DISSERTATION

APPLICATION-AWARE TRANSPORT SERVICES FOR SENSOR-ACTUATOR NETWORKS

Submitted by

Tarun Banka

Department of Electrical and Computer Engineering

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

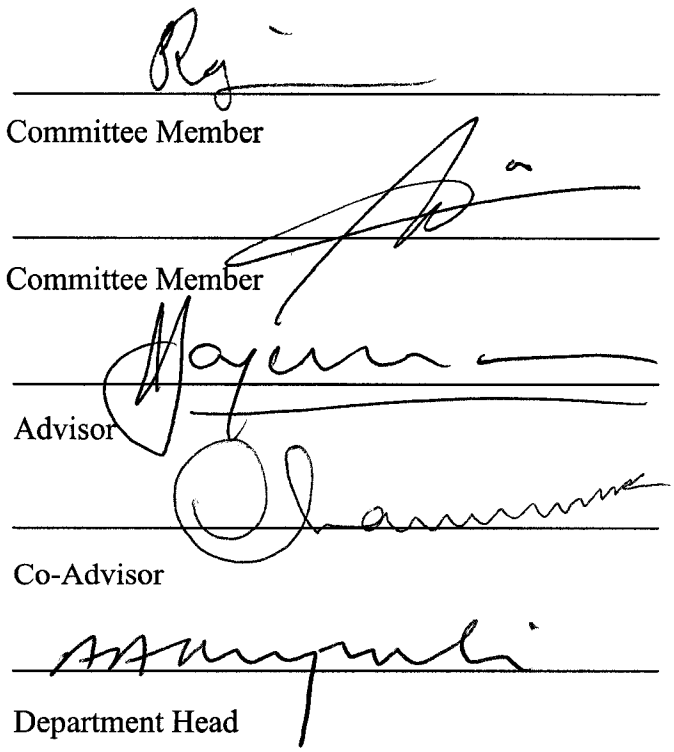
Summer 2007

COLORADO STATE UNIVERSITY

MAY 15, 2007

WE HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER OUR SUPERVISION BY TARUN BANKA ENTITLED APPLICATION-AWARE TRANSPORT SERVICES FOR SENSOR-ACTUATOR NETWORKS BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

Committee on Graduate Work



Committee Member

Committee Member

Advisor

Co-Advisor

Department Head

ABSTRACT OF DISSERTATION

APPLICATION-AWARE TRANSPORT SERVICES FOR SENSOR-ACTUATOR NETWORKS

Many emerging mission-critical sensor actuator network applications rely on the best-effort service provided by the Internet for data dissemination. This dissertation investigates the paradigm of application-aware networking to meet the QoS requirements of the mission-critical applications over best-effort networks that do not provide end-to-end QoS support. An architecture framework is proposed for application-aware data dissemination using overlay networks. Using the proposed architecture framework, an overlay network based application-aware one-to-many high-bandwidth data dissemination application is implemented. The application-aware architecture framework enables application-aware processing at overlay nodes in the best-effort network to meet the QoS requirements of the heterogeneous end users of mission-critical sensor-actuator network applications. Some of the examples of application-aware processing at overlay nodes include application-aware rate adaptation during congestion control, and selective packet forwarding/drops within the network. An application-aware congestion control protocol performs data selection and real-time scheduling of data for transmission while considering different bandwidth and data quality requirements of heterogeneous end users. A packet-marking scheme is proposed that enables application-aware selective

drop and forwarding of packets at intermediate overlay nodes during network congestion to further enhance the QoS received by the end users under dynamic network conditions. Effectiveness of the transport services based on application-aware architecture framework is demonstrated by one-to-many high-bandwidth time-series radar data dissemination protocol for CASA (Collaborative Adaptive Sensing of the Atmosphere) application. Performance analysis is performed using Internet based Planetlab test bed and emulation test bed. Experiment results demonstrate that under similar network conditions and available bandwidth, application-aware processing at overlay nodes significantly improves the quality of the time-series radar data delivered to the end users compared to case when no such application-aware processing is performed. Moreover, it is shown that application-aware congestion control protocol is friendly to the already existing TCP cross-traffic on the network as long as bandwidth requirements of the mission-critical applications are met. Scalability analysis of application-aware congestion control protocol shows that it is able to schedule data at cumulative rates of more than 700Mbps without degrading the QoS received by multiple end users.

Freshness of the data received by end users is an important QoS parameter for mission-critical sensor network applications. A model for tardiness of data is developed for evaluating the impact of network dynamics such as packet losses, random delay, packet reordering caused by random delays or multiple paths selection, and sampling rate on the freshness of the data in sensor networks. Tardiness profiles can be generated using this model for a given sensor network, which are useful for analyzing the suitability of the network infrastructure/configuration for a given mission-critical application. Alternatively, applications may use the tardiness model to adapt network operating

parameters such as transmission power and sampling rate to achieve the application-specific freshness of the data. Tradeoffs between energy consumption and tardiness of the data in a wireless sensor network are also investigated. Tardiness model based result shows that it is significantly more energy efficient to achieve the desired freshness of data by adapting transmission power instead of sampling rate. It is shown that in a multi-hop wireless network there exists an optimal number of relay nodes between source and sink node that leads to minimum tardiness of data. Applications of tardiness model include (i) the estimation of error in the end results due to use of stale data in computations, and (ii) performance analysis of sensor network routing protocols by comparing tardiness of data due to selection of different paths by different routing protocols between source and sink nodes.

Tarun Banka
Department of Electrical and Computer Engineering
Colorado State University
Fort Collins, CO 80523
Summer 2007

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor Prof. Anura P. Jayasumana for all his support and guidance during my research. My sincere thanks to my co-advisor Prof. V. Chandrasekar for being so helpful at all times and providing me with an opportunity to become part of his research group. Special thanks to my committee members Prof. Wim Bohm and Prof. Sanjay Rajopadhye for their valuable suggestions and critiques that motivated me to perform my best in my research. I take this opportunity to thank my research collaborator Prof. Jim Kurose at University of Massachusetts, Amherst for his valuable advice and for showing unstinted confidence in my abilities as a researcher.

This research would not have been possible without the support of many of the peer graduate students with whom I had the privilege to work in last 4 years. My special thanks to Panho Lee for helping me conduct many experiments and in implementation of different proposed algorithms. I would like to thank other peer graduate students Aditya Maroo, Nitin Bharadwaj, and Nischal Piratla for their help and suggestions during the course of my research. I owe a special thanks to Department of Computer Science and ACNS for supporting me financially during the first year as a doctorate student. This work is supported by the Engineering Research Centers Program of the National Science Foundation under NSF award number 0313747.

Last but not the least I thank every member of my family in India who have stood with me and provided me with all their support and encouragement since last 6 years during my life as a graduate student.

DEDICATION

To my grand mother, parents, uncle, aunt, brothers, sister, sister-in-law, and my niece Prisha

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

Sensing is a process of sampling the environment to help us measure, process, interpret and react based on the sensed phenomena. Need for sensing is being felt in many real-world applications for environmental control, structure monitoring, weather monitoring, patient health monitoring, and much more [Ma02, Xu04b]. Associated benefits of near real-time sensing and ability to process and take actions based on the sensed information have led to the emergence of sensor actuator/actor networks [Ak04]. In this dissertation the terms sensor networks and sensor actuator/actor networks are used interchangeably.

Most of the sensor network applications are "highly specialized and highly mission-specific" [Ku06]. These sensor network applications are driven by the end user defined mission-specific goals. For such sensor network applications, sensing, computing and communication infrastructure should be adapted to meet overall goals of the applications [Ku06]. There exists broad spectrum of emerging sensor network applications. At one end of the spectrum are low bandwidth (tens of Kbps) sensor networks that use short-range wireless links for communication. On the other end of spectrum are very high bandwidth (hundreds of Mbps) sensor networks that consist of sensors such as radars and radio telescopes that use both wired and wireless network infrastructure for the communication. Alternatively, there are emerging medium bandwidth (tens of Mbps) sensor networks based on sensors such as cameras that may also use both wired and wireless network infrastructure for communication [Yu04]. Many of these sensor network

applications are specialized and mission-specific. In such applications, QoS requirements of the end users such as bandwidth requirements, acceptable losses, and delay tolerance need to be met under available network resources and dynamic network conditions. CASA (Collaborative Adaptive Sensing of the Atmosphere) [Cas, Chi, Do05, Mc05] is an emerging high-bandwidth sensor network application that requires distribution of mission-critical sensor data from multiple weather radars to multiple end users with distinct QoS requirements over a best-effort network such as Internet. The key challenge is to meet diverse set of QoS requirements such as bandwidth requirements and data quality requirements of multiple end users under severe network resource constraints and dynamic network conditions. Moreover, for real-time sensor network applications it is also desired to measure the quality of data in terms of freshness of data. Freshness of the data at receiver node is one of key QoS parameter for evaluating the quality of service received by end users in real-time sensor networks. A framework is required that relates different network conditions to the perceived freshness of data available to the end users in such systems.

The focus of this dissertation is on adaptive use of communication infrastructure to meet the QoS requirements of the end user defined application's requirements for real-time sensing applications such as CASA. To realize the goal of adaptive communication infrastructure this dissertation proposes application-aware transport services for the sensor actuator networks. An application-aware overlay network based multicast protocol is proposed that performs real-time scheduling of the data for transmission/drop from a weather radar node for multiple end users while meeting both bandwidth and data quality requirements. A content-aware packet marking scheme is proposed that allows in-network processing to be performed at overlay nodes in the network for meeting end users QoS requirements. A token-bucket based rate control algorithm is integrated with the packet marking scheme to select most appropriate subset of the data for forwarding/drop at intermediate overlay multicast node to meet end user QoS bandwidth and data quality requirements. AWON (Application-aWare Overlay Network) overlay node architecture framework is presented for development of such application-aware protocols and services using

overlay networks. Performance analysis of the AWON based application-aware multicast protocol is performed using network emulation test-bed and Planetlab test-bed. Experiment results demonstrate the effectiveness of the application-aware multicast protocols in meeting heterogeneous QoS requirements under dynamic network conditions. Performance results shows that packet-marking based data selection and drop at intermediate overlay delivers better quality data compared to when no application-aware processing is performed at intermediate overlay nodes in the network.

In order to measure the quality of the data delivered to the end users this dissertation proposes a framework for measuring freshness of the data in sensor networks. Tardiness measure capture the impact of different network dynamics such as network delay, packet losses, packet reordering and sampling rate on the freshness of data. The model is validated using simulation results and it is shown that tardiness measure can be used to investigate tradeoffs between freshness of data and energy consumption in sensor networks. We also show that tardiness measure can be used to compare performance of different energy constrained routing protocols based on the freshness of the data delivered to the end users.

Design of application-aware transport services spans two key areas of research (i) Development of application-aware transport protocols and architecture to meet QoS requirements of the applications, and (ii) Framework for the evaluation of QoS delivered to the end user. Section 1.1 describes emerging distributed collaborative adaptive sensing (DCAS) systems and their QoS requirements. Section 1.2 provides the motivation for application-aware protocols and architectures. Section 1.3 describes the need for framework for measurement of the freshness/tardiness of the data as a QoS parameter in DCAS. In Section 1.4 we discuss scope of the dissertation and the objectives. Section 1.5 presents outline of the dissertation.

1.1 Distributed Collaborative Adaptive Sensing (DCAS) Systems

Distributed collaborative adaptive sensing (DCAS) systems [Mc05] are an emerging class of sensor networks that are increasingly used for applications such as weather monitoring, sniper tracking, and distributed target tracking [Ak04, Es02, Ku06, Li02, Si04a]. The QoS requirements, e.g., required bandwidth, latency, acceptable data quality, and reliability are interdependent, and critical to the operation of DCAS systems. Collaborative Adaptive Sensing of the Atmosphere (CASA) [Cas, Mc05] is an example of the DCAS systems. CASA is based on a dense network of low-power X-band radars that operate in a distributed, collaborative and adaptive manner to detect tornadoes and other hazardous atmospheric conditions [Ch01, Mc05, Ku06]. In such systems variety of mission-critical data must be distributed in real time to multiple end users such as emergency managers and researchers at distant and distributed geographical locations. These data streams and end users have differing QoS requirements for the data based on the ultimate use of the data. CASA relies on dedicated network links as well as shared Internet based infrastructure for large-scale dissemination of weather related data. The underlying network infrastructure itself may be affected by such adverse weather conditions, and as such one cannot rely on ISP-provided QoS guarantees or service-level agreements. Network traffic may suffer congestion that may lead to random drop of weather radar data in the network. Under these conditions the partial data delivered to end users may be useless for the application. Some of the possible effects of random drop of information may lead to inability of the emergency managers to make reliable and precise prediction about the hazardous weather events in real-time. CASA application software must thus monitor the underlying network, link availability, link quality, and other performance measures, and then use this information to adapt its operation in real-time to get the best possible service out of the available network facilities. An adaptive networking infrastructure is needed that is cognizant of the requirements of the application [He99, He00].

1.2 Overlay-Based Application-Aware Transport Protocols and Architectures for DCAS

Today's best-effort Internet does not provide end-to-end QoS to the real-time sensor network applications. As mentioned in Section 1.1 there is a need for CASA application to adapt its operation to extract the best QoS performance from the available networking infrastructure and dynamic network conditions. Some examples of the application-aware adaptation include application-aware rate adaptation and data selection for transmission during network congestion. During network congestion packet losses may be controlled by selecting most relevant data for transmission at available bandwidth as per end user's bandwidth and data quality requirements. Moreover, such an adaptation may take place at source nodes or within the network at intermediate nodes on the path between source and the receiver node.

Late 90's saw the emergence of overlay network concept that provides a practical deployment path for new protocols and services over the already existing networks without the need for changing underlying network infrastructure [An01, To02]. Moreover, overlay networks provide a scalable solution for supporting application specific QoS requirements as it is not always efficient to support such requirements for every application at the IP layer. Beside that overlay nodes are special nodes in the overlay network with significantly more resources in terms of computation, memory and storage. It enables complex application-specific operations as well as transport oriented operations to be performed at the overlay node level rather than performing them at the router level in case of Internet. Under normal operating conditions, an ISP can be relied upon to meet end user specific critical QoS requirements using a scheme such as DiffServ [Ni98]. As mentioned before CASA like system often has to operate under adverse conditions, such as severe weather or tornados that can potentially disrupt services provided by some of the links or ISP's. These systems need to be designed to operate even under adverse conditions, adapting to degradation of service in parts of the network. Some of the current applications of overlay

networks include selection of alternate bandwidth rich paths, and ability to adapt to available bandwidth in an application specific manner [An01, Su04]. Thus overlay-network based transport solutions have the potential to meet the application-specific QoS requirements of the DCAS systems. There is also a need to develop an architecture framework for deployment of application-specific services on overlay networks.

1.3 Freshness of the Data in DCAS

In DCAS systems, one of the key QoS requirements is the freshness of the data that is delivered to the end users for real-time actuation and decision making. Input data may be useless for such applications if it arrives and is processed after a critical deadline. There may be application-specific bounds on the desired freshness of the data. Therefore, it is important to be aware of the age of the data that is used for processing and computing results. Freshness of the data thus can be characterized as an important QoS parameter for measuring the quality of the data in sensor actuator network applications. There are different network dynamics that may impact the freshness of the data. For e.g., network delay suffered and losses encountered in the network may impact the age of the data at the receiver node. Different protocols such as transport protocols, routing protocols, and MAC protocols may impact the delay and losses suffered by packets in the network. There is a need to understand precisely how different network dynamics may impact the age of the data aka freshness of the data in sensor-actuator networks. Having models that relate different network parameters to the perceived age of the data help in controlling network dynamics and achieving timing and accuracy goals of applications. Moreover, it helps in the design of application-aware protocols to meet the freshness QoS requirements of sensor actuator network applications and other real-time applications in general.

1.4 Scope of Dissertation and Objectives

The goal of this dissertation is to develop application-aware transport services to meet QoS requirements of mission-critical sensor network applications. This dissertation spans two key areas of research under application-aware transport services for sensor networks. First area of research is focused on the design and development of application-aware transport protocols and an architecture framework for the deployment of application-aware protocols/services in overlay networks. Under this application-aware congestion control protocol is developed that performs rate based congestion control while meeting end user specific bandwidth constraints and selects the most relevant subset of the sensor data for transmission in real-time under available bandwidth conditions. This application-aware congestion control protocol is then extended to serve multiple end users with heterogeneous bandwidth and data quality requirements. A sender-driven application-aware multicast protocol DOOM (Deterministic Overlay One-to-Many) protocol is proposed that uses a time-multiplexed scheduling scheme to dynamically select most appropriate subset of the sensor data for transmission at source node for multiple end users at different available bandwidth [Ba05d, Ba06]. This dissertation then explores the effectiveness of performing application-aware processing at intermediate nodes in the overlay network for enhancing quality of service received by multiple end users in sensor networks. A packet marking scheme is proposed that marks the packet at the source node which enables application-aware data selection for forwarding and drop at intermediate overlay nodes during network congestion. A variant of DOOM protocol is designed that uses packet-marking at intermediate multicast node to perform on-the-fly data selection of data for forwarding or drop for multiple end users while considering available bandwidth for each end user [Le06]. This dissertation then defines a generic node architecture framework AWON (Application-aWare Overlay Networks) for deployment of different application-specific protocols at overlay nodes in the overlay networks [Ba07b]. Effectiveness of the AWON based implementation is

demonstrated for the CASA application for distributing weather radar data to multiple heterogeneous end users with diverse QoS requirements [Ba05a, Ba05b]. Performance of the application-aware protocols is analyzed in the real-world Internet environment using Planetlab test-bed, and additional performance evaluations are performed over an emulation network environment.

Second key contribution of this dissertation is the development of a framework for measuring QoS in terms of freshness of data delivered to end users. A model is presented for the tardiness of data, a measure relating network dynamics to the age of the data delivered to the end users [Ba07a]. The random process corresponding to the age of the data delivered to the end user is modeled as the expectation of the age of the data is derived as a function of mean network delay, probability of random losses perceived by the end user application, and sampling rate at the sensor node. The model is then extended to consider additional losses due to packet reordering for applications that do not accept out-of-order packets. Tardiness model is used to investigate tradeoffs between energy consumption and the tardiness. This dissertation then demonstrates the application of tardiness model for estimating error in the end computation due to tardiness of data. Performance of wireless sensor network routing protocols is compared based on the difference in the tardiness of data due to difference in the characteristics of the paths selected between source and sink nodes by different routing algorithms.

1.5 Dissertation Outline

As mentioned in Section 1.4 the focus of our research is on application-aware transport services for sensor actuator networks. Under that we develop application-aware protocols and architecture and propose a framework for understanding and evaluating the impact of network dynamics on the QoS delivered to the end users in terms of freshness of the data.

Chapter 2 provides background on current state of the art in QoS over Internet and resource constrained sensor networks in general. Different transport layer solutions are compared to understand their effectiveness and limitations for emerging DCAS applications. We then investigate current state of the art in the overlay networks and QoS aware architecture for deployment of applications on overlay networks. We describe current research in the study of data freshness in the context of information systems such as data integration systems (DIS), and Data Warehouses [Bo04].

Chapter 3 defines the problem statement. Two key areas are identified as enabler for providing application-aware transport services for DCAS networks, i.e. Overlay network based application-aware protocols and (ii) Framework for measuring freshness of data for evaluating QoS received by end users in DCAS systems.

Chapter 4 describes a framework for application-aware congestion control protocols. It describes the proposed integration of application-aware, dynamic data-selection scheme with the TRABOL [Bg02, Bg03a, Bg03b, Bg03c] based congestion control protocol. Effectiveness of the application-aware congestion control protocol is demonstrated for the CASA application using results obtained over an emulation testbed.

Chapter 5 proposes the application-aware DOOM (Deterministic Overlay One-to-Many) multicast protocol that performs application-aware congestion control for multiple heterogeneous end users. A time-multiplexed scheduling algorithm is described that selects the most appropriate subset of the data for transmission for multiple end users under dynamic network congestion conditions while considering their data quality and bandwidth QoS requirements. Performance of the DOOM protocol based approach is analyzed using planetlab [Pe02] and an emulation testbed.

Chapter 6 illustrates the use of in-network processing for DCAS systems. An application-aware packet marking scheme is presented, that marks the packets based on the usefulness of the data it contains for the end users considering constraint imposed by the available bandwidth. Effectiveness of the packet marking scheme is demonstrated by performing on-the-fly

application-aware data selection for forwarding/drop during network congestion at intermediate overlay DOOM multicast node in a planetlab testbed.

Chapter 7 presents AWON (Application aWare Overlay Network) architecture for deployment of application-aware services, as explained in Chapter 4-6, at overlay nodes. Effectiveness of the architecture framework is demonstrated by implementing distributed DOOM multicast protocol over a planetlab testbed.

In Chapter 8, a tardiness measure is proposed for understanding the impact of network dynamics on the QoS received by end users in terms of freshness of the data in real-time sensor actuator networks. An analytical model is derived for the tardiness measure relating different network operating parameters such as random network delay, packet losses, packet reordering, and sensor sampling rate to the freshness of data available to end users in sensor actuator networks. Tardiness model is validated using simulation results and tradeoffs between energy consumption and tardiness is also investigated.

Chapter 9 investigates the impact of multi-hop communication on the tardiness of the data in sensor networks. Application of the tardiness measure is demonstrated by comparing performance of different routing protocols for wireless sensor networks. Usefulness of the tardiness measure is demonstrated in estimating errors in the end results due to tardiness of data.

Chapter 10 presents the conclusions of the dissertation.

Chapter 2

QUALITY OF SERVICE IN BROADBAND AND WIRELESS SENSOR NETWORKS

There exists a broad spectrum of sensor network applications. Advances in sensing technologies, power sources, and emergence of high bandwidth wireless links are changing the landscape of sensor networks. These technologies have hastened the arrival of broadband sensor network applications [Cha02]. The broadband sensor network applications have medium to high bandwidth requirements. Each sensor in these networks may generate data at rate in order of tens of Mbps to hundreds of Mbps and these networks are rich in energy and computation resources. Moreover, physical scale of the sensor network is increasing, there are emerging applications where sensors will be deployed in a significantly larger physical area and Internet has the potential to play a critical role in supporting communication for such worldwide sensor networks such as Iristnet, and Sensorweb [Chi05, Gi03]. CASA [Cas, Mc05] in an emerging broadband sensor network application for real-time monitoring of hazardous weather conditions.

On the other end of spectrum, sensor networks consist of power constrained, low bandwidth sensing nodes with short range wireless communication links between them. Example of such wireless sensor network applications include environment monitoring, structural monitoring, and target tracking systems [Ak04, Es02, Ku06, Li02]. Most of these sensor networks are required to operate for years without the need for maintenance or changing power sources. Thus, majority of

the research in resource constrained sensor networks is motivated by the need for energy efficient hardware and software design solutions.

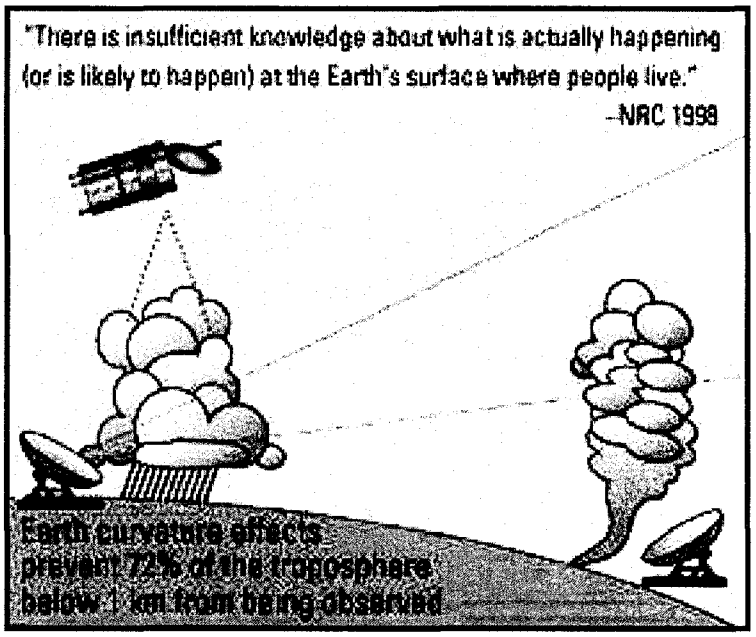
Many of these emerging broadband and wireless sensor network applications are mission-critical and have certain QoS requirements such as bandwidth requirement and latency requirements of end users that need to be met for their proper operation. There is very limited research that has focused on providing QoS support in sensor networks. A wide spectrum of sensor network applications offers significant challenges for providing generic solutions for meeting QoS requirements in such systems. This chapter highlights QoS requirements in a broadband sensor networks such as CASA and provides motivation for the application-aware transport services for broadband sensor networks. In mission-critical sensor networks it is also important to measure the QoS delivered to the end users to understand the effectiveness of the networking infrastructure in meeting requirements of the application. Freshness of data is one such key QoS parameter that is crucial for the success of real-time mission-critical sensor networks operation. This chapter investigates the current state of the art in measuring freshness of the data in the context of information systems and data warehouses.

Section 2.1 describes the CASA broadband sensor network application. Section 2.2 describes QoS requirements of broadband sensor networks. Section 2.3 shows current state of the art in QoS support for wired broadband sensor networks. Section 2.4 compares different transport protocols for high-bandwidth wired networks. Section 2.5 describes QoS requirements of resource constrained wireless sensor networks. Section 2.6 describes current state of the art in QoS support for wireless sensor networks. Section 2.7 provides motivation for need for overlay network based transport services for meeting QoS requirements of broadband sensor networks. Section 2.8 describes the prior work on need for understanding freshness of data in the context of information systems and data warehouses. Concluding remarks are presented in Section 2.9.

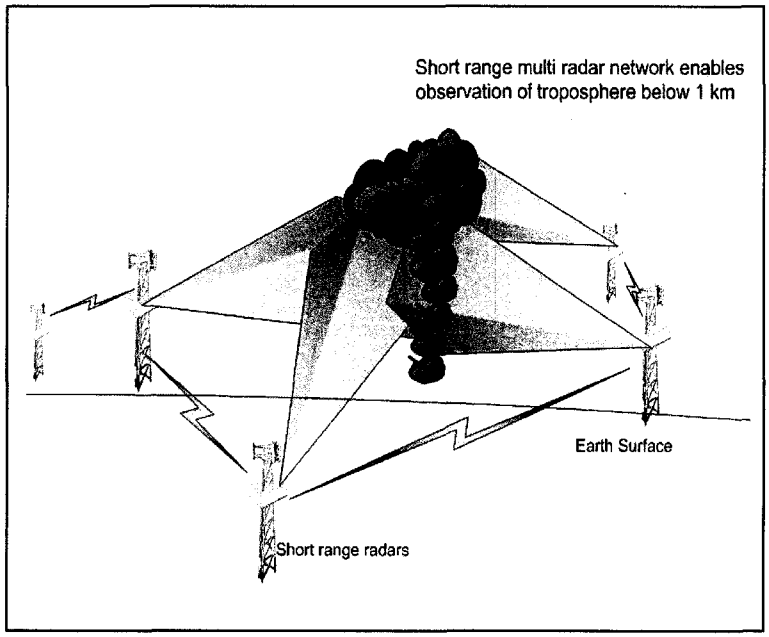
2.1 Collaborative Adaptive Sensing of the Atmosphere (CASA)

The vision of the CASA is to revolutionize our ability to observe lower troposphere through Distributed Collaborative Adaptive Sensing (DCAS), vastly improving our ability to detect, understand, and predict severe storms, tornados, floods, and other atmospheric and airborne hazards [Cas, Mc05]. Fig. 2.1(a) shows the limitations of current state of the art for observing atmospheric phenomena using long-range autonomous radars. Current technology in weather monitoring is unable to monitor the lower troposphere due to earth curvature limitations. In CASA, a network of short range radars is used instead to sample the previously unobserved region of the atmosphere as shown in Fig. 2.1(b). CASA network forms a tightly coupled network of radar and processing nodes. Some of the key features of CASA network include presence of heterogeneous network infrastructure such as mix of both wired and wireless links, both resource rich and resource constrained sensor nodes.

Historically radars have been designed and operated in a "central unit" environment where the radar transmitter/receiver and information processing were all carried out at the radar node. Due to advances in high-speed networking, there is no need to do computation at the radar node itself any more. Moreover, multiple smaller, cheaper radars can be networked to sample the atmosphere in an efficient manner and can be deployed over rooftops or cell towers as shown in Fig. 2.1(b). A network of small radars provide more flexibility by enabling re-tasking of the radar for effective sampling of the atmosphere based on the existing atmospheric conditions. Moreover, different radars can now operate in different scanning modes/bands unlike static operations of autonomous long-range radars. This system is a sensor-actuator network in that the radars sense the atmosphere, yet the scanning strategies of radars are controlled dynamically in real-time depending on the features being sensed and the requirements of the end-users. Depending on the radar operating parameters, it can generate data at rates of tens of Mbps to hundreds of Mbps.



(a)



(b)

Figure 2.1 Need for CASA based monitoring system (a) Limitation of current state of the art in observing atmospheric phenomena [Cas] (b) Network of short-range radars for observing lower troposphere

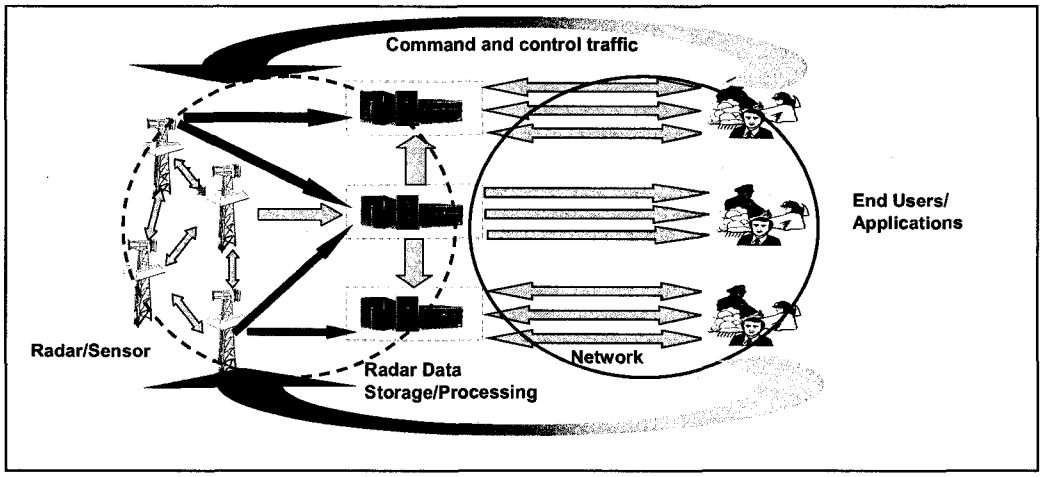


Figure 2.2 Different data transfer scenarios in a CASA network

Under certain scenarios it is required to transmit high bandwidth data to the remote end users. In certain cases, this high bandwidth data can be processed at the radar node and only low bandwidth processed data transmitted over the network.

2.2 QoS Requirements in CASA Broadband Sensor Networks

In a CASA network shown in Fig. 2.2, there are heterogeneous QoS requirements that need to be satisfied by transport services for the proper operation of the system [Ba05a]. The solid red and dotted blue circles in Fig. 2.2 show different regions of the CASA network with different QoS requirements. The blue dotted circle highlights the part of the network that supports communication between radar nodes and the distributed processing/storage nodes. Depending on the resources available at the radar node, data can be locally processed or transmitted over the network (which may consist of both wired and wireless links) in real time for remote processing. In this case sustained data rates may be in order of tens of Mbps to hundreds of Mbps. At the same time, low bandwidth streams like command and control signals, monitoring probes, and

radar health traffic streams share the bottleneck links with high bandwidth radar data streams. Thus the transport protocols need to be friendly to the cross traffic streams in the network. Moreover it is necessary to support one-to-many, many-to-one and many-to-many data transfer scenarios for both high and low bandwidth data. For example, many-to-one communication will be required to gather data from multiple radars for integration at a central server. Radar data streaming from the radar nodes to the points of computation also has unique QoS requirements: a minimum acceptable rate based on the end application (CASA needs to support multiple applications with different requirements simultaneously), better accuracy of end results with higher bandwidth, a bound on delivery time beyond which data is not useful, a bound on bursty losses, and more.

The solid red circle in Fig. 2.2 highlights the network that provides communication between processing/storage nodes and the end users. In CASA one-to-many communication may be required to deliver high-bandwidth radar data to multiple heterogeneous users such as emergency managers and researchers in real-time. Each of these end users may have different bandwidth requirement and latency requirements. Many of these end users may tolerate certain types of losses in the transmitted data based on the desired accuracy in the end results. Each end user may thus specify heterogeneous real-time and data framing requirements for acceptable data accuracy due to variable resources like bandwidth and computation resources availability at receiver end. End users may send request for real-time processed or unprocessed data; similarly non real-time access of archived data may be requested. In certain cases, depending on the mode of operation and end user requests, reliability of data may be important. It is imperative to meet diverse set of QoS requirements of CASA like applications to achieve the goals of the system. There are different QoS solutions that are required for broadband and wireless sensor networks. Next sections investigate current state of the art in meeting QoS requirements for broadband and wireless sensor networks.

2.3 QoS Support for Broadband Senor Networks

This section investigates the current state of the art in QoS support for wired networks that may be used for providing QoS in broadband sensor networks. There are multiple proposed solutions to ensure QoS in wired networks for instance, over provisioning of computation and networking resources helps application meet its QoS requirement [Ni02]. The limitation of this approach is that all end users of an application receive similar type of QoS, for example available bandwidth is equally shared among all users. This may not be acceptable for certain applications where multiple end users have heterogeneous end user requirements which are typical case for emerging mission-critical broadband sensor network applications. Network traffic may vary and as a result QoS received by end users may degrade during peak traffic conditions because of increase in jitter. Moreover, over-provisioning may not be always cost effective when number of end users increases significantly. Alternative approaches are resource reservation and reservation-less based approach. As the name suggests, in resource reservation based approach end-to-end resources in terms of computation and bandwidth are reserved for a particular application. In this case application uses Resource Reservation Protocol (RSVP) [Br97] to request and reserve networking resources. IntServ [Br94, Ma99, Sh05] based model uses the resource reservation based approach to deliver desired QoS to the end users. However, this requires intermediate routers/switches to manage state of the sessions that requires resource reservation. Like over-provisioning IntServ approach also suffers from scalability limitations. In the case of reservation-less approach, as the name suggests no resource reservation is performed for the application in the network. Instead, mechanisms such as traffic classes, traffic shaping, queuing mechanism, admission control, and policy managers are used [Ni02]. Diffserv [Bl98, Sh05] model uses reservation-less based approach to provide the desired QoS to the end users. In DiffServ model packets are marked according to type of the service they need. This marking is then used at the DiffServ enabled routers/switches to select appropriate queuing mechanism to deliver the desired QoS performance

indicated by the packet marking. Note that desired QoS service is achieved when traffic confirms to the service level agreement (SLA) agreed between end users and the service providers.

Due to lack of end-to-end deployment of above mentioned QoS architecture, there are no guarantees to meet QoS requirements of applications like CASA on a best-effort Internet infrastructure. This dissertation explores effectiveness of application-aware transport services in meeting QoS requirements of broadband sensor networks over a best-effort network. One of key component of the application-aware transport services includes application-aware congestion control protocol for meeting bandwidth and data quality requirements of the end users. Next section compares different transport protocols available for the high-bandwidth wired networks.

2.4 Transport Protocols for High-Bandwidth Wired Networks

Emergence of broadband sensor network application is changing the paradigm of protocols that might be used for the sensor network applications. Most of the high speed transport protocol research has been done while considering Internet as the communication medium. This section surveys of high-speed transport protocols for the wired networks such as Internet.

TCP is the dominating reliable transport protocol over the Internet that has been extremely successful in keeping the current Internet working quite efficiently. However, TCP is shown to be significantly inefficient over network with high bandwidth-delay product. As new high bandwidth applications are emerging and network speeds are increasing, it offers significant challenge for the TCP in future to meet needs of the evolving Internet. This has led to the research and development of transport protocols for high speed networks that include High speed TCP[Fl04], Scalable TCP[Ke03], Fast TCP[Ji04], H-TCP[Sh04], BIC-TCP[Xu04], and TRABOL [Bg02, Bg03a, Bg03b]. Except TRABOL, all other above mentioned protocols are reliable transport protocols. All these reliable transport protocols are variants of TCP protocol as they have their

own window based distinct congestion control mechanisms. One of the key requirements for the deployment of these emerging protocols on the Internet is that they should offer tangible benefits without degrading the performance of already existing protocols on the network.

Different congestion control algorithms have different ways for adjusting the congestion windows [Ch89]. In case of Scalable-TCP, basic idea is to make the recovery time independent of the window size. Scalable-TCP uses AIMD based congestion control and it updates the current TCP *cwnd* sending window as follows

$$\begin{aligned} &ACK : cwnd \leftarrow cwnd + \alpha \\ &LOSS : cwnd \leftarrow cwnd \times \beta \end{aligned} \tag{2.1}$$

where α and β are 0.01 and 0.875, respectively.

High Speed TCP congestion control is also based on AIMD based congestion control mechanism and unlike Scalable-TCP, it uses current *cwnd* as an indication of the bandwidth-delay product on the path. AIMD increase and decrease functions are varied as follows.

$$\begin{aligned} &ACK : cwnd \leftarrow cwnd + \frac{f_\alpha(cwnd)}{cwnd} \\ &LOSS : cwnd \leftarrow cwnd \times g_\beta(cwnd) \end{aligned} \tag{2.2}$$

$f_\alpha(cwnd)$ and $g_\beta(cwnd)$ are the logarithmic functions such that $f_\alpha(cwnd)$ increases and $g_\beta(cwnd)$ decreases with increase in *cwnd*.

H-TCP protocol determines the path bandwidth-delay product based on the elapsed time Δ since the last congestion event rather than *cwnd* as done with Scalable-TCP and HS-TCP.

$$\begin{aligned} &ACK : cwnd \leftarrow cwnd + \frac{2(1-\beta) f_\alpha(\Delta)}{cwnd} \\ &LOSS : cwnd \leftarrow cwnd \times g_\beta(B) \end{aligned} \tag{2.3}$$

where, $f_\alpha(\Delta)$ and $g_\beta(B)$ are defined in [Sh04]. Similarly, FAST TCP has a different congestion control algorithm for the TCP. It is equation based algorithm that eliminates packet-level oscillations. Unlike other congestion measure schemes based on losses or buffer occupancy, it

uses queuing delay as a measure of congestion. It achieves proportional fairness and does not penalize flows with larger RTT.

We have seen different TCP variants for high speed networks, which ensure reliable transfer of data over high bandwidth-delay product networks. Alternatively, there are many applications that do not require reliable transfer of data and can tolerate certain types of losses. These protocols include Rate Adaptation Protocol (RAP) [Re99] and TCP Friendly Rate Adaptation Based On Losses (TRABOL). RAP protocol uses AIMD based increase/decrease algorithm for rate adaptation. In the absence of packet loss transmission rate is increased periodically in a step-like fashion, Inter-packet gap (IPG) is used to adapt the transmission rate. RAP adjusts the transmission rate every estimated RTT. This approach of fine tuning the sending rate every RTT can make RAP more aggressive for flows with shorter round trip time, resulting in unfairness among the TCP and RAP flows. In case packet loss is detected, RAP decreases its rate by increasing the IPG, and it can safely ignore reacting to packet losses if they all are part of same burst loss.

TRABOL is a UDP based congestion control protocol. TRABOL has been designed for broadband sensor networks where data is generated at very high rates and is required to be transmitted to the loss tolerant end users with critical minimum rate and target rate QoS requirements. Key feature of TRABOL protocol is that on congestion detection, transmission rate is decreased in one step to the end user specific minimum rate requirement. This is an important feature of TRABOL, as none of the above mentioned high speed transport protocols consider such application specific minimum and target rate requirements. In many mission critical broadband sensor network applications, different end users can have different minimum rate requirements for their proper operation. Thus TRABOL is a suitable protocol for such sensor network applications. In TRABOL, packet loss is used as an indication of presence of congestion on the network. In absence of packet loss, transmission rate is increased additively and it does not exceed end user specific target rate requirement [Tr06]. Rate adaptation is based on the

knowledge that end users are able to tolerate certain types of losses. This protocol is shown to be TCP friendly as long as minimum rate requirements of end users are met. Alternatively, TFRC (TCP-Friendly Rate Control) [Ha03] is an application layer congestion control mechanism that has smoother rate variation compared to TCP. TFRC is suitable for real-time applications such as voice and video that cannot tolerate high degree of variation in the transmission rate during network congestion. However, like TCP, TFRC does not guarantee that transmission rate do not fall below the minimum rate requirements of the end users.

2.5 QoS Requirements in Wireless Sensor Networks (WSN)

QoS mechanisms that are used for the wired networks cannot be directly applied to wireless sensor networks because of bandwidth constraints and dynamic network topology. There are different QoS solutions available for wireless ad-hoc networks. Support for QoS in ad-hoc network includes QoS signaling for resource reservation, QoS routing, and QoS MAC. For e.g., QoS routing can be used in ad-hoc network that searches for path with enough resources to meet application needs. Once paths with sufficient resources are found, QoS signaling is used to reserve those resources.

However, QoS solutions for wireless ad-hoc network do not consider energy efficiency as their key goal which is critical to the operation of resource constrained wireless sensor networks. Some of the key challenges for wireless sensor networks are as follows [Che04].

1. Resource constraints in WSN: It involves bandwidth, memory, computation constraints. This requires that QoS support mechanism in wireless sensor network should be simple and less resource consuming.

2. Unbalance traffic: In most wireless sensor networks, traffic is mainly from large number of sensor to small finite number of sink nodes. Thus QoS mechanism should be considerate of such scenarios in sensor networks.
3. Spatiotemporal dependency of data: There can be redundancy in the data from a single sensor or a group of sensors. This can lead to unnecessary wastage of sensor resources thus require QoS support mechanism to be cognizant of such cases.
4. Network dynamics: Links can fail or degrade thus QoS support can become complex in such an environment.
5. Scalability: Sensor network can grow from tens of nodes to thousands of nodes, thus QoS support for sensor network should also scale with parameters like number of nodes, density of nodes.
6. Multiple end users: There are heterogeneous end users with different bandwidth, latency, data quality requirements. It is required to concurrently meet QoS requirements of multiple end users for mission critical sensor networks.
7. Multiple priorities: Multiple traffic streams with different priority can traverse the sensor network for e.g., in CASA different traffic streams are raw time-series data, health monitoring signals, command and control signals. QoS support should be considerate of such requirements.

2.6 Protocols for QoS Support for Wireless Sensor Networks

Different solutions have been proposed to meet QoS requirements in wireless sensor networks. Sequential Assignment Routing (SAR) [So00] is one of the earliest protocols for wireless sensor network that considers QoS while routing packets. When there are multiple paths towards the sink node, SAR considers energy resources, QoS availability, and priority of packets for selecting

path for the packets towards the sink node. SPEED [He03] is a stateless QoS routing protocol that provides soft real-time guarantees to the end users. It achieves the soft real-time bound by combination of feedback control and non-deterministic geographical forwarding. SPEED aims to provide uniform delivery speed between source-sink pairs such that end-to-end delay is proportional to the distance between sources and sink nodes. Alternatively different protocols have been proposed that provides end-to-end reliability in wireless sensor networks [Bh01, De03a, De03b]. In service differentiation based approach for sensor networks [Bh01] priority is assigned to each packet based on the content and per-hop-behavior (PHB) is determined based on the packet marking to meet reliability and latency requirement of the application.

A sensor network may under go varying degree of network congestion due to sudden occurrence of events in the network. This has the potential to degrade the QoS received by end users in the wireless sensor networks. Need for congestion control in wireless sensor networks is emphasized in [Ti02]. It was shown that exceeding network capacity can be detrimental to the observed goodput and thus can degrade the performance of the end application. It is thus imperative to deal with the congestion in sensor networks. Many sensor network applications cannot tolerate loss of data, e.g., command and control information or re-tasking of the sensors operations. It is thus required for transport protocols to ensure reliability of the data for certain applications under high loss conditions due to wireless link errors or congestions.

One of the earliest solutions that address the congestion problem in wireless sensor network is CODA [Wa03]. It is an energy efficient congestion avoidance and detection algorithm for effectively dealing with both transient and persistent congestion scenarios. For congestion detection, CODA uses a combination of past and present channel loading conditions, and the current buffer occupancy. Once the congestion is detected, actions are taken to recover from the congestion using backpressure mechanism. Open-loop backpressure, a protocol to recover from transient congestion includes hop-by-hop propagation of slow-down signal to upstream nodes until congestion is detected on the corresponding downstream node. For persistent congestion

control, CODA asserts congestion control over multiple sources from a single sink using closed-loop multi-source rate regulation. In this case, source expects to receive ACKs at slow rate from the sink, thus sink can control the transmission rate of the source by adjusting ACKs rate.

SenTCP [Wa05] is an open-loop hop-by-hop congestion control protocol for WSN, in addition to buffer occupancy like CODA, it uses inter arrival packet gap and service time for congestion detection. This approach helps in effectively differentiating between losses due to congestion or link errors. Both CODA and SenTCP reduces source traffic during network congestion. Alternatively, recent novel approach recommends dynamically adapting resource allocation to alleviate network congestion [Ka04] instead of dynamic traffic rate control. Basic idea of this algorithm is based on the fact that in most sensor networks under dormant conditions, a large number of nodes remain in the sleep state for energy conservation. As soon as the congestion is detected, these nodes are transitioned to active state and are made part of alternate paths known as path multiplexing to the sink node. Congested node later distributes the extra traffic over these alternate paths to alleviate the congestion. Alternate paths can result in increase in energy consumption, so it is recommended to switch off the nodes on the alternate paths as soon as congestion is alleviated. Advantage of this kind of resource provisioning is that it is able to maintain the required throughput at the application under network congestion. It is also shown that when congestion is transient, increasing resources by creating multiple alternate paths around the hotspots effectively increases the packet delivery as well as consume less energy by avoiding collisions and retransmissions.

In some wireless sensor network applications, it is required to reliably deliver data from source to sink or from sink to source. Pump Slowly, Fetch Quickly (PSFQ) [Wa02] is one of the earliest reliable transport protocol proposed to meet some of these goals. Design of PSFQ require the sender to transmit the data at slow speed ("Pump Slowly"), but in case of loss of data nodes are allowed to fetch any missing data from its neighboring nodes very aggressively ("Fetch Quickly"). The motivation behind this simple approach is to achieve loose delay bounds while

minimizing the recovery cost by opting for localized recovery due to any lost segment. One of the drawbacks of the PSFQ is that it assumes loss occurrence is because of transmission error due to poor quality of the wireless links rather than the network congestion. This limits its applicability to sensor network that generate light traffic.

Event to Sink Reliable Transport (ESRT) [Sa03] is a transport protocol with a dual goal of providing reliability and congestion control from source to sink. Key difference between PSFQ and ESRT is that beside congestion control ESRT proposes notion of event to sink reliability, and provides solution for achieving end application specific desired reliability with minimum energy expenditure. However, PSFQ guarantee end-to-end reliability by recovering from data loss at intermediate node and is focused on sink to source reliability instead. ESRT is aimed at reliably delivering occurrence of event information in some region of the network to the sink nodes. In many sensor network applications, individual sensors data are correlated and thus can tolerate certain types of losses and delivers event information to the sinks with subset of data. ESRT does not guarantee end-to-end delivery of each event data due to spatiotemporal dependency of the data in the sensor network. ESRT goal is to achieve the optimal event reporting rate of the source node so that the required event detection reliability R is met at the sink node with minimum resource utilization. For congestion control, ESRT monitors the local buffer of the sensor nodes and in case buffer overflows, sets the congestion bit in the packets forwarded to the sink node. When sink node detects that congestion bit is set, then it broadcast a slow down signal with high energy to the sources to throttle down their sending rate. Disadvantage of this approach is that any on-going transmission would be disrupted by high power signal. Moreover, all sources will be forced to reduce their rate irrespective of the source of the congestion. As described before, CODA approach of selective throttling of the sources of the congestion mitigates the problem of ESRT congestion control; without the need for transmission of high energy slow down signal.

2.7 QoS Support Using Overlay Networks

This section focuses on the current state of the art in the overlay networks and their suitability in meeting QoS requirements of emerging mission-critical sensor network applications. As mentioned in Section 2.3 there have been ongoing efforts for providing QoS on Internet using IntServ and DiffServ models. However, due to lack of end-to-end deployment of IntServ and DiffServ models Internet still provides best-effort services to the applications. There are two key alternative schools of thought in the networking community that support two different mechanisms for enhancing QoS support in the Internet.

In the first school of thought researchers have proposed using overlay network based solutions for deploying new protocols and application-specific functionalities to enhance the QoS support available on the best-effort Internet. This does not require change in the underlying IP infrastructure and provides application-developers enhanced ability to adapt their operation in an application-friendly manner under varying network conditions.

Overlay networks [Pe02] enable applications to have more control over the routing decision thus helps in selecting paths that would meet application specific constraints (delay, bandwidth, etc.). The Internet itself began as an overlay network above the traditional telephone network, using the long-distance telephone links to connect remote routers. Emerging overlay networks are similarly using the existing Internet to route data between the overlay nodes. Overlay nodes allows implementing application specific functionality and thus overlay networks have the potential to hasten the deployment of new applications without waiting for years for the underlying routers to change. Some of the well known examples of overlay networks include Resilient Overlay Networks (RON), QoS-Aware Routing in Overlay Networks (QRON), Application-layer multicast, and Content distribution networks [Aka, An01, Ch00, Zh04a]. RON uses overlay networks to improve the failure resilience of the Internet. The RON nodes monitors the functioning and quality of internet paths among themselves, and under path outage or

degradation it can decide to route packets through other RON nodes instead of default Internet paths. This protocol architecture is shown to avoid 50% of the Internet outages by this approach. QRON tries to meet QoS requirements of the application by selecting alternate paths when ever performance of existing path degrades. Alternatively, it distributes the overlay traffic among the overlay broker nodes such that application should not suffer significantly due to varying cross-traffic. Overlay Multicast [Ch00, Ke02, Fa03] is the one of the promising applications for distributing content to multiple end users over a wide-area. End system multicast [Ch00] successfully demonstrated that overlay multicast solution can be practical solution and is easy to deploy compared to IP multicast. Overlay networks based packet recovery at overlay nodes during network losses and rapid rerouting during link failure is shown to be effective in delivering VoIP quality in par with PSTN networks [Am06].

Recently different overlay architectures have been proposed with a goal of deployment of application-aware services to enhance the QoS received by the end users. OverQoS [Su04], an overlay-based architecture can provide a variety of QoS-enhancing in-network services in the intermediate nodes of overlay networks, such as eliminating the loss bursts, prioritizing packets within a flow, and statistical bandwidth and loss guarantees. Our current work on AWON [Ba07b] architecture is motivated by the same vision of enhancing QoS support within the network without the support from IP routers. An important difference between the AWON and the OverQoS architectures is that in the AWON-based approach, quality of service provided to an application is enhanced by performing application-aware processing within the network. Moreover, the AWON architecture is highly flexible and can accommodate QoS requirements of large class of applications. OCALA [Jo06] and Oasis [Ma06] enable the users of legacy applications to leverage overlay functionality without any modifications to their applications and operating systems. Opus [Br02b], which is motivated by active networking, provides a large-scale common overlay platform and the necessary abstractions to service multiple distributed applications. In contrast to our work, Opus focuses on the wide-area issues associated with

simultaneously deploying and allocating resources for competing applications in a large-scale overlay networks. XPORT [Pa06] is a tree-based overlay network, which can create dissemination trees based on diverse performance requirements of the applications.

Second school of thought in the research community recommends clean slate solution to enhance the functionality of the Internet. The motivation behind this approach is that incremental solutions build upon the existing Internet network infrastructure such as overlay networks may not be able to solve the fundamental limitations such as QoS and security of the current Internet architecture. Mid 1990s saw emergence of active networks that focused on solving the problem of difficult and lengthy process of introducing new protocols and services in the network using the concept of programmable network nodes. Active networks allows switches/routers to perform computation on the end user data that is being routed through them as well as it gives the ability to the end users to tailor the operations of these intermediate nodes to perform sophisticated application-specific operations such as customized fusion algorithm, or data compression [Sc99, Te96, Te97, We98]. In [Ke00] it is shown that active network based application-aware processing such as video content scaling during network congestion has the potential to significantly enhance the quality of the video delivered to multiple end users. It is shown that active network enabled routers can be configured to select most appropriate subset of video packets for forwarding on-the-fly based on tag information. It helps in delivering acceptable video quality under severe network congestion conditions. More recently under clean slate based approach, GENI (Global Environment for Network Innovation) [Gen] has been started with a hope that it will facilitate validation and deployment of new protocols/services on the next generation secure network to meet the needs of the 21st century applications. One of the key goals of the GENI is to develop a world-wide secure and trustworthy network environment bottoms-up for current and future applications.

For emerging applications such as CASA, overlay network based solutions provide an immediate practical deployment path without having to wait for deployment of clean-slate based

solutions. The focus of this dissertation is on developing overlay network based application-aware transport services to provide enhanced QoS to the end users in DCAS [Cas, Mc05] systems such as CASA.

2.8 Freshness of Data as a QoS Performance Parameter

Freshness of data is considered as an important data quality parameter for many real-time systems and information systems [Or98, Bo04]. Consider an example where multiple copies of the WebPages may be maintained at different distributed remote locations. It is important for such applications to keep the most updated recent copy of the webpage at all different locations. To meet this requirement effective page refresh policies for web crawlers have been proposed to achieve the freshness requirements of such applications. In such systems data is considered fresh when local copy of the data is same as the remote sources [Cho03]. However, there are different definitions of the freshness that have been proposed depending on the applications where data is used. Traditional definition of freshness of data is known as *currency* [Bo04] in information systems and it describes how stale the local copy of the data is with respect to remote. An alternate definition of freshness also considers timeliness of the data as a data quality parameter, which captures age of the data available at the user node. This aspect of alternate dimensions of freshness of data has been studied in greater detail in [Bo04]. There are different factors that may impact the freshness of data such as rate of change of data, network delays, and synchronization policies. In the context of mission-critical sensor network applications, it is imperative to be aware of freshness of the data received at a sink node. However, there exists minimal focus on understanding factors that may impact the freshness of data in sensor networks. In the context of information systems, there are recent efforts on developing framework for understanding different factors that may impact the freshness of the data [Cho03, Bo04]. In [Ch03] analytical model is

proposed that relates the rate of change of data at source nodes and synchronization policies to the freshness and age of the data. There is a need for similar framework to understand the freshness of data in the context of sensor networks. However, in the context of sensor networks there are multitude of network dynamics that play an important role in determining the freshness or age of the data available at the sink node. There is a need to understand the complex interplay between different network dynamics such as random network delay, network losses, packet reordering, and sampling rate on the age of the data. In this dissertation we use freshness and age of the data interchangeably in the context of sensor networks. This framework has the potential to help application-developer to control certain network parameters to best meet the application-specific data freshness requirements. Moreover, framework for freshness of data for sensor networks can be used to evaluate the performance of application-aware transport services in meeting end user data freshness requirements

2.9 Remarks

This chapter describes the current state of the art in QoS support available in broadband and wireless sensor networks. There are alternate school of thoughts for improving the performance of best-effort Internet based on Incremental approach and clean slate approach. For emerging broadband sensing applications like CASA we make an argument in favor of suitability of overlay network based approach for developing application-aware transport services for such systems. We then discuss current state of the art in overlay networks and transport protocols for both wired and wireless environment.

Freshness of data is considered as an important performance parameters for real-time sensor networks. We discuss the existing work in the field of information systems on understanding the freshness of data. There is a need to develop a framework to understand the freshness of data in

sensor network that relates different network parameters to the freshness of the data at the sink node.

Chapter 3

PROBLEM STATEMENT

Mission-critical broadband sensor network applications such as CASA requires effective transport services for meeting heterogeneous QoS requirements of multiple end users in such systems. QoS requirement of different end users in broadband sensor networks include bounded end-to-end delay, bandwidth requirement, and acceptable loss threshold which are critical to their operation. Moreover, these QoS requirements may vary from one user to another in such systems. Heterogeneity of broadband sensor networks due to presence of both wired and wireless links, sensor nodes with data generation rates from tens of Mbps to hundred of Mbps offer significant challenges for providing effective transport services to multiple heterogeneous end users. In broadband sensor network CASA multiple radar data streams with different QoS requirements may concurrently share the common bottleneck links in the network, e.g., raw time-series radar data, cross-traffic, sensor health status signals, and command and control signals can share the same bottleneck link. It is desired that transport services be cognizant of multiple different streams traversing the networks and should adapt to meet heterogeneous QoS requirements for different streams under dynamic network conditions while remaining friendly to cross-traffic streams. As mentioned in Chapter 2, Internet may play a critical role as a communication infrastructure for broadband sensor network applications. Current Internet operates on the principles of best effort service with no end-to-end QoS guarantee. For mission-critical

broadband sensor network applications, best-effort shared networks like Internet may not always meet the end user QoS requirements. However, due to ubiquitous nature of the Internet and its global reach, it is imperative to leverage this infrastructure for deployment of large scale broadband sensor network applications. As mentioned in Chapter 2, overlay network enables development and deployment of new protocols and services on the Internet without the need for changing underlying network infrastructure. This dissertation proposes and demonstrates effectiveness of overlay network based transport services in meeting QoS requirement of the CASA like sensor networks using Internet. One of key challenge for overlay network based transport services is to steer their operations in an application-aware manner based on the existing network conditions and the feedback received from the end user about the received data. Moreover it is necessary that application-aware adaptation performed by transport services for meeting QoS requirements of broadband sensor network applications should not degrade the performance of other applications sharing the network.

3.1 Research Goals

The goal of this research is to design and demonstrate effectiveness of application aware transport services in meeting the end user QoS requirements for the broadband sensor networks. Following are the key requirements that transport services should meet:

- Adapt to prevailing network traffic conditions and steer protocols and sensor operations to best meet QoS requirements of the application.
- Meet end user requirements for both low bandwidth and very high bandwidth over wired and wireless network infrastructure while efficiently using server and network resources.

- Concurrently meet distinct time constraints of multiple end users for their real-time operations.
- Concurrently meet heterogeneous data quality QoS requirements of multiple end users.
- Scale with parameters such as the number of sensing nodes, number of end users, and bandwidth requirements in the sensor network.
- Measure QoS received by end users for application-aware operations.

3.2 Research Objectives

Application awareness is at the heart of design of proposed transport services. For application aware transport services, research objective will traverse two key areas application aware Transport protocols and Framework for evaluating QoS received by end users. This section further identifies key objectives that need to be accomplished by application aware transport protocols and framework for QoS evaluation in order to realize the goals of application-aware transport services.

3.2.1 Application-aware Transport Services

Application awareness is at the heart of design of proposed transport services. For application aware transport services, research objective will traverse two key areas (i) Design of application-aware transport protocols, and (ii) Framework for measuring QoS received by end users. In this section, we further identify key objectives that need to be accomplished to realize the goals of application aware transport services. Following section lists individual objectives for the application-aware transport protocol, and framework for measuring QoS received by end users.

3.2.1.1 Objectives for Application-aware Transport Protocol

Application-aware transport protocols form the first pillar of transport service design. Network may suffer congestion because of either high bandwidth requirements of many end users or when burst of traffic is generated in the sensor network due to sudden event occurrence. It is desired that transport protocol adapt to the congestion in an application-aware manner while remaining friendly to other cross-traffic streams on the shared links. In this research, out objective is to design, develop, and demonstrate overlay network based application-aware transport protocols that performs application-aware congestion control for multiple end users in real-time.

(i) **One-to-many Data Dissemination:** This type of data transfer support is required in sensor network when multiple end users are interested in receiving data from a single sensor. In one-to-many data transfer scenario, transport protocol should meet heterogeneous real-time as well as distinct data quality needs of multiple end users. It should support both high bandwidth and low bandwidth requirements of the end users. Moreover, it is also necessary for multiple sensor data streams to be fair to each other when sharing a common bottleneck bandwidth link.

(ii) **Architecture for Deployment of Application-aware Protocols in Overlay Networks:** Applications relying on overlay-based implementations to achieve performance, reliability and other application specific requirements must be able to configure overlay nodes to perform in-network application-aware processing. A flexible, efficient approach for the deployment of QoS-sensitive applications using overlay networks should facilitate the monitoring of the QoS received by an application in the overlay network, and allow easy deployment of application-aware processing at intermediate overlay nodes. The architecture framework should be flexible to consider all the above mentioned requirements for development and deployment of application-aware transport protocols on the overlay networks.

3.2.1.2 Framework for Measuring QoS received by End Users

There is a need for a framework to evaluate the QoS received by end users in mission-critical broadband sensor network applications. There are different ways in which QoS may be measured; some of these metrics can be application specific such as error in the end results after computation or generic metrics like freshness of the data received at the processing node. In CASA context, for e.g., end algorithms can compute quality of the received data based on the standard deviation [Ba05b] in the end results. A generic measure such as freshness of data received by the end users may provide significant amount of information about the quality of the data that are critical to the operation of real-time sensor network applications. It is also important to understand how different network dynamics may impact the freshness of data delivered to the end users in sensor network. One of the key goals of the dissertation is to develop a model to understand the precise relationship between network dynamics such as packet loss rate, network delay to the freshness of the data delivered to end users.

Chapter 4

APPLICATION-AWARE CONGESTION CONTROL PROTOCOL

Network dynamics such as packet drops and delay may degrade the perceptual quality of the applications [Ji00, Am06]. Ubiquitous transport protocol like TCP and UDP are not sufficient by themselves for meeting real-time rate and data framing requirements of multiple end users under dynamic network conditions [Ba05b]. There is a need for development of application-aware congestion control algorithm that adapts their transmission rate in an application friendly as well as network friendly manner. In the case of video transmission NAÏVE encoding scheme enables graceful degradation of video quality under network congestion [Br99]. This chapter proposes an application-aware congestion control protocol for high-bandwidth data dissemination using overlay networks. This chapter considers a weather monitoring application CASA for distributing high-bandwidth time-series radar data to one or more heterogeneous end users. Radar time-series data is required to detect a meteorological signal and make estimates of the fundamental moment parameters like reflectivity, Doppler velocity, and spectral width [Br01]. There are different ways in which time-series data can be requested by the end users. It may be required to stream time-series radar data in real-time or in non real-time. For real-time time-series data streaming the bandwidth requirement could be orders of magnitude higher than common Internet applications

like Voice/Video streaming. In case of voice transfer, bandwidth requirements can vary from 6 Kbps to 128 Kbps, and for video streaming, bandwidth requirement can vary from 50 Kbps to 6 Mbps. Alternatively, bandwidth requirement of real-time time-series radar data is in the order of tens of Mbps to hundreds of Mbps.

Quality of the time-series data delivered to end users has the potential to impact performance of the end users. Number of time-series samples received for a resolution volume determines the accuracy of the end moment parameters; higher the number of samples higher is the accuracy. Many end algorithms have a limit on the errors that they can tolerate in the moment parameters. Maximum acceptable error in the moment parameters determines the minimum number of time-series samples required per resolution volume. Due to real time requirements of end algorithms, it is necessary to deliver the minimum number of samples in a bounded time. This determines the minimum bandwidth required for the end algorithm. Therefore it is necessary for protocols to always transmit data at or above this minimum rate for a particular algorithm. It is possible that network becomes a bottleneck due to limited bandwidth availability. Under these scenarios, the transport protocol should not transmit data at a rate that the network cannot support; it would not only aggravate the network congestion but may lead to high losses for the end applications as well. Similarly depending on the resources available at the destination, an application can dictate the maximum rate at which it can receive data from the radar server. An end algorithm may dictate sample requirements depending on the acceptable error in the moment parameters. We use TRABOL congestion protocol proposed in [Bg02, Bg03a, Bg03b, Bg03c] to determine the current transmission rate during network congestion while considering minimum rate and maximum rate requirement.

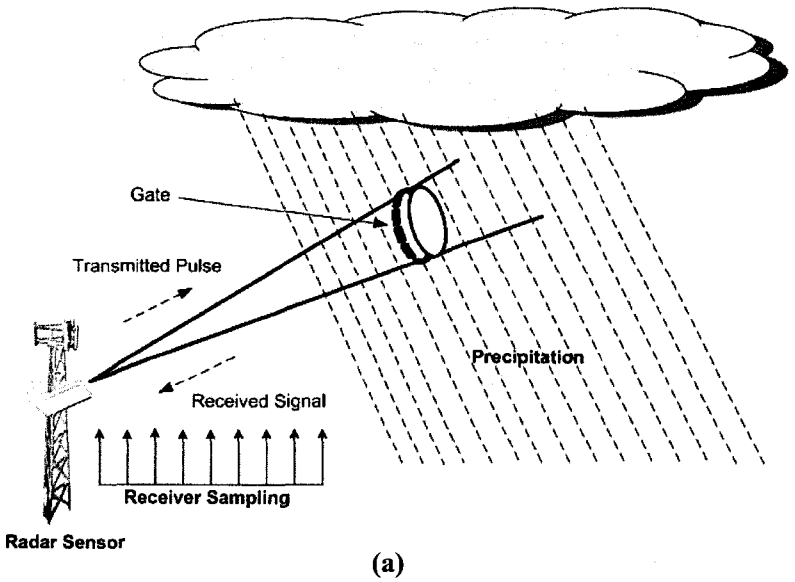
One of the key contributions of the work presented in this chapter is that it proposes application-aware congestion control framework by coupling TRABOL congestion control protocol with the application-aware data selection scheme for transmission of high-bandwidth sensor data. Subsequently this application-aware congestion control framework is extended in

Chapter 5 to propose a DOOM (**D**eterministic **O**verlay **O**ne-to-**M**any) protocol for high bandwidth weather radar data dissemination to multiple heterogeneous end users with distinct bandwidth and data quality requirements. Section 4.1 provides goals for the application-aware congestion control for DCAS Systems. Section 4.2 describes the radar data format. Section 4.3 describes schemes for selection of the subset of radar data for transmission during network congestion. Section 4.4 investigates the impact of integration of TRABOL congestion control and data selection scheme on the quality of data delivered to the end users. Experiment results are presented in Section 4.5. Concluding remarks about this chapter are presented in Section 4.6.

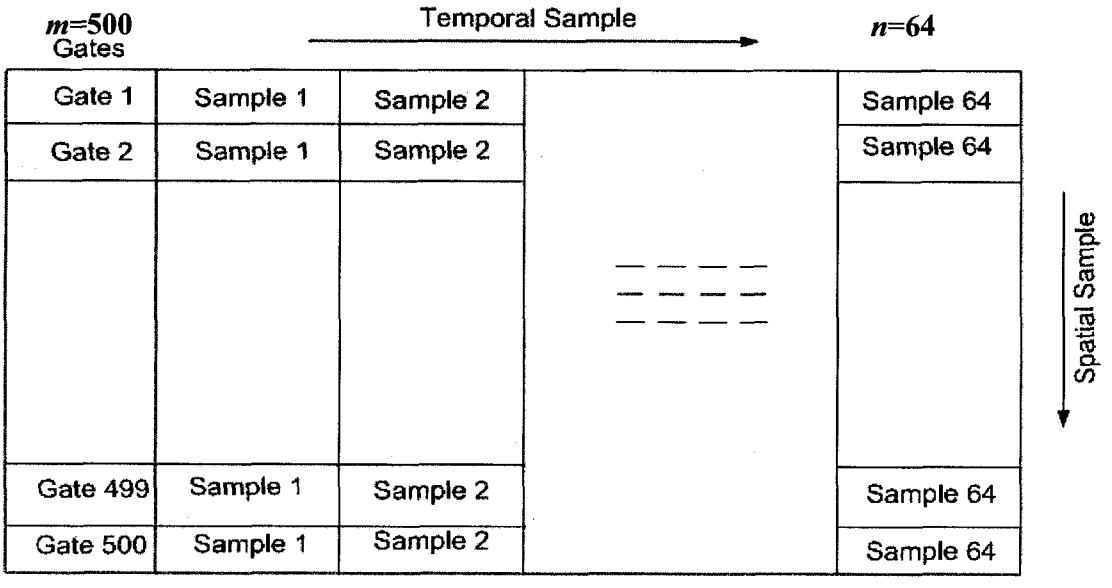
4.1. Application-Aware Congestion Control for Mission-Critical DCAS Systems

The application-aware congestion control protocols for a mission critical DCAS system such as CASA needs to meet following key goals:

1. Ensure that the application specific minimum data rates (*MR*) and its time constraints are met. This is of paramount importance in a mission critical system where failure can affect property and lives. Furthermore, receiving data at rates below this minimum rate (*MR*) renders the received data essentially useless;
2. Be TCP friendly to the maximum extent possible without violating Goal 1;
3. Provide the application its desired data transmission capacity for best quality. While radar applications can operate with minimum threshold of data, the accuracy of end results will improve with higher data rates, up to the target rate (*TR*). Thus the protocol should have the ability to provide improved data rates, provided it can be done without violating Goal 2; and
4. It should facilitate the application to select appropriate subset of the data generated by the sensor node for transmission at lower rates during network congestion.



(a)



(b)

Figure 4.1 Weather Radar Data Generation (a) Radar operations (b) Digitized Radar Signal (DRS) block generated by radar for each scanning direction (fixed azimuth and elevation angle)

TRABOL meets the above goals while overcoming the limitations of both TCP and UDP protocol by providing application level congestion control such that bandwidth

requirements of the end users are met while remaining friendly to the TCP cross-traffic. The key feature of the TRABOL is that during network congestion, it performs rate adaptation while considering end user specific minimum and target rate requirements. TRABOL adapts the transmission rate such that during congestion it does not fall below the required minimum rates. Similarly, when bandwidth is available, TRABOL increases rate to target rate. As mentioned in before for such mission-critical sensing systems it is not sufficient to receive data at the required bandwidth but it is also important to select most appropriate subset of the data for transmission at the current rate.

4.2. Digitized Radar Signals

Radar is a high-speed sensor that generates data at rates of tens of Mbps to hundreds of Mbps. In a typical operating scenario, radar transmits short pulses of energy, which are scattered back by the target, received by the receiver, and digitized for further signal processing [Br01, Ch04]. Fig. 4.1(a) shows the radar operation and Fig. 4.1(b) shows a block of data, corresponding to a ray, while scanning a particular direction in the atmosphere. The transmitter radiates one pulse every Pulse Repetition Time (PRT), which is usually about one millisecond. The received signal is sampled, typically at sampling rates of 1 MHz to 4 MHz. The sampled signal at the receiver is referred to as Digitized Radar Signal or DRS. The regularly spaced times are referred to as gates or range gates and there would typically be 500 to 1000 gates in a ray [Br01, Ch05, Chi]. The distance between consecutive measurements in a radial direction, typically between 100-500m, is called 'range resolution' and this gives the number of gates in a ray. For example for a pulse duration of 1 μs, pulse length corresponds to 150m (using the formula, $r = c\tau /2$, where r is distance and 'c' is the speed of light and 'τ' is the pulse duration). For each transmitted pulse, one

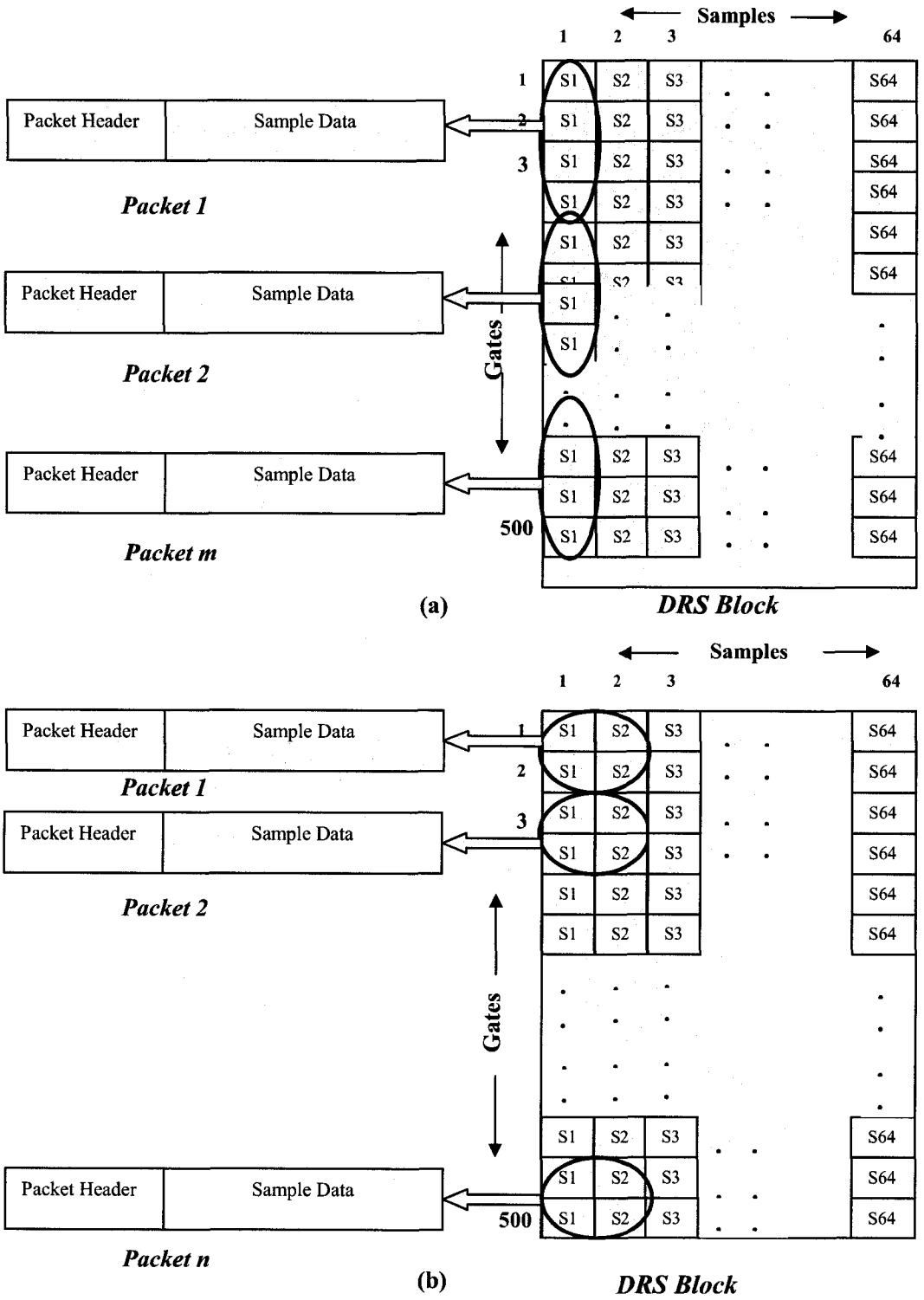


Figure 4.2 Sample Dependency Type (a) Type 1 Single Sample (b) Type 2 Pair of Samples

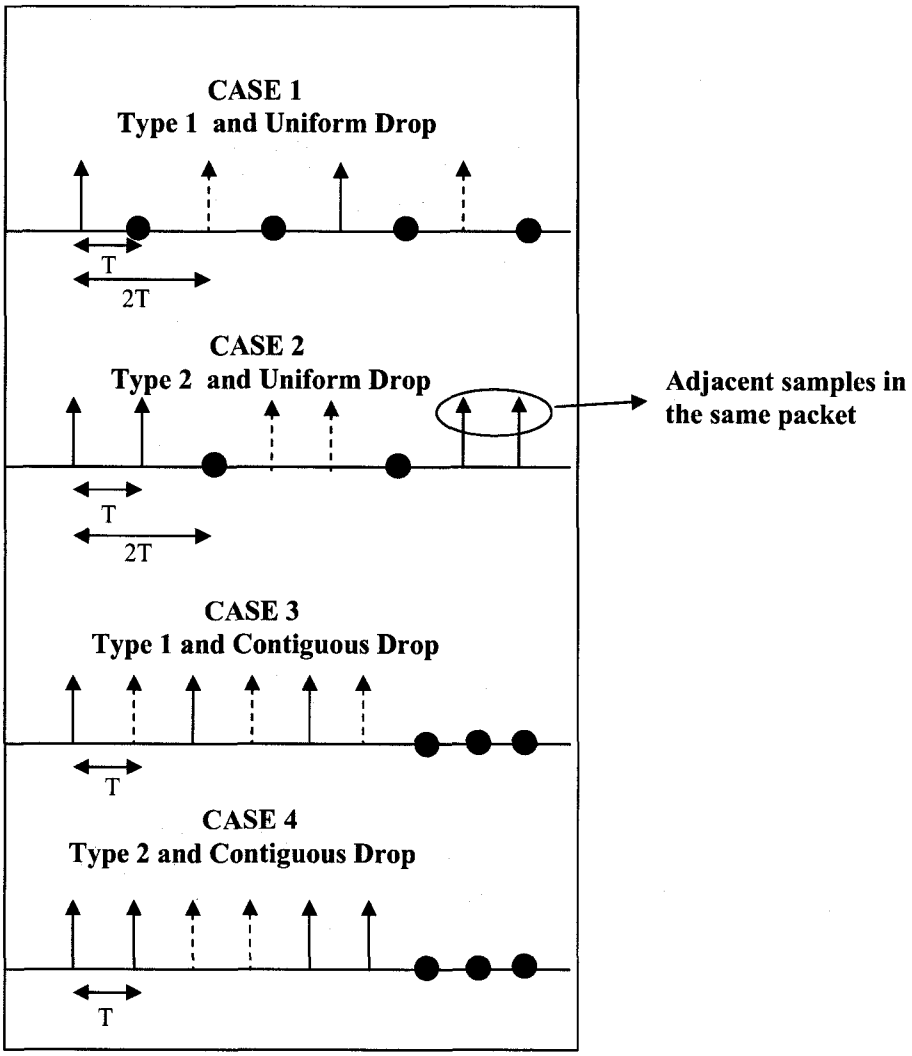


Figure 4.3 Sample selection schemes while considering sample dependency and sample drop requirements of end user applications

sample per gate is generated, which is known as range sample data set. A collection of the multiple range sample data sets corresponding to n transmitted pulses is referred to as a ray digitized radar signal (DRS) block. Time taken generate 1 DRS block of data is known as *dwell time* or *heart-beat* of the radar. In Fig. 4.1(b), DRS block has m (=500) gates and n (=64) pulses. Note that data is generated by column, and the ray DRS block can be conceptually visualized as a two-dimensional array. The maximum range that the

radar can scan is divided in to a number of range gates also known as resolution volume. A row represents all the samples of a given gate. The position of the data in each column indicates the distance of the "object" from the radar. The collection of all the rays for a fixed elevation angle or azimuth angle is referred to as a sweep, and a complete set of sweeps is referred to as a volume scan as it generally contains a complete 3-D view of a storm.

4.3 Application-Aware Sample Selection Scheme

During network congestion network may drop packets randomly, which has the potential to degrade performance of the end users. With congestion control, when all the information cannot be transmitted over the network in real-time then subset of the data may be selected for transmission based on its usefulness for the end users. In case of time-series radar data, the subset of the samples in the DRS block shown in Fig. 4.1(b) may be selected at lower transmission rate for each gate during network congestion [Cl90]. Different radar end user applications may have different sample requirements, e.g., Doppler velocity computation needs adjacent samples for processing and reflectivity computation can be done using single sample at a time. Sample selection scheme proposed in this section describes different mechanism of selecting subset of the data from the DRS block during network congestion while considering end user/algorithm requirements. There are two factors that are considered for selecting subset of the data from the DRS block during network congestion. The first factor is known as dependency type, which determines if a selected sample in a DRS block is useful for the end user computation in case it's adjacent or other neighboring samples do not arrive at the destination node due to network losses or selective drop at the sender node or at intermediate nodes within the network. At present two

dependency types are considered based on the requirements specified by end users of the CASA, i.e., (i) Type 1, and (ii) Type 2.

(i) Type 1: In this case end user application can perform computation on-the-fly with one sample at a time, e.g. reflectivity computation. As shown in Fig. 4.2(a), a single sample from multiple resolution volumes is included in the same packet. Advantage of the proposed approach is that under lossy network conditions, when a packet is dropped only one sample for any resolution volume is lost thus not impacting the quality of the end result significantly.

(ii) Type 2: This type is required for end user applications that need two adjacent samples for performing the computation, e.g., Doppler velocity computation. As shown in Fig. 4.2(b), a pair of adjacent sample for multiple resolution volumes is included in the same packet. In case of packet loss, samples are dropped in pairs for a particular resolution volume. Whenever packets are received, they always have samples in pairs.

Second factor known as sample drop scheme considers which samples can be dropped within the DRS block without significantly degrading the performance of the end user. When all the samples cannot be transmitted because of network bandwidth or client end limitations, then it is required to transmit less number of samples to the destination. Thus it is necessary to drop some samples at the sender end; there are different ways in which samples can be dropped within DRS block, e.g., uniformly spaced drop, drop in contiguous group, or random drops. Each of these sample drop scheme may have different impact on the accuracy of the moment parameters. So algorithms may specify different sample drop schemes requirement that would minimize the errors in the moment parameters. We consider two sample drop schemes, i.e., (i) Uniform Drop and (ii) Contiguous Drop.

These sample drop schemes are used to select samples to be dropped for each gate, i.e., resolution volume in a DRS block at the source node during network congestion. Fig. 4.3 explains the sample drop schemes, each arrow in Fig. 4.3 represents a transmitted sample, adjacent arrows (dashed or solid) represent samples that are included in the same packet and dot represents a

sample that is dropped at the sender end. Each end user may specify its Type 1 or Type 2 requirements along with sample drop schemes to the radar data server, which is then used to select appropriate subset of the samples during network congestion.

(i) Uniform drop: In this sample drop scheme, sample are dropped at a regular interval within a given resolution volume. In Fig. 4.3, Case 1 shows sample drops using uniform drop scheme. All the samples that are selected for transmission using this scheme also meet the sample selection requirement Type 1 or Type 2 of the application. This means that, while samples are dropped at regular interval within a resolution volume, at the same time they are transmitted either as Type 1 or Type 2 as shown in Fig. 4.3 under Case 1 and Case 2.

(ii) Contiguous drop: In this drop scheme, a single cluster of adjacent samples is dropped. Number of samples to be dropped in a single cluster is determined by the rate at which data is to be transmitted. Remaining samples are transmitted using the user specified dependency type requirement. Fig. 4.3, Case 3 and Case 4 shows the sample drops using contiguous drop scheme while meeting the sample selection requirement of the application.

4.4 Impact of Integration of Application-Aware Data Selection Scheme and TRABOL Congestion Control Protocol

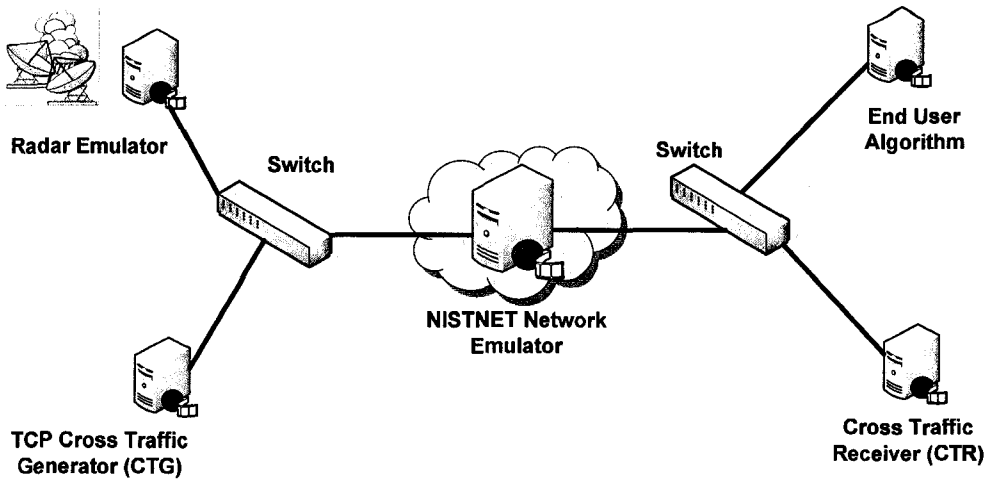
This section investigates the impact of integration of data selection scheme with the TRABOL congestion control protocol. Effectiveness of the application-aware congestion control protocol is illustrated by comparing the performance of TCP, UDP, and TRABOL based implementations.

In case of TCP, all samples are delivered to the destination; therefore sample selection scheme is not required for the transmission. In case of UDP, dependency type requirement of end users is considered during the transmission. However, UDP does not drop packets at the sender

end to avoid network congestion, thus no sample drop requirements are considered with the UDP. Instead packets are dropped randomly in the network during network congestion. Alternatively, it is possible to select both sample dependency and sample drops schemes as per the end application needs with TRABOL. Since TRABOL can dynamically adapt its transmission rate, it increases its transmission rate by increasing the number of samples to be transmitted as per the sample dependency and sample drop requirement of the end user. Similarly transmission rate can be reduced by sending less number of samples while considering both sample dependency and sample drop requirements. Note that in [Ba05b] sample dependency scheme is referred as sample group requirement and sample drop scheme is referred as sample pattern requirement.

Effect of different protocols on the quality of data is determined using application-specific metric standard deviation for weather radar data.

Standard deviation in moment parameters: We consider a case when sender node sends multiple realizations of simulated radar data [Ch86], where each realization corresponds to a DRS block of data for the same azimuth and elevation angle. For each realization that is delivered to the end user, moment parameters such as reflectivity and Doppler velocity are computed for all resolution volumes. Moment parameters are computed for all gates/resolution volume for all realizations. However, moment value of a particular gate may vary from realization to realization. Standard deviation in the moment parameter is computed to estimate variation in the moment parameters. Moment parameter for a particular resolution volume over all realizations of DRS block is likely to have minimum standard deviation when all samples of a resolution volume are used for the computation. However due to network dynamics all the samples may not be delivered to the end users for a particular resolution volume. As the number of samples decreases for a particular resolution volume, it leads to an increase in the standard deviation in the moment parameters while considering all the realizations. Standard deviation in moment parameters also depends on the factors like, whether single samples, pairs of adjacent samples, or triplets of adjacent samples are used for the computation. E.g., reflectivity computation can be performed



Figurer 4.4 Emulation Network to compare performance of application-aware TRABOL, TCP and UDP

on-the-fly using one sample at a time within a resolution volume. Alternatively, for Doppler velocity computations it is desired to use pairs of adjacent samples within a resolution volume for accurate estimation of Doppler velocity. Other factor that may impact variation in moment parameters values is how do samples losses are distributed for a given resolution volume, i.e., missing samples are at regular interval or in bursts for a given resolution volume. If the desired samples are not received by the end user for a particular algorithm then that may lead to wide variation in moment parameter values for a given resolution volume for different realization. Therefore, standard deviation in moment parameters can be used as an application-specific metric to evaluate the quality of the time-series radar data received by the end users.

Under ideal conditions, when there is no congestion on the network, then end user receives all samples of all the resolution volumes with high probability in a wired network environment. In reality packet loss is a common phenomenon on the Internet; under network congestion time-series radar data can suffer variable losses. Thus it is imperative to understand the impact of dynamics of the network on the quality of data in terms of standard deviation in the moment data. Transport protocols may have distinct behaviors under similar network conditions, thus it is possible to receive different samples using different protocols for the same resolution volume that

can lead to different standard deviation in the moment parameters. Therefore standard deviation in the moment parameters can be used to compare performance of transport protocols.

Experiments are conducted using a radar network emulation test-bed. There are five components of this test bed as shown in Fig. 4.4: (i) Radar emulator, (ii) Network emulator, (iii) End User Algorithm, (iv) Cross traffic generator (CTG), and (v) Cross traffic receiver (CTR).

Radar emulator, network emulator, and client machine are the Dual Xeon processor 3.06GHz server machines with 2GB RAM. Radar emulation is done using archived time-series data. The time-series data consists of multiple realizations of DRS block of data that consists of 300 gates and 64 samples per gate. On the end user node, all meteorological algorithms are executed and performance analysis is done. Network emulator NISTNet is used to emulate different network bandwidth and loss scenarios. Since radar emulation is a disk intensive operation, RAID 0 functionality is used to enhance the read and write performance of the disks in server machine.

Radar emulator generates data at 90 Mbps, with 300 gates per ray and each gate has 64 samples. Different network dynamics like bottleneck bandwidth, packet losses etc. are emulated using cross traffic generator (CTG) and cross traffic receiver (CTR). Under different network conditions like variable packet losses, impact of different transport protocols and different sample selection schemes is studied on the end algorithms. Performance comparison of impact of UDP, TCP and TRABOL on TCP cross traffic is done.

4.5 Performance Results

Experiments are performed to investigate the effect of different sample selection schemes with UDP and TRABOL on the moment parameter computations.

Radar data is transmitted using different sample selection schemes as explained in Section 4.3. Radar data quality is determined by estimating standard deviation in the moment parameters,

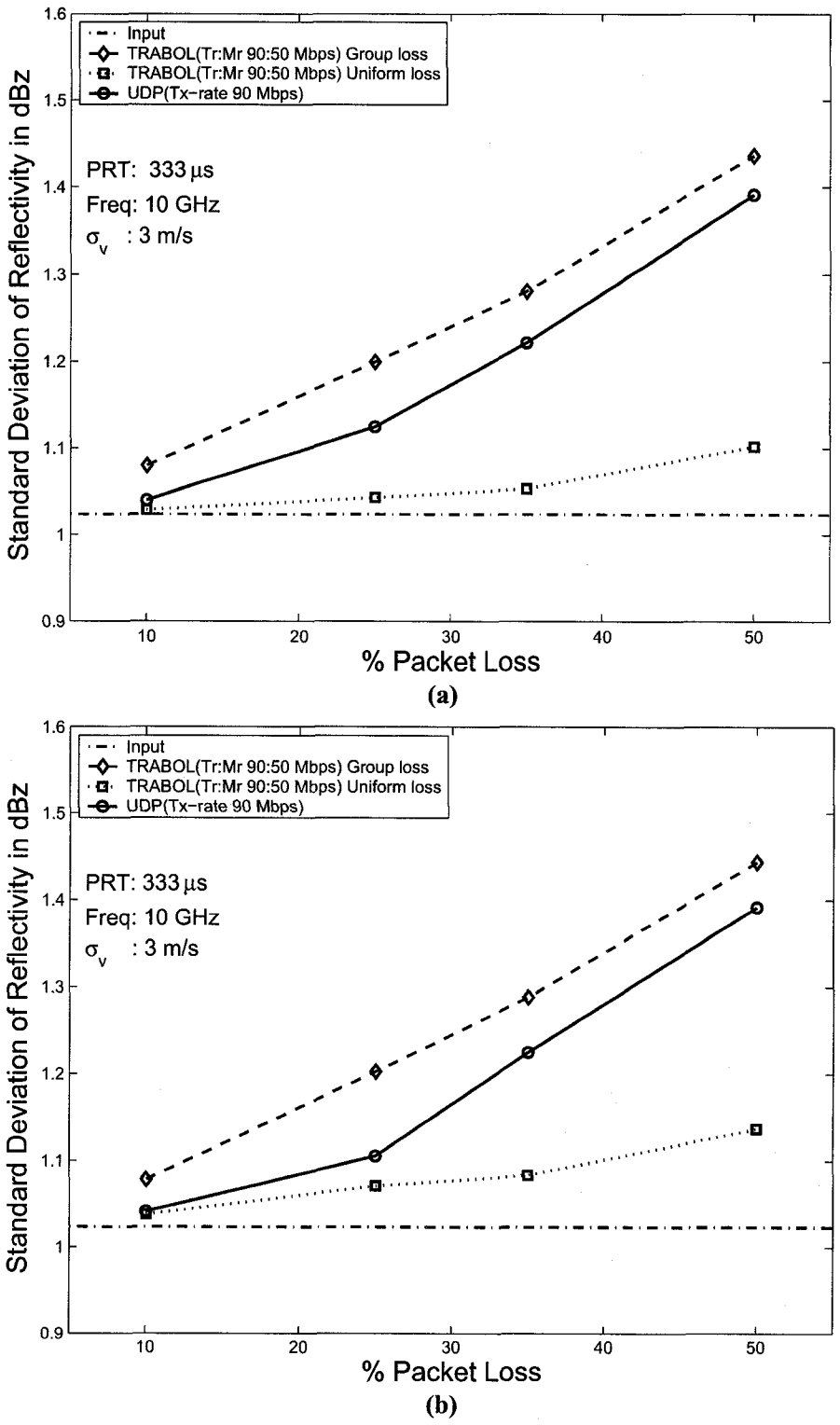
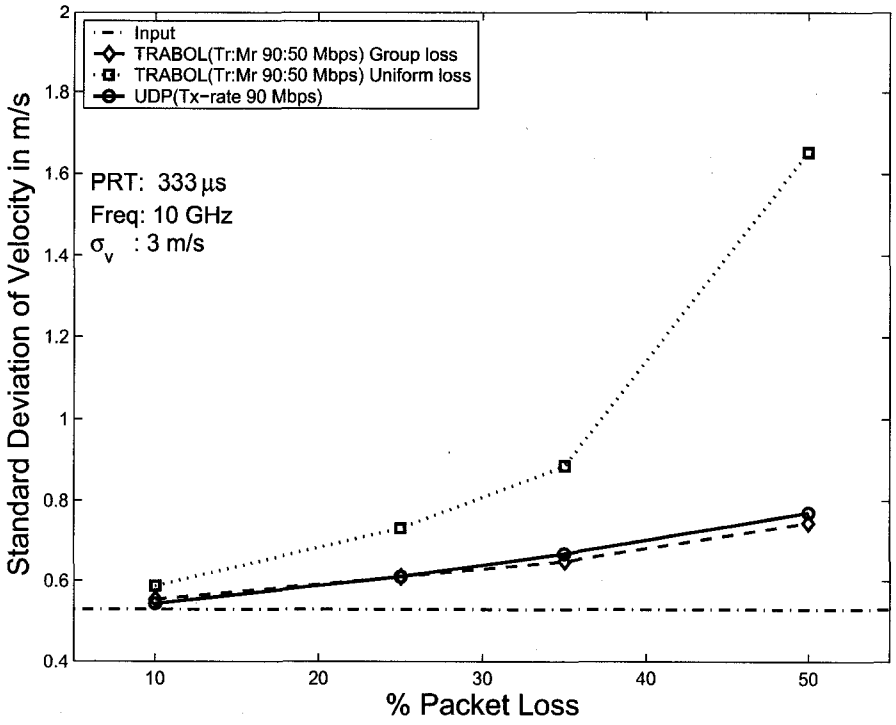
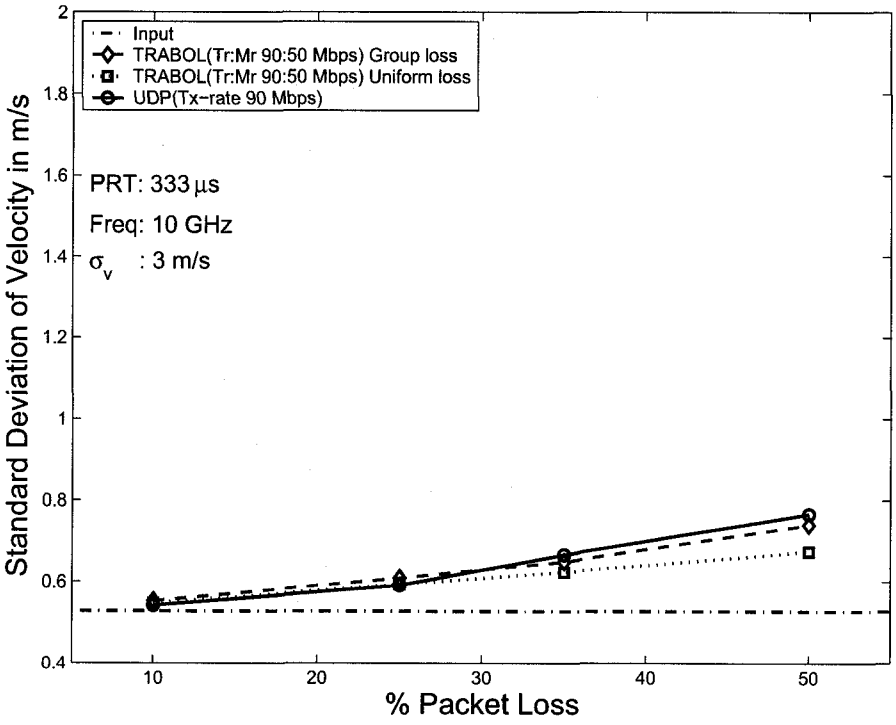


Figure 4.5 Impact of sample selection schemes using UDP and TRABOL on the radar data quality (a) SD in reflectivity with Type 1 sample dependency under uniform and contiguous drops (b) SD in reflectivity with Type 2 dependency under uniform and contiguous drops. (Group loss in figure correspond to contiguous drop)



(a)



(b)

Figure 4.6 Impact of sample selection schemes using UDP and TRABOL on the radar data quality (a) SD in velocity with Type 1 dependency type under uniform and contiguous drops (b) SD in velocity with Type 2 dependency type under uniform and contiguous drops. (Group loss in figure correspond to contiguous drop)

computed using the received samples for a particular resolution volume. In case of TCP, since all the samples are received, the standard deviation in the moment data is minimal. For a fair comparison of TRABOL and UDP, it is desired that radar data quality is compared under same network loss conditions. Same sample dependency types are used by both protocols, i.e., either Type 1 or Type 2 samples are sent using UDP and TRABOL. For the experimental results, available bottleneck bandwidth is set to 45, 60, 70 and 80Mbps; that correspond to 50%, 35%, 25% and 10% packet loss respectively. When UDP is used for radar data streaming, there is no mechanism for rate control, thus data is always transmitted at the rate at which data is generated by the radar emulator, i.e., 90Mbps. Since available bandwidth is less than the 90Mbps, the packets losses introduced are random in nature for the radar data. Experiments are performed for a case when end application has target rate requirement of 90Mbps and minimum rate requirement of 50Mbps. TRABOL adapts transmission rate dynamically between maximum rate 90Mbps and minimum rate 50Mbps as per the available bandwidth. In case of TRABOL overall same amount of information is lost as in the case of UDP but TRABOL drops most of the samples at the sender end deterministically as per the end applications sample group and sample pattern requirements. When bottleneck bandwidth link is 45Mbps, TRABOL end user receives data below its minimum rate requirement.

Fig. 4.5 and Fig. 4.6 compare the impact of different data selection schemes on the radar data quality for reflectivity computation and Doppler velocity computation respectively under different network congestion/packet loss conditions. Experimental results in Fig. 4.5 and Fig. 4.6 show that sample selection scheme when integrated with TRABOL congestion control protocol, significantly enhances the accuracy of the moment parameters during network congestion and high packet loss conditions. UDP packet losses are random in nature due to network dynamics, thus standard deviation is high in most cases when compared under same receiver throughput conditions for both UDP and TRABOL. In Fig. 4.5(a), samples are selected based on Type 1 sample dependency requirement of end users for both UDP and TRABOL. It can be seen that

TRABOL with deterministic uniform spaced drops within DRS block has minimum standard deviation in reflectivity when compared to UDP with random losses and TRABOL with contiguous losses. Fig. 4.5(b) shows a case when samples are selected based on Type 2 sample dependency requirement of a end user for both UDP and TRABOL. Same behavior is observed in Fig. 4.5(a), i.e., TRABOL with uniform loss with Type 2 dependency has minimum standard deviation for reflectivity compared to UDP with random loss and TRABOL with contiguous loss.

In Fig. 4.6(a), samples are selected using Type 1 dependency requirement of the end use for both UDP and TRABOL. TRABOL with contiguous drops and UDP with random loss performances are quite similar. TRABOL with uniform drops with Type 1 data dependency requirement shows worst performance because uniformly distributed drops within DRS block with Type 1 sample dependency leads to minimum number of sample pair delivery, increasing standard deviation in end parameters. Fig. 4.6(b) shows a case when Type 2 data dependency is used; in this case once again TRABOL with uniform drop of sample pairs performs better than the rest. It is thus evident that, TRABOL along with deterministic sample selection scheme can out perform UDP in terms of better quality moment data under similar network conditions. Note that in case of TCP, all samples for a given gate are received therefore quality of the data is superior to either UDP or TRABOL under lossy network conditions. In Fig. 4.5 and Fig. 4.6, dotted line corresponding to input can be taken as a measure of standard deviation in moment parameters due to TCP protocol.

These results have demonstrated that integration of application-aware data selection scheme and congestion control algorithm helps in delivering better quality data to the end users under dynamic network conditions. The other part of the study that investigates the TCP friendliness of Application-aware TRABOL, UDP, and TCP is performed in [Ma05].

4.6 Remarks

This chapter proposed the integration of application-aware data selection scheme with the TRABOL congestion control protocol for sending digitized radar data. We also compared the performance of TCP, UDP and TRABOL for sending digitized radar data. Radar data quality results show that for same sample selection scheme and under similar available bandwidth, quality of the data received using TRABOL is better than the UDP case. After evaluating performance results, we can conclude that TRABOL, along with the different sample selection schemes, is able to meet the radar quality requirement without overly degrading the performance of other applications on the network. Chapter 5 extend this work to propose an overlay network based application-aware multicast protocol to deliver high bandwidth time-series radar data to multiple heterogeneous end users while considering their distinct QoS requirements.

Chapter 5

DOOM PROTOCOL FOR APPLICATION-AWARE ONE-TO-MANY DATA DISSEMINATION USING OVERLAY NETWORKS

In distributed collaborative adaptive systems (DCAS) such as CASA, multiple end users at distant geographical locations may need real-time access to the weather radar data. Each of the end users may have distinct QoS requirements in terms of data quality and bandwidth requirement. For e.g., each end user may specify its critical minimum rate requirements below which data is not useful. Depending on the available computation and network resources, end user also specifies the target rate above which data cannot be received by the user. Similarly each end user may specify their acceptable data quality requirement. As mentioned in chapter 4, when data suffers random losses in the network due to network congestion, then quality of data delivered to the end users may degrade. It is shown that when congestion control protocol is integrated with the data selection scheme at the source node then it is more effective in delivering higher quality data under dynamic network congestion conditions. The key reason for this gain in performance during network congestion is that the subset of information to be transmitted is determined by knowing the characteristics of the sensor data and tolerance of end users to the missing information.

In this chapter we propose a DOOM (**D**eterministic **O**verlay **O**ne-to-**M**any) protocol for distributing high-bandwidth sensor data to multiple end users concurrently while considering

their bandwidth and data quality QoS requirements. We consider CASA application to demonstrate the suitability of the DOOM protocol for application-aware radar data dissemination in DCAS systems. This protocol uses knowledge about end users sensitivity to subset of the data as explained in Chapter 4 while determining the most relevant information that should be transmitted at lower rates during network congestion for each end user independently. Section 5.1 describes the DOOM protocol architecture. Performance evaluation results using planetlab and emulation test bed are presented in Section 5.2. Section 5.3 provides the concluding remarks on this chapter.

5.1 DOOM: Deterministic Overlay One-to-Many Protocol

This section explains the sender-driven, time multiplexed, **D**eterministic **O**verlay **O**ne-to-**M**any (**DOOM**) protocol for high bandwidth data dissemination to multiple end users. While we develop the concept of DOOM based on radar applications, the protocol is general purpose for use in a broader class of high-bandwidth sensor actuator networks where data has spatiotemporal dependencies [Ba05b, Ba05d, Br01].

DOOM initiates with the knowledge of maximum transmission rate it should support for any particular end user. This is a fair assumption, because in a sensing system similar to CASA, maximum data generation rate of a sensing node is known. At the time of data request, end user informs DOOM overlay server about its data quality and critical minimum rate and target rate requirements. This information is stored in the *user-list* as shown in Fig. 5.1. Current implementation of DOOM supports finite number of different data quality requirements, Type 1 and Type 2 data dependency with uniform drops, with a focus on radar data end applications as shown in Fig. 5.2.

A static *rate-table* of supported transmission rates is defined starting with lowest rate to the maximum possible transmission rate as shown in Fig. 5.1. Maximum possible transmission rate

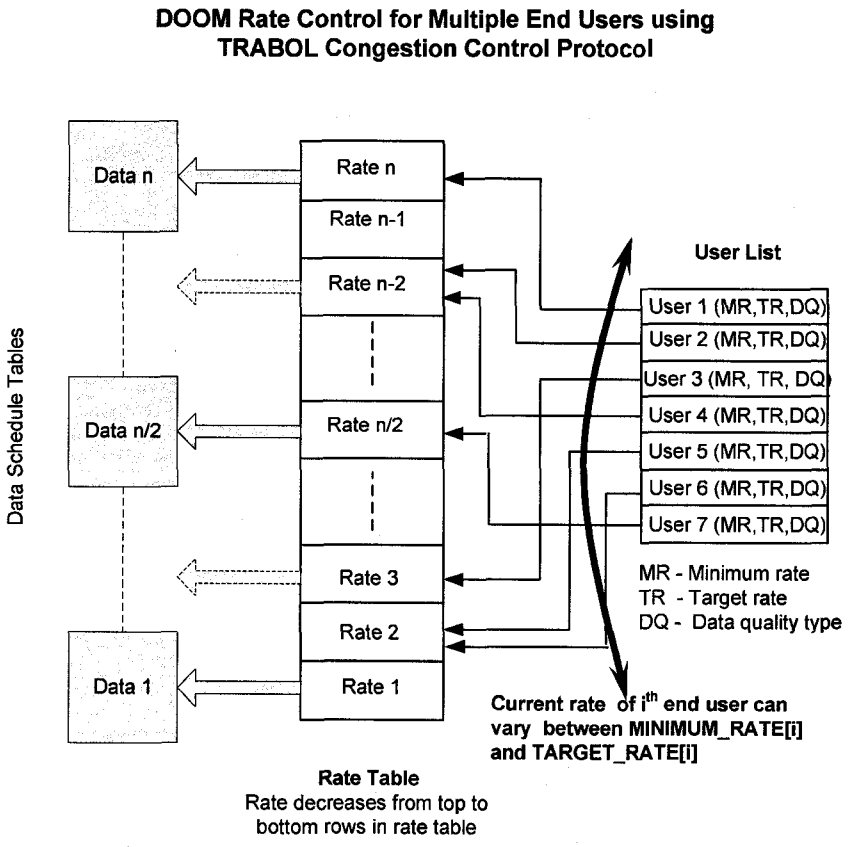


Figure 5.1 DOOM Rate Control for multiple end users using TRABOL congestion control protocol

(*Rate n* in Fig. 5.1) can be determined by the data generation rate of a single sensor node. In case of CASA, each radar sensor generate data at 100Mbps. Minimum rate can be determined by the lowest rate overlay server want to support for any end users (Rate 1 in Fig. 5.1). Current implementation considers minimum supported rate as 1 Mbps and maximum rate 100Mbps. Number of rates supported in rate table is determined by the granularity requirement of the end user applications. In the current implementation, 1Mbps granularity is supported, i.e., two adjacent rates in rate table differ by 1Mbps.

In many sensor based applications, sensor data have a fixed format and data generation rate. Thus it may be possible to determine subset of the total data that can be transmitted at transmission rates lower than the generation rate of the sensing node. Data quality needs

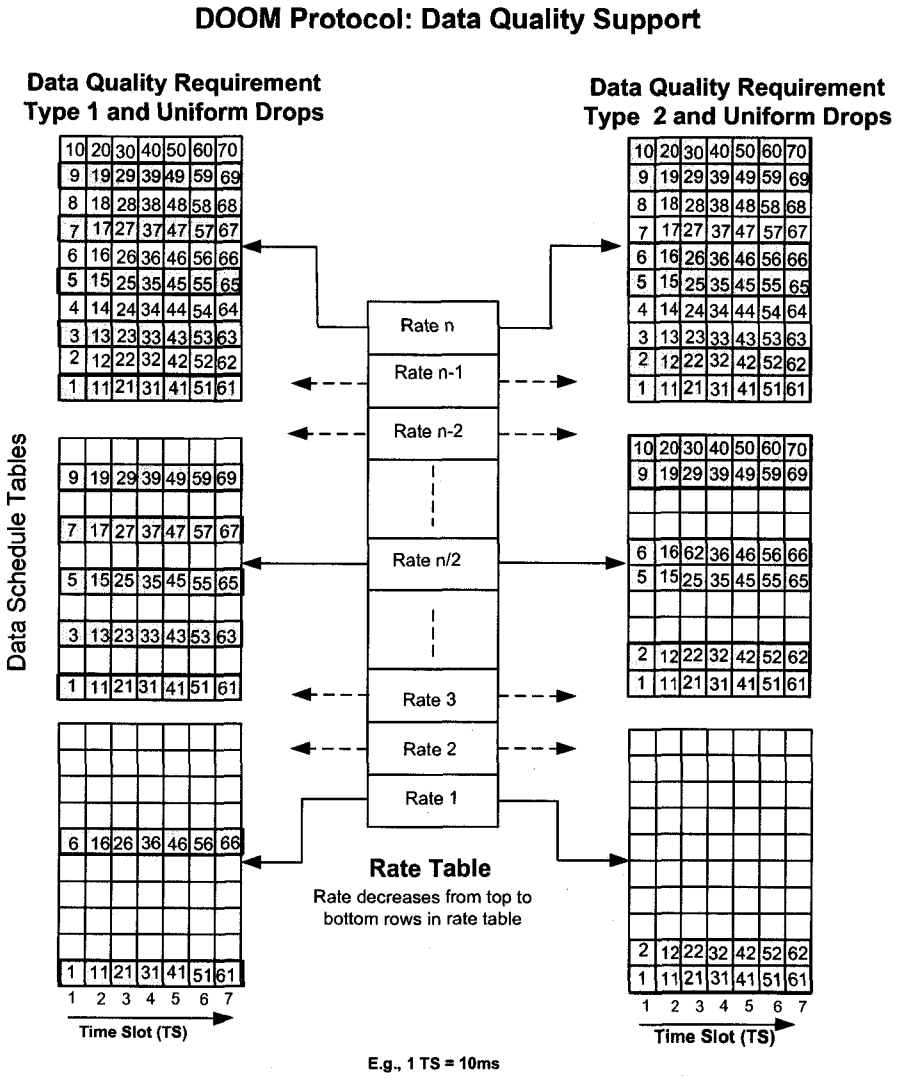


Figure 5.2 DOOM support for multiple data quality requirements

determine how to select subset of data from the total data generated by a sensing node in fixed interval of time, known as *heart-beat* of a sensor, for a given transmission rate. This information is included in the *rate-table* corresponding to each supported rate, shown by data schedule tables in Fig. 5.1 and Fig. 5.2. For supporting multiple data quality requirements, more than one data selection schemes is required, resulting in multiple entries corresponding to each supported rate in the *rate-table*. Fig. 5.2 shows a case when DOOM protocol supports two different data quality requirements Type 1 and Type 2 sample dependency with uniform drops as explained in Chapter 4. For radar data applications, data schedule tables in Fig. 5.2 consists of samples that are selected

for transmission out of DRS block of data generated by radar sensor (similar to DRS block shown in Chapter 4) within *heart-beat* interval. DOOM performs deterministic data selection by run-time lookup in the static data schedule table for each scheduled end user based on the end user data quality requirement and its current transmission rate. Note that with this approach it is possible for two end users to get different data even at the same transmission rate because of their different data quality needs. Spatial and Temporal dependency of data also influence how transmitted information is encoded for a particular end user. For example, depending on the observed event by a sensor there may be dependency between the two adjacent samples of the generated data. In certain cases, one sample may not be useful without the other for the end applications. Thus it may be required to guarantee delivery of both adjacent samples, which can be achieved by sending both samples in the same packet. As shown in Fig. 5.2, data schedule tables corresponding to data quality requirement Type 2 and uniform drops is the case where adjacent samples in column have temporal dependency (indicated by same background color for adjacent samples in a column). In case client needs Type 2 with uniform drops, adjacent samples are selected for transmission in the same packet. As shown in Fig. 5.3, TRABOL congestion control protocol determines the next transmission rate for each end user independently based on the packet loss count feedback received from the end user. A particular transmission rate is achieved by transmitting the data given in corresponding data schedule table in a *heart-beat* interval. Note that in case of DOOM protocol, TRABOL always determines next transmission rate which is supported by the rate table in Fig. 5.1. In order to avoid sending all the data to a particular user in single burst and to support multiple data quality requirements, *heart-beat* interval is divided into multiple scheduling time slots of 10ms. Fig. 5.2 shows a case for 70ms *heart-beat* that result in 7 time slots of 10ms each. For time multiplexing, a periodic 10ms timer is used, and when this timer times out a table lookup is performed in the corresponding data schedule table for a

```
                    DOOM Server
heart-beat: Time for one block data generation
time_slot:  Time window for scheduling (10ms)
data        :  Data scheduled for transmission
user        :  End user scheduled to get data
user-list:  List of all users getting data
USER_COUNT: Number of user requests
ACK   :     Acknowledgments received from a
            user (received packet count)
WHILE (1)
 {
 // Repeat following every heart-beat
 // interval
 IF (USER_COUNT >0)
  {
  // Determine new transmission rate
  // of each client every heart-beat interval
  FOR (EACH USER IN client-list)
   {
   // Use TRABOL congestion control to
   // determine next transmission rate
    IF (ACK RECEIVED)
      {
      determine_TRABOL_rate(user, ACK)
      }
     else
      {
      determine_TRABOL_rate(user,
                                  NO_ACK)
      }
    // User next rate information is updated
    update_rate_table(user)
   }
  // Use Time multiplexing to transmit data
   FOR (EACH TIME SLOT )
     {
     FOR (EACH USER)
       {
       // Get data schedule table for the client
       data_schedule_table =
                          get_reference(user)
       // Determine data to transmit for a given
       // client in the current time slot
        data = lookup(time_slot,
                      data_schedule_table)
       send_data(data, user)
       }
     }
  }
 }
```

Figure 5.3 DOOM algorithm for one-to-many data dissemination

particular end user to select the data for transmission. Data corresponding to current *time-slot* is then encoded and transmitted as per end user requirements. Note that in any one 10ms *time-slot* multiple end users can be scheduled to get data at different transmission rates. Cumulative transmission rate for all scheduled end users within *time-slot* interval should be less than the output link bandwidth. This time multiplexed data scheduling scheme enables concurrent transmission of different subset of data from the same DRS block to different end users in order to satisfy their unique rate and data quality requirements. Moreover excessive bursty-ness of data is avoided by scheduling the data for transmission uniformly over the *heart-beat* interval, i.e., over multiple *time-slots*.

Each end user transmits received packet count for data transmitted in the heart-beat interval. This helps in avoiding flood of ACK traffic from multiple end users to the DOOM server. At the start of periodic heart-beat interval, new transmission rates are computed for each end user using TRABOL based on the last feedback from an individual user. After new transmission rates are determined for all the requesting end users, data can be transmitted using new data transmission schedules for that particular rate.

5.2 Performance Evaluation

Planet-Lab [Pla, Pe02] and the emulation test-bed shown in Fig. 5.4 are used for the performance analysis of DOOM protocol. Latter is based on NISTNET [Ca03] network emulator which emulates different network dynamics such as bandwidth and delay variations. We consider the case when a radar node generates data at a constant rate of 100Mbps. Experiments are performed to evaluate performance of DOOM overlay server in meeting different rate and data framing requirements of multiple end users simultaneously. Friendliness of DOOM streams to each other

as well as TCP cross-traffic, sharing the bottleneck link is evaluated.

Fig. 5.5 shows emulation based results for evaluating DOOM effectiveness in meeting heterogeneous rate requirements of the multiple end users under varying bottleneck bandwidth conditions. In Fig. 5.5, different end users are identified on the x-axis by C1-C6. Each of the end user has different ACK delay and different target rate and minimum rate requirements. Bottleneck bandwidth varies between 105Mbps and 215Mbps because a cumulative minimum rate requirement of all users is 100Mbps and cumulative target rate is 210Mbps. Two cases of sensors are considered when radar sensor *heart-beat* (periodic interval between data generation) is 170ms and 20ms, in both cases radar sensor generate data at 100Mbps. Table 5.1 and Table 5.2 shows the data corresponding to Fig. 5.5 (a) and Fig. 5.5(b) respectively.

As seen in Fig. 5.5(a) and Fig. 5.5(b), when bottleneck bandwidth exceeds the cumulative minimum requirements of all end users, then each user is able to meet its minimum rate requirement. As bottleneck bandwidth increase, all users get fairly equal share of the extra

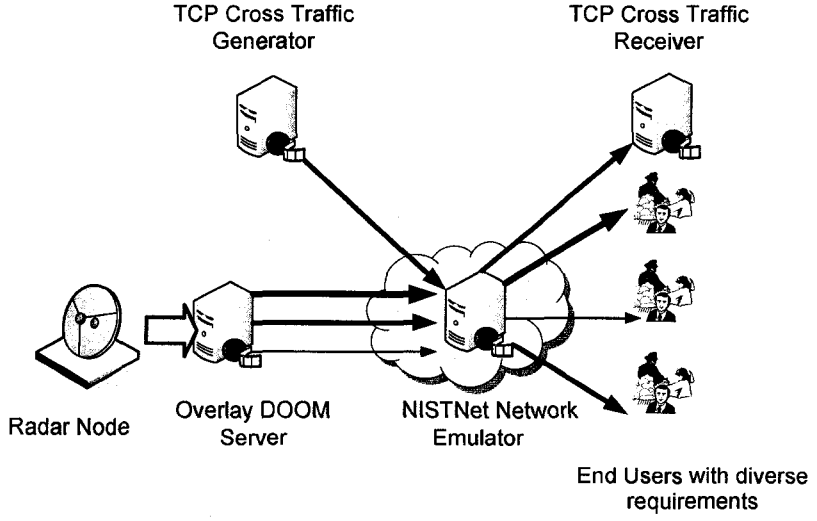


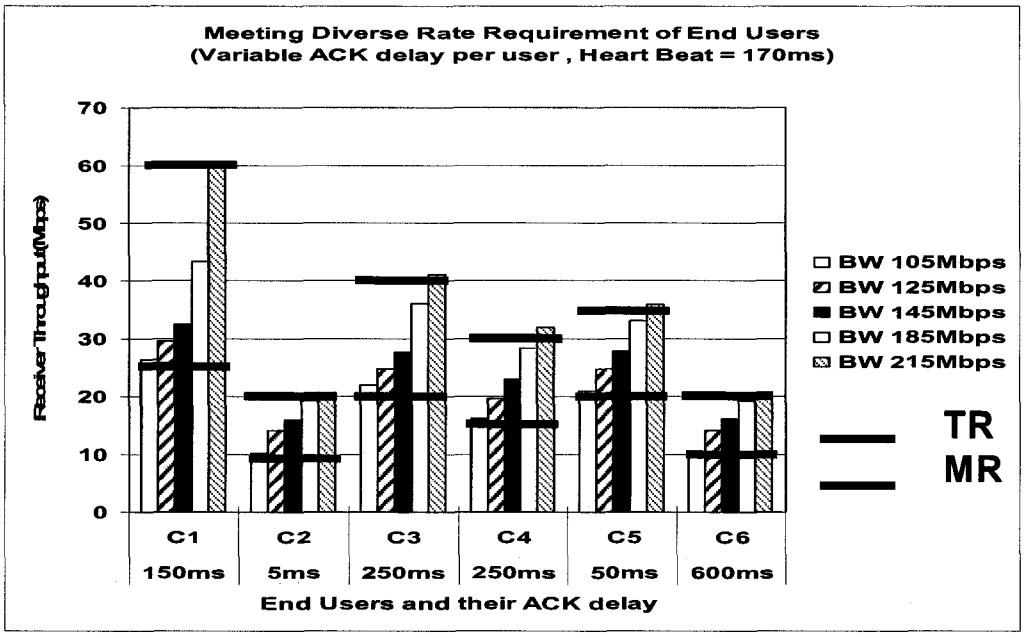
Figure 5.4 Network emulation test bed

Table 5.1 DOOM performance in meeting bandwidth requirements of multiple heterogeneous end users under variable bottleneck BW when heart-beat interval is 170ms

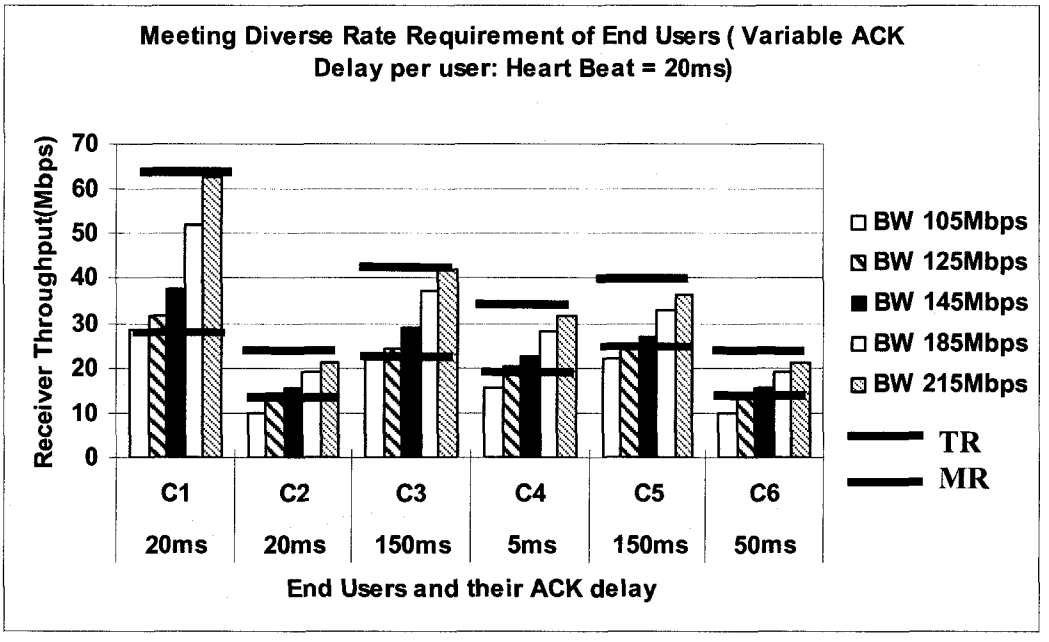
RTT (ms)	*MR (Mbps)*	*TR (Mbps)*	*BW1 (Mbps)*	*BW2 (Mbps)*	*BW3 (Mbps)*	*BW4 (Mbps)*	*BW5 (Mbps)*
150	25	60	26.36	29.52	32.55	43.43	60.42
5	10	20	10.18	14.14	16.04	19.19	20.55
250	20	40	21.95	24.64	27.65	36.06	41.11
250	15	30	16.05	19.67	22.78	28.21	31.95
50	20	35	20.83	24.6	27.69	32.93	35.88
600	10	20	10.44	14.18	16.13	19.26	20.53

Table 5.2 DOOM performance in meeting bandwidth requirements of multiple heterogeneous end users under variable bottleneck BW when heart-beat interval is 20ms

RTT (ms)	*MR (Mbps)*	*TR (Mbps)*	*BW1 (Mbps)*	*BW2 (Mbps)*	*BW3 (Mbps)*	*BW4 (Mbps)*	*BW5 (Mbps)*
20	25	60	28.61	31.6	37.42	51.96	62.51
20	10	20	10.05	14.15	15.75	19.06	21.18
150	20	40	22.44	24.32	29.11	37.1	41.83
5	15	30	15.52	20.27	22.65	28.09	31.49
150	20	35	21.94	24.1	26.98	33.02	36.41
50	10	20	10.06	13.93	15.61	18.98	21.18



(a)



(b)

Figure 5.5 DOOM performance in meeting heterogeneous rate requirements of multiple end users simultaneously with different ACK delays (a) Sensor heart-beat is 170ms, and (b) Sensor heart-beat is 20ms

available bandwidth. When bottleneck bandwidth exceeds cumulative target rate requirements of all end users, then all end users are able to receive data at their target rate requirements.

Fig. 5.6 shows DOOM's effectiveness in meeting similar and different rate requirements of multiple end users based in different countries over Planetlab test-bed, served by DOOM overlay node in Colorado (USA). Table 5.3 shows data corresponding to results shown in Fig. 5.6. 16 end users at different geographical locations throughout the world requests radar data with their bandwidth and data quality requirements. Two cases are considered (i) when all end users have same bandwidth requirement TR=3Mbps and MR=1Mbps, (ii) when different end users have different bandwidth requirement as indicated by solid red and blue lines in Fig. 5.6. Most end users, receiver throughput lies between their target and minimum rate for both cases. As shown in Fig. 5.6, the end user in Oregon (USA) receives data below its minimum rate because of either bandwidth or end node limitations. Fig. 5.7 shows results for quality of the received data over the Planetlab by multiple users with similar and different rate requirements. Table 5.4 shows data corresponding to results shown in Fig. 5.7. In case of CASA radar data streaming, quality of received data is measured by computing standard deviation in the reflectivity algorithm end results for each user as explained in Chapter 4. Lower standard deviation is a measure of better quality data. In Fig. 5.7, most end users in different countries of the world receive similar quality of the data using DOOM protocol at different receiver throughputs. Two end users, Oregon and Canada-1 show comparatively high standard deviation due to lower receiver throughput in case of Canada-1 node and due to random losses in the network for Oregon node because of bandwidth or end node limitations.

DOOM streams may share the network with already existing TCP traffic. Because of the high bandwidth requirements of different end users, TCP friendliness operation of the DOOM protocol is important. Fig. 5.8(a) shows the impact of multiple DOOM streams on the TCP cross traffic in the bottleneck bandwidth link. Table 5.5 shows data corresponding to results shown in Fig. 5.8(a). In this case, three end users with similar target rate requirement of 100Mbps,

Table 5.3 Planetlab based result of DOOM performance in meeting bandwidth requirements of multiple end users

End User	*Case: Similar Rate Requirements TR =3Mbps, MR=1Mbps (Mbps)*	*Case: Different Requirements (Mbps)*	*TR and MR (Mbps)*
Denver	3.05	2.05	TR=2, MR=1
Finland	3.05	2.05	TR=2, MR=1
Korea	3.04	8.08	TR=8, MR=5
Cornell	3.05	2.05	TR=2, MR=1
China	3	2.89	TR=3MR=1
Berkeley	2.92	5.79	TR=8 MR=5
Massachusetts	3.05	2.05	TR=2, MR=1
Cambridge	3.04	2.03	TR=2, MR=1
Canada 1	1.34	1.01	TR=3, MR=1
Canada 2	3.05	3.05	TR=5, MR=3
Oregon	1.76	2.22	TR=2, MR=1
Duke	3.05	2.05	TR=2, MR=1
Japan	3.05	2.05	TR=2, MR=1
Houston 1	3.05	2.05	TR=2, MR=1
Houston 2	3.05	2.05	TR=2, MR=1
Purdue	3.05	2.05	TR=2, MR=1

Table 5.4 Planetlab based result of DOOM performance in meeting data quality requirements of multiple end users

End User	*Case: Standard deviation Similar Rate Requirements (dBz)*	*Case: Different Requirements (dBz)*	*TR and MR (Mbps)*
Denver	1.30	1.35	TR=2, MR=1
Finland	1.297	1.36	TR=2, MR=1
Korea	1.302	1.24	TR=8, MR=5
Cornell	1.306	1.35	TR=2, MR=1
China	1.313	1.34	TR=3MR=1
Berkeley	1.371	1.25	TR=8 MR=5
Massachusetts	1.297	1.35	TR=2, MR=1
Cambridge	1.302	1.35	TR=2, MR=1
Canada 1	2.129	2.36	TR=3, MR=1
Canada 2	1.297	1.29	TR=5, MR=3
Oregon	1.876	1.72	TR=2, MR=1
Duke	1.306	1.72	TR=2, MR=1
Japan	1.305	1.35	TR=2, MR=1
Houston 1	1.297	1.35	TR=2, MR=1
Houston 2	1.297	1.35	TR=2, MR=1
Purdue	1.302	1.35	TR=2, MR=1

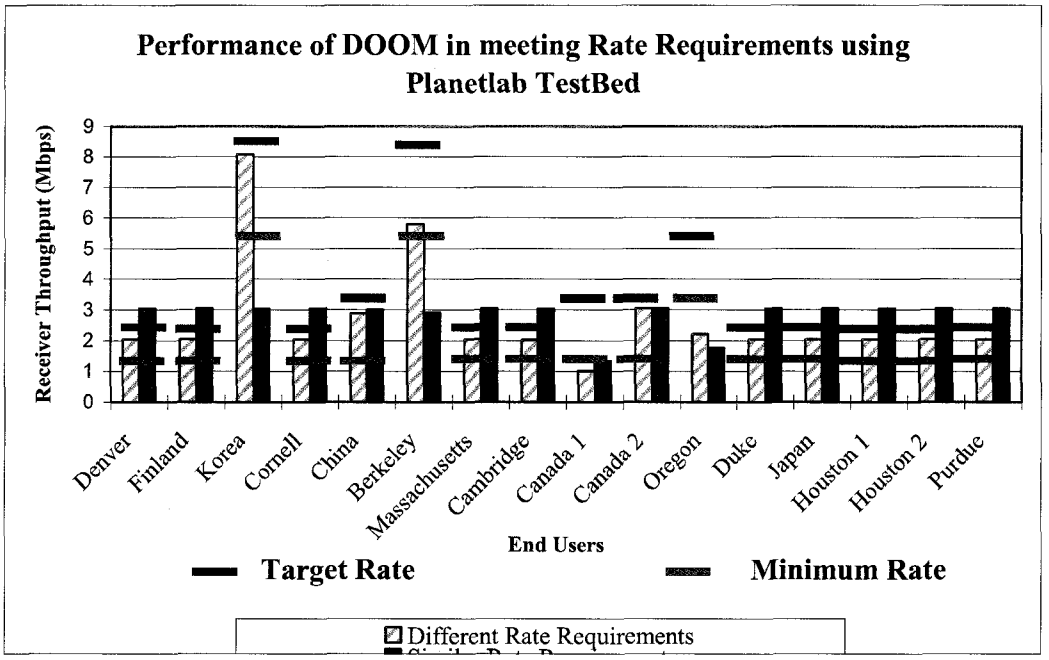


Figure 5.6 DOOM performance in meeting rate requirements of different end users with similar and different rate requirements (data generation rate = 10Mbps, heart-beat = 220ms). For similar rate requirement, TR = 3Mbps, MR=1Mbps

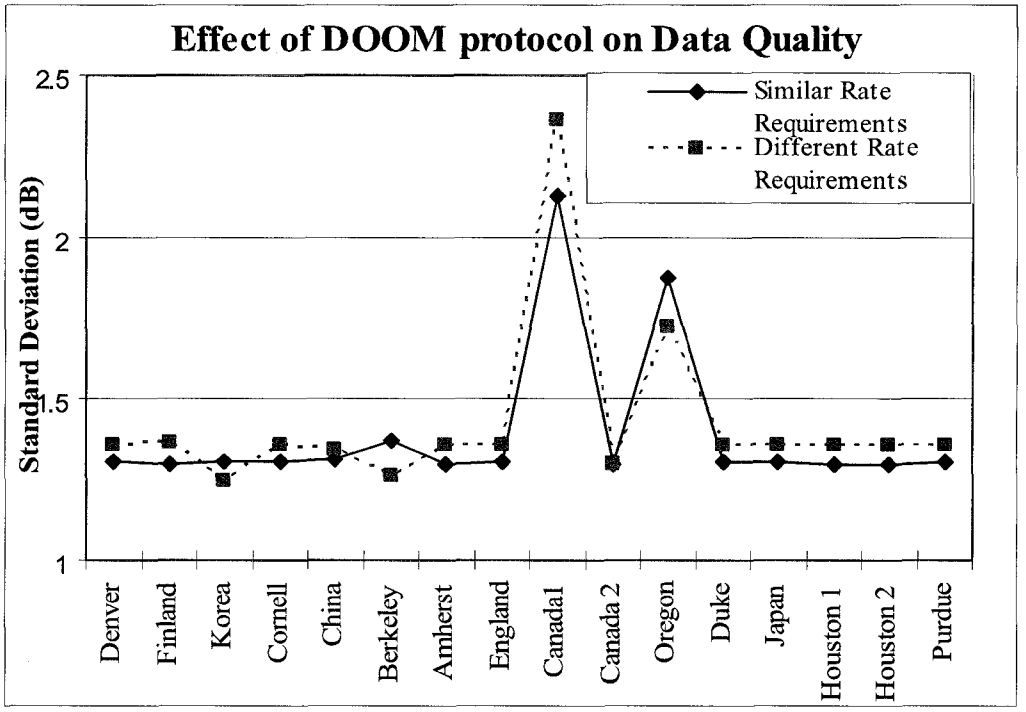


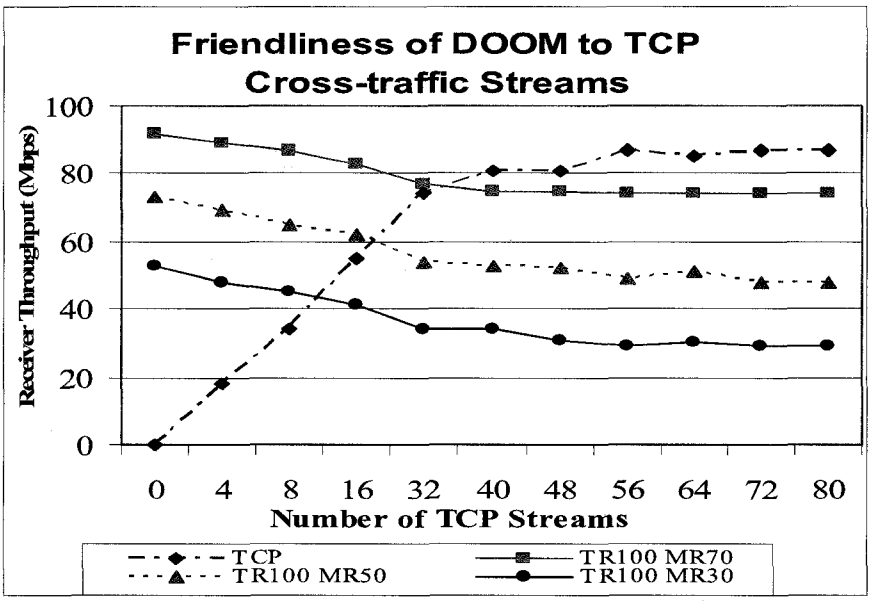
Figure 5.7 Effect of DOOM protocol on data quality for different end users with similar and different rate requirements

Table 5.5 TCP Friendliness of DOOM protocol when bottleneck bandwidth is 250Mbps

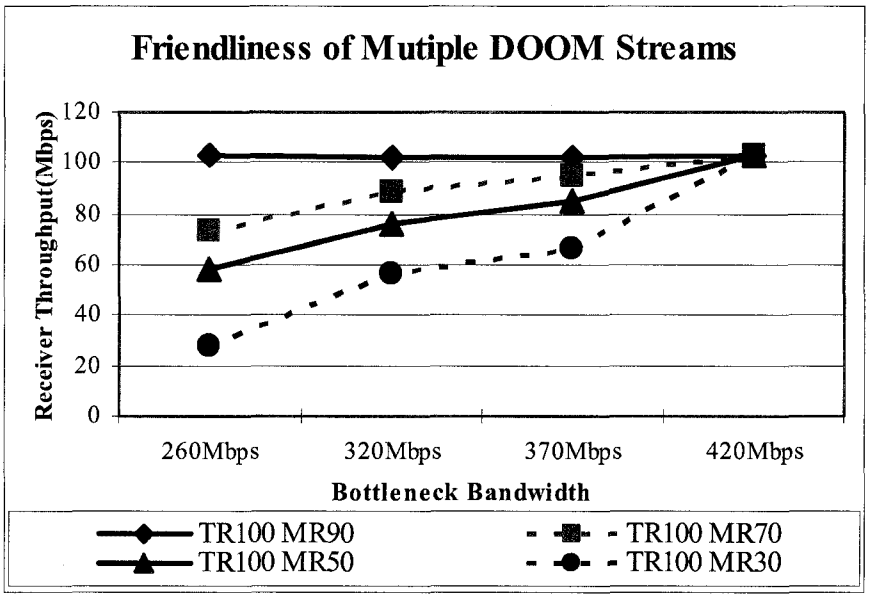
Num TCP Streams	*TCP Throughput (Mbps)*	*Stream 1 TR=100, MR=70 (Mbps)*	*Stream 2 TR=100, MR=50 (Mbps)*	*Stream 3 TR=100, MR=30 (Mbps)*
0	0	92	73	53
4	18	89	69	48
8	34	87	65	45
16	55	83	62	41
32	74	77	54	34
40	81	75	53	34
48	81	75	52	31
56	87	74	49	29
64	85	74	51	30
72	87	74	48	29
80	87	74	48	29

(referred as TR100) and different minimum rate requirement of 30Mbps, 50Mbps and 90Mbps (referred as MR30, MR50, and MR90) are considered. Bottleneck bandwidth is 250Mbps, which lies between sum of target rate requirements and sum of minimum rate requirements of all end users. As seen in Fig. 5.8(a), when there is no TCP cross traffic, all end users share the bottleneck bandwidth while satisfying their target rate and minimum rate requirements. As the number of TCP streams increases, receive throughput of all DOOM stream's sharing the link decrease and cumulative receive throughputs of TCP streams increases. Note that for each DOOM stream, throughput does not fall below the minimum rate requirements of individual end users.

It is possible that different DOOM streams may share the same bottleneck link. Fig. 5.8(b) shows the emulation test-bed results for the DOOM streams friendliness to each other under varying bottleneck bandwidth conditions. Table 5.6 shows data corresponding to results shown in



(a)



(b)

Figure 5.8 Friendliness of DOOM protocol (a) to TCP cross traffic streams (Bottleneck bandwidth = 250Mbps and RTT = 50ms) (b) to DOOM traffic streams sharing the same bottleneck link (RTT=10ms)

Fig. 5.8(b). Four end users with similar target rate and different minimum rate requirements are considered. Bottleneck bandwidth varies between 260Mbps and 420Mbps. When bottleneck bandwidth is 260Mbps, three users receive data at or above their minimum rate except the user with target rate of 100Mbps and minimum rate of 90Mbps, which is always receiving data at its target rate. As the bottleneck bandwidth increase, receive rate for the end users starts increasing. Note that throughput of all end users equally share any extra available bandwidth till the target rates of end users are achieved.

Fig. 5.9 and Fig. 5.10 illustrate DOOM protocol performance in concurrently meeting radar application specific data quality requirements of multiple end users. Fig. 5.9 shows the performance result when all end users have similar data quality requirement. Alternatively, Fig. 5.10 shows performance result when end users have different data quality requirement. Fig. 5.9(a) shows standard deviation in the end results when data from radar sensor is used for reflectivity computation by end users. Fig. 5.9(b) shows the standard deviation in end results when data is used for Doppler velocity computation. Table 5.7 shows data corresponding to results shown in Fig. 5.9(b). As mentioned before, in Fig. 5.9(a) and 5.9(b), all end users have similar data quality request, referred as Type 2 with uniform drop scheme. In Fig.5.10, multiple end users request for different quality of data for reflectivity computation. Table 5.8 shows data corresponding to results shown in Fig. 5.10. As seen in the figure, two end users request Type 1 with uniform drop scheme and two other user request Type 2 data with uniform drop scheme. Fig. 5.9(a) shows that as the bottleneck bandwidth increases, standard deviation in reflectivity for all users decreases, improving the quality of the end results. Though for some users change is so small to be noticeable. Similar conclusions can be made from the Fig. 5.9(b) for the Doppler velocity algorithm and here variation is more prominent for most users compared to Fig. 5.9(b). In Fig. 5.10 quality of receive data increases with increase in bottleneck bandwidth and also change is more prominent in this case for Type 1 data with uniform drop scheme.

Table 5.6 Fairness of DOOM streams to each other

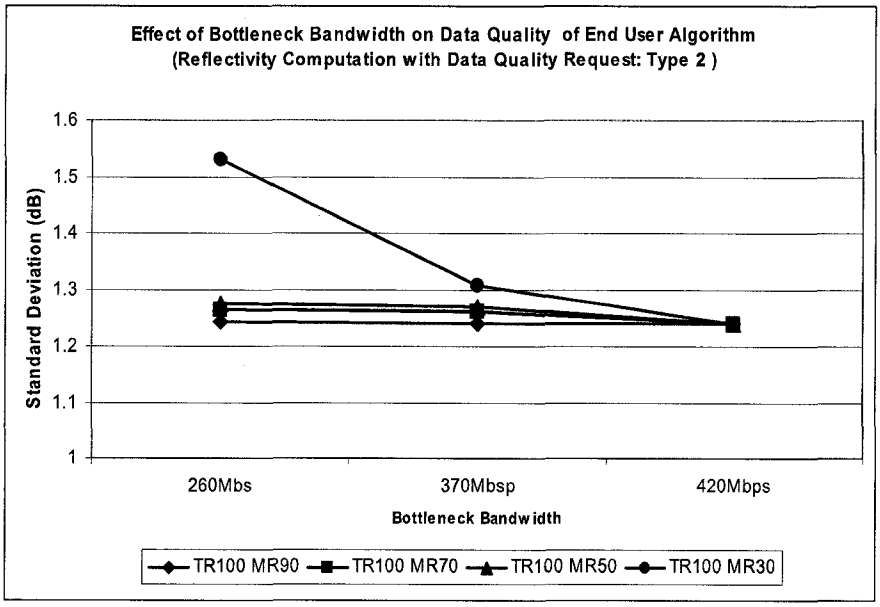
Bottleneck BW (Mbps)	*Stream 1 TR=100, MR=90 (Mbps)*	*Stream 2 TR=100, MR=70 (Mbps)*	*Stream 3 TR=100, MR=50 (Mbps)*	*Stream 4 TR=100, MR=30 (Mbps)*
260	103	73	58	28
320	102	89	76	56
370	102	95	85	66
420	103	103	103	103

Table 5.7 Effect of bottleneck bandwidth on quality of time-series data delivered to end user for Doppler velocity computation (All end users have Type 2 with uniform drop data requirement). Standard deviation in the moment parameters is used to measure the quality of the data.

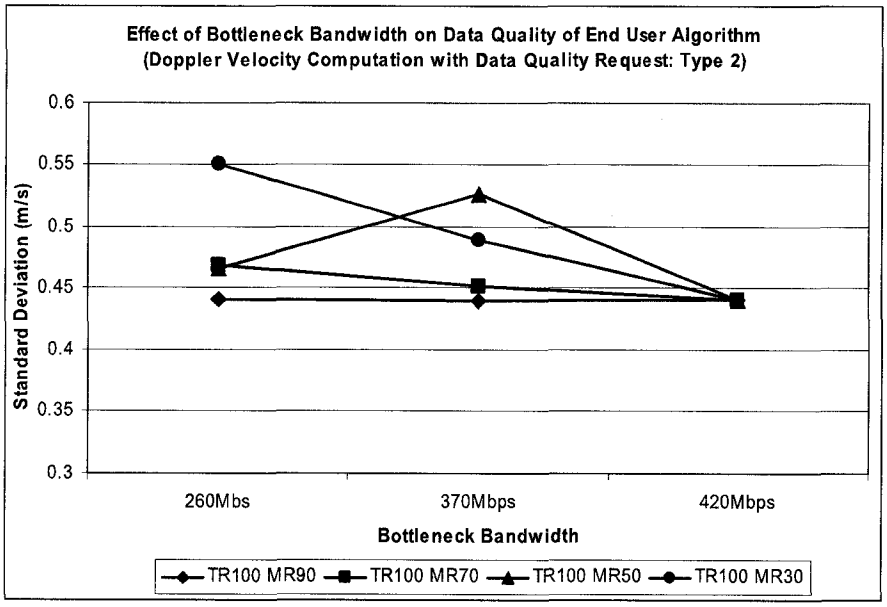
Bottleneck BW	*Stream 1 TR=100, MR=90 (dBz)*	*Stream 2 TR=100, MR=70 (dBz)*	*Stream 3 TR=100, MR=50 (dBz)*	*Stream 4 TR=100, MR=30 (dBz)*
260	0.44	0.46	0.46	0.55
370	0.44	0.45	0.52	0.48
420	0.44	0.44	0.44	0.44

Table 5.8 Effect of bottleneck bandwidth on quality of time-series data delivered to end user for Reflectivity Computation (Stream 1 and Stream 2 have Type 2 with uniform drop data requirement, Stream 3 and Stream 4 have Type 1 with uniform drop requirement). Standard deviation in the moment parameters is used to measure the quality of the data.

Bottleneck BW	*Stream 1 TR=100, MR=90 (dBz)*	*Stream 2 TR=100, MR=70 (dBz)*	*Stream 3 TR=100, MR=50 (dBz)*	*Stream 4 TR=100, MR=30 (dBz)*
260	1.24	1.26	1.38	1.46
370	1.24	1.24	1.25	1.26
420	1.24	1.24	.124	1.24



(a)



(b)

Figure 5.9 DOOM performance in meeting similar data quality requirements of multiple end users under varying bottleneck bandwidth conditions (a) When four end users have similar Type 2 and uniform drop data quality requirement for reflectivity computation algorithm, (b) When four end users have similar Type 2 and uniform drop data quality request for Doppler velocity computation

Fig. 5.5, Fig. 5.6, and Fig. 5.8 shows that DOOM is able to meet the heterogeneous rate requirements of the multiple users over the long duration. In order to see instantaneous behavior of the protocol, receive rate of blocks of data generated by sensor every *heart-beat* interval are monitored. Table 5.9 shows instantaneous throughput performance results of the DOOM protocol using emulation testbed. Two end users with similar target rate, 80Mbps each, and minimum rate requirement, 20Mbps each are considered. When bottleneck bandwidth is close to sum of the minimum rate requirements of both end users, i.e., 55Mbps, then receive rate per data block (e.g. DRS block for radar sensor) of sensor is concentrated near the minimum rate for both the users with low standard deviation. As the bandwidth starts increasing, i.e., 105, 155, 205Mbps then both users have same average throughput with similar standard deviation. These results are able to show that when multiple end users have similar QoS requirements and share common bottleneck link, then both have similar instantaneous real-time performance when served concurrently by the DOOM server.

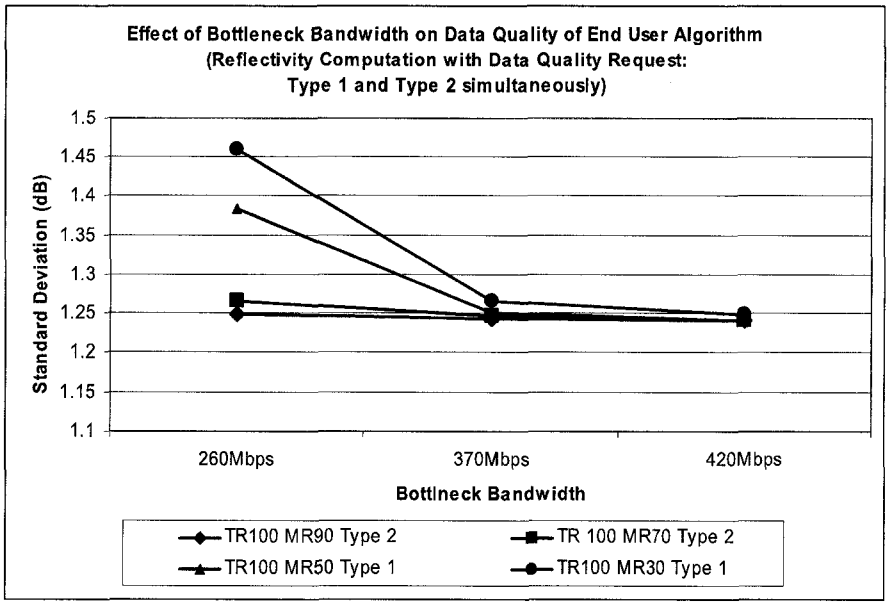


Figure 5.10 Performance of DOOM in meeting different data quality requirements when four users have different Type 1 and Type 2 with uniform drop data quality requirements for reflectivity computation

Table 5.9 Instantaneous throughput performance (TR = 80Mbps and MR = 20Mbps)

Bottleneck Bandwidth (Mbps)	*RTT (ms)*	*Receiver Throughput (Mbps)*	*Standard Deviation (Mbps)*
55	25	26.7	5.8
	150	28.5	5.9
105	25	37.5	12.1
	150	39.3	14.4
155	25	71.1	20.4
	150	61.4	22.8
205	25	83.1	0.1
	150	83.1	0.1

In DCAS systems some end users may have very high bandwidth requirements; therefore it is important to study the performance of DOOM overlay node server under varying load conditions. Experiments are performed in an emulation test bed to evaluate DOOM server load handling performance for variable number of end user requests. Performance results are shown in Fig. 5.11. In this case number of end users varies from 2 to 12 with similar target rate and minimum rate requirements. In Fig. 5.11, cumulative target rate requests for all users vary from 120Mbps to 720Mbps. In this case bottleneck bandwidth is a Gigabit link. As seen in the figure, average throughput of all users for different number of user requests remains close to their target rate requirement of 60Mbps. Thus this experiment is able to demonstrate that DOOM overlay node does not become a bottleneck and is able to handle cumulative high bandwidth load while satisfying application specific data quality needs.

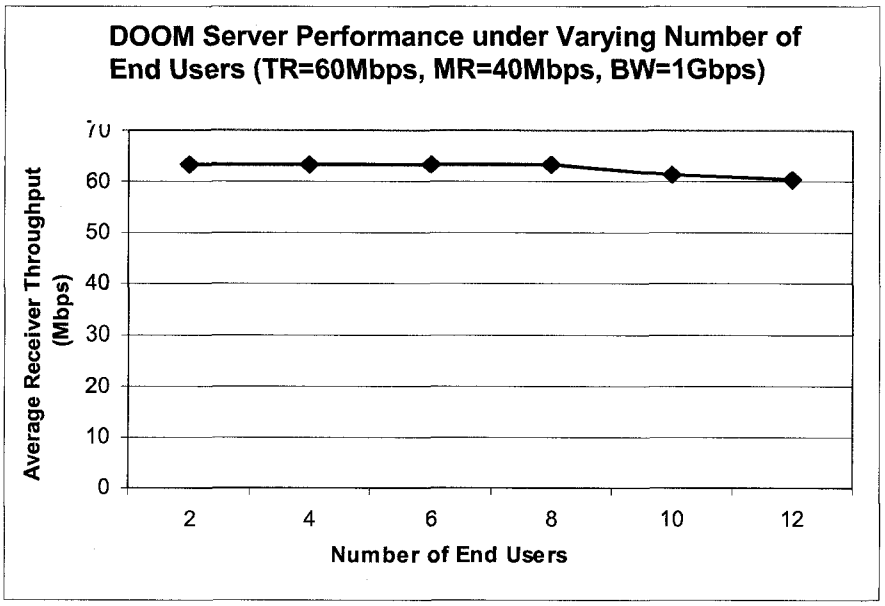


Figure 5.11 Performance of DOOM server under variable number of users

5.3 Remarks

An overlay DOOM protocol for One-to-Many data dissemination of high-bandwidth radar data is presented. DOOM protocol concurrently meets heterogeneous real-time and data quality requirements of multiple end users under diverse network congestion condition. A time multiplexed data scheduling scheme is proposed and is integrated with TRABOL based congestion control protocol for meeting heterogeneous end user requirements. Performance results using planetlab test bed and emulation test bed show effectiveness of DOOM in meeting diverse requirements of multiple end users under dynamic network and load conditions. DOOM protocol is shown to be TCP friendly as long as requirements of all end users are satisfied. Experiments results show that DOOM protocol is able to handle high bandwidth requirements of multiple end users without becoming a bottleneck.

Chapter 6

CONTENT-AWARE PACKET MARKING FOR APPLICATION-AWARE PROCESSING IN OVERLAY NETWORKS

Most of the prior work on application-aware data selection mechanisms relied on end-host applications to adapt to network conditions [An00, Zh99]. However, network conditions are dynamic and end-hosts cannot predict a change in the available bandwidth at intermediate nodes during the transmission leading to random losses of the subset of data selected by end-hosts. Thus we cannot assure that most important subset of the data will be delivered to the end-users by using the end-host application-aware data selection scheme alone. Intermediate nodes can perform congestion control within the network by performing selective drop/forwarding that may enhances the probability of delivering application-specific critical packets to end users during network congestion. Overlay networks enable deployment of such application-aware data dissemination services over the Internet [Su04]. Fig. 6.1 shows an overlay network for application-aware data dissemination to multiple end users. An overlay network consists of different nodes, such as forwarding nodes, multicast nodes, and fusion nodes, each configured to perform application-specific tasks to best meet QoS requirements of different end users. For example, a source node may perform packet marking based on the properties of the data for a particular application.

In Fig. 6.1, the multicast node is responsible for accepting connection requests from multiple

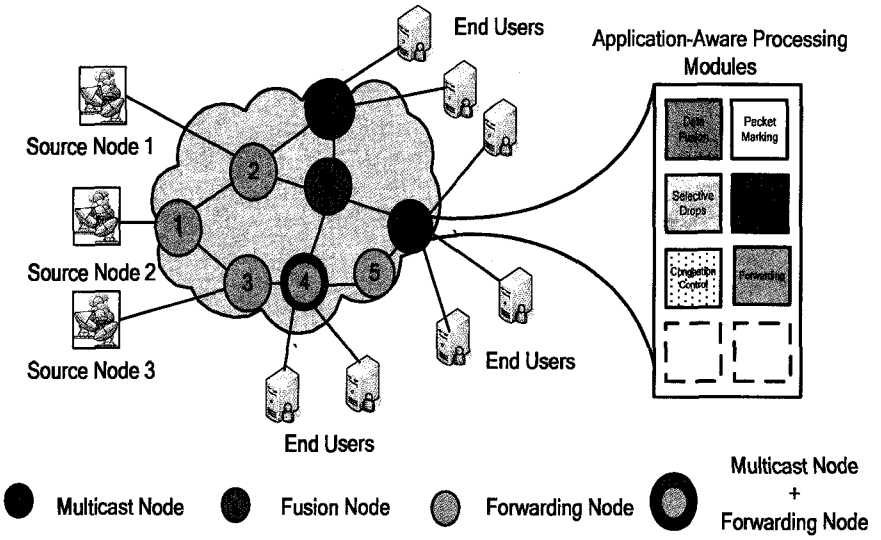


Figure 6.1 Overlay network for application-aware data dissemination

end users with different data quality and bandwidth requirements. Multicast node then forwards the aggregated request to the source node. Source node sends data to the multicast node as per the requested bandwidth and data quality requirements. It is important to note that in DCAS applications, a multicast node may simultaneously send different subset of the data to each end user as per their bandwidth and data quality requirements. The multicast nodes may perform functions such as independent flow and congestion control for each end user in an application-aware manner considering their distinct bandwidth requirements and fusion of data from different sources prior to multicast. Existing multicast solutions such as RLM [Mc96], are required to scale to millions of end users, which is significantly higher than the scalability requirements of the DCAS applications. Moreover, unlike RLM, each of the end user in DCAS may have distinct bandwidth and data quality requirements that need to be satisfied by a single multicast server. A token-bucket based rate control may be implemented at the multicast node to achieve the desired rate for a particular end user under existing network conditions. Alternatively, forwarding nodes can use packet marking to select packets for forwarding or drop while considering available

output link bandwidth. A forwarding node can be considered as a special case of multicast node where only one end user is requesting data at a rate equal to the available output link bandwidth.

In this chapter we propose an application-aware content based packet marking and a token-bucket based rate control algorithm to meet content quality and bandwidth requirements of the DCAS applications using overlay networks. The proposed algorithm enables transfer of most suitable subset of the data at the intermediate overlay nodes to the end user while providing soft bandwidth guarantees within bounds under available network infrastructure and dynamic network congestion conditions. We also demonstrate the effectiveness of the proposed scheme in overlay multicast where multiple end users receive weather radar data with different QoS requirements.

Section 6.1 describes the proposed application aware packet-marking scheme. Section 6.2 shows packet marking for weather radar data. Section 6.3 describes the application of packet-marking in token-bucket based application-aware rate control algorithm. Section 6.4 discusses experimental results. Concluding remarks about this chapter are presented in Section 6.5.

6.1 Application-Aware Packet-Marking

A packet marking scheme presented in [Le06] may be used at the intermediate overlay nodes to select packets for forwarding such that subset of the data delivered to the end users meets their data quality requirements. It is considered that the source node is aware of the properties of the data generated by sensors such as cameras and weather radars. The source node marks packets at transmission time according to the end users data quality requirements and different rates at which data may be delivered to the end users.

Consider an example as shown in Fig. 6.2(a), where a sensor node generates 8 application data units (ADU) within the bounded time at rate R1. The ADU is defined as a fundamental

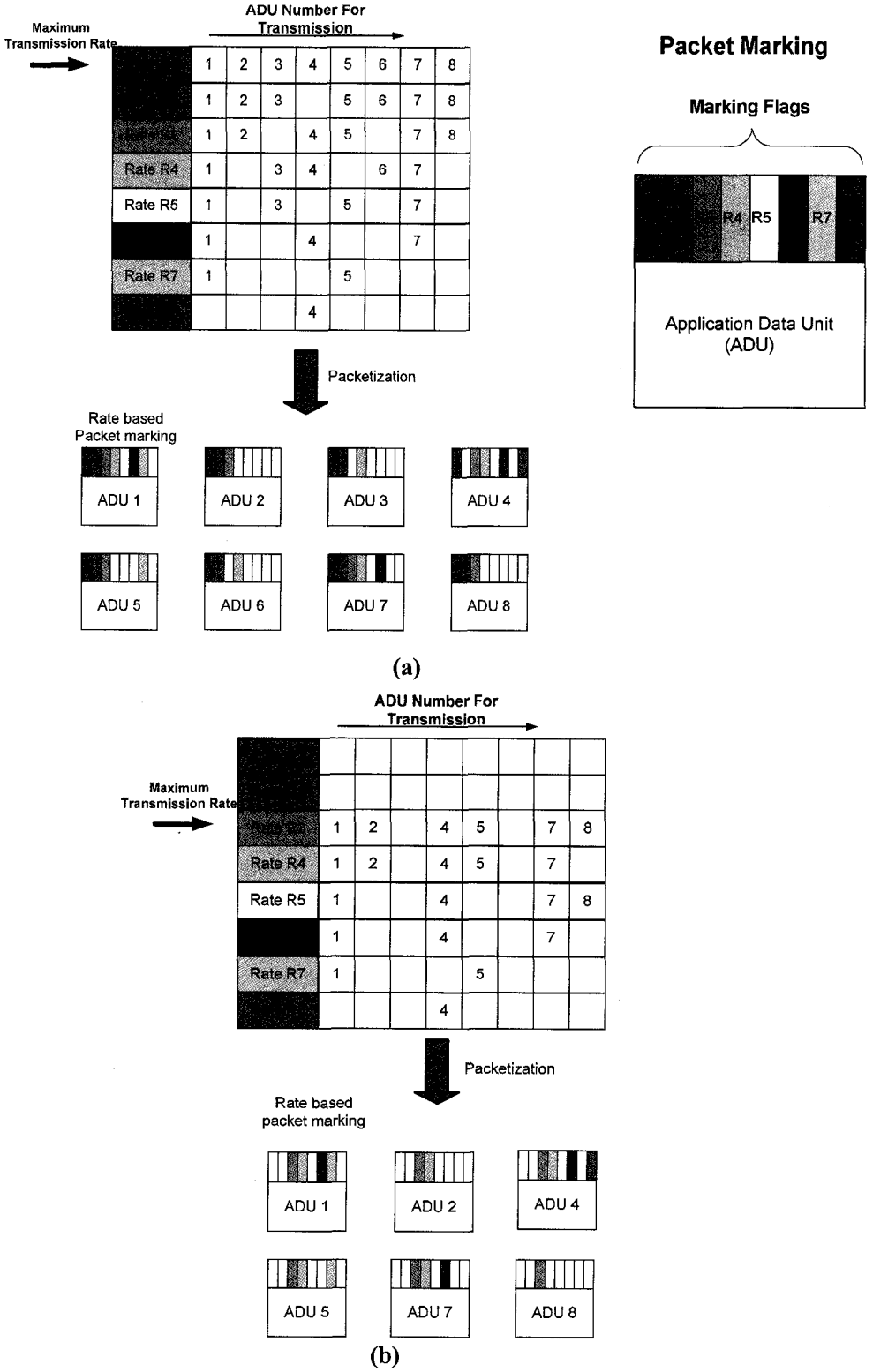


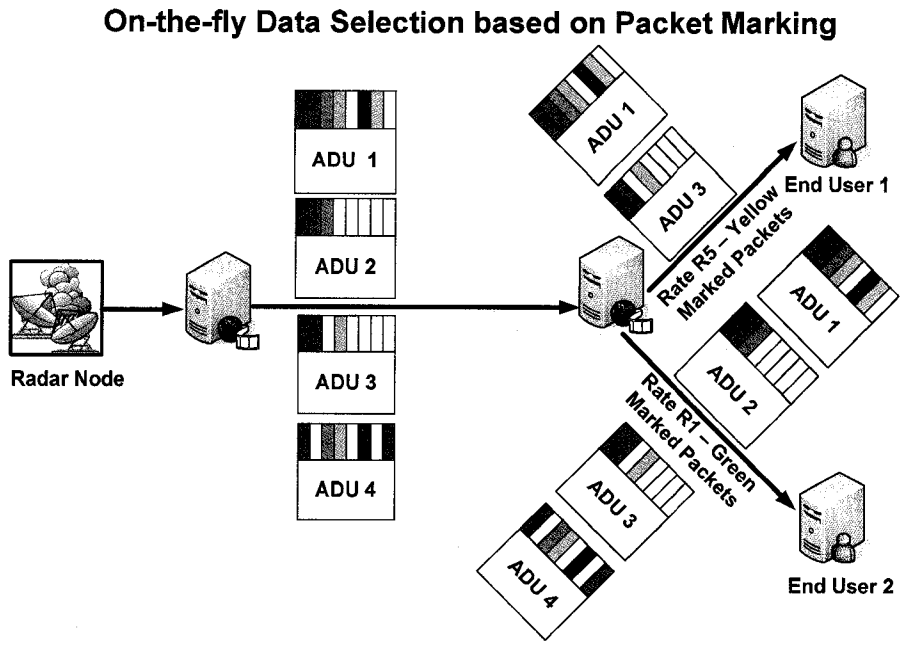
Figure 6.2 Rate based packet marking. Each non-white color represent rate for which packet is marked ,i.e., rate R1-R8 (a) Marking of packets when maximum transmission rate is R1, (b) Marking of packets when maximum transmission rate is R3

application data entity that can be used by an end user algorithm for processing. Each row in Fig. 6.2(a) and Fig. 6.2 (b) shows the subset of ADUs that are selected for transmission at a lower transmission rate when a higher rate cannot be supported because of bandwidth constraints. The subset of data selected at lower rate depends on the end user data quality requirements. For example, certain end users need uniformly spaced ADUs when only a subset of the data can be selected for transmission. Alternatively, other end users prefer a contiguous group of ADUs when bandwidth is constrained. Fig. 6.2 (a) considers the case when the source node transmits data at rate R1, and as seen in the figure the data transmitted at lower rates is a subset of the data transmitted at rate R1 and ADUs are selected that are spaced uniformly within a block of data at lower rates. Each packet consist of multiple of flag bits shown in Fig. 6.2 (a) corresponding to bandwidth at which the data included in the packet is most appropriate for transmission for a given application. The packet containing ADU 1 is marked with different color flag bits corresponding to different rates, i.e., rates R1-R7. Similarly packet containing ADU 3 is marked with different color flag bits, i.e., flag bits corresponding to different rates, i.e., R1, R2, R4, and R5 flag bits are set as indicated by different colors. As shown in the Fig. 6.2, every packet contains a flag for each rate for which it is transmitted indicated by different colors. Note that multiple flags can be set to indicate suitability of the packet for multiple transmission rates. It is important to note that packets are encoded at the time of transmission such that there is no dependency between different packets in the stream; end users can decode the packets on-the-fly without waiting for later packets to arrive. Following are the two key advantages of the packet marking when used at intermediate nodes:

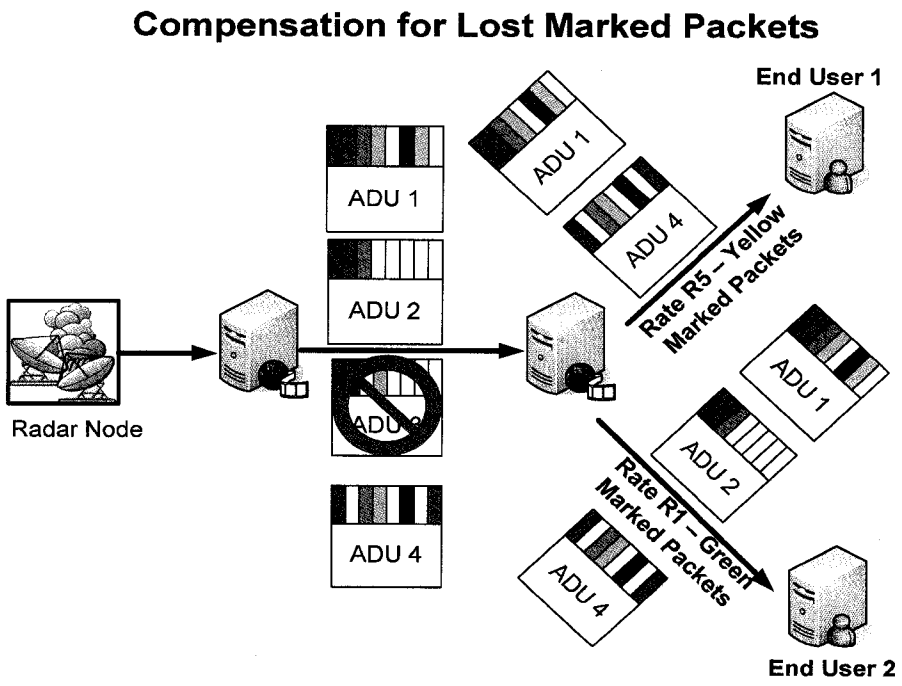
(i) On-the-fly selection of packets to one or more end users for forwarding at different rates: Consider the case when marked packets are received at the multicast node from the source node. In this case multicast node may select data on-the-fly for forwarding using packet marking to multiple end users at their respective transmission rates which are determined based on the network congestion for each end user. Note that in DCAS applications, multicast node may send

different subset of the received data to each end user concurrently while considering their individual bandwidth requirements. Packets with marking corresponding to end user's transmission rate are selected for forwarding for a particular end user. Fig. 6.3(a) shows a case when multicast node receives all packets transmitted at rate R1 from the source node. First four packets are shown in the figure. Two end users, i.e., end user 1, and end user 2 are considered that need data at different rates, i.e., rate R5 and rate R1 respectively. As seen in Fig. 6.3(a), in packets corresponding to rate R1 packets, flag bit is set corresponding to rate R1 is set as indicated by the green color and in packets corresponding to rate R5, flag bit corresponding to R5 is set as indicated by yellow color. In this case, packets corresponding to rate R5 are a subset of the packets transmitted at rate R1. Multicast node selects packets on-the-fly for forwarding to end users 1 and 2 based on the marking flag corresponding to rate R1 and R5 in the packet. Note that multicast node creates a copy of the packet to be forwarded and replaces destination address to the address of an end user for which it is selected for transmission. As seen in Fig. 6.3(a), out of first 4 packets received at multicast node, packets with ADU 1 and 3 are forwarded to end user 1 and packets with ADUs 1-4 are forwarded to end user 2.

***(ii) On-the-fly compensation for missing marked packets to maintain receiver data quality*:** It is possible that some of the packets with desired marking are dropped or suffers significant delay in the network when data is sent from a source node to the multicast node for further distribution. If the multicast node further distributes the partial data then this has the potential to degrade the performance of the end user application. Therefore it is desirable to compensate for the missing packets by selecting packets with the markings corresponding to higher rates than the current transmission rate for a particular end user. When the number of missing packets exceeds some application-specific threshold then multicast node initiate compensation process by selecting packets with marking corresponding to higher transmission rates for forwarding for a particular end user. For example, in Fig. 6.3(b), a packet with ADU 3 is lost. In this case multicast node may not meet the rate R1 and rate R5 requirements for two end users as both rates



(a)



(b)

Figure 6.3 Applications of packet marking. (a) On-the-fly data selection based on packet marking (b) On-the-fly compensation for missing marked packets to meet bandwidth and data quality requirements [Le06]

need packet with ADU 3. Alternatively, to meet rate R5, multicast node can select packets for forwarding with marking corresponding to rate higher than rate R5, i.e., packets with marking corresponding to any rate between rate R1 and rate R4 may be selected for forwarding to compensate for the missing sample. In Fig. 6.3(b), the multicast node decides to compensate for the missing packet with ADU 3 by selecting a packet with ADU 4 who's marking corresponds to rate R1, rate R3, and rate R4 as indicated by three set flag bits. Note that compensation for the missing packets with the desired marking is performed only for packets with rates lower than the rate at which data is transmitted by the source node. By performing compensation within the network at intermediate nodes, retransmission of the data may be avoided and thus associated delay is reduced. Note that it is assumed that all nodes that perform marking based packet selection are aware of rate to marking mapping. Each node maintains a static table of flags and the corresponding rate each flag represents in the packet. Subsequent performance results show that little computation penalty paid for performing application-aware processing at each node helps in significantly enhancing the quality of the content delivered to the end users during network congestion. Next section explains the application of this packet marking strategy for streaming weather radar data in CASA environment.

6.2. Packet-Marking for Radar Data – An Example

Fig. 6.4 shows the packet marking of the DRS block of data generated by a radar node. In this example it is assumed that radar data is generated at 10Mbps and current source transmission rate is 8Mbps. A subset of samples is selected at rate 8Mbps for transmission. It is assumed that end users can tolerate uniform loss of samples. Therefore, as seen in Fig. 6.4, sample drops are uniformly distributed among 64 samples generated by radar for different transmission rates. In Fig. 6.4, data may be transmitted at ten different rates between 1Mbps and 10Mbps

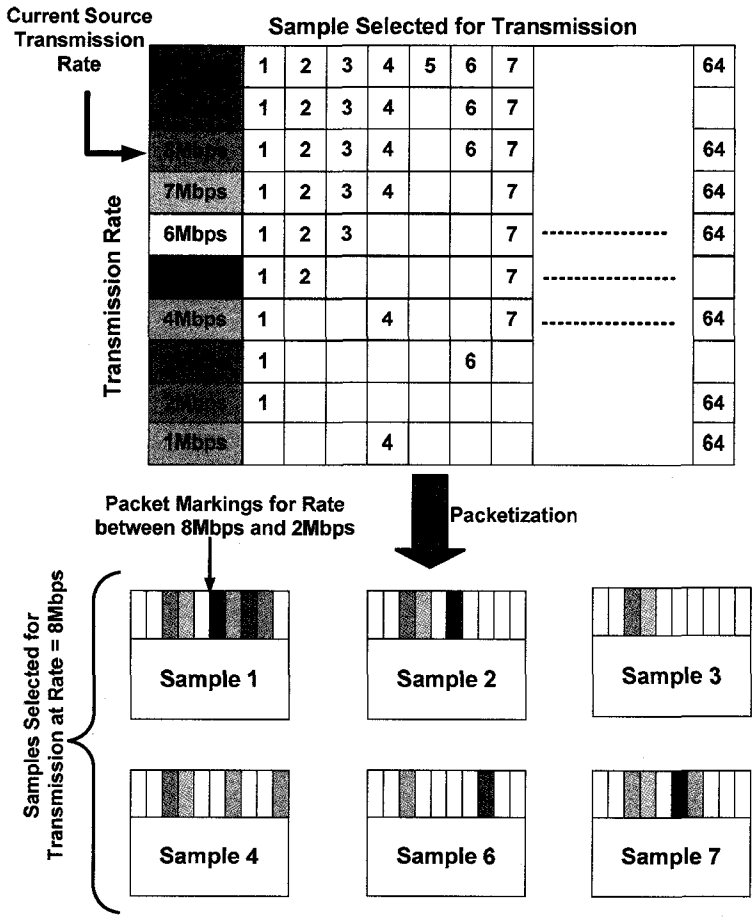


Figure 6.4 Packet marking for radar data when current transmission rate is 8Mbps. Sample is application data unit (ADU) for the radar data [Le06]

indicated by ten marking flags in each packet. Note that flag is marked when packet is suitable for the current transmission rate.

6.3 Applications of Packet-Marking

In Fig. 6.1, different overlay nodes use packet marking to perform application-aware processing within the network to meet the QoS requirements of the DCAS application. Following section describes the application of packet marking in multicast node and the forwarding node.

6.3.1. Token-bucket based rate control

At intermediate nodes such as multicast nodes, packets are selected for transmission based on the current transmission rate and the corresponding packing marking. A token bucket based rate control algorithm is used to achieve the desired transmission rate at these intermediate nodes.

6.3.1.1. Multicast Node: Fig. 6.5 shows architecture of the multicast node. In a CASA application, each end user may specify its rate requirement and content-quality requirement in terms of tolerance towards bursty losses or uniform losses within the DRS block at the time of request for the data. The rate at which the data is transmitted to the end users can vary from user to user based on the available bandwidth. A congestion control protocol determines the transmission rate for each end user independently. We consider TRABOL (TCP Friendly Rate Adaptation Based on Losses) congestion control protocol to determine transmission rate of each end user. During network congestion, TRABOL performs rate adaptation while considering end user specific minimum rate (MR) and target rate (TR) requirements [Ba05b]. When a packet with application-specific markings arrives at the multicast node, multicast node determines the users to which this packet must be transmitted. Decision to transmit a packet to a particular end user depends on the packet markings and the rate requirements of the end users. The multicast node maintains separate output queues for sending data to each end user. Fig. 6.5 shows a case for on-the-fly selection of the data for transfer to multiple end users when radar data shown in Fig. 6.4 is received at the multicast node. In Fig. 6.5, three end users, i.e., end user 1-3 are considered with current transmission rate of 8Mbps, 7Mbps, and 3Mbps respectively. Multicast node maintains the mapping between marking flag and the corresponding transmission rate. Note that in Fig. 6.5 the packet with sample 1 is forwarded to all three end users because packet is marked for transmission rate 8Mbps, 7Mbps, and 3Mbps, i.e., the current transmission rates of the end user 1, end user 2, and end user 3 respectively. Alternatively, the packet with sample 2 is not forwarded to end user 3 because the packet is not marked for transmission rate 3Mbps.

A token-bucket based scheme is used to maintain the required average transmission rate for

each end user. A token bucket size for each end user is determined periodically every 'heart-beat' interval based on its transmission rate. As explained in Chapter 4, *heart_beat* interval is the periodic interval after which radar node generates DRS block of data at a constant rate. We consider a case where 1 packet is removed from the end user output queue for transmission for every 1 token present in the bucket. Therefore, the token bucket size gives the upper bound on the number of packets that the multicast node can transmit during a *heart_beat* interval to a particular end user. Fig. 6.6 shows the token-bucket scheme and the packet compensation process to meet the current transmission rate and data quality requirement of the end users. Consider an example in Fig. 6.6 where 20 tokens present in the bucket for a given transmission rate within a *heart_beat* interval of 200ms are evenly distributed among 10 time slots of 20ms each. A counter is maintained to track if number of received packets with desired marking is equal to the expected number of packets within each time slot. If the number of packets received with the desired marking is less than the number of expected packets then it indicates that some of the desired

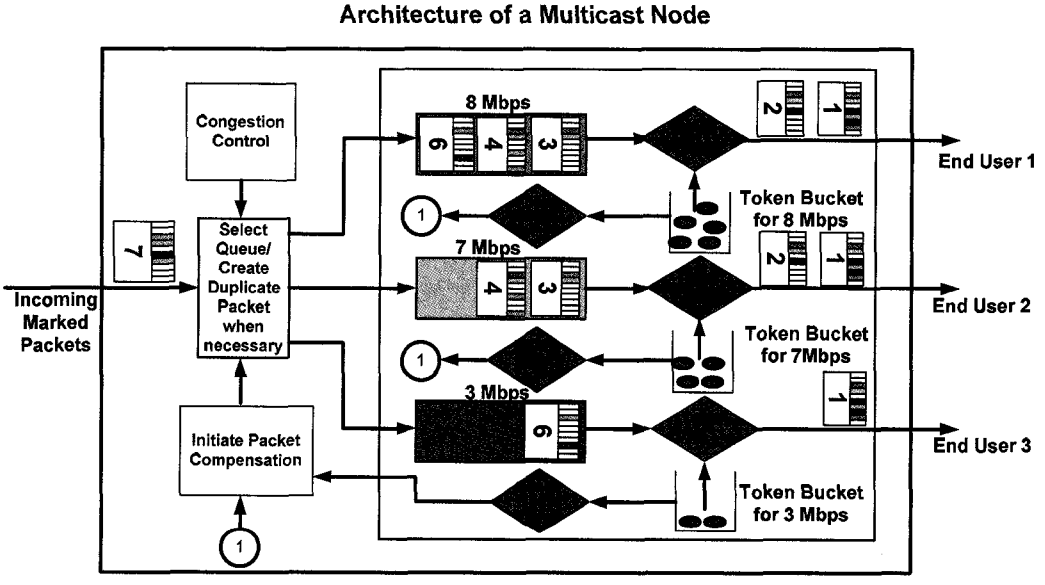


Figure 6.5 Architecture of a multicast node. Three end user's current transmission rates are 8Mbps, 7Mbps, and 3Mbps and arrival rate of the data at multicast node is 8Mbps. Initial number of tokens in the bucket depends on the end user current transmission rate. [Le06]

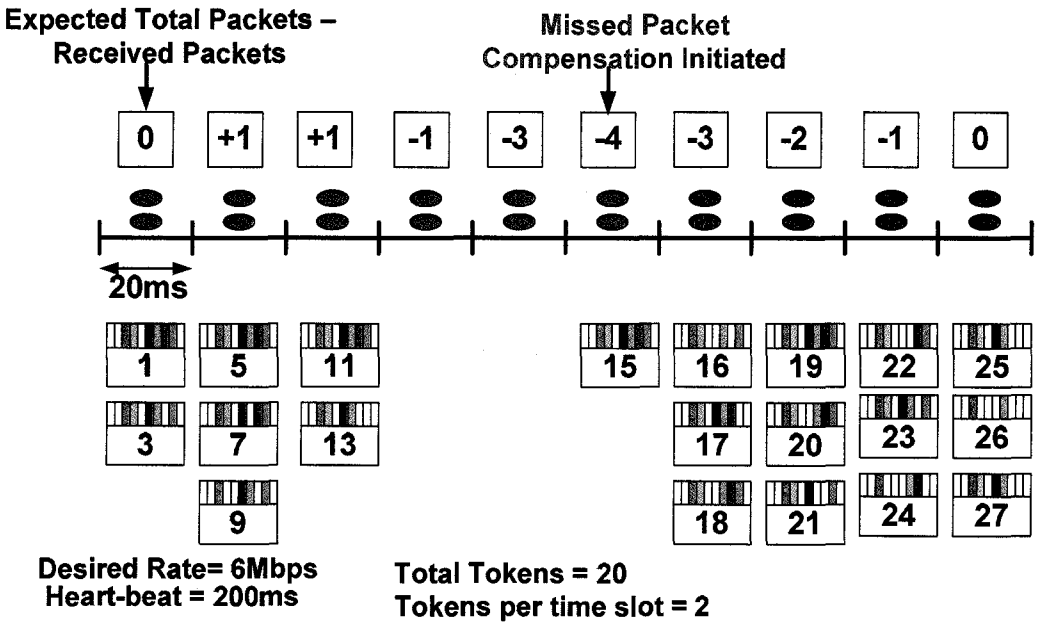


Figure 6.6 Token bucket based packet compensation to meet bandwidth and data quality requirement [Le06]

marked packets are lost or significantly delayed in the network. When this difference exceeds some threshold then it indicates the start of a compensation process. For example, in Fig. 6.6, compensation for missing packets start when the difference between arriving and expected packets falls below threshold (=-3). As seen in the figure, after compensation is initiated, packets with marking corresponding to a rate higher than the current transmission rate are used to meet the transmission rate requirements. At the end of 'heat-beat' interval, if the difference between total arriving packets and total expected packets becomes 0 then it indicate that compensation process succeeded in meeting the transmission rate requirements. More implementation details are presented in [Li06].

6.3.1.2. Forwarding Node: Fig. 6.1 shows forwarding nodes in the overlay path. Main task of the forwarding node is to select the appropriately marked packets for forwarding according to the available output link bandwidth. Consider a case, where a high-bandwidth upstream flow relays packets through a forwarding node to a low-bandwidth output link. In this case forwarding node

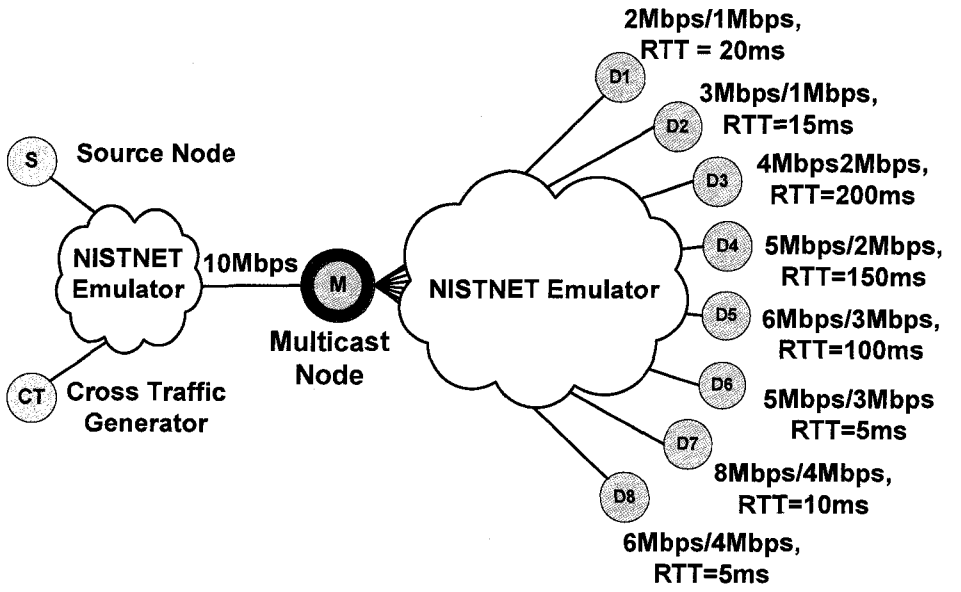


Figure 6.7 Emulation network for application-aware multicasting of weather radar data [Le06]

may either buffer or selectively discard packets received from the upstream node. The implementation of a forwarding node is similar to the implementation of a multicast node as explained in Fig. 6.5-6.6. However, forwarding node may determine output link bandwidth using bandwidth measuring tools instead of congestion control algorithm used by the multicast node.

6.4 Performance Evaluation

Performance of the packet-marking scheme and token-bucket based rate control algorithm is evaluated in a network emulation environment as shown in Fig. 6.7. The NISTNET [Ca03] based network emulator along with TCP cross-traffic is used to control the bandwidth between source node and the multicast node, and to control bandwidth between multicast node and end users. In all experiments, we consider the case where source node generates data at a constant rate of 10Mbps with different bandwidth requirements for different end users. The multicast node receives a single copy of the packet from the source node and is transmitted to multiple end users

with different round trip delays, and with different bandwidth and data quality requirements. Experiments were conducted to investigate: (i) Performance in meeting bandwidth requirements of end users, (ii) Impact of packet-marking scheme on data quality, and (iii) Impact of packet compensation algorithm on data quality.

The first set of experiments evaluates the effectiveness of the proposed packet-marking scheme along with token-bucket rate control scheme in meeting bandwidth requirements of the end users. As seen in Fig. 6.8, when bottleneck is 30Mbps, which lies between sum of target rate requirements and sum of minimum rate requirements of all the end users, then all end users shares the bandwidth while meeting their bandwidth constraints. Table 6.1 shows data corresponding to results shown in Fig. 6.8. Next experiment results investigate the effect of application-aware selective drop and packet-marking on data quality of the end users. Eight different end users with different RTT requests multicast node for the data with their data quality and bandwidth requirements. In this experiment, the multicast node combines the bandwidth requests of all end user and makes an aggregate request to the source node with target rate (TR) = 8Mbps, minimum rate (MR) = 4Mbps requirement. Source node initially selects subset of the data for transmission at 8Mbps from the DRS block. We consider random sample drop and selective sample drop as the sample selection scheme at the source node for determining subset of data for transmission at rates lower than the data generation rates [Ba05b,Ba06]. In the selective drop, the source node considers end user's sensitivity to different components of the DRS block while selecting a subset of samples according to the current transmission rate for a multicast node as explained in Chapter 4. Alternatively, in the random drop case; the source node randomly selects subset of the samples from the DRS block for a given transmission rate.

Experiments were conducted to compare quality of the data delivered to the end users when packet-marking is used, i.e., packet-marking based forwarding to the case when no packet marking is used, i.e., random forwarding. In the packet-marking based forwarding, the source node mark packets as explained in Section 6.3 and the packet marking enables the overlay

Table 6.1 Impact of different degree of application-aware processing on the throughput of multiple end users with different bandwidth requirement (RD: Random Drop, RF: Random Forwarding, SD: Selective Drop, SF: Selective Forwarding [Le06]

End User Id	*TR (Mbps)*	*MR (Mbps)*	*RD, RF Receiver Throughput (Mbps)*	*SD, RF Receiver Throughput (Mbps)*	*SD,SF Receiver Throughput (Mbps)*
D1	2	1	1.61	1.61	1.61
D2	3	2	2.64	2.66	2.63
D3	4	2	2.93	2.9	2.93
D4	5	3	4.01	4.02	4.01
D5	6	3	4.78	4.76	4.78
D6	5	2	3.26	3.26	3.25
D7	8	4	6.16	6.15	6.19
D8	6	3	4.14	4.16	4.16

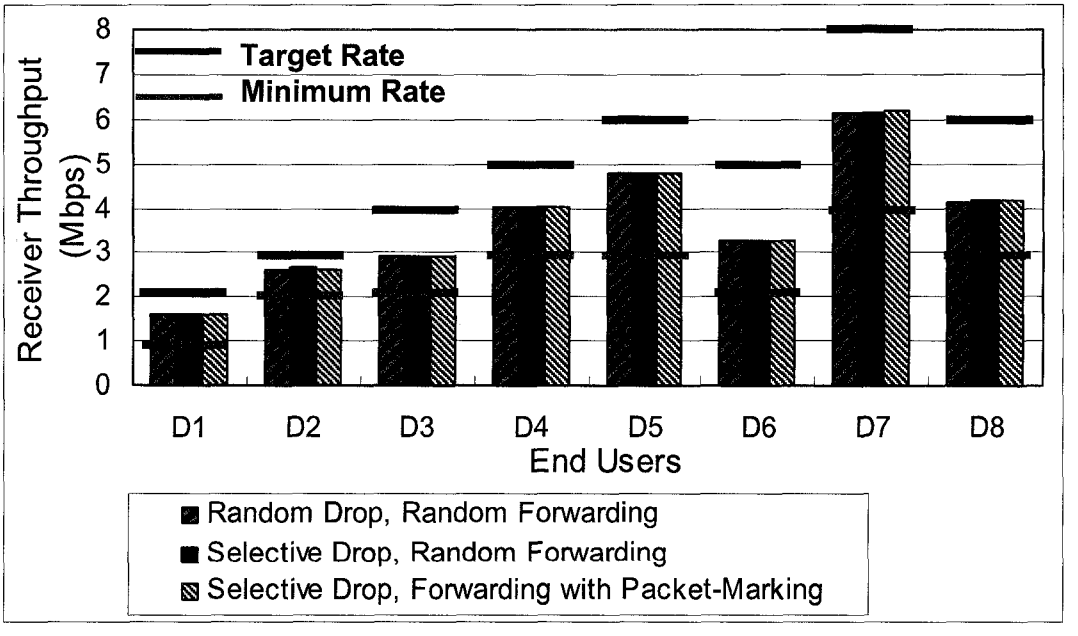


Figure 6.8 Receiver throughputs of the end users with different rate requirements [Le06]

multicast node to determine packets to be forwarded to the each end user independently based on the available bandwidth for each end user. In the random forwarding case packets are not marked; packets that arrive at the multicast node are transmitted to an end user in FIFO basis as long as there are tokens present in the bucket within 'heart-beat' interval. We compare the performance for three cases based on how data is dropped at the source node and whether packet-marking is supported or not: (1) Random drop, Random forwarding, (2) Selective drop, Random forwarding, and (3) Selective drop, Packet-marking based forwarding.

In case of weather radar data, quality of the received raw data is measured by computing standard deviation in the reflectivity parameter [Ba05b] for each end user. Lower standard deviation is a measure of better quality data [Ba05b, Ba06]. Fig. 6.9 shows the standard deviation of reflectivity parameter of 10 gates (141 ~ 150) of one end user with TR=4Mbps, MR=2Mbps. Table 6.2 shows data corresponding to results shown in Fig. 6.9. We compare the results with the baseline case in which all samples generated by a radar node is delivered to the end user application. As seen in Fig. 6.9, standard deviation is minimum for the baseline case and is highest for a case when data is randomly dropped at source node and no packet-marking is supported. Alternatively, quality of the data improves, i.e., lower standard deviation, with selective drop and packet-marking support. Improvement in data quality at end user is due to end user receiving the required subset of the data at any given transmission rate.

Experiments are conducted to study the effect of the packet compensation scheme on the receiver throughput and data quality of the end users. Bandwidth between source node and multicast node is configured as 10Mbps using NIST Net. TCP cross-traffic is used to introduce random losses in the network between source node and the multicast node. Initially source node transmits marked packets with radar data to the multicast node at 8Mbps but due to competing TCP cross-traffic, random radar data packet may get lost between source node and the multicast node. Alternatively, bandwidth between multicast node and end users is sufficient to support the target rate all end users thus no losses are encountered in network between multicast node and the

Table 6.2 Impact of different degree of application-aware processing on the quality of the time-series data. Standard deviation in moment parameters is used for data quality estimation [Le06]

Gates	*RD, RF Standard Deviation (dBz)*	*SD, RF Standard Deviation (dBz)*	*SD, SF Standard Deviation (dBz)*	*No Loss Standard Deviation (dBz)*
141	2.35	2.14	1.75	1.52
142	2.60	2.35	1.89	1.70
143	3.30	2.92	2.46	2.20
144	2.74	2.48	2.00	1.78
145	2.81	2.48	2.03	1.79
146	2.66	2.46	1.93	1.68
147	2.80	2.56	2.09	1.92
148	2.44	2.20	1.93	1.72
149	1.93	1.71	1.41	1.18
150	1.60	1.49	1.20	0.97

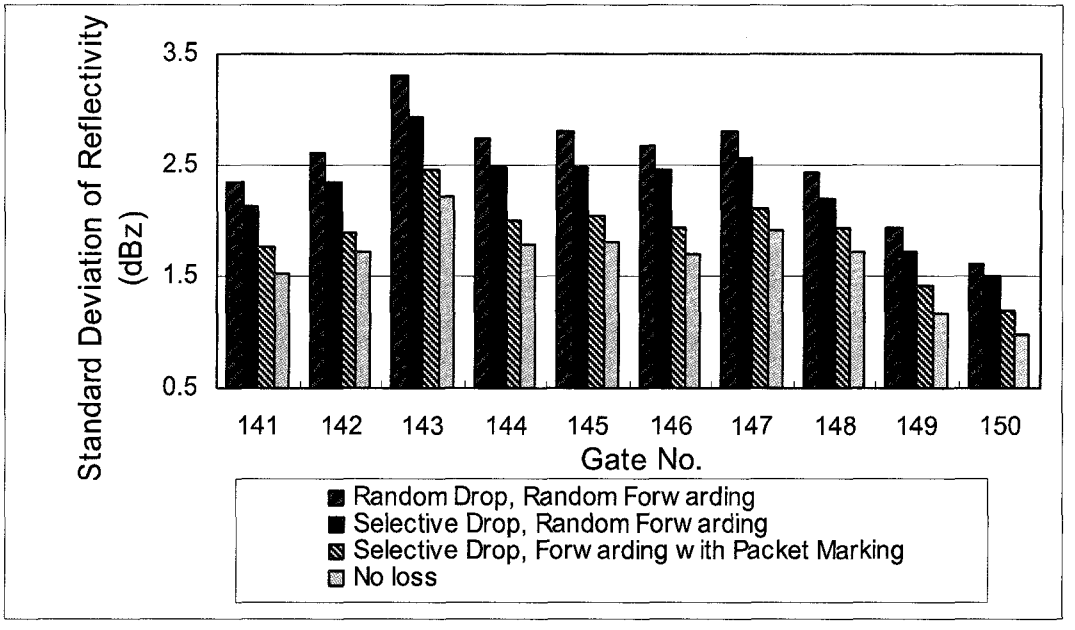


Figure 6.9 Effect of the application-aware selective drop and packet-marking on data quality for the end user with TR=4Mpbs, MR=2Mbps [Le06]

Table 6.3 Impact of packet marking based compensation scheme on the receiver throughput for multiple end users. (TR, MR, and Receiver throughput in Mbps) [Le06]

Number of TCP Streams	*0*	*4*	*32*	*64*	*84*	*96*	*128*
TR=3, MR=2 With No Compensation	3.13	2.98	2.82	2.69	2.2	1.96	1.54
TR=3, MR=2 With Compensation	3.15	3.15	3.15	3.15	3.15	3.12	2.98
TR=5, MR=3 With No Compensation	5.21	4.97	4.6	4.45	3.73	3.37	2.67
TR=5, MR=3 With Compensation	5.27	5.27	5.23	5.14	4.45	4	3.4
TR=6, MR=3 With No Compensation	6.23	5.94	5.48	5.3	4.53	4.12	3.28
TR=6, MR=3 With Compensation	6.30	6.30	6.15	5.97	5.06	4.52	3.79
TR=8, MR=4 With No Compensation	8.21	7.45	6.35	6.11	5.14	4.64	3.65
TR=8, MR=4 With Compensation	8.26	7.40	6.35	6.13	5.15	4.58	3.81

end user. Performance is compared when marked data is transmitted using packet compensation and without any compensation. In Fig. 6.10, dotted lines show receiver throughput of the end users without any packet compensation. In Fig. 6.10, solid lines show end users receiver throughput using packet compensation scheme. Table 6.3 shows the data corresponding to results shown in Fig. 6.10. As seen in the figure, when network losses increase due to increase in TCP cross-traffic, the throughput of end users decreases. However, when the packet compensation is used end user receives higher throughput compared to no compensation case. Since TR=8 Mbps, MR=4 Mbps end user receives data at the highest rate, the multicast node cannot find any higher stream to compensate for the missing packets, so no compensation is performed for that particular

end user. Fig. 6.11 shows the impact of the packet compensation scheme on the quality of the data received by the end users for different gates. Fig. 6.11 shows standard deviations of the reflectivity parameter for two end users with bandwidth requirements TR=6 Mbps, MR=3 Mbps, and TR=3 Mbps, MR=2 Mbps. Table 6.4 shows data corresponding to results shown in Fig. 6.11. As seen in the figure data quality improves when packet compensation is used at the multicasting node indicated by decrease in standard deviation in presence of packet compensation.

6.5 Remarks

This chapter presented content-based packet marking and a token-bucket rate control scheme to support application-aware transport requirements using overlay networks. Although simpler schemes can be used to meet the bandwidth requirement, the results show that application aware schemes at intermediate nodes can result in better quality of the end result. Effectiveness of the proposed approach is demonstrated by evaluating its performance in a network emulation

Table 6.4 Impact of packet marking based compensation scheme on the quality of the time-series data for multiple end users. Standard deviation in moment parameters is used for data quality estimation [Le06]

Gate Number	*141*	*142*	*143*	*144*	*145*	*146*	*147*	*148*	*149*	*150*
TR=3, MR=2 With No Compensation (dBz)	1.21	1.27	1.23	1.28	1.25	1.28	1.21	1.23	1.29	1.53
TR=3, MR=2 With Compensation (dBz)	1.10	1.14	1.04	1.04	1.10	1.11	1.07	1.14	1.28	1.44
TR=6, MR=3 With No Compensation	1.00	1.03	0.99	0.95	1.01	1.01	1.00	1.00	1.18	1.38
TR=6, MR=3 With Compensation (dBz)	0.98	0.98	0.87	0.87	0.96	0.93	0.93	0.98	1.14	1.33

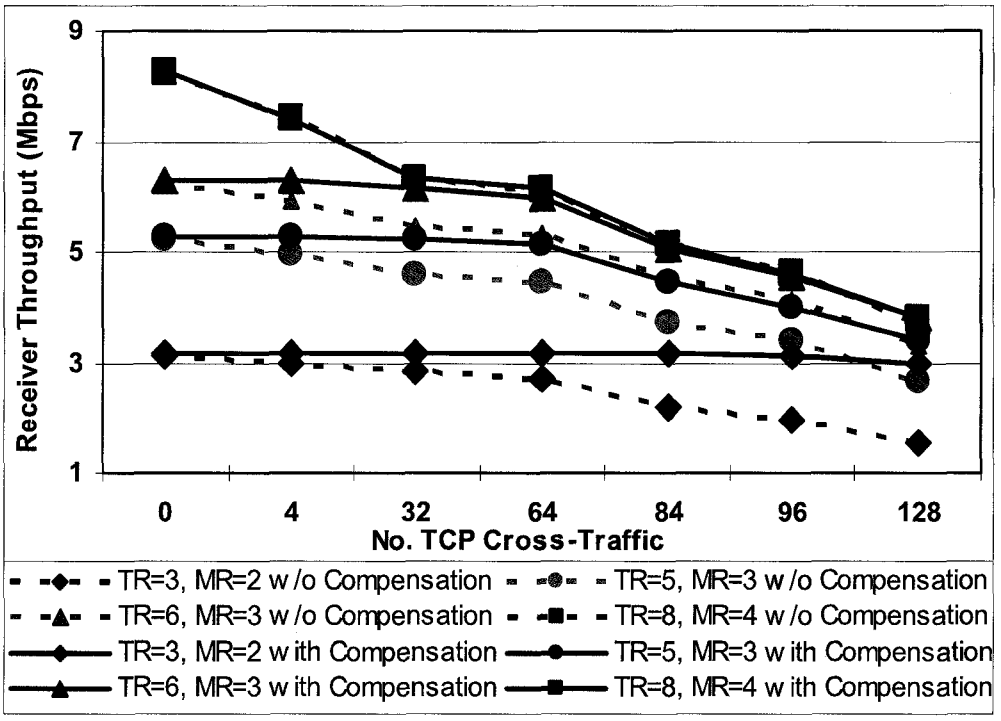


Figure 6.10 Effect of packet compensation on the receiver throughput at the end users [Le06]

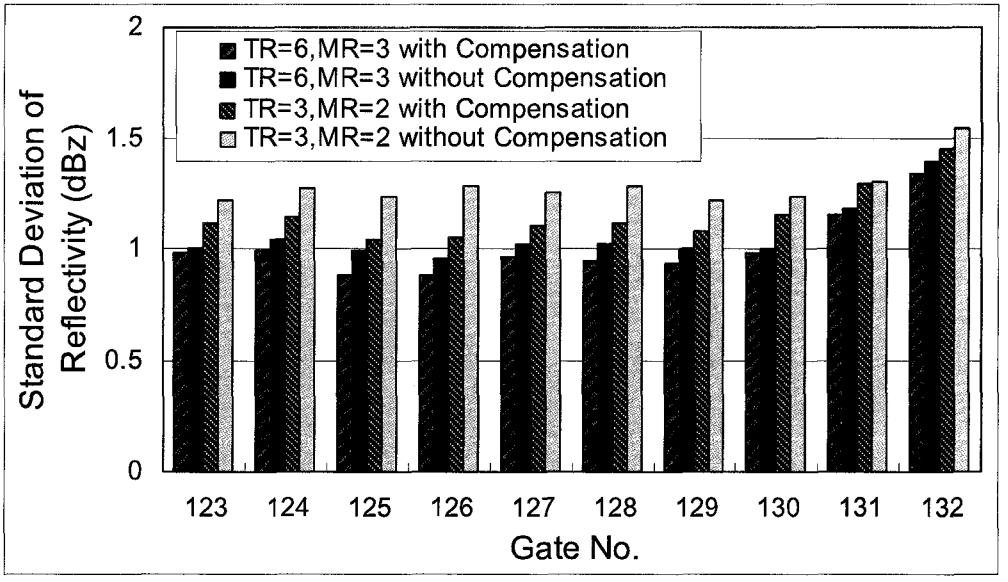


Figure 6.11 Effect of packet compensation on the data quality at the end users [Le06]

environment. During network congestion, packet marking is very effective in delivering high quality data to the end user. Moreover, when packet compensation technique is used it further improves the received bandwidth and data quality of the end users.

Chapter 7

AWON ARCHITECTURE FOR DEPLOYMENT OF APPLICATION-AWARE SERVICES USING OVERLAY NETWORKS

Overlay networks have been proposed to provide a range of useful services for enhancing QoS for Internet applications including bandwidth guarantees [Am06, An01, Ko03, Su03, Zh04a]. With overlay networking, application-aware processing can be implemented at intermediate nodes, thus significantly enhancing the ability of the application to adapt to network conditions and improve the QoS provided to the end users. Examples of these functionalities include application-aware data forwarding and data drops, as well as application-aware rate control during network congestion at intermediate nodes [Ba06, Ba07a]. It is often desirable to use the same overlay infrastructure for multiple simultaneous applications such as weather radar data streaming, and video streaming to multiple end users. A general-purpose overlay architecture that supports deployment of application-aware services on the overlay nodes in the network, and a programming interface required for such services that can leverage such an overlay network infrastructure to support application-specific QoS requirements will significantly enhance the overlay-based application deployment. This chapter proposes the AWON (**A**pplication a**W**are **O**verlay **N**etworks) architecture for application-aware overlay networking, and presents a motivation for a general purpose programming interface. The AWON architecture is presented in

this chapter that enables of development of application-aware services using overlay networks. It allows the applications to regulate the flow of data through overlay nodes in an application-aware manner, selecting data to be forwarded, and extracting/repackaging data, taking application-specific constraints into account.

A significant amount of research has been done on the design and development of overlay routing protocols to improve an underlay network's resilience and performance [An01, Li04, Sa99]. Our work complements and takes advantage of such ongoing research effort of performing QoS-aware routing in overlay networks such as RON [An01]. OverQoS [Su04], an overlay-based architecture can provide a variety of QoS-enhancing in-network services in the intermediate nodes of overlay networks, such as eliminating the loss bursts, prioritizing packets within a flow, and statistical bandwidth and loss guarantees. Our work is motivated by the same vision of enhancing QoS support within the network without the support from IP routers. An important difference between the AWON and the OverQoS architectures is that in the AWON-based approach, quality of service provided to an application is enhanced by performing application-aware processing within the network. Moreover, the AWON architecture is highly flexible and can accommodate QoS requirements of large class of applications. OCALA [Jo06] and Oasis [Ma06] enable the users of legacy applications to leverage overlay functionality without any modifications to their applications and operating systems. Opus [Br02b], which is motivated by active networking, provides a large-scale common overlay platform and the necessary abstractions to service multiple distributed applications. In contrast to our work, Opus focuses on the wide-area issues associated with simultaneously deploying and allocating resources for competing applications in a large-scale overlay networks. XPORT [Pa06] is a tree-based overlay networks, which can create dissemination trees based on diverse performance requirements of the applications.

Section 7.1 provides motivation for AWON and the programming interface for overlay networks. Section 7.2 explains the AWON architecture for deploying application-aware services

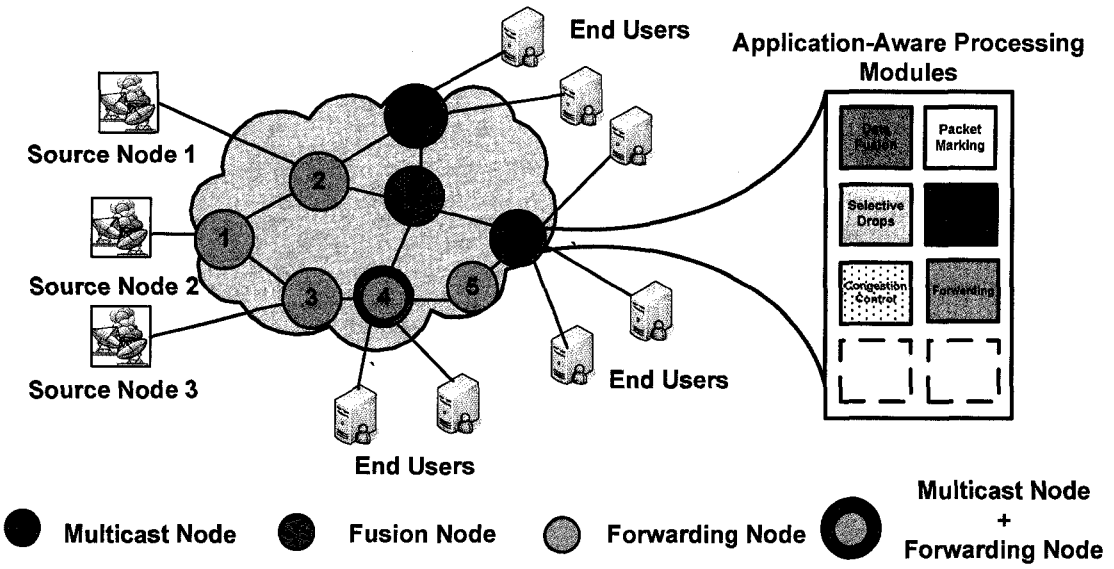


Figure 7.1 Overlay network for application-aware data dissemination

in overlay networks. Section 7.3 briefly describes the API. An example implementation is illustrated in Section 7.4. Section 7.5 shows Planetlab-based experimental results that demonstrate the effectiveness of the AWON and the corresponding API for weather radar data streaming. Concluding remarks are presented in Section 7.6.

7.1 Motivation

Applications relying on overlay-based implementations to achieve performance, reliability and other application specific requirements must be able to configure overlay nodes to perform in-network application-aware processing. A flexible, efficient approach for the deployment of QoS-sensitive applications using overlay networks should facilitate the monitoring of the QoS received by an application in the overlay network, and allow easy deployment of application-aware processing at intermediate overlay nodes. A framework is thus required for realizing such

application-aware overlay networks. A programming interface is needed to facilitate development and deployment of applications within this application-aware framework.

The API provides a layer of abstraction between an application and the underlying dynamics of the network infrastructure. It is desirable for the API to support application-aware adaptation in the overlay network, with each participating node possibly performing different application-aware operations to meet the overall goals of the application(s). The API must support node configuration in an application-aware manner, with each node being configurable to support multiple applications concurrently. There is also a need for communication between the application and the underlying overlay layers for supporting application-specific QoS requirements [An01, Ba03, Su04, Zh04a]. For this to be realized, the API must allow an application to specify its QoS requirements to the system. When the underlying system is able to accept the application with its QoS requirements, the API should be able to communicate this

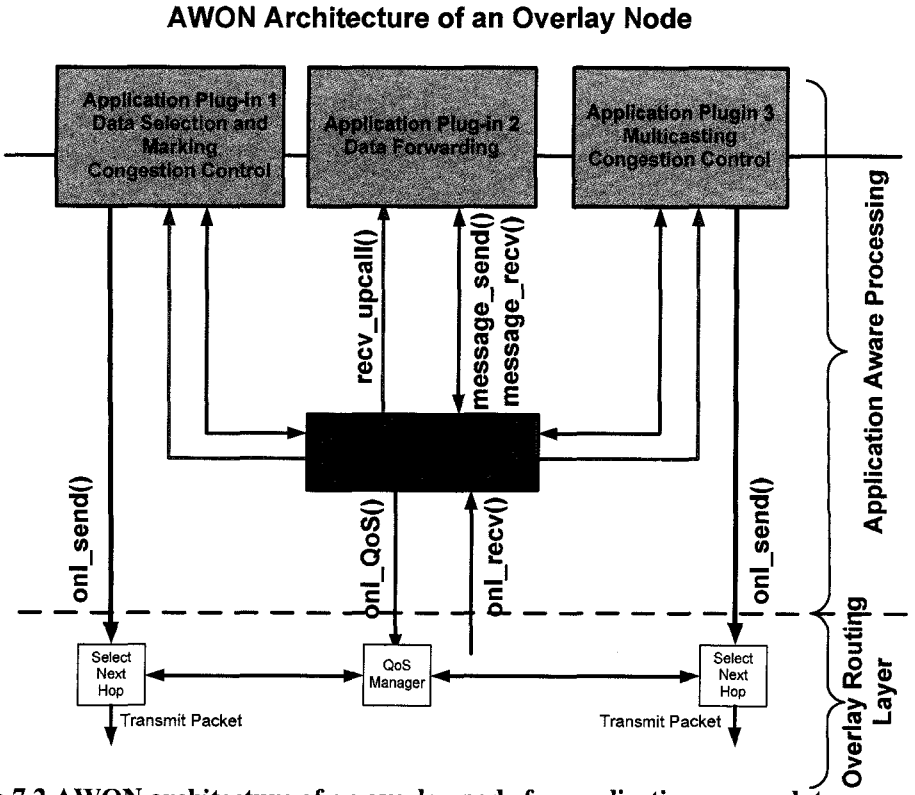


Figure 7.2 AWON architecture of an overlay node for application-aware data dissemination using overlay networks – An example node with multiple plug-ins [Ba07b]

acceptance to the application. CASA application is considered to illustrate the need for an application-aware architecture and a programming interface for such overlay networks.

7.2 Application aWare Overlay Network (AWON) Architecture

Fig. 7.1 shows an application-aware overlay network for distributing data to multiple sink odes with different end user requirements such as data quality and bandwidth requirements. Let us now illustrate the myriad roles overlay nodes may play in meeting application requirements.

In Fig. 7.1 source nodes 1-3 may perform application-level packet-marking to indicate the usefulness of the data to a particular application; nodes colored blue (nodes 1-5) may perform packet forwarding/drop based on the marking done by the source node; nodes colored green (multicast nodes 4, 7, and 8) may distribute data to multiple end users and perform independent congestion control for each end user in an application-aware manner. The multicast nodes combine the requests from the end users and send an aggregate upstream request to the specific source node.

If the network experiences congestion, congestion-based packet (information) discard can be performed at the source or at intermediate nodes, according to the available bandwidth. A source node can thus mark packets based on the relative importance of the information sent to the multicast nodes 4, 7, and 8. This facilitates application-aware selective drops (rather than random drops) within the network. Intermediate forwarding nodes 1-5 may use this marking information at the time of forwarding during network congestion. Similarly node 6, a fusion node, may combine data from multiple sources to reduce the downstream data bandwidth requirements.

In addition to the packet handling functions discussed above, there are two other classes of functions a node may implement. First, there is a need to support multiple applications

simultaneously on the same overlay network. Also, it may be necessary for an application to track performance of the underlying networking infrastructure in meeting the application requirements.

Fig. 7.2 shows the AWON architecture of an overlay node to support application-aware data-dissemination services. There are two key components of the AWON: *(i) Application Manager, (ii) Application Plug-ins.* Each of these components focuses on two different areas of functions with a common goal of providing best effort QoS services to the applications and providing a layer of abstraction to the application developers. Application developers are not required to be aware of other applications deployed on the same node.

Moreover, they need not be aware of the implementation of the underlying overlay routing infrastructure.

(i) Application Manager: The key responsibilities of the application manager are:

1. De-multiplexing packets received for different applications at the same node
2. Logging QoS status information for each application and informing (when appropriate) the underlying overlay routing layer about the QoS status/requirements of the applications
3. Authorization of a new user/application in the system based on a local policy

(ii) Application Plug-ins: In the application-aware paradigm, each application is required to configure its functionality in the participating overlay nodes. The AWON architecture supports application-specific plug-ins that implement the functions performed by the participating overlay nodes in the data dissemination. For a particular application, multiple nodes can play different roles, motivating the need to deploy relevant plug-ins on those nodes that implement particular functions. For an example, with a collaborative radar application [Mc05], the source node in Fig. 7.1 has application plug-in 1 shown in Fig. 7.2 for supporting data selection and marking. Similarly nodes 1-5 in Fig. 7.1 may have application plug-in 2 to support application-aware forwarding based on the source's marking. Nodes 4, 7, and 8 may have application plug-in 3 to support application-aware multicasting and congestion control. Note that the same node may have multiple plug-ins to support multiple functions performed by a node for a given application or

different applications. For an example, in Fig. 7.1, node 4 acts as a forwarding node and a multicasting node for the same application.

As seen in Fig. 7.2, the AWON architecture requires communication between application-manager and plug-ins, application manager and routing layer, and between plug-ins and routing layer. Next section briefly describes the programming interface to support deployment of application-aware services using AWON.

7.3 Application Programming Interface

Following are the key goals of the application programming interface:

(i) Enable deployment of application-aware services on the overlay network infrastructure.

(ii) Provide real-time monitoring of the QoS status of the application.

(iii) Facilitate communication between application-manager and plug-ins, application manager and routing layer, and between plug-ins and routing layer.

There are three broad categories of the API calls to deploy applications within the AWON framework:

(i) API calls for node configuration

(ii) API calls for communication between application plug-ins and application manager

(iii) API calls for communication with overlay routing layer

More details on the API and its implementation can be found in [Ba07b, Lee].

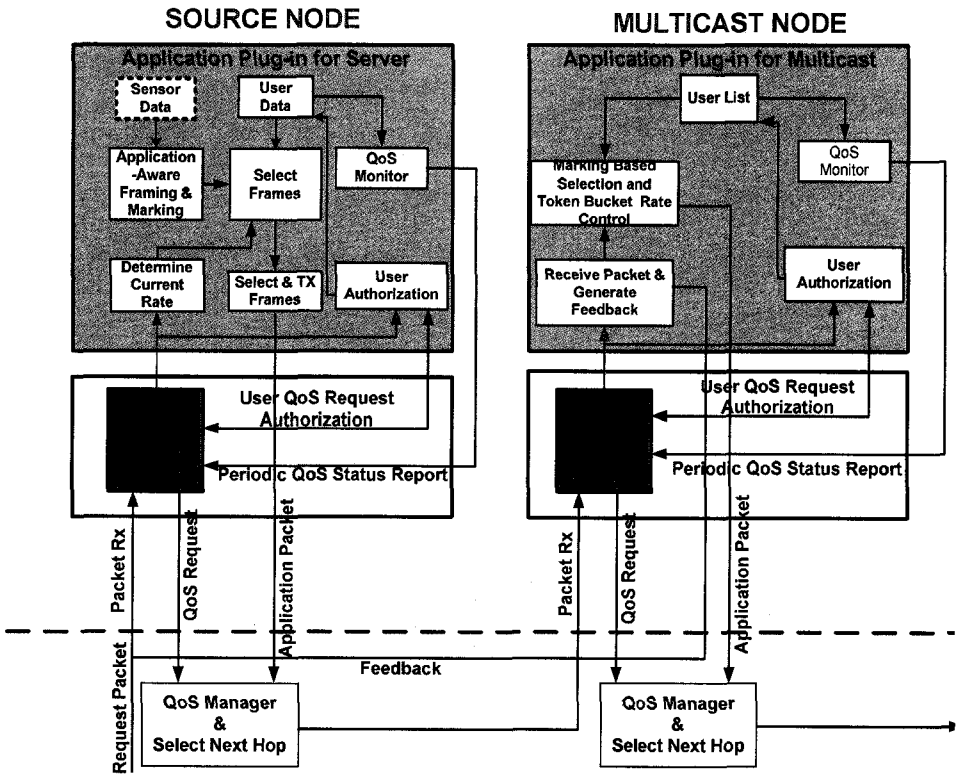


Figure 7.3 Implementation example based on AWON architecture [Ba07b, Lee]

7.4 AWON Implementation Example for the CASA Application

To demonstrate AWON capabilities, let us consider a CASA application as shown in Fig. 7.1, where data from a radar source node is distributed to multiple end users with distinct bandwidth and data quality requirements. In this application, an application-aware multicast node receives data from the source node for further distribution to multiple end users. AWON architecture is used to enable application-aware processing at source node and multicast node to best meet the QoS requirements of multiple end users.

Fig. 7.3 shows the implementation details of a source node and a multicast node based on the AWON architecture. Both nodes use application-specific plug-ins to implement application-

specific functionalities. The application manager implementation is same for all nodes in the overlay network. As shown in Fig. 7.3, the source node plug-in implements application-level packet marking and a rate-based congestion control algorithm. Packet marking determines the subset of the information that should be transmitted at a lower transmission rate for acceptable data quality at the receiver end. Fig. 7.4 explains the marking scheme used in the current implementation [Le06].

Consider an example as shown in Fig. 7.4, where a sensor node generates 8 application data units (ADU) within the bounded time at rate R1. The ADU is defined as a fundamental application data entity that can be used by an end user algorithm for processing. Each row in Fig. 7.4 shows the subset of ADUs that are selected for transmission at a lower transmission rate

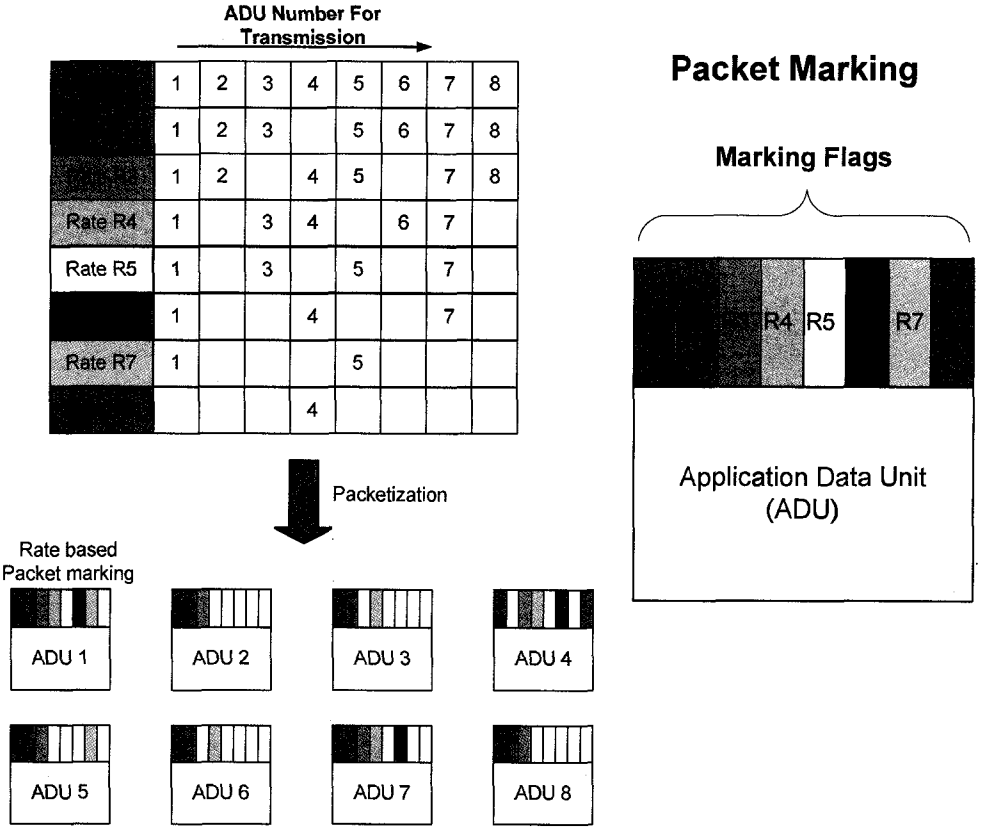


Figure 7.4 Application-aware framing and packet marking where each non-white color represent rate for which packet is marked, i.e., rate R1-R8 [Le06]

when a higher rate cannot be supported because of bandwidth constraints. The subset of data selected at lower rate depends on the end user data quality requirements. For example, certain end users need uniformly spaced ADUs when only a subset of the data can be selected for transmission. Alternatively, other end users prefer a contiguous group of ADUs when bandwidth is constrained. Consider the case when the source node transmits data at rate R1, and as seen in the figure the data transmitted at lower rates is a subset of the data transmitted at rate R1 and ADUs are selected uniformly at lower rates. The packet containing ADU 1 is marked with different color flags corresponding to different rates, i.e., rates R1-R7. Similarly packet containing ADU 3 is marked with different colors corresponding to different rates, i.e., R1, R2, R4, and R5. As shown in the Fig. 7.4, every packet contains a flag for each rate for which it is transmitted indicated by different colors. Note that multiple flags can be set to indicate suitability of the packet for multiple transmission rates. In the current implementation we consider a case when all end users have similar data quality requirement and can tolerate uniform drop of ADUs under bandwidth constrained conditions.

The QoS monitoring component of the plug-in monitors the quality of the service received by the application users at a source node. Currently, the component monitors whether end users' bandwidth requirements are met. The multicast application plug-in supports application-aware rate control using a token-bucket scheme and on-the-fly forwarding of data based on the packet marking. More information on the packet-marking and token bucket scheme used for the implementation can be found in Chapter 6 and [Le06]. This application-specific plug-in selects data for forwarding based on the available network bandwidth and the packet marking for multiple end users. Note that the packet marking performed at the sender node determines the priority of the packet to be forwarded at the multicast node. In such systems, each end user may need a different subset of the data from the radar source based on the intended use of the data [Ba05b, Ba06]. During network congestion, overlay nodes can perform a better job by selectively

dropping [An00, Ba06, Gu05, Zh99] packets (information) instead of dropping randomly within the network, taking into account end-user requirements for different subsets of the data.

7.5 Performance Evaluation

In this section, we demonstrate the effectiveness of the real-time application-aware processing, which is implemented using AWON architecture and the API over overlay networks such as planetlab [Pe02, Pla].

Application: We consider a mission-critical CASA [Mc05] application for the performance evaluation. One of the requirements of CASA application is to distribute high bandwidth real-time weather radar data to multiple end users [Ba06] with distinct critical bandwidth and data quality needs. For such applications, it is not only important to meet the bandwidth and latency requirement, but also to meet the minimum content-quality requirement for the proper operation of the system. For example, each CASA end user may specify its critical minimum rate (MR) requirement that should be met for the proper operation of the system. Moreover, each end user may also dictate a target rate (TR), i.e., the maximum rate at which data can be received by the end user. A source node periodically generates a block of digitized radar data, referred to as a DRS block [Bg03b, Ba05b]. Each end user specifies its content-quality requirement in terms of tolerance towards bursty losses or uniform losses within the DRS block. In the current implementation, we consider a case in which all end users prefer uniform drops of information instead of bursty drops within a DRS block. In case of our CASA application, during network congestion, the desired rates are between MR and TR and the desired packets are those that contain subset of the DRS block of data with uniform drops. All these selected packets are marked for rate between MR and TR at the source node. We implement this application using the AWON architecture, as it enables application-aware processing within overlay nodes to enhance

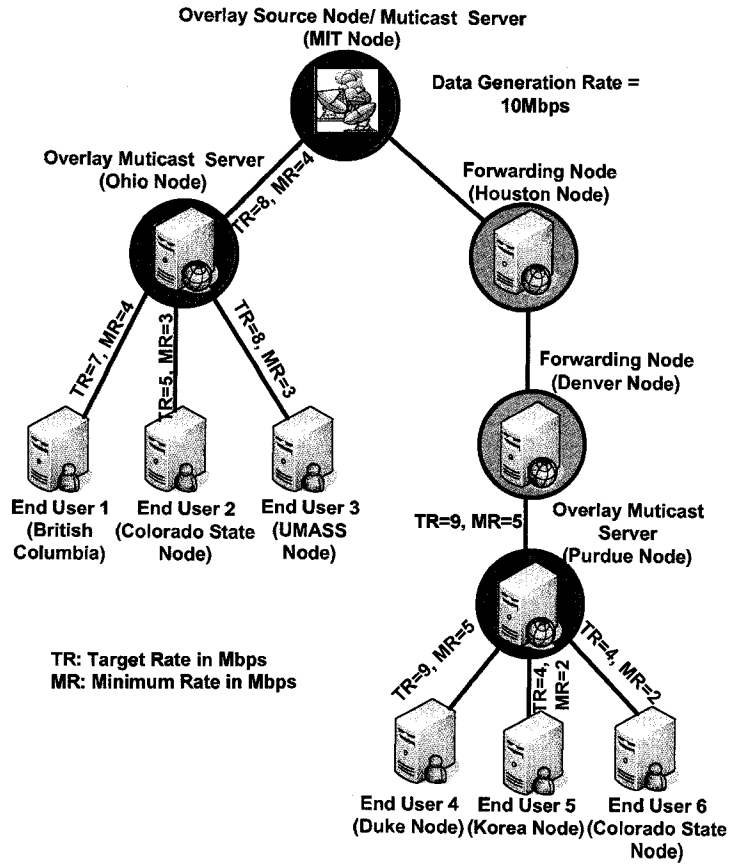


Figure 7.5 Planetlab test-bed for application-aware multicasting [Ba07b]

the QoS under dynamic resource-constrained conditions.

Overlay Network Topology: Fig. 7.5 shows the Planetlab- based overlay network topology used for application-aware data distribution and performance evaluation. It consists of 11 overlay nodes, each configured to perform application-specific tasks to meet the overall QoS requirements of the application. In Fig. 7.5, there are four different types of nodes that are present in the overlay network - a *source node*, a *multicast node*, a *forwarding node*, and an *end user node.* The *source node* performs selective data drop during network congestion as well as application-aware packet marking based on the end user's data quality requirement as explained in Chapter 4. The goal of the marking scheme is to deliver the most appropriate subset of data for the end user under congested network conditions. The *forwarding node* may decide to forward a

packet based on a packet's marking and the available downstream link bandwidth. The *multicast node* performs on-the-fly selection of the data for forwarding based on packet marking to the respective end users at the current transmission rate. The *multicast node* uses TRABOL (TCP-Friendly Rate Adaptation Based On Losses), a UDP-based rate-based congestion control algorithm [Bg03b, Ba05b], to independently determine the transmission rate for each end user. The *end-user node* performs content quality evaluation using application-specific performance metrics and provides periodic feedback to the *multicast node* about its current receive rate. In Fig. 7.5, six different *end-user nodes* 1-6 at geographically different locations receive weather radar data streams from the source node at *MIT*, Cambridge at their required TR and MR over the planetlab. The source node generates data at a constant rate of 10Mbps. *End user nodes* 1-3 make their data request with the desired TR and MR requirement to the multicast node at *Ohio*. Similarly *end-user nodes* 4-6 make data requests with their desired TR and MR to the multicast node at *Purdue*. After requests are received from the end users, both multicast nodes independently send aggregate bandwidth requests to the source node at *MIT*. A single stream of radar data is delivered from *MIT* to the *Ohio* node for further distribution to *end user nodes* 1-3. Similarly, a single stream from the *MIT* source node is delivered to the multicast node at *Purdue* for further distribution to *end user nodes* 4-6.

Performance Metrics: The effectiveness of the application-aware processing using AWON architecture and the programming interface can be evaluated by measuring the quality of the content delivered to the end users under different network congestion conditions. For most real-time applications, application-specific metrics are used to measure quality of the content; for multimedia applications, these metrics include PESQ [Am06, Su04] for voice quality and PSNR [Su04] for video streaming. For the CASA application we use the standard deviation of the estimated sensed values (specifically, reflectivity and wind velocity) to evaluate quality of the radar data [Ba05b, Ba06]. A lower standard deviation indicates better radar data quality. A minimum standard deviation, i.e., the highest content quality, is achieved when all the data from

the source node is delivered to the end users. Alternatively, we also evaluate the content quality by measuring the frequency of the desired packets at the receiver node based on their markings. For better quality of the data, it is necessary to receive more packets with the desired markings. For an application with TR and MR bandwidth requirements, the "most appropriate" packets are marked to result in data rates between MR and TR.

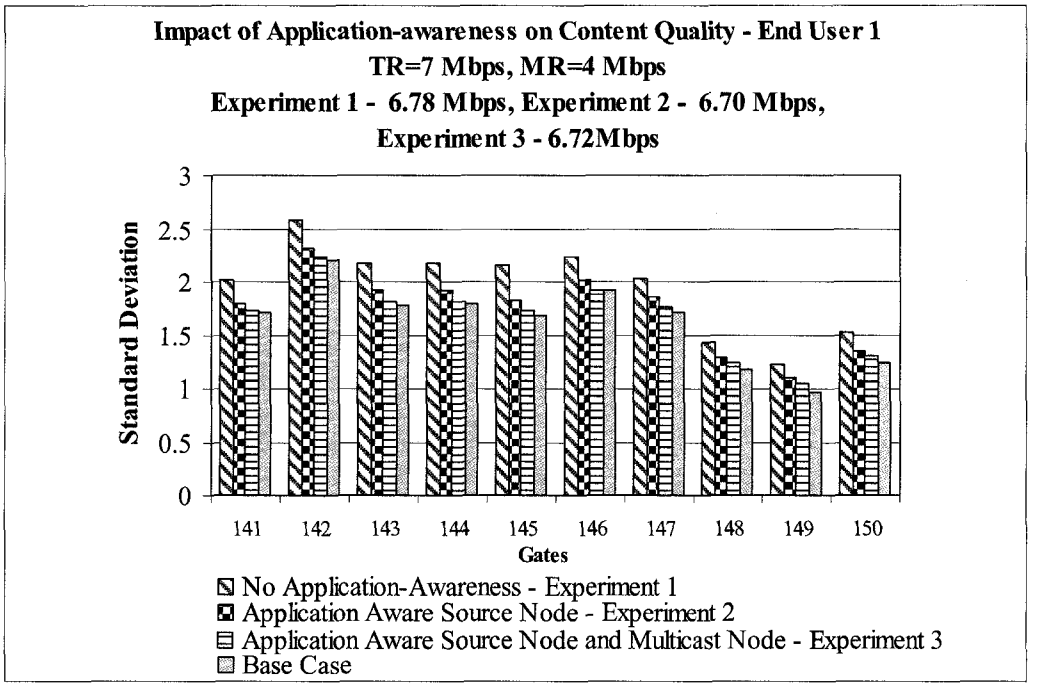
Methodology: We perform three sets of experiments to demonstrate the effectiveness of the application-aware processing within overlay networks implemented using AWON architecture and the API. In the first set of experiments, i.e., experiment 1, no application-aware processing is performed in the network, i.e., the source node randomly selects data from a DRS block of radar data for transmission, without considering end-user loss tolerance requirements. Packet marking is performed but packet marks are not used at the forwarding nodes or at the multicast nodes for on-the-fly selection of packets for transmission. In experiment 2, the source node performs application-aware selective drop during network congestion and marks packets at the time of transmission. However, packet marking is not used at forwarding nodes and multicast nodes for on-the-fly selection of data for transmission to the end users. Experiment 2 is equivalent to a network that supports limited application-aware processing at end hosts without the support of AWON architecture. Experiment 3 is an example of the AWON-based implementation that enables in-network processing by performing different application-specific tasks within the network. In Experiment 3, the source node at *MIT* performs application-aware selective drops and packet marking. The multicast nodes at *Ohio* and *Purdue* use token-bucket based rate control scheme along with packet marking to select appropriate packets on-the-fly for transmission to individual end users at their respective transmission rate. At present, in experiment 3, nodes at *Houston* and *Denver* act as simple forwarding nodes and do not make use of packet marking when forwarding packets. Fig. 7.6 and Fig. 7.7 show the result of experiments 1-3. Table 7.1 and Table 7.2 shows data corresponding to results shown in Fig. 7.6. Table 7.3 and Table 7.4 show

Table 7.1 Impact of AWON based implementation on the data quality of time-series data for End User 1 under varying degree of application-aware processing

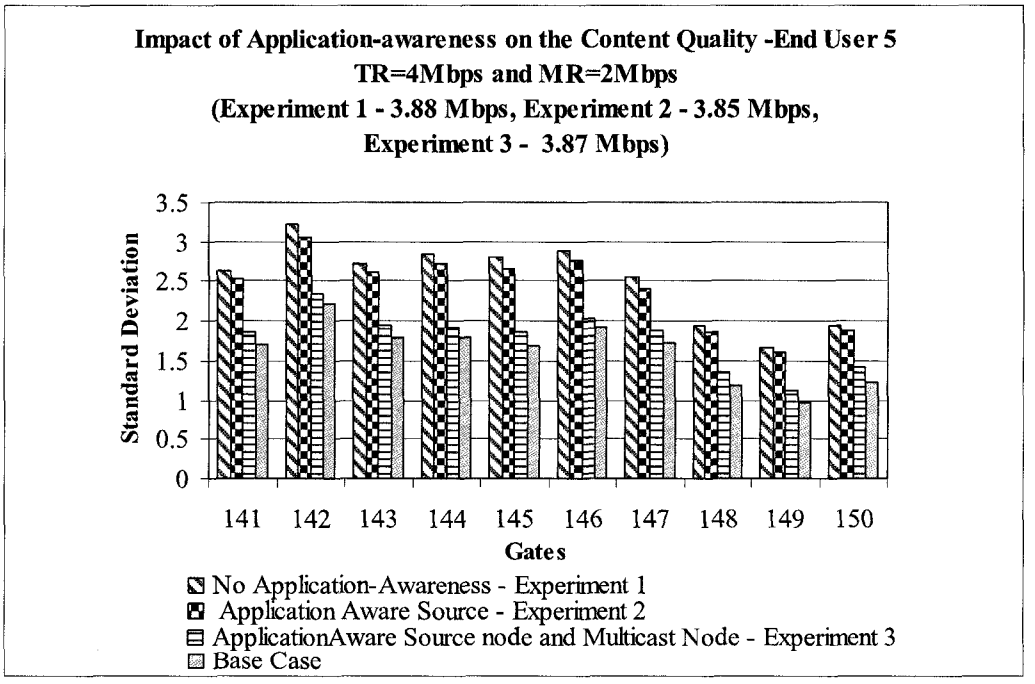
Gate Number	*RD, RF Standard Deviation (dBz)*	*SD, RF Standard Deviation (dBz)*	*SD, SF Standard Deviation (dBz)*	*No Loss Standard Deviation (dBz)*
141	2.01	1.79	1.73	1.70
142	2.59	2.31	2.23	2.20
143	2.17	1.92	1.8	1.78
144	2.17	1.92	1.81	1.79
145	2.15	1.82	1.72	1.68
146	2.23	2.02	1.92	1.92
147	2.02	1.84	1.76	1.72
148	1.42	1.29	1.24	1.18
149	1.21	1.09	1.0	0.97
150	1.52	1.34	1.30	1.23

Table 7.2 Impact of AWON based implementation on the data quality of time-series data for End User 5 under varying degree of application-aware processing

Gate Number	*RD, RF Standard Deviation (dBz)*	*SD, RF Standard Deviation (dBz)*	*SD, SF Standard Deviation (dBz)*	*No Loss Standard Deviation (dBz)*
141	2.63	2.53	1.84	1.70
142	3.21	3.06	2.32	2.20
143	2.72	2.61	1.93	1.78
144	2.84	2.72	1.92	1.79
145	2.80	2.66	1.85	1.68
146	2.89	2.76	2.02	1.92
147	2.55	2.41	1.86	1.72
148	1.93	1.85	1.33	1.18
149	1.67	1.60	1.12	0.97
150	1.94	1.87	1.41	1.23



(a)



(b)

Figure 7.6 Impact of application-aware architecture on the content quality delivered to the end users (a) Standard deviation of data for end user 5 with low bandwidth requirement TR=4, MR=2, (b) Standard deviation of data for end user 1 with high bandwidth requirement TR=7, MR=4 [Ba07b]

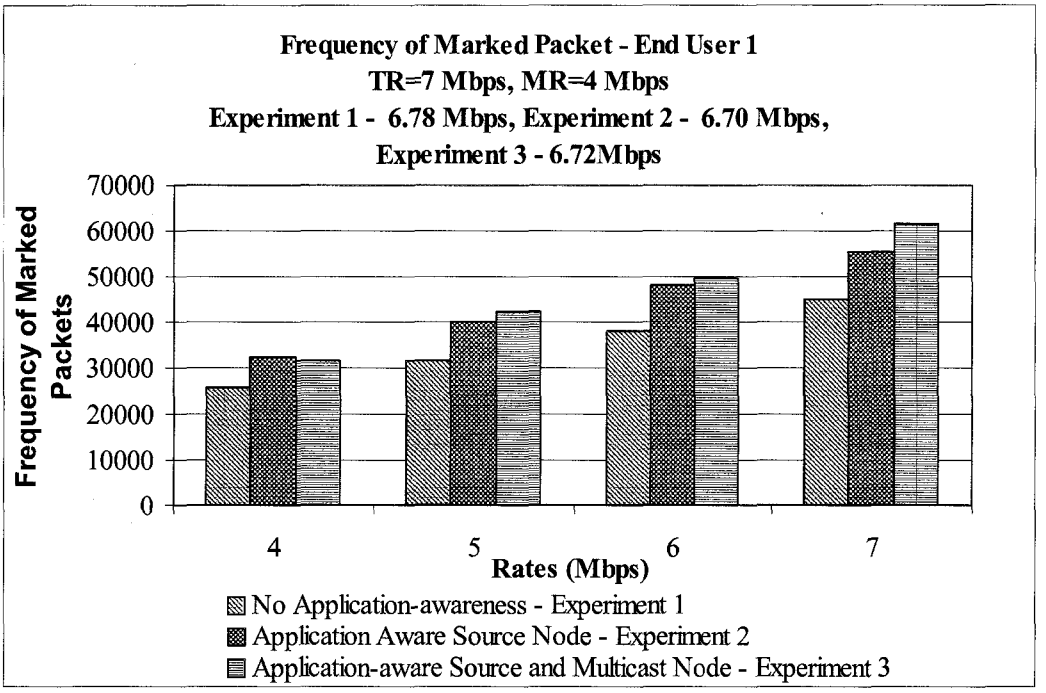
data for results shown in Fig. 7.7. Performance is compared by measuring the quality of the content delivered to the end users for different experiment scenarios under different network congestion conditions. Two metrics are used to evaluate the content quality delivered to the end users. In Fig. 7.6 application-specific metric standard deviation is used to measure the quality of the content. In Fig. 7.7 frequency of the packets is used to determine the impact of application-ware processing on the delivery of application-relevant packets to the end users under network congestion condition. Results are shown for End user 1 and End user 5 in Fig. 7.6 and Fig. 7.7 but similar trends are observed for all other users. As mentioned earlier, data is generated at 10Mbps at the source node but end user 1 requests for TR=7Mbps and MR=4Mbps. End user 5 has relatively lower bandwidth requirement with TR=4Mbps and MR=2Mbps. Both end users can tolerate uniform drop of data within the DRS block. Both end users compute reflectivity [Ba05b]

Table 7.3 Impact of AWON based implementation on frequency of packet with desired marking for end user 1

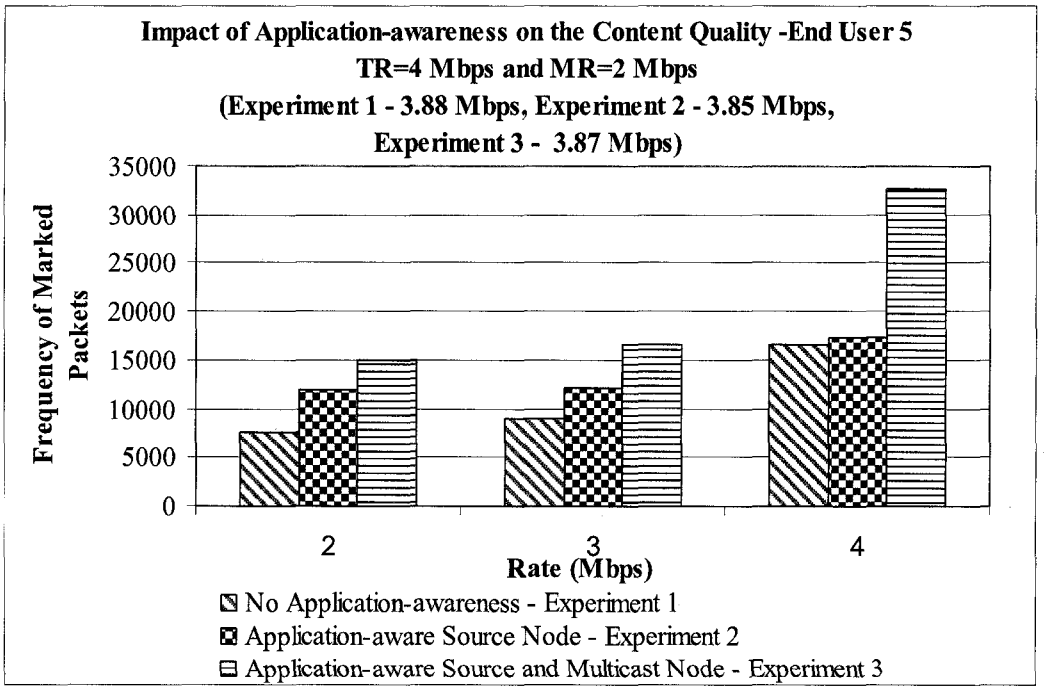
Transmission Rate	*RD, RF Marked Packets Frequency*	*SD, RF Marked Packets Frequency*	*SD, SF Marked Packets Frequency*
4 Mbps	25851	32138	31503
5 Mbps	31725	39995	42256
6 Mbps	38100	48160	49572
7 Mbps	44824	55447	61728

Table 7.4 Impact of AWON based implementation on frequency of packet with desired marking for end user 5

Transmission Rate	*RD, RF Marked Packets Frequency*	*SD, RF Marked Packets Frequency*	*SD, SF Marked Packets Frequency*
2Mbps	7493	9048	16531
3 Mbps	11965	12126	17354
4 Mbps	14965	16644	32711



(a)



(b)

Figure 7.7 Impact of application-aware processing on the delivery of application-specific relevant packets (a) Marked packet frequency for end user 5, (b) Marked packet frequency for end user 1

using raw data received from the radar source node. Fig. 7.6(a) and 7.6(b) show the standard deviation of reflectivity for all three experiments. In CASA application each end user computes reflectivity for multiple gates [Ba05b] as explained in Chapter 4. (In radar terminology, a gate refers to a volume in the atmosphere at a particular distance from the radar source node for which data is collected by a radar.) Fig. 7.6 thus shows content quality, i.e., standard deviation for subset of gates. As seen in Fig. 7.6(a) and 7.6(b), experiment 1, with no application-aware processing support within the network, has highest standard deviation and hence has the worst data quality among three cases. In experiment 2, when limited application-aware drops are performed at the source node, the quality of the data improves in comparison to experiment 1, as indicated by decrease in standard deviation. Experiment 3, which has support for application-aware drop at the source node and marking-based selective drop at the multicast nodes, delivers data with the highest quality, i.e., with the smallest standard deviation. As shown in Fig. 7.6(b) under high loss conditions, the AWON based implementation of application-aware one-to-many protocol is very effective in improving the quality of the data delivered to end users. Indeed, the standard deviation of the AWON based implementation approaches that of the *base case* standard deviation, which corresponds to a scenario when *all* data from the source node generated at 10Mbps is delivered to the end users. Note that in experiments 1-3, end users receive data at approximately the same rate, but the content quality is different. For an example, in Fig. 7.6(b), end user 1 receives data at 3.88Mbps, 3.85Mbps, and 3.87Mbps for experiment 1, 2 and 3 respectively. However, the application-level quality of data delivered to the end users is significantly different for all gates. The gain in performance in terms of content quality is achieved because AWON modules deliver the most appropriate application-specific content to the end user within the available bandwidth resources. This is made possible by performing application-aware processing of the data as it traverses the network.

Fig. 7.7(a) and 7.7(b) show the impact of the three experiment scenarios on the delivery of most appropriate information to the end user at a given rate. Packets are marked for different rates

for which it is most suitable for transmission as explained in Chapter 6. When an end user receives more packets with markings corresponding to the desired rate, this is an indication of a higher quality of received data. As mentioned before, for CASA end users, the desired rates are between MR and TR and the desired packets are those that are marked for rates between TR and MR. In Fig. 7.7(a) and 7.7(b), we show the number of packets delivered with the marking corresponding to rates between TR and MR requirements of the end users. Fig. 7.6(a) and 7.7(a) both measure content quality using different metrics and correspond to the same end user 1. Fig. 7.6(b) and 7.7(b) illustrate the content quality for end user 5. As seen in the Fig. 7.7(a), and 7.7(b), experiment 1 with no application-awareness, delivers fewer packets with the desired marking. Alternatively, the frequency of the packets with desired marking increases with experiment 2 resulting in a higher content quality. In the case of experiment 3, the frequency of desired marked packets is the maximum over all three cases. As seen in Fig. 7.7(b), during high network congestion, AWON based architecture is able to deliver 50% more desired packets than the case when no application-aware processing is done in the network. These results corroborate the results shown for data quality in Fig. 7.6, which uses standard deviation quality metric for end user 1 and end user 5 respectively.

The above experiments demonstrate that the AWON architecture enables the deployment of application-aware services in the overlay networks and that such overlay services can be very effective in improving the performance of an application in resource-constrained conditions.

7.6 Remarks

This chapter presented the AWON architecture for the application-aware data dissemination using overlay networks. Planetlab experiments demonstrate the suitability of the AWON for the deployment of application-aware services in overlay networks. Experiment results show that

during network congestion, an AWON-based application-aware transport services delivers better quality data to the end users than a non-application-aware implementation while using a similar amount of bandwidth. The AWON architecture and programming interfaces are generic and are not limited to a particular application and can thus be used to deploy applications that need application-specific processing within the network to meet its QoS requirements.

Chapter 8

TARDINESS MEASURE FOR CHARACTERIZATION OF SENSOR NETWORK PERFORMANCE

As mentioned in Chapter 1-3, there is a need to evaluate QoS delivered to the end users in terms of freshness of the data in order to provide effective application-aware transport services in mission-critical sensor networks. In mission-critical sensor network applications, the sensor network collects information about a physical phenomenon, processes the data and then takes appropriate actions based on the processed data [Ak02, Ak04, Es99]. In such applications, action should be taken in bounded time for the proper operation of the system. Input data may be useless for such applications if it arrives and is processed after a critical deadline. Therefore, it is important to be aware of the age of the data that is used for processing and computing results. Data freshness has been studied in the context of information system such as data integration system (DIS), and Data Warehouse [Bo04]. In that context, data freshness is considered as a critical component that determines the success of many information systems. A significant amount of research has been done for studying efficient refresh policies for web crawlers to keep the local copies of the remote source data fresh [Cho03]. The key factor that impact the age of the data in such traditional information systems is the rate of change of data at the remote source node. However, in sensor networks besides the rate of change of data some of the other factors that may impact the age of the data are the high network delays, random loss of packets, and

packet re-ordering. It is significantly more challenging to quantify and understand the impact of each of these parameters on the age of the data in sensor networks. Our focus on understanding impact of network dynamics on the age of the data makes it different from the existing work that is focused on investigating freshness of data in the context of information systems.

Sensor networks are typically resource constrained in terms of computation capability and available energy. Moreover, most sensor networks use error prone wireless links for communication. Different energy conserving transport, routing, and MAC protocols are used for transferring data from source nodes to the sink nodes [He03, Hu04, In00, Li99, Wa02, Wa03, Ye04]. Alternatively, there is an emerging class of real-time sensor networks that uses Internet to distribute sensor data to different computing nodes and end users. Depending on the applications, sensor nodes may be configured to monitor the environment for rare and ephemeral events [Du05]. Whenever such events occur, they are detected by one or more sensing nodes, and the event information is transmitted to a data fusion node or a remote sink node for further processing. Alternatively, in many sensor networks, sensor nodes continually sample environmental processes such as temperature or vibrations as shown in Fig. 8.1. Sink nodes such as fusion nodes may access receive buffer for the data after receiving an interrupt due to arrival of

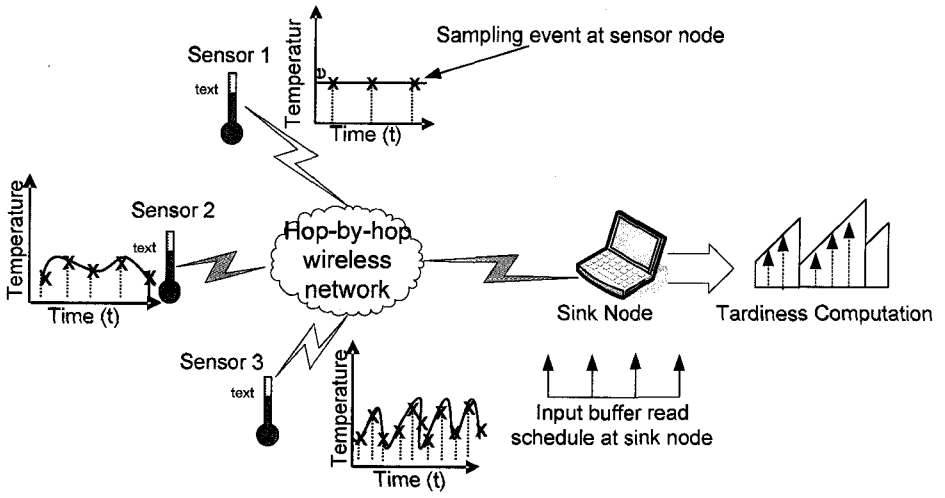


Figure 8.1 Process Monitoring Sensor Network

a new packet. Alternatively, sink nodes may poll their receive buffers recurrently for the arrival of the data. In both scenarios, data available in the buffer at the time of 'read' is used for the computation. However, it is possible that data read from the input buffer at the sink node may be different from what is available at a particular instant of time at the sensor node. This scenario is possible because either the data gets late within the sensor network due to network delays [Chi04, Zh04b] or gets dropped because of reasons such as network congestion or wireless link errors in the sensor network [Hu04, Wa03]. When data is dropped in the network, the prior sample of data available is used until new data arrives. In many closed-loop applications, old copy of the data that does not reflect the current state of the environment may be used to generate actuating signal to control the remote environment. This has the potential to compromise the integrity of such systems. It is therefore desired to get a quantitative estimate of age of the data used in the computation at every 'read' event at the sink node.

This chapter proposes a framework to capture network introduced tardiness of data used for end computations in sensor networks. A tardiness measure is defined to capture the age of the data used for computation under different network conditions. An analytical model is proposed that relates network delays, wireless loss rate, degree of packet re- ordering, and sampling rate with the observed tardiness of the data. Moreover, we study the tradeoffs between energy consumption and tardiness of the data delivered to the end user. Such an analysis will abstract the impact of the sensor network characteristics, such as losses and delay due to routing scheme employed, in terms of statistics of tardiness. The statistical characteristics of tardiness may then be used to evaluate the accuracy and reliability of the application, without delving into detailed network characteristics such as the routing protocol. We envision wider applications of the tardiness measure. Tardiness may be used to evaluate and compare the performance of routing protocols in terms of age of the data delivered to the sink node, adaptive sampling techniques [Ja04, Ma03], and effect of network topologies in meeting real-time requirements of the applications. This work may also help in configuration of sampling rates, transmission energy,

sleep/active schedules at MAC layer, and input buffer read frequency at sink nodes for minimizing the error in the end results due to tardiness of the data. Section 8.1 describes the tardiness measure and analytical model is derived in Section 8.2. Section 8.3 presents results for the model verification. Section 8.4 discusses tradeoffs between tardiness and energy consumption in sensor networks. Concluding remarks are presented in Section 8.5.

8.1 Tardiness Measure

The difference between an ideal monitoring system, in which the processing/decision node has instantaneous access to the values/events/parameters being monitored, and a distributed sensor network lies in the age or tardiness of data available at the processing node. With latter, the available data from different sensor nodes have different ages, which depend on network characteristics and protocols.

For example, consider the scenario where the processing/decision nodes receives data from multiple sensors. Let at time t, $X_1(t)$, ...$X_N(t)$ be the data values available at the sink node from N different source nodes about the phenomena under observation as. Let F(t) be the processing/decision function that combines the most recent data from N source nodes at time t as shown in Eq. 8.1. For an ideal monitoring system, as described above,

$$F(t) = h(X_1(t), X_2(t) \ldots X_N(t)) \qquad (8.1)$$

i.e., instantaneous data from all sensors is available at the processing node. However, in distributed sensor network based system,

$$F(t) = h(X_1(t-\Delta_1), X_2(t-\Delta_2) \ldots . X_N(t-\Delta_N)) \qquad (8.2)$$

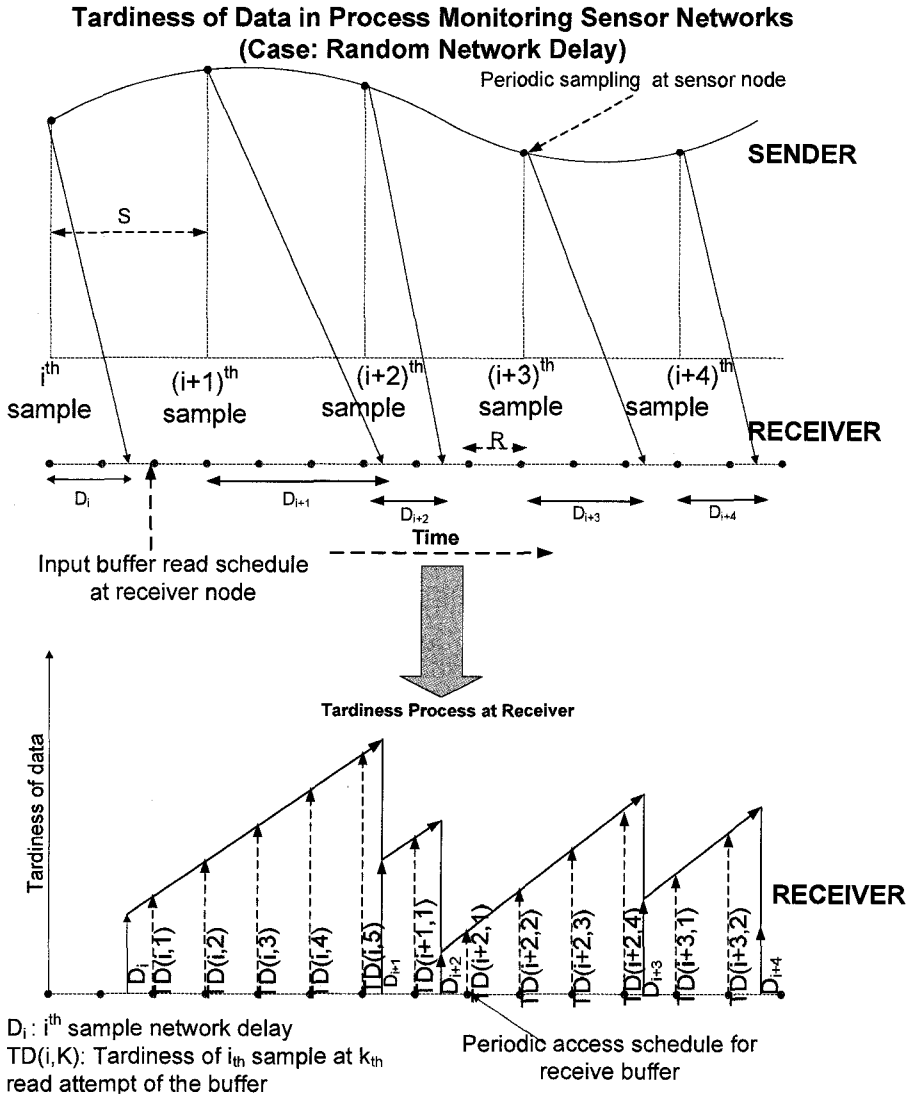


Figure 8.2 Tardiness measure due to random network delay in process monitoring sensor networks

Where Δ_i corresponds to the tardiness of data from source i to the processing/decision sink node. The Δ_i in Eq. 8.2 values form a random process that is affected by the network protocols, losses, sampling frequency, sleep schedule, etc. Characterizing this tardiness process allows the impact of network characteristics to be summarized in a way that its influence on different applications (based on different decision functions) can be evaluated more conveniently.

Tardiness measure captures the age of the data used for computation at the receiver node. Age of the data is defined as the time lag from the time data is generated at the sensor node to the time data is used at the sink node by the application.

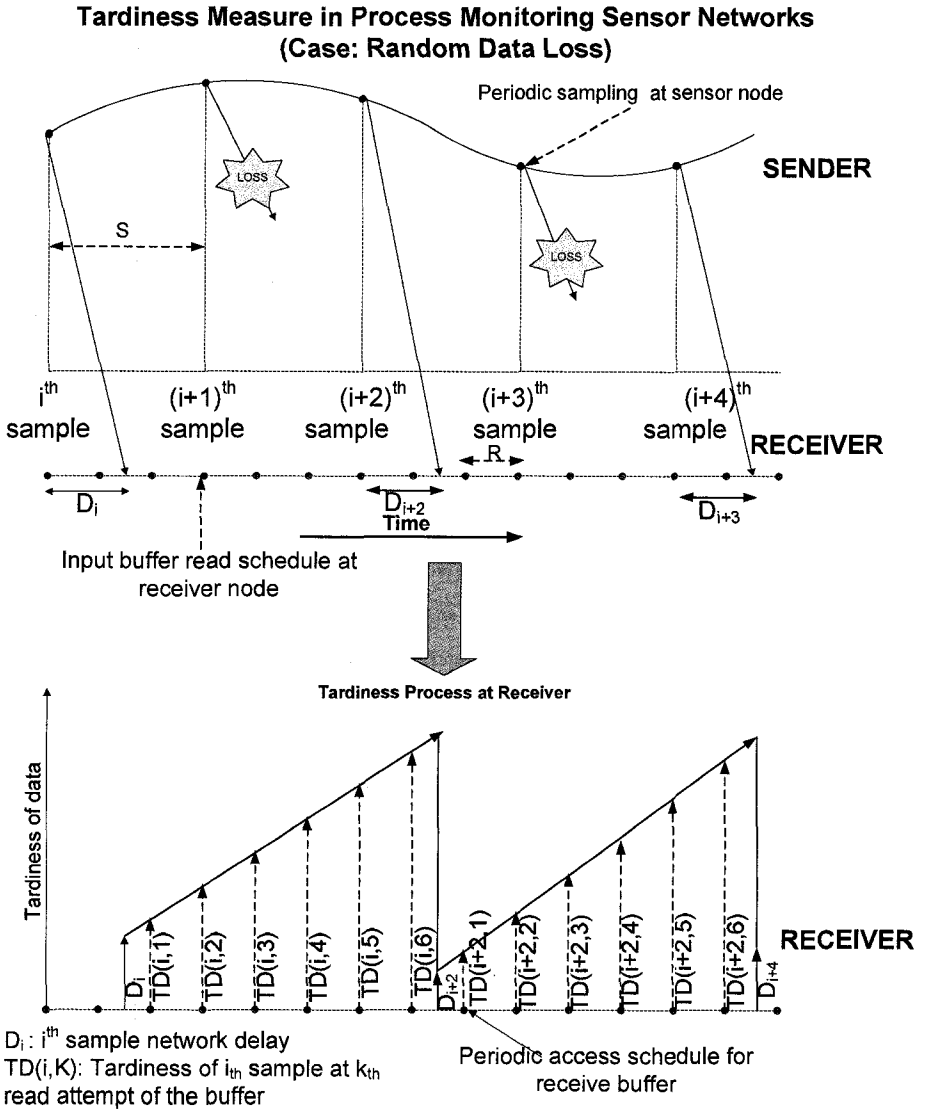


Figure 8.3 Tardiness measure under random data loss in process monitoring sensor networks

8.1.1 Tardiness under Dynamic Network Conditions

A process monitoring sensor network shown in Fig. 8.1 is considered. In this scenario, sensor node periodically samples the physical environment for the data every 'S' time units as shown by the vertical dotted lines in Fig. 8.2 and Fig. 8.3. At every sampling instant, generated sample is packetized, and is immediately transmitted over single or multi-hop wireless network towards the sink node. In this example, the sink node periodically accesses the input buffer for the received data every 'R' interval as shown by red color dots. For the purpose of tardiness evaluation it is assumed that each transmitted sample is time-stamped at the sender node, and all nodes in the sensor network are time synchronized [Si04b]. In Fig. 8.2 and Fig. 8.3, D_i is the network delay suffered by the sample *i*, and TD[i,k] is the tardiness of the data read at k^{th} successive read attempt when sample *i* is present in the buffer. Dotted vertical lines with arrows shows the age of the data read from the buffer at a particular input buffer read attempt. We now illustrate how random delay and random losses impact the age of the data at the sink node. Two cases are considered; in the first case packets only suffer random delay and no packet losses. In the second case, packet suffers both random delay and random packet losses.

8.1.1.1. Tardiness Measure under Random Delay and No Network Packet Loss

Fig. 8.2 shows the case when samples suffer random delay in the network and does not suffer any network packet loss. Network delay depends on factors such as active/sleep schedule in MAC layer, paths selected by the routing protocol, and node distribution in the sensor network. Under these conditions samples arrive at the sink node after suffering random delays. Depending on the arrival time and the periodic read interval time 'R', same sample can be read multiple times by the end application. As shown in the Fig. 8.2, the age of the data increases linearly with time until the next sample arrives.

Table 8.1 Parameters for tardiness analytical model

Parameter	*Description*
$\mathbf{D_i}$	Network delay of sample i
$\mathbf{A_i}$	Arrival time of sample i at sink node
$\mathbf{B_m}$	Time spent by a given sample in the receive buffer until m^{th} read attempt
$\mathbf{B_{i\_max}}$	Maximum time sample i spends in the receive buffer
$\mathbf{P_L}$	Total packet loss perceived by the application at a sink node
$\mathbf{P_N}$	Network packet loss probability
L	Random variable that models loss characteristics of the network
D	Random variable that model network delays
S	Sampling Interval used at the source node to periodically sample environment
$\mathbf{P_{I/A}}$	Total probability that packet is in-order on condition that it arrives at the sink node
$P_{I/A}^{jS}$	Probability that packet is in-order on the condition that it arrives at the sink node and it suffers network delay between $(j-1)S$ and jS
M(t)	Total packet arrivals in interval [0,t] at a sink node
$\mathbf{W_k}$	Weight of the data generated by sensor k in the network
$f_d(c)$	Delay distribution of the sensor network
$F_D(c)$	CDF of delay in sensor network
$P_{RR}[d]$	Packet reception rate at distance d from the source node
$\gamma(d)$	SNR at distance d from the source node
f	Frame Length
l	Preamble Length
τ	Absolute time at which input buffer is read and tardiness is computed at sink node

Note that age of the data is computed every time the input buffer is read at the receiver node. In Fig. 8.2, sample $i+1$suffers higher delay than sample $i+2$ sample. In this case sample i is read multiple times at the scheduled read times until the sample $i+1$ arrives at the sink. Alternatively, sample $i+1$ is read only once as the sample $i+2$ arrives immediately after the first read of the sample $i+1$. Thus in this case tardiness of data read from the buffer is more when sample i is present in the buffer compared to when sample $i+1$ is present in the buffer. Since sampling times at the sensor and read from buffer occur independently, the area under the curve divided by the time interval gives the average tardiness of data from the source.

8.1.1.2. Tardiness Measure under Random Delay and Network Packet Loss

Fig. 8.3 illustrates the tardiness process when packets suffer from both random delays and network packet losses. In wireless sensor networks packet losses depend on the network congestion, collisions, and wireless link errors. When a data sample is lost then the prior sample of data is used until the new data arrives at the receiver node. This is not different from the way tardiness is computed in lossless but random delay case in Section 8.1.1. In Fig. 8.3, sample $i+1$ and sample $i+3$ are randomly dropped in the network. It results in increase in tardiness at successive read as sample i is read multiple time until sample $i+2$ arrives. In Fig. 8.3, relative larger area of trapezoid indicates greater tardiness compared to Fig. 8.2.

From Fig. 8.2 and Fig. 8.3 it can be inferred that tardiness measure will be impacted by the random delays and the random losses suffered by the data in the sensor network. Intuitively, it is a function of network delay and packet loss probability, read buffer frequency, and sampling frequency. In Section 8.2 we derive an analytical model for the tardiness of the data in process monitoring sensor networks that relates different network parameters to the age of the data.

8.2 Analytical Model for Tardiness of Data in Process Monitoring Sensor Network

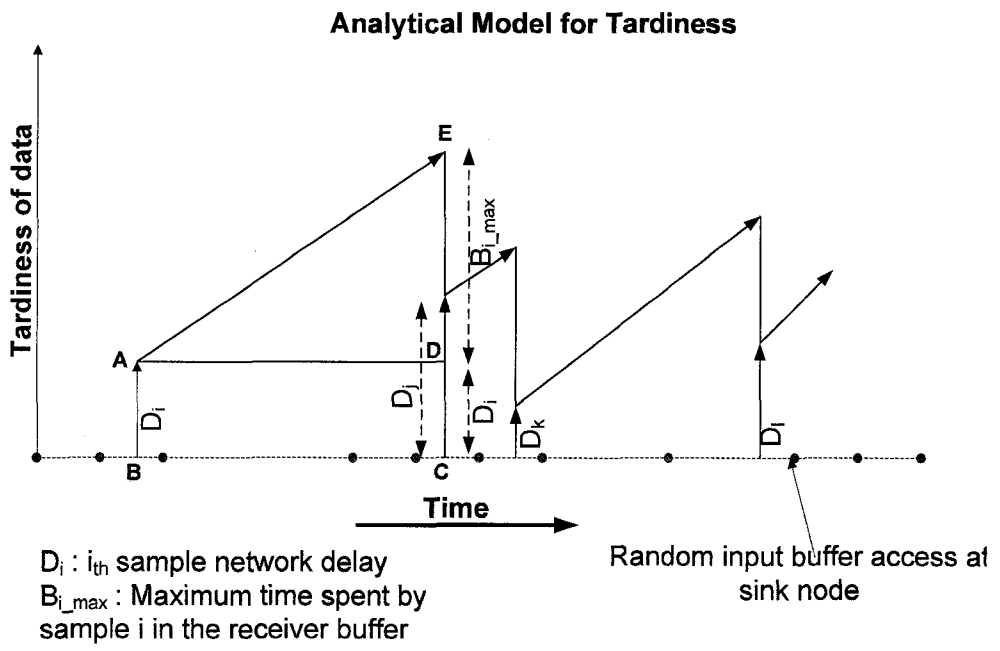


Figure 8.4 Analytical Model for Tardiness Measure

This section derives an analytical model for the tardiness of data from a single source node to a single sink node and then extends it for a multi-source to single sink node scenario. Analytical model for tardiness is a function that relates random network delay, random packet loss probability, and sampling interval to the mean age of the data at the sink node. Table 8.1 shows list of different parameters used for the analytical model. Following assumptions are considered for deriving the tardiness analytical model:

(i) Sensor node samples the environment periodically every 'S' interval and transmit data to a single sink node.

(ii) Sink node may randomly access the input buffer for data.

(iii) We consider a case where source node transmits one sample per packet, so sample and packets are used interchangeably in this chapter.

(iv) All nodes in the sensor network are assumed to be time synchronized for measurement purposes. This does not mean that all the nodes carry out sampling at the same instant.

(v) It is assumed that packets arrives in-order at a sink node.

(vi) Data may get dropped in the network because of wireless link errors, collisions, or network congestion.

8.2.1. Tardiness of Data from a Single Source

Fig. 8.4 illustrates a tardiness computation at the sink node. It considers a scenario when sink node receives samples *i, j, k,* and *l* from a single source node without re-ordering after suffering random delays and random network packet losses. Red solid dots are the random time at which input buffer is accessed for the data at the sink node. Sample *i* which arrives at sink node at time indicated by 'B' is randomly read from the input buffer until sample *j* arrives at time 'C' at a sink node. It is assumed that most recently received sample remains in the input receive buffer until next in-order sample arrives at the sink node. At every read schedule, tardiness of the data read from the buffer is evaluated. Average height of a trapezoid in Fig. 8.4 indicates the average tardiness during each consecutive read attempt until next in-order packet arrives.

In Fig. 8.4, let D_i be the random delay suffered by the sample *i* in the network. Let B_m be the total time spent by the latest sample in the input receive buffer at sink node at the time of m^{th} periodic read. Consider a case when the latest sample read is *i*. Let TD(i,m) be the instantaneous tardiness of data read during m^{th} read attempt when sample *i* is present in the input receive buffer. Let τ be the absolute time at which m^{th} read of input buffer is performed. Then the tardiness of data during read operation performed at time τ is $T(\tau)$, *i.e.,*

$$T(\tau) = TD\ (i,m) = D_i + B_m \tag{8.3}$$

For each arriving sample i at the sink node $0 \le B_m \le B_{i\_max}$, where B_{i_max} is the maximum time spent by sample i in the buffer until a sample j>i arrives. Thus the tardiness experienced at the sink changes with time as shown in Fig. 8.4.

By the application of the law of large numbers, we can assume that long-term time averages can be replaced by the ensemble averages [Bert]. We can compute mean tardiness using graphical argument. Using Fig. 8.4, time average of the tardiness $T(\tau)$ in the interval [0,t]

$$\frac{1}{t}\int_0^t T(\tau)d\tau = \left(\frac{M(t)}{t}\right)\left(\frac{\sum_{i=0}^{M(t)} B_{i\_max} D_i + \frac{B_{i\_max}^2}{2}}{M(t)}\right) \tag{8.4}$$

where M(t) is the total sample arrivals in the interval within [0,t] for which tardiness is computed. Maximum time B_{i_max} that sample i stays in the buffer is determined by the difference in the arrival time of the sample i and sample j where j>i.

$$B_{i\_\max} = (S_j - S_i) + (D_j - D_i) \tag{8.5}$$

Where S_i and S_j are the sampling time at the source node and D_i and D_j are the network delays of sample i and sample j respectively. Under certain conditions such as no network packet losses and when delays are within certain bounds, j=i+1, i.e., adjacent samples are received at the sink node without re-ordering or loss. Replacing B_{i_max} in Eq. 8.2 with Eq. 8.3 and taking $\lim_{t\to\infty}$, we get

$$\lim_{t\to\infty}\frac{1}{t}\int_0^t T(\tau)d\tau = E[T] = \mu\left(\frac{E[L^2]}{2} + E[L]E[D]\right) \tag{8.6}$$

where $L = (j\text{-}i)\cdot S$, such that S is the periodic sampling interval and i and j are the adjacent samples received at the sink node where j>i. L is a r.v. that models loss characteristics of the network. D

is a r.v. that models random delays of different samples that arrive at sink node and μ is the arrival rate of the samples at sink node. Let P_L be the packet loss probability as perceived by the sink node. Then we can determine expectation of mean time interval between two losses and its second moment at the sink node as follows:

$$E[L] = \sum_{i=1}^{\infty} i(P_L^{i-1})(1-P_L)S = \frac{S}{1-P_L} \tag{8.7}$$

$$E[L^2] = \sum_{i=1}^{\infty} i^2 P_L^{i-1}(1-P_L)S^2 = (1+P_L)\left(\frac{S}{1-P_L}\right)^2 \quad \text{and} \tag{8.8}$$

Reciprocal of *E[L]* given by Eq. 8.7 is the mean arrival rate μ, i.e.,

$$\mu = \frac{1-P_L}{S} \tag{8.9}$$

Combining Eq. (8.6)–(8.9), we can determine expectation of the tardiness T, i.e., mean tardiness of the data is:

$$E[T] = E[D] + \left(\frac{1}{2}\right)\left(\frac{1+P_L}{1-P_L}\right)S \tag{8.10}$$

and second moment of tardiness is given by

$$E[T^2] = E[D^2] + (1+P_L)\left(\frac{S}{1-P_L}\right)^2 + 2\left(\frac{S}{1-P_L}\right)E[D] \tag{8.11}$$

From Eq. 8.10, we conclude that tardiness of the data at each read attempt depends on the sampling rate, network delay characteristics and network packet loss probability. Increase in network delay, network packet losses and sample time period results in an increase in the tardiness of the data in sensor networks.

8.2.2 Aggregate Tardiness of Data from Multiple Sources to a Single Sink

The model for tardiness given by Eq. 8.10 is valid for tardiness for the source-sink pair. In sensor networks, many-to-one data transfer is the common data flow scenario. We consider a case when sink node such as fusion node in the network receives data from multiple sources. As tardiness associated with different sources may not be similar, it is necessary to evaluate the aggregate tardiness of the result computed using inputs from multiple sources. Let there be N sensors from where data is aggregated at the sink node. Let W_k be the weight of the data from the sensor k. The weight can be assigned to each sensor according to the criticality of the data generated by each sensor node. When the data gathered from different sensors are equally important, equal weights may be used. Then aggregate weighted tardiness is defined as:

$$E[T_{aggregate}] = \frac{\sum_{k=1}^{N} E[T_k].W_k}{\sum_{k=1}^{N} W_k} \qquad (8.12)$$

8.2.3 Consideration for Re-ordered Packets at a Sink Node

In a sensor network packets may arrive out-of-order because of random delays suffered by them in the network. Depending on the routing algorithm, different packets may follow different paths thus may suffer variable delays. For certain real-time applications, out-of-order packet arrival may not be acceptable and are treated as lost packets. For example, in real-time target tracking applications, it is important to have most recent estimate of the position of the target for real-time tracking and prediction. Therefore, all late arrivals of packets with old information may not be of any use to the application and can be treated as lost. In this section, we consider impact of re-

ordered packets on the tardiness of data and adapt the tardiness model given by Eq. 8.10 to provide accurate estimate of tardiness in presence of packet reordering.

There are two steps involved in developing model that consider re-ordering of packets (i) Estimation of packet loss probability, i.e., P_L perceived by the application at the sink node, and (ii) Estimation of mean delay of the packets that arrive in-order at a sink node, i.e., $E[D|I]$ which is conditional expectation of the delay given that packet arrives in-order.

(i) Estimation of P_L: When packet-reordering is not considered then P_L is equal to the network packet loss probability P_N. However, when packets are treated as lost because of out-of-order arrival then packet loss probability is:

$$p_L = 1 - p_{I/A}(1 - p_N) \tag{8.13}$$

where $P_{I/A}$ is the probability that packet is in-order given that it has arrived at the sink node. P_N is the probability that packets are lost in the network because of wireless link errors. It is assumed that network is not congested and there are no collisions. We estimate conditional probability of in-order arrival, i.e., $P_{I/A}$ as follows:

Consider packet i, $i+1$, and $i+2$ in the order of generation at a source node. Let S_i, S_{i+1}, and S_{i+2} be their generation time such that

$$S_{i+1} - S_i = S_{i+2} - S_{i+1} = S \tag{8.14}$$

Let D_i, be the random network delay suffered by the i^{th} packet then its arrival time A_i is:

$$A_i = S_i + D_i \tag{8.15}$$

By definition, packet i arrives in order if $A_i < A_{i+j}$ where $1 \leq j \leq N$ where N is the total number of arrivals after packet i. A delay distribution of a network provides an estimate of delay suffered by different packets transmitted from the sensor node. Given the delay suffered by a packet generated by the sensor, then that packet arrive in-order if all future packets delays are such that they arrive after the current packet. Note that we consider a scenario when a sensor node periodically generates a packet every S time interval.

Let $f_d(c)$ be the delay distribution of a sensor networks and let P_N be the network packet loss probability because of wireless link errors. Consider different possible ranges of delay of size S that a given packet i may suffer in the network, and for each possible delay range, probability of the packet i arriving in-order is computed as follows:

Case $0 \leq D_i < S$: In this case packet i is not re-ordered, because packet i arrives at the sink node before packet $i+1$ and subsequent packets are generated at the source node. Then probability of in-order arrival of packet i, when packet delay is between 0 and S is:

$$p_{I/A}^{S} = \int_{0}^{S} f_d(c)dc \tag{8.16}$$

Case $S \leq D_i < 2S$: In this case, packet i suffers delay between S and $2S$. In this case packet i arrives in-order if either packet $i+1$ is lost in the network or when packet $i+1$ arrives at sink node such that arrival time $A_i < A_{i+1}$. Note that all other future packets other than $i+1$ will always arrive after packet i as they are generated after the worst case arrival time of packet i, i.e., when delay suffered by packet i approaches $2S$. Therefore, probability of packet i arriving in-order when its delay is between S and $2S$ is:

$$\begin{aligned} p_{I/A}^{2S} &= p_N \int_{S}^{2S} f_d(c)dc + (1-p_N) \int_{S}^{2S} p(D_{i+1} > c - S) f_d(c)dc \\ &= p_N \int_{S}^{2S} f_d(c)dc + (1-p_N) \int_{S}^{2S} (1 - F_D(c-S)) f_d(c)dc \end{aligned} \tag{8.17}$$

where $F_D(a)$ is the CDF of the r.v. D that models delay in the network.

Case $2S \leq D_i < 3S$: Similarly i^{th} sample arrives in-order when both $i+1$ and $i+2$ samples are dropped in the network or if they arrive then their arrival time is greater than A_i. Therefore probability of in-order arrival after considering all possible combinations of arrival and loss of $i+1$, $i+2$ samples is given by:

$$p_{I/A}^{3S} = p_N^2 \int_{2S}^{3S} f_d(c)dc + p_N(1-p_N) \int_{2S}^{3S} (1-F_D(c-S)) f_d(c)dc +$$
$$p_N(1-p_N) \int_{2S}^{3S} (1-F_D(c-2S)) f_d(c)dc + \qquad (8.18)$$
$$(1-p_N)^2 \int_{2S}^{3S} (1-F_D(c-S))(1-F_D(c-2S)) f_d(c)dc$$

Case $k \le D_i < (k+1)S$: For a general case, packet i arrives in order if for all possible combinations of arrival and loss of future k packets, the arrival time $A_i < A_{i+j}$ where $1 \le j \le k$. For lost packets, arrival time is treated as infinite. Therefore

$$p_{I/A}^{(k+1)S} = p_N^k \int_{ks}^{(k+1)s} cf_d(c)dc + p_N^{k-1}(1-p_N) \int_{ks}^{(k+1)s} c(1-F_D(c-S)) f_d(c)dc +$$
$$p_N^{k-1}(1-p_N) \int_{ks}^{(k+1)s} c(1-F_D(c-2S)) f_d(c)dc + \quad \ldots \qquad (8.19)$$
$$+(1-p_N)^k \int_{ks}^{(k+1)s} c(1-F_D(c-S))(1-F_D(c-2S))\ldots(1-F_D(c-kS)) f_d(c)dc$$

We can thus conclude that conditional probability that packet is in-order given that it has arrived is given by summation of all terms given by Eq. 8.16-8.19:

$$p_{I/A} = \sum_{j=1}^{\infty} p_{I/A}^{jS} \qquad (8.20)$$

Next step is to estimate network packet loss probability P_N using realistic wireless link loss model. We use the model presented in [Zu04], which provides packet reception rate as a function of distance from the transmitter when Manchester encoding and NCFSK modulation schemes are used, i.e.,

$$prr[d] = (1 - \frac{1}{2}\exp^{-\frac{\gamma(d)}{2}\frac{1}{0.64}})^{16f-8l} \qquad (8.21)$$

where $\gamma(d)$ is the SNR at a distance d between source node and the sink node, f is the frame size, and l is the preamble length. SNR at a distance d is a function of transmission power.

$$\gamma(d)_{dB} = P_{t\,dB} - PL(d)_{dB} - P_{n\,dB} \qquad (8.22)$$

where $P_{t\,dB}$ is the transmission power, $PL(d)_{dB}$ is the log-normal shadowing path loss radio propagation model [Rapp] and $P_{n\,dB}$ is the noise floor.

Using Eq. 8.21, network packet loss probability between source node and sink node separated by distance d is determined as:

$$p_N = 1 - prr[d] \qquad (8.23)$$

Substituting $P_{I/A}$ and P_N from Eq. 8.20, and Eq. 8.23 in Eq. 8.13 we get total packet loss probability P_L due to packet reordering and network losses, i.e.,

$$p_L = 1 - \left(\sum_{j=1}^{\infty} p_{I/A}^{jS} \right) (prr[d]) \qquad (8.24)$$

(ii) Estimation of mean delay of packets given that they arrive in-order E[D|I]: In tardiness model given by Eq. 8.10, *E[D]* is the expectation of the delay of all the packets that arrive at the sink node irrespective of their order of arrival. However, for applications that cannot tolerate reordered packets, the delay of reordered packets should not be included in the estimation of the expected delay at the sink node. However when we estimate expected delay based on the delay distribution of a network, it also includes delay for packets that arrive out-of-order at the sink node. Therefore, given the delay distribution $f_d(c)$ of the network, it is necessary to include correction to estimate expected delay for all packets that arrive in-order. We can estimate the conditional expected delay *E[D|I]* given that packet arrives in-order with the similar approach used for computing $p_{I/A}$ given by Eq. 8.20. We get

$$E[D \mid I] = \frac{\sum_{j=1}^{\infty} p_{D/I}^{jS}}{p_{I/A}} \qquad (8.25)$$

where

$$p_{D/I}^{jS} = p_N^{j-1} \int_{(j-1)s}^{js} c f_d(c)dc + p_N^{j-2}(1-p_N) \int_{(j-1)s}^{js} c(1-F_D(c-S)) f_d(c)dc +$$

$$p_N^{j-2}(1-p_N) \int_{(j-1)s}^{js} c(1-F_D(c-2S)) f_d(c)dc + \quad \ldots \qquad (8.26)$$

$$+(1-p_N)^{j-1} \int_{(j-1)s}^{js} c(1-F_D(c-S))(1-F_D(c-2S))\ldots(1-F_D(c-(j-1)S)) f_d(c)dc$$

Eq. 8.24 and Eq. 8.25 gives the new estimates of packet loss probability and the conditional expectation of delay respectively that considers impact of packet reordering. Now mean tardiness as given by Eq. 8.10 can be written as follows that is valid for estimating mean tardiness for applications that does not tolerate reordered packets. Therefore, new estimate of mean tardiness is:

$$E[T'] = E[D \mid I] + \frac{1}{2}\left(\frac{2 - \left(\sum_{j=1}^{\infty} p_{I/A}^{jS} \right)(prr[d])}{\left(\sum_{j=1}^{\infty} p_{I/A}^{jS} \right)(prr[d])} \right) S \qquad (8.27)$$

8.3 Verification of Analytical model for Tardiness

Analytical model for estimation of mean tardiness given by Eq. 8.27 for the tardiness measure is validated using simulation results. The model considers impact of random network packet losses, network delay, packet reordering, and periodic sampling interval on the tardiness of the data between a source-sink pair. Source node is configured to generate samples periodically after S interval. We consider uniform network packet loss probability P_N and exponential distribution for the packet delays with mean delay D. Fig. 8.5 compares the simulation and tardiness model results for single source case under different network packet loss and delay conditions. Table 8.2 shows data corresponding to results shown in Fig. 8.5. For these experiments we consider an

application that does not tolerate packet re-ordering, i.e., one that uses most recent measurement available in the input receive buffer, and all late out-of-order packets are treated as lost. In Fig. 8.5, model is verified for three cases 1-3 in order of increase in packet re-ordering. Intuitively, degree of re-ordering depends on the standard deviation of the delays suffered by packets between source-sink pair. Case 1 is an example of no packet re-ordering where sampling interval $S = 5.0$ seconds and standard deviation and mean delay is 0.1 seconds. However for Case 2, packet reordering increases where sampling interval $S = 5.0$ seconds and mean delay and standard deviation of exponential delay distribution is 20.0 seconds. We consider high network delays based on the observation that in sensor networks age of the data can be in tens of seconds [Exp]. For Case 3, $S=5.0$ seconds, and mean delay and standard deviation of delay is 50.0 seconds. Case 3 corresponds to highest degree of packet re-ordering and Case 1 corresponds to no packet reordering. As seen in Fig. 8.5, simulation results and model results for tardiness are in close agreement with each other for all three cases of varied degree of reordering. For case 1 of

Table 8.2 Tardiness Model Verification under different network loss and delay conditions, Case 1 Mean exponential delay=0.1 second, Case 2 Mean exponential delay= 20.0 seconds, and Case 3 mean exponential delay = 50.seconds when sampling interval S= 5.0 seconds

Simulation & Model	*P=0.0 Tardiness (Seconds)*	*P=0.1 Tardiness (Seconds)*	*P=0.3 Tardiness (Seconds)*	*P=0.5 Tardiness (Seconds)*	*P=0.7 Tardiness (Seconds)*	*P=0.9 Tardiness (Seconds)*
Model Case1	2.57	3.14	4.76	7.62	14.26	47.6
Simulation Case 1	2.6	3.15	4.74	7.6	14.26	47.6
Model Case 2	14.45	15.55	18.45	23.14	32.71	71.2
Simulation Case 2	12.66	13.57	16.01	20.03	28.43	64.63
Model Case 3	19.9	21.17	24.59	30.05	41.05	83.84
Simulation Case 3	23.62	25.2	29.37	35.99	49.07	96.79

Tardiness Model Verification under different Network Loss and Network Delay S=5.0 sec. and Exponential Delay Distribution Case 1: Mean Delay D=0.1 sec., Case 2: Mean Delay=20.0 sec., Case 3: Mean Delay=50.0 sec.

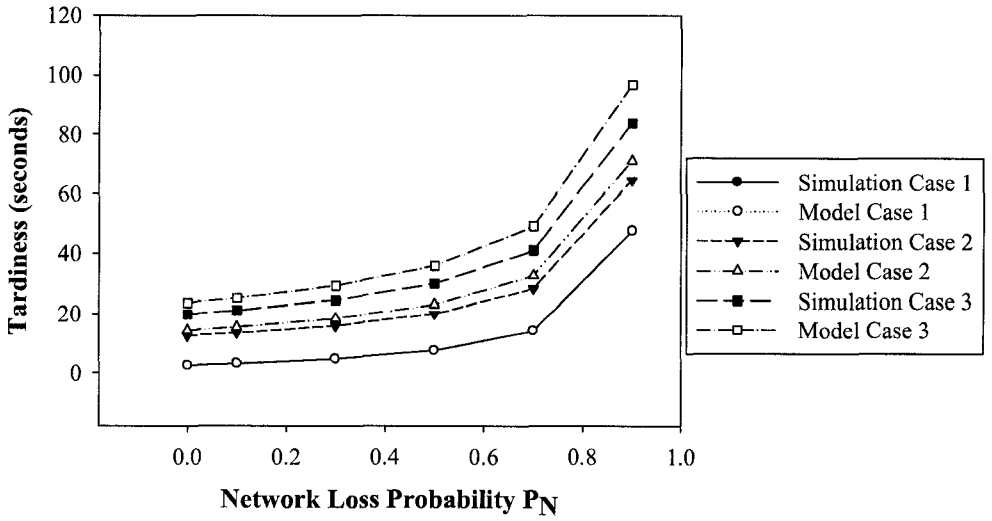


Figure 8.5 Verification of Tardiness model under for different Network Loss Rates and Network Delays, Case 1 corresponds to random losses and no packet reordering, Case 2 and Case 3 corresponds to random network losses and high to very high degree of reordering

Table 8.3 Impact of packet delay, and loss probability on the packet reordering when sampling interval S=5.0 seconds

Mean Exponential Delay (seconds)	*p=0.0 (Fraction of out-of-order packets)*	*p=0.4 (Fraction of out-of-order packets)*	*p=0.7 (Fraction of out-of-order packets)*
0.1	0	0	0
5	0.192	0.133	0.075
10	0.344	0.26	0.166
20	0.49	0.404	0.286
50	0.65	0.577	0.461
80	0.71	0.651	0.547
100	0.744	0.68	0.584
150	0.788	0.735	0.648

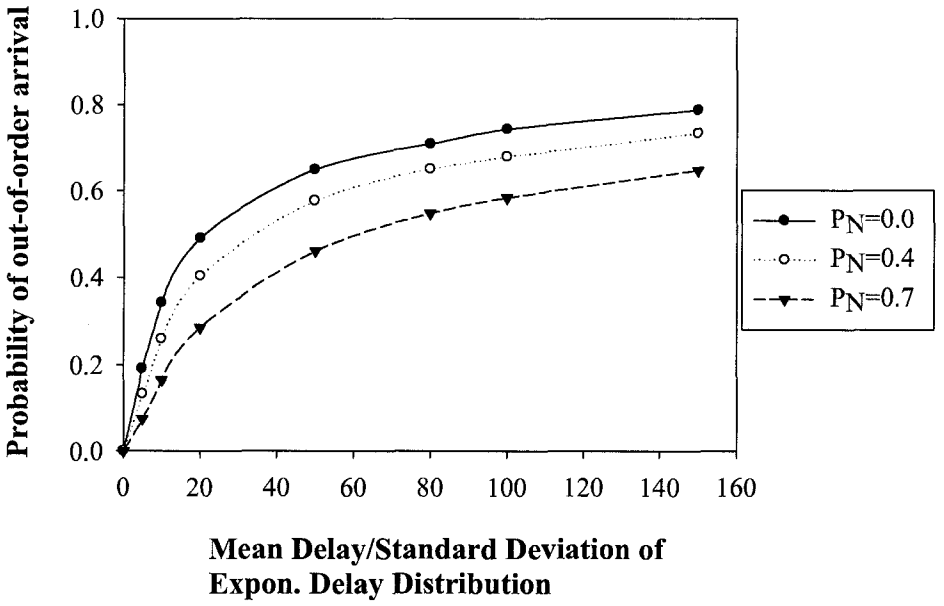


Figure 8.6 Impact of mean delay and standard deviation on the packet re-ordering under different network packet loss conditions

Table 8.4 Tardiness model verification under varying sampling interval S

Sampling Interval (seconds)	*Tardiness based on Simulation (seconds)*	*Tardiness based on Model (seconds)*
1	5.61	6.79
5	12.66	14.47
10	18.09	19.89
20	26.15	27.53
40	38.63	39
60	49.47	49.74
80	59.81	59.99

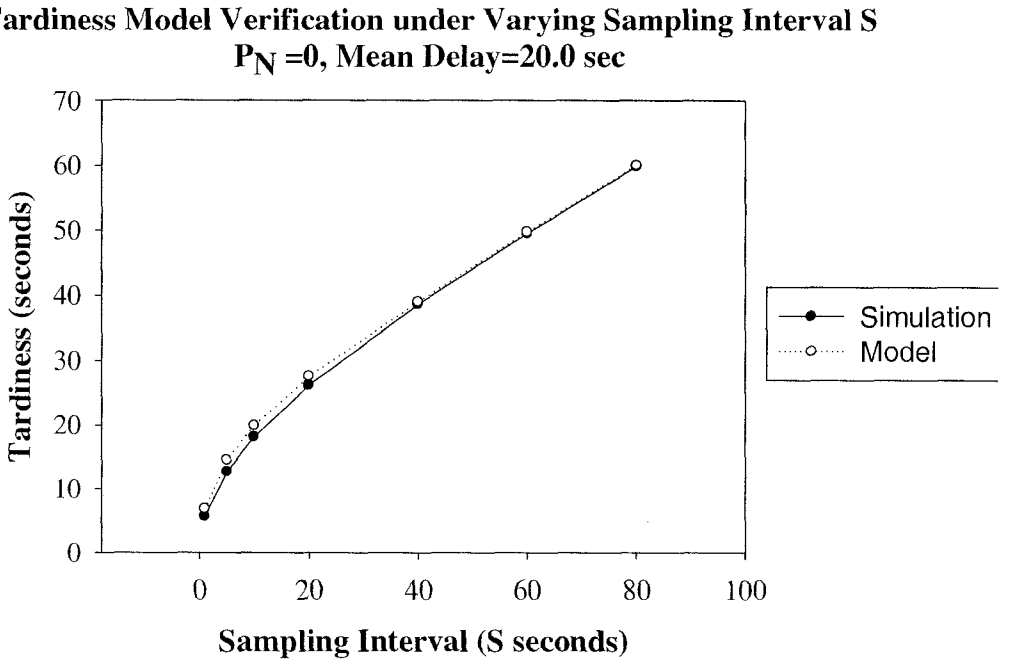


Figure 8.7 Verification of tardiness model with varying sampling interval

no packet reordering, simulation and model results overlap each other. As seen in the figure, for a given network packet loss probability P_N packet reordering may lead to significant increase in the tardiness of the data. Fig. 8.6 corroborates the impact of standard deviation of the network delay on the fraction of the packets that arrives out-of-order. Table 8.3 shows data corresponding to results shown in Fig. 8.6. For a given network packet loss probability, fraction of packets that arrive out-of-order, increases with increase in standard deviation. Note that exponential delay distribution is considered in Fig. 8.6. In the second set of experiment for model verification, sampling interval S is varied for a given network packet loss probability and mean delay, and its impact on the tardiness is studied for a source-sink pair. Fig. 8.7 shows the simulation and model results of tardiness under varying sampling interval S. Table 8.4 shows data corresponding to results shown in Fig. 8.7. In this experiment P_N =0.0, and mean exponential delay D=20.0 seconds. As seen in the figure, tardiness increases with the increase in sampling interval and model results are in close agreement with the simulation results.

8.4 Tradeoffs between Energy Consumption and Tardiness of Data

Most of the sensor networks are energy constrained. In this section, we investigate the tradeoffs between energy consumption and the tardiness of the data. As seen in Section 8.2 packet loss probability, network delays, and sampling interval are the key factors that impact the tardiness of the data. Application can achieve desired tardiness bound between source and a sink node by configuring sampling interval, by controlling network packet losses (by adjusting transmit power for example), or network delays (via routing algorithms, adjusting sleep schedule etc.). Configuring each of these parameters to achieve desired tardiness may impact the total energy consumption in the sensor network. Fig. 8.8 shows the simulation network used for performance analysis. It consists of 221 nodes in a grid of 15m x 15m sensing field. All data to the sink node at the center of the grid, indicated by red dot at X=7, Y=7. In this section, we consider single hop transmission. We consider a scenario when sink node periodically reads input buffer for the most recent available data with read interval *'R'*. Each packet transmitted by a

Table 8.5 Parameters to determine packet reception rate for MICA2 platform [Zu04, Chp]

Parameter Name	*Parameter Value*
Preamble	18 bytes
Frame Length	36 bytes
Encoding	Manchester (2:1)
Modulation	NCFSK
Packet Time	23.3ms
Noise Floor	-105.0dBm
Output Power	-20dBm to 10dBm (Chipcon CC1000 radio 433/315 MHz)
Path Loss Exponent	4.7 (outdoor)
Shadowing Standard Deviation	3.2
Close in reference distance	1m
Close in reference power	55dBm

Table 8.6 CC1000 radio current consumption at different transmission power [Chp]

Tx Power (dBm)	Current Consumption (mA)	Tx Power (dBm)	Current Consumption (mA)
-20	5.3	-5	8.9
-19	6.9	-4	9.4
-18	7.1	-3	9.6
-17	7.1	-2	9.7
-16	7.1	-1	10.2
-15	7.4	0	10.4
-14	7.4	1	11.8
-13	7.4	2	12.8
-12	7.6	3	12.8
-11	7.6	4	13.8
-10	7.9	5	14.8
-9	7.9	6	15.8
-8	8.2	7	16.8
-7	8.4	8	20
-6	8.7	9	22.1
		10	26.7

source node to the sink node may suffer random losses in the network. In this chapter we consider network traffic such that it does not lead to packet losses because of network congestion. All packet losses in the network are considered to be due to wireless link errors. We use a wireless link loss model given by Eq. 8.21 and Eq. 8.23 for simulating wireless link losses in the simulator. Table 8.5 shows different operating parameters used for determining packet loss rate as a function of distance between source and a sink node for a MICA2 hardware platform. As seen the table, we consider outdoor wireless environment and packet length is 54 bytes. In Fig. 8.8, maximum distance between any source-sink pair is less than 10m and all nodes can communicate

with the sink node in a single hop. Therefore, for this case we ignore delay suffered by packets between source and sink node. Moreover, all packets arrive in-order in a single hop communication. Three sets of experiments are performed for understanding tradeoffs between energy consumption and tardiness.

In the first experiment, all source nodes in Fig. 8.8 sample their environment periodically with sampling interval S=5.0 seconds. Moreover, transmission power of each source node is kept constant, i.e., T_x=7dBm. Average loss rate is computed between all source-sink pairs for a network topology shown in Fig. 8.8 using Eq. 8.21, and Eq. 8.23 over 500 iterations. Fig. 8.9(a) and Fig. 8.9(b) shows results for the experiment 1.

Fig. 8.9(a) shows tardiness of the data from each source to the sink node at X=7, and Y=7 in Fig. 8.8. In this case average tardiness for all source-sink pairs is 2.81 seconds with standard deviation of 1.39 seconds. In Fig. 8.9(a), results show that tardiness of different source-sink pairs varies widely between 2.0 and 10.17 seconds in a grid. Fig. 8.9(b) shows the corresponding energy consumption at each node in the network after 500 seconds of simulation time. Total

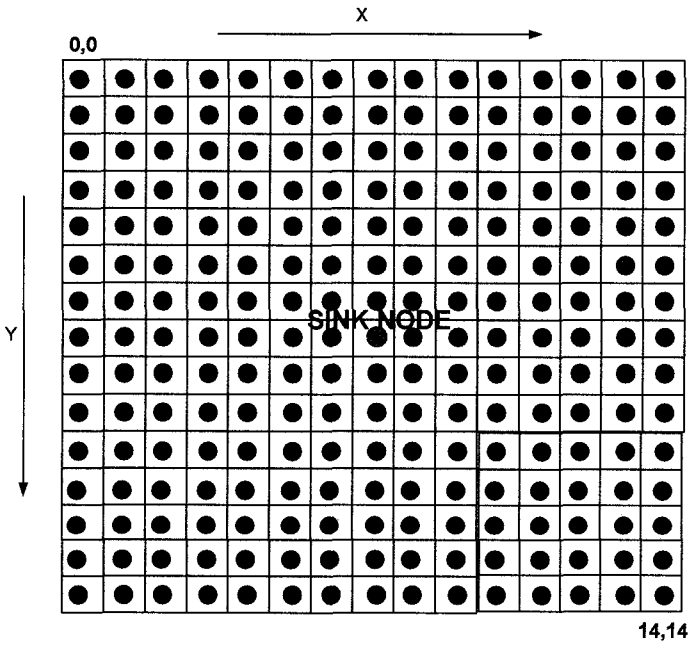
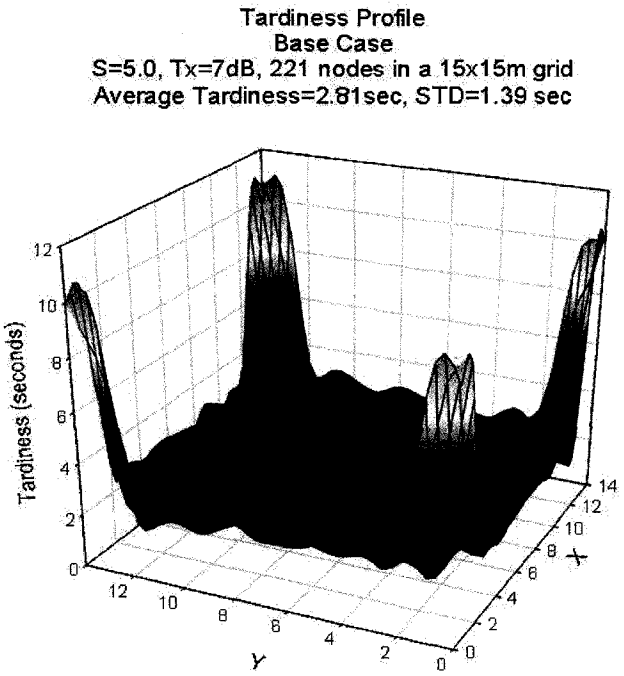


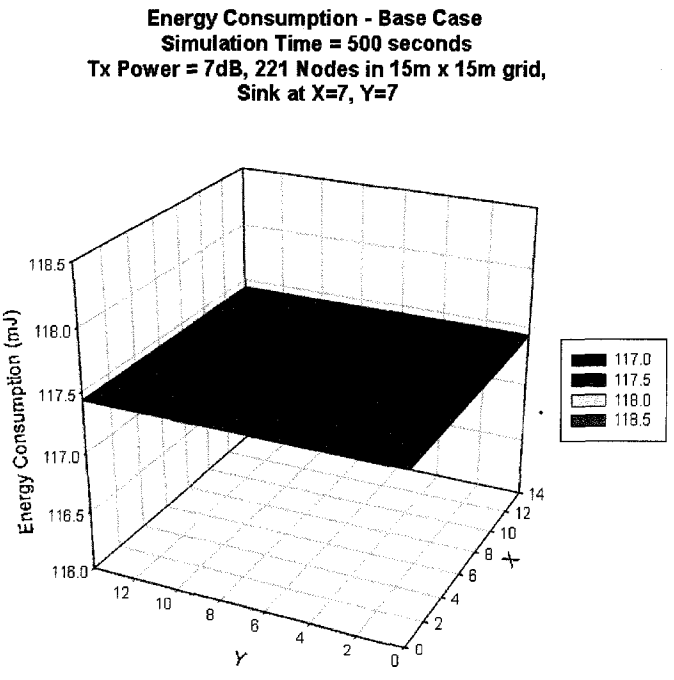
Figure 8.8 Network with 225 nodes in a 15m x 15m grid, sink at X=7, Y=7

energy consumption is computed using Table 8.6 that gives current consumption for a given output transmission power for the CC1000 radio, which is used by the MICA2 hardware platform. For energy consumption computation it is considered that operating voltage of MICA2 based platform is 3V. As seen in Fig. 8.9(b), when sampling rate and transmission power of all nodes is same, then each node consumes constant amount of energy, i.e., 117.432mJ in 500 seconds.

Consider a data fusion application where it is necessary to have data with similar tardiness at the sink node. Using Eq. 8.27, desired tardiness can be achieved by adapting network delay, total packet losses or sampling interval. In the second experiment, sampling interval S of each source node is adapted to achieve the constant mean tardiness of 2.81 seconds between all source-sink pairs. In this case transmission power of each node is kept constant, i.e., T_x=7dBm. Fig. 8.10(a) shows the sampling interval of all source nodes in a grid to achieve the desired tardiness. As seen in the Fig. 8.10(a), sensor nodes that are closer to the sink may sample environment at a slower rate. However, as the distance between source node and sink node increases, sampling interval decreases, resulting in higher sampling frequency. The intuition behind increasing the sampling frequency with distance is that when there is high network packet loss probability then sending larger number of packets has the potential to deliver more information to the sink node hence results in decreasing tardiness. It is important to note that this study does not consider occurrence of hot-spots, i.e., network congestion in the network due to increase in sampling rate at certain regions of the network. In all experiments, link bandwidth is not exceeded and there are no packet collisions. All losses are due to wireless link errors. Fig. 8.10(b) shows the energy consumption at all nodes in a grid when sampling intervals are configured to achieve the desired tardiness. As seen in the figure, energy consumption shows significant amount of variation depending on sampling frequency of the node. Nodes that are closer to the sink node consume significantly less amount of energy compared to nodes at corner of the network grid. Average energy consumption



(a)

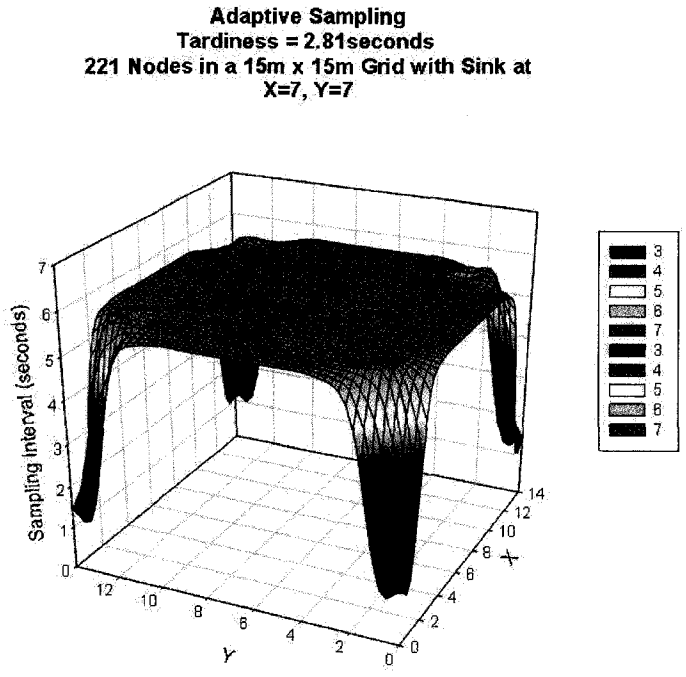


(b)

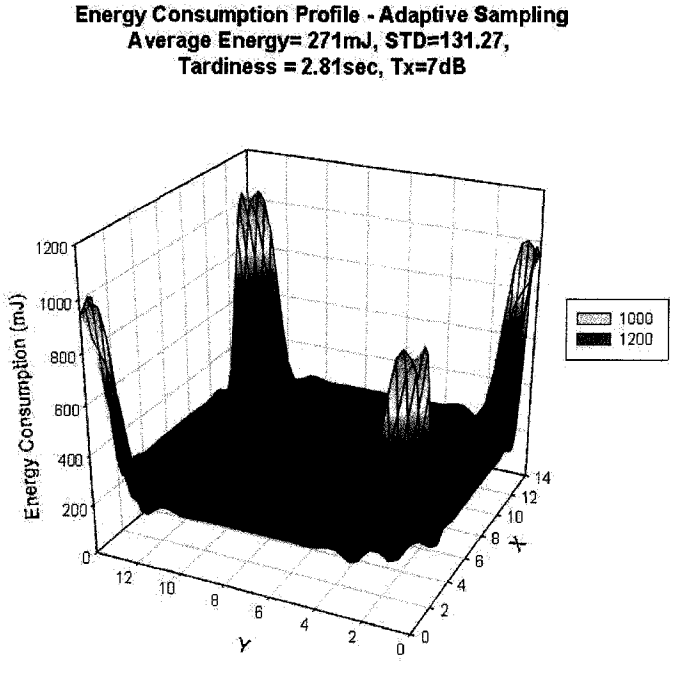
Figure 8.9 Tardiness and energy consumption profile of a sensor network field

is 271 mJ with high standard deviation of 131.27 mJ in 500 seconds of simulation time.

In the third experiment, transmission power of each source node is varied while keeping sampling interval S constant, i.e., S=5.0 seconds to achieve the constant tardiness of 2.81 seconds for each source-sink pair. By changing the transmission power of the source node, packet loss rate between source-sink pair can be adjusted to achieve the desired tardiness. Aforementioned, packet delivery rate depends on different environment factors such as distance between source and sink node, path loss exponent, and the transmission power. As a first step, for each source node, acceptable network packet loss probability P_N is estimated using tardiness Eq. 8.27 to achieve the desired tardiness of 2.81 seconds. In the next step transmission power is estimated to achieve the acceptable P_N (calculated in the first step) using wireless loss model proposed in [Zu04] given by Eq. 8.21 and Eq. 8.23. Fig. 8.12(a) shows the transmission power of each source node such that mean tardiness between source-sink pair is 2.81 seconds. MICA2 Chipcon CC1000 radio can transmit at power between -20dBm and 10dBm at 433/315 MHz [Chp]. In Fig. 8.11(a), all nodes that require transmission power below -20dBm are configured to operate at -20dBm. In this experiment none of the source node needs transmission power greater than 8.75 dBm. For energy consumption computations, we have used the power consumption at each node which is not the same as the transmitted power. The relationship between actual power required and transmitted power for MICA2 specified in [Chp] was used assuming that MICA2 mote operates at 3V. Note that nodes that are closer to the sink node require low transmission power compared to nodes at farther distance from the sink node in order to maintain same tardiness between source-sink pair. Note that we have assumed that increase in transmission power does not lead to interference with the neighboring node. This is possible when a TDM based MAC protocol is used for packet transmission. Fig. 8.11(b) shows the energy consumed by the different nodes in the grid during 500 seconds of simulation time. Note that average transmission energy is 77.42 mJ with standard deviation 26.55 mJ. This is significantly better than the case when sampling interval was adjusted to meet the desired tardiness. Fig. 8.12 shows the total energy

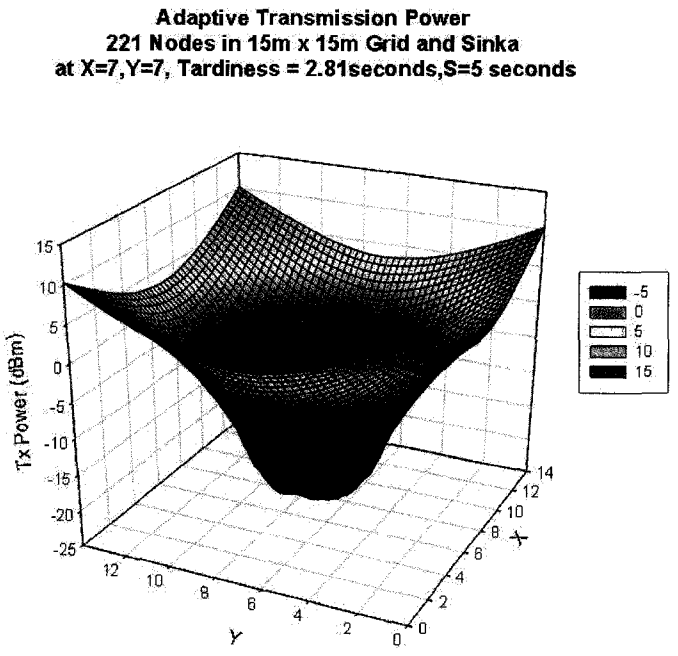


(a)

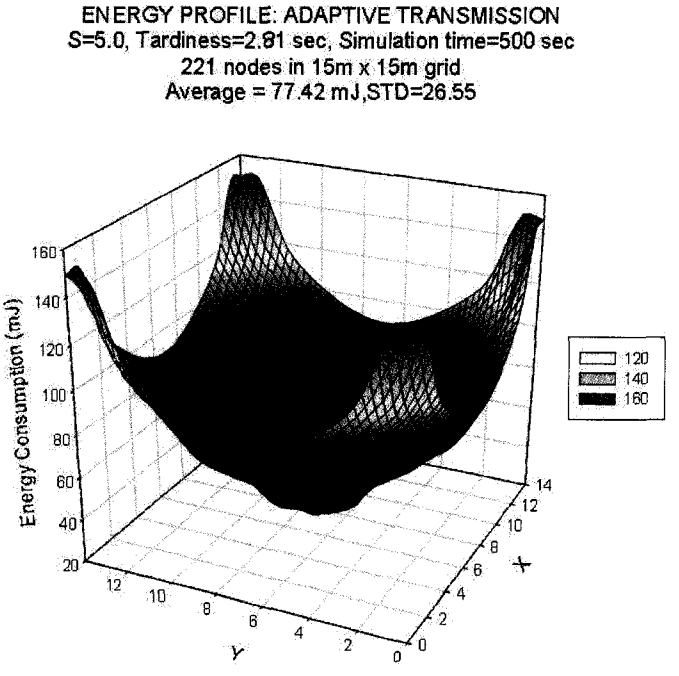


(b)

Figure 8.10 Application of tardiness measure in adapting sampling rate of source nodes to achieve the desired Tardiness 2.81 seconds with standard deviation =0



(a)



(b)

Figure 8.11 Application of tardiness measure in adapting transmission power of source nodes to achieve the desired tardiness 2.81 seconds with standard deviation =0

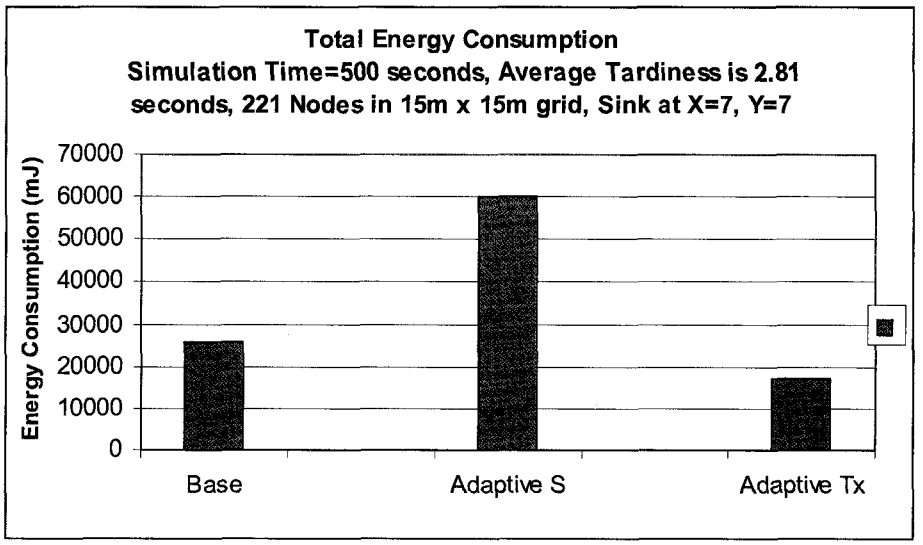


Figure 8.12 Comparison of total energy consumption in a 500 second interval for three different sensor network configurations with similar average tardiness characteristics

consumed in 500 seconds of simulation time for all three experiments when similar average tardiness was achieved. As seen in the figure, adaptive transmission power performs best in terms of energy efficiency while maintaining same tardiness for all source link pairs. Adaptive sampling does not help in conserving energy but in this case all source sink pairs have the same tardiness of 2.81 seconds. This study illustrates one of the applications of the tardiness model to optimize network configuration parameters to achieve tardiness goals of an application.

8.5 Remarks

In sensor networks, network dynamics, networking protocols have significant impact on the tardiness of the data delivered to the processing nodes. The application requirements and the characteristics of the process being monitored will impose a limit on the tardiness that can be tolerated. This chapter presented an analytical model for the tardiness that relates age of the data used for computation with network characteristics such as the network delays and loss rates, and transmission power, as well as the sampling frequency. Analytical model was validated using

simulation results. Tardiness model is then used to generate tardiness profile of a sensor network. The use of model to evaluate alternate strategies to achieve tardiness targets required by an application was also illustrated. Simulation results for the parameters considered show that adaptive transmission power scheme is more energy efficient compared to adaptive sampling for meeting the desired tardiness requirements of the end users. Application of tardiness measure is demonstrated in evaluating the performance of three different routing protocols, i.e., random routing, rumor routing and zonal rumor routing (ZRR). There are many potential applications of the tardiness measure such as configuring sleep/active schedules of the MAC layer, sampling rate and transmission energy to meet real-time requirements. The tardiness process characteristics effectively capture the impact of network characteristics on the data used at sink nodes for decision making. Application-specific tardiness bound requirements can be used to tune the network parameters to achieve sensor network application's real-time requirements.

Chapter 9

IMPACT OF MULTI-HOP COMMUNICATION ON TARDINESS OF DATA IN WIRELSSS SENSOR NETWORKS

Multi-hop communication is required in a sensor network to route data from the source node to the sink node using multiple relay nodes. Due to limited transmission power of the node transmitter and need for long range transmission, multi-hop communication becomes unavoidable. Introduction of multiple relay nodes in the path from source to the sink node introduce added delay; alternatively there may be decrease in total energy consumption for communication between source and sink nodes. The focus of this chapter is on investigating the impact of multi-hop communication on tardiness of data between source and sink nodes. This chapter addresses following questions:

Given the transmission power of the sensor nodes

1. What's the impact of multiple relay nodes on the tardiness between source and sink nodes?
2. What are the tradeoffs between energy consumption and tardiness of data in multi-hop sensor networks?

Section 9.1 derives the tardiness model for the multi-hop path between source and the sink node. Section 9.2 illustrates the impact of multi-hop communication on tardiness based on the analytical

model. Section 9.3 demonstrates the application of tardiness measure to compare routing protocols based on the characteristics of the path selected by different routing protocol between source and the sink nodes. Concluding remarks are presented in Section 9.4.

9.1 Multi-Hop Communication Analysis

Fig. 9.1(a) shows a direct communication scenario between source node A and a sink node C separated by distance *d*. Alternatively, Fig. 9.1(b) shows multi-hop communication scenario between source node A and a sink node C through a relay node B at the center. For direct communication case as shown in Fig. 9.1(a) the tardiness of data between source and sink node separated by distance d is given by Eq. 8.10 in Chapter 8 when no packet reordering is present in a direct communication. Alternatively, when multiple intermediate relay nodes are used for the communication then the effective loss rate between source and sink decreases due to decrease in the distance between two adjacent nodes in the path while keeping other factors such as Tx

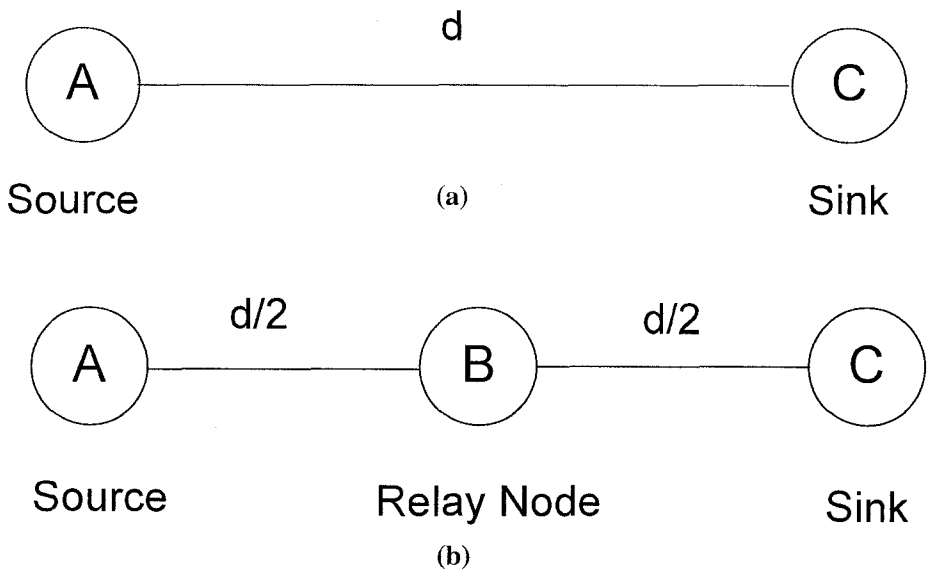


Figure 9.1 Single-hop and multi-hop communication (a) Single-hop communication, (ii) Multi-hop communication

power and channel characteristics same. Using packet reception rate model given by Eq. 8.21 in Chapter 8, packet loss during network errors between Node A and Node B and between Node B and Node C is given by:

$$p_{AB} = p_{BC} = 1 - P_{RR}(d/2) \tag{9.1}$$

where $P_{RR}(d/2)$ is determined using Eq. 8.21. Then end-to-end packet loss rate between node A and C in Fig. 9.1(b) due to network errors is given by

$$p_{AC} = 1 - \left(P_{RR}(d/2)\right)^2 \tag{9.2}$$

When there are total n nodes in the path between source and sink nodes (inclusive of source and sink nodes), separated by equal distance then the end-to-end loss rate due to network error is given by

$$p_{source-\sin k} = 1 - \left(P_{RR}(d/n-1)\right)^{n-1} \tag{9.3}$$

Intermediate relay nodes have the potential to introduce random delay because of processing/communication overheads at relay nodes. Let each node in the path from source to sink introduces uniform random delay between *MIN* and *MAX*. When propagation delay is ignored then mean delay between source and sink nodes in a multi-hop network is the sum of delay overhead due to relay nodes.

$$E[D] = (n-1)(\frac{MIN + MAX}{2}) \tag{9.4}$$

In Eq. 9.4, it is assumed that source node and all intermediate relay nodes introduce random delay. When source samples with sampling interval S, where source and sink nodes are separated by distance d, and no packet-reordering is considered then using Eq. 8.10 in Chapter 8, Eq. 9. 3 and Eq. 9.4, the tardiness between a source and sink node is given by

$$E[T]_{MULTI-HOP} = (n-1)(\frac{MIN+MAX}{2}) + \left(\frac{1}{2}\right)\left(\frac{2-\left(prr(d/n-1)\right)^{n-1}}{\left(prr(d/n-1)\right)^{n-1}}\right)S \quad \text{where } n\geq 2 \text{ (9.5)}$$

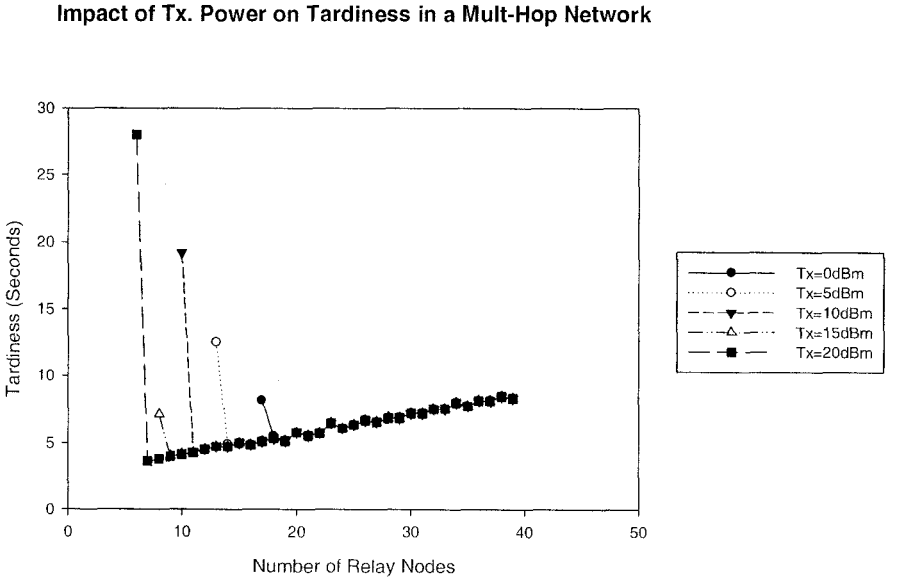
when average delay suffered by packets at intermediate relay node including source node is D_{avg} then expectation of tardiness is given by

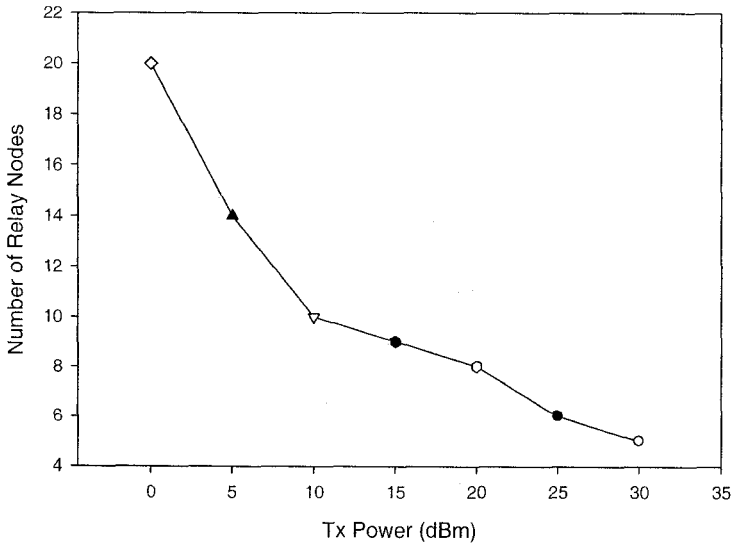
$$E[T]_{MULTI-HOP} = (n-1)(D_{avg}) + \left(\frac{1}{2}\right)\left(\frac{2-\left(prr(d/n-1)\right)^{n-1}}{\left(prr(d/n-1)\right)^{n-1}}\right)S \quad \text{where } n\geq 2 \text{ (9.6)}$$

9.2 Impact of Multi-Hop Communication on Tardiness

In this section we use the model given in Eq. 9.5 to understand the impact of multi-hop communication on that tardiness and the total energy consumption in the sensor network.

For a given node transmission power and distance *d* between source and sink node, as the number of nodes increases in a path between source and sink nodes the tardiness of data may show monotonic decrease and increase within certain range of number of hops as shown in Fig. 9.2(a). It is assumed that at each intermediate node There is a critical inflection point in terms of number of nodes in the path up to which tardiness decreases monotonically for a given transmission power and distance between source and sink nodes and after that it monotonically increases. Fig. 9.2(b) shows the inflection point for tardiness in terms of number of hops for a scenario when distance between source and sink node is 100m and transmission power is varied between 0dBm and 30dBm. As seen in Fig. 9.2(b) as the transmission power increases inflection point decreases. Inflection point indicates the optimal number of intermediate nodes that are necessary such that tardiness is minimum between source and a sink node. When number of total nodes is below inflection point then that result in increase in end-to-end network error loss probability between source and sink node with a tendency to increase the tardiness. Moreover,





(b)

Figure 9.2 (a) Impact of multi-hop communication on tardiness under varying node transmission power when distance between source and sink node = 100m, S=5.0 seconds, (b) Optimal number of relay nodes to achieve minimum tardiness for varying transmission power of a node.

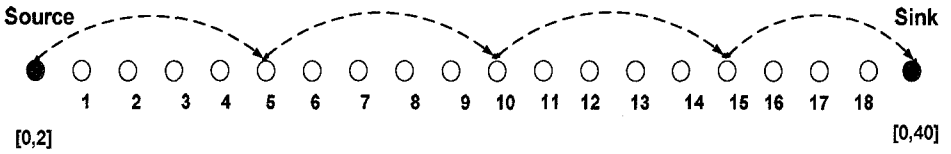


Figure 9.3 Multi-hop linear network topology (distance in meters)

less number of intermediate nodes means that less amount of time is wasted in the network due to processing/communication delay within intermediate nodes thus resulting in decrease in tardiness. When number of intermediate nodes is greater than inflection point then time wasted at intermediate nodes due to processing/communication overheads is the dominating factor. Thus there exist an optimal number of nodes such that competing factors end-to-end network loss probability and time wasted in the intermediate nodes results in minimum tardiness.

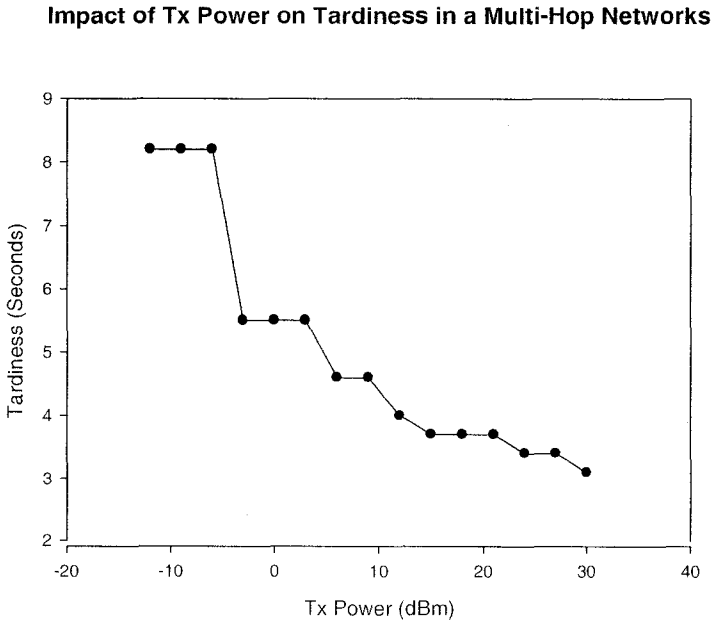
In the second set of experiments we are interested in understanding the tradeoffs between tardiness and energy consumption under varying transmission power. Fig. 9.3 shows the linear multi-hop network topology used for communication between a source and sink node. In Fig. 9.3 source and sink nodes are separated by a distance of 38m. All white colored nodes acts as relay nodes between source and sink node. A subset of the relay nodes may be used for forwarding data for a given transmission power. A relay node is selected such that it is the farthest node from the source and the loss rate between source and relay node remains 0. For example, for a certain transmission power relay nodes 5, 10, and 15 are selected for forwarding data between source and sink node. In this case these particular nodes are selected because they form the minimum set of nodes in the path that are necessary to maintain 100% reliable delivery between source and sink nodes without any retransmissions in the network.

Fig. 9.4(a) shows impact of transmission energy on the tardiness of data between source and sink node shown in Fig. 9.3. As the transmission power increase tardiness decrease monotonically when optimal number of intermediate relay nodes are selected for forwarding in the path between source and a sink node. When transmission power increases then optimal number of intermediate

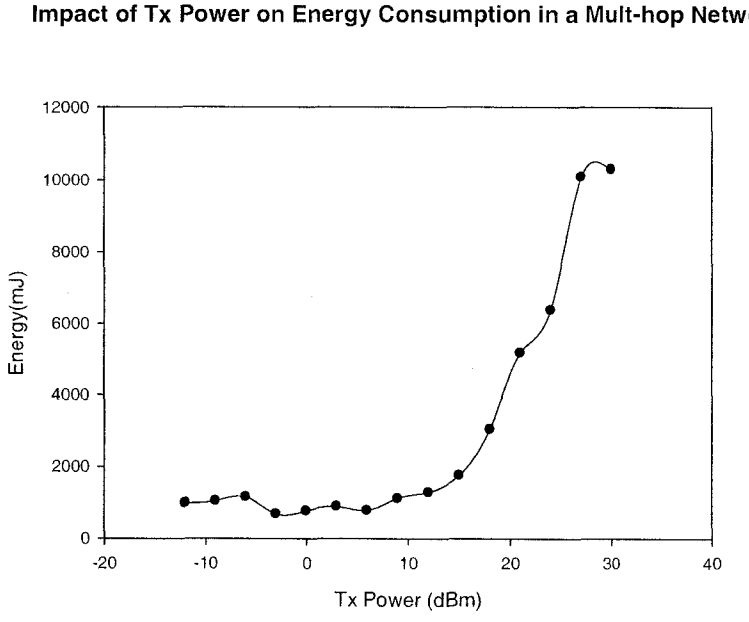
nodes required for reliable forwarding of data decreases monotonically; It results in decrease in total processing and communication overheads at relay nodes hence the decrease in tardiness of data between source and a sink node. Fig. 9.4(b) shows the amount of energy consumed during reliable delivery of data from source to a sink node using multi-hop network under different

Table 9.1 Impact of Tx. Power on Tardiness and Energy consumption in a multi-hop network, for 100 packets generated with packet time 23.3ms, operating voltage = 3V for CC1000 radio (extrapolated Tx power)

Transmission Power (dBm)	*Tardiness (seconds)*	*Energy Consumption (mJ)*
-12	8.2	8.2
-9	8.2	8.2
-6	8.2	8.2
-3	5.5	5.5
0	5.5	5.5
3	5.5	5.5
6	4.6	4.6
9	4.6	4.6
12	4	4
15	3.7	3.7
18	3.7	3.7
21	3.7	3.7
24	3.4	3.4
27	3.4	3.4
30	3.1	3.1



(a)



(b)

Figure 9.4 Impact of Transmission power in multi-hop network (a) Impact of varying transmission power on tardiness on a multi-hop path between source and sink node. (b) Impact of transmission power on energy consumption. Source and Sink nodes are separated by 38 m

transmission power conditions. In Fig. 9.4(b) it is observed that there exists an optimal transmission power range at which energy consumption is minimum for reliable delivery of data between source and sink node using multi-hop network. This observation was also previously made in [Ba02a, Ba02b]. Table 9.1 shows data corresponding to results shown in Fig. 9.4. In this chapter a case is considered when source node generates 100 packets of 54 Bytes in 500 seconds of simulation time with sampling interval S=5.0 seconds with packet transmission time of 0.0233 seconds[Chp]. In Fig. 9.4(b) when Tx power is - 3dBm, total minimum energy 679.42mJ is consumed by all the relay nodes including source node in the network for reliable transmission of data from source node to the sink node without any retransmissions. In Fig. 9.4(a) tardiness corresponding to Tx power -3dBm is 5.5 seconds and minimum tardiness of 3.1 seconds is at higher transmission power 30dBm. At current transmission power, minimum tardiness may be achieved by increasing the sampling rate, i.e., by decreasing sampling interval S as shown by tardiness model in Eq. 8.27. Using Eq. 8.27 it is determined that with S=0.2 seconds while keeping all other factors same, tardiness of 3.1 seconds can be achieved for Tx power -3dBm. However, it is determined that total amount of energy that is required to achieve the desired tardiness at lower transmission power -3dBm is significantly greater than what is required when transmission power is 30dBm. It is determined that for -3dBm case total energy consumed is 16985 mJ compared to 10302 mJ for 30dBm case during 500 seconds of simulation time.

9.3 Comparison of Routing Protocol Performance Using Tardiness Measure

This section compares the performance of three routing protocols (i) random routing, (ii) rumor routing, and zonal rumor routing [Ba05c, Br02a] using tardiness measure.

A network grid of 215x215 is considered in which 10, 000 sensor nodes are randomly distributed in the grid. All nodes that are within its 5m radial distance of a particular node are considered as its neighbors. Each sensor node has a 5m sensing range, i.e., all nodes present

within 5m radial distance from the sink location are able to detect that event. 100 events are randomly generated in the grid and multiple sensor nodes can detect the same event Moreover, each node can also detect multiple events. 25 agents are randomly generated in the network that propagates the information about the events in the network until their TTL expires. 1500 random

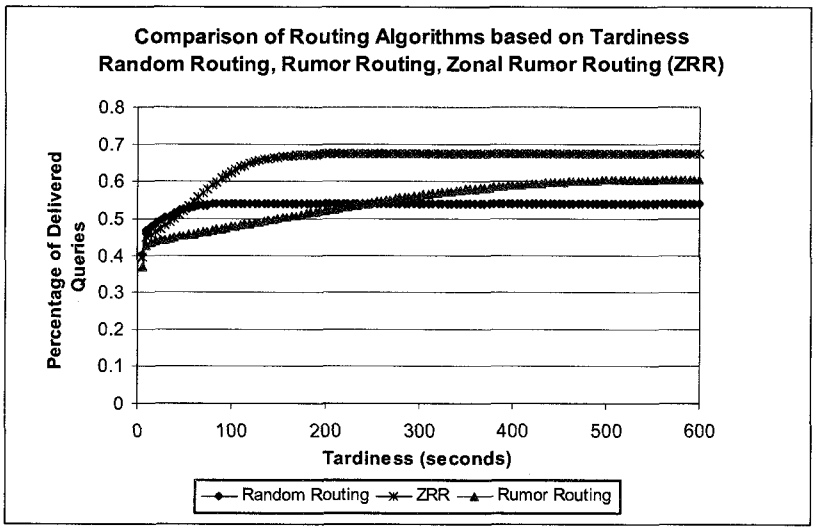


Figure 9.5 Comparison of routing protocols performance for real-time sensing applications using tardiness measure

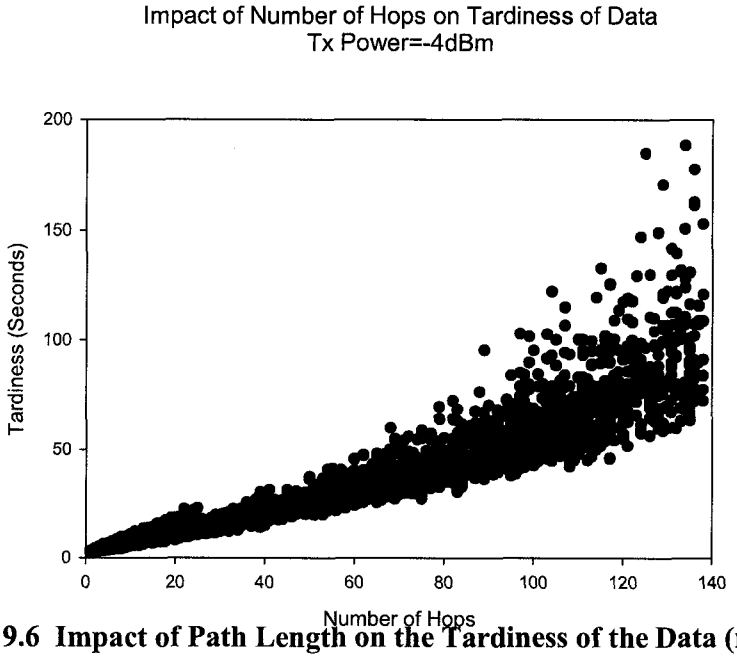


Figure 9.6 Impact of Path Length on the Tardiness of the Data (maximum distance between two adjacent nodes is 5m)

queries are generated in the network for the random events. All three routing protocols mentioned above are used to spread the event as well as queries in the networks. Three routing protocols differ in their approach for selecting the next hop for agent and query forwarding. A query is said to be delivered to the event when query finds a node that has the information about the requested event. At this time event information can be routed from the source of the event to the sink node where query was generated. It is assumed that queries and agents are reliably delivered to their next hop. However, once the path is determined between event source and query source, data is transmitted unreliably over lossy wireless link whose loss characteristics are determined using Eq. 8.21 given in Chapter 8. Depending on the routing protocols, different numbers of queries may be delivered to the desired event source and moreover characteristics of the path between event source and query source may vary in terms of number of relay nodes and the loss characteristics resulting in different tardiness for the delivered queries. At each intermediate relay node packet suffers random delay between 100ms and 500ms. Therefore, we use tardiness of the delivered queries to compare performance of the routing protocols. Fig. 9.5 shows the performance of different routing algorithms in terms of tardiness of the data from event source to the query source. As seen in the figure, in case of random routing, 53 % of the total queries are delivered with average tardiness of 7.8 seconds. Moreover, 90% of the delivered queries have tardiness less than 18 seconds. Alternatively, in case of traditional rumor routing, 60% of the total queries were delivered and their average tardiness is 61 seconds. In this case 90% of the delivered queries have tardiness below 181 second. Comparing random routing and rumor routing we can say that rumor routing delivers higher percentage of queries. However, the extra queries that are delivered leads to significantly higher tardiness of the data in rumor routing algorithm. In case of zonal rumor routing algorithm 70% of queries are delivered with average tardiness of 26.32 seconds. Zonal rumor routing algorithm delivers larger number of queries to the event source compare to traditional rumor routing algorithm and at the same time paths lengths between query

source and the event source are significantly smaller in case of ZRR compared to traditional rumor routing leading to significantly lower tardiness.

The key reason for lower tardiness in case of ZRR is the selection of shorter paths compared to the traditional rumor routing algorithm. Fig. 9.6 shows the impact of number of hops in a path on the tardiness measure. As seen in the Fig. 9.6, tardiness increases with the increase in path length. However, tardiness does not increase linearly with the increase in number of hops. Note that results shown in Fig. 9.6 are different from the results shown in Fig. 9.2(a) that also investigates the impact of number of hops on tardiness. The difference between two experiment results is that in Fig. 9.2(a) distance between source and sink is considered constant and number

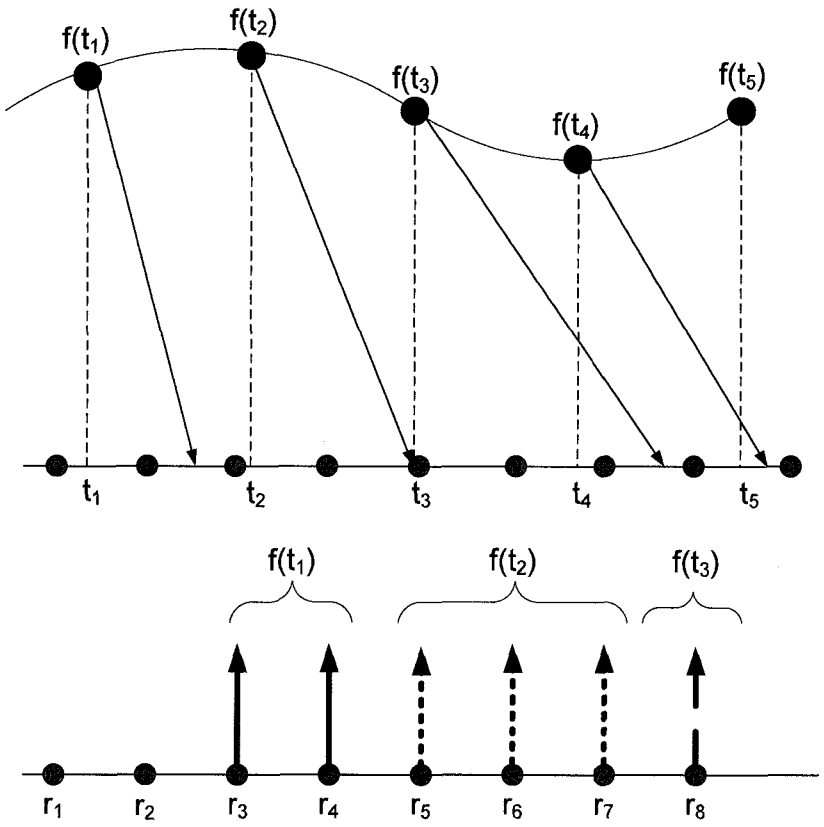


Figure 9.7 Effect of tardiness on end applications

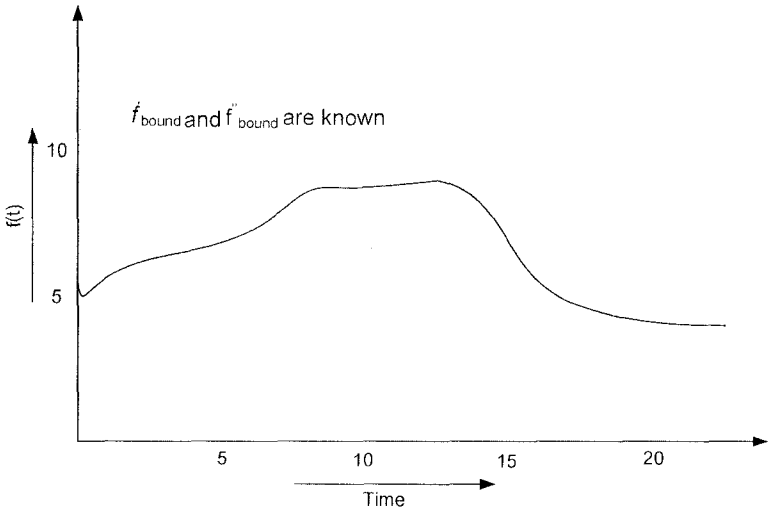


Figure 9.8 Physical process under observation

of hops is changed between source and sink node. Alternatively, in Fig. 9.6 distance between source and sink is not considered constant. As seen in Fig. 9.6, tardiness may vary for two paths with similar number of hops because of difference in the loss characteristics of the paths resulting in different tardiness.

9.4 Impact of Tardiness on the Accuracy of the Results

After the tardiness is computed for the data, its impact on the accuracy of the end results can be evaluated. The impact depends on the nature of processing, and varies from application to application. One of the factors that determine the utility of the tardiness measure is the rate of change of information at the sensor node. If the process under observation is a rapidly changing process then tardiness beyond a certain threshold will render received data useless or error prone [Ma03]. In this case, it is desirable to sample it at a faster rate. Alternatively, when the phenomenon under observation is changing slowly; the application is likely to able to tolerate a higher tardiness in the received data. Fig. 9.7 considers a temperature monitoring and control

application. It consists of sensor nodes that periodically samples environment for any temperature variation and transmit the sampled data to the remote sink nodes. Based on the available temperature information, sink node can generate an actuating signal for controlling the temperature of the remote environment. For such closed-loop applications it is necessary that the actuating signal is generated based on the current state of the remote environment Due to network dynamics such as network delays and losses, the data available at the sink nodes may not represent the current state of the environment. As seen in Fig. 9.7, data generated at time t_1, i.e., $f(t_1)$ at sensor node is read from the receiver buffer at time r_3 and r_4. When $f(t_1)$ data is read from the buffer at time r_4, it provides false indication that temperature has not changed at the remote location. However, as seen in the figure actual data available at the remote location is $f(t_2)$ which is still in transit. Similarly, sink node reads $f(t_2)$ at time r_5, r_6, and r_7. However, sink node is unaware of change of temperature from $f(t_2)$ to $f(t_4)$ during interval r_7-r_5. This has the potential to introduce an error by generating an actuating signal to control the temperature based on the old information.

Now we demonstrate the use of tardiness measure to provide error bounds in the end computation. By using such a bound, tardiness model can be used to limit the error in the end results by adapting the sampling rate at the sensor node or by using alternative routing protocol that selects low delay paths. Consider a process that represents environment, for which first and second derivatives of the signal being monitored are bounded, i.e., we know the limits on the first and second derivative of a process. This is a valid assumption for many processes because depending on the physical nature of the process there are always natural bounds because of fundamental principles of physics or environment that keeps the process transitions rate within bound. With that assumption lets assume we know $\left|f'_{bound}\right|$, and $\left|f''_{bound}\right|$ bounds for a process under observation, i.e.,

$$f'_{bound} \; s.t. \; f'(t) \le \left|f'_{bound}\right| \; t \in [0,\infty) \text{ and} \qquad (9.6)$$

$$f''_{bound} \ s.t. \ f''(t) \leq \left| f''_{bound} \right| \ t \in [0, \infty) \qquad (9.7)$$

Using principles of Taylor series we know that

$$f(t) \leq f(t_0) + (t - t_0)\left| f'_{bound} \right| + R_1 \qquad (9.8)$$

where

$$R_1 \leq \frac{\left| f''_{bound} \right|}{2!}(t - t_0)^2 \qquad (9.9)$$

$f(t)$ is the current state of the environment at time t and $f(t_0)$ is the most recent data available at the sink node.

Then the error bound on the data read from the buffer at time t_0 is given by E_t

$$E_t = f(t) - f(t_0) \leq (t - t_0)\left| f'_{bound} \right| + \frac{\left| f''_{bound} \right|}{2!}(t - t_0)^2 \qquad (9.10)$$

$t - t_0$ is the tardiness of the sample $f(t_0)$ used at time t at the sink node.

If application knows the acceptable error threshold then using Eq. 9.10, acceptable tardiness bound for $t - t_0$ can be evaluated. Once bound on the tardiness is known then that may be used to adjust sampling rate 'S' in mean tardiness Eq. 9.5 or use alternate routing protocols to decrease mean delay to meet the acceptable mean tardiness.

Consider a process shown in Fig. 9.8 for which we know f'_{bound} and f''_{bound} during the life of the process. Using Eq. 8.10, Eq. 8.11 in Chapter 8 and Eq. 9.10, maximum mean error $E_{\max}$ is given by

$$E_{\max} = E[T]\left|f'_{bound}\right| + \frac{E[T^2]\left|f''_{bound}\right|}{2} \qquad (9.11)$$

Alternatively, when maximum acceptable error is known then bound on the acceptable mean tardiness $E[T]$ is given by

$$E[T] \leq \frac{E_{\max}}{\left|f'_{bound}\right|} \qquad (9.12)$$

9.5 Remarks

This chapter investigated the impact of multi-hop communication on the tardiness. An analytical model for tardiness presented in Chapter 8 is adapted to consider multi-hop communication. It is shown that for a given distance between source and a sink node and given transmission power, there exist an optimal number of relay nodes at which tardiness is minimum. It is also shown that for a given distance between source and sink node there exist an optimal transmission power at which total energy consumption at all nodes in the path from source to the sink node is minimum. Application of tardiness measure in comparing the performance of three different routing protocols, i.e., random routing, rumor routing and zonal rumor routing (ZRR) is demonstrated using simulation. Simulation results shows that ZRR has higher delivery rate than the traditional rumor routing algorithm and at the same time tardiness of the events is significantly lower in case of ZRR when compared rumor routing algorithm. This chapter also demonstrates the impact of tardiness on the accuracy of the end results and derives an error estimation model that provides the upper bound on the maximum error that can be introduced in the end results due to tardiness of the data.

Chapter 10

CONCLUSIONS

This dissertation proposed and demonstrated the effectiveness of the application-aware transport services in meeting heterogeneous QoS requirements of the end users for mission-critical sensor network applications. This research was focused on two key areas for providing application-aware transport services, i.e., (i) Design and development of application-aware transport protocols for broadband sensor networks, and (ii) Development of a model for evaluating freshness of data in sensor networks. CASA was used as an example application for demonstrating the suitability of application-aware services for such systems.

Under application-aware transport protocols for broadband sensor networks this dissertation proposed an overlay network based application-aware congestion control protocol. The key goal of an application-aware congestion control protocol is to consistently meet data quality and bandwidth QoS requirements for the data from the radar node to the end users under dynamic network conditions on a best-effort network such as Internet. Application-aware congestion control achieves this goal by performing rate adaptation while considering both bandwidth and data quality requirements of an individual end user. In this approach source node selects and schedules most suitable subset of the weather radar data for transmission within bounded time for a particular end user under varying available bandwidth conditions. The framework for

application-aware congestion control is extended to support multiple heterogeneous end users in CASA network.

An Overlay network based DOOM (Deterministic Overlay One-to-Many) application-aware multicast protocol was proposed that performs application-aware congestion control for multiple heterogeneous end users. The current implementation of DOOM protocol supports CASA application and is used for distributing high-bandwidth radar data to multiple heterogeneous end users such as emergency managers and researchers. Effectiveness of the DOOM protocol in meeting QoS requirements of heterogeneous end users of CASA is demonstrated using a Planetlab based testbed and an emulation testbed. Performance results shows that time multiplexed scheduling scheme along with on-the-fly data selection scheme at source node delivers significantly better quality data to end users when compared non-application-aware congestion control scheme. It is also shown that DOOM streams are friendly to each other as well as TCP cross-traffic streams until the minimum bandwidth requirements of all end users are met. This dissertation then further explores the suitability of performing application-aware processing at intermediate nodes in the overlay network between source and destination. An application-aware packet-marking scheme was proposed to enable in-network processing using overlay networks to help enhance the QoS received by the end users on a best-effort network.

The proposed packet marking scheme marks the packets based on the available bandwidth and the suitability of the data present in the packet for an available bandwidth. A variant of DOOM protocol was implemented that performs application-aware congestion control at intermediate overlay nodes. During network congestion DOOM protocol performs application-aware drops and forwarding based on the marking of the packets and the available bandwidth at intermediate overlay nodes. It uses token-bucket based scheme to achieve the desired transmission rate which is determined using TRABOL congestion control protocol in the current implementation. Performance of the degree of application-aware processing in the overlay networks is analyzed. Planetlab based results show that application-aware congestion control

using packet-marking scheme was most effective in meeting QoS requirements of end users. This dissertation then proposed an AWON (Application-aWare Overlay Networks) architecture framework for the deployment of different types of application-aware services on overlay networks. The suitability of the AWON architecture is demonstrated for the deployment of application-aware transport protocol services for CASA application.

Another contribution of the dissertation is development of a model for evaluating freshness of the data in sensor networks. Freshness of data acts as a key QoS parameter for evaluating quality of the data used for computations and decision making in many mission-critical sensor network applications. Tardiness model was derived that relates the freshness of the data to the mean network delay, network losses perceived by the end users, and the sampling rate at the sensor nodes. This model is validated using simulation results. The model can be used to determine how the application-specific tardiness can be achieved by adjusting transmission power or sampling rate at the sensor nodes. However, it is more energy efficient if desired tardiness is achieved by increasing transmission power compared to increase in sampling rate. The tardiness model was adapted to also consider late arrival packets as lost for certain real-time sensor network applications. Impact of the multi-hop communication on tardiness is analyzed. It is shown that for a given distance between source and a sink node and given transmission power, there exist an optimal number of relay nodes at which minimum tardiness of data can be achieved. It is also shown that for a given distance between source and sink node there exist an optimal transmission power at which total energy consumption at all nodes in the path from source to the sink node is minimum. This dissertation then demonstrated application of the tardiness measure in comparing real-time performance of different routing protocols for sensor networks based on the tardiness of the data delivered to the sink nodes. It is also shown that tardiness measure can be an effective in estimating maximum bound on errors in the end results.

Future goal is to demonstrate the suitability of tardiness measure in estimating error in end computations when data from multiple weather radars is combined in CASA. Tardiness measure

can be used to compare performance of different transport and MAC protocols. Alternatively, AWON architecture framework can be used to develop application-aware many-to-one protocol for gathering data from multiple weather radars at a processing node in CASA. Moreover, it is desired to demonstrate the effectiveness of the AWON architecture for streaming voice and video data using overlay networks.

BIBLIOGRAPHY

[Ak02] I.F. Akyildiz, Su Weilian, Y. Sankarasubramaniam, and E. Cayirci, "A Survey on Sensor Networks," in IEEE Comm. Magazine, Vol 40, Issue 8, Aug. 2002

[Ak04] I.F. Akyildiz, and I. Kasimoglu, ``Wireless Sensor and Actor Networks: Research Challenges," Ad Hoc Networks Jour. (Elsevier), Vol. 2, No. 4, pp. 351-367, Oct. 2004

[Am06]Amir, Y, Danilov, C., Goose, S., Hedqvist, D., Terzis, A., "An Overlay Architecture for High Quality VOIP Streams," in IEEE Trans. On Multimedia, Vol. 8, Issue 6, pp. 1250-1262, Dec. 2006

[An00] D. Andersen, D. Bansal, D. Curtis, S. Seshan, and H. Balakrishnan, "System Support for Bandwidth Management and Content Adaptation in Internet Applications," 4th USENIX OSDI Conf., San Diego, California, Oct. 2000

[An01] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris,"Resilient Overlay Networks," Proc. of 18th ACM SOSP, 2001

[Ba02a]S. Banerjee, and A. Misra, "Minimum Energy Paths for Reliable Communication in Multi-Hop Wireless Networks," Proc. of ACM MobiHoc, June 2002

[Ba02b]S. Banerjee, and A. Misra, "Adapting Transmission Power for Optimal Energy Reliable Multi-Hop Wireless Communication," Technical report, UMIACSTR- 2002-103 and CS-TR 4424, Department of Computer Science, University of Maryland, College Park, MD 20742, USA, Nov. 2002

[Ba03]S. Banerjee, C. Kommareddy, K. Kar, B. Bhattacharjee, and S. Khuller, "Construction of an Efficient Overlay Multicast Infrastructure for Realtime Applications," in Proc. IEEE INFOCOM, June 2003

[Bg02]S. Bangolae, A.P. Jayasumana, V. Chandrasekar, "TCP-friendly Congestion Control Mechanism for a UDP-based High-Speed Radar Application and Characterization of its Fairness," Proc. 8th IEEE Int. Conf. on Communication Systems (ICCS), Singapore, pp.164-168, Nov. 2002

[Bg03a]S. Bangolae, A.P. Jayasumana, V. Chandrasekar, "Performance of Memory-based TCP-friendly Rate Adaptation Based On Loss Algorithm for a Real-time Radar Application," Proc. IEEE Local Computer Network Conf. (LCN), pp. 319-322, 2003

[Bg03b]S. Bangolae, A. P. Jayasumana and V. Chandrasekar, "Gigabit Networking: Digitized Radar Data Transfer and Beyond," Proc. IEEE International Conf. on Communications (ICC'03), Vol. 1, pp. 684-688, Anchorage, 2003

[Bg03c]S. Bangolae, "A TCP-friendly Congestion Control Mechanism for High Bandwidth Radar Application," MS Thesis, Colorado State University, Fall 2003

[Ba05a]T. Banka, B. Donavan, V. Chandrasekar, A. P. Jayasumana, J. F. Kurose, ``Data Transport Challenges in Emerging High-Bandwidth Real-Time Collaborative Adaptive Sensing Systems," Poster/Demo Session, IEEE INFOCOM 2005, Miami, FL, March 2005

[Ba05b]T. Banka, A. Maroo, A.P. Jayasumana, V. Chandrasekar, N. Bharadawaj, S.K. Chittababu, "Radar Networking: Considerations for Data transfer Protocols and Network Characteristics," Proc. 21st Int. Conf. on Interactive Information Processing Systems (IIPS) for Meteorology, Oceanography, and Hydrology, American Meteorological Society (AMS), 19.11. Jan. 2005

[Ba05c]T. Banka, G. Tandon, and A. P. Jayasumana, "Zonal Rumor Routing for Wireless Sensor Sensor Networks," in Proc. of IEEE Intl. Conf. on Information Tech.: Coding and Computing, ITCC 2005, Las Vegas, NV April 2005

[Ba05d]T. Banka, P. Lee, A. P. Jayasumana and V. Chandrasekar, "Performance Evaluation of Application-Aware Transport Services for High-Bandwidth Sensor-Actuator Networks," Proc. 7th International Information Technology Conference (IITC 2005), Colombo, Sri Lanka, Nov. 2005, pp. 93-101

[Ba06]T. Banka, P. Lee, A. P. Jayasumana, and V. Chandrasekar, "Application Aware Overlay One-to-Many Data Dissemination Protocol for High-Bandwidth Sensor Actuator Network," in Proc. of IEEE COMSWARE 2006, New Delhi, India, Jan. 2006

[Ba07a] T. Banka and A.P. Jayasumana, "Impact of Network Dynamics on Tardiness of Data in Sensor Networks," Proc. Second IEEE/Create-Net/ICST International Conference on Communication System Software and Middleware (COMSWARE 2007), Bangalore, India, Jan. 2007

[Ba07b] T. Banka, P. Lee, A.P. Jayasumana and J.F. Kurose, "An Architecture and a Programming Interface for Application-Aware Data Dissemination Using Overlay Networks," Proc. Second IEEE/Create-Net/ICST International Conference on Communication System Software and Middleware (COMSWARE 2007), Bangalore, India, Jan. 2007

[Bert] D. Bertsekas, and R. Gallanger, "Data Networks,", 2nd Edition, Prentice Hall

[Bo04] M. Bouzeghoub, and V. Peralta, "A Framework for Analysis of Data Freshness," in Proc. of Intl. Workshop on Information Quality in Information Systems, IQIS 2004, pp 59-67, Paris, France, 2004

[Bh01] S. Bhatnagar, B. Deb, and B. Nath, "Service Differentiation in Sensor Networks," in Proc. of 4th Int. Symp. on Wireless Personal Multimedia Comm., Sept. 2001

[Bl98] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "An Architecture for Differentiated Services," RFC 2475, IETF, Dec. 1998

[Br94] R. Braden, D. Clark, and S. Shenker, "Integrated Services in the Internet Architecture: An Overview," RFC 1633, IETF, June 1994

[Br97] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin, "Resource ReSerVation Protocol," RFC 2205, IETF, Sept. 1997

[Br99] H. M. Briceno, S. Gortler, L. Mcmillan, "NAIVE - Network Aware Internet Video Encoding," 7th ACM Int. Multimedia Conf., pp. 251-260, Oct. 1999

[Br01] V. N. Bringi, V. Chandrasekar, "Polarimetric Doppler Weather Radar: Principles and Operations," Cambridge University Press, Aug. 2001

[Br02a]D. Braginsky, and D. Estrin, "Rumor Routing Algorithm for Sensor Networks," In Proc. 1st ACM Workshop on Sensor Networks and Applications, pp. 22–31, Atlanta, GA, Oct. 2002

[Br02b]R. Braynard, D. Kosti´c, A. Rodriguez, J. Chase, and A. Vahdat, "Opus: An Overlay Peer Utility Service," Proc. of the 5th Intl. Conf. on Open Architectures and Network Programming (OPENARCH), June 2002.

[Ca03] M. Carson, D. Santay, "NIST Net: A Linux-based Network Emulation Tool," ACM SIGCOMM Computer Communications Review, 33(3): pp. 111–126, 2003

[Ch86] V. Chandrasekar, V. N. Bringi, and P. J. Brockwell, "Statistical properties of Dual-polarized Radar Signals," in Proc. Of 23rd Conf. on Radar Meteorology, Amer. Meteor. Soc., Snow-mass, CO, 193–196, 1986

[Ch01] V. Chandrasekar and A.P. Jayasumana, "Radar Design and Management in a Networked Environment," in Proc. of SPIE, vol. 4527, pp. 142–147, 2001

[Ch05] V. Chandrasekar, Y. G. Cho, D. A. Brunkow, and A.P. Jayasumana, "Principles of Radar Operation over the Internet: The VCHILL," Proc. 21st Int. Conf. on Interactive

Information Processing Systems (IIPS) for Meteorology, Oceanography, and Hydrology, American Meteorological Society (AMS), 19.14., Jan. 2005

[Chi05]S. Chien, B. Cichy, A. Davies, D. Tran, G. Rabideau, R. Castano, R. Sherwood, D. Mandl, S. Frye, S. Shulman, J. Jones, S. Grosvenor, "An Autonomous Earth-Observing Sensorweb," in Proc. of IEEE Intelligent Systems, Vol. 20, Issue 3, pp. 16-24, May-June 2005

[Ch89] D. Chiu and R. Jain, "Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks," Jour. of Computer Networks, pp.1-14, June 1989

[Ch00] Y. Chu, S. Rao, H. Zhang, "A Case For End System Multicast," in Proc. of ACM Sigmetrics, Santa Clara, CA, June 2000

[Ch04] Y.G. Cho, "A High Bandwidth Radar Operation over the Internet: Signal Analysis, Network Protocols and Experimental Validation," PhD Thesis, Colorado State University, Spring 2004

[Chi04] C.F. Chiasserini, and M. Garetto, "Modeling the Performance of Wireless Sensor Networks," Proc. of IEEE Infocom 2004, Vol. 1, March 2004

[Cha02]V. Chandramohan, K. Christensen, "A First Look at Wired Sensor Networks for Video Surveillance Systems," in Proc. of the 27th Annual IEEE Conference on Local Computer Networks, pp.728-729, Nov. 2002

[Che04] D. Chen, and P. K. Varshney, "QoS Support in Wireless Sensor Networks: A Survey," Proc. of the 2004 Intl Conf. on Wireless Networks (ICWN 2004), Las Vegas, NV, June 21-24, 2004

[Cho03]J. Cho, and H. Garcia-Molina, "Effective Page Refresh Policies for Web Crawlers," ACM Trans. On Database Systems, Vol. 28, Issue 4, pp. 390-426, December 2003

[Cl90] D. D. Clark, and D.L. Tennenhouse, "Architectural Considerations for a New Generation of Protocols," Proc. ACM SIGCOMM 1990, Computer Comm. Review, 20(4):200-208, Sept. 1990

[De03a]B. Deb, S. Bhatnagar, and B. Nath, "ReInForM: Reliable Information Forwarding using Multiple Paths in Sensor Networks," in 28th IEEE Conf. on Local Computer Networks (LCN 2003), Bonn, Germany, Oct. 2003

[De03b]B.Deb, S. Bhatnagar, and B. Nath, "Information Assurance in Sensor Networks," in 2nd Intl. Workshop on Wireless Sensor Networks (WSNA), San Diego, Sept. 2003

[Do05]B. Donovan, D. J. McLaughlin, J. Kurose, and V. Chandrasekar, 2005: "Principles and Design Considerations for Short-Range Energy Balanced Radar Networks," Proceedings of IGARSS05, Seoul, July, 2005

[Du05]P. Dutta, M. Grimmer, A. Arora, S. Bibyk, and D. Culler, "Design of a Wireless Sensor Network Platform for Detecting Rare, Random, and Ephemereal Events," Special Track on Platform Tools and Design Methods for Network Embedded Sensors (SPOTS), Los Angeles, April 2005

[Es99] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar, "Next Century Challenges: Scalable Coordination in Sensor Networks," in Proceedings of the 5th Int. Conf. on Mobile Computing and Network (MobiCOM '99),Seattle, Washington, Aug. 1999

[Es02] D. Estrin, D. Culler, K. Pister, and G. Sukhatme, "Connecting the Physical World with Pervasive Networks," IEEE Pervasive Computing, Vol. 1, No. 1, pp. 59--69, 2002

[Fa03] S. Fahmy, M. Kwon, "Characterizing Overlay Multicast Networks," in IEEE Intl. Conf. on Network Protocols, pp. 61-70, Nov. 2003

[Fl04] S. Floyd, "HighSpeed TCP for Large Congestion Windows," RFC 3649, Dec. 2004

[Fl99] S. Floyd, and K. Fall, "Promoting the use of End-to-End Congestion Control in the Internet," IEEE/ACM Trans. on Networking, pp. 458-472, August 1999

[Ga05] A. Ganjam, H. Zhang, "Internet Multicast Video Delivery," in Proc. of the IEEE, Vol. 93, Issue 1, pp. 159-170, Jan. 2005

[Ga03]D. Ganesan, S. Ratnasamy, H. Wang, and D. Estrin, "Coping with Irregular Spatio-temporal Sampling in Sensor Networks," 2nd Workshop on Hot Topics in Networks (HotNets-II) 2003, Cambridge (MA) USA, Nov. 2003

[Gh02] S. Ghiasi, A. Srivastava, Z. Yang, and M. Sarrafzadeh, "Optimal Energy Aware Clustering in Sensor Networks," Special Issue: Special Section on sensor network technology and sensor data management, Vol. 2, Issue 7, pp. 258-269, July 2002

[Gi03] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan, "IrisNet: An Architecture for a Worldwide Sensor Web," in Trans. of Pervasive Computing, Vol 2, Issue 4, pp. 22-33, Oc-Dec 2003

[Gu05]E. Gurses, G. B. Akar, N. Akar, "A simple and Effective Mechanism for Stored Video Streaming with TCP Transport and Server-side Adaptive Frame Discard," in Computer Networks Elsevier, Vol. 48, Issue 4, pp. 489-501, Jan. 2005

[Ha03]M. Handley, S. Floyd, J. Padhye, and J. Widmer, "TCP Friendly Rate Control (TFRC): Protocol Specification," IETF RFC 3448, Jan. 2003

[He03] T. He, J. A. Stankovic, C. Lu, and T. Abdelzaher, "Speed: A stateless Protocol for Real-time Communication in Sensor Networks," in Proc. of Intl. Conf. on Distributed Computing Systems (ICDCS 2003), Providence, RI, May 2003

[Hu04] B. Hull, K. Jamieson, and H, Balakrishnan, "Mitigating Congestion in Wireless Sensor Networks ," ACM SenSys 2004, Baltimore, MD, Nov., 2004

[He99] W. Heinzelman, J. Kulik, and H. Balakrishnan, "Adaptive Protocols for Information Dissemination in Wireless Sensor Networks," in Proc. Fifth ACM/IEEE Int. Conf. on Mobile Computing and Networking, pp. 174–185, 1999

[He00] W. Heinzelman, A. Chandrakasan, and H. Balakrishnan, "Energy-efficient communication protocol for wireless microsensor networks," In Proc. of 33rd Annual Hawaii Intl. Conf. on System Sciences, 2000

[In00] C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks," In Proc. 6th Int. Conf. on Mobile Computing and Networks (MobiCOM 2000), Boston, MA, August 2000,

[Ja04] A. Jain, and E.Y. Chang, "Adaptive Sampling for Sensor Networks," Proc. of the 1st Intl. workshop on Data management for sensor networks, Aug. 2004

[Ji00] W. Jiang, and H. Schulzrinne, "Modeling of Packet Loss and Delay and their Effect on Real-time Multimedia Service Quality," in Proc. 10th Intl. Workshop Network and Operations System Support for Digital Audio and Video, June 2000

[Ji04] C. Jin, D. X. Wei, and S. H. Low, "FAST TCP: Motivation, Architecture, Algorithms, Performance," Proc. of IEEE INFOCOM, March 2004

[Jo06] D. Joseph, J. Kannan, A. Kubota, K. Lakshminarayanan, I. Stoica, and K. Wehrle, "OCALA: An Architecture for Supporting Legacy Applications over Overlays," in Proc. of 3rd USENIX/ACM NSDI '06, May 2006

[Ka04] J. Kang, Y. Zhang, B. Nath, "Adaptive Resource Control Scheme to Alleviate Congestion in Sensor Networks," 1st Workshop on Broadband Advanced Sensor Networks, San-Jose, Oct. 2004

[Ka01] A. Kassler, A. Neubeck, and P. Schulthess, "Classification and Evaluation of Filters for Wavelet Coded Videostreams," Signal Processing: Image Communication, 16(8):795-807, Elsevier May, 2001

[Ke02] Urvoy-Keller, and G., Biersack, E.W. "A Congestion Control Model for Multicast Overlay Networks and its Performance," in Fourth Intl. Workshop on Networked Group Communication, NGC 2002, Oct. 2002.

[Ke00] R. Keller, S. Choi, M. Dasen, D. Decasper, G. Fankhauser, and B. Plattner, "An Active Router Architecture for Multicast Video Distribution," Proc. IEEE INFOCOM, Mar. 2000

[Ke03] T. Kelly, "Scalable TCP: Improving Performance in High-speed Wide Area Networks," in Proc. of ACM SIGCOMM, Volume 33, Issue 2, pg. 83-91, April 2003

[Ko03]D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat, "Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh," Proc. of *SOSP'03,* Bolton Landing, New York, Oct., 2003

[Ku06] J. Kurose, E. Lyons, D. McLaughlin, D. Pepyne, B. Philips, D. Westbrook, and M. Zink, 2006: "An End User Responsive Sensor Network Architecture for Hazardous Weather Detection, Prediction, and Response", Asian Internet Conference (AINTEC) 2006, Pathumthani, Thailand, Nov. 2006

[Le06] P. Lee, T. Banka, A. P. Jayasumana, and V. Chandrasekar, "Content based Packet Marking for Application Aware Processing in Overlay Networks," Proc. of IEEE Conf. on Local Computer Networks, (LCN 2006), Tampa FL, Nov. 2006

[Li02] D. Li, K. Wong, Y.H. Hu, and A. Sayeed, "Detection, Classification and Tracking of Targets in Distributed Sensor Networks," IEEE Signal Processing Magazine, Volume: 19 Issue: 2, Mar 2002

[Li04] Y. Liu, Y. Gu, H. Zhang, W. Gong, D. Towsley, "Application Level Relay for High-Bandwidth Data Transport," Proc of 1st Workshop on Networks for Grid Applications (GridNets) , Oct. 2004

[Li99] M. Lin, K. Marzullo, and S. Masini, "Gossip versus Deterministic Flooding: Low Message Overhead and High Reliability for Broadcasting on Small Networks," UCSD Technical Report TR CS99-0637, 1999

[Ma99] I. Mahadevan, and K. M. Sivalingam, "Quality of Service architectures for wireless networks: IntServ and DiffServ models," in Proc. of 4th Intl. Symp. on Parallel Architectures, Algorithms, and Networks, pp. 420-425, June 1999

[Ma02] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson, "Wireless Sensor Networks for Habitat Monitoring," in Proc. ACM Int. Workshop on Wireless Sensor Networks and Applications, Sept. 2002

[Ma03]A. D. Marbini, and L. E. Sacks, "Adaptive Sampling Mechanisms in Sensor Networks," London Communications Symposium, London, UK, 2003

[Ma05] A. Maroo, "Data Quality Control, Modeling and Performance Analysis of TRABOL: A Streaming Protocol for High-bandwidth Radar Data," M.S Thesis, Colorado State University, Spring 2005

[Ma06]H. Madhyastha, A. Venkataramani, A. Krishnamurthy, and T. Anderson, "Oasis: An Overlay-Aware Network Stack.," Proc. of ACM SIGOPS Operating Systems Review Vol. 40, Issue 1, pp41-48, Jan. 2006

[Mc96] S. McCanne, V. Jacobson, and M. Vetterli, "Receiver-driven Layered Multicast," in Proc. on Applications, technologies, architectures, and protocols for computer communications, pp.117-130, August 28-30, 1996

[Mc05] D.J. McLaughlin, V. Chandrasekar, K. Droegemeier, S. Frasier, J. Kurose, F. Junyent, B. Philips, S. Cruz-Pol, and J. Colom, "Distributed Collaborative Adaptive Sensing (DCAS) for Improved Detection, Understanding, and Prediction of Atmospheric Hazards," Proc. 21st Int. Conf. on Interactive Information Processing Systems (IIPS) for Meteorology, Oceanography, and Hydrology, American Meteorological Society (AMS), 11.3, Jan. 2005

[Ni98] K. Nicolas, S. F. Blake, and D. Black, "Definition of the Differentiated Service Field (DS Field) in the IPv4 and IPv6 Headers", RFC2474, Dec. 1998

[Ni02] N. Nikaein, and C. Bonnet, "A Glance at Quality of Services in Mobile Ad-Hoc Networks," in Proc. of DNAC 2002: 16th Conf. of New Architectures for Communications, Paris 2002

[Or98] K. Orr, "Data Quality and Systems Theory," in Proc. of ACM Communication, Vol.48, Issue 10, pp. 75-81, 1998

[Pa98] J. Padhye, J. Kurose, D. Towsley, and R. Koodli, "A Model Based TCP-Friendly Rate Control Protocol," UMass-CMPSCI Technical Report TR 98-04, Oct. 1998

[Pa06] O. Papaemmanouil, O., Y. Ahmad, U. Cetintemel, and J. Jannotti, "Application-aware Overlay Networks for Data Dissemination," in Proc. of the Intl. Workshop on Semantics enabled Networks and Services (ICDE SeNS 2006), Atlanta, April 2006

[Pe02] L. Peterson, T. Anderson, D. Culler, and T. Roscoe, "A Blueprint for Introducing Disruptive Technology into the Internet," 1st ACM Workshop on Hot Topics in Networks, HotNets-I, Oct. 2002

[Rapp] T. S. Rappaport, "Wireless Communications Principles & Practice," Prentice Hall

[Re99] R. Rejaie, M. Handley, and D. Estrin, "RAP: An End-to-end Rate based Congestion Control Mechanism for Real-time Streams in Internet," Proc. of IEEE INFOCOM, pp. 1337-1345, 1999

[Sa99] S. Savage, T. Anderson, A. Aggarwal, D. Becker, N. Cardwell, A. Collins, E. Hoffman, J. Snell, G. Voelker, and J. Zahorjan, "Detour: A Case for Informed Internet Routing and Transport," IEEE Micro Vol. 19 , Issue1, pp. 50–59, Jan. 1999

[Sa03] Y. Sankarasubramaniam, O. B. Akan, I. F. Akyildiz, "ESRT: Event-to-Sink Reliable Transport in Wireless Sensor Networks," in Proc. ACM MOBIHOC 2003, pp. 177-188, Annapolis, Maryland, June, 2003

[Sc99] B. Schwartz, A. Jackson, T. Strayer, W. Zhou, R. Rockwell, and C. Partridge, "Smart packets for active networks," in Proc. of the IEEE 2nd Conference on Open Architectures and Network Programming (OPENARCH'99), March 1999

[Sh05] S. Shionda, and K. Mase, "Performance comparison between IntServ-based and DiffServ-based networks," in Proc. of IEEE Global Telecomm. Conf., Globecom 2005, Vol 1, Nov 2005

[Sh04] R. N. Shorten, D.J.Leith, "H-TCP, TCP for high-speed and long-distance networks," in Proc. PFLDnet, Argonne, IL, Feb. 2004

[Si04a]G. Simon, M. Maroti, A. Ledeczi, G. Balogh, B. Kusy, A. Nadas, G. Pap, J. Sallai, and K. Frampton, "Sensor Network based Counter Sniper System," Proc. of 2nd ACM Conf. on Embedded Networked Sensor Systems (SenSys 2004), Baltimore, MD, Nov. 2004

[Si04b]F. Sivrikaya, and B. Yener, "Time Synchronization in Sensor Networks: A Survey," Proc. of IEEE Networks, Vol. 18, Issue 4, pp. 45-50, July-Aug. 2004

[So00] K. Sohrabi, J. Gao, V. Ailawadhi, and G. Pottie, "Protocols for Self-Organization of a Wireless Sensor Networks," IEEE Personal Communications, Vol. 7, 16-27, Oct. 2000

[Su04] L. Subramanian, I. Stoica, H. Balakrishnan, and R. Katz, "OverQoS: An Overlay Based Architecture for Enhancing Internet QoS," Proc. 1st Symposium on Networked Systems Design and Implementation (NSDI), San Francisco, CA, Mar. 2004

[Ta04] D. Talbot, "Seamless Surveillance," in Technology Review, Feb. 2004

[Te96] D. L. Tennenhouse, and D. Wetherall, "Towards An Active Network Architecture," Computer Communication Review Vol. 26, Issue 2, 1996

[Te97] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden, "A Survey of Active Network Research," IEEE Communications Magazine, Vol. 35, Issue 1, pp. :80–86, Jan. 1997

[Ti02] S. Tilak, N. B Abu-Ghazaleh, and W. Heinzelman, " Infrastructure Tradeoffs for Sensor Networks," In Proc. WSNA 2002, Atlanta, GA, Sept. 2002

[To02] J. Touch, Y. Wang, L. Eggert, "Virtual Internets," ISI Technical Report ISI-TR-2002-558, July, 2002

[Tr06] A. Trimmer, T. Banka, P. Lee, A.P. Jayasumana and V. Chandrasekar, "Performance of High-Bandwidth TRABOL Protocol for Radar Data Streaming," Proc. 2006 IEEE Region 5 Technical, Professional and Student (TPS) Conference, San Antonio TX, April 2006

[Wa02] C.Y. Wan, A. T. Campbell, and L. Krishnamurthy, "PSFQ: A Reliable Transport Protocol for Wireless Sensor Networks," in Proc. ACM Int. Workshop on Sensor Networks and Architectures, Atlanta, Sept. 2002

[Wa03] C. Y. Wan, S. B. Eisenman, and A. T. Campbell, "CODA: Congestion detection and avoidance in sensor networks," in Proc. ACM SenSys 2003, pages 266–279, Los Angeles, Nov. 5-7 2003

[Wa05] C. Wang, K. Sohraby, and B. Li, "SenTCP: A hop-by-hop congestion control protocol for wireless sensor networks," in Proc. of IEEE INFOCOM 2005 (Poster Paper), Miami, FL, Mar. 2005

[We98] D. Wetherall, U. Legedza, and J. Guttag, "Introducing New Internet Services: Why and How," in IEEE Network Magazine July/August 1998

[Xu04a]L. Xu, K. Harfoush, and I. Rhee, "Binary Increase Congestion Control for Fast Long-Distance Networks," in Proc. of IEEE INFOCOM 2004, Mar. 2004

[Xu04b]N. Xu, S. Rangwala, K. K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin, "A wireless sensor network for structural monitoring," in Proc. of the 2nd Intl. Conf. on Embedded Networked Sensor systems (SenSys), 2004

[Ye04] W. Ye, J. Heidemann, and D. Estrin, "Medium Access Control with Co-ordinated Adaptive Sleeping for Wireless Sensor Networks," IEEE/ACM Trans. on Networking, Vol. 12, pp. 493-506, June 2004

[Yu04]L. Yuan, C. Gui, C. Chuah, and P. Mohapatra, "Applications and Design of Heterogeneous and/or Broadband Sensor Networks," in Intl. Conf. on Broadband Networks, BROADNETS, Oct. 2004

[Zh99] Z.-L Zhang, S. Nelakuditi, R. Aggarwal, and R. Tsang, "Efficient Selective Frame Discard Algorithms for Stored Video Delivery across Resource Constrained Networks," in Proc. of IEEE INFOCOM, Mar. 1999

[Zh04a]Li. Zhi, P. Mohapatra, "QRON: QoS-aware routing in overlay networks," IEEE Jour. on Selected Areas in Communication, Vol. 22, Issue 1, Jan. 2004

[Zh04b]J. Zhao, and R. Govindan, "Understanding Packet Delivery Performance in Dense Wireless Sensor Networks," Proc. of 1st Intl. Conf. on Embedded Networked Sensor Systems (SenSys 2003), Los Angeles, Nov. 2003

[Zu04] M. Zuniga, and B. Krishnamachari, "Analyzing the Transitional Region in Low Power Wireless Links," 1st IEEE International Conf. on Sensor and Ad hoc Comm. and Networks (SECON), Santa Clara, CA, Oct. 2004

[Aka] Akamai Content Distribution Services www.akamai.com

[Cas] "CASA: Collaborative Adaptive Sensing of Atmosphere," Website: http://www.casa.umass.edu

[Chp] Chipcon CC1000 Data Sheet, http://www.chipcon.com

[Chi] "Virtual CSU-CHILL National Radar Facility," Website: http://chill.colostate.edu/

[Exp] Exploratory Project: Heterogeneous Sensor Networks, Intel, http://www.intel.com/research/exploratory/heterogeneous.htm

[Gen] Global Environment for Network Innovation (GENI) http://www.geni.net/

[Lee] Application-Aware Overlay Networking for Distributed Adaptive Sensing Systems (*PhD thesis- tentative)*

[Pla] Planetlab: www.planet-lab.org

[Pow] Powercast: Wireless Power Platform http://www.powercastco.com/

APPENDIX A

Computing Tardiness of Data in Sensor Networks

I. Simulation Program for Generating Tardiness Profile

```
/* Simulation Program for generating Tardiness Profile in a Grid Network

   Tardiness Profile of network grid where sink is at the center of the grid and all
   other nodes transmit data to the sink node in single hop Losses are considered because
   of wireless losses, Network is assumed to be without collision and network congestion
   realistic wireless loss model is used between source and sink node is considered
   based on Prof. Krishnamachari @ USC work for MICA2 platform

   Program allows user to specify network topology, sampling rate, read rate
*/

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "tardiness.h"
#include <sys/time.h>
#define MAX_SAMPLES   400000                  // Maximum Samples generatated by a Node
#define MAX_ACCESS_ATTEMPTS MAX_SAMPLES*6     // Number of time buffer is read
#define MAX_X 5000                            // Number of data points at sensor node
#define MAX_TIME 10000                        // MAX_X corresponds to MAX_TIME duration at
                                              // sensor node
// 20*MAX_SAMPLES 20.0 is the max sample time considered
//in experiments, but this is used only while collecting data
#define SLOPE 10.0            // Data Slope at Sensor node
#define INFINITE 999999999.0  // To indicate arrival time of lost or reordered packet

int CONSTANT=5;
struct timeval tp;
int drop_count=0;             // Track number of samples dropped
int reorder_count=0;          // Track number of samples reordered
double max_generate_time=0.0; // Time for which we have data values, used for error in
                              // data value due to tardiness
double periodic_increment =0.001;
struct data_info_type data_samples_list[MAX_SAMPLES],
error_computation_list[MAX_SAMPLES]; // Error in the end results
struct tardiness_type *tardiness_info;
double sample_period;         // Determine sample generation rate
double read_period;           // Periodic Read frequency for input buffer
double delay_lambda;          // Mean Delay
double delay_sum = 0.0;       // Track sum of delay suffered by delivered samples
float loss_probability;
int seed1, seed2,seed3, seed4;
FILE *fp, *in_fp;
double constant_delay;
double mean_delay;
float *tardiness_list;
float tardiness_val=0.0;
```

```
double periodic(double periodic_incr);
double poisson(double lambda);
double exponential();
double uniform();
void data_generate_events();
void network_delay_events();
void buffer_access_events();
void new_tardiness_computation();
void initialize();

struct node
        {
        float x;
        float y;
        float *prr;
        };

// Used by sorting algorithm
int  lo2hi(const void *vp0, const void *vp1)
{
        int val;
        const struct data_info_type *ip0 = (const struct data_info_type *)vp0;
        const struct data_info_type *ip1 = (const struct data_info_type *)vp1;
        if (ip0->arrival_time < ip1->arrival_time)          val = -1;
        else if (ip0->arrival_time==ip1->arrival_time) val =  0;
        else if (ip0->arrival_time >ip1->arrival_time) val =  1;
        return(val);
}

// Main Function for Tardiness Profile Simulation
int main(int argc, char *argv[])
{
        char filename[100];
        struct node *network_grid;
        int i,j, total_nodes=0;
        int sink_node;
        if (argc!=7)
                {
                printf("<tardiness <sample_period> <read_period> <Mean-Delay>
                        <loss_probability> <input_file> <output_filename>\n");
                exit (1);
                }
        sample_period    = atof(argv[1]);
        read_period      = atof(argv[2]);
        mean_delay       = atof(argv[3]);
        loss_probability = atof(argv[4]);

        strcpy(filename, argv[6]);
        strcat(filename,"_s_");
        strcat(filename,argv[1]);
        strcat(filename,"_r_");
        strcat(filename,argv[2]);
        strcat(filename,"_delay_");
        strcat(filename,argv[3]);
        strcat(filename,"_p_");
        strcat(filename,argv[4]);
        fp = fopen(filename,"w");
        if (!fp)
                exit(1);
        in_fp = fopen(argv[5],"r");

        if (!in_fp)
                {
                printf("Error opening input file\n");
                exit(1);
                }

        // Number of nodes present in the topology
        fscanf(in_fp,"%d\n", &total_nodes);
        network_grid = (struct node *)malloc(sizeof(struct node)*total_nodes);
        if (!network_grid)
```

```
        {
        printf("error allocation network grid\n");
        exit(1);
        }
for (i=0;i<total_nodes;i++)
        {
        network_grid[i].prr = (float *)malloc(sizeof(float)*total_nodes);
        if (!network_grid[i].prr)
                {
                printf("Error allocating PRR\n");
                exit(1);
                }
        }

tardiness_info = (struct tardiness_type *)malloc(sizeof(struct
                  tardiness_type)*MAX_ACCESS_ATTEMPTS);
if (tardiness_info==NULL)
        {
        printf("Error in Allocation\n");
        exit(1);
        }
tardiness_list = (float *)malloc(sizeof(float)*total_nodes);
if (!tardiness_list)
        {
        printf("Error allocating tardiness list\n");
        exit(1);
        }
 //Read input file for the coordinates
for (i=0;i<total_nodes;i++)
        {
        int temp;
        fscanf(in_fp,"%d %f %f\n",&temp, &network_grid[i].x, &network_grid[i].y);
        }

 // Read Packet Reception Probability
for (i=0;i<total_nodes;i++)
        {
        for (j=0;j<total_nodes;j++)
                {
                fscanf(in_fp,"%f ", &network_grid[i].prr[j]);
                }
        fscanf(in_fp,"\n");
        }

// Let Sink node is the total_nodes/2 then compute
// tardiness of data from each node to the sink node
// This will help us get the Tardines profile
sink_node = total_nodes/2;
for (j=0;j<total_nodes;j++)
        {
        float distance;
        // Determine the Network loss Probability as 1- Packet Reception_Rate
        loss_probability = 1.0 - network_grid[j].prr[sink_node];
        distance = sqrt((network_grid[j].x-
                        network_grid[sink_node].x)*(network_grid[j].x-
                        network_grid[sink_node].x)+(network_grid[j].y-
                        network_grid[sink_node].y)*(network_grid[j].y-
                        network_grid[sink_node].y));
        memset(tardiness_info,0,sizeof(struct
               tardiness_type)*MAX_ACCESS_ATTEMPTS);
        memset(data_samples_list, 0, sizeof(struct data_info_type)*MAX_SAMPLES);
        initialize();
        gettimeofday(&tp, NULL);
        seed1 = tp.tv_usec;             // For generating delay
        for (i=0;i<50;i++);             // Just spend some time in the loop for the
                                        // new seed
        gettimeofday(&tp,NULL);
        seed2 = tp.tv_usec;             // For making loss decisions
        for (i=0;i<100;i++);             // Just spend some time...
        gettimeofday(&tp,NULL);
        seed3 = tp.tv_usec;             // For start sampling time
        for (i=0;i<75;i++);             // spend some time....
```

```
                gettimeofday(&tp,NULL);
                seed4 = tp.tv_usec;             // For start buffer access


                data_generate_events();
                network_delay_events();
                buffer_access_events();
                new_tardiness_computation();
                tardiness_list[j]=tardiness_val; // Tardiness from jth node the sink node
                }
}

// Used for Exponential Delay Distribution
double uni(int *seed)
{
        int a,b;
        a= (int) (*seed/16384);
        b= (int) (*seed %16384);
        *seed = (((13205*a + 74505*b) % 16384)*16384+13205*b) % 268435456;
        return((double) (*seed)/268435456);
}

double poisson(double lambda)
{
        double deltat, ln;
        ln = log(1-uni(&seed1));
        deltat = (-ln)/lambda;
        return(deltat);
}

double periodic(double periodic_incr)
{
        return(periodic_incr);
}

// Initialization Routine
void initialize()
{
        drop_count     = 0.0;
        reorder_count = 0.0;
        delay_sum      = 0.0;
        max_generate_time = 0.0;
}
void data_generate_events()
{
        double start_time= 0.0;
        int i;
        int data_seed     = 136776;
        int flag          = 0;
        float value       = 0.0;
        float data_value[MAX_X];
        int data_index,resolution;
        int k;
        data_samples_list[0].generate_time  = uni(&seed3);

        // Generate Data Value and then sample it
        for (i=0;i<250;i++)
                {
                data_value[i]=SLOPE*(float)i+ CONSTANT;
                }
        for (i=250;i<500;i++)
                {
                data_value[i]=data_value[500-i-1];
                }
        k=0;
        for (i=500;i<750;i++)
                {
                k++;
                data_value[i] = data_value[500-k];
                }
        for (i=750;i<1000;i++)
                {
```

```
                data_value[i]= data_value[1000-i-1];
                }
        resolution = MAX_TIME/MAX_X; // One entry of X corresponds to 'resolution' units
                                     // of time
        for (i=1;i<MAX_SAMPLES;i++)
                {
                data_samples_list[i].generate_time = data_samples_list[i-1].generate_time
                                                     + sample_period;
                data_index = data_samples_list[i].generate_time/resolution;
                if (data_index<1000)
                        {
                        data_samples_list[i].data_val = data_value[data_index];
                        max_generate_time = data_samples_list[i].generate_time;
                        }
                error_computation_list[i].generate_time  =
                                                    data_samples_list[i].generate_time;
                error_computation_list[i].data_val       = data_samples_list[i].data_val;
                error_computation_list[i].sample_number  = i;
                }
}

// Use Exponential Distribution for the packet delay
void network_delay_events()
{
        int   i,current_largest;
        float rand_value;
        float sum_ipg=0.0;
        float delay_sum=0.0;
        int   drop_count;
        int   reorder_count;
        for (i=0;i<MAX_SAMPLES;i++)
                {
                /*
                 Here code to insert loss would be implemented
                 Given is loss probability p
                */
                rand_value = poisson(1.0/mean_delay);
                data_samples_list[i].arrival_time =
                                data_samples_list[i].generate_time+rand_value;
                rand_value   = uni(&seed2);    // random variable for loss For loss
                delay_sum += data_samples_list[i].arrival_time-
                             data_samples_list[i].generate_time;
                if (rand_value<=loss_probability)
                        {
                        /* Mark the packet as lost, this can be done by setting the arrival
                           time of the packet after the arrival to large value which is
                            not feasible e.g. max access_time, Hopefully following
                           statement would make is useless.
                        */
                        delay_sum -= data_samples_list[i].arrival_time-
                                     data_samples_list[i].generate_time;
                        data_samples_list[i].arrival_time    = INFINITE;
                        drop_count++;
                        }
                data_samples_list[i].sample_number = i;
                }
        qsort(data_samples_list, MAX_SAMPLES, sizeof(data_samples_list[0]), lo2hi);
        reorder_count=0;
        current_largest=data_samples_list[0].sample_number;
        for (i=0;i<MAX_SAMPLES;i++)
                {
                if (data_samples_list[i].arrival_time==INFINITE)
                        {
                        break; // All lost samples are treated to arrive after long
                               //  time..don't consider them for measuring reordering.
                        }
                if (data_samples_list[i].sample_number<current_largest)
                        {
                        reorder_count++;
                        /* The sample that is treated as reordered should not be included
                           in the tardiness computation, it is treated as lost
                            so mark its arrival time as the lost packet so that it is not
```

```
                        used for computation, and also adjust the total delay sum.
                     */
                     delay_sum-=data_samples_list[i].arrival_time-
                                data_samples_list[i].generate_time;
                     data_samples_list[i].arrival_time = INFINITE;
                     }
              else
                     {
                     current_largest = data_samples_list[i].sample_number;
                     }
              }
       if (i!=0)  // i indicates total samples that have arrived, then compute the
                  // probability of reordering based on that.
              {
              printf("Reorder percentage %f\n", (float)reorder_count/(float)i);
              printf("Packet that arrive without reordering/network loss = %f\n", 1.0-
                     (float)(reorder_count+drop_count)/(float)MAX_SAMPLES);
              }

       // Sort Data again that such that all packets that are marked lost or reordered
       //  are at the end.
       qsort(data_samples_list, MAX_SAMPLES, sizeof(data_samples_list[0]), lo2hi);
       // Test Code to see the delay of packets that arrive without any reordering and
       //  loss
       for (i=0;i<MAX_SAMPLES-drop_count-reorder_count;i++)
              {
              static float temp_delay_sum=0;
              temp_delay_sum+=data_samples_list[i].arrival_time-
                              data_samples_list[i].generate_time;
              if (i==MAX_SAMPLES-drop_count-reorder_count-1)
                     printf("Average Delay of Arrived packet is %f\n",
                            temp_delay_sum/(MAX_SAMPLES-reorder_count-drop_count));
              if (i>0)
                     {
                     sum_ipg+=data_samples_list[i].arrival_time-data_samples_list[i-
                              1].arrival_time;
                     }
              }
}

// Randomly access receiver buffer
void buffer_access_events()
{
       int i;
       tardiness_info[0].access_time=data_samples_list[0].arrival_time+uni(&seed4); //
Start randomly  after 1st sample arrives in the
       for (i=1;i<MAX_ACCESS_ATTEMPTS;i++)
              {
              tardiness_info[i].access_time =tardiness_info[i-1].access_time +
read_period;
              }
}


// Every time buffer is accessed, tardiness of the data is computed.
void new_tardiness_computation()
{
       /*
          Determine first two arrival times
          Determine first access time
       */
       double current_arrival_time  = data_samples_list[0].arrival_time;
       double current_generate_time = data_samples_list[0].generate_time;
       int current_sample           = data_samples_list[0].sample_number;
       double current_data_val      = data_samples_list[0].data_val;
       double next_arrival_time     = data_samples_list[1].arrival_time;
       double next_generate_time    = data_samples_list[1].generate_time;
       int next_sample              = data_samples_list[1].sample_number;
       double next_data_val         = data_samples_list[1].data_val;
       int sample_count=0;    // Number of distinct in-order samples arrives at the
                              // destination
       // For error computation
```

```
int source_index_offset = 0;
double source_data_val  = 0.0,sum_error_square=0.0;
double access_time      = tardiness_info[0].access_time;
int data_index, access_index;

/*
 Determine all the arrivals in the given access time slot,
 then determine the most recently generated arrival, compute tardiness for that if
 there are no arrivals in the given access slot then use the same data used in the
 previous slot.
*/

access_index =0 ; //
data_index   =1 ; // while(access_time<=data_samples_list[MAX_SAMPLES-
                  // 1].arrival_time)

while (data_samples_list[data_index].arrival_time!=INFINITE)
        {
        // Keep checking buffer until the last arrival of the packet
        if (data_index>=MAX_SAMPLES || access_index>=MAX_ACCESS_ATTEMPTS) break;
        access_time = tardiness_info[access_index].access_time;
        if (access_time<next_arrival_time)
                {
                /* At this time we are in a position to compute tardiness of the
                   data current_sample is the data that should be used for the
                   computation as it is most recently generated data and thus
                   tardiness is computed for that
                */
                tardiness_info[access_index].tardiness     = access_time -
                 current_generate_time; // Store age of the data
                tardiness_info[access_index].sample_number = current_sample;
                // Note down the sample used for the computation
                tardiness_info[access_index].generate_time = current_generate_time;
                tardiness_info[access_index].arrival_time  = current_arrival_time;
                tardiness_info[access_index].data_val      = current_data_val;
                if (access_time<max_generate_time)
                        {
                        /*
                         Computer Error in the data read at the access time
                         1. Determine value of data at source node at access_time?
                         2. Determine data read from buffer and compute difference
                         */
                        source_index_offset   = (access_time-
                                                current_generate_time)/sample_period;
                        if (data_index+source_index_offset<MAX_SAMPLES)
                                {
                                source_data_val       =
                                                error_computation_list[data_index+
                                                source_index_offset].data_val;
                                sum_error_square += (source_data_val-
                                 current_data_val)*(source_data_val-
                                 current_data_val);
                                }
                        }
                access_index++;          // Consider next access time
                }
        else
                {
                // Check if the next is the most recently generated data then make
                // it current
                if (current_sample<next_sample)
                        {
                        current_arrival_time = next_arrival_time;
                        current_generate_time = next_generate_time;
                        current_sample = next_sample;
                        current_data_val = next_data_val;
                        sample_count++;
                        }
                next_arrival_time   = data_samples_list[data_index+1].arrival_time;
                next_generate_time  =
                                      data_samples_list[data_index+1].generate_time;
                next_sample         =
```

```
                                        data_samples_list[data_index+1].sample_number;
                    next_data_val       = data_samples_list[data_index+1].data_val;
                    data_index++;
                    }
            }
        // Computer tardiness in terms of buffer access count K at each read buffer
        {
            int max_access = access_index-1;
            int i;
            double tardiness_sum, tardiness_square_sum;
            current_sample      = tardiness_info[0].sample_number;
            tardiness_info[0].k = 1;
            for (i=1;i<max_access;i++)
                    {
                    if (tardiness_info[i].sample_number==current_sample)
                            {
                            tardiness_info[i].k = tardiness_info[i-1].k+1;
                            }
                    else
                            {
                            current_sample = tardiness_info[i].sample_number;
                            tardiness_info[i].k =1;
                            }
                    }
            tardiness_sum=0.0;
            tardiness_square_sum=0.0;
            for (i=0;i<max_access;i++)
                    {
                    fprintf(fp,"%d %f %f %f %f %d %f
                    %f\n",tardiness_info[i].sample_number,
                    tardiness_info[i].generate_time,
                     tardiness_info[i].arrival_time,tardiness_info[i].access_time,
                      tardiness_info[i].tardiness,
                     tardiness_info[i].k,tardiness_info[i].arrival_time-
                      tardiness_info[i].generate_time,
                      tardiness_info[i].tardiness*tardiness_info[i].tardiness
                     );
                     if (tardiness_info[i].sample_number<MAX_SAMPLES-10)
                            {
                            /* Many times last few samples are not received that leads
                               to significant increase in tardiness thus does not
                               include tardiness of those samples
                            tardiness_sum+=tardiness_info[i].tardiness;
                            tardiness_square_sum+=
                            (tardiness_info[i].tardiness*tardiness_info[i].tardiness);
                            }
                    }
            printf("Total Samples arriving in order: %d\n", MAX_SAMPLES-drop_count-
                                                            reorder_count);
            printf("Average Delay: %f\n", delay_sum/(double)(MAX_SAMPLES-drop_count-
                                        reorder_count));
            printf("First Moment: Mean Tardiness: %f\n",
                                                tardiness_sum/(double)(max_access));
            printf("Second Moment: Tardiness:
                                        %f\n",tardiness_square_sum/(double)max_access);
            printf("Inorder no loss Sample count is %d\n",sample_count);
            tardiness_val = tardiness_sum/(double)max_access;
        }
}

// Compute Tardiness
void tardiness_computation()
{
        int index,i=0;
        double access_time;
        double current_arrival_time  = data_samples_list[0].arrival_time;
        double current_generate_time = data_samples_list[0].generate_time;
        int    current_sample        = data_samples_list[0].sample_number;
        double next_arrival_time     = data_samples_list[1].arrival_time;
        double next_generate_time    = data_samples_list[1].generate_time;
        int next_sample              = data_samples_list[1].sample_number;
```

```
        index=1;
        access_time =tardiness_info[i].access_time;
        tardiness_info[i].k = 1;
        while (access_time<=data_samples_list[MAX_SAMPLES-1].arrival_time)
                {
                if (index>=MAX_SAMPLES || i>=MAX_ACCESS_ATTEMPTS) break;
                access_time = tardiness_info[i].access_time;
                if (access_time<next_arrival_time)
                        {
                        tardiness_info[i].tardiness = access_time-current_generate_time;
                        tardiness_info[i].sample_number = current_sample;
                        if (i>=1 && tardiness_info[i-1].sample_number==current_sample)
                                {
                                tardiness_info[i].k = tardiness_info[i-1].k+1;
                                }
                        }
                else
                        {
                        while (access_time>next_arrival_time)
                                {
                                /*
                                In this we assume that of all the data that arrives in the
                                last read window slot data which is generated most recently
                                is used, so tardiness is computed only for that one sample.
                                */
                                if (next_sample>current_sample)
                                        {
                                        current_arrival_time  = next_arrival_time;
                                        current_generate_time = next_generate_time;
                                        current_sample        = next_sample;
                                        }
                                next_arrival_time  =
                                                data_samples_list[index+1].arrival_time;
                                next_generate_time =
                                                data_samples_list[index+1].generate_time;
                                next_sample        =
                                                data_samples_list[index+1].sample_number;
                                index++;
                                if (index>=MAX_SAMPLES)break;
                                }
                        if (next_sample>current_sample)
                                {
                                tardiness_info[i].k=1;
                                tardiness_info[i].sample_number = next_sample;
                                }
                        else if (i>=1 && tardiness_info[i-1].sample_number==current_sample)
                                {
                                tardiness_info[i].k = tardiness_info[i-1].k+1;
                                tardiness_info[i].sample_number = current_sample;
                                }

                        tardiness_info[i].tardiness = access_time-current_generate_time;

                        printf("%d %f %f %f
                                %d\n",current_sample,access_time,current_generate_time,
                                tardiness_info[i].tardiness,tardiness_info[i].k);
                        }
                i++;
                }
}
```

II. Computation of probability of in-order arrival at sink node

```
/*   Program to compute probability of in-order arrival of samples given exponential
delay distribution

    This program estimate Probability of in-order arrival. This is necessary for
estimating tardiness of data for applications that cannot tolerate out-of-order packets
Exponential Delay Distribution is considered for the data between source and a sink node
This information is used by the tardiness model to predict tardiness while considering
losses due to out-of-order arrival.
*/

#include <math.h>
#include <stdio.h>
#include <stdlib.h>

struct list_node
        {
        short int len;
        short int sum;
        };

double get_combination_sum(int, int, double, double);

int main(int argc, char *argv[])
{
        double s      = 0;      // Sampling Time
        double lambda = 0.0;    // For sample delay
        double p      = 0.0;
        double product= 0.0;
        double sum    = 0.0;
        double integral=0.0;
        double in_order_prob=0.0;
        int i,j,k,l;          // For delay index
        int i_max;
        int *input_list;
        if (argc!=5)
                {
                printf("Enter: tardiness <i> <s> <lambda> <p>\n");
                exit(1);
                }
        i_max  = atoi(argv[1]);
        s      = atof(argv[2]);
        lambda = atof(argv[3]);
        p = (double)atof(argv[4]);
        product       = 0.0;
        in_order_prob = 0.0;

        /*
        For Generating all Combinations of possible packet arrivals
        */
        input_list= (int *)malloc(sizeof(int)*i_max);
        for (i=0;i<i_max;i++)
                {
                input_list[i]=i+1;
                }
        // Consider first intergral in the range 0-S
        k=0;
        i=1;
        in_order_prob = (exp(-(k+1)*(i-1)*lambda*s)-exp(-(k+1)*i*lambda*s))/(k+1);
        //Consider Remaining Integral S-2S, 2S-3S, .....
        for (i=2;i<=i_max;i++)
                {
                combination_count=0;
                j=0;
                for (k=0;k<i;k++)
                        {
                        integral = (exp(-(k+1)*(i-1)*lambda*s)-exp(-
                                (k+1)*i*lambda*s))/(k+1);
                        product = pow(p,i-1-k)*pow(1-p, k);
```

```
                        sum=0.0;
                        /* It is assumed that combination list stores combinations in
                        increasing order of number of terms, e.g., {1}, {2}, {3}, {1,2},
                        {1,3}, {2,3}
                        Get Sum of elements in all combinations of size K
                         */
                        sum= get_combination_sum(i-1,k,lambda,s); // This call adds sum of
                                //all possible combinations of size K after multiplied with
                                //lambda*s and return it.
                        if (k==0)
                                {
                                sum=1.0;
                                }
                        product *=(sum*integral);
                        in_order_prob +=product;
                        }
                }
        printf("After i_maxt %d prob of IN ORDER arrival is %f\n", i_max, in_order_prob);
        printf("P(IN_ORDER/ARRIVE)*(1-P) = %f\n", in_order_prob*(1-p));
}

// From KNUTH book
// Generate all possible permutations of the arrival and loss of samples
double get_combination_sum (int n, int k, double lambda, double s)
{
        int i, j=1, *c, x;
        double temp_sum=0.0;
        double total_sum=0.0;
        double count=0;
        c = malloc( (k+3) * sizeof(int));
        if (!c)
                {
                printf("Memory Allocation Failure\n");
                exit(1);
                }
        if (k==0)
                return 0.0;
        if (n==k)
                {
                temp_sum= n*(n+1)/2;
                total_sum = exp(temp_sum*lambda*s);
                count ++;
                free(c);
                return total_sum;
                }
        for (i=1; i <= k; i++)
                c[i] = i;
        c[k+1]   = n+1;
        c[k+2]   = 0;
        j        = k;
        visit:
        temp_sum=0.0;
        for (i=k; i >= 1; i--)
                {
                temp_sum+=c[i];
                }
        count++;
        total_sum +=exp(temp_sum*lambda*s);
        if (j > 0)
                {
                x = j+1; goto incr;
                }
        if (c[1] + 1 < c[2])
                {
                c[1] += 1;
                goto visit;
                }
        j = 2;
        do_more:
        c[j-1] = j-1;
        x = c[j] + 1;
        if (x == c[j+1])
```

```
                {
                j++; goto do_more;
                }
        if (j > k)
                {
                free(c);
                return total_sum;
                }
        incr:
        c[j] = x;
        j--;
        goto visit;
}
```

III Computation of mean delay of packets/samples that arrives in-order

```
/* Program to estimate mean delay of the packets/samples that arrive in-order at the sink
node. This program is based on the computation performed for computing in-order
probability.

This program computes the average delay of all the samples that arrive in-order. This
estimate is used to estimate tardiness of data for applications that cannot tolerate out-
of-order late packets for computation. Exponential Delay Distribution is considered for
the samples transmitted from the source node
*/

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
struct list_node
        {
        short int len;
        short int sum;
        };

double get_combination_sum(int, int, double, double);

int main(int argc, char *argv[])
{
        double s        = 0;       // Sampling Time
        double lambda   = 0.0;     // For sample delay
        double p        = 0.0;     // Loss probability
        double product  = 0.0;
        double sum      = 0.0;
        double integral = 0.0;
        double expected_delay = 0.0;
        int i,j,k,l;               // For delay index
        int i_max;

        int *input_list;
        if (argc!=5)
                {
                printf("Enter: tardiness <i> <s> <lambda> <p>\n");
                exit(1);
                }
        i_max  = atoi(argv[1]);
        s      = atof(argv[2]);
        lambda = atof(argv[3]);
        p      = (double)atof(argv[4]);
        product= 0.0;

        //For Generating all Combinations of possible packet arrivals
        input_list= (int *)malloc(sizeof(int)*i_max);
        for (i=0;i<i_max;i++)
                {
                input_list[i]=i+1;
                }
        // Consider first intergral in the range 0-S
        k=0;
        i=1;
        expected_delay= ((exp(-(k+1)*lambda*(i-1)*s)*((i-1)*s+(1/((k+1)*lambda))))- (exp(-
                        (k+1)*lambda*i*s)*(i*s+(1/((k+1)*lambda)))))/(k+1);
        //Consider Remaining Integral S-2S, 2S-3S, .....
        for (i=2;i<=i_max;i++)
                {
                j=0;
                k=i-1;
                {
                        integral = ((exp(-(k+1)*lambda*(i-1)*s)*((i-
                                    1)*s+(1/((k+1)*lambda))))- (exp(-
                                   (k+1)*lambda*i*s)*(i*s+(1/((k+1)*lambda)))))/
                                   (double)(k+1);
                        product = pow(p,i-1-k)*pow(1-p, k);
                        sum=0.0;
                        /* It is assumed that combination list stores combinations in
                           increasing order of number of terms, e.g., {1}, {2}, {3}, {1,2},
```

```
                        {1,3}, {2,3}
                        Get Sum of elements in all combinations of size K
                    */
                    sum= get_combination_sum(i-1,k,lambda,s);
                    // This call adds sum of all possible combinations of size K after
                    // multiplied with lambda*s and return it.
                    if (k==0)
                            {
                            sum=1.0;
                            }
                    product *=(sum*integral);
                    expected_delay +=product;
                    printf("Expected_Delay is %f\n", expected_delay);
            }
            }
        printf("After i_maxt %d Expected Delay Is %f\n", i_max, expected_delay);
        printf("Expected_Delay*(1-P) = %f\n", expected_delay*(1-p));
}


/*
 This algorithm is based on the Art of programming by Knuth
 It is modified slightly for this program.
*/
double get_combination_sum (int n, int k, double lambda, double s)
{
        int i, j=1, *c, x;
        double temp_sum=0.0;
        double total_sum=0.0;
        double count=0;
        c = malloc( (k+3) * sizeof(int));
        if (!c)
                {
                printf("Memory Allocation Failure\n");
                exit(1);
                }
        if (k==0)
                return 0.0;
        if (n==k)
                {
                temp_sum= n*(n+1)/2;
                total_sum = exp(temp_sum*lambda*s);
                count ++;
                free(c);
                return total_sum;
                }
        for (i=1; i <= k; i++)
                c[i] = i;

        c[k+1] = n+1;
        c[k+2] = 0;
        j = k;
        visit:
        temp_sum=0.0;
        for (i=k; i >= 1; i--)
                {
                temp_sum+=c[i];
                }
        count++;
        total_sum +=exp(temp_sum*lambda*s);
        if (j > 0)
                {
                x = j+1; goto incr;
                }
        if (c[1] + 1 < c[2])
                {
                c[1] += 1;
                goto visit;
                }
        j = 2;
        do_more:
```

```
        c[j-1] = j-1;
        x = c[j] + 1;
        if (x == c[j+1])
                {
                j++; goto do_more;
                }
        if (j > k)
                {
                free(c);
                return total_sum;
                }
        incr:
        c[j] = x;
        j--;
        goto visit;
}
```

APPENDIX B

Estimating Probability of In-order Arrival

I. Probability of in-order arrival computation when delay distribution is exponential with mean λ, S is sampling interval at sensor node, and p is the network loss probability. D_{i+1} is the delay suffered by sample i+1.

$$p_{in-order} = \int_0^S \lambda e^{-\lambda c} dc + p\int_S^{2S} \lambda e^{-\lambda c} dc + (1-p)\int_S^{2S} p(D_{i+1} \geq c - S)\lambda e^{-\lambda c} dc +$$
$$p^2 \int_{2S}^{3S} \lambda e^{-\lambda c} dc + p(1-p)\int_{2S}^{3S} p(D_{i+1} \geq c - S)\lambda e^{-\lambda c} dc +$$
$$p(1-p)\int_{2S}^{3S} p(D_{i+1} \geq c - 2S)\lambda e^{-\lambda c} dc +$$
$$(1-p)^2 \int_{2S}^{3S} p(D_{i+1} \geq c - S)p(D_{i+1} \geq c - 2S)\lambda e^{-\lambda c} dc +$$
$$p^3 \int_{3S}^{4S} \lambda e^{-\lambda c} dc + ...$$

$$p_{in-order} = \int_0^S \lambda e^{-\lambda c} dc + p\int_S^{2S} \lambda e^{-\lambda c} dc + (1-p)\int_S^{2S} e^{-\lambda(c-S)}\lambda e^{-\lambda c} dc +$$
$$p^2 \int_{2S}^{3S} \lambda e^{-\lambda c} dc + p(1-p)\int_{2S}^{3S} e^{-\lambda(c-S)}\lambda e^{-\lambda c} dc +$$
$$p(1-p)\int_{2S}^{3S} e^{-\lambda(c-2S)}\lambda e^{-\lambda c} dc +$$
$$(1-p)^2 \int_{2S}^{3S} e^{-\lambda(c-S)}e^{-\lambda(c-2S)}\lambda e^{-\lambda c} dc +$$
$$p^3 \int_{3S}^{4S} \lambda e^{-\lambda c} dc + ...$$

II. Expected delay of packet arriving in-order at sink node, given that delay distribution is exponential with mean λ, S is sampling interval at sensor node, and p is the network loss probability. D_{i+1} is the delay suffered by sample i+1.

$$E[D]_{in\_order} = (\int_0^S c\lambda e^{-\lambda c}dc + p\int_S^{2S} c\lambda e^{-\lambda c}dc + (1-p)\int_S^{2S} cp(D_{i+1} \geq c-S)\lambda e^{-\lambda c}dc +$$
$$p^2\int_{2S}^{3S} c\lambda e^{-\lambda c}dc + p(1-p)\int_{2S}^{3S} cp(D_{i+1} \geq c-S)\lambda e^{-\lambda c}dc +$$
$$p(1-p)\int_{2S}^{3S} cp(D_{i+1} \geq c-2S)\lambda e^{-\lambda c}dc +$$
$$(1-p)^2\int_{2S}^{3S} cp(D_{i+1} \geq c-S)p(D_{i+1} \geq c-2S)\lambda e^{-\lambda c}dc +$$
$$p^3\int_{3S}^{4S} c\lambda e^{-\lambda c}dc + ...) / p_{in\_order}$$

OR

$$E[D]_{in\_order} = (\int_0^S c\lambda e^{-\lambda c}dc + p\int_S^{2S} c\lambda e^{-\lambda c}dc + (1-p)\int_S^{2S} ce^{-\lambda(c-S)}\lambda e^{-\lambda c}dc +$$
$$p^2\int_{2S}^{3S} c\lambda e^{-\lambda c}dc + p(1-p)\int_{2S}^{3S} ce^{-\lambda(c-S)}\lambda e^{-\lambda c}dc +$$
$$p(1-p)\int_{2S}^{3S} ce^{-\lambda(c-2S)}\lambda e^{-\lambda c}dc +$$
$$(1-p)^2\int_{2S}^{3S} ce^{-\lambda(c-S)}e^{-\lambda(c-2S)}\lambda e^{-\lambda c}dc +$$
$$p^3\int_{3S}^{4S} c\lambda e^{-\lambda c}dc + ...) / p_{in\_order}$$

APPENDIX C

Implementation of DOOM Protocol

```
/*

   This module implements the DOOM multicast scheduling algorithm as explained in Chapter
   5 for meeting QoS requirements of multiple end users

   DOOM multicast server performs TRABOL based congestion control for heterogeneous end
   users select data according to Type 1 or Type 2 requirement with UNIFORM Drops

   Static Tables are maintained for data selection and TDM scheme is used to schedule data
   for transmission for multiple end users.

   A variant of this algorithm with TDM based approach is used to select data/samples at
   intermediate overlay nodes using packet marking explained in Chapter 6

   Input is the archive file that contains multiple realizations of the simulated radar
   data client computes both reflectivity and Doppler velocity
 */

#include "multicast.h"
#include <stdlib.h>
#include <malloc.h>
#include <pthread.h>
#include <time.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/uio.h>
#include <netinet/in.h>
#include <stdio.h>
#include <math.h>

#define MAX_RATE_SUPPORTED 100                              // In Mbps per radar source
#define RAY_SIZE_IN_MEGA_BYTES 2                            // 2 Mega Bytes
#define MEGA_BITS                   1000000                 // BW
#define MEGA_BYTES          1048576                         // Programming use
#define RAY_SIZE_IN_BYTES   NUM_SAMPLES*SAMPLE_SIZE*NUM_GATES   // Ray Size in Bytes
#define RAY_SIZE_IN_BITS    RAY_SIZE_IN_BYTES*8             // 2 Megabits is the Ray
Size
#define CLOCK_TICK_RESOLUTION 10.0          // Clock Tick Resolution is 10ms
#define MAX_CLIENT_IN_WINDOW  10            // How many clients that can be assigned one
schedule
#define NUM_MAX_RATES   100                 // Define Different Max Rates supported at
the Server end. Number of different LRT Tables.
#define RATE_INCREMENT  1.0                 // Increment for Increase Phase
#define PACKET_SIZE     1500                // Total bytes transmitted to client in one
time slot
#define MAX_BW                      100             // Highest RT BW entry in Megabits
per second
#define MAX_CLIENTS     100                 // How to determine number of clients being
served ?
#define MAX_CYCLES      10                  // Clock Ticker Cycles
#define FAIL           -1                   // Fail flag

// Types of Packets sent by Client
#define REQUEST         1
#define FEEDBACK        2
#define END             3
#define DATA            4

#define ADDRESS_LEN    sizeof(struct sockaddr_in)

#define NUM_RAYS        500                 // Number of Rays transmitted by
#define SAMPLE_SIZE     16                  // Number of Bytes in Each Sample of a gate
#define NUM_GATES       256

#define NUM_SAMPLES     64                  // Samples per gate

#define MAX_SWEEP       20                  // Number of sweeps for which data to be
transmitted


/*****************************************************************************************

DATA QUALITY RELATED DEFINITION
 *****************************************************************************************/

#define DATA_QUALITY_SINGLE  1
#define DATA_QUALITY_PAIR    2
#define DATA_QUALITY_TRIPLET 3

#define DATA_LOSS_UNIFORM    1
#define DATA_LOSS_GROUP      2


#define DATA_TYPE_1         1           // Data quality single and uniform drop
#define DATA_TYPE_2         2           // Data quality pair   and uniform drop

float MAX_RATE_SUPPORTED_IN_BITS =  MAX_RATE_SUPPORTED*MEGA_BITS;
int MIN_RATE_SUPPORTED;
int resolution;                     // resolution of each time slot in terms of BW
int main_data_len,last_data_len; // Size of the Data Information sent to the client in the 1
clock tick
int total_data_transmitted =0;
typedef struct _timer_definition_
          {
          int type;
          struct sigevent evp;
          struct itimerspec timeout;
          } timer_definition;

RT_table_type RT_table[NUM_MAX_RATES]; //Stores Different Highest Rates supported, Each bit
of entry refers to the time slot
RT_table_type RT_table_second[NUM_MAX_RATES];

struct _client_list_table_ *client_list_table_head_ptr;// Head ptr to linklist of nodes
int client_list_table_count;         // Number of Active Clients in the Linklist of client
nodes.
int client_count[NUM_MAX_RATES];
int schedule_to_client_map_table[NUM_MAX_RATES][MAX_CLIENT_IN_WINDOW];
// Maps Clients requesting particular schedule

// Function Declarations
void init_tables();
void rate_2_sample_mapper_type_1();
void rate_2_sample_mapper_type_2();
void set_rt_table();
void add_clients_request(uint32_t client_ip,uint16_t client_port, int target_rate, int
min_rate, int data_type) ;
```

```
void simulate_time_ticker();
struct _client_list_ *remove_client_node(struct _client_list_ *temp_ptr, struct
_client_list_ *prev_temp_ptr, int data_type);
void add_client_node(int current_rate_index,struct _client_list_ *client_ptr, int
data_type);
void change_client_schedule(int current_rate_idx, int target_rate_idx, int client_ip, int
client_port, short int packet_count, short int ray_num,int flag,int data_type_requirment);
void time_ticker();
void send_data_type_1(int server_send_sock, int tick, int rate_index,int
data_type_requirement);
void send_data_type_2(int server_send_sock, int tick, int rate_index, int
data_type_requirement);
void print_data(int);
void release_memory();
inline void create_send_packet();
sig_handler();

struct sigaction sig_act;
sigset_t mask;
int size_of_schedule;
int total_active_clients;
data_packet_type send_data_packet;
int actual_raw_data_per_packet,ray_data_offset; // Used for Packetization
short int current_ray_num;
int sweep_count=0;          // Count to keep track of number of sweeps of data already sent
int tick_count;          // Keep track of current tick

pthread_t *thread_ptr;
timer_definition timers = {0,{0,SIGUSR1},{0,9500000,1,0}};
// For 10ms timer on Itanium Processor with Enterprise Linux
volatile int usr1;
int server_send_sock;
timer_t timer_id;
FILE *data_fp;
char *file_buffer,*file_buffer_ptr,*release_file_buffer;
int packet_send_count =0;
int packet_data=0;
struct sockaddr_in my_client_addr;
workload_table_type window_load[sizeof(int)*8];
struct _client_list_ *last_client_ptr;
int client_schedule_index,next_seq_number;
request_node_type *request_node_list_head_ptr=NULL;
// To maintain a list of all those client who's request arrives in the current window
feedback_node_type *feedback_node_list_head_ptr=NULL;
// To maintain list of all clients who's feedback has arrived
int primary_table = 1;

int main(int argc, char *argv[])
{
          int i,rate;
          int length,server_sock;
          int return_num;
          uint32_t client_ip_address;
          uint16_t client_port;
          int port ;
          char temp_data_pkt_ptr[20];
          struct sockaddr_in serv_addr,client_addr;
          packet_format input_packet,temp_input_packet;

          if (argc<3)
                    {
                    printf("Enter <server> <port> <packet size>\n");
                    exit(1);
                    }
          port = atoi(argv[1]);
          packet_data = atoi(argv[2]);
          init_tables();
          set_rt_table();
          rate_2_sample_mapper_type_2();
          rate_2_sample_mapper_type_1();

          // Open Simulated Radar Data File
          data_fp    =
          fopen("/space/radar_data/X_Band_data_256gates_64samp_500rays_06_11_2005.bytes"
               ,"rb");
          if (data_fp==NULL)
                    {
                    printf("Error Opening File\n");
                    exit(1);
                    }
          file_buffer = (char *)calloc((RAY_SIZE_IN_BYTES+sizeof
                    (file_ray_header_type))*NUM_RAYS,1);
          release_file_buffer = file_buffer;
          if (file_buffer==NULL)
                    {
                    printf("CALLOC FAILED\n");
                    exit(1);
                    }
          fread(file_buffer,(RAY_SIZE_IN_BYTES+sizeof(file_ray_header_type))*NUM_RAYS,1,dat
a_fp);
          for (i=0;i<(RAY_SIZE_IN_BYTES+sizeof(file_ray_header_type)/sizeof(float));i++)
                    {
                    char temp_buffer[4];
                    unsigned long temp_long;
                    temp_long = ntohl(*(unsigned int *)(file_buffer+4*i));
                    memcpy(temp_buffer,&temp_long,4);
                    }
          sigemptyset(&mask);
          sig_act.sa_handler = (void *)sig_handler;
          sig_act.sa_flags = 0;
          sigemptyset(&sig_act.sa_mask);
          usr1= 0;
          if ((server_sock = socket(AF_INET,SOCK_DGRAM,0)) < 0)
                    {
                    printf("RADAR server: receive request socket error \n");
                    exit(0);
                    }
          if ((server_send_sock = socket(AF_INET,SOCK_DGRAM,0)) < 0)
                    {
                    printf("RADAR server: send socket error \n");
                    exit(0);
                    }
          if ((thread_ptr=(pthread_t *)malloc(sizeof(pthread_t)))==0)
                    {
                    perror("Error in Thread Allocation");
                    exit(1);
                    }
          // Start Timer Thread
          if (pthread_create(thread_ptr,NULL, (void *)time_ticker, NULL)!=0)
                    {
                    perror("Error Starting a Thread");
                    exit(1);
                    }
          bzero((char *) &serv_addr,sizeof(serv_addr));
          serv_addr.sin_family      = AF_INET;
          serv_addr.sin_port        = htons(port);
          serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
          if (bind(server_sock,(struct sockaddr *)&serv_addr,sizeof(serv_addr)) < 0)
                    {
                    printf("Multicast server: Bind error \n");
                    exit(0);
                    }
          length = sizeof(struct sockaddr_in);
          while (1)
                    {
                    if ((return_num=recvfrom(server_sock, &temp_input_packet,
               sizeof(input_packet),0,(struct sockaddr *)&my_client_addr,&length))<0)
                              {
```

```
                    perror("FTP server: Couldn't Read Datagram");
                    exit(1);
                    }
                client_ip_address          = ntohl(my_client_addr.sin_addr.s_addr);
                client_port                = ntohs(my_client_addr.sin_port);
                input_packet.current_rate =
(int)ntohl(temp_input_packet.current_rate);
                input_packet.pkt_type      = (int)ntohl(temp_input_packet.pkt_type);
                input_packet.target_rate   = (int)ntohl(temp_input_packet.target_rate);
                input_packet.min_rate      = (int)ntohl(temp_input_packet.min_rate);
                input_packet.ray_num       = (int)ntohs(temp_input_packet.ray_num);
                input_packet.packet_count =
(int)ntohs(temp_input_packet.packet_count);
                input_packet.data_type_requirement =
            (int)ntohl(temp_input_packet.data_type_requirement);

                switch (input_packet.pkt_type)
                        {
                        case REQUEST:
                                {
                                        if (request_node_list_head_ptr==NULL)
                                                {
                                                // First request from
client during the
                                        current window.
                                                request_node_list_head_ptr
= (struct
                                        _request_node_list_ *)malloc(sizeof(struct
                                        _request_node_list_));
                                        request_node_list_head_ptr->client_ip_address
                                        = client_ip_address;
                                                request_node_list_head_ptr-
>client_port
                                        = client_port;
                                                request_node_list_head_ptr-
>target_rate
                                        = input_packet.target_rate;
                                                request_node_list_head_ptr-
>min_rate
                                        = input_packet.min_rate;
                                                request_node_list_head_ptr-
                                        >data_type_requirement=
                                        input_packet.data_type_requirement;
                                                request_node_list_head_ptr-
>next = NULL;
                                                }

                                        else
                                                {
                                                // Request is not from the
first client in
                                        // this CYCLE
                                                request_node_type *
temp_ptr;
                                                temp_ptr = (struct
_request_node_list_
                                        *)malloc(sizeof(struct
                                        _request_node_list_));
                                                temp_ptr->client_ip_address
=
                                        client_ip_address;
                                                temp_ptr->client_port
= client_port;
                                                temp_ptr->target_rate
=
                                        input_packet.target_rate;
                                                temp_ptr->min_rate
=
                                                input_packet.min_rate;
                                                temp_ptr-
>data_type_requirement =
                                                input_packet.data_type_requirement;
                                                temp_ptr->next
=
                                                request_node_list_head_ptr;
                                                request_node_list_head_ptr
= temp_ptr;                                                     //Latest
request is made head in this list
                                        // though it came last.
                                                }

                                                break;
                                        }
                                case FEEDBACK:
                                        {
                                                int target_idx, current_idx;
                                                // Client Index points to the location
of client
                                        entry in the client table.
                                                target_idx=
(int)((double)(MAX_RATE_SUPPORTED -
                                         input_packet.target_rate)/(double)RATE_INCREMENT);
                                                current_idx =
(int)((double)(MAX_RATE_SUPPORTED -
                                        input_packet.current_rate)/(double)RATE_INCREMENT);

                                                if (feedback_node_list_head_ptr==NULL)
                                                        {
                                                        // This is the first
feedback received
                                                during the CYCLE

                                                feedback_node_list_head_ptr=
                                                (feedback_node_type
                                                *)(malloc(sizeof(feedback_node_type)));

        feedback_node_list_head_ptr->current_index =
                                                current_idx;

        feedback_node_list_head_ptr->target_index  =
                                                target_idx;

        feedback_node_list_head_ptr-
                                                >client_ip_address = client_ip_address;

        feedback_node_list_head_ptr->client_port
                                                = client_port;

        feedback_node_list_head_ptr->packet_count
                                                = input_packet.packet_count;

        feedback_node_list_head_ptr->ray_num
                                                = input_packet.ray_num;

        feedback_node_list_head_ptr->flag
                                                = 0;

        feedback_node_list_head_ptr-
                                                 >data_type_requirement =
                                                input_packet.data_type_requirement;

        feedback_node_list_head_ptr->next
                                                = NULL;
                                                        }
                                                else
                                                        {
```

```
*temp_ptr;

(feedback_node_type *)malloc(sizeof(feedback_node_type));

current_idx;

target_idx;

= client_ip_address;

= client_port;

= input_packet.packet_count;

= input_packet.ray_num;

= 0;

= feedback_node_list_head_ptr;

>data_type_requirement = input_packet.data_type_requirement;

= temp_ptr;
                                                        feedback_node_type

                                                        temp_ptr =

                                                        temp_ptr->current_index =

                                                        temp_ptr->target_index  =

                                                        temp_ptr->client_ip_address

                                                        temp_ptr->client_port

                                                        temp_ptr->packet_count

                                                        temp_ptr->ray_num

                                                        temp_ptr->flag

                                                        temp_ptr->next

                                                        temp_ptr-

                                                        feedback_node_list_head_ptr

                                                        }
                                                break;
                                        }
                                case END:
                                        {
                                                int target_idx, current_idx;
                                                /* Request for End of Transmission
                                                   Server should stop sending the data
to this client.
                                                */
                                                target_idx  =
(int)((double)(MAX_RATE_SUPPORTED - input_packet.target_rate)/(double)RATE_INCREMENT);
                                                current_idx =
(int)((double)(MAX_RATE_SUPPORTED - input_packet.current_rate)/(double)RATE_INCREMENT);

        change_client_schedule(current_idx,target_idx, client_ip_address,client_port,
input_packet.packet_count, input_packet.ray_num, 1,input_packet.data_type_requirement);
                                                break;
                                        }
                                default:
                                        {
                                                break;
                                        }
                                }
                        }
}


/*
  Given a rate table, determine statically what are the samples that need to be transmitted.
All these computations can be done statically and only things that should be done at the
execution time is lookup of this static table

This is required to implement the Data Quality support in multicast server.
Goal is to transmit maximum number of complete samples for all gates that are possible for a
given rate. At this time implementation is for the case

SAMPLE GROUP REQUIREMENT is : SINGLE or  PAIRS at a time
LOSS REQUIREMENT         is : UNIFORM, based on similar approach GROUP LOSS can be
implemented

Prepare a static table of samples to be sent for a particular rate.
In each time window, it is decided to transmit even number of samples because that would
help satisfying both single and pair requirements (one at a time)

When group sample cannot be transmitted completely then only sample of first few gates are
transmitted.

*/

// SINGLE SAMPLE AND UNIFORM DROP

void rate_2_sample_mapper_type_1()
{
          int i,j,k, num_samples_per_time_slot;
          float ray_time;
          int drop_per_group,group_per_drop,drop_count,group_count;
          int last_partial_gate_count = NUM_GATES,last_partial_gate_number = NUM_GATES;
      // To track the number of gates transmitted for last sample group in last time
      // slot
          sample_schedule_type sample_list[NUM_SAMPLES];
          sample_schedule_type **sample_schedule_table;
          num_samples_per_time_slot =
(int)ceilf((double)NUM_SAMPLES/(double)size_of_schedule);
          sample_schedule_table = (sample_schedule_type
**)malloc(sizeof(sample_schedule_type)*size_of_schedule);
          for (i=0;i<size_of_schedule;i++)
                    {
                    sample_schedule_table[i] = (sample_schedule_type
*)malloc(sizeof(sample_schedule_type)*num_samples_per_time_slot);
                    }
          for (i=0; i<NUM_MAX_RATES; i++)
                    {
                    int j,k,l,sample_index,
sample_group_requirement,num_complete_samples_to_transmit, num_gates,num_data_group_samples;
                    int total_samples_to_transmit,total_samples_to_drop;
                    last_partial_gate_count=NUM_GATES;
                    // sample_table_ptr field is a list that stores the number of gates
for which particular sample is transmitted
                    total_samples_to_transmit  =
(int)(double)NUM_SAMPLES*(double)RT_table[i].base_rate/(double)MAX_RATE_SUPPORTED;
                    sample_group_requirement   = (int) DATA_QUALITY_SINGLE;
                    num_data_group_samples     =
total_samples_to_transmit/sample_group_requirement;
                    // For the last group in the sample schedule, all gates may not be
transmitted, so for how many gates
                    // samples can be transmitted is computed below, note it is only for
the last group for a given rate.
                    num_gates = (int)((((double)NUM_SAMPLES*
(double)RT_table[i].base_rate/ (double)MAX_RATE_SUPPORTED) -
                                                                        (double)
num_data_group_samples*sample_group_requirement)*(double)NUM_GATES/(double)sample_group_requ
irement);

                    if (num_gates>0)
                              if (total_samples_to_transmit%sample_group_requirement)
                                        {
                                        total_samples_to_transmit +=
sample_group_requirement - (total_samples_to_transmit%sample_group_requirement);
                                        }
                              else

          total_samples_to_transmit+=sample_group_requirement;

                    // Make total samples to transmit multiple of sample group requirement

                    if (total_samples_to_transmit%sample_group_requirement)
                              {
                              total_samples_to_transmit += (int)sample_group_requirement-
(total_samples_to_transmit%sample_group_requirement);
                              }
                    total_samples_to_drop      = (int)NUM_SAMPLES-
total_samples_to_transmit; // Sample of all gates are dropped
```

```
                    num_data_group_samples     =
total_samples_to_transmit/sample_group_requirement;
                    memset(sample_list, 0, sizeof(sample_schedule_type)*NUM_SAMPLES);
                    group_per_drop = num_data_group_samples; // Number of groups to
transmit, one of the last group may be partial
                    drop_per_group = total_samples_to_drop;  // Number of samples that
need to be dropped
                    sample_index=0;
                    if (num_data_group_samples>=total_samples_to_drop)
                              {
                              for (j=0;j<total_samples_to_drop-1;j++)// Distribute loss
samples uniformly
                                        {
                                        for
(k=0;k<num_data_group_samples/total_samples_to_drop;k++) //After number of groups 1 loss is
introduced
                                                  {
                                                  for
(l=0;l<sample_group_requirement;l++)
                                                            {

          sample_list[sample_index].sample_number = sample_index+1;

          sample_list[sample_index].total_gates   = NUM_GATES;

          sample_list[sample_index].first_gate    = 1;

          sample_list[sample_index].last_gate     = NUM_GATES;
                                                                      sample_index++;
                                                                      }
                                                  }

                                        // At this time schedule one lost sample
                                        sample_list[sample_index].sample_number   =
sample_index+1;
                                        sample_list[sample_index].total_gates     = 0;
                                        sample_list[sample_index].first_gate      = 0;
                                        sample_list[sample_index].last_gate       = 0;
                                        sample_index++;
                                        }

                              // Note that at this time total_samples_to_drop -1 samples
have been scheduled for drop.
                              // Schedule the last part of the sample_list
                              for (k=0;k<((num_data_group_samples/total_samples_to_drop)
+ (num_data_group_samples%total_samples_to_drop)-1);k++)
                                        {
                                        for (l=0;l<sample_group_requirement;l++)
                                                  {

          sample_list[sample_index].sample_number = sample_index+1;
                                                            sample_list[sample_index].total_gates
= NUM_GATES;
                                                            sample_list[sample_index].first_gate
= 1;
                                                            sample_list[sample_index].last_gate
= NUM_GATES;
                                                            sample_index++;
                                                            }
                                        }

                              // Last group gates has not been set yet, as it might be
required to send only partial number of gates for that so do it here

                              if (num_gates>0)
                                        {
                                        for (l=0;l<sample_group_requirement;l++)
                                                  {
          sample_list[sample_index].sample_number = sample_index+1;
                                                            sample_list[sample_index].total_gates
= num_gates;
                                                            sample_list[sample_index].first_gate
= 1;
                                                            sample_list[sample_index].last_gate
= num_gates;
                                                            sample_index++;
                                                            }
                                                  }
                                        else
                                                  {
                                                  // When num_gates is 0 then it indicates that for
last group all gates were sent
                                                  for (l=0;l<sample_group_requirement;l++)
                                                            {

          sample_list[sample_index].sample_number = sample_index+1;
                                                            sample_list[sample_index].total_gates
= NUM_GATES;
                                                            sample_list[sample_index].first_gate
= 1;
                                                            sample_list[sample_index].last_gate
= NUM_GATES;
                                                            sample_index++;
                                                            }
                                                  }

                                        // Schedule last packet for the drop
                                        sample_list[sample_index].sample_number   = sample_index+1;
                                        sample_list[sample_index].total_gates     = 0;
                                        sample_list[sample_index].first_gate      = 0;
                                        sample_list[sample_index].last_gate       = 0;
                                        sample_index++;

                                        }
                              else
                                        {
                                        for (j=0;j<num_data_group_samples-1;j++) // Schedule
num_Data_group_Samples - 1 groups
                                                  {
                                                  // schedule following sample for transmission
                                                  for (k=0;k<sample_group_requirement;k++)
                                                            {

          sample_list[sample_index].sample_number = sample_index+1;
                                                            sample_list[sample_index].total_gates
= NUM_GATES;
                                                            sample_list[sample_index].first_gate
= 1;
                                                            sample_list[sample_index].last_gate
= NUM_GATES;
                                                            sample_index++;
                                                            }
                                                  // Drop following samples
                                                  for
(l=0;l<total_samples_to_drop/num_data_group_samples;l++)
                                                            {

          sample_list[sample_index].sample_number = sample_index+1;
                                                            sample_list[sample_index].total_gates
= 0;
                                                            sample_list[sample_index].first_gate
= 0;
                                                            sample_list[sample_index].last_gate
= 0;
                                                            sample_index++;
                                                            }
```

```
                                        }
                                // Last group has not been scheduled because it migh need
only partial number of gates
                                if (num_gates>0)
                                        {
                                        for (l=0;l<sample_group_requirement;l++)
                                                {

        sample_list[sample_index].sample_number = sample_index+1;
                                                sample_list[sample_index].total_gates
= num_gates;
                                                sample_list[sample_index].first_gate
= 1;
                                                sample_list[sample_index].last_gate
= num_gates;
                                                sample_index++;
                                                }
                                        }
                                else
                                        {
                                        for (l=0;l<sample_group_requirement;l++)
                                                {

        sample_list[sample_index].sample_number = sample_index+1;
                                                sample_list[sample_index].total_gates
= NUM_GATES;
                                                sample_list[sample_index].first_gate
= 1;
                                                sample_list[sample_index].last_gate
= NUM_GATES;
                                                sample_index++;
                                                }
                                        }

                                // Drop last chunk
                                for
(l=0;l<total_samples_to_drop/num_data_group_samples+total_samples_to_drop%num_data_group_sam
ples;l++)
                                        {
                                        sample_list[sample_index].sample_number =
sample_index+1;
                                        sample_list[sample_index].total_gates   = 0;
                                        sample_list[sample_index].first_gate    = 0;
                                        sample_list[sample_index].last_gate     = 0;
                                        sample_index++;
                                        }
                                }
                        for (l=0;l<NUM_SAMPLES;l++)
                                {
                                printf("For rate %d sample %d had gates %d \n",
RT_table[i].base_rate, sample_list[l].sample_number,sample_list[l].total_gates);
                                }

                        /*
                          If partial sample is left, that is sample of all gates cannot be
transmitted then we should send whatever is possible while meeting the group requirement.
Thus we should compute number of gates for which samples can be transmitted while
satisfying group requirements. At this time distribution of samples is ready, now next task
is to assign samples to time slots
                        */

                        RT_table[i].sample_table_ptr_type_1     = (sample_schedule_type
**)malloc(sizeof(&size_of_schedule)*size_of_schedule);
                        RT_table[i].time_slot_data_size_type_1 = (int
*)malloc(size_of_schedule*sizeof(int));  // Allocate memory to store lenght of data to be
transmitted in the given window.

        memset(RT_table[i].time_slot_data_size_type_1,0,size_of_schedule*sizeof(int));
                        for (j=0;j<size_of_schedule;j++)
                                {
                                RT_table[i].sample_table_ptr_type_1[j]=
(sample_schedule_type *)malloc(sizeof(sample_schedule_type)*num_samples_per_time_slot);
                                }
                        j=0;  // Index for the Time Slot
                        k=0;  // Index for the Sample in the Main Sample List
                        while (j<size_of_schedule && k<NUM_SAMPLES) // Time Slots, Divide the
level 1 schedule among different time slots
                                {
                                int time_slot_done=0;
                                int start_sample_index = k;         // Index of First Sample
in the given time slot
                                int l =0;                           // Index of sample
within the Time Slot
                                while (!time_slot_done)
                                        {
                                        if (k<NUM_SAMPLES &&
sample_list[k].total_gates>0)
                                                {
                                                if (l==0 &&
last_partial_gate_count==NUM_GATES)         // There was no truncation of samples between two
adjacent windows.
                                                        {

          RT_table[i].sample_table_ptr_type_1[j][l].sample_number =
sample_list[k].sample_number;

          RT_table[i].sample_table_ptr_type_1[j][l].total_gates   =
sample_list[k].total_gates;

          RT_table[i].sample_table_ptr_type_1[j][l].first_gate    = 1;

          RT_table[i].sample_table_ptr_type_1[j][l].last_gate     =
sample_list[k].total_gates;

          RT_table[i].time_slot_data_size_type_1[j] +=
RT_table[i].sample_table_ptr_type_1[j][l].total_gates*SAMPLE_SIZE*sample_group_requirement;

          last_partial_gate_count=RT_table[i].sample_table_ptr_type_1[j][l].total_gates;
                                                        last_partial_gate_number
= RT_table[i].sample_table_ptr_type_1[j][l].last_gate;
                                                        l++;
                                                        k
+=sample_group_requirement;
                                                        if (k-
start_sample_index>=num_samples_per_time_slot)
                                                                {

          time_slot_done=1;

                                                                j++;
                                                                }
                                                        }
                                                else if (l==0)        // There is need
to trucate, so send the remaining gates in the new time slot.
                                                        {

          RT_table[i].sample_table_ptr_type_1[j][l].sample_number =
sample_list[k].sample_number;

          RT_table[i].sample_table_ptr_type_1[j][l].total_gates   = NUM_GATES-
last_partial_gate_count;

          RT_table[i].sample_table_ptr_type_1[j][l].first_gate    =
last_partial_gate_number+1;

          RT_table[i].sample_table_ptr_type_1[j][l].last_gate     = NUM_GATES;

          RT_table[i].time_slot_data_size_type_1[j] +=
RT_table[i].sample_table_ptr_type_1[j][l].total_gates*SAMPLE_SIZE*sample_group_requirement;
```

```
                                                last_partial_gate_count  =
RT_table[i].sample_table_ptr_type_1[j][l].total_gates;
                                                last_partial_gate_number =
RT_table[i].sample_table_ptr_type_1[j][l].last_gate;
                                                l++;
                                                k +=
sample_group_requirement;
                                                if (k-
start_sample_index>=num_samples_per_time_slot)
                                                        {

          time_slot_done=1;

                                                        j++;
                                                        }
                                                }
                                        else if (l>0)
                                                {
                                                /*
                                                   // Check if k is within
bounds
                                                */
                                                if
(k+sample_group_requirement<=num_samples_per_time_slot*(j+1))
                                                        {
                                                        // When complete
samples can be sent int he time slot then include it

          RT_table[i].sample_table_ptr_type_1[j][l].sample_number =
sample_list[k].sample_number;

          RT_table[i].sample_table_ptr_type_1[j][l].total_gates   =
sample_list[k].total_gates;

          RT_table[i].sample_table_ptr_type_1[j][l].first_gate    = 1;

          RT_table[i].sample_table_ptr_type_1[j][l].last_gate     =
sample_list[k].total_gates;

          RT_table[i].time_slot_data_size_type_1[j] +=
RT_table[i].sample_table_ptr_type_1[j][l].total_gates*SAMPLE_SIZE*sample_group_requirement;

          last_partial_gate_count  = RT_table[i].sample_table_ptr_type_1[j][l].total_gates;

          last_partial_gate_number = sample_list[k].total_gates;
                                                l++;
                                                if
(k+sample_group_requirement>=NUM_SAMPLES)
                                                        {

          time_slot_done=1;
                                                        k =
k+sample_group_requirement;     // Set it more than the NUM_SAMPLES to break the outermost
loop

          break;
                                                        }
                                                k
+=sample_group_requirement;
                                                if (k-
start_sample_index>=num_samples_per_time_slot)
                                                        {

          time_slot_done=1;

                                                        j++;
                                                        }
                                                }
                                        else
                                                {
                                                // All samples
cannot be transmitted for a given data quality
                                                int
temp,gates_per_sample;
                                                temp =
(k+sample_group_requirement)- num_samples_per_time_slot*(j+1); // number of samples that
cannot be transmitted
                                                temp =
sample_group_requirement - temp;  // Number of samples that can be transmitted
                                                gates_per_sample
= temp*NUM_GATES/sample_group_requirement;

          RT_table[i].sample_table_ptr_type_1[j][l].sample_number =
sample_list[k].sample_number;

          RT_table[i].sample_table_ptr_type_1[j][l].total_gates   = gates_per_sample;

          RT_table[i].sample_table_ptr_type_1[j][l].first_gate    = 1;

          RT_table[i].sample_table_ptr_type_1[j][l].last_gate     = gates_per_sample;

          RT_table[i].time_slot_data_size_type_1[j] +=
RT_table[i].sample_table_ptr_type_1[j][l].total_gates*SAMPLE_SIZE*sample_group_requirement;
                                                // Donot change
k, k sample is incomplete so remaining gates are sent in next time slot window.

          last_partial_gate_count  = gates_per_sample;

          last_partial_gate_number = gates_per_sample;
                                                time_slot_done =
1;        // Move to next slot
                                                j++;
                                                }
                                                } // l>0
                                        }
                                else if (k<NUM_SAMPLES)
                                        {
                                        k++;   // Sample was not scheduled, so
check next sample
                                        if (k-
start_sample_index>=num_samples_per_time_slot)
                                                {
                                                time_slot_done=1;
                                                j++;
                                                }
                                        }
                                if (k>=NUM_SAMPLES)
                                        time_slot_done=1;
                                }
                        }
                }
}


// GROUP PAIR AND UNIFORM DROP
void rate_2_sample_mapper_type_2()
{
          int i,j,k, num_samples_per_time_slot;
          float ray_time;
          int drop_per_group,group_per_drop,drop_count,group_count;
          int last_partial_gate_count = NUM_GATES,last_partial_gate_number = NUM_GATES;  //
To track the number of gates transmitted for last sample group in last time slot
          sample_schedule_type sample_list[NUM_SAMPLES];

          sample_schedule_type **sample_schedule_table;
          num_samples_per_time_slot =
(int)ceilf((double)NUM_SAMPLES/(double)size_of_schedule);
          sample_schedule_table = (sample_schedule_type
**)malloc(sizeof(sample_schedule_type)*size_of_schedule);
```

```
        for (i=0;i<size_of_schedule;i++)
                {
                sample_schedule_table[i] = (sample_schedule_type
*)malloc(sizeof(sample_schedule_type)*num_samples_per_time_slot);
                }
        for (i=0; i<NUM_MAX_RATES; i++)
                {
                int j,k,l,sample_index,
sample_group_requirement,num_complete_samples_to_transmit, num_gates,num_data_group_samples;
                int total_samples_to_transmit,total_samples_to_drop;
                last_partial_gate_count=NUM_GATES;
                // sample_table_ptr field is a list that stores the number of gates
for which particular sample is transmitted
                total_samples_to_transmit  =
(int)(double)NUM_SAMPLES*(double)RT_table[i].base_rate/(double)MAX_RATE_SUPPORTED;
                sample_group_requirement   = (int) DATA_QUALITY_PAIR;
                num_data_group_samples     =
total_samples_to_transmit/sample_group_requirement;

                // For the last group in the sample schedule, all gates may not be
transmitted, so for how many gates
                // samples can be transmitted is computed below, note it is only for
the last group for a given rate.
                num_gates = (int)((((double)NUM_SAMPLES*
(double)RT_table[i].base_rate/ (double)MAX_RATE_SUPPORTED) -
                                                                        (double)
num_data_group_samples*sample_group_requirement)*(double)NUM_GATES/(double)sample_group_requ
irement);
                if (num_gates>0)
                        if (total_samples_to_transmit%sample_group_requirement)
                                {
                                total_samples_to_transmit +=
sample_group_requirement - (total_samples_to_transmit%sample_group_requirement);
                                }
                        else

        total_samples_to_transmit+=sample_group_requirement;

                // Make total samples to transmit multiple of sample group requirement
                if (total_samples_to_transmit%sample_group_requirement)
                        {
                        total_samples_to_transmit += (int)sample_group_requirement-
(total_samples_to_transmit%sample_group_requirement);
                        }
                total_samples_to_drop      = (int)NUM_SAMPLES-
total_samples_to_transmit; // Sample of all gates are dropped
                num_data_group_samples     =
total_samples_to_transmit/sample_group_requirement;
                memset(sample_list, 0, sizeof(sample_schedule_type)*NUM_SAMPLES);
                group_per_drop = num_data_group_samples; // Number of groups to
transmit, one of the last group may be partial
                drop_per_group = total_samples_to_drop;      // Number of samples that
need to be dropped
                sample_index=0;
                if (num_data_group_samples>=total_samples_to_drop)
                        {
                        for (j=0;j<total_samples_to_drop-1;j++)     // Distribute
loss samples uniformly
                                {
                                for
(k=0;k<num_data_group_samples/total_samples_to_drop;k++) //After number of groups 1 loss is
introduced
                                        {
                                        for
(l=0;l<sample_group_requirement;l++)
                                                {

        sample_list[sample_index].sample_number = sample_index+1;
        sample_list[sample_index].total_gates   = NUM_GATES;
        sample_list[sample_index].first_gate    = 1;
        sample_list[sample_index].last_gate     = NUM_GATES;
                                                sample_index++;
                                                }
                                        }

                                // At this time schedule one lost sample
                                sample_list[sample_index].sample_number   =
sample_index+1;
                                sample_list[sample_index].total_gates     = 0;
                                sample_list[sample_index].first_gate      = 0;
                                sample_list[sample_index].last_gate       = 0;
                                sample_index++;
                                }

                        // Note that at this time total_samples_to_drop -1 samples
have been scheduled for drop.
                        // Schedule the last part of the sample_list
                        for (k=0;k<((num_data_group_samples/total_samples_to_drop)
+ (num_data_group_samples%total_samples_to_drop)-1);k++)
                                {
                                for (l=0;l<sample_group_requirement;l++)
                                        {

        sample_list[sample_index].sample_number = sample_index+1;
                                        sample_list[sample_index].total_gates
= NUM_GATES;
                                        sample_list[sample_index].first_gate
= 1;
                                        sample_list[sample_index].last_gate
= NUM_GATES;
                                        sample_index++;
                                        }
                                }

                        // Last group gates has not been set yet, as it might be
required to send only partial number of gates for that so
                        // do it here
                        if (num_gates>0)
                                {
                                for (l=0;l<sample_group_requirement;l++)
                                        {

        sample_list[sample_index].sample_number = sample_index+1;
                                        sample_list[sample_index].total_gates
= num_gates;
                                        sample_list[sample_index].first_gate
= 1;
                                        sample_list[sample_index].last_gate
= num_gates;
                                        sample_index++;
                                        }
                                }
                        else
                                {
                                // When num_gates is 0 then it indicates that for
last group all gates were sent
                                for (l=0;l<sample_group_requirement;l++)
                                        {

        sample_list[sample_index].sample_number = sample_index+1;
                                        sample_list[sample_index].total_gates
= NUM_GATES;
                                        sample_list[sample_index].first_gate
= 1;
```

```
                                        sample_list[sample_index].last_gate
= NUM_GATES;
                                        sample_index++;
                                        }
                                }

                        // Schedule last packet for the drop
                        sample_list[sample_index].sample_number   = sample_index+1;
                        sample_list[sample_index].total_gates     = 0;
                        sample_list[sample_index].first_gate      = 0;
                        sample_list[sample_index].last_gate       = 0;
                        sample_index++;

                        }
                else
                        {
                        for (j=0;j<num_data_group_samples-1;j++) // Schedule
num_Data_group_Samples - 1 groups
                                {
                                // schedule following sample for transmission
                                for (k=0;k<sample_group_requirement;k++)
                                        {

        sample_list[sample_index].sample_number = sample_index+1;
                                        sample_list[sample_index].total_gates
= NUM_GATES;
                                        sample_list[sample_index].first_gate
= 1;
                                        sample_list[sample_index].last_gate
= NUM_GATES;
                                        sample_index++;
                                        }
                                // Drop following samples
                                for
(l=0;l<total_samples_to_drop/num_data_group_samples;l++)
                                        {

        sample_list[sample_index].sample_number = sample_index+1;
                                        sample_list[sample_index].total_gates
= 0;
                                        sample_list[sample_index].first_gate
= 0;
                                        sample_list[sample_index].last_gate
= 0;
                                        sample_index++;
                                        }

                                }
                        // Last group has not been scheduled because it migh need
only partial number of gates
                        if (num_gates>0)
                                {
                                for (l=0;l<sample_group_requirement;l++)
                                        {

        sample_list[sample_index].sample_number = sample_index+1;
                                        sample_list[sample_index].total_gates
= num_gates;
                                        sample_list[sample_index].first_gate
= 1;
                                        sample_list[sample_index].last_gate
= num_gates;
                                        sample_index++;
                                        }
                                }
                        else
                                {
                                for (l=0;l<sample_group_requirement;l++)
                                        {
        sample_list[sample_index].sample_number = sample_index+1;
                                        sample_list[sample_index].total_gates
= NUM_GATES;
                                        sample_list[sample_index].first_gate
= 1;
                                        sample_list[sample_index].last_gate
= NUM_GATES;
                                        sample_index++;
                                        }
                                }

                        // Drop last chunk
                        for
(l=0;l<total_samples_to_drop/num_data_group_samples+total_samples_to_drop%num_data_group_sam
ples;l++)
                                {
                                sample_list[sample_index].sample_number =
sample_index+1;
                                sample_list[sample_index].total_gates   = 0;
                                sample_list[sample_index].first_gate    = 0;
                                sample_list[sample_index].last_gate     = 0;
                                sample_index++;
                                }
                        }

                /*
                 If partial sample is left, that is sample of all gates cannot be
transmitted then we should send whatever is possible while
                  meeting the group requirement. Thus we should compute number of gates
for which samples can be transmitted while
                  satisfying group requirements.
                 At this time distribution of samples is ready, now next task is to
assign samples to time slots
                */
                RT_table[i].sample_table_ptr_type_2    = (sample_schedule_type
**)malloc(sizeof(&size_of_schedule)*size_of_schedule);
                RT_table[i].time_slot_data_size_type_2 = (int
*)malloc(size_of_schedule*sizeof(int));  // Allocate memory to store lenght of data to be
transmitted in the given window.

        memset(RT_table[i].time_slot_data_size_type_2,0,size_of_schedule*sizeof(int));
                for (j=0;j<size_of_schedule;j++)
                        {
                        RT_table[i].sample_table_ptr_type_2[j]=
(sample_schedule_type *)malloc(sizeof(sample_schedule_type)*num_samples_per_time_slot);
                        }
                j=0;  // Index for the Time Slot
                k=0;  // Index for the Sample in the Main Sample List
                while (j<size_of_schedule && k<NUM_SAMPLES) // Time Slots, Divide the
level 1 schedule among different time slots
                        {
                        int time_slot_done=0;
                        int start_sample_index = k;         // Index of First Sample
in the given time slot
                        int l =0;                                                   //
Index of sample within the Time Slot
                        while (!time_slot_done)
                                {
                                if (k<NUM_SAMPLES &&
sample_list[k].total_gates>0)
                                        {
                                        if (l==0 &&
last_partial_gate_count==NUM_GATES)         // There was no truncation of samples between two
adjacent windows.
                                                {

        RT_table[i].sample_table_ptr_type_2[j][l].sample_number =
sample_list[k].sample_number;
```

```
                RT_table[i].sample_table_ptr_type_2[j][l].total_gates   =
sample_list[k].total_gates;

                RT_table[i].sample_table_ptr_type_2[j][l].first_gate    = 1;

                RT_table[i].sample_table_ptr_type_2[j][l].last_gate     =
sample_list[k].total_gates;

                RT_table[i].time_slot_data_size_type_2[j] +=
RT_table[i].sample_table_ptr_type_2[j][l].total_gates*SAMPLE_SIZE*sample_group_requirement;

                last_partial_gate_count=RT_table[i].sample_table_ptr_type_2[j][l].total_gates;
                                                                last_partial_gate_number
= RT_table[i].sample_table_ptr_type_2[j][l].last_gate;
                                                                l++;
                                                                k
+=sample_group_requirement;
                                                                if (k-
start_sample_index>=num_samples_per_time_slot)
                                                                        {

                time_slot_done=1;
                                                                        j++;
                                                                        }
                                                        }
                                                else if (l==0)          // There is need
to trucate, so send the remaining gates in the new time slot.
                                                                {

                RT_table[i].sample_table_ptr_type_2[j][l].sample_number =
sample_list[k].sample_number;

                RT_table[i].sample_table_ptr_type_2[j][l].total_gates   = NUM_GATES-
last_partial_gate_count;

                RT_table[i].sample_table_ptr_type_2[j][l].first_gate    =
last_partial_gate_number+1;

                RT_table[i].sample_table_ptr_type_2[j][l].last_gate     = NUM_GATES;

                RT_table[i].time_slot_data_size_type_2[j] +=
RT_table[i].sample_table_ptr_type_2[j][l].total_gates*SAMPLE_SIZE*sample_group_requirement;

                                                                last_partial_gate_count  =
RT_table[i].sample_table_ptr_type_2[j][l].total_gates;
                                                                last_partial_gate_number =
RT_table[i].sample_table_ptr_type_2[j][l].last_gate;
                                                                l++;
                                                                k +=
sample_group_requirement;
                                                                if (k-
start_sample_index>=num_samples_per_time_slot)
                                                                        {

                time_slot_done=1;
                                                                        j++;
                                                                        }
                                                        }
                                                else if (l>0)
                                                        {
                                                        /*
                                                                Check if k is
within bounds
                                                        */
                                                        // Make Sanity Check, I
don't know if the following logic makes sense at all
                                                        if
(k+sample_group_requirement<=num_samples_per_time_slot*(j+1))
                                                                {
                                                                // When complete
samples can be sent int he time slot then include it

                RT_table[i].sample_table_ptr_type_2[j][l].sample_number =
sample_list[k].sample_number;

                RT_table[i].sample_table_ptr_type_2[j][l].total_gates   =
sample_list[k].total_gates;

                RT_table[i].sample_table_ptr_type_2[j][l].first_gate    = 1;

                RT_table[i].sample_table_ptr_type_2[j][l].last_gate     =
sample_list[k].total_gates;

                RT_table[i].time_slot_data_size_type_2[j] +=
RT_table[i].sample_table_ptr_type_2[j][l].total_gates*SAMPLE_SIZE*sample_group_requirement;

                last_partial_gate_count  = RT_table[i].sample_table_ptr_type_2[j][l].total_gates;

                last_partial_gate_number = sample_list[k].total_gates;
                                                                l++;
                                                                if
(k+sample_group_requirement>=NUM_SAMPLES)
                                                                        {

                time_slot_done=1;
                                                                        k =
k+sample_group_requirement;     // Set it more than the NUM_SAMPLES to break the outermost
loop

                break;
                                                                        }
                                                                k
+=sample_group_requirement;
                                                                if (k-
start_sample_index>=num_samples_per_time_slot)
                                                                        {

                time_slot_done=1;
                                                                        j++;
                                                                        }
                                                                }
                                                        else
                                                                {
                                                                // All samples
cannot be transmitted for a given data quality
                                                                int
temp,gates_per_sample;
                                                                temp =
(k+sample_group_requirement)- num_samples_per_time_slot*(j+1); // number of samples that
cannot be transmitted
                                                                temp =
sample_group_requirement - temp;  // Number of samples that can be transmitted
                                                                gates_per_sample
= temp*NUM_GATES/sample_group_requirement;

                RT_table[i].sample_table_ptr_type_2[j][l].sample_number =
sample_list[k].sample_number;

                RT_table[i].sample_table_ptr_type_2[j][l].total_gates   = gates_per_sample;

                RT_table[i].sample_table_ptr_type_2[j][l].first_gate    = 1;

                RT_table[i].sample_table_ptr_type_2[j][l].last_gate     = gates_per_sample;

                RT_table[i].time_slot_data_size_type_2[j] +=
RT_table[i].sample_table_ptr_type_2[j][l].total_gates*SAMPLE_SIZE*sample_group_requirement;
```

```
                                                                    // Donot change
k, k sample is incomplete so remaining gates are sent in next time slot window.

          last_partial_gate_count  = gates_per_sample;

          last_partial_gate_number = gates_per_sample;
                                                                    time_slot_done =
1;        // Move to next slot
                                                                    j++;
                                                                    }
                                                          } // l>0
                                                }
                                        else if (k<NUM_SAMPLES)
                                                {
                                                k++;   // Sample was not scheduled, so
check next sample
                                                if (k-
start_sample_index>=num_samples_per_time_slot)
                                                        {
                                                        time_slot_done=1;
                                                        j++;
                                                        }
                                                }
                                        if (k>=NUM_SAMPLES)
                                                time_slot_done=1;
                                        }
                                }
                        }
}


/*
  This function creates RT and LRT Tables
*/
void init_tables()
{
          // Variable Declaration Section
          int i,j,temp_main_data_len;      // For loop index variable
          // Set RT & LRT table enteries to 0, Before setting the schedule
          // Each Bit in each location specify the schedule for given time slot
          // MSB is the 1st Time slot of the schedule and LSB is the last Time slot of the
schedule

          int total_packets_required = 0;  // Number of packets required to send all the RAY
information
          float time_for_one_ray;
          actual_raw_data_per_packet    = packet_data - sizeof(file_ray_header_type)-
sizeof(local_header_type); // Actual raw data in packet
          actual_raw_data_per_packet   -= actual_raw_data_per_packet % SAMPLE_SIZE;  //
Should Store all bytes of last sample in the packet, cannot split

          // one sample among different packets.
          total_packets_required  =  RAY_SIZE_IN_BYTES/actual_raw_data_per_packet;
          if (RAY_SIZE_IN_BYTES % actual_raw_data_per_packet)
                    total_packets_required++;

          // Total time required to send the all the data of the ray with local and file
ray header is computed.
          // File ray header is included in the every packet at present.
          time_for_one_ray     = (RAY_SIZE_IN_BITS)*1000.0/MAX_RATE_SUPPORTED_IN_BITS;
          total_active_clients = 0;
          size_of_schedule     = (int)ceilf(time_for_one_ray/CLOCK_TICK_RESOLUTION);

          /*************************************************************/
          size_of_schedule     = 2; // Temporary solution, remove it after
          /*************************************************************/;

          temp_main_data_len   = (RAY_SIZE_IN_BYTES)/(size_of_schedule);    //Number of
Bytes sent in Each Tick, Only consider raw data only
          main_data_len        = temp_main_data_len - temp_main_data_len % 4;
                                  // Word Boundry
          last_data_len        = (size_of_schedule)*(temp_main_data_len%4) +
main_data_len+(RAY_SIZE_IN_BYTES)%(size_of_schedule);
          if (size_of_schedule>sizeof(int)*8)
                    {
                    printf("This MAX RATE cannot be supported , select higher value\n");
                    exit(1);
                    }

          for (i=0;i<NUM_MAX_RATES;i++)
                    {
                    client_count[i]=0;
                    for (j=0;j<MAX_CLIENT_IN_WINDOW;j++)
                              {
                              schedule_to_client_map_table[i][j] = MAX_CLIENTS; //
Initialize all the schedule rows of the LRT table.
                              }
                    RT_table[i].base_schedule_ticks  = 0;
                    RT_table[i].base_rate            = 0;
                    RT_table[i].client_list_type_1   = NULL;
                    RT_table[i].client_list_type_2   = NULL;

                    RT_table_second[i].base_schedule_ticks  = 0;
                    RT_table_second[i].base_rate            = 0;
                    RT_table_second[i].client_list_type_1   = NULL;
                    RT_table_second[i].client_list_type_2   = NULL;
                    }

          primary_table = 1;
          // Initialize the Send Data Packet used for packet transmission.
          memset(&send_data_packet,0,sizeof(send_data_packet));
          memset(window_load, 0, sizeof(window_load));
          current_ray_num              = 0; //Set no of rays to zero
          client_list_table_head_ptr   = NULL; // Set client list table head ptr to NULL as
there are no ptrs.
          client_list_table_count      = 0;            // Set number of active client nodes
in client link list to zero
          last_client_ptr = NULL;
          client_schedule_index = 0;
}

// Make RT Table Entries, it should be the rates supported.
void set_rt_table()
{
          int i,decrease_value=0;
          unsigned int  schedule = 0xFFFFFFFF;
          if (MAX_RATE_SUPPORTED%size_of_schedule>size_of_schedule/2)
                    resolution   =
ceilf((float)MAX_RATE_SUPPORTED/(float)size_of_schedule);
          else
                    resolution   =
floorf((float)MAX_RATE_SUPPORTED/(float)size_of_schedule);

          schedule = schedule<< 8*sizeof(schedule) - size_of_schedule;
          for (i=0;i<NUM_MAX_RATES;i++)
                    {
                    // Note that number of bits set to 1 in schedule approximate desired
base rate
                    RT_table[i].base_rate          = MAX_RATE_SUPPORTED- i;
                    if (i>0 && i%resolution==0)
                              decrease_value++;
                    RT_table[i].base_schedule_ticks = size_of_schedule - decrease_value;
                    // Store number of Windows in which data should be sent.
                    MIN_RATE_SUPPORTED             = RT_table[i].base_rate;
                    }
}

int next_free_client_index = 0;
```

```
/*
  First time client make a REQUEST for data to the server, entry is made in client table and
schedules are assigned.
  This function to add new client request is always added in TICK 0.
*/

void add_clients_request(uint32_t client_ip,uint16_t client_port, int target_rate, int
min_rate, int data_type_requirement)
{
          int i;
          struct _client_list_ *temp_ptr;
          if (target_rate>MAX_RATE_SUPPORTED)
                    {
                    printf("Target Rate is more than the Max Rate supported..do set target
to upper bound only ");
                    target_rate = MAX_RATE_SUPPORTED;
                    }

          if (min_rate < MIN_RATE_SUPPORTED)
                    {
                    printf("Requested Min Rate is below MIN SUPPORTED\n");
                    min_rate = MIN_RATE_SUPPORTED;
                    }

          if (total_active_clients < MAX_CLIENTS)
                    {
                    int target_idx,ticks_remaining,num_active_slots;
                    target_idx  = (int)( (double)(MAX_RATE_SUPPORTED -
target_rate)/(double)RATE_INCREMENT);
                    num_active_slots = RT_table[target_idx].base_schedule_ticks;  //
Number of Slots in which data is transmitted while maintaining target rate
                    //Add node for the client in the link list
                    temp_ptr = (struct _client_list_ *)malloc(sizeof(struct
_client_list_));
                    temp_ptr->target_rate_idx    = target_idx;
                    temp_ptr->min_rate_idx       = (int)((double)(MAX_RATE_SUPPORTED -
min_rate)/(double)RATE_INCREMENT);
                    temp_ptr->curr_rate_idx      = target_idx;
                    temp_ptr->client_ip          = client_ip;
                    temp_ptr->client_port        = client_port;
                    memset(temp_ptr->packet_sent_count, 0,sizeof(temp_ptr-
>packet_sent_count));;             // Nothing has been sent to new client, so set it to 0
                    temp_ptr->ticks_remaining = RT_table[target_idx].base_schedule_ticks;

                    /*
                     Since request can come any time in any tick, so first time send data
in windows from the current+1 to the last one only.
                     This approach will make sure that at the end of 1 cycle of ticks all
the clients remaining ticks will be 0 at the end.
                     we don't want to carry any remainder ticks from the previous cycle to
the current cycle, that would break the initializtion
                     of ticks logic otherwise.
                    */

                    // New client node has been allocated, now find the location where it
should be inserted, At present it is always the first element in the list.
                    switch (data_type_requirement)
                              {
                              case DATA_TYPE_1:
                                        {
                                                  if (primary_table==1)
                                                            {
                                                            if (
RT_table[target_idx].client_list_type_1==NULL)
                                                                      {
                                                                      temp_ptr->next
= NULL;

          RT_table[target_idx].client_list_type_1 = temp_ptr;
                                                                                        // First client
entry for the requested rate
                                                                      }
                                                            else
                                                                      {
                                                                                        // Already
clients are present in the list, make this new client the first element of the list
                                                                      temp_ptr->next =
RT_table[target_idx].client_list_type_1;

          RT_table[target_idx].client_list_type_1= temp_ptr;
                                                                      }
                                                            }
                                                  else
                                                            {
                                                            if
(RT_table_second[target_idx].client_list_type_1==NULL)
                                                                      {
                                                                      temp_ptr->next
= NULL;

          RT_table_second[target_idx].client_list_type_1 = temp_ptr;
                                                                                        // First client
entry for the requested rate
                                                                      }
                                                            else
                                                                      {
                                                                                        // Already
clients are present in the list, make this new client the first element of the list
                                                                      temp_ptr->next =
RT_table_second[target_idx].client_list_type_1;

          RT_table_second[target_idx].client_list_type_1= temp_ptr;
                                                                      }
                                                            }
                                                  break;
                                        }
                              case DATA_TYPE_2:
                                        {
                                                  if (primary_table==1)
                                                            {
                                                            if (
RT_table[target_idx].client_list_type_2==NULL)
                                                                      {
                                                                      //printf("Client
list is NMULL ENTRY made for target idx %d in FIRST TABLE\n",target_idx);
                                                                      temp_ptr->next
= NULL;

          RT_table[target_idx].client_list_type_2 = temp_ptr;
                                                                                        // First client
entry for the requested rate
                                                                      }
                                                            else
                                                                      {
                                                                                        // Already
clients are present in the list, make this new client the first element of the list
                                                                      temp_ptr->next =
RT_table[target_idx].client_list_type_2;

          RT_table[target_idx].client_list_type_2= temp_ptr;
                                                                      }
                                                            }
                                                  else
                                                            {
                                                            if
(RT_table_second[target_idx].client_list_type_2==NULL)
                                                                      {
```

```
                                                                        temp_ptr->next
= NULL;

          RT_table_second[target_idx].client_list_type_2 = temp_ptr;
                                                                        // First client
entry for the requested rate
                                                                        }
                                                              else
                                                                        {
                                                                        // Already
clients are present in the list, make this new client the first element of the list
                                                                        temp_ptr->next =
RT_table_second[target_idx].client_list_type_2;

          RT_table_second[target_idx].client_list_type_2= temp_ptr;
                                                                        }
                                                                  }
                                                        break;
                                              }
                                    default: break;
                                    }
                          total_active_clients++;           // Increment the Total active
clients, it should be decremented once client is removed.
                          }
          else
                          printf(" Maximum Client limit reached %d \n", total_active_clients);
}


// Change Client Schedule
void change_client_schedule(int current_rate_idx, int target_rate_idx, int client_ip, int
client_port,short int packet_recv_count,short int ray_num, int remove_flag,int
data_type_requirement)
{
          int client_loss, current_rate,i;
          int new_schedule_idx;
          struct _client_list_ *temp_client_ptr,*prev_temp_client_ptr;

          switch (data_type_requirement)
                    {
                    case DATA_TYPE_1:
                              {
                                        if (primary_table==1)
                                                  temp_client_ptr =
RT_table[current_rate_idx].client_list_type_1;
                                        else
                                                  temp_client_ptr       =
RT_table_second[current_rate_idx].client_list_type_1;
                                        if (primary_table==1)
                                                  for (i=0;i<NUM_MAX_RATES;i++)
                                                            {
                                                            if
(RT_table[i].client_list_type_1!=NULL)
                                                                      printf("Found a
Entry in RT table for index i=%d\n",i);
                                                            }
                                        else
                                                  for (i=0;i<NUM_MAX_RATES;i++)
                                                            {
                                                            if
(RT_table_second[i].client_list_type_1!=NULL)
                                                                      printf("Found a
entry in RT TABLE SECOND for index %d\n",i);
                                                            }
                                        break;
                              }
                    case DATA_TYPE_2:
                              {
                                        if (primary_table==1)
                                                  temp_client_ptr =
RT_table[current_rate_idx].client_list_type_2;
                                        else
                                                  temp_client_ptr       =
RT_table_second[current_rate_idx].client_list_type_2;
                                        if (primary_table==1)
                                                  for (i=0;i<NUM_MAX_RATES;i++)
                                                            {
                                                            if
(RT_table[i].client_list_type_2!=NULL)
                                                                      printf("Found a
Entry in RT table for index i=%d\n",i);
                                                            }
                                        else
                                                  for (i=0;i<NUM_MAX_RATES;i++)
                                                            {
                                                            if
(RT_table_second[i].client_list_type_2!=NULL)
                                                                      printf("Found a
entry in RT TABLE SECOND for index %d\n",i);
                                                            }
                                        break;
                              }
                    default: break;
                    }
          if (temp_client_ptr==NULL)
                    printf("TEMP CLIENT PTR IS NULL...!!!\n");
          prev_temp_client_ptr=temp_client_ptr;
          while (temp_client_ptr!=NULL)
                    {
                    if (temp_client_ptr->client_ip==client_ip  && temp_client_ptr-
>client_port==client_port)
                              {
                              break;
                              }
                    else
                              {
                              prev_temp_client_ptr = temp_client_ptr;
                              temp_client_ptr=temp_client_ptr->next;
                              }
                    }
          if (temp_client_ptr==NULL)
                    {
                    uint i;
                    for (i=0;i<NUM_MAX_RATES;i++)
                              {
                              current_rate_idx= (current_rate_idx+1)%NUM_MAX_RATES;

                              if (data_type_requirement==DATA_TYPE_1)
                                        {
                                        if (primary_table==1)
                                                  temp_client_ptr =
RT_table[current_rate_idx].client_list_type_1;
                                        else
                                                  temp_client_ptr       =
RT_table_second[current_rate_idx].client_list_type_1;
                                        }
                              else if (data_type_requirement==DATA_TYPE_2)
                                        {
                                        if (primary_table==1)
                                                  temp_client_ptr =
RT_table[current_rate_idx].client_list_type_2;
                                        else
                                                  temp_client_ptr       =
RT_table_second[current_rate_idx].client_list_type_2;
                                        }
                              prev_temp_client_ptr=temp_client_ptr;
                              while (temp_client_ptr!=NULL)
                                        {
```

```
                                        if (temp_client_ptr->client_ip==client_ip  &&
temp_client_ptr->client_port==client_port)
                                                {
                                                break;
                                                }
                                        else
                                                {
                                                prev_temp_client_ptr =
temp_client_ptr;
                                                temp_client_ptr=temp_client_ptr->next;
                                                }
                                        }
                                if (temp_client_ptr!=NULL)
                                        break;
                                }
                        if (temp_client_ptr==NULL)
                                {
                                printf("ERROR- Should not see this message, Client is not
present in the schedule list\n");
                                return;
                                }
                        }

                current_rate         = RT_table[current_rate_idx].base_rate;
                client_loss          = temp_client_ptr-
>packet_sent_count[ray_num%MAX_COUNT_HISTORY]-packet_recv_count;
                if (client_loss < 0)
                        {
                        printf("ERROR: This message should not been seen under normal
operation, otherwise try increasing the size of the HISTORY list of rays\n");
                        // Simply remove the client under such conditions
                        remove_client_node(temp_client_ptr,
prev_temp_client_ptr,data_type_requirement);
                        free(temp_client_ptr);
                        return;
                        }
                if (client_loss==0)
                        {
                        if
(RT_table[current_rate_idx].base_rate<RT_table[target_rate_idx].base_rate)
                                {
                                // Increment the rate
                                // Point to schedule of next higher rate. Top Entry in RT
table is highest rate
                                new_schedule_idx = current_rate_idx - 1;
                                }
                        else
                                {
                                new_schedule_idx = current_rate_idx;
                                //else don't change the rate yet as protocol is already
operating at target rate.
                                }
                        }
                else if (client_loss>0)
                        {
                        //Reduce the rate to minimum required rate when loss is suffered
                        new_schedule_idx    = temp_client_ptr->min_rate_idx;
                        }
                if (remove_flag)
                        {
                        // This flag is to inform this method to remove the client entry.
                        remove_client_node(temp_client_ptr,prev_temp_client_ptr,
data_type_requirement);
                        free(temp_client_ptr);
                        return ;
                        }

                if (new_schedule_idx!=current_rate_idx)
                        {
                        // Change only if Rate is changed.
                        // Always remove node from PRIMARY table

                remove_client_node(temp_client_ptr,prev_temp_client_ptr,data_type_requirement);
                        temp_client_ptr->curr_rate_idx   = new_schedule_idx;
                        // Always add entry to the SECONDARY table

                add_client_node(new_schedule_idx,temp_client_ptr,data_type_requirement);
                        }
                else
                        {
                        // In case there is no change in schedule due to this feedback
                        // Still it is desired to change the table from current primary to
current secondary table

                remove_client_node(temp_client_ptr,prev_temp_client_ptr,data_type_requirement);
                // Remove from PRIMARY

                add_client_node(new_schedule_idx,temp_client_ptr,data_type_requirement);
                // ADD to secondary

                        }
}

//Remove client node from the scheduling table when END request arrives
struct _client_list_ *remove_client_node(struct _client_list_ *temp_ptr, struct
_client_list_ *prev_temp_ptr, int data_type_requirement)
                {
                if (temp_ptr!=prev_temp_ptr)
                        {
                        prev_temp_ptr->next = temp_ptr->next;  // Remove the temp_ptr, let
prev node point to next node.
                        }
                else
                        {
                        // Both are equal when the first node is to be removed
                        if (data_type_requirement==DATA_TYPE_1)

                                if (primary_table==1)
                                        {
                                        RT_table[temp_ptr-
>curr_rate_idx].client_list_type_1 = temp_ptr->next;
                                        }
                                else
                                        {
                                        RT_table_second[temp_ptr-
>curr_rate_idx].client_list_type_1 = temp_ptr->next;
                                        }
                        else if (data_type_requirement==DATA_TYPE_2)
                                {
                                if (primary_table==1)
                                        {
                                        RT_table[temp_ptr-
>curr_rate_idx].client_list_type_2 = temp_ptr->next;
                                        }
                                else
                                        {
                                        RT_table_second[temp_ptr-
>curr_rate_idx].client_list_type_2 = temp_ptr->next;
                                        }
                                }
                        }
                total_active_clients --;
                }


// Add New Client when request is accepted
void add_client_node(int current_rate_idx, struct _client_list_ *client_ptr, int
data_type_requirement)
```

```
{
        int i, schedule_ticks;
        // Make it as the first node in the list
        // printf("Adding node with current rate idx %d\n", current_rate_idx);
        // Add entry to the Secondary Table

        if (data_type_requirement==DATA_TYPE_1)
                if (primary_table==1)
                        {
                        client_ptr->next =
RT_table_second[current_rate_idx].client_list_type_1;
                        RT_table_second[current_rate_idx].client_list_type_1 =
client_ptr;
                        }
                else
                        {
                        client_ptr->next =
RT_table[current_rate_idx].client_list_type_1;
                        RT_table[current_rate_idx].client_list_type_1 = client_ptr;
                        }
        else if (data_type_requirement==DATA_TYPE_2)
                {
                if (primary_table==1)
                        {
                        client_ptr->next =
RT_table_second[current_rate_idx].client_list_type_2;
                        RT_table_second[current_rate_idx].client_list_type_2 =
client_ptr;
                        }
                else
                        {
                        client_ptr->next =
RT_table[current_rate_idx].client_list_type_2;
                        RT_table[current_rate_idx].client_list_type_2 = client_ptr;
                        }

                }
        total_active_clients++;
        schedule_ticks = RT_table[current_rate_idx].base_schedule_ticks;

}


// Thread for scheduling timer
void time_ticker()
{
        int status;
        struct tm *ptr;
        time_t tm;
        status = timer_create(CLOCK_REALTIME,&timers.evp,&timer_id  );
        if (status==FAIL)
                {
                printf("Unable to Create Timer\n");
                exit(1);
                }
        sigaction(timers.evp.sigev_signo,&sig_act,0);
        sigaction(SIGINT, &sig_act,NULL);
        status = timer_settime(timer_id,timers.type,&timers.timeout,NULL);
        if (status==FAIL)
                {
                printf("Error Starting Timer\n");
                exit(1);
                }
        tm = time(NULL);
        for (;;)
                pause();
}
```

```
int timeout_count = 0;
int set =0;
struct timeval start_tp;
sig_handler(int signo)
{
        int rc;
        struct timeval tp;
        switch (signo)
                {
                case SIGUSR1:
                        tick_count =tick_count%size_of_schedule;
                        if (!set && total_active_clients==1)
                                {
                                rc = gettimeofday(&start_tp,NULL);

                                set = 1;
                                }
                        simulate_time_ticker(tick_count);
                        tick_count++;
                        break;
                case SIGINT:
                        exit(1);
                default:
                        break;
                }
}

// Free all dynamically allocated memory at exit time
void release_memory()
{
        free(release_file_buffer);
}

unsigned int base_tick=0x80000000;

/*
This function is called for every tick periodically.
*/
int last_ray_num=0;
void simulate_time_ticker(int tick)
{
        int i,k,l,data_sent=0;
        unsigned int current_tick;
        int rc;
        struct timeval tp;
        current_tick = base_tick>>tick;
        if (tick==0 && feedback_node_list_head_ptr!=NULL)
                {
                feedback_node_type *temp_ptr;
                temp_ptr = feedback_node_list_head_ptr;
                while (temp_ptr!=NULL)
                        {
                        change_client_schedule(temp_ptr->current_index,temp_ptr-
>target_index, temp_ptr->client_ip_address,temp_ptr->client_port,temp_ptr-
>packet_count,temp_ptr->ray_num,temp_ptr->flag, temp_ptr->data_type_requirement);
                        feedback_node_list_head_ptr = temp_ptr;
                        temp_ptr = temp_ptr->next;
                        free(feedback_node_list_head_ptr);
                        }
                feedback_node_list_head_ptr = NULL;

                for (i=0;i<NUM_MAX_RATES;i++)
                        {
                        int temp_index;
                        if (primary_table==1)
                                {
                                struct _client_list_ *client_ptr_no_feedback;
                                // Primary Table is 2 so no node should be
present int he FIRST table, if any it is because no feedback was received
```

```
                                        // for that node, so pretend that NO loss was
suffered by that client, so remove that node from first and transfer it
                                        // to SECOND table

                                        // Check Client in list 1 for no feeback
                                        client_ptr_no_feedback =
RT_table[i].client_list_type_1;
                                        while (client_ptr_no_feedback!=NULL)
                                                {

          change_client_schedule(client_ptr_no_feedback-
>curr_rate_idx,client_ptr_no_feedback->target_rate_idx, client_ptr_no_feedback->client_ip,
client_ptr_no_feedback->client_port, client_ptr_no_feedback-
>packet_sent_count[current_ray_num%MAX_COUNT_HISTORY],current_ray_num, 0,DATA_TYPE_1);
                                                client_ptr_no_feedback =
client_ptr_no_feedback->next;
                                                }
                                        // Check clients in list 2 for no feedback
                                        client_ptr_no_feedback =
RT_table[i].client_list_type_2;
                                        while (client_ptr_no_feedback!=NULL)
                                                {

          change_client_schedule(client_ptr_no_feedback-
>curr_rate_idx,client_ptr_no_feedback->target_rate_idx, client_ptr_no_feedback->client_ip,
client_ptr_no_feedback->client_port, client_ptr_no_feedback-
>packet_sent_count[current_ray_num%MAX_COUNT_HISTORY],current_ray_num, 0,DATA_TYPE_2);
                                                client_ptr_no_feedback =
client_ptr_no_feedback->next;
                                                }
                                        }
                                else
                                        { // Move from SECOND table to FIRST table as
primary is FIRST now
                                        struct _client_list_ *client_ptr_no_feedback;
                                        client_ptr_no_feedback =
RT_table_second[i].client_list_type_1;
                                        while (client_ptr_no_feedback!=NULL)
                                                {

          change_client_schedule(client_ptr_no_feedback-
>curr_rate_idx,client_ptr_no_feedback->target_rate_idx, client_ptr_no_feedback->client_ip,
client_ptr_no_feedback->client_port, client_ptr_no_feedback-
>packet_sent_count[current_ray_num%MAX_COUNT_HISTORY],current_ray_num, 0,DATA_TYPE_1);
                                                client_ptr_no_feedback =
client_ptr_no_feedback->next;
                                                }
                                        client_ptr_no_feedback =
RT_table_second[i].client_list_type_2;
                                        while (client_ptr_no_feedback!=NULL)
                                                {

          change_client_schedule(client_ptr_no_feedback-
>curr_rate_idx,client_ptr_no_feedback->target_rate_idx, client_ptr_no_feedback->client_ip,
client_ptr_no_feedback->client_port, client_ptr_no_feedback-
>packet_sent_count[current_ray_num%MAX_COUNT_HISTORY],current_ray_num, 0,DATA_TYPE_2);
                                                client_ptr_no_feedback =
client_ptr_no_feedback->next;
                                                }
                                        }
                                }
                        if (primary_table==1)
                                primary_table = 2;
                        else
                                primary_table = 1;
                        }
                if (tick==0 && request_node_list_head_ptr!=NULL)
                        {
                        request_node_type *temp_ptr;
                        temp_ptr = request_node_list_head_ptr;
                        while (temp_ptr!=NULL)
                                {
                                add_clients_request(temp_ptr->client_ip_address, temp_ptr-
>client_port,temp_ptr->target_rate,temp_ptr->min_rate,temp_ptr->data_type_requirement);
                                request_node_list_head_ptr=temp_ptr;
                                temp_ptr = temp_ptr->next;
                                free(request_node_list_head_ptr);          // Release the
memory allocated for the client node as its request has been added
                                }
                        request_node_list_head_ptr=NULL; // By this time all client request are
added thus head ptr is NULL
                        }

                /*
                  PRIMARY table has node of all the active clients corresponding to different
rate requirements of the
                    clients. Now next step should be to create a scheduling table for this list.
                */
                if (total_active_clients>0)
                        {
                        //  Atleast One Client is Active
                        struct _client_list_ *temp_ptr;
                        int current_tick_load;
                        if (tick==0)
                                {
                                // Read ray header from a file
                                send_data_packet.local_header.type =htons(DATA); // Set
Type that packet is carrying data.
                                current_ray_num++;
                                next_seq_number=1;                  // For every new ray, set
the sequence number of 1st packet to 1
                                if (current_ray_num>1)
                                        file_buffer +=RAY_SIZE_IN_BYTES;
                                send_data_packet.local_header.ray_num =
htons(current_ray_num);            // Increment Ray No at every 0 tick, ray transmission
starts at 0th tick only
                                // Set it to 0, as first packet of ray is yet to be sent
                                // Since File buffer is in BIG ENDIEN, therfore no need to
apply HTONL at present.
                                memcpy(&send_data_packet.file_header,
file_buffer,sizeof(file_ray_header_type)); // Read Header from the file and use it for
                                // All subsequent packets of the ray
                                file_buffer += sizeof(file_ray_header_type);
                                ray_data_offset = 0;
                                }

                        /*
                        Traverser RT_table or RT_table_second and see which rate has client
scheduled
                        Then depending on the tick, follow the sample schedule, packetize and
send data

                        */
                        if (primary_table==1)
                                {
                                int num_samples_per_time_slot =
(int)ceilf((double)NUM_SAMPLES/(double)size_of_schedule);
                                struct _client_list_ *temp_client_ptr;
                                // Traverse RT_table
                                for (i=0;i<NUM_MAX_RATES;i++)
                                        {
                                        temp_client_ptr = RT_table[i].client_list_type_1;
                                        if (temp_client_ptr!=NULL)
                                                {
                                                // Client is present for that
particular rate
                                                // Send data as per the scheduled for
the tick at that particular rate
```

```
                                                int k=0;
                                                /*
                                                Sample number is the first sample of
the group, so using sample group requirement, we know what adjacent samples should be sent.
At this time give control to the sending routine, that should do the packetization depending
on the number of samples to be transmitted within the time slot, so send reference to rate
index and tick and client address and it should finish the task of sending and return the
control back here. send data sends data to all the clients that are scheduled to get data at
the given rate
                                                */

          send_data_type_1(server_send_sock,tick,i,DATA_TYPE_1);
                                                }
                                        temp_client_ptr = RT_table[i].client_list_type_2;
                                        if (temp_client_ptr!=NULL)
                                                {
                                                // There are clients present for the
second data type request, so send data to them

          send_data_type_2(server_send_sock,tick,i,DATA_TYPE_2);
                                                }
                                        }
                                }
                        else
                                {
                                // Traverse RT_table_second
                                int num_samples_per_time_slot =
(int)ceilf((double)NUM_SAMPLES/(double)size_of_schedule);
                                struct _client_list_ *temp_client_ptr;
                                // Traverse RT_table
                                for (i=0;i<NUM_MAX_RATES;i++)
                                        {
                                        temp_client_ptr =
RT_table_second[i].client_list_type_1;
                                        if (temp_client_ptr!=NULL)
                                                {
                                                // Client is present for that
particular rate
                                                // Send data as per the scheduled for
the tick at that particular rate
                                                int k=0;
                                                /*
                                                Sample number is the first sample of
the group, so using sample group requirement, we know what adjacent samples should be sent.
At this time give control to the sending routine, that should do the packetization depending
on the number of samples to be transmitted within the time slot, so send reference to rate
index and tick and client address and it should finish the task of sending and return the
control back here.
                                                */

          send_data_type_1(server_send_sock,tick,i,DATA_TYPE_1);
                                                }
                                        temp_client_ptr =
RT_table_second[i].client_list_type_2;
                                        if (temp_client_ptr!=NULL)
                                                {

          send_data_type_2(server_send_sock,tick,i,DATA_TYPE_2);
                                                }
                                        }
                                }
                        } // if total active clients >0


          if (current_ray_num>0 &&
current_ray_num%NUM_RAYS==0&&last_ray_num!=current_ray_num &&tick==size_of_schedule-1)
                        {
                        last_ray_num=current_ray_num;
                        sweep_count ++;                 //Increment the sweep count
                        file_buffer= release_file_buffer;// Retransmit the same buffer again
and again
                        total_data_transmitted = 0;     // Before the start of any sweep set
this byte count to 0
                        rc = gettimeofday(&tp,NULL);
                        if (sweep_count==MAX_SWEEP)
                                {
                                // Release file buffer once all the data for all the sweeps
is transmitted.
                                release_memory();
                                exit(1);
                                }
                        }
          current_tick = base_tick>>tick;
}

/*
 This function gets called for each client
*/
void send_data_type_1(int server_send_sock, int tick, int rate_index, int
sample_group_requirement)
{
          struct in_addr *inptr;
          struct _client_list_ *client_ptr;
          struct sockaddr_in client_addr;
          char data_ptr[500];
          int num_of_complete_packet, last_packet_size,i;
          int rc;
          struct timeval tp;
          int sample_group_first_sample, num_gates_to_transmit,
packet_buffer_offset,payload_size,total_data_to_send_in_tick,data_transmitted,
num_max_gates, sample_index, last_sample_pending_gates,packet_sent;
          int first_gate, last_gate,first_sample_in_packet;
          uint16_t base_pattern;
          inptr = (struct in_addr *) malloc(sizeof(struct in_addr));
          /*
           Divide Data to be transmitted among different packets.
           Set the current rate in the send packet, this information is included in the
packet every time feedback about loss is  sent by client it can happen that all packets
containing the new current rate gets lost, in that case client won't be aware of current
rate thus they will have stale value of current rate, server should be able to handle that
case.
           */

          send_data_packet.local_header.current_rate =
htonl(RT_table[rate_index].base_rate);

          /*
           Next step is to copy the sample data from the input buffer to the packet payload
           Data should be copied considering sample group requirement
           Sample schedule is already there, associated with each rate.
          */
          total_data_to_send_in_tick =
RT_table[rate_index].time_slot_data_size_type_1[tick];
          data_transmitted =0;

          //Number of gates that can be transmitted while satisfying data group requirement
in the same packet.
          num_max_gates    =
actual_raw_data_per_packet/((int)SAMPLE_SIZE*(int)sample_group_requirement);
          //Iterate following loop till all the data for a given tick is transmitted to a
particular client
          last_sample_pending_gates=0;
          base_pattern = 0x8000;          // 16 bit mask, used to set the appropriate bit
in the sample pattern in each packet
          sample_index = 0;

          while (data_transmitted<total_data_to_send_in_tick)
                  {
```

```
                    // Get how many gates for the sample to be transmitted, make sure that
it is taken care here
                    // Also get the reference to the first sample of the group in the
buffer.
                    packet_sent  = 0;
                    payload_size = 0;
                    packet_buffer_offset=0;

                    send_data_packet.local_header.payload_size   = 0;
                    send_data_packet.local_header.sample_pattern = 0;
                    num_max_gates =
actual_raw_data_per_packet/(SAMPLE_SIZE*sample_group_requirement);
                    while (!packet_sent)
                              {
                              sample_group_first_sample =
RT_table[rate_index].sample_table_ptr_type_1[tick][sample_index].sample_number;
                              if (sample_group_first_sample==0)
                                        {
                                        printf("Control should not come here, because if
proper number of bytes are transmitted then while loop should exit\n");
                                        break;
                                        }
                              else
                                        {
                                        // Determine if some gates data have already been
transmitted in previous packet
                                        if (last_sample_pending_gates==0) // Initiate
sample transfer from this new packet
                                                  {
                                                  num_gates_to_transmit  =
RT_table[rate_index].sample_table_ptr_type_1[tick][sample_index].total_gates; // All gates
to be sent
                                                  first_gate =
RT_table[rate_index].sample_table_ptr_type_1[tick][sample_index].first_gate;
                                                  last_gate  =
RT_table[rate_index].sample_table_ptr_type_1[tick][sample_index].last_gate;
                                                  }
                                        else
                                                  {
                                                  num_gates_to_transmit =
last_sample_pending_gates;
                                                  first_gate =
RT_table[rate_index].sample_table_ptr_type_1[tick][sample_index].first_gate+RT_table[rate_in
dex].sample_table_ptr_type_1[tick][sample_index].total_gates-last_sample_pending_gates;
                                                  last_gate =
RT_table[rate_index].sample_table_ptr_type_1[tick][sample_index].last_gate;
                                                  }

                                        // When first sample is being sent, store first
sample number and first gate information in the header field
                                        if (packet_buffer_offset==0)
                                                  {
                                                  first_sample_in_packet =
sample_group_first_sample;

                    send_data_packet.local_header.start_sample           =
htons(first_sample_in_packet);

                    send_data_packet.local_header.start_sample_first_gate = htons(first_gate);
                                                  send_data_packet.local_header.seq_num
= htonl(next_seq_number++);
                                                  }
                                        }

                              // Get reference in the main buffer
                              file_buffer_ptr = file_buffer + (sample_group_first_sample-
1)*NUM_GATES*SAMPLE_SIZE;
                              file_buffer_ptr = file_buffer_ptr + (first_gate-
1)*SAMPLE_SIZE;       // This points to the first gate to be transmitted
                                        if (num_gates_to_transmit >= num_max_gates) // when number
of gates to transmit is more than that can be included in the same packet
                                                  {
                                                  // more gates than same packet can include, thus
include whatever can be sent in the same packet and then send remaining in the next packet,
remember
                                                  for (i=0;i<sample_group_requirement;i++)
                                                            {

                    memcpy(send_data_packet.payload+packet_buffer_offset,file_buffer_ptr,SAMPLE_SIZE*
num_max_gates);
                                                  data_transmitted
+=SAMPLE_SIZE*num_max_gates;
                                                  packet_buffer_offset
+=SAMPLE_SIZE*num_max_gates;
                                                  file_buffer_ptr = file_buffer_ptr +
NUM_GATES*SAMPLE_SIZE;

                    send_data_packet.local_header.sample_pattern
|=base_pattern>>(sample_group_first_sample+i - first_sample_in_packet);
                                                  payload_size
+=SAMPLE_SIZE*num_max_gates;
                                                  }
                                                  // Last sample gates has been included in the
packet, so store the first sample of last group transmitted and also gates as well.
                                                  send_data_packet.local_header.end_sample =
htons(sample_group_first_sample);
                                                  if(send_data_packet.local_header.start_sample!=
                                        send_data_packet.local_header.end_sample)
                                        // already in network byte order
                                                  {
                                                  // Then fill the remaining fields of
the header

                    send_data_packet.local_header.start_sample_last_gate = htons(NUM_GATES);       //
last gate is same as total number of gates per ray

                    send_data_packet.local_header.end_sample_first_gate = htons(1);                //
1st gate is always sent here

                    send_data_packet.local_header.end_sample_last_gate  = htons(num_max_gates);
                                                  }
                                                  else
                                                  {
                                                  // Both first sample and last sample
are same , this will happen when packet sizes are small compared to ray size

                    send_data_packet.local_header.end_sample_first_gate =
send_data_packet.local_header.start_sample_first_gate;

                    send_data_packet.local_header.start_sample_last_gate =
htons(ntohs(send_data_packet.local_header.start_sample_first_gate)+num_max_gates-1);

                    send_data_packet.local_header.end_sample_last_gate  =
send_data_packet.local_header.start_sample_last_gate;
                                                  }
                                                  // Remember how many more gates to be transmitted
for this sample in the next packet,
                                                  // so do not jump to next sample group, first
transmit pending gates.
                                                  // Fill remaining fields of the header here and
then send the packet
                                                  last_sample_pending_gates = num_gates_to_transmit
- num_max_gates;
                                                  num_gates_to_transmit     =
last_sample_pending_gates;
                                                  if (last_sample_pending_gates==0)
                                                            sample_index++;
```

```
                                        send_data_packet.local_header.sample_pattern =
htons(send_data_packet.local_header.sample_pattern);
                                        send_data_packet.local_header.payload_size   =
htonl(payload_size);

                                        // At this place, send the packet to all clients
scheduled for this rate.
                                        if (primary_table==1)
                                                client_ptr =
RT_table[rate_index].client_list_type_1;
                                        else
                                                client_ptr =
RT_table_second[rate_index].client_list_type_1;
                                        while (client_ptr!=NULL)
                                                {
                                                bzero((char
*)&client_addr,sizeof(client_addr));
                                                client_addr.sin_family      = AF_INET;
                                                inptr->s_addr               =
htonl(client_ptr->client_ip);
                                                bcopy((char *)inptr,(char
*)&client_addr.sin_addr,sizeof(struct in_addr));
                                                client_addr.sin_port        =
htons(client_ptr->client_port);
                                                if (next_seq_number==1)
                                                        {
                                                        /*
                                                        First Packet of Ray being
sent, so at this time initial count of ray packet sent in the history list can be set to 0.
                                                        */
                                                        client_ptr-
>packet_sent_count[current_ray_num%MAX_COUNT_HISTORY]=0;
                                                        }
                                                client_ptr-
>packet_sent_count[ntohs(send_data_packet.local_header.ray_num)%MAX_COUNT_HISTORY]+=1;
                                                if
((sendto(server_send_sock,&send_data_packet,payload_size+sizeof(local_header_type)+sizeof(fi
le_ray_header_type), 0,
          (struct sockaddr *)&client_addr,sizeof(struct sockaddr_in))) < 0)
                                                        {
                                                        perror("RADAR Server: Error
in  sending DATA");
                                                        exit(0);
                                                        }
                                                client_ptr = client_ptr->next;
                                                }
                                        packet_sent = 1;
                                        }
                                else
                                        {
                                        /* There are two possibilities, either packet
size is big enough to include all the information or Secondly there is no more information
left to include in the packet, so in that case fill all remaining header fields and send
packet.
Some intermediate samples being filled, packet is not full yet, so at this time only
possibly last gate information of first sample may be filled */
                                                if
(ntohs(send_data_packet.local_header.start_sample)==sample_group_first_sample)
                                                {
                                                // First sample group can be included
in the packet and there is still space left for more.

          send_data_packet.local_header.start_sample_last_gate = htons(NUM_GATES);
// Last sample is same as total number of gates.
                                                //Remaining field should be filled
when packet is filled and sent.
                                                }
                                        for (i=0;i<sample_group_requirement;i++)
                                                {
                                                // Number of gates to be transmitted
are less than what can go into single packet.

          memcpy(send_data_packet.payload+packet_buffer_offset,file_buffer_ptr,SAMPLE_SIZE*
num_gates_to_transmit);
                                                data_transmitted +=
SAMPLE_SIZE*num_gates_to_transmit;

          packet_buffer_offset+=SAMPLE_SIZE*num_gates_to_transmit;
                                                file_buffer_ptr   = file_buffer_ptr +
NUM_GATES*SAMPLE_SIZE;

          send_data_packet.local_header.sample_pattern
|=base_pattern>>(sample_group_first_sample+i - first_sample_in_packet);

          payload_size+=SAMPLE_SIZE*num_gates_to_transmit;
                                                }
                                        // These are the gates that can be transmitted
for next sample
                                        num_max_gates -=num_gates_to_transmit;
                                        // Since packet is not full, so check next group
present
                                        last_sample_pending_gates=0;     // All required
gates included in the same packet for the last sample
                                        sample_index++;
                                // Sample group is completely sent, so go to next sample group in the
tick.
                                        if (data_transmitted==total_data_to_send_in_tick)
                                                {
                                                // There is no more data to include in
this packet in the current tick
                                                // so fill the remaining fields of the
header and send this packet

          send_data_packet.local_header.end_sample = htons(sample_group_first_sample);
                                                if
(send_data_packet.local_header.start_sample!=send_data_packet.local_header.end_sample)
                                                        {
                                                        // Then fill the remaining
fields of the header

          send_data_packet.local_header.start_sample_last_gate = htons(NUM_GATES); // last
gate is same as total number of gates per ray

          send_data_packet.local_header.end_sample_first_gate  = htons(1);               //
1st gate is always sent here

          send_data_packet.local_header.end_sample_last_gate   = htons(num_max_gates);// I
think this is correct
                                                        }
                                                else
                                                        {
                                                        // Both first sample and
last sample are same, this will happen when packet sizes are small compared to ray size

          send_data_packet.local_header.end_sample_first_gate =
send_data_packet.local_header.start_sample_first_gate;

          send_data_packet.local_header.start_sample_last_gate =
htons(ntohs(send_data_packet.local_header.start_sample_first_gate)+num_gates_to_transmit-1);

          send_data_packet.local_header.end_sample_last_gate  =
send_data_packet.local_header.start_sample_last_gate;
                                                        }

          send_data_packet.local_header.sample_pattern =
htons(send_data_packet.local_header.sample_pattern);
```

```
        send_data_packet.local_header.payload_size   = htonl(payload_size);
                                                    if (primary_table==1)
                                                            client_ptr =
RT_table[rate_index].client_list_type_1;
                                                    else
                                                            client_ptr =
RT_table_second[rate_index].client_list_type_1;
                                                    while (client_ptr!=NULL)
                                                            {
                                                            bzero((char
*)&client_addr,sizeof(client_addr));
                                                            client_addr.sin_family
= AF_INET;
                                                            inptr->s_addr
= htonl(client_ptr->client_ip);
                                                            bcopy((char *)inptr,(char
*)&client_addr.sin_addr,sizeof(struct in_addr));
                                                            client_addr.sin_port
= htons(client_ptr->client_port);
                                                            if (next_seq_number==1)
                                                                    {
                                                                    /*
                                                                    First Packet of
Ray being sent, so at this time initial count of ray packet sent in the history list can be
set to 0.
                                                                    */
                                                                    client_ptr-
>packet_sent_count[current_ray_num%MAX_COUNT_HISTORY]=0;
                                                                    }
                                                            client_ptr-
>packet_sent_count[ntohs(send_data_packet.local_header.ray_num)%MAX_COUNT_HISTORY]+=1;
                                                            if
((sendto(server_send_sock,&send_data_packet,payload_size+sizeof(local_header_type)+sizeof(fi
le_ray_header_type), 0, (struct sockaddr *)&client_addr,sizeof(struct sockaddr_in))) < 0)
                                                                    {
                                                                    perror("RADAR
Server: Error in  sending DATA");
                                                                    exit(0);
                                                                    }
                                                            client_ptr = client_ptr-
>next;
                                                            }
                                                    packet_sent=1;
                                                    } // when data transmitted == total
data to send
                                            } // else
                                    } // while packet not sent
                            /* It is possible that all the gates of a packet were not included in
the same packet thus it needs to be taken care in next packet.*/
                            } // While all data for a tick is not sent
}


// Send GROUP PAIR Data
void send_data_type_2(int server_send_sock, int tick, int rate_index, int
sample_group_requirement)
{
        struct in_addr *inptr;
        struct _client_list_ *client_ptr;
        struct sockaddr_in client_addr;
        char data_ptr[500];
        int num_of_complete_packet, last_packet_size,i;
        int rc;
        struct timeval tp;
        int sample_group_first_sample, num_gates_to_transmit,
packet_buffer_offset,payload_size,total_data_to_send_in_tick,data_transmitted,
num_max_gates, sample_index, last_sample_pending_gates,packet_sent;
        int first_gate, last_gate,first_sample_in_packet;
        uint16_t base_pattern;
        printf("ray num %d tick in send data type 2 is %d\n",current_ray_num, tick);
        inptr = (struct in_addr *) malloc(sizeof(struct in_addr));

        /*
          Divide Data to be transmitted among different packets. Set the current rate in
the send packet, this information is included in the packet everytime feedback about loss is
sent by client it can happen that all packets containing the new current rate gets lost, in
that case client won't be aware of current rate thus they will have stale value of current
rate, server should be able to handle that case.
        */
        send_data_packet.local_header.current_rate =
htonl(RT_table[rate_index].base_rate);

        /*
         Next step is to copy the sample data from the input buffer to the packet payload
         Data should be copied considering sample group requirement
         Sample schedule is already there, associated with each rate.
        */
        total_data_to_send_in_tick =
RT_table[rate_index].time_slot_data_size_type_2[tick];
        data_transmitted =0;

        //Number of gates that can be transmitted while satisfying data group requirement
in the same packet.
        num_max_gates    =
actual_raw_data_per_packet/((int)SAMPLE_SIZE*(int)sample_group_requirement);
        //Iterate following loop till all the data for a given tick is transmitted to a
particular client
        last_sample_pending_gates=0;
        base_pattern = 0x8000;          // 16 bit mask, used to set the appropriate bit
in the sample pattern in each packet
        sample_index=0;
        while (data_transmitted<total_data_to_send_in_tick)
                {
                // Get how many gates for the sample to be transmitted, make sure that
it is taken care here
                // Also get the reference to the first sample of the group in the
buffer.
                packet_sent=0;
                payload_size =0;
                packet_buffer_offset=0;
                send_data_packet.local_header.payload_size   = 0;
                send_data_packet.local_header.sample_pattern = 0;
                num_max_gates =
actual_raw_data_per_packet/(SAMPLE_SIZE*sample_group_requirement);
                while (!packet_sent)
                        {
                        sample_group_first_sample =
RT_table[rate_index].sample_table_ptr_type_2[tick][sample_index].sample_number;
                        if (sample_group_first_sample==0)
                                {
                                printf("Control should not come here, because if
proper number of bytes are transmitted then while loop should exit\n");
                                break;
                                }
                        else
                                {
                                // Determine if some gates data have already been
transmitted in previous packet
                                if (last_sample_pending_gates==0) // Initiate
sample transfer from this new packet
                                        {
                                        num_gates_to_transmit  =
RT_table[rate_index].sample_table_ptr_type_2[tick][sample_index].total_gates; // All gates
to be sent
                                        first_gate =
RT_table[rate_index].sample_table_ptr_type_2[tick][sample_index].first_gate;
```

```
                                                last_gate  =
RT_table[rate_index].sample_table_ptr_type_2[tick][sample_index].last_gate;
                                                }
                                        else
                                                {
                                                num_gates_to_transmit =
last_sample_pending_gates;
                                                first_gate =
RT_table[rate_index].sample_table_ptr_type_2[tick][sample_index].first_gate+RT_table[rate_in
dex].sample_table_ptr_type_2[tick][sample_index].total_gates-last_sample_pending_gates;
                                                last_gate =
RT_table[rate_index].sample_table_ptr_type_2[tick][sample_index].last_gate;
                                                }
                                        // When first sample is being sent, store first
sample number and first gate information in the header field
                                        if (packet_buffer_offset==0)
                                                {
                                                first_sample_in_packet =
sample_group_first_sample;

          send_data_packet.local_header.start_sample            =
htons(first_sample_in_packet);

          send_data_packet.local_header.start_sample_first_gate = htons(first_gate);
                                                send_data_packet.local_header.seq_num
= htonl(next_seq_number++);
                                                }
                                        }

                                // Get reference in the main buffer
                                file_buffer_ptr = file_buffer + (sample_group_first_sample-
1)*NUM_GATES*SAMPLE_SIZE;
                                file_buffer_ptr = file_buffer_ptr + (first_gate-
1)*SAMPLE_SIZE;       // This points to the first gate to be transmitted

                                if (num_gates_to_transmit >= num_max_gates) // when number
of gates to transmit is more than that can be included in the same packet
                                        {
                                        // more gates than same packet can include, thus
include whatever can be sent in the same packet and then send remaining in the next packet,
remember   */
                                        for (i=0;i<sample_group_requirement;i++)
                                                {

          memcpy(send_data_packet.payload+packet_buffer_offset,file_buffer_ptr,SAMPLE_SIZE*
num_max_gates);
                                                data_transmitted
+=SAMPLE_SIZE*num_max_gates;
                                                packet_buffer_offset
+=SAMPLE_SIZE*num_max_gates;
                                                file_buffer_ptr = file_buffer_ptr +
NUM_GATES*SAMPLE_SIZE;

          send_data_packet.local_header.sample_pattern
|=base_pattern>>(sample_group_first_sample+i - first_sample_in_packet);
                                                payload_size
+=SAMPLE_SIZE*num_max_gates;
                                                }
                                        // Last sample gates has been included in the
packet, so store the first sample of last group transmitted and also gates as well.
                                        send_data_packet.local_header.end_sample =
htons(sample_group_first_sample);
                                        if
(send_data_packet.local_header.start_sample!=send_data_packet.local_header.end_sample)//
already in network byte order
                                                {
                                                // Then fill the remining fields of
the header

          send_data_packet.local_header.start_sample_last_gate = htons(NUM_GATES);
// last gate is same as total number of gates per ray

          send_data_packet.local_header.end_sample_first_gate = htons(1);
// 1st gate is always sent here

          send_data_packet.local_header.end_sample_last_gate  = htons(num_max_gates);
                                                }
                                        else
                                                {
                                                // Both first sample and last sample
are same , this will happen when packet sizes are small compared to ray size

          send_data_packet.local_header.end_sample_first_gate =
send_data_packet.local_header.start_sample_first_gate;

          send_data_packet.local_header.start_sample_last_gate =
htons(ntohs(send_data_packet.local_header.start_sample_first_gate)+num_max_gates-1);

          send_data_packet.local_header.end_sample_last_gate  =
send_data_packet.local_header.start_sample_last_gate;
                                                }
                                        /*Remember how many more gates to be transmitted
for this sample in the next packet, so donot jump to next sample group, first transmit
pending gates. Fill remaining fields of the header here and then send the packet */
                                        last_sample_pending_gates = num_gates_to_transmit
- num_max_gates;
                                        num_gates_to_transmit     =
last_sample_pending_gates;
                                        if (last_sample_pending_gates==0)
                                                sample_index++;
                                        send_data_packet.local_header.sample_pattern =
htons(send_data_packet.local_header.sample_pattern);
                                        send_data_packet.local_header.payload_size    =
htonl(payload_size);

                                        // At this place, send the packet to all clients
scheduled for this rate.
                                        if (primary_table==1)
                                                client_ptr =
RT_table[rate_index].client_list_type_2;
                                        else
                                                client_ptr =
RT_table_second[rate_index].client_list_type_2;
                                        while (client_ptr!=NULL)
                                                {
                                                int temp_idx;
                                                char *temp_ptr;
                                                bzero((char
*)&client_addr,sizeof(client_addr));
                                                client_addr.sin_family      = AF_INET;
                                                inptr->s_addr               =
htonl(client_ptr->client_ip);
                                                bcopy((char *)inptr,(char
*)&client_addr.sin_addr,sizeof(struct in_addr));
                                                client_addr.sin_port        =
htons(client_ptr->client_port);
                                                if (next_seq_number==1)
                                                        {
                                                        /*
                                                        First Packet of Ray being
sent, so at this time initial count of ray packet sent in the history list can be set to 0.
                                                        */
                                                        client_ptr-
>packet_sent_count[current_ray_num%MAX_COUNT_HISTORY]=0;
                                                        }
                                                client_ptr-
>packet_sent_count[ntohs(send_data_packet.local_header.ray_num)%MAX_COUNT_HISTORY]+=1;
```

```
                                                temp_ptr = send_data_packet.payload;

                                                if
((sendto(server_send_sock,&send_data_packet,payload_size+sizeof(local_header_type)+sizeof(fi
le_ray_header_type), 0,


          (struct sockaddr *)&client_addr,sizeof(struct sockaddr_in))) < 0)
                                                        {
                                                        perror("RADAR Server: Error
in  sending DATA");
                                                        exit(0);
                                                        }

                                                client_ptr = client_ptr->next;
                                                }
                                        packet_sent = 1;
                                        }
                                else
                                        {
                                        /* There are two possibilities, either packet
size is big enough to include all the information or Secondly there is no more information
left to include in the packet, so in that case fill all remaining header fields and send
packet.
Some intermediate samples being filled, packet is not full yet, so at this time only
possibly last gate informtion of first sample may be filled.*/

                                        if
(ntohs(send_data_packet.local_header.start_sample)==sample_group_first_sample)
                                                {
                                                // First sample group can be included
in the packet and there is still space left for more.

          send_data_packet.local_header.start_sample_last_gate = htons(NUM_CATES);
// Last sample is same as total number of gates.
                                                //Remaing field should be filled when
packet is filled and sent.
                                                }
                                        for (i=0;i<sample_group_requirement;i++)
                                                {
                                                // Number of gates to be transmitted
are less than what can go into single packet.

                                memcpy(send_data_packet.payload+packet_buffer_offset,
                                file_buffer_ptr,SAMPLE_SIZE*num_gates_to_transmit);
                                        data_transmitted +=
SAMPLE_SIZE*num_gates_to_transmit;

          packet_buffer_offset+=SAMPLE_SIZE*num_gates_to_transmit;
                                                file_buffer_ptr   = file_buffer_ptr +
NUM_CATES*SAMPLE_SIZE;

          send_data_packet.local_header.sample_pattern
|=base_pattern>>(sample_group_first_sample+i - first_sample_in_packet);

          payload_size+=SAMPLE_SIZE*num_gates_to_transmit;
                                                }

                                        // These are the gates that can be transmitted
for next sample
                                        num_max_gates -=num_gates_to_transmit;
                                        // Since packet is not full, so check next group
present
                                        last_sample_pending_gates=0;     // All required
gates included in the same packet for the last sample
                                        sample_index++;   // Sample group is completly
sent, so go to next sample group in the tick.
                                        if (data_transmitted==total_data_to_send_in_tick)
                                                {
                                                // There is no more data to include in
this packet in the current tick so fill the remaining fields of the header and send this
packet */

          send_data_packet.local_header.end_sample = htons(sample_group_first_sample);
                                                if
(send_data_packet.local_header.start_sample!=send_data_packet.local_header.end_sample)
                                                        {
                                                        // Then fill the remining
fields of the header

          send_data_packet.local_header.start_sample_last_gate = htons(NUM_GATES); // last
gate is same as total number of gates per ray

          send_data_packet.local_header.end_sample_first_gate  = htons(1);      // 1st gate
is always sent here

          send_data_packet.local_header.end_sample_last_gate   = htons(num_max_gates);
                                                        }
                                                else
                                                        {
                                                        // Both first sample and
last sample are same , this will happen when packet sizes are small compared to ray size

          send_data_packet.local_header.end_sample_first_gate =
send_data_packet.local_header.start_sample_first_gate;

          send_data_packet.local_header.start_sample_last_gate =
htons(ntohs(send_data_packet.local_header.start_sample_first_gate)+num_gates_to_transmit-1);

          send_data_packet.local_header.end_sample_last_gate  =
send_data_packet.local_header.start_sample_last_gate;
                                                        }

          send_data_packet.local_header.sample_pattern =
htons(send_data_packet.local_header.sample_pattern);

          send_data_packet.local_header.payload_size   = htonl(payload_size);

                                                if (primary_table==1)
                                                        client_ptr =
RT_table[rate_index].client_list_type_2;
                                                else
                                                        client_ptr =
RT_table_second[rate_index].client_list_type_2;

                                                while (client_ptr!=NULL)
                                                        {
                                                        int temp_idx;
                                                        char *temp_ptr;
                                                        bzero((char
*)&client_addr,sizeof(client_addr));
                                                        client_addr.sin_family
= AF_INET;
                                                        inptr->s_addr
= htonl(client_ptr->client_ip);
                                                        bcopy((char *)inptr,(char
*)&client_addr.sin_addr,sizeof(struct in_addr));
                                                        client_addr.sin_port
= htons(client_ptr->client_port);
                                                        if (next_seq_number==1)
                                                                {
                                                                /*
                                                                First Packet of
Ray being sent, so at this time initial count of ray packet sent in the history list can be
set to 0.
                                                                */
                                                                client_ptr-
>packet_sent_count[current_ray_num%MAX_COUNT_HISTORY]=0;
```

```
                                                                    }
                                                                    temp_ptr=
send_data_packet.payload;
                                                                    client_ptr-
>packet_sent_count[ntohs(send_data_packet.local_header.ray_num)%MAX_COUNT_HISTORY]+=1;
                                                                    if
((sendto(server_send_sock,&send_data_packet,payload_size+sizeof(local_header_type)+sizeof(fi
le_ray_header_type), 0,(struct sockaddr *)&client_addr,sizeof(struct sockaddr_in))) < 0)
                                                                    {
                                                                    perror("RADAR
Server: Error in  sending DATA");
                                                                    exit(0);
                                                                    }
                                                                    client_ptr = client_ptr-
>next;
                                                                    }
                                                            packet_sent=1;
                                                            } // when data transmitted == total
data to send
                                                    } // else
                                            } // while packet not sent
                                    /* It is possible that all the gates of a packet were not included in
the same packet thus it needs to be taken care in next packet. */
                                    } // While all data for a tick is not sent
}

void print_data(int size)
{
          int i;
          for (i=0;i<size;i+=4)
                    {
                    printf("Data word i %x \n",*(int *)(&send_data_packet+i));
                    }
}


/*
This function should create packets for the data to be transmitted within the given time
window

*/
inline void create_send_packet(int payload_size)
{
          int start_sample, end_sample, start_sample_first_gate;
          int remainder,data_already_sent_of_sample;

          /*
          Compute How much data need to be transmitted within the given window.
          */
          data_already_sent_of_sample= ray_data_offset%(SAMPLE_SIZE*NUM_GATES);
      // remainder is 0, if sample of all gates are sent otherwise non zero
          start_sample = ray_data_offset/(SAMPLE_SIZE*NUM_GATES)+1;
          start_sample_first_gate = (data_already_sent_of_sample/SAMPLE_SIZE) +1;
          end_sample = (ray_data_offset+payload_size)/(SAMPLE_SIZE*NUM_GATES)+1;
          send_data_packet.local_header.seq_num                = htonl(next_seq_number++);
          send_data_packet.local_header.start_sample           = htons(start_sample);
          send_data_packet.local_header.start_sample_first_gate =
htons(start_sample_first_gate) ; // Gate after the earlier packets sent
          send_data_packet.local_header.end_sample             = htons(end_sample);
          if ((remainder=(ray_data_offset+payload_size)%(SAMPLE_SIZE*NUM_GATES)))
                    {
                    if (start_sample!=end_sample)
                              {
                              send_data_packet.local_header.start_sample_last_gate =
htons(NUM_GATES);
                              send_data_packet.local_header.end_sample_first_gate  =
htons(1);
                              send_data_packet.local_header.end_sample_last_gate   =
htons(remainder/SAMPLE_SIZE);
                    }
                    else
                    {
                    // Both Start and End Samples are same, so determine the
last gate
                    send_data_packet.local_header.start_sample_last_gate =
htons(remainder/SAMPLE_SIZE);
                    // Set End Sample Gates same as Start Sample Gates,Client
may not use it but still set it.
                    send_data_packet.local_header.end_sample_first_gate  =
send_data_packet.local_header.start_sample_first_gate;
                    send_data_packet.local_header.end_sample_last_gate   =
send_data_packet.local_header.start_sample_last_gate;
                    }
          }
          else // After Adding Raw Data, all samples of packets are included.
                    {
                    // After Adding RAW data, all samples of all gates are included in the
packet.
                    if (start_sample!=end_sample)
                    {
                    // Both Samples are different, so set the Gate Bounds.
                    send_data_packet.local_header.start_sample_last_gate =
htons(NUM_GATES);
                    send_data_packet.local_header.end_sample_first_gate  =
htons(1);
                    send_data_packet.local_header.end_sample_last_gate   =
htons(NUM_GATES);
                    }
                    else
                    {
                    // Both Start and End Samples are same, so determine the
last gate
                    send_data_packet.local_header.start_sample_last_gate =
htons(NUM_GATES);
                    // Set End Sample Gates same as Start Sample Gates,Client
may not use it but still set it.
                    send_data_packet.local_header.end_sample_first_gate  =
send_data_packet.local_header.start_sample_first_gate;
                    send_data_packet.local_header.end_sample_last_gate   =
send_data_packet.local_header.start_sample_last_gate;
                    }
          }
          send_data_packet.local_header.payload_size = htonl(payload_size);
          ray_data_offset +=payload_size;
          packet_send_count++;
}
```