and** Collaborating on the Net." *Proceedings of the First International Conference on Peer-to-Peer Computing (P2P.01), IEEE Computer Society Press, 2002*

[75] Fran Berman, Geoffrey Fox, Tony Hey. "Grid Computing: Making the Global Infrastructure a Reality". *ISBN 0-470-85319-0, John Wiley & Sons*

[76] John Hunt, "Broadcast News", JayDee Technology; http://www.jaydeetechnology.co.uk/broadcast/broadcast.htm

[77] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Byers. "BRITE: An Approach to Universal Topology Generation". *In Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems- MASCOTS '01*, Cincinnati, Ohio, August 2001

[78] XML path language (XPath) version 1.0. Technical report, W3C Recommendation, http://www.w3.org/TR/xpath.

[79] C. Barton, P. Charles, D. Goyal, M. Raghavachari, IBM T.J. Watson Research Center, M. Fontoura, V. Josifovski, IBM Almaden Research Center - "XAOS - Streaming XPath Processing with Forward and Backward Axes"

[80] Extensible Markup Language (XML*) LO.* http://www.w3.org/TR/RECxml

[81] *Xalan-Java 2.5.* http://xml.apache.org.

[82] M. Altinel and M. Franklin - "Xfilter - Efficient Filtering of XML Documents for Selective Dissemination of Information". *Proceedings of the 26th VLDB Conference, Egypt, 2000.*

[83] Y Diao, M. Altinel, M. J. Franklin, Hao Zhang, P.Fischer - "YFilter - Path Sharing and Predicate Evaluation for High-Performance XML Filtering".

[84] B. Ozen, O. Kilic, M. Altinel and A. Dogac - "CQMC - Highly Personalized Information Delivery to Mobile Clients"

[85] N. Bruno, L. Gravano and N. Koudas, D. Srivastava - "lndexFilter - Navigation- vs. Index-Based XML Multi-Query Processing"

[86] C.Y. Chan, P. Felber, M. Garofalakis, R Rastogi "Xtrie - Efficient filtering of XML documents with XPath expressions" *VLDB 2002*

[87] W. Rao, Y. Chen- CSED, Shanghai Jiaotong University and X. Zhang and Fanyuan ma - "Mtrie - Scalable Filtering of Well-Structured XML Message Stream"

[88] A. Tozawa and M. Murata, "Tableau Construction of Tree Automata from Queries on Structured Documents".

[89] J. Chen, D. DeWitt, F. Tian, and Y.Wang, "NiagaraCQ: A Scalable Continuous Query System for Internet Databases".

[90] V Josifovski, M Fontoura, Attila Batta, "TurboXPath - Querying XML Streams"

[91] Ray Whitmer, "Document Object Model (DOM) Level 3 XPath Specification" http://www.w3.org/TR/DOM-Level-3-XPath

[92] Tim Furche, "MQSPEX: Optimizing Multiple Queries against XML Streams". Diploma thesis, University of Munich, July 2003

[93] Georg Gottlob, Christoph Koch, and Reinhard Pichler, "XPath Processing in a Nutshell", *ACM SIGMOD Record* - Volume 32, Issue 2 (June 2003) Pages: 21 - 27

[94] Iliana Avila-Campillo, Todd J. Green, Ashish Gupta, Makoto Onizukaz, Demian Raven and Dan Suciu, "XMLTK: An XML Toolkit for Scalable XML Stream Processing", *In Proceedings of PLANX, 2002.*

[95] J. Pereira, F. Fabret, H. Jacobsen and F. Llirbat., "WebFilter: A high-throughput XML-based publish and subscribe system". *In Proceedings of the 27th VLDB Conference, Roma, Italy, 2001*

[96] Z. Ives, A. Levy, and D. Weld. "Efficient evaluation of regular path expressions on streaming XML data." University of Washington Tech. Report CSE000502. http://citeseer.ist.psu.edu/article/ivesOOefficient.html

[97] Feng Peng and Sudarshan S. Chawathe, "XSQ: XPath Queries on Streaming Data" , *Proceedings of the 2003 ACM SIGMOD*

[98] Olteanu, D., Meuss, H., Furche, T., Bry, F., "Xpath: Looking forward." In: *Proceedings of Workshop on XML Data Management (XMLDM),* 2002

[99] M Li Lee, B C Chua, W Hsu, K L Tan, "MQX-Scan - Efficient Evaluation of Multiple Queries on Streaming XML Data", School of Computing, National University of Singapore

[100] Andrew Watt, "Xpath Essentials ", *Wiley XML Essential Series*

[101] Dan Olteanu, Tim Furche, François Bry, "The XML Stream Query Processor SPEX (Demonstration") - February 2004.

[102] F. Peng and S.S. Chawathe, "Optimal Buffering for Streaming XPath Evaluation." Technical Report CS-TR-4376 (UMIACS-TR-2002-56). Computer Science Department, University of Maryland. January 2004.

[103] Z. Bar-Yossef, M. F. Fontoura, and V. Josifovski, "On the Memory Requirements of XPath Evaluation over XML Streams", *Symposium on Principles of Database Systems (PODS '2004),* Paris, France, 2004

[104] "SAX - Simple API for XML" http://sax.sourceforge.net

[105] Fred Douglis, M. Frans Kaashoek, "Guest Editors' Introduction: Scalable Internet Services" - *IEEE Internet Computing,* July 2001 pp. 36-37

[106] http://www.akamai.com

[107] http://www.cs.washington.edu/research/networking/ants/

[108] A. Bestavros, M. Crovella, J. Liu, and D. Martin "Distributed Packet Rewriting and its application to Scalable Web Server Architectures," in *Proceedings of ICNP'98: The 6th IEEE International Conference on Network Protocols,* (Austin, TX), October 1998.

[109] T. Brisco, "DNS Support for Load Balancing". IETF RFC-1794. April 1995.

[110] http://gnutella.wego.com

[111] Revised by David H. Crocker, "Standard For The Format Of ARPA Internet Text Messages", IETF RFC 822, Aug 13, 1982

[112] S. Barber, "Common NNTP Extensions" IETF RFC 2980, Oct 2000

[113] R. Fielding et al, Hypertext Transfer Protocol -- HTTP/1.1, IETF RFC 2616, June 1999

[114] B. Quinn et al, "IP Multicast Applications: Challenges and Solutions" IETF RFC 3170, September 2001

[115] B. Fenner, "IANA Considerations for IPv4 Internet Group Management Protocol (IGMP)", IETF RFC 3228, February 2002

[116] D. Waitzman et al, "Distance Vector Multicast Routing Protocol", IETF RFC 1075, November 1988

[117] C.L.Liu, "Elements of Discrete Mathematics", 2nd Edition, McGraw-Hill, ISBN 0-07-038133-X. Pg-147-149

[118] E. Horowitz and S. Sahni, "Fundamentals of Data Structures". Rockville, MD: Computer Science, 1982, chap. 6.

[119] The Extensible Stylesheet Language Family (XSL)http://www.w3.org/Style/XSL/

[ 120] W3C XML Pointer, XML Base and XML Linking http://www.w3.org/XML/Linking

[121] XML Schema http://www.w3.org/XML/Schema

## List of Research Papers

| S.No. | Description |
|---|---|
| 1. | V. Radha and V.P.Gulati, **"Preventing Technology Based Bank Frauds",** published in The CID Review, Journal of Crime Branch, CID. Tamil Nadu, March 2003, Vol III, Issue: 3, pp 31-44 |
| 2. | V. Radha, V. P. Gulati and Arun K. Pujari. **"Appcast - A Low Stress and High Stretch Overlay Protocol",** in Proceedings of the 2nd International Workshop on Grid and Cooperative Computing, 2003, Shanghai, China [Dec. 2003], Vol. II, pp. 967 - 978 |
| 3. | V. Radha,V. P. Gulati and A. K. Pujari. **"Efficient Multicast E-Services over Appcast",** in proceedings of the International Conference on Information Technology • ITCC 2004, [April 05-07, 2004], Las Vegas, USA, TEEE Computer Society |
| 4. | V. Radha, V. P. Gulati and Rajcsh T. **"Evolution of Web Services in SFMS - A Case Study",** proceedings of the IEEE International Conference on Web Services (1CWS; 2004), p 640, [June 6-9, 2004], |
| 5. | V. Radha, V. P. Gulati and A. K. Pujari. **"A Balanced Multicast Overlay Protocol",** in Proceedings of 18th International Conference on Advanced Information Networking and Applications [AINA 2004] at FIT, Fukuoka, Japan |
| 6. | V. Radha , **"Banks as Intermediaries",** in Proceedings of the National Conference on "e-services: Dream to Reality" August 26-28, 2004 at 1 lyderabad |
| 7. | V. Radha and V.P.Gulati, **"Identity Theft - Hindering E-Commerce"** in the Journal of Banking Laws, August 2004, Vol. III, No.3, p 28-40 |
| 8. | V. Radha , V. P. Gulati and Arun K. Pujari. **"Appcast - A Low Stress and High Stretch Overlay Protocol",** accepted for the International Journal of Grid and Utility Computing, Inderscience Publications, UK |
| 9. | V.Radha and A.VedaSree, **"E-Banking",** in Proceedings of the National Conference on E-Commerce, Nov 1-2, 2004, Rourkela |
| 10. | V.Radha and S.RamaKrishna, **"XSI -XML Schema Based Universal User Interface",** accepted in IEEE International Conference on Enabling Technologies for Smart Appliances - ETSA 2005 to be held in Hyderabad |
| 11. | R.V.R.Pavan Kumar and V.Radha, **"YALXP - Yet Another Light Weight XPATH Processor",** accepted in International Conference on Information Technology, CIT 2005 |