INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

WMTP: Wireless Modular Transport Protocol

A Modular Approach to
Wireless Sensor Network Transport Layer Protocols

Luís David Figueiredo Mascarenhas Moreira Pedrosa

Dissertação para obtenção do Grau de Mestre em Engenharia de Redes de Comunicação

Júri

Presidente:	Prof. José Manuel Rego Lourenço Brázio
Orientador:	Prof. Rui Manuel Rodrigues Rocha
Vogais:	Prof. Jorge Miguel Sá Silva
	Prof. Rodrigo Seromenho Miragaia Rodrigues

Novembro de 2007

This dissertation is dedicated to my Father, Luís Filipe Pedrosa,
for his constant support throughout my life.

Acknowledgments

First and foremost, I would like to thank my supervisor, Professor Rui Rocha, for introducing me to the world of wireless sensor networks. Working with him has been a great honor and it is needless to say that, without his subtle pressure, vast wisdom, and strong support, this project would not have become a reality.

I also extend my thanks to the whole GEMS team (Group of Embedded networked Systems and Heterogeneous Networks) over at LEMe (Laboratory of Excellence in Mobility) for being excellent soundboards. Not only has their helpful input been an excellent sanity check for many of the ideas that were bounced around before being put into practice, but their company has made our work place all the more enjoyable.

Most importantly, I would like to thank my father, Luís Filipe Pedrosa. Not only has his encouragement and patience helped push me forward during these difficult times, but his constant support, throughout my life, has made me the person I am today.

Sumário

Esta dissertação propõe um novo protocolo de transporte modular para redes de sensores sem fios: o WMTP – Wireless Modular Transport Protocol. Este protocolo não só permite a utilização simultânea de todas as principais funcionalidades que são frequentemente encontradas nos protocolos de transporte deste tipo de redes, nomeadamente o controlo de congestão, a justiça na utilização dos recursos da rede, e a fiabilidade, como também o faz de uma forma modular. Deste modo, a aplicação pode utilizar exactamente as funcionalidades que são requeridas, sem ter que aceitar as contrapartidas inevitáveis de quaisquer outras que não o são. Adicionalmente, o WMTP oferece um conjunto único de funcionalidades menos comuns, como a capacidade de regular o débito da geração dos dados, o controlo de fluxo, a qualidade de serviço ao nível de transporte e a integração opcional com descoberta de serviços.

Por outro lado, a utilização desta arquitectura modular permite ao WMTP suportar ambientes heterogéneos, onde aplicações diferentes, que utilizam funcionalidades diferentes, podem coexistir, pacificamente, dentro da mesma rede. Adicionalmente, o administrador da rede pode ainda preparar versões reduzidas do protocolo que não suportam as funcionalidades que nunca virão a ser utilizadas durante a vida útil da rede, libertando, desta forma, preciosos recursos computacionais que poderão ser utilizados para outro fins mais úteis.

Palavras-chave

Redes de Sensores sem Fios, Protocolos de Transporte, Modularidade, WMTP

Abstract

This dissertation proposes a new modular transport layer protocol for wireless sensor networks (WSNs): WMTP – Wireless Modular Transport Protocol. This protocol not only allows the simultaneous use of all the main features commonly found in WSN transport protocols, namely congestion control, fairness, and reliability, but also does so in a modular fashion. This way, the application layer can choose to use exactly the features that it requires, without having to deal with the inevitable trade-offs associated with the ones that it doesn't. Additionally, WMTP provides its own unique set of uncommon features such as throttling, flow-control, transport layer quality-of-service, and optional integration with service-discovery.

On the other hand, the use of this specialized modular architecture also allows WMTP to support heterogeneous environments where different applications, using different features, coexist peacefully within the same network. Moreover, the network administrator may also build stripped down versions of the protocol that don't support the features that will never be used during the network's life-time, thus freeing up additional resources to be used for an otherwise more useful purpose.

Keywords

Wireless Sensor Networks, Transport Layer Protocols, Modularity, WMTP

Table of Contents

List of Tables

List of Figures

List of Annexes

List of Acronyms

AIMD	Additive Increase, Multiplicative Decrease
ARC	Adaptive Rate Control
ATM	Asynchronous Transfer Mode
ATP	Ad-hoc Transport Protocol
CCF	Congestion Control and Fairness
CODA	Congestion Detection and Avoidance
DTC	Distributed TCP Cache
ESRT	Event-to-Sink Reliable Transport
MAC	Medium Access Control
MPLS	Multiprotocol Label Switching
PCCP	Priority-Based Congestion Control Protocol
PSFQ	Pump Slowly, Fetch Quickly
QOS	Quality of Service
RBC	Reliable Bursty Convergecast
RMST	Reliable Multi-Segment Transport
STCP	Sensor Transmission Control Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
WMTP	Wireless Modular Transport Protocol
WSN	Wireless Sensor Network

1 Introduction

The recent evolution of networking in embedded systems has brought about new challenges to tackle and new problems to solve. The special needs related to these new kinds of networks and the paradigms that evolved from them ultimately led to the creation of the concept of Wireless Sensor Networks (WSNs), as described in [1]. These networks differ from conventional networks in many key aspects. Generally speaking, these networks are built upon low cost nodes with restrictive power and processing abilities. These nodes, in turn, are comprised of the sensing or actuating unit, a microcontroller and a wireless transceiver, thus extending the node's functionality beyond mere sensing and enabling the creation of intelligent data centric networks.

Although traditional network performance issues and challenges also apply, to some extent, to WSNs, the key performance attribute that is most frequently analyzed and optimized in the literature is energy efficiency. Since the wireless sensor nodes are frequently battery powered, the energy consumed during their operation equates directly into the overall network life-time. While changing the batteries on a common household device may seem simple enough, for large-scale WSN deployments, it is commonly impractical, more expensive, or completely impossible to change the batteries after deployment. This ultimately means that the use of energy efficient network protocols can lead to the need to redeploy the network with new sensor nodes equipped with fresh batteries after a year, or even longer, rather than every few months. Aside from the complex, often overseen, considerations and procedures that a large-scale WSN deployment ultimately entails, which were effectively assessed in [3] and [20], economic factors and environmental concerns may also come into play, thus making the energy efficiency performance attribute the most important aspect behind these new networks, being one of the key factors that can make or break this technology.

Given this key paradigm shift in what is considered as a performance attribute in WSNs, the use of traditional network protocols, that have already been tried and proven on typical wire-line and wireless networks, is generally found to be inefficient or impractical to implement on WSNs. This has led to the need to develop new network protocols specifically tailored for these networks, either by developing entirely new concepts from scratch, or by adapting the already existing protocols to better perform under these special conditions.

Traditionally, research and development efforts were application driven, thus embedding, within the applications themselves, most of the functionality that would otherwise be offered by one of the network layers. This would lead to the need to redevelop most of the protocol stack, from scratch, every time a new application was designed. With the recent surge in applications that rely on these networks, however, came a demand for new protocols that could fulfill the needs of, and thus be reused by, a broader range of applications, therefore

relieving the application developers of the additional effort of developing the underlying network layers.

Focusing specifically on the transport layer, one may find a large variety of protocols that have already been designed to provide some specific functionality that may be used by a broad range of applications. A more detailed analysis of some of the available protocols is available in [25], but, generally speaking, the functionality that these protocols provide can fit into one of the following categories: reliability, congestion control or fairness.

Reliability is the ability to retransmit lost packets, either locally, or end-to-end, to ensure their successful delivery. This feature has already been implemented in several protocols such as ATP ([19]), DTC ([7]), ESRT ([17]), GARUDA ([16]), PSFQ ([22]), RBC ([27]), RMST ([18]), and STCP ([13]). Congestion control, in turn, is the ability to delay or inhibit packet forwarding and generation in order to avoid network congestion at bottleneck nodes. This feature also has its fair share of protocols, namely ARC ([26]), ATP ([19]), CCF ([8]), CODA ([23]), DTC ([7]), ESRT ([17]), Fusion ([11]), PCCP ([24]), Siphon ([21]), and STCP ([13]). Finally, fairness is the ability to divide network resources in an equitable fashion between all nodes, thus ensuring that all have an equal share of bandwidth to communicate with the sink node. This feature has been covered by ARC ([26]), ATP ([19]), CCF ([8]), Fusion ([11]), and PCCP ([24]).

The weakness behind most of these protocols is one all too common in WSNs: most of these protocols were designed with the needs of a specific application in mind, and are either not suitable or inefficient for most other purposes. Additionally, most transport protocols designed for reliability do not offer congestion control and vice-versa. The few protocols that do offer both of these features ([7, 17, 13]), in turn, come short in performance, when compared to other protocols that either just implement reliability or just congestion control. Fairness, on the other hand, is mostly associated with congestion control, but the protocols that follow this path don't generally provide any reliability semantics. The one key area, where all of these protocols come short, is modularity. Each of these protocols provides its own particular features, or combination thereof, yet none present the application layer with the explicit option of using whichever features they need, while leaving out the others.

1.1 Motivation and Goals

This dissertation proposes the Wireless Modular Transport Protocol (WMTP), a novel modular approach to transport layer protocol functionality that not only seamlessly integrates all of the above mentioned functionalities, but also provides some new features, not commonly found in this area, such as the support for optionally integrated service discovery and the ability to provide quality-of-service (QoS). This means that each application may choose which features it requires and which it doesn't, and the protocol will assure that the basic requirements are complied with, without incurring the additional burden and the efficiency toll associated with any unused functionality. Additionally, WMTP

supports environments with heterogeneous applications. In other words, different applications, using different features, may coexist in the same network. On the other hand, the use of this modular architecture also allows the network administrator to build a stripped down version of WMTP that doesn't support any features that will never be used during the network's life-time, thus freeing up valuable processor and memory resources on the sensor nodes.

Although, as a generic protocol, WMTP could be implemented on any system, its reference implementation has been developed on the TinyOS 1.x platform, and has been primarily tested on Crossbow MICAz sensor nodes ([6]). The Crossbow MICAz is a commercially available sensor platform that couples an 8-bit AVR RISC microcontroller, the Atmel ATmega128L ([2]), with an IEEE 802.15.14 Zigbee-ready transceiver, the Chipconn CC2420 ([5]). The use of these platforms, in itself, is a challenge to overcome, especially due to the reduced amount of available memory on the MICAz sensor node (4 kB of SRAM).

TinyOS, in turn, is an open-source operating system initially developed at the University of California, Berkeley. This operating system was specifically designed to work on embedded systems with very severe computational constraints, such as those found in WSNs, and has, since then, become the *de facto* standard operating system for several WSN platforms. TinyOS accomplishes its extreme resource economy through the use of an event-driven, component-based architecture with cooperative process multitasking. This specialized architecture is further supported by a specialized C-like programming language, with which all TinyOS components must be programmed, called nesC ([9]). Additionally, TinyOS provides a powerful simulation environment, TOSSIM ([14, 15]), which enables the simulation of entire WSNs using the same source-code that is used on real sensors.

1.2 Organization

The remainder of this dissertation is organized into five main chapters. The following chapter, chapter two, provides an overview of the current state of the art of WSN transport protocols. This chapter starts by providing an in-depth insight into each transport feature, followed by a brief outline of each of the most commonly used WSN transport protocols. Finally, each protocol is cross-referenced with the features that it provides in a series of condensed tables.

Chapter three, in turn, provides a detailed account on how the WMTP protocol architecture was designed. Starting off by explaining WMTP's initial design goals and the requirements that ultimately derived from them, this chapter provides an insight into the various alternative solutions that were a part of the initial design phase. Finally, the architecture that ultimately developed into WMTP will be explained in further detail, followed by some implementation considerations.

Chapter four provides an objective validation of WMTP's functionality, as well as an evaluation of its performance. This validation and evaluation is processed through simulation, hence this chapter starts by describing the test application and scenarios developed specifically for this purpose. Once the simulation procedure is made clear, the simulation results are presented and discussed.

Finally, chapter five draws some final conclusions and lays out the foundation for future work in this area, followed by chapter six with a list of references.

1.3 Summary of Contributions

This dissertation was partially developed as a contribution to the CRUISE Network of Excellence IST Project (CReating Ubiquitous Intelligent Sensing Environments), a part of the Sixth EU Framework Programme for Research and Technological Development (FP6). As such, the following technical reports have been delivered:

- L. Pedrosa, R. Rocha, R. Neves, "Protocol comparison and new features", CRUISE/WP220/IT/032/0.3/13.11.2006: This report provides an insight into the state of the art of WSN transport protocols and ultimately became a part of the CRUISE/WP220/D220.1/version 2.0/04.10.2007 milestone.
- L. Pedrosa, R. Rocha, R. Neves, "WMTP – Wireless Modular Transport Protocol" CRUISE/WP220/IT/046/0.4/07.07.2007: This report contains an initial specification for WMTP.

Although WMTP's initial specification has been submitted as a contribution, the latest version is not, as of yet, publicly available.

2 State-of-the-Art of WSN Transport Layer Protocols

In this section, the current state-of-the art of WSN transport protocols is reviewed in further detail. After explaining the main transport layer features that are commonly found in the already existing WSN transport protocols, a list of atypical features will be presented and further explained. Once this basic foundation is established, each of the main previously existing WSN transport protocols will be briefly outlined. Next, a list of summarized tables will be presented, cross-referencing each typical feature with the protocols that implement it. Finally, some brief conclusions will be drawn.

2.1 WSN Transport Protocol Common Functionalities

2.1.1 Reliability

Reliability can be described as the ability that the network has to ensure the proper delivery of information to its final destination. In wireless sensor networks reliability can fit within one of two categories: packet reliability and event reliability. The former ensures that all packets (or a configurable percentage of them) arrive to their final destination, while the latter ensures that, at least, the minimum amount of packets required to correctly detect an event, are safely delivered. Additionally, to provide either of these reliability semantics, most algorithms are further divided into two main stages: loss detection and notification and loss recovery. Furthermore, reliability algorithms can be classified by the nodes that directly intervene in them, being either end-to-end or hop-by-hop.

The loss detection and notification stage of the reliability algorithm is used to detect when a packet has been lost and is thus responsible for initiating any action to recover the loss. In order to perform this task, one of the following methods may be used:

- *ACK Feedback*: The ACK feedback mechanism is based on the receiver, be it either the final destination or just one of the hops, explicitly acknowledging the reception of each packet. Using this mechanism, the sender, be it either the original source or just the previous hop, detects that a packet has been lost if it has not been acknowledged by the receiver within a specific time-frame.
- *NACK Feedback*: NACK feedback, in turn, is based on the receiver explicitly notifying the sender that it did not receive a packet. Just as before, this receiver may either be the final destination, or just another hop along the way. Since, in traditional networks, packets successfully arrive at their destination more often than not, this mechanism generally implies a smaller protocol overhead on the network. On the other hand, it also brings a new challenge: if all packets are lost, the receiver will never be the wiser, and will therefore never notify the sender.

- *IACK Feedback*: IACK (Implicit Acknowledge) feedback is a new mechanism that takes advantage of the promiscuous nature of the radio environment. In this mechanism, when a network node forwards a packet to the next hop, it implicitly acknowledges the packets reception to its previous sender. Although this mechanism presents even less overhead than the NACK mechanism, its basic assumption, that a node may overhear the packet being forwarded, may not always be applicable, specially if the underlying link layer protocol works with multiple non-interfering channels, or is TDMA based.
- *Sequence Number Out-of-Order*: In this specific mechanism, packets are tagged with consecutive sequence numbers. The receiving node can then detect a missing packet when it receives another packet with the sequence number out of the expected order. Once the loss is detected, a NACK mechanism may be used to notify the sender.
- *Time-Out*: The Time-Out loss detection mechanism, like the Sequence Number Out-of-Order variant, requires the aid of the NACK mechanism to notify the sender. However, this specific mechanism does not rely on sequence numbers in messages to detect when one has been lost but, rather, expects that a new message will be delivered within a certain time-frame, after which, the message will be considered lost.

Once the loss detection and notification stage has detected that a packet has been lost, one of the following recovery mechanisms may be used:

- *Increase Source Sending Rate*: This mechanism is frequently found in the event reliability semantic. Since individual packets may be lost without hindering the overall application functionality, the sending node can increase the total quantity of packets received on the other end by simply increasing the total number of packets it sends.
- *Packet Retransmission*: This mechanism, on the other hand, is the general choice for most packet reliability protocols. On the end-to-end variant, the original source node retransmits the packet across the entire network, hopefully reaching the final destination. The hop-by-hop variant, in turn, performs local retransmissions on each hop and, by doing so, reduces overall protocol overhead and latency. It is also possible to reach a mid-term solution, where not all nodes along the path cache the transmitted packets. This intermediate solution still requires multi-hop retransmissions, but still manages to avoid end-to-end retransmissions.

2.1.2 Congestion Control

Given the convergent nature of most WSN data forwarding schemes, network congestion is likely to happen on nodes closer to the sink node. This is especially the case when these nodes forward data for particularly large networks which need to convey vast amounts of information in frequent status reports. If this situation is not taken into account, then congestion will inevitably occur sooner or later, leading to overloaded radio links, degraded channel utilization and the wasteful transmission of packets that will eventually be dropped.

Given the general need to maximize network utilization while avoiding the wasteful use of energy to transmit packets that may be dropped, a general mechanism to control source rates in a manner that avoids downstream congestion is required. The algorithms that implement this functionality can be broken down into three stages: congestion detection, congestion notification, and rate adjustment.

The congestion detection stage oversees the local node's status and decides if congestion is either already taking place or likely to take place in the near future, if nothing else is done to prevent it. This decision may either produce a single binary congestion notification (CN) bit, a multilevel congestion degree value, or a precise rate at which each child node that is using the current node as its next hop should send its packets. In order to provide this functionality, one of the following mechanisms may be used:

- *Packet Sending Success*: In this simple mechanism, the success or failure to send a packet is used to infer congestion. This can be used in a hop-by-hop basis, where a node establishes its own congestion when it is unable to send a packet over the next hop, or in an end-to-end basis, similar to the way TCP works.
- *Queue Length*: This mechanism relies on the local node's message queue occupancy. Once the relative quantity of queued packets surpasses a certain predefined threshold (e.g. 75%), the node is considered to be congested, and shall proceed to notify its peers of that fact.
- *Packet Service Time*: Unlike the queue length mechanism, this specific mechanism does not rely on the local message queue status, but rather on the local node having the ability to precisely quantify the maximum data rate at which it may send packets over the next hop. Provided that this information is available, the node can limit its own rate and calculate the rate at which its own children may send packets. This information may then be propagated further down, effectively limiting the rate of all nodes throughout the network in a way that may keep congestion under control.

- *Ratio of Packet Service Time over Packet Inter-arrival Time*: Unlike the packet service time mechanism, this mechanism does not need to establish the maximum data rate for the following hop. By simply taking into account the mean packet service time (the elapsed time from when the packet is delivered to the link layer up until its last bit is successfully transmitted over the next hop) and the mean packet inter-arrival time (the elapsed time between consecutive packet arrivals, be them from the link layer or locally generated), a simple congestion degree value can be obtained by calculating the ratio of the former over the latter.
- *Channel Loading*: In this specific mechanism, the radio channel is constantly monitored, allowing the node to measure the channel's relative load, thus detecting local congestion. Since continuous channel monitoring may entail a high energy toll, a channel sampling scheme is implemented in practice.

The congestion notification stage, in turn, defines the method used by the parent node to notify its children of its current congestion status. One of the following mechanisms may be used to provide this functionality:

- *Explicit Congestion Notification*: Once the congestion status has been determined, all of the local node's children nodes must be notified so they may take action and prevent further congestion. In explicit congestion notification, specific protocol management messages are used for this purpose.
- *Implicit Congestion Notification*: Unlike in the explicit variant, implicit congestion notification piggybacks the congestion status information on normal data packets, reducing the overall protocol overhead on the network. On the other hand, just like the IACK reliability mechanism, implicit congestion notification assumes that a node may overhear forwarded packets, which may not always be true.

Finally, the rate adjustment stage defines how children nodes should limit their transmission rates in order to avoid further congestion at the parent node. In order to provide this functionality, one of the following mechanisms may be used:

- *Stop-and-Start Rate Adjustment*: This mechanism invokes a simple principle: once the parent node notifies its children that it is congested, the children stop sending packets over the next hop, allowing the parent to free up its queues. If the children, themselves, get congested, they will continue to apply back pressure upstream, ultimately reaching the data source and temporary halting packet generation.

  When this method is allied with implicit congestion notification, some special attention must be paid to the fact that the node may still need to send

messages to its congested parent to be able to effectively notify its own children of its congestion status.

- *Additive Increase, Multiplicative Decrease (AIMD) Rate Adjustment*: Unlike before, this mechanism does not completely stop sending packets when parents report that they are congested. Instead, an additive increase, multiplicative decrease scheme is used to regulate the rate at which packets are sent.
- *Exact Rate Adjustment*: This specific mechanism relies on the node's ability to precisely determine the rate at which it may send packets over the next hop. Provided that this information is available, the node simply schedules the sending of its packets using specific timings in order to fulfill that calculated rate.

2.1.3 Fairness

Given the probabilistic model of packet loss at each hop, be it by the hands of radio link interference or by congestion, a natural consequence is that sources that require more hops to arrive to the sink node tend to have a larger probability of packet loss than those closer to it. This problem ultimately limits the diameter of the network and creates an imbalance where nodes closer to the sink have an unfair advantage over those farther away.

To counteract this natural imbalance, some special care is needed in the transport protocol's design to guarantee network fairness. Generally, these guarantees are associated with congestion control protocols and can be divided into two categories: simple fairness and priority based, or weighted, fairness.

In simple fairness, all traffic sources are considered equal and will receive equal transmission opportunities. In order to provide this functionality, the following mechanisms may be used:

- *Simple Rate Limiting*: This mechanism is, in many ways, similar to the exact rate adjustment congestion control mechanism, although some additional care must be taken to provide fairness guarantees. To insure fairness, the local node must send packets over the next hop at a rate proportional to its parent's total upstream rate and the ratio of the number of source nodes served by the local node over the total number of source nodes served by its parent. This information may either be explicitly broadcasted by the parent node or can be inferred by overhearing the parent node's forwarded traffic.
- *Differentiating AIMD Coefficients for Local and Forwarded Traffic*: When using an additive increase, multiplicative decrease rate adjustment congestion

control mechanism, simply using differentiated coefficients for locally generated traffic and forwarded traffic can supply a certain degree of fairness.

- *Traffic Shaping based on Multiple Queues at Each Node*: If each node maintains separate queues for traffic forwarded from each of its children, simple traffic shaping techniques may be used to guarantee fairness. These techniques should serve each queue at a rate proportional to the number of source nodes served by the child associated to the queue.

Priority based fairness, on the other hand, is more flexible and complex, since it allows each traffic source to have a different priority attributed to it. This being the case, only some of the previously mentioned mechanisms are extensible to provide this more advanced functionality:

- *Exact Rate Adjustment using Priority Based Scheduling*: This mechanism is designed as an extension of simple rate limiting that takes into account different priorities for different data sources. Traditionally, the rate at which the local node sends its traffic over the next hop is calculated based upon the number of source nodes that are served by the local node and the amount served by its parent. If, instead of just counting source nodes, a sum of their relative weights is performed, priority based fairness guarantees may be achieved.

- *Priority Based Traffic Shaping based on Multiple Queues at Each Node*: This mechanism, just like its simple fairness counterpart, requires that each node maintain a separate queue for traffic forwarded from each of its children. Once again, simple traffic shaping techniques may be used to guarantee fairness but, this time, each queue should be served at a rate proportional to the sum of the priorities of each source node served by the child associated to it, rather than the number of sources *per se*.

2.2 WSN Transport Protocol Atypical Functionalities

2.2.1 Throttling

Throttling is a basic functionality that allows the sending node to easily regulate the rate at which it generates its data, through the transport layer. This feature, albeit quite simple in nature, may be quite useful, since it relieves the application layer of the burden of managing its own timers.

2.2.2 Flow Control

Flow control is a functionality that allows the receiving node to regulate the rate at which the sending node generates its data. In fact, this feature is quite similar to

throttling, with the key difference lying in the fact that it is the receiving node that specifies the data generation rate, rather than the sending node.

This feature is implemented simply by allowing the receiving node to send feed-back packets towards the sending node, thus providing the explicit rate at which it should generate its data. Albeit simple in nature, this functionality is not always trivial to provide, especially since the basic assumption that it is possible to send data along the reverse path, from sink to source, is not always true.

Due to the limitations that the routing layer may have, only some of the following flow-control mechanisms may be applicable:

- *Single-Hop Flow-Control*: This mechanism works upon the premise that the receiving node, probably a sink node with an external power source, may use a stronger radio signal to reach all of the sending nodes within a single hop. The sending nodes, in turn, may use weaker radio transmissions and a multi-hop network, thus saving their own energy. By doing this, the routing problem is completely circumvented, but at the cost of limiting the total size of the network to the sink's extended radio range.
- *Broadcast Flow-Control*: This mechanism, in turn, assumes the existence of a multi-hop broadcast routing protocol, either based on flooding or through the use of a more complex multicast-like distribution system. Using this specialized routing layer, the receiving node may broadcast a sender address and its explicit rate, across the entire network, in order to configure a specific sending node. This solution, when used in this particular way, is very inefficient but, in many cases, it may very well be the only alternative. A more efficient use of this mechanism may be applied when all sender nodes may be configured to generate data at the same rate. This being the case, the effort of broadcasting the configuration data across the entire network will have been put to a good use.
- *Unicast Flow-Control*: Finally, this mechanism uses a unicast routing layer, if one exists, to send the explicit generation rate configuration directly to the sender node, thus avoiding that the configuration data be forwarded where it is not needed.

2.2.3 Quality-of-Service

Quality-of-service is a functionality that has a different meaning, depending on the layer from which it is viewed. According to [4], quality-of-service can be either perceived from an application standpoint or from a network standpoint. This way, while applications generally concern themselves with such quality-of-service parameters as network coverage, exposure to phenomena, measurement errors, and optimum number of

active sensors, the network generally attempts to ensure that local or end-to-end performance guarantees are met.

This being the case, application quality-of-service is primarily addressed during the deployment phase, although some network support is also required. Network quality-of-service, on the other hand, attempts to use prioritization schemes, admission control and reservation mechanisms to differentiate traffic, thus providing performance guarantees for some flows, while using “best effort” policies for the others. This can be seen both at the link layer, where quality-of-service is used to ensure that priority traffic is sent over the radio within a bounded time frame, as well as the upper layers, where end-to-end guarantees, such as maximum overall packet delay and minimum throughput, are pursued.

At the transport layer, the main concerns are related to these last kinds of guarantees. The main idea behind transport layer quality-of-service is the ability to allow applications to attempt to reserve network resources for specific network flows. Once the network asserts that it meets all the conditions necessary to honor the request, it proceeds to reserving the associated resources, and signaling the application that its request was completed successfully. Under these new circumstances, the transport layer must provide a high level of assurance that the application’s performance requirements will be met, regardless of future traffic conditions. Evidently, given the unpredictability of the wireless medium, not to mention the possible network topology changes that may occur in a mobile scenario, these assurance are far from being considered guarantees, but are rather a form of conditional hope that, while all stays as is, the previously negotiated conditions will continue to honored.

2.3 Existing WSN Transport Protocols

2.3.1 Adaptive Rate Control (ARC)

The Adaptive Rate Control (ARC) protocol, as described in [26], provides congestion control and simple fairness semantics. The idea behind the protocol is that when a node transmits data over to its parent node, it uses the success or failure to send the data as a congestion detection mechanism. This way, AIMD mechanisms may be used to regulate the rates at which data is locally generated or forwarded for other nodes.

Additionally, simple fairness is achieved by using different AIMD parameters for local traffic generation and remote traffic forwarding. By monitoring the traffic that it forwards, each node establishes how many children nodes it ultimately serves. It then uses this knowledge to carefully calculate the AIMD parameters for both local traffic generation and remote traffic forwarding.

2.3.2 Ad-hoc Transport Protocol (ATP)

The Ad-hoc Transport Protocol (ATP), as described in [19], is yet another TCP alternative, specifically developed for ad-hoc networks. This protocol intends to resolve several of the key problems that TCP has traditionally suffered over wireless ad-hoc networks by using a novel approach that not only decouples congestion control from reliability, but also uses additional feed-back from intermediate forwarding nodes to calculate precise estimates of the network's state. These mechanisms, alongside a three-phase congestion control mechanism, allow ATP to adapt to network conditions much faster and more accurately than TCP.

In order to provide congestion control and simple fairness, the intermediate nodes along the data's forward path collaborate to calculate the maximum packet service time by piggy-backing the highest value amongst them alongside the packet's data. The receiver node then uses this information to periodically send packets back along the reverse path, ultimately providing the sender with an explicit rate. The sender node, in turn, uses this rate to feed a three-phase congestion-control mechanism that uses distinct rate progression procedures during the increase, decrease, and maintain phases.

As for reliability, ATP uses a selective ACK (SACK) mechanism to allow the receiver to specify exactly which packets it has received and which remain to be received, being detected as holes in the sequence number's natural progression. The sender node then uses this information to retransmit the lost packets, without relying on any retransmission timeouts.

Overall, ATP outperforms TCP but, ultimately, its mechanisms are not optimized to minimize energy consumption, but rather to maximize performance at high data rates and are, thus, not quite tailored for WSNs.

2.3.3 Congestion Control and Fairness (CCF)

The Congestion Control and Fairness (CCF) protocol, as described in [8], is used to provide congestion control and fairness semantics. To accomplish this, each node measures the rate at which it can send data and then divides it by the number of children nodes it serves, thus obtaining the per-node packet rate. This per-node packet rate is then piggy-backed alongside forwarded data, thus taking advantage of the broadcast nature of wireless networks to implicitly transmit this data to the node's children. If a node overhears its parent's rate and it is slower than its own, it will apply it locally, as well as use it in future broadcasts.

Additionally, each node creates individual packet queue for each of its children nodes, as well as one for its locally generated data. This being the case, the node now uses

the already established per-packet rate, as well as a specialized traffic shaper, to provide the intended fairness.

2.3.4 Congestion Detection and Avoidance (CODA)

The Congestion Detection and Avoidance (CODA) protocol, as described in [23], is used to avoid congestion in data flows from sources to sink. To accomplish this, two detection techniques are employed: a traditional queue occupancy threshold and a channel loading mechanism.

Once congestion has been detected, using a combination of these methods, CODA then uses two feed-back mechanisms to recover from it: open-loop backpressure for light, transient, congestion and closed-loop sink generated ACKs for heavy, long-term, congestion.

2.3.5 Distributed TCP Cache (DTC)

The Distributed TCP Cache (DTC) protocol, as described in [7], is an extended version of the traditional TCP protocol that is optimized for energy efficiency. This is accomplished through the use of local caches, along the connection's route, that enable a quicker and more efficient local loss recovery process, instead of relying solely on TCP's end-to-end loss recovery mechanisms.

Additionally, DTC uses a network aided process, called flying-start, to provide more accurate initial round-trip-time estimates to the source node, thus providing a 10% to 25% performance increase over the conservative values that TCP traditionally uses.

2.3.6 Event-to-Sink Reliable Transport (ESRT)

Event-to-Sink Reliable Transport (ESRT), as described in [17], is used to provide event reliability. Instead of the traditional concept of packet reliability, where individual packets are retransmitted, in order to assure their safe delivery at the sink, this protocol defines event reliability as the received packet rate associated to a particular sensing process. This way, higher event reliability is synonymous with the reception of more packets, thus providing a more accurate perception of the sensed phenomena.

The true problem that ESRT intends to solve is how to reach the ideal event reliability, thus providing the right amount of data to produce the required accuracy, while also avoiding network congestion. If sources generate data too slowly, the required reliability metric won't be met, on the other hand, if data is generated too quickly, too much reliability may occur, or worse, network congestion may lead to dropped packets, thus lowering the event reliability once again, possibly even below the required threshold. This concept is illustrated in Figure 2.1.

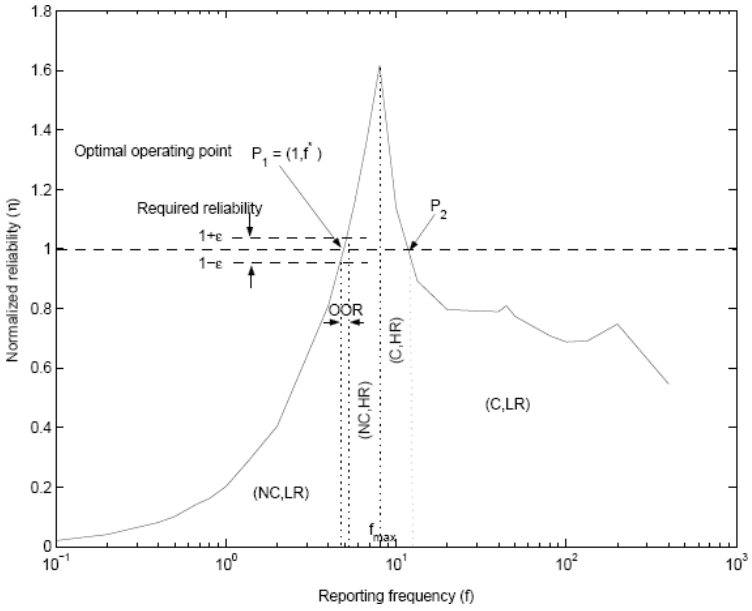


Figure 2.1: ESRT Event Reliability Behavior, as a Function of the Reporting Frequency

In order to solve this problem, ESRT uses a flow-control mechanism that assumes the existence of a single high-powered sink node that can control all of the source nodes within a single hop. The source nodes, in turn, may have low-power radios, and thus may use a multi-hop network to reach the sink. Under these circumstances, the sink calculates the optimal reporting frequency and directly regulates sources.

2.3.7 Fusion

The Fusion protocol, as described in [11], is used to provide congestion control and simple fairness semantics. In order to provide congestion control, each node detects congestion by monitoring its packet queue. The node then simply piggybacks a congestion notification bit alongside its data, thus taking advantage of the broadcast nature of the wireless medium to broadcast it congestion state to its neighbors. Once each node knows its parent's congestion state, it refrains from sending it more than one packet, thus implementing an open-loop back pressure mechanism. This ability to send a single packet to a congested parent is specifically allowed, so that the child node may to still warn its neighbors when it eventually becomes congested.

Fairness, on the other hand, is accomplished through a separate mechanism. The idea is that each node monitors its own forwarded traffic, as well as that of its parent. Using this information, the node is able to determine not only the number of source nodes that it serves, but also the number of source nodes its parent serves. Armed with this knowledge, the node then uses a token bucket mechanism to regulate the rate at which it forwards it own data, thus ensuring that it is forwarded at the appropriate fraction of the rate of its parent.

2.3.8 GARUDA

The GARUDA protocol, as described in [16], is used to reliably send data from sink to sources. In order to provide this functionality, this protocol starts off by creating a loss recovery core, during its first packet flood. This core is formed by all nodes with a hop-count that is a multiple of three, thus approximating a minimum dominating set. Each core node then piggybacks, alongside its forwarded data, special availability maps (A-Maps), containing lists of packets that it is holding in local cache. These core nodes are also responsible for retransmitting any NACKed data packets.

Under these circumstances, each node, whether it belongs to the core or not, detects packet loss through out-of-order sequence numbers, yet these out-of-order packets are still forwarded. Once a lost packet is detected, the node may only initiate its recovery process when it receives an availability map, containing the lost packet, from its preceding core node.

Additionally, GARUDA circumvents the single-packet loss problem, which is traditionally associated with NACK based mechanisms, through the use of out-of-band Wait-for-First-Packet (WFP) pulses, that are assumed to be reliable.

2.3.9 Priority-Based Congestion Control Protocol (PCCP)

The Priority-Based Congestion Control Protocol (PCCP), as described in [24], provides congestion-control and weighted fairness semantics. In order to provide this functionality, PCCP uses a novel congestion detection mechanism, based the ratio of its packets' service time (the time it takes the link layer to dispatch them) over inter-arrival time (the time elapsed between consecutive receptions).

Each node then piggybacks this congestion information, together with the number of nodes it serves and the sum of their weights, alongside its forwarded data, thus taking advantage of the broadcast nature of the wireless medium to broadcast its status to its neighbors. Each node then uses its parent's status information to perform exact rate adjustments on the rate at which it forwards its data, as well as the rate at which it generates it.

2.3.10 Pump Slowly, Fetch Quickly (PSFQ)

The Pump Slowly, Fetch Quickly (PSFQ) protocol, as described in [22], is used to reliably disseminate send large amounts of information across the entire network, being primarily designed to enhance network retasking or reprogramming. Under these circumstances, data cache sizes are not relevant, since all nodes must hold all data segments anyway.

The idea behind this protocol is to flood the network with new segments (pump) slowly, while recovering from a detected loss (fetch) quickly. These losses are detected

through out-of-order sequence number and recovered using a simple NACK mechanism. Additionally, nodes do not propagate out-of-order segments to prevent downstream nodes from initiating a fetch operation.

Finally, when the entire operation is complete, the nodes cooperate to send aggregated report messages.

2.3.11 Reliable Bursty Convergecast (RBC)

The Reliable Bursty Convergecast (RBC) protocol, as described in [27], is used to provide reliability semantics for traffic with real-time requirements.

In order to provide this functionality, this protocol uses a novel implicit windowless block acknowledgement approach that, in conjunction with a specialized prioritization mechanism, allows new packets to be generated and forwarded, without ever having to wait for old lost ones to be recovered. Additionally, in some special cases where the receiving node is able to proactively detect that it did not receive a sent packet, it accelerates the recovery process through the use of implicit block NACKs.

2.3.12 Reliable Multi-Segment Transport (RMST)

The Reliable Multi-Segment Transport (RMST) protocol, as described in [18], was designed to run over Directed-Diffusion (see [12] and [10]), specifically to reliably fragment and reconstruct data sets for sending over reinforced gradients.

This being the case, fragment loss is detected through out-of-order sequence numbers and time-outs and is notified through a NACK mechanism. Additionally, a special caching mode may be used, where intermediate nodes may cache fragments and participate in the recovery process.

2.3.13 Siphon

The Siphon protocol, as described in [21], provides congestion control through the use of a specialized, high-powered, long-range, secondary radio network.

This way, aside from a single physical sink that collects and consumes the network's data, siphon proposes the use of several virtual sinks, laid across the network. These virtual sinks are equipped with two radio modules, one low-powered and short-ranged and one high-powered and long-ranged. Although most conventional communications will go through the low-power radio network, these virtual sinks can use the alternative high-power radio to divert traffic, thus avoiding congestion on the primary network.

Additionally, two distinct congestion detection methods are used, a node initiated method and a sink aided, "post-facto", mechanism. The node associated method uses a local channel load measurement mechanism, as well as a traditional packet queue monitoring scheme, to perform an early detection of transient or deep network

congestion situations. The sink aided, “post-facto”, mechanism, in turn, allows the physical sink to perform a high level assessment of the application data fidelity, thus forcing the use of the secondary radio network whenever it sees fit.

2.3.14 Sensor Transmission Control Protocol (STCP)

The Sensor Transmission Control Protocol (STCP), as described in [13], is a flexible protocol that is used to provide both end-to-end reliability and congestion control functionality.

Within the reliability realm, STCP also provides the option to either use full or partial reliability semantics, thus allowing the application to specify how many packets, within a fixed window size, must be reliably delivered. Additionally, if the data is to be generated continuously at a fixed predictable rate, the receiving node may use a time-out mechanism to detect a packet’s loss, followed by a NACK packet to initiate its recovery. For unpredictable, event driven packets, a less efficient alternative ACK mechanism may be used.

As for congestion-control, STCP uses a simple, end-to-end, closed loop, congestion detection mechanism, based on packet queue occupancy monitoring. The idea is that, if an intermediate node detects that it is congested, it sets a congestion notification bit that is piggybacked along to the receiving node. Once the receiving node receives this information, it also piggybacks a congestion notification bit, but this time onto one of the ACK packets that return along the reverse path, thus instructing the source node to reduce the rate at which it generates its data.

2.4 Protocol vs. Feature Cross-Reference

In this section, each protocol is cross-referenced with the features that it provides, as well as with the methodology used to implement them. This information is condensed within three tables, one for reliability, one for congestion control, and one for fairness.

Protocol	Reliability									
				Loss Detection and Notification					Loss Recovery	
	Category	Direction	Type	ACK	NACK	IACK	Sequence Number Out-of-Order	Time Out	Increase Source Sensing Rate	Packet Retransmission
ARC										
ATP	Packet	Both	End-to-End	●			●			●
CCF										
CODA										
DTC	Packet	Both	Hop-by-Hop	●			●	●		●
ESRT	Event	Upstream	Event-to-Sink					●	●	
Fusion										
GARUDA	Packet	Down-stream	Hop-by-Hop		●		●			●
PCCP										
PSFQ	Packet	Down-stream	Hop-by-Hop		●		●	●		●
RBC	Packet	Upstream	Hop-by-Hop		●	●				
RMST	Packet	Upstream	Hop-by-Hop		●			●		●
Siphon										
STCP	Event / Packet	Upstream	End-to-End	●	●			●		●

Table 2.1: Reliability Protocol Comparison

Protocols	Congestion Control									
	Congestion Detection					Congestion Notification		Rate Adjustment		
	Packet Sending Success	Queue Length	Service Time	Service Time / Inter-arrival Time	Channel Loading	Explicit	Implicit	Stop-and-Start	AIMD	Exact
ARC	Hop-by-Hop						●		●	
ATP			●			●				●
CCF			●				●			●
CODA		●			●	●			●	
DTC	End-to-End						●		●	
ESRT		●					●			●
Fusion		●					●	●		
GARUDA										
PCCP				●			●			●
PSFQ										
RBC										
RMST										
Siphon *	End-to-End	●			●					
STCP		●					●		●	

* – Siphon does not perform any rate adjustment to mitigate congestion but, rather, redirects traffic through "Virtual Sinks" which use an alternative long-range radio network to reach the "Physical Sink".

Table 2.2: Congestion Control Protocol Comparison

Protocols	Fairness			
	Simple Fairness			Priority Based Fairness
	Simple Rate Limiting	Differentiating AIMD Coefficients	Traffic Shaping	Exact Rate Adjustment
ARC		●		
ATP	●			
CCF			●	
CODA				
DTC				
ESRT				
Fusion	●			
GARUDA				
PCCP				●
PSFQ				
RBC				
RMST				
Siphon				
STCP				

Table 2.3: Fairness Protocol Comparison

2.5 Discussion

A critical analysis of the previously mentioned protocols and features merely confirms what had been stated in chapter one: several transport protocols for WSNs have already been developed, but most of them were specifically designed to solve problems posed a particular application. This being the case, these protocols excel when used for the specific application for which they were designed, but are ultimately inadequate for any other use. Additionally, the few of these protocols that are actually generic in design, namely ATP ([19]), DTC ([7]), and STCP ([13]), attain their generality at the expense of performance (at least from a WSN's point of view). On the other hand, most of these protocols have been developed isolated from other research areas that are becoming progressively more popular in the WSN world, such as possible integration with service discovery mechanisms, or the inclusion of transport layer quality-of-service semantics.

This situation ultimately stresses the need for the development of a new modular approach to WSN transport layer functionality, a need that ultimately led to the development of WMTP. This new approach is generic by nature, yet it allows the application to include exactly the features it requires, while leaving out any others, a simple principle that is not followed by any of the mentioned protocols. This basic approach not only avoids the inevitable trade-offs one must concede to when using additional features, but also cuts down on the use of unnecessary overheads and, thus, will ultimately contribute to saving energy, therefore extending the network's useful life-time.

3 WMTP Design and Implementation

In this chapter, the design considerations behind WMTP's architecture will be further discussed. To start off with, the initial design goals, as well as the requirements that derived from them, will be presented, while outlining some of the alternative solutions that were thought of, during the initial design phase. Then, the interface that WMTP provides to applications will be explained, followed by a detailed coverage of WMTP's system architecture. Finally, some special considerations will be discussed regarding WMTP's reference implementation.

3.1 Design Goals and Requirements

As previously mentioned, the basic transport feature categories, namely, congestion-control, fairness, and reliability, have already been covered by several established and well proven transport protocols. What these protocols fail to provide is a modular architecture that enables the optional use of any one of these features, or a combination thereof, in a manner that is completely transparent to the application.

This is where WMTP comes in. WMTP was born from a proposal to develop an architecture that could integrate all of these features into a single protocol. Additionally, several new features were proposed, some simple such as throttling and flow-control, some not so simple, like quality-of-service. Finally, the ability to easily integrate with service-discovery systems was also proposed, although the service discovery system, in itself, would not be a direct part of the WMTP protocol.

The simplest way to do this would be to create a single protocol, using a design based on principles already implemented by a selection of preexisting protocols, while also cramming in the new features. This hypothetical protocol would effectively offer all of the proposed features, but not in a modular fashion, thus forcing the application to apply all of these features, even when not needed. This rigid design is clearly not practical, as the utilization of unnecessary features leads to the wasteful use of network resources and entails inevitable trade-offs that could otherwise be avoided.

Unlike before, a modular architecture would allow the application to choose which features it would need and which it could dispense, on demand. Ideally, the system would analyze the application's requirements and provide a solution that not only complied with them, but would also do so in the most efficient way possible.

To comply with this additional requirement of modularity, a simple requirement-matching system could be developed. This system would use a repository of several protocols of varying complexity, each supplying a list of the features it provides. The system would simply analyze the applications request and choose the right protocol from the pool, accordingly. This choice could be optimized to use the simplest protocol that still complies

with the application's request. Additionally, the system could base its decision not only a list of required features, but also on a list of optional ones, allowing the application to further participate in the process. This system, albeit functional, requires that a great effort be invested into the design and development of a large variety of sub-protocols that would go into the pool. Not only would the need arise to develop a sub-protocol for each individual feature, but also for every conceivable combination thereof. Otherwise, the system would be forced, like before, to apply unneeded features in a wasteful way. Additionally, the simultaneous use of several of these sub-protocols across the network could have unforeseen behavior, as it would be unconceivable to predict how each sub-protocol could interact with all of the others.

Another possible solution could be based on a layered approach. Each feature could be developed as a stackable sub-layer that would communicate with its upper and lower layers using a single common interface. Since all feature sub-layers would both use and provide the same interface to its upper and lower layers, several sub-layers could be modularly stacked together, on demand, to offer the features that were requested, while keeping out those that weren't. Although, on face value, this seems to be a good idea, this simplistic architecture entails several problems that are not trivial to solve. On one hand, the order in which the layers are stacked is not irrelevant and, thus, would require further study. On the other, this solution does not provide any additional mechanisms to isolate each feature's functionality from the others, in other words, each feature would have to be designed to take into account the possible effects of all the others. This not only off-loads an additional burden onto the feature development effort, but is also close-minded by nature, since it does not facilitate the design of new, unthought-of, features in the future.

Finally, a different solution was found that not only provides the required modularity but also does so in a way that both facilitates current feature development and also leaves room for future work to be done on new or improved features. This solution, the one that eventually was used in WMTP, is based on the creation of a transport layer *framework*. This framework is based on the existence of a central core that, by itself, provides only minimal functionality. The true features would be developed as additional modules or plug-ins that would interact with the core through very specific interfaces. These interfaces would be designed to minimize the possibility of unforeseen interactions between distinct modules, while also making it clear exactly what kind of interactions are predictable. This way, each feature module could be developed as a stand-alone feature, and yet, could still be used in conjunction with others. Additionally, this architecture would interact with the upper and lower layers using specialized interfaces that would enable additional features, although this might require the development of specialized "translation" modules or adaptation layers. Under these circumstances, the core's true functionality would reside in coordinating the overall activity, acting as the "glue" between all of the feature modules and the upper and lower layers, thus providing a coherent and integrated functionality that is presented to the application in a way that is as transparent as possible.

In order to provide the flexibility that WMTP's modular architecture requires, a specialized network protocol stack was designed. This unconventional protocol stack, illustrated in Figure 3.1, partially breaks the traditional layered paradigm that is commonly used in conventional network protocol stacks, since, contrary to what would be expected, WMTP does not sit neatly between the application layer and the network layer.

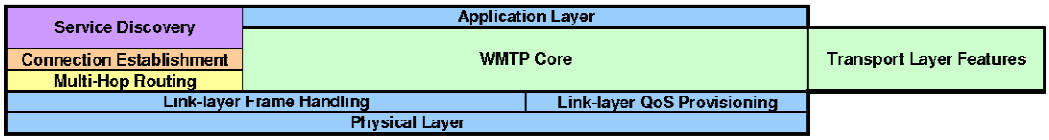


Figure 3.1: WMTP Protocol Stack

As illustrated above, the WMTP core actually resides directly between the application layer and the link layer. Although the network layer is not directly a part of WMTP, tight integration with the core functionality is required in order to provide some of the advanced functionality that WMTP's modular framework uses.

As for the interface with the application layer, WMTP was designed to meet the needs of typical WSN applications and thus this interface could not deviate excessively from the traditional interfaces typically used for this purpose. On the other hand, in order to make full use of some of WMTP's features, some extensions had to be employed. While the use of most of these extensions is not mandatory, the application would not be able to take full benefit of what WMTP has to offer without them. Although most applications shouldn't need to be recoded to use WMTP, some may benefit from a little remodeling, especially if they want to take full advantage of all the features the protocol can provide.

WMTP's interface with the link layer, in contrast, is much more traditional. The only extension that needed to be employed was an additional interface to obtain the layer's quality-of-service characteristics. On the other hand, this interface can be implemented by a dummy module that simply conveys the absence of quality-of-service or, alternatively, a more elaborated adaptation module that provides statistical assurance levels, based on the measured characteristics of the preexisting link layer protocol. Additionally, WMTP fully benefits from the promiscuous nature of the wireless medium by piggybacking any management information on forwarded traffic. On the other hand, if the link layer does not provide this feature, WMTP is flexible enough to still manage to operate, albeit not as efficiently, by using explicit management packets, thereby effectively creating a transparent graceful degradation mechanism.

As already previously mentioned, the multi-hop network layer needs to be tightly integrated into the WMTP protocol stack. This approach was followed due to the high level of collaboration that WMTP pursues with this layer, going beyond the scope of what the traditional layered approach envisions. Additionally, WMTP needs to manage its own core queue, which is a job that traditionally would be delegated to the network layer. Amongst other advantages of using this approach is the ability of the network layer to collaborate

with WMTP to implement connection oriented routing, a service that is a basic requirement of some of WMTP's features, namely the quality-of-service feature (which depends on the pre-reservation of network resources), and the fairness feature (which requires that each node have a previous context for each connection that passes through it). In order to provide this service, not only does the network layer have to identify to which connection does an incoming packet belong to, it also has to collaborate with WMTP to establish the connection in the first place. Additionally, this cross-layered approach also allows WMTP to use multiple concurrent network layer implementations with ease.

Although connection oriented routing has its benefits, the initial delay during which the connection is established, and its context is propagated across the network, can be undesirable under certain circumstances. An example of a situation where this feature would be a burden would be a highly mobile scenario, where a node may have to change routes very frequently, thus leading to the connection being broken and rebuilt. On the other hand, since connections require that each forwarding node maintain an individual context, this poses serious scalability problems, especially when large-scale deployments come into play. To overcome this problem, WMTP also supports connectionless routing modules which, albeit using the same interfaces as their connection-oriented counterparts, use a dummy connection establishment process that entails little or no additional delay. Additionally, in the absence of connections, the use of local memory is not proportional to the number of nodes that forward their data through the local node, thus also enabling large-scale deployments.

The connection establishment functionality, in turn, is directly related to the multi-hop routers in use. This functionality may also be integrated with more advanced naming and addressing schemes that go beyond WMTP's scope, or otherwise provide service discovery semantics. If the underlying router is connection-oriented, then this module will not only be responsible for discovering the routes to one or more destination nodes that fulfill the requested connection's criteria but also to establish and configure the connection's context. On the other hand, if the router is connectionless, then this module may simply be a stub for neighbor or gateway discovery.

Once this basic protocol stack is established, the specific features that characterize WMTP may be built upon it. Although this framework is designed to be flexible and to facilitate the implementation of additional features and improvements, the following feature set has been listed for initial development:

- Throttling: Packet generation is automatically throttled, thus relieving the application of this task.
- Flow Control: The receiving node may regulate the rate at which the source node generates its packets;

- Congestion Control: Packet transmission is delayed along the forwarding nodes, ultimately delaying packet generation, to avoid bottleneck node congestion;
- Fairness: Packet generation along the several sources that share a sink is throttled in order to divide the available throughput in either an equitable fashion or providing weighted differentiation;
- Reliability: Packets are automatically retransmitted, when losses are detected, in order to guarantee the reliable delivery of all packets;
- Quality-of-Service: The application may specify the minimum requirements and/or the desired values for certain QoS metrics such as end-to-end packet delay and packet throughput;

Although the WMTP protocol may support all of these features and even allow them to coexist peacefully on the same network, the user may wish to entirely disable unneeded features. This allows the partial compilation of only the components that are required for the provisioning of the remaining features, thus freeing resources that would otherwise be wasted.

3.2 Application Level Interface

In order to fulfill its purpose, WMTP offers the application developer a specialized interface that, while still maintaining some similarities with conventional packet dispatching interfaces, has some special nuances that must be carefully taken into consideration. Although the complete reference of this interface is provided in Annex 1, the most relevant commands and events are illustrated in Figure 3.2.

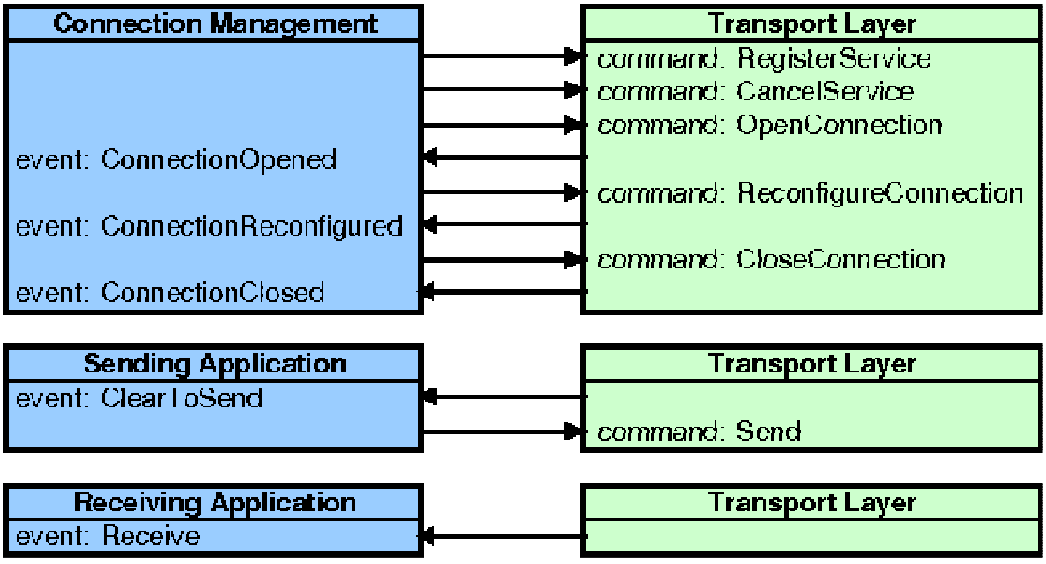


Figure 3.2: WMTP Application Level Interface

The application level interface is divided into three major parts. The connection management interface, the sending application interface, and the receiving application interface.

The connection management interface is responsible for all functionality that relates to the successful establishment of connections between nodes. If a connectionless router is in use, this functionality may be reduced to a mere stub or, eventually, neighbor and gateway discovery.

Basically, the application starts off by registering the service that it provides to the network through the RegisterService command. By doing this, the application is essentially opening the door to start accepting new inbound connections. This function call is also used to register any naming, addressing or service discovery conventions that may be used by other network nodes to find this particular service and connect to this application. Using a traditional, socket based, analogy, this command resembles the *bind* system call, as it associates an identifier that the network can recognize, to the application, the difference being that the identifying data is a more generic *service specification*, rather than a port number. Later on, the service may also be cancelled with the CancelService command, thus denying any future connections.

At the other end, the initiating application opens a connection with the OpenConnection command. This function call receives a *connection specification* that, amongst other possible options, specifies how the connection is established and routed, how each feature should operate on this connection, and any quality-of-service requirements that should be met. As a part of this connection specification, in some cases, a service specification may be supplied using the same basic format that was used with RegisterService command. This service specification, when applicable, can be used during the connection establishment phase to feed a service discovery mechanism and will also be used to connect to the appropriate remote application.

Once a connection has been successfully established, both ends will be notified through the ConnectionOpened event. This event also provides a new connection specification that is unique to the established connection and must be passed on to all further function calls related to it. Unless the connection specification is specifically configured to do otherwise, the OpenConnection command may lead to the establishment of multiple connections to several nodes that fulfill the specified criteria. If this is the case, the connection initiator, and not just the service provider, must also be prepared to handle multiple connections. When multiple connections are established, the ConnectionOpened event will be signaled for each one, providing a different connection specification each time.

Once a connection has been established, some of its parameters may be dynamically updated. This action is handled through the ReconfigureConnection command. Accordingly, the ConnectionReconfigured event is used to notify the application that the

remote end has reconfigured the connection and that the new parameters will be effective immediately.

Finally, either end may terminate a connection, on demand, by using the CloseConnection command. The ConnectionClosed event is signaled whenever the connection is terminated by any means other than a local call to CloseConnection, in other words, when the remote node explicitly terminates the connection, or when it is simply lost due to a broken wireless network link.

If a connectionless router is in use, the Connection Management interface must still be used, albeit with some additional peculiarities. This is necessary because WMTP virtualizes connectionless routing as a special case of connection-oriented routing that does not establish any initial context across the network. Nonetheless, this pseudo connection establishment procedure must still be followed so that WMTP may establish the local context associated with the connectionless data. Under these circumstances, the receiving application must register a service, using the same methodology followed by a connection-oriented application that accepts incoming connections. On the other hand, the sending application must establish a pseudo-connection using the same methodology a connection-oriented application would, to establish a traditional one. Soon after the OpenConnection command is called, the ConnectionOpened event will be signaled, thus providing the connection specification that must be passed on to any future function calls. The receiving application, in turn, will not be signaled with a ConnectionOpened event, since, under these circumstances, it is a mere passive listener that can neither send data back to the sources, nor have any influence on the configuration that is applied to any incoming packets. As usual, received packets are signaled with the Receive event.

The sending application interface, in turn, is much simpler and maintains a closer resemblance to traditional packet dispatching interfaces. Like any normal packet dispatching interface, there is a Send command that may be used to pass new data from the application layer on to the transport layer. The novel aspect in this interface is the ClearToSend event. This event is used by WMTP to notify the application when it should ideally provide its data, if it intends to conform to the restrictions implied by the connection's configuration. If all the applications running on each node in the network use this event to coordinate the rate at it which they generate data, then all the conditions will be met to provide the guarantees that each feature promises (i.e. avoid bottleneck node congestion, provide fairness, etc.). If, on the other hand, the application chooses to ignore this event, a certain amount of local queuing may absorb short bursts, but most features will probably not be able to guarantee the overall functionality that they were designed for.

The receiving application interface, on the other hand, is the simplest of all the interfaces. Just like any other packet receiving interface, there is one simple Receive event that is used to pass received data on to the application.

3.3 System Architecture

As previously explained, WMTP's functionality is based on the existence of a modular transport layer *framework*. In this section, this framework will be further broken down and explained in full detail. After an initial overview of how the system architecture works, each individual interface will be thoroughly explained. Then, all of WMTP's initially developed features will be systematically viewed, covering not only the basic mechanisms that back them up, but also the way they interact with the core, using one or more interfaces, in order to fulfill their ultimate purpose.

WMTP's basic architecture is composed of a common coordinating core and a collection of specialized modules that are linked to it using standardized interfaces (see Figure 3.3 and Annex 2) and that share with it a common set of data types (see Annex 6). These modules can use these interfaces to either implement a transport layer feature or perform a specific task that the core explicitly delegates (see Annex 3, Annex 4, and Annex 5). While feature modules will generally use a combination of the *Traffic Shaper*, *Reliable Transmission Hook*, *Feature Configuration*, *Connection Management Data Handler*, *Local Management Data Handler*, or *Core Monitor* interfaces to provide their functionality, the *Service Specification Data Handler*, *Connection Establishment Data Handler*, *Multi-Hop Router*, and *Link Layer QoS Indicator* interfaces were specifically designed to allow the core to delegate certain functions to external modules.

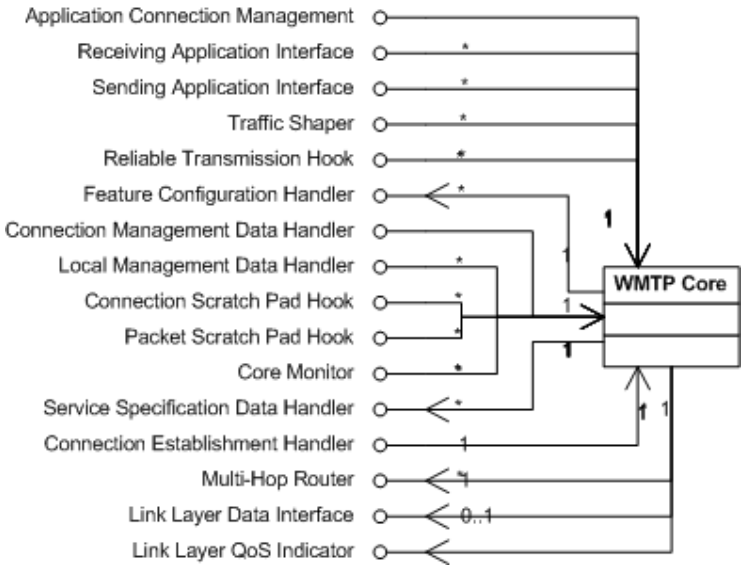


Figure 3.3: WMTP Core Interfaces

In the case of the *Connection Establishment Data Handler* and the *Multi-Hop Router* interfaces, the delegated tasks actually constitute WMTP's specialized network layer interface, used to outsource the specific functionality that belongs to this layer and, thus, goes beyond WMTP's scope. The *Link Layer QoS Indicator* interface, in turn, is the additional extended link layer interface that WMTP uses to obtain this layer's quality-of-

service characteristics. The remaining delegation interface, the *Service Specification Data Handler*, as will be further explained in the corresponding section below, is used to match together remote service data (requested interests), with local service specifications (offered services). Aside from determining if the local node is capable, or not, of accepting the incoming request, the core also uses this functionality to determine to which local application the incoming connection should be associated with. The use of external modules to provide this matching service is a key extension point that enables WMTP to work with more advanced naming, addressing, or service discovery mechanisms that go beyond its scope.

Finally, the *Connection Scratch Pad Hook* and *Packet Scratch Pad Hook* interfaces are additional helper interfaces that, albeit not directly used to regulate the core's behavior in a way that provides the desired feature's functionality, provide convenient tools that greatly simplify the development of feature modules.

3.3.1 WMTP Core Interfaces

3.3.1.1 The Traffic Shaper Interface

As previously mentioned, the traffic shaper interface is specifically designed to be used by WMTP feature modules. This interface basically allows these modules to regulate the rate at which packets are forwarded or generated. Although the interface provides a broader range of commands and events (see Annex 2), its basic operation is achieved through the use of specialized Start and Stop commands. These commands, in turn, can either be applied to individual packets, in which case they regulate when the packet is forwarded, or to connections, thus regulating packet generation at the application layer, through the ClearToSend event, as already seen in the sending application interface.

Additionally, multiple modules may use traffic shaping to influence the same packet or connection, in which case the core manages each module's state and uses an AND logic to only forward or generate packets once all relevant modules have sanctioned the operation.

The intelligent use of this interface is essential to provide many features. For example, when providing fairness, a traffic shaper may be used to throttle data generation, thus regulating its throughput. It is important to note that, due to the AND logic used to check if a connection or packet is clear to send, when multiple modules use traffic shaping, the resulting rate is limited by the "slowest" module. In other words, the result rate is approximately the minimum of all the rates specified by all of the traffic shaping modules.

3.3.1.2 The Reliable Transmission Hook Interface

The reliable transmission hook is a special interface provided specifically for reliability feature modules. Through a specialized set of commands and events (see Annex 2), this interface not only provides the means for these modules to tell the core when a packet must still be cached for future reference, and when it is no longer of any use, but also to detect repeated packets upon reception.

Unlike the traffic shaper interface, each packet can only be controlled by a single reliability feature module. In other words, although WMTP may have several reliability feature modules available, each individual packet may be controlled by at most one of them.

3.3.1.3 The Feature Configuration Handler Interface

As the name implies, the feature configuration handler interface is also specifically designed to be used by feature modules. This specific interface is used to allow each feature module to manage its own configuration, thus providing specialized commands that not only allow the module to initialize a connection's configuration with default values, but also to convert this configuration to and from a format that may be transported along the network (see Annex 2). This way, the WMTP core may use this interface to perform a connection's configuration across the network, as well as to make sure that individual connectionless packets are also appropriately configured.

3.3.1.4 The Connection Management Data Handler Interface

The connection management data handler interface, also specifically used by feature modules, allows these modules to piggy-back their own management data, alongside a packets payload, across the network. This is especially useful to append packet specific information, such as identifiers or sequence numbers.

This being the case, this interface provides both the means to generate and handle these headers, as well as to access them from within a currently enqueued packet (see Annex 2).

3.3.1.5 The Local Management Data Handler Interface

The local management data handler interface, like its connection management data counterpart, allows feature modules to generate and handle their own management data. Unlike connection management data, that is piggy-backed alongside a packets payload across the network, local management data is not associated with any particular packet and is, thus, repeatedly broadcasted to the local neighborhood. This particular kind of management data is especially useful to keep the neighboring nodes up to date on the local node's status (e.g. whether it is congested, or not).

This being the case, this interface provides the commands and events that allow the application to generate and handle these headers (see Annex 2). Additionally, the feature module is signaled when its data is being broadcasted, thus allowing it to update or delete it own data, immediately after it is broadcasted.

Although local management data is not directly associated with any packets, the WMTP core does not necessarily have to broadcast it in its own dedicated messages. If the link layer in use supports radio snooping, then the WMTP core will automatically piggy-back all management data onto some, otherwise unassociated, data packet. Since using the radio transmitter is, generally, one of the most expensive operations that sensor node's perform, in terms of energy consumption, the ability to piggy-back local management data along with data packets is a key factor in saving energy and, thus, extending the network's overall life-time.

3.3.1.6 The Connection Scratch Pad Hook Interface

The connection scratch pad hook is a special interface that allows multiple feature modules to maintain independent, per-connection, state variables locally associated to each open connection, thus relieving the individual feature modules from the burden of managing their own memory buffers for this purpose.

3.3.1.7 The Packet Scratch Pad Hook Interface

The packet scratch pad hook, like its connection counterpart, relieves the feature modules of the burden of managing their own state memory. Unlike the connection scratch pad hook, this interface provides per-packet state management, thus allowing feature modules to associate their own state variables to each individual packet.

3.3.1.8 The Core Monitor Interface

The core monitor is a special interface that allows any module to obtain the WMTP core's current status, as well as to be notified whenever this status is changed (see Annex 2). This interface provides the means to monitor registered services, open connections, and the core queue. Additionally, the monitoring module is notified whenever a packet is generated, received, sent or dropped from the core queue, as well as whenever a radio message is received, about to be sent, or when its sending process has completed.

3.3.1.9 The Service Specification Data Handler Interface

Service specification data handlers are the components that match together locally registered service specifications (locally provided services) with incoming service data (remote interests). By delegating this functionality to external modules, the WMTP core maintains its generality by not creating any strong ties to any specific

service discovery architecture. This approach avoids unnecessary cross-layer dependencies and eases the development of future extensions that may support more complex or advanced naming, addressing or service discovery systems.

Although this interface provides a broader set of commands (see Annex 2), its main functionality is centered on verifying if a local service specification matches a remote interest, as well as generating the data that expresses this interest in remote services, from an equivalent service specification object.

3.3.1.10 The Connection Establishment Handler Interface

Although some of WMTP's features rely on connections, the components that establish these connections don't have any strong ties to the core itself. In fact, when an application requests to open a connection, the request is simply rerouted to the appropriate connection establishment handling module. This module, in turn, then uses its own mechanisms to establish the connection as requested.

In order to accomplish this task, this interface provides several commands that the module may use during the connection establishment phase (see Annex 2). These commands provide the following functionalities:

- The core extends the functionality that the service specification data handlers offer it, thus allowing the connection establishment modules to generate service data representing interests, as well as to find local services that match remote interests;
- Similarly, the core extends the functionality provided by the feature configuration handlers, thus giving the connection establishment modules the ability to manage a connection's configuration. This way these modules are provided with the tools that enable them to both generate configuration data from an already configured connection specification object, as well as to initialize a new connection with remotely generated configuration data;
- The core itself must also be notified when a connection has been opened, not only so that it can maintain its own records up to date, but also to notify any other modules of the fact. This being the case, this interface also provides a set of commands that are used to notify the WMTP core that a connection has been established, be it either a locally terminated connection, or simple one that is forwarded over the local node. The WMTP core then uses these commands to update its lists of open connections, as well as to signal any other resulting events.
- The core provides the ability to generate dummy keep-alive packets for an established local connection. These packets, when dispatched, will be forwarded across the network towards the opposite end of the connection.

- Special commands are provided to interact with the WMTP core quality-of-service reservation subsystem. This subsystem will be further explained, in a section of its own, further below.

3.3.1.11 The Multi-Hop Router Interface

As previously mentioned, WMTP uses an unconventional protocol stack that doesn't sit upon the network layer, but rather uses it through a specialized interface. The multi-hop router is the interface used for this purpose.

The WMTP core uses this interface to query connection-oriented and connectionless routing modules alike, on how to forward its packets. The only difference between these two kinds of routing modules is whether they supply a connection specification object, alongside the response, or not.

This command is also used to generate and handle routing headers. This special kind of header is piggy-backed alongside the packet's data, in a way similar to connection management data (see Annex 7) and may contain information (e.g. a connection identifier or the destination node's addresses) that will be used by forwarding nodes as a part of the routing decision process.

Additionally, the core may use different routers for different packets, thus enabling the coexistence of several different routing schemes within the same network.

3.3.1.12 The Link Layer QoS Indicator Interface

The link layer QoS indicator is the specialized interface that the WMTP core uses to obtain the link layer's quality-of-service characteristics, namely the maximum time the WMTP core should have to wait between dispatching the packet to the link layer and the packet actually being sent.

Normally, a specialized module should be developed for the link-layer in use, in order to correctly reflect its behavior, since the absence of such a module will make the WMTP core assume that the link layer does not provide any quality-of-service semantics, thus disabling the entire quality-of-service reservation subsystem. On the other hand, WMTP also provides a special implementation of this module, the StatisticalQoSIndicator (see Annex 3), that, if used, measures the link layer's behavior in real-time and calculates a statistically assured maximum delay.

3.3.2 WMTP Core Functionality

As previously mentioned, the WMTP core is essentially the glue that sticks together all of the individual modules. Its job is to call upon the appropriate components for each packet or connection and to aggregate the information from multiple components of the same kind to make coherent decisions.

In this section, the specific functionalities, that the WMTP core is responsible for, will be explained in more detail. These functionalities are further categorized into several subsystems, each explained in its own subsection. These subsystems, in turn, are not isolated compartments within the WMTP core and are only presented here for the sake of clarity.

3.3.2.1 General Functionality

This section covers the functionality that is not directly associated with any subsystem in particular but is still a basic part the WMTP core. To be more specific, the core must handle the following duties:

- Maintain a list of open connections that feature modules may consult through their specialized interfaces;
- Signal any associated event handlers whenever any relevant event occurs;
- Implement each command or event specified by the core interfaces, either by supplying the desired functionality as described in the WMTP specification, or by providing a dummy response indicating that this particular functionality is not supported, when the specification explicitly allows this.

3.3.2.2 Message Generation and Parsing Subsystem

WMTP uses a specialized message format that supports the encapsulation multiple instances of local management data, as well as multiple data packets, each containing a routing header as well as multiple instances of connection management data and the packet's payload (see Annex 7). This particular subsystem is responsible for generating and handling these messages, thus entailing the following functionalities:

- Accept local management data from multiple feature modules and ensure that it is periodically broadcasted. If the link layer supports radio snooping and there are data packets ready to be sent out, piggy-back the local management data alongside the outgoing data packets, otherwise, use a dedicated management packet;
- Whenever a message is received from the link layer, sequentially parse its contents by calling the appropriate handling modules, be them local or connection management data handlers, multi-hop routers or applications;
- Whenever a new packet is generated by an application, call the appropriate multi-hop router to generate its routing header. Next call all of the individual connection management data handlers and append any data that they may generate, alongside the packet's payload;

- Handle any received connection management data by passing it along to the appropriate feature modules. Keep a record of the individual management headers in case the feature module requests it at a later time.

3.3.2.3 Data Forwarding and Delivery Subsystem

This particular subsystem is responsible for ensuring that data packets are sent across the network ultimately reaching their destination. This requires the following functionalities:

- Ignore received data packets that are not intended to be forwarded through this node (i.e. that were received promiscuously);
- Whenever a packet is generated by an application or otherwise received from another node, use the appropriate multi-hop router to obtain the next hop's address. If the specified next hop is the local node, deliver the data to the appropriate application, otherwise, enqueue the data so it may be forwarded.

3.3.2.4 Queuing Subsystem

The queuing subsystem, in turn, is responsible for holding a limited number of packets in local cache so that they may be forwarded at a later time. Additionally, this subsystem also cooperates with reliability modules by only dropping packets from the core queue when the appropriate module tells it to. This subsystem, thus, entails the following functionalities:

- Use the appropriate reliability module, when applicable, to detect repeated packets, and discard them.
- If a packet is configured to use reliability semantics, retain it in the core queue until the appropriate reliability module indicates that it may be dropped, even if it is destined for local delivery;
- Send the next packet from the core queue, as soon as the link layer signals that the previous packet has already been sent;
- When selecting which packet to send next, ignore all packets that have been marked, by at least one traffic shaping module, as inactive. Of the remaining packets, choose the one with the highest quality-of-service priority. If there are several packets with the same priority level or if all the packets don't have quality-of-service enabled, choose the oldest packet.

3.3.2.5 Traffic Shaping Subsystem

This particular subsystem is responsible for maintaining the Boolean state of each traffic shaper, associated to each connection and packet. Each traffic shaper can control its own state through the StartConnection, StopConnection, StartPacket, and

StopPacket commands, thus leaving the connection or packet in an *active* or *inactive* state, respectively. This way, this subsystem will provide the following functionalities:

- Maintain the individual active/inactive status of all open connections and queued packets for each traffic shaping module;
- Whenever a traffic shaping module starts a local connection, infer the global status using an AND logic and, if the connection changes its global status from the inactive to the active state, signal the appropriate application with the ClearToSend event.

3.3.2.6 Memory Management Subsystem

Since dynamic memory is a rare commodity in embedded systems, especially on the resource constrained ones used in WSNs, the WMTP core must manage its own memory for all of its dynamic structures. There are two data structures that require the WMTP core's direct action as a memory manager, core queue elements, and connection specifications. This memory management is achieved through the use of a basic list of idle elements from which a new element may be retrieved and put to use, on demand. Accordingly, once an old element is no longer needed, it may be "destroyed" simply by returning it to the list of idle elements.

In the case of the connection specifications, these elements are not only used internally by the WMTP core, but also by external modules, including applications. This way, these modules may use the GetNewConnectionSpecification and the DestroyConnectionSpecification commands to take advantage of the core queue's memory management abilities.

Additionally, the WMTP core provides two convenient interfaces that relieve feature modules of some additional memory management hassles, namely, the connection and packet scratch pad hooks. Although these interfaces do not require the use of dynamic data structures, the WMTP core must still maintain individual connection and packet scratch pads, for each module that uses them, associated to each open connection or queued packet, respectively.

3.3.2.7 Configuration Management Subsystem

The configuration management subsystem allows the WMTP core to manage the configuration of each feature module. Albeit most of the work is done by the feature module, the WMTP core is still responsible for the following functionalities:

- Whenever a connectionless packet is generated, the applicable configuration data must be appended using a special type of connection management data.

Accordingly, when one of these packets is received, its configuration must be initialized using said data;

- When a connection establishment handler requests the generation of connection configuration data, it should be pieced together by sequentially requesting that each feature module generate its own data;
- When a connection establishment handler requests that a connection be configured using configuration data, the WMTP core must first initialize the connection for every feature module. Next, each feature that is specified within the configuration data should be configured by calling the appropriate feature module.

3.3.2.8 Service Management Subsystem

Although the WMTP core does not specifically implement a service discovery system, it does use services to identify how each application is accessible from the network (in the same way that TCP and UDP use port numbers). This entails the following functionalities:

- The WMTP core must manage a list of locally registered services;
- When a connection establishment handler requests the generation of service specification data representing an interest, the WMTP core delegates this task on to the appropriate service specification data handler;
- Whenever remote service data, representing an interest, is matched against a registered local service specification, once again, the WMTP core must delegate the matching operation to the appropriate service specification data handler;
- When a connection establishment handler requests the first service specification that matches remote service data, representing and interest, the WMTP core must successively test each registered service specification until one that matches is found;

3.3.2.9 Quality-of-Service Reservation Subsystem

As previously mentioned, the WMTP core provides a quality-of-service reservation system to the connection establishment handling modules. This system extends the quality-of-service guarantees provided by the link layer, to provide end-to-end transport layer quality-of-service semantics.

The WMTP core quality-of-service reservation system is based on the idea of reserving a maximum sending delay for a connection, during its establishment phase, thus assuring that its packets will never have to wait more than the reserved delay to be forwarded to the next hop. As new reservations gradually take effect, the

system essentially makes sure that if a single packet from each quality-of-service enabled connection were to arrive from all of them at once, then each one could be dispatched within its individual required time frame.

This reservation process associates a simple priority level to the connection. This priority level, in turn, will be used by the queuing subsystem to prioritize the connection's packets, thus assuring that its service levels are met. Additionally, when quality-of-service resources are reserved for a connection, a single queue element is also reserved specifically for the connection, thus providing the additional guarantee that it will not be affected by network congestion.

In order to access this specific system, the connection establishment handling modules are provided with a specialized interface. Since the end-to-end quality-of-service reservation process is similar, in many aspects, to a transaction, it is often desirable to reserve and/or cancel quality-of-service resources prior to actually opening the connection. In order to provide these reservation semantics the core quality-of-service reservation system requires that a special procedure be followed when a quality-of-service enabled connection is being opened. This procedure is based on the following principles and is illustrated in Figure 3.4:

- The connection establishment handling module may request that the core reservation system calculate the shortest available delay, above a specified threshold, that it is willing to dispense, by using the GetQoSShortestDelay command. The request itself does not reserve any resources and its answer is only valid during the current context (i.e. until the handler returns). Additionally, the WMTP core may enact arbitrary policies that limit the minimum delay calculated by this function. This means that a quality-of-service reservation with a delay that is smaller than the one calculated by this function may still be accepted;
- Connections with local delivery do not need to reserve any resources since they can reach their destination without any delay. In this case, the connection may be opened directly with the AddLocalConnection command;
- Non-local quality-of-service enabled connections, on the other hand, must first reserve their associated resources through the ReserveQoSResources command. Once the quality-of-service resources have been successfully reserved throughout the entire route, the connection may be opened, using the AddLocalConnection or AddNonLocalConnection commands, as usual;
- The FreeQoSResources command must only be used to free the resources associated with unopened connections. The resources of an open connection are automatically freed when the connection is closed.

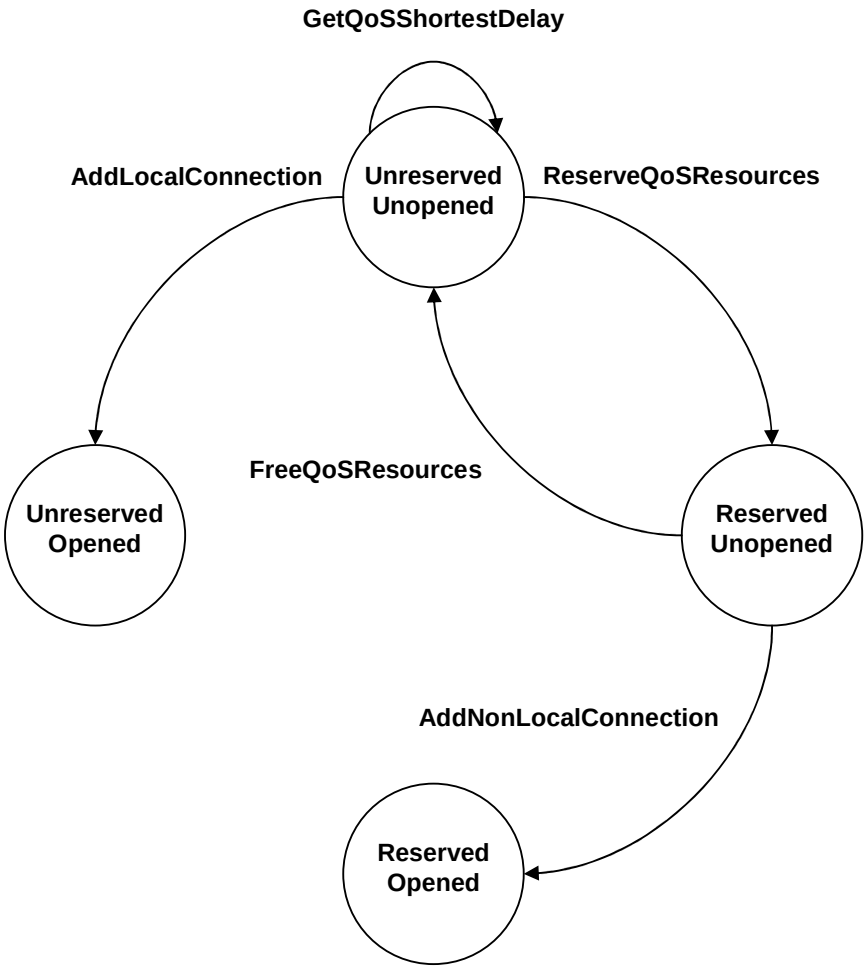


Figure 3.4: Quality-of-Service Reservation Procedure

Based upon this reservation system, the connection establishment handling modules may provide the following end-to-end quality-of-service metrics:

- Maximum end-to-end delay: The connection establishment handler sequentially attempts to reserve the shortest available delay on each node along the connection's route. These calculated delays are accumulated as the connection is established and, if, at a certain point, the specified maximum delay is exceeded, the connection is dropped before the establishment process finishes, thus guaranteeing that the connection is only successfully established if the quality-of-service requirements can be met.
- Maximum and desired generation periods: This quality-of-service metric is equivalent to the minimum and desired throughputs, but expressed as packet periods (in milliseconds between packets) rather than in throughputs (in packets per second). The idea behind this quality-of-service metric is the

guarantee that if the connection generates packets at a rate that is lesser than or equal to what was reserved, than its packets will never be dropped due to congestion. The use of two parameters, the maximum and the desired periods, enables the application to establish the hard requirement that must be met to establish the connection (the maximum generation period) and a soft preference, which ideally would be what is actually reserved. In order to provide this quality-of-service metric, the connection establishment handler sequentially attempts to reserve the shortest available delay that is larger than the preferred period, on each node along the connection's route. If the shortest delay is higher than the maximum period, then the connection establishment handler attempts to perform the reservation using the maximum period. If the reservation is ultimately unsuccessful, the connection is dropped before the establishment process finishes, thus guaranteeing that the connection is only successfully established if the quality-of-service requirements can be met.

Although the mechanism used by the WMTP core to manage the quality-of-service reservations is not standardized, the recommended solution, which was used in the reference implementation, is based on a binary reservation tree. The idea behind this mechanism is to represent each reservation as a node on the tree in such a way that, if a node is reserved, then none of its children nodes may be used for other reservations. Furthermore, the depth of the node (starting from zero at the tree's root), represents the connection's priority level, as will be used by the core queuing subsystem. Under these circumstances, a reserved node is guaranteed to be able to send out its data with a delay of $(2^n+1)\times D$, or less, where n is the connection's priority and D is the maximum delay reported by the link layer.

In order to prevent a delay constrained connection from tying up the reservation system and cutting off any future reservations, the WMTP core may also enforce a minimum level that it is willing to offer. Using this policy, whenever a connection establishment handler requests the shortest available delay, the core quality-of-service reservation system will never provide a value below $(2^n+1)\times D$, where n is the minimum level, dictated by the policy, and D is the maximum delay reported by the link layer. Although this policy does limit the values that are reported by the GetQoSShortestDelay command, it does not interfere with the reservation process in itself, thus creating the situation where a quality-of-service reservation with a delay that is smaller than the one calculated by this function may still be accepted.

An example showing this reservation system in action is illustrated in Figure 3.5.

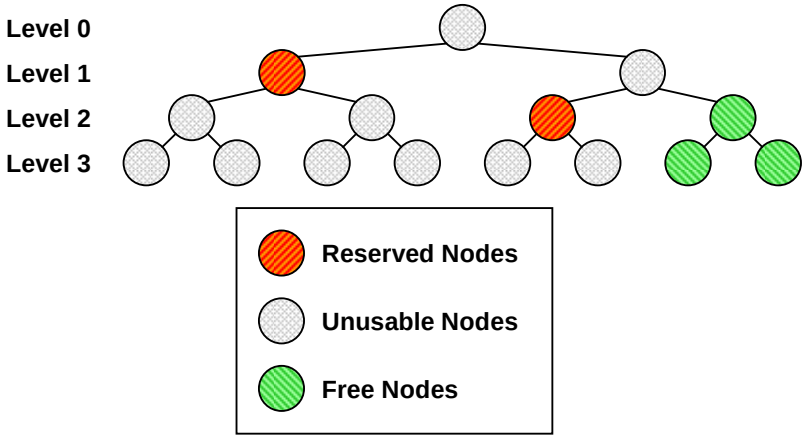


Figure 3.5: Example of the Binary Tree Reservation System in Action

This particular example illustrates a situation where there are two quality-of-service reservations. For the sake of clarity, the maximum send delay that is reported by the link layer is one second. This being the case, the following conclusions may be drawn:

- The quality-of-service reservation that was made at level 1 is guaranteed to have priority over all others. This means that, when a packet is received for this connection, it will be forwarded as soon as the previously sent packet is complete. This way, the packet will have to wait at most two seconds to be forwarded, the maximum time it may take to finish the previously sent packet and the maximum time it will take for it to be sent itself. This value is smaller than the predicted maximum delay of $(2^n+1)\times D = 3$ s.
- The quality-of-service reservation that was made at level 2, on the other hand, must yield priority to the reservation made at level 1. This means that, when a packet is received for this connection, it will not only have to wait for the previously sent packet to finish, but also for the packet associated with the reservation at level 1, as well as the packet associated with a possible future reservation made at level 2. This way, the packet will have to wait at most four seconds to be forwarded, the maximum time it may take to finish the previously sent packet, the higher priority packet, a possible equal priority packet, and itself. This value is smaller than the predicted maximum delay of $(2^n+1)\times D = 5$ s.
- The remaining free nodes may support either one reservation at level 2 or two reservations at level 3. In the former case, as before, the maximum delay would be five seconds. In the latter, the maximum delay would be $(2^n+1)\times D = 9$ s.

3.3.3 Message Formats

In order to allow nodes to communicate amongst themselves and properly implement the above mentioned functionality, a special message format is used in WMTP. In this section WMTP's message format will explained but, for a complete specification of all of the message structures and fields, see Annex 7.

All of WMTP's messages are based on a single super-structure, the *WMTP Message*, which uses a hierarchical approach to encapsulate all of the different kinds of data into a single unit. Although this is not uncommon in other protocols, there is one key aspect that differentiates this approach from traditional ones: when a sub-section is included within a section, the sub-section's size is not saved within the section itself and may not be sent anywhere within the packet at all. The idea behind this is that most sections have either a constant size or a size that can be trivially calculated; in which case, adding an explicit field with the section's size is just redundant overhead that may be used only for validation purposes.

This way, when the WMTP core calls a handler to process a specific section of a message, instead of passing it a data buffer and its size, it passes a data buffer and receives the size when the call ends, which, in turn, will be used to locate the beginning of the next section. This concept is used extensively throughout WMTP's message format scheme.

As already mentioned, the WMTP message uses a hierarchical structure. This relatively complex structure is necessary not only to be able to hold data packets and headers, but to also be able to piggyback management headers. Having this in mind, the WMTP packet may be further broken into multiple *Local Parts*. These parts may each hold either a local management header or a *Connection Local Part,* which contains a data packet alongside any connection specific headers. These *Local Parts* are appended sequentially within the *WMTP Message*, leaving the *Connection Local Parts* for last. The *Connection Local Part*, in turn, may be further broken into a *Routing Header*, followed by multiple Connection Parts and, finally, the *Data Connection Part*, which holds the actual data payload.

3.3.4 Feature Implementation

Using the previously explained architecture, complete features may be implemented as additional modules that use multiple interfaces to interact with the WMTP core, in a way that provides a coherent result. In this section all of WMTP's initial features will be presented and further explained.

3.3.4.1 Queue Availability Shaper

The queue availability shaper is not exactly a formal transport layer feature, but rather a convenient utility. This module prevents the application from generating packets when the core queue doesn't have enough room to accept them in the first place. This may seem like an oxymoron, but, in the absence of this feature, the application could be led to generate packets at a rate higher than the core queue could absorb them, thus leading to all surplus packets being dropped immediately after being generated.

To implement this mechanism the module needs only to use the traffic shaping and the core monitor interfaces. Whenever a packet is generated, received or dropped from the core queue, the queue availability shaper uses the core monitoring interface to check if the core queue still has any space available for an additional packet and starts or stops all connections that use this feature, accordingly.

3.3.4.2 Throttling

Throttling is one of the simplest of all features that WMTP offers, being basically a mechanism that allows the application to specify the minimum packet generation period.

To implement this mechanism the module needs only to use the traffic shaping and the connection scratch pad hook interfaces. Whenever the connection generates a new packet the throttling module basically stops the connection and sets a wake-up timer to restart it, after the specified period. The WMTP core will eventually use the ClearToSend event to generate the next packet. The connection scratch pad hook interface is used merely for the convenience of storing each connections individual wake-up time.

3.3.4.3 Flow Control

The flow control feature, in turn, is a simple mechanism where the receiving node may regulate the rate at which the sending node generates data. This functionality is achieved in exactly the same way as was done with the throttling feature, except that the desired generation period is configured by the remote application, rather then the local one.

3.3.4.4 Congestion Control

Congestion control is a feature that delays packet forwarding in order to avoid congestion on bottleneck nodes. As these packets accumulate across upstream nodes, the sources will eventually be affected, thus regulating packet generation as well.

This feature is implemented using the local management data handler, the traffic shaper, and the core monitor interfaces. Each node determines its own congestion status by using the core monitor interface to analyze its local queue availability, thus creating a Boolean congestion notification bit. In order to decide whether the nodes is congested or not, two distinct thresholds are used, a minimum queue availability, under which the node will be considered as congested, and a maximum queue availability, over which it will cease to be considered as so. In between these two thresholds, the node retains its last state, thus creating a memory effect.

Once this congestion status is established, this information is shared with all local neighbors through the use of local management data. Furthermore, each of the local neighbor's congestion state is cached in a local neighbor table. This way, the traffic shaping interface may be used to stop packets that will be forwarded over to a congested node and restart them when the node is ready. Additionally, whenever the local node changes its congestion state, all local connections are stopped or started, accordingly, thus regulating the rate at which the application generates packets.

A message sequence diagram, illustrating this feature in action, may be seen in Annex 8.

3.3.4.5 Fairness

The fairness feature, in turn, allows all connections that are transmitting to a common sink to share the available network resources, either in an equitable fashion, or using a weighted differentiation algorithm.

To provide this feature the module must use the local management data handler; the traffic shaper, the connection scratch pad hook, and the core monitor interfaces. The basic idea is that the module measures how much time passes between packets being dropped, thus determining the local period. This local period is then smoothed out with an exponentially weighted moving average and multiplied by the sum of the weights of all open connections that go through the node and use fairness, resulting in what is called the normalized local period.

Additionally, each node broadcasts a normalized period and the address of the limiting node. More specifically, each node starts by broadcasting its own local normalized period, setting itself as the limiting node. As this information is shared amongst neighbors, each node remembers the highest remote normalized period it has heard of recently and that wasn't limited by itself, as well as the address of the limiting node. If its own local normalized period is greater than or equal to the cached remote version, the node continues to broadcast its own version, while still setting itself as the limiting node. If, on the other hand, the remote normalized period is greater, the node will broadcast the remote version, while setting the remote

address as the limiting node. What this does, in practice, is implement a distributed algorithm that determines which node has the highest normalized period and what its value is, while also avoiding the creation of dependency loops within the network.

Now that each node knows the value of the highest normalized period of the entire network, it uses this value to determine the packet generation period of each of its local connections by dividing this normalized period with the connection's own weight. Once this packet period is determined it is slightly boosted using a multiplicative factor (80% in the reference implementation), and the connection is throttled through the use of a mechanism identical to the one used in the throttling feature.

This boosting mechanism is used to help the fairness feature react faster to occasional changes in the network's characteristics. If it were not used, the fairness feature would still effectively and quickly react to a radio link slow-down but, on the other hand, it would have a very slow reaction if the link were to speed-up.

All of the above mentioned functionality is further replicated to provide multi-sink fairness. In other words, each state variable is instantiated for each registered sink, a distinct local period is measured for each sink, and the local management data is used to broadcast multiple [normalized period, limiting node address] pairs. This way, several fairness enabled sinks can operate independently across the network. Additionally, the core monitor interface is used to detect when a *Sink ID* service is registered, thus creating an identified sink to which sources may route their data.

3.3.4.6 WMTP Reliability

WMTP reliability is a feature that enables packet retransmission in case of loss, thus providing an additional level of assurance that a packet will reach its destination. This feature uses a link level reliability mechanism in which a packet is only retransmitted over the local link where it was lost, instead of an end-to-end solution, where lost packets are only repeated at the source nodes.

This feature is implemented through the use of the local and connection management data handler interfaces, as well as the reliable transmission hook, the traffic shaper, and the packet scratch pad hook interfaces.

The connection management data handler is used to associate a simple identifier to each data packet that uses this feature. The local management data handler, on the other hand, is used to broadcast an availability map (A-Map), in other words, a list of the identifiers of each packet that uses this feature, currently held in the local queue.

Now that this feature has a way of identifying its individual packets and of telling its neighbors which packets it currently has in its core queue, it uses the reliable transmission hook interface to hold these packets until it is certain that they won't be

needed any longer. This way, a packet is considered droppable when the next hop has already announced having received it and the previous hop no longer announces that it is retaining it. Naturally, the sending node needs only to wait for the former condition and the receiving node, the latter. Additionally, this specific per-packet state is easily maintained through the packet scratch pad hook interface.

On the other hand, this feature must also make sure that a packet is only retransmitted if some benefit may come from it. In other words, the packet must not be retransmitted if the next hop has already acknowledged it and it should only be retransmitted after a certain amount of time has elapsed, thus giving the next node enough time to broadcast its availability map. This waiting period is implemented through the use of the traffic shaper interface, by stopping the packet as soon as it is transmitted and only restarting it after its time out timer has expired. Once again, the packet scratch pad is used to manage these per-packet timers. Some simple examples showing this mechanism in action are provided in Annex 9, Annex 10, and Annex 11.

A special note must be made as for the availability map's format. Since each packets identifier is relatively large (32 bits in the reference implementation), and since the core queue may have several packets with WMTP reliability, the availability map may become quite large, which may pose a problem, especially since the maximum packet size on WSNs tends to be quite small (29 bytes, by default, in TinyOS). In order to circumvent this problem, the availability map must be segmented into several local management data messages, each with a relatively small size (12 bytes in the reference implementation). This segmentation, in itself, also poses a problem of its own. Since not only the presence of a packet in the availability map, but also its absence, conveys information, the use of segmented availability maps no longer transmits this information atomically, and thus could generate confusion as to whether a packet is absent or if it is present but just not included in this fragment. To work around this problem, WMTP reliability builds its availability maps by using a special packet sorting algorithm that clearly conveys the absence of a packet, as well as its presence. This is done by ordering the availability map's packet ids in an ascending order and by adding special dummy packet ids to represent the list's beginning and end. Once the availability map is arranged in this fashion, it may be fragmented and sent out in multiple pieces, so long as the last packet id from one fragment is included as the first packet id in the next one. Since the packet ids are ordered, the absence of one may be detected as either the presence of two consecutive packet ids, one smaller the absent one, and one larger, or as the indication that the first packet id is larger than the absent one, or that the last one is smaller. Some examples of how availability maps may be fragmented, using this algorithm, are illustrated in Figure 3.6.

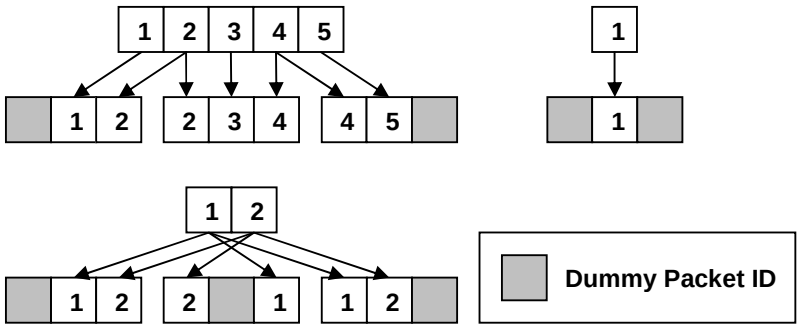


Figure 3.6: WMTP Reliability Availability Map Fragmenting Examples

3.3.5 Additional Modules

Aside from the core and the feature modules, WMTP also provides an additional set of modules that provide other kinds of functionality. In this section these additional module will be individually presented and explained.

3.3.5.1 PacketSinkServiceSpecificationHandler

This module uses the service specification data handler interface to manage the *Packet Sink* service specification type. This type of service specification is used to describe a generic catch-all sink. In other words, a *Packet Sink* interest will always match a registered *Packet Sink* service.

3.3.5.2 SinkIDServiceSpecificationHandler

This module also uses the service specification data handler interface but, this time, to manage the *Sink ID* service specification type. This type of service specification is used to describe the identified sinks that are used with the fairness feature, by using a seven bit sink identifier. This being the case, a *Sink ID* interest only matches a registered *Sink ID* service if they have the same identifiers.

3.3.5.3 TOSMultihopRouter

This module, unlike the previous two, is a routing module. In other words, it provides the Multi-Hop Router interface for the WMTP core. This routing module uses TinyOS's native routing layer to provide a simple connectionless router that can be used with WMTP. Once the data reaches the sink node (defined as node 0 by TinyOS) the TOSMultihopRouter module searches for a local *Packet Sink* service and delivers the data to the associated application.

3.3.5.4 TagRouter

This module, just like the previous one, is a routing module and, thus, provides the Multi-Hop Router interface for the WMTP core, but, unlike the TOSMultihopRouter module, this routing module provides connection oriented routing.

This router works by associating to each data packet a routing header that contains a simple eight bit tag. This tag is then used to identify which connection the packet is associated to, as well as the address of the packet's next hop. Additionally, this tag is translated, over every hop, in a manner similar to what is done in ATM and MPLS technologies.

In order to provide this functionality, this module requires an external Connection Establishment Handling module that adds the local context with the tag associations. This being the case, this module also provides an additional specialized interface that can be used by Connection Establishment Handling modules to manage this local context. Although this interface has additional commands and events (see Annex 5), its main functionality is provided through the AddTagAssociation command. This command creates a new tag association, which maps the previous hop address and previous tag to the next hop address and next tag, while also associating a connection specification object. Additionally, this tag association may be used in bidirectional connections, as the translation mechanism is bilateral.

3.3.5.5 SourceRoutedConnectionEstablishmentHandler

This module, in turn, is used to establish source routed connections. In other words, it uses the connection establishment handler interface, provided by the WMTP core, as well as the specialized interface provided by the TagRouter module, to establish a connection's context across the network. Additionally, this module uses a source routing mechanism that allows the application to dictate exactly which route the connection should follow.

In order to establish these connections, this module uses local management data to send out connection initiation messages (see Annex 7). These messages, in turn, not only contain the information required to establish the tag routing context between neighboring nodes, but also a list of all of the remaining hops required to reach the packet's destination, as well as the configuration data that is used to establish the connection's context, the service specification data that expresses an interest that must be matched at the destination node, and, finally, any quality-of-service parameters that are used to reserve these kinds of connections. Once it is in the possession of all of this information, this module simply uses the previously mentioned functionality that the WMTP core and the TagRouter module provide.

3.3.5.6 StatisticalQoSIndicator

This module, unlike the previous ones, provides statistically inferred quality-of-service characteristics of a link layer, through the Link Layer QoS Indicator interface. This is particularly useful when the existing link layer does not explicitly support quality-of-service, or when its true characteristics are, otherwise, unknown.

In order to provide this functionality, this module uses the Core Monitor interface to establish how much time elapses from when each packet is passed to the link layer and when the packet is completely sent. Once in the possession of these individual measurements, the module then performs a simple statistical analysis that calculates a maximum expected delay that is larger than the observed delays approximately 95% of the times.

Since a complex statistical analysis, based on a confidence interval, would require more memory and processor resources than is considered reasonable for an embedded sensor node, an alternative method was used, based on exponentially weighted moving averages. This way, the maximum expected delay is calculated using the following methodology:

- An average delay is calculated by smoothing out individual delay measurements with an exponentially weighted moving average that attributes 5% to new values;
- After the average delay is updated, a current deviation is calculated as the absolute value of the difference between the current delay value and the average;
- An average deviation is calculated by smoothing out individual deviation values with an exponentially weighted moving average that attributes 5% to new values;
- Finally, the maximum expected delay is calculated as the average delay plus three times the average deviation.

Simulations have shown that this maximum delay estimator provides a relatively stable value that is greater than or equal the individual delay measurements, approximately 95% of the time.

3.4 Implementation Considerations

Concurrently with WMTP's design, a reference implementation was also developed. This parallel effort not only served the purpose of demonstrating that such an implementation was conceivable but, in the end, also produced a complete and functional system that could be used to validate the protocols functionality, as well as to evaluate its performance.

This reference implementation was developed for the TinyOS platform and, thus, benefits from its modular component based architecture. Under these circumstances, each of WMTP's modules was cleanly mapped into an individual TinyOS component that could be easily included or excluded from the compiled binary through the simple manipulation of a configuration file. Using this approach, it is easy to quantify exactly how much program and data memory is used by each feature. This way Table 3.1 shows the actual values that apply to the MICAz platform.

	ROM Code Size (Bytes)	RAM Footprint (Bytes)
TinyOS + Core	30974	3268
Queue Availability Shaper	470	21
Throttling	1664	119
Flow-Control	1992	139
Congestion-Control	1908	73
Fairness	5268	151
WMTP Reliability	5054	152

Table 3.1: WMTP Feature Memory Usage

An additional advantage of the TinyOS platform is that the same source code that implements WMTP on real sensor nodes can also be run through the TOSSIM simulator. This way, the simulation process is actually running a complete implementation of the WMTP protocol, rather than a basic model that emulates its functionality. On the other hand, this also provides a very convenient development environment, since the simulated application can be further debugged through the use of traditional toolkits like the GNU debugger.

Although the TOSSIM simulation environment mimics very closely real-world sensor networks, there is one key aspect where it fails to emulate a sensor node's true behavior. Since TOSSIM is an event based simulator it cannot quantify the time the real embedded processor would take to execute a certain portion of code. This way, although the reference implementation is guaranteed to compile and fit within the limited memory restrictions found on these nodes, it is yet to be asserted whether or not their limited processing capabilities impose any significant restrictions on WMTP's functionality.

4 WMTP Test and Evaluation

To assess WMTP's effectiveness, a special test application was developed to work with WMTP while being simulated under the TOSSIM environment. This simulator was built to reproduce real-life WSN conditions very closely and, as such, simulates multiple nodes arranged according to a specific topology.

Under these conditions, a series of simulations were run to show each individual feature in action, as well as the most relevant combinations thereof. Since these simulations were designed solely to show how WMTP's features operate, in a qualitative manner, each test case was run only once. Being this the case, the simulated results may only be used as a comparison between distinct feature combinations or scenarios, since they lack the statistical relevance required to extrapolate reference performance values.

4.1 Test Scenarios

Given the modular nature of WMTP's functionality, a common base-line scenario was created in order to be able to successfully assess each feature's effectiveness. Using this common ground, all of WMTP's features can not only be evaluated by themselves, but also compared amongst each other.

This kind of comparison amongst different features is especially useful when evaluating the combined effects of multiple features. This way, the positive or negative effects of any single feature, or combination of features, on any of the others may be assessed.

The test scenario chosen for this purpose is described in Figure 4.1. In this scenario, there are a total of six sensor nodes, of which three are source nodes, which generate data, two are relay nodes, that simply forward the data, and one is a sink node, which consumes the data.

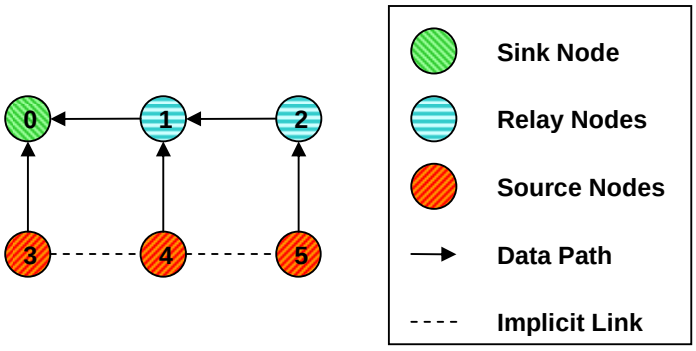


Figure 4.1: Common Simulation Scenario Topology

This particular scenario was chosen due to the differing distances of the sources from the sink. This simple difference is enough to create an imbalance that naturally leads to an

unfair advantage for the source nodes closest to the sink. By using this additional challenge for WMTP's features to overcome, the use of a larger and more complex topology would be redundant, since it would only add more nodes under similar conditions. This being the case, the use of only these six simulated nodes suffices to effectively validate WMTP's functionality and evaluate its performance, while also avoiding excessively cluttering the results.

A special exception was made from this common scenario for the quality-of-service assessment simulations. Since quality-of-service is supposed to provide assured performance levels, even in the most unfair of environments, a special scenario was used for these tests. The scenario chosen for this purpose, illustrated in Figure 4.2, simply transfers the responsibility of generating data from node 3 to node 1. Under these conditions, if node 1 operates under no special restrictions, it will generate data at the highest rate physically possible, thus ensuring that its core queue will always be completely filled with its own packets. On the other hand, since node 1 is also a forwarding node for the remaining source nodes, under natural conditions, it will completely prevent any data from the remaining sources from getting through to the sink.

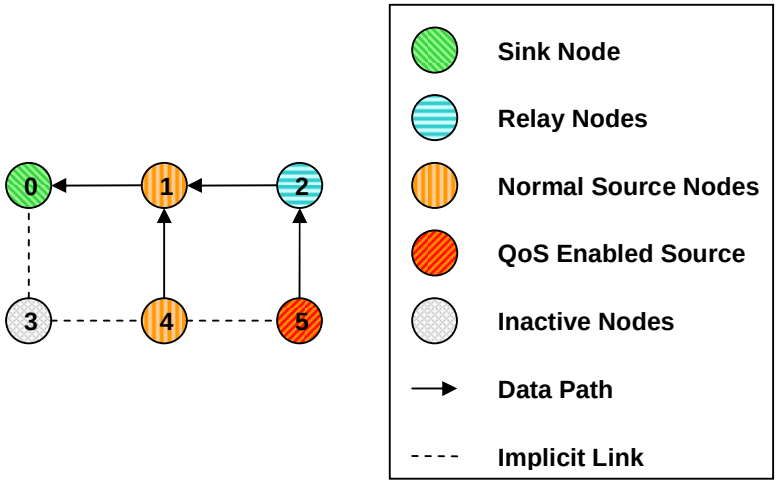


Figure 4.2: Quality-of-Service Simulation Scenario Topology

Under these exceptionally dire conditions, node 5 will establish a quality-of-service enabled connection, in hope of being able to overcome the challenge that it is presented with.

4.2 Test Application and Methodology

In order to produce meaningful results from the test simulations, a special test application was developed to establish connections and generate data while using the WMTP features under the above specified scenarios. Additionally, a performance monitoring module was also developed, in order to measure certain specific metrics, such as the number of data

packets and radio messages generated and received, as well as the minimum core queue availability during the measured period. This performance monitor module then periodically dumps and resets its statistical counters every 10 seconds, thus providing the raw data that is further processed using a traditional spreadsheet application.

With the exception of the base line scenario test-case and the WMTP reliability test-case, which were tested under varying network load conditions, the remaining tests cases were designed to show how their respective features operated under pressure. This being the case, the source nodes in these simulations generate data at the highest rate that they physically can.

The quality-of-service test case also presents a distinct exception from this rule. Although the non-quality-of-service enabled nodes, within the test, do also generate data at the highest rate that they physically can, the quality-of-service enabled node initiates a throughput constrained connection and thus generates data at a constant packet rate.

4.3 Simulation Results

In this section, the processed simulation results will be presented and briefly discussed. The following indicators have been used across multiple simulations:

- *Generated Packet Rate*: As the name suggests, this value represents the average number of packets each node generated, measured in packets per second;
- *Received Packet Rate*: This value, as the name also clearly suggests, represents the average number of packets received at the sink from each source, measured in packets per second;
- *Unreceived Packets*: This value, in turn, represents the accumulated number of packets that have been sent from each source but, for some reason or another, have not yet reached the sink. It is perfectly normal for this value to be non-zero, even if all packets are eventually received, as this merely indicates a delay between the moment the packet was generated and the moment at which it was received at the sink. True packet losses are indicated by this indicator's growth tendency during relatively long time spans;
- *Minimum Queue Availability*: This value indicates the lowest queue availability reached by all nodes in the network. The queue availability can be seen as the opposite of queue occupancy, and represents the number of packets a node may still receive before it will be obliged to drop incoming data. This indicator is specifically useful for evaluating the network's congestion state.
- *Overhead*: Finally, this value represents the overall percentage of transmitted radio messages that were not directly associated to a received packet. Both explicit management messages and lost data packets contribute to increasing this

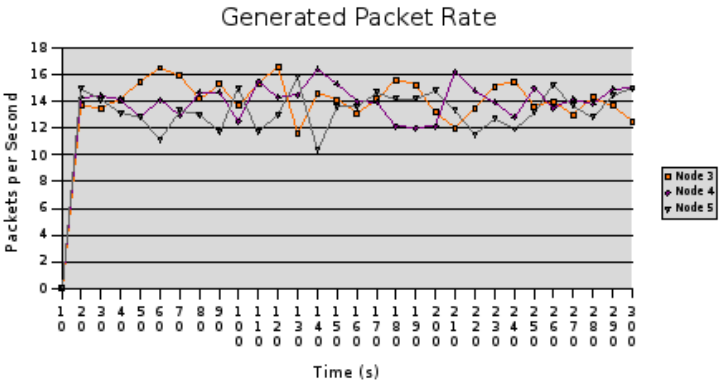
ratio. This value is particularly useful to evaluate a feature's energy efficiency, since transmitting radio messages is one of the most expensive operations that a sensor can do.

4.3.1 Base-line Scenario

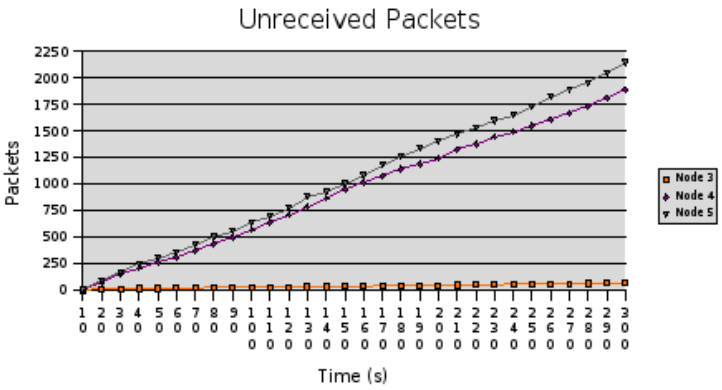
The base-line scenario is the basic case where all source nodes generate data at the highest rate that they physically can. Since neither congestion-control, nor fairness is in use, this scenario shows how the common base-line scenario reacts under natural conditions.

This scenario should represent the lowest possible level for WMTP's performance, since the protocol is making no efforts whatsoever to improve the situation. The use of any of WMTP's features should do nothing but improve on this situation.

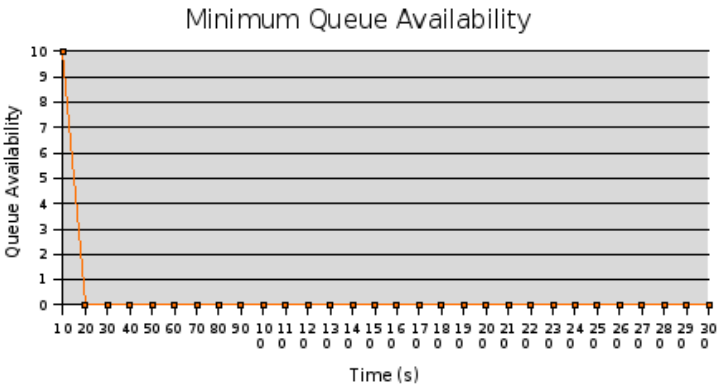
A.



B.



C.



D.

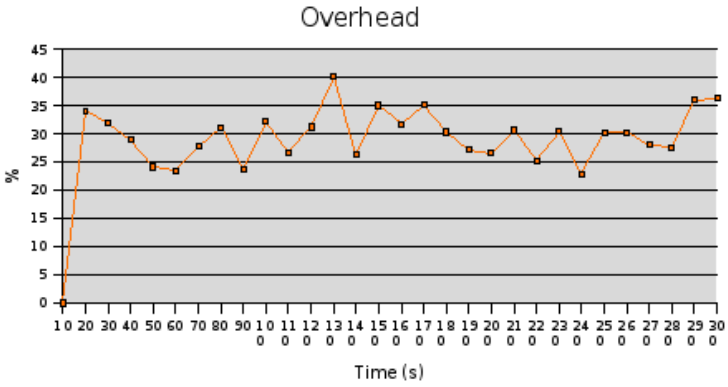


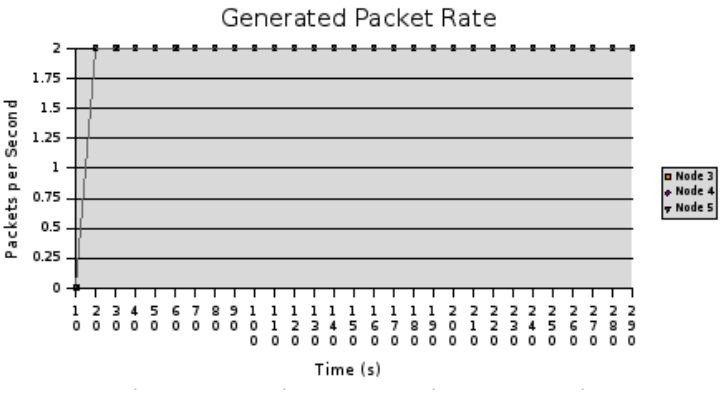
Figure 4.3: Baseline Simulation Results

The results clearly show the natural imbalance present in the common base-line scenario, as Figure 4.3.B indicates that the source node closest to the sink, node 3, hardly suffers any packet losses, while its peers have nearly half of their packets lost along the way. Additionally, Figure 4.3.C holds a constant value of zero, indicating a constant state of severe congestion throughout the network, thus justifying the massive packet losses suffered by the two source nodes that are not in direct contact with the sink. These lost packets, in turn, represent a useless waste of energy, thus justifying the large overhead that can be seen.

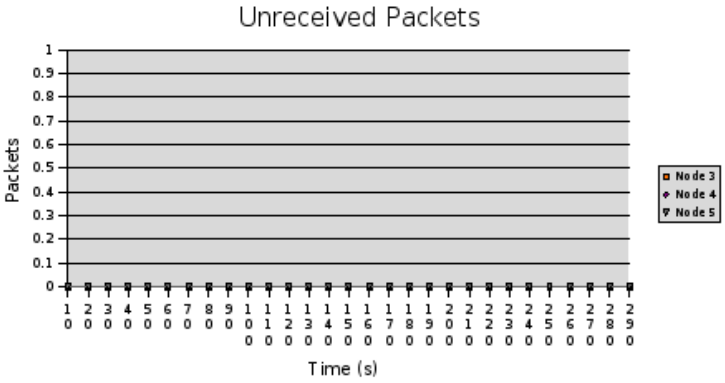
4.3.2 Throttling and Flow-control

Throttling and flow-control are the two most basic features that WMTP has to offer, and these simulations reflect that. The only difference between the two is which node decides at what rate the data is generated. While, for throttling, the source nodes set their own data generation rate, in flow-control, the sink ceases this privilege. For both cases, a data generation rate of two packets every second was established.

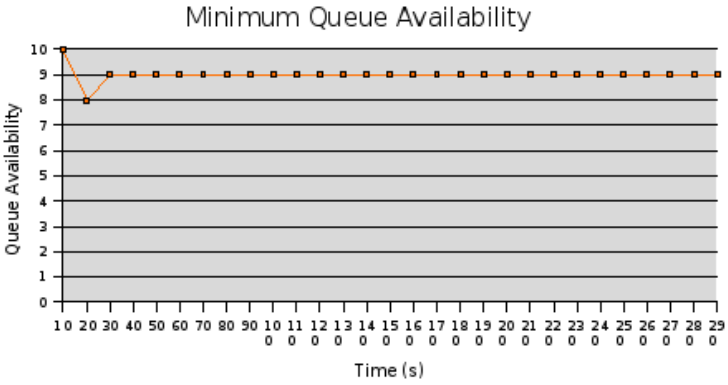
A.



B.



C.



D.

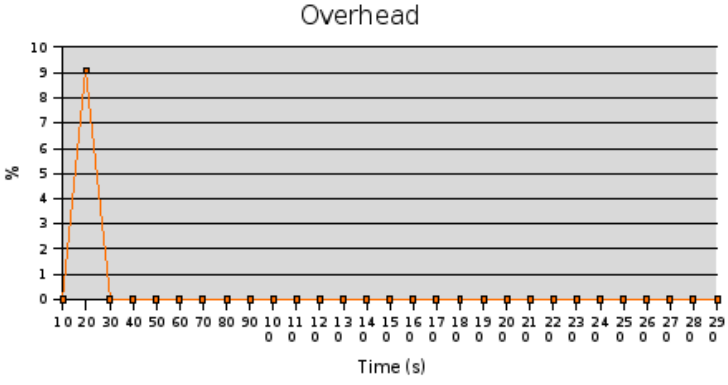


Figure 4.4: Throttling Simulation Results

These results clearly show that the throttling feature works as expected, with the further advantage of using no additional overhead once the connection has been established. On the other hand, this test is also a useful insight into how the common base-line scenario operates when not under extreme load conditions. Additionally, Figure 4.4.C shows that the network is no longer congested thus avoiding the massive packet losses and overheads from before.

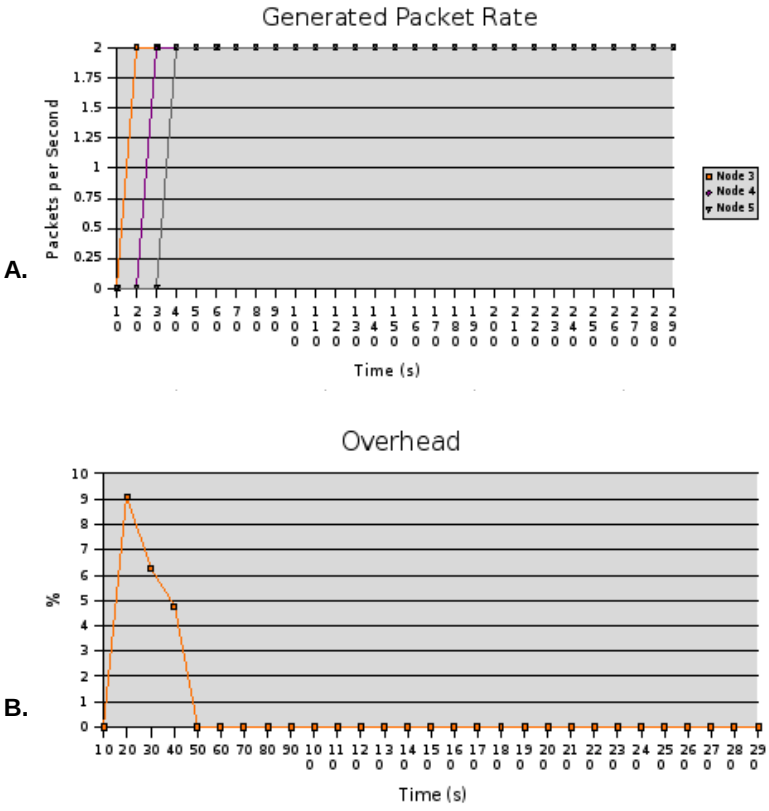


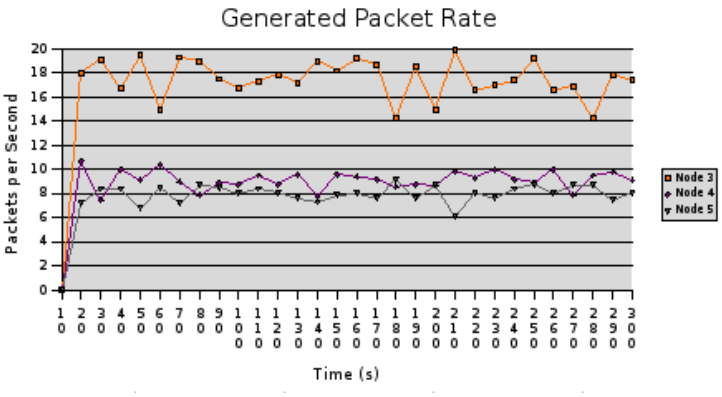
Figure 4.5: Flow-control Simulation Results

These results, in turn, also confirm that the flow-control feature operates as expected, since the expected packet generation rate is met and, as before, there are no additional overheads once the connection has been established.

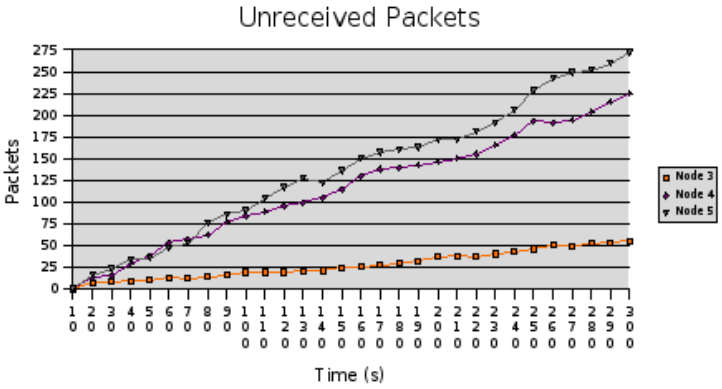
4.3.3 Congestion-Control and Fairness

The congestion-control and fairness features, like throttling and flow-control, are both traffic shapers. But, unlike their more basic siblings, these traffic shapers regulate how data is generated and forwarded based on the network's current state. The following simulation results will not only show how each of these features fairs on its own, but also in combination.

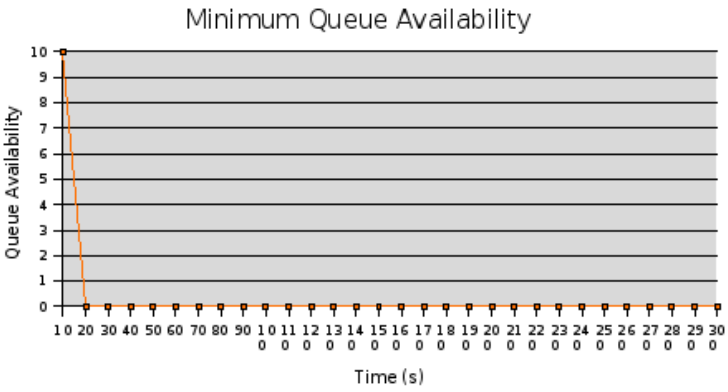
A.



B.



C.



D.

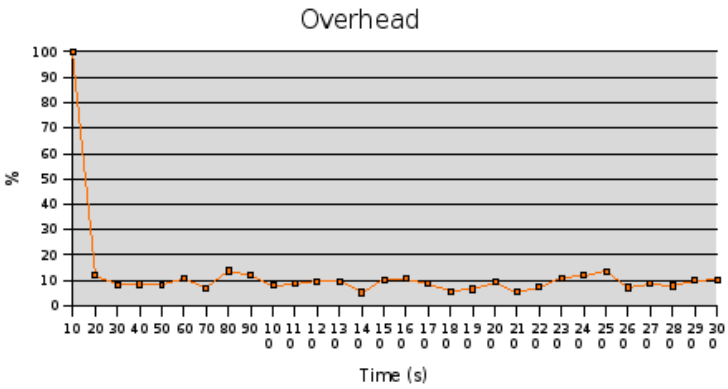
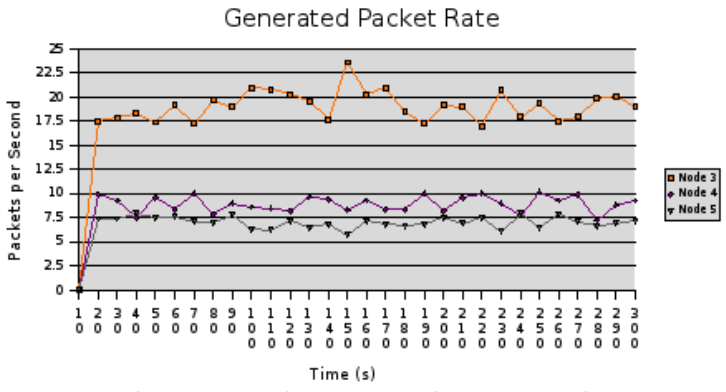


Figure 4.6: Congestion-Control Simulation Results

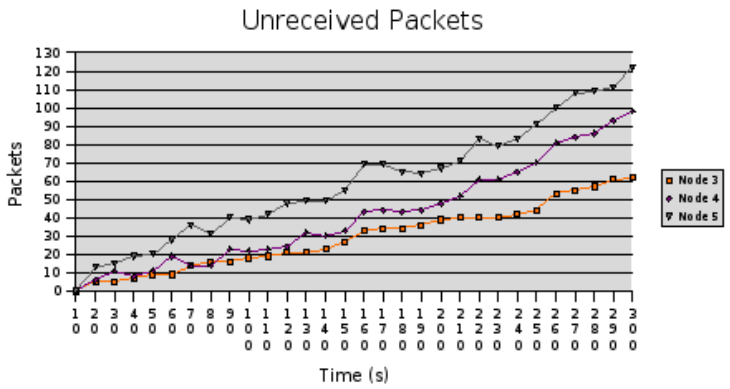
These results appear to be a total failure on the congestion-control feature's behalf, as congestion is by no means avoided or even mitigated. This is due to the fact that, although the congestion-control module correctly detects and acts upon the first signs of congestion (as soon as the queue availability falls below 50%), the link layer protocol used by the simulator does not send out the congestion warning packet soon enough, and thus several packets are still received before the nearly congested node is able to prevent further congestion. This problem could be solved either by using an alternative MAC layer protocol, as suggested in [11], which would prioritize traffic from congested nodes, or by tweaking the congestion thresholds and increasing the core queue size to accommodate these bursts of packets. Unfortunately, the limited memory resources available on real-world sensor nodes (4 kB on the MICAz platform) limit the core queue size to the current value of 10 packets when all of WMTP's features are compiled into the nodes binary image.

On the other hand, this test only needs the congestion control feature, thus the additional freed up memory may be used to increase the core queue size up to 22 packets. Under these circumstances, the congestion control module may be tweaked to detect congestion when the core queue falls beneath 90% availability, and to assert its absence once the core queue returns above the 95% availability threshold. This tweaked version was also simulated and the results are presented below.

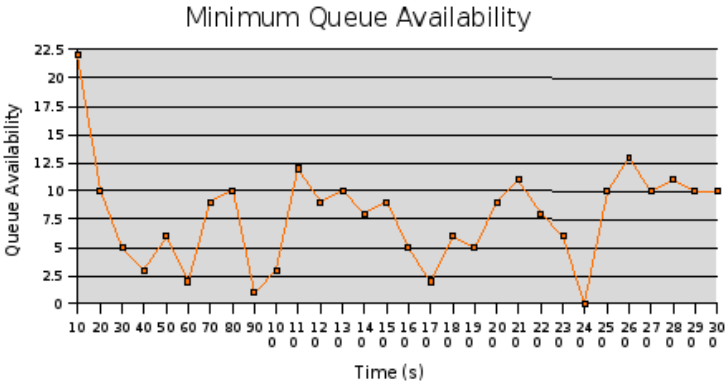
A.



B.



C.



D.

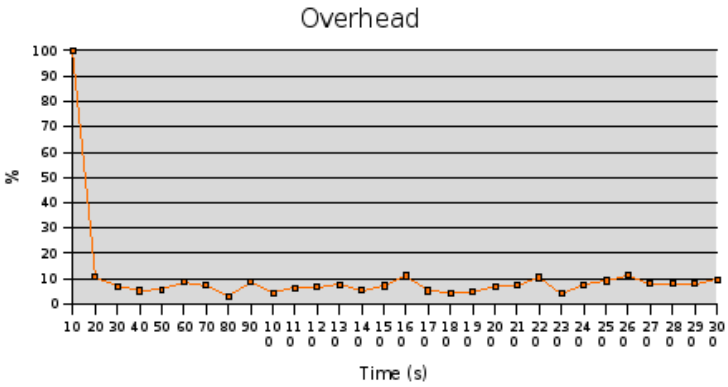
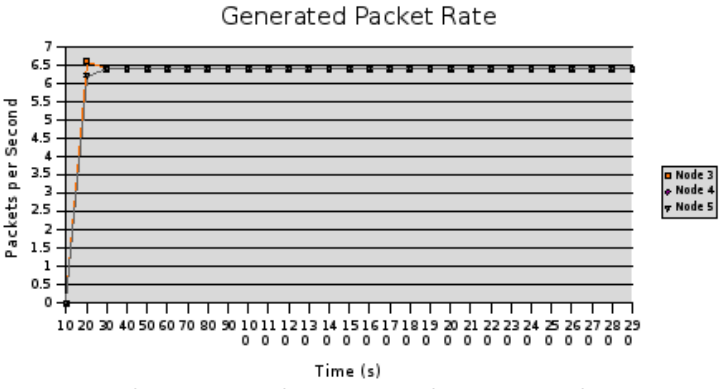


Figure 4.7: Congestion-Control Simulation Results (Tweaked Version)

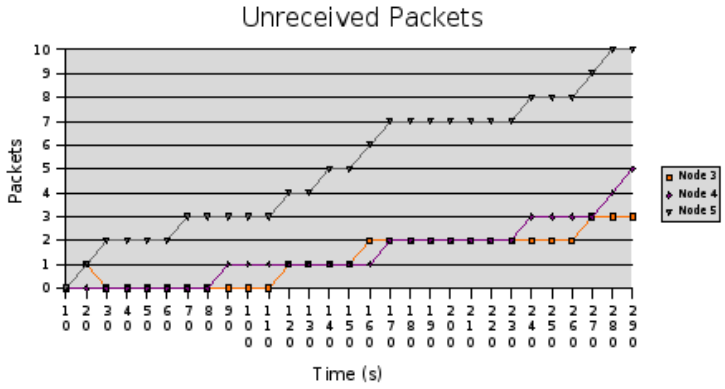
This time, the results clearly show that the congestion control feature was able to mitigate congestion, as the values in Figure 4.7.C don't tend towards zero over time. Additionally, although the sources generate their data at approximately the same rates, Figure 4.7.B shows that far fewer packets are lost, especially from the nodes that are not directly connected to the sink, thus decreasing the associated overhead.

The fairness feature, unlike congestion control, does not operate based on core queue availability, but rather on the rate at which the core is able to send packets. Although avoiding congestion is not this feature's primary design goal, it ultimately also contributes to attaining it.

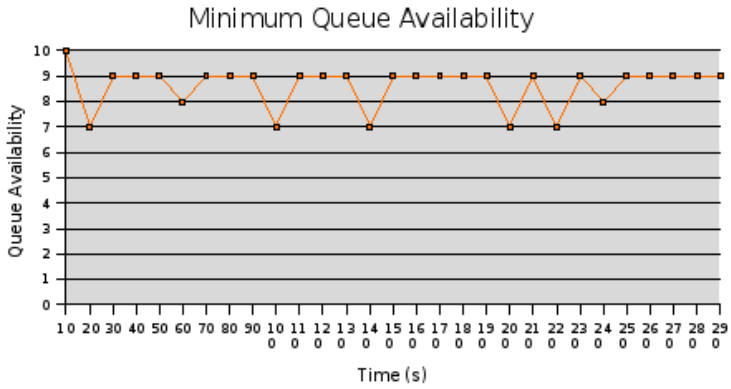
A.



B.



C.



D.

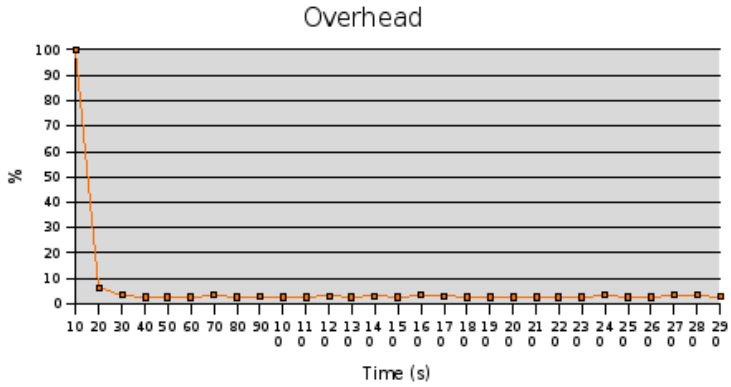
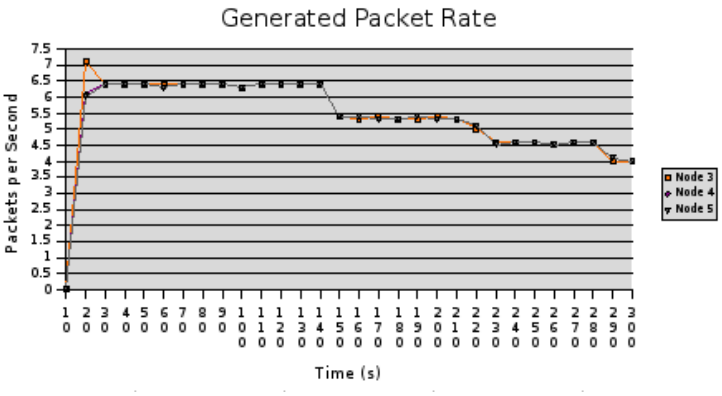


Figure 4.8: Fairness Simulation Results

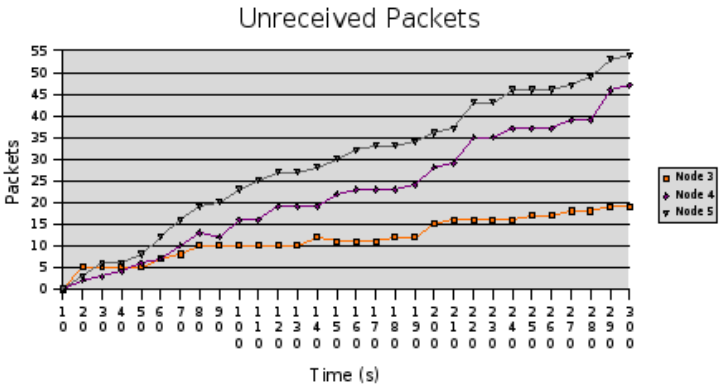
The simulation results show that the fairness feature operates as expected, since, after an initial period of slight instability, packets are generated, from all three sources, at approximately the same rate. Additionally, congestion is completely avoided throughout the network as not once does any node reach the limit of its core queue capacity. This leads to the loss of much fewer packets (the few that are lost are due to link layer collisions) and, thus, a much smaller overhead.

The following simulation shows the combined effects of using both congestion-control and fairness simultaneously.

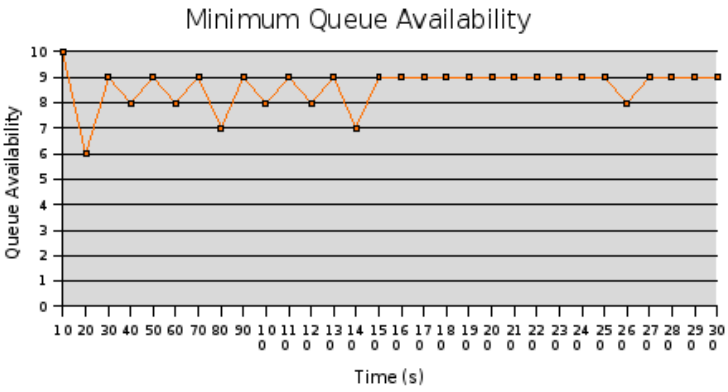
A.



B.



C.



D.

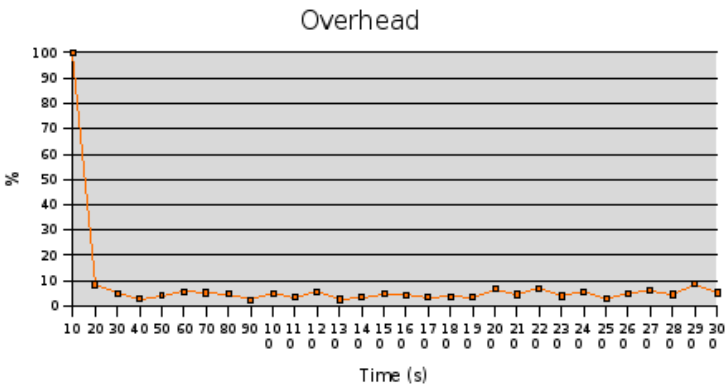
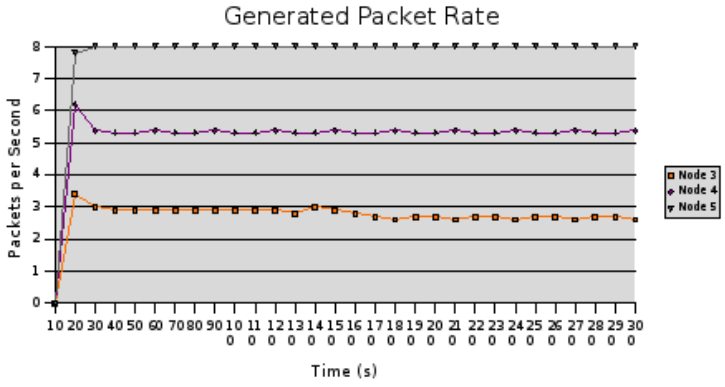


Figure 4.9: Congestion-Control and Fairness Simulation Results

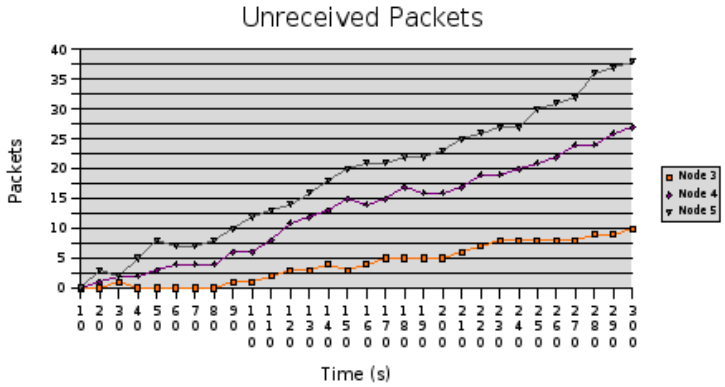
These results indicate that, although congestion is still avoided throughout the network, the simultaneous use of congestion control and fairness, in this scenario, is redundant and leads to a conservative packet generation rate. Additionally, the congestion control feature adds a slight burstiness to the in-network data flows, thus leading to the occurrence of more packet collisions, as can be seen in Figure 4.9.C.

The following simulation makes use of the weighting capabilities of WMTP's fairness feature. While previous simulations set equal weights to all sources, this simulation sets a weight of one, two, and three to each source, respectively.

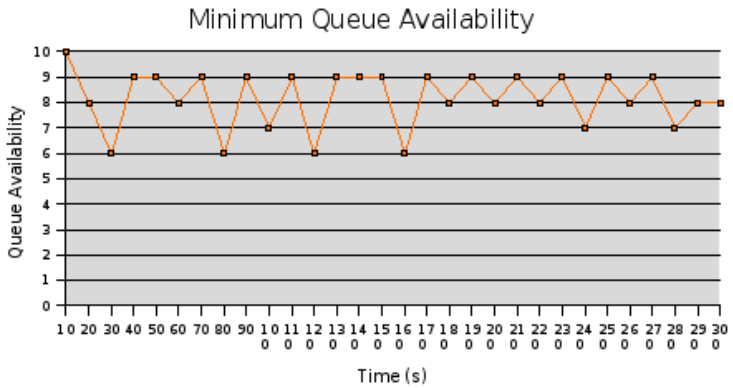
A.



B.



C.



D.

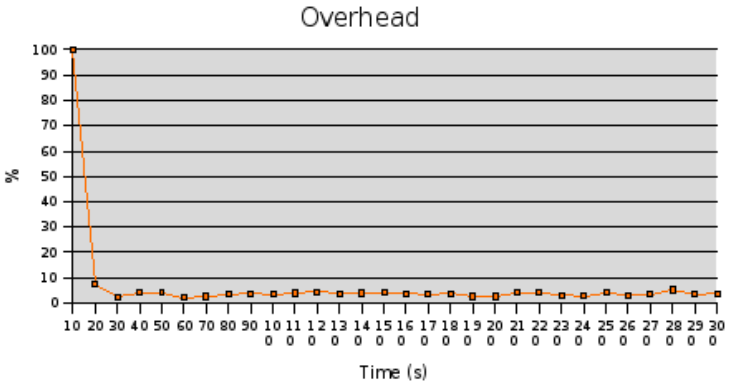
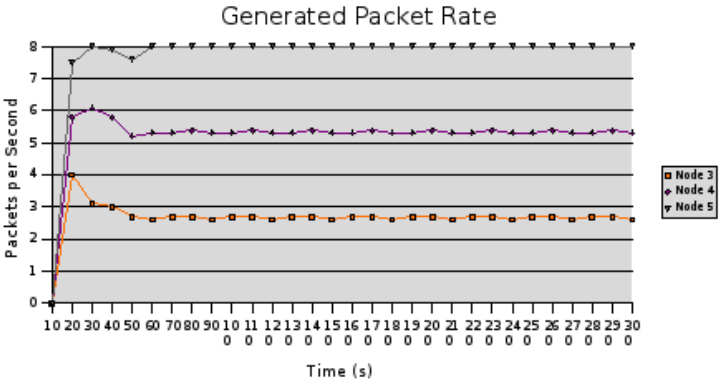


Figure 4.10: Weighted-Fairness Simulation Results

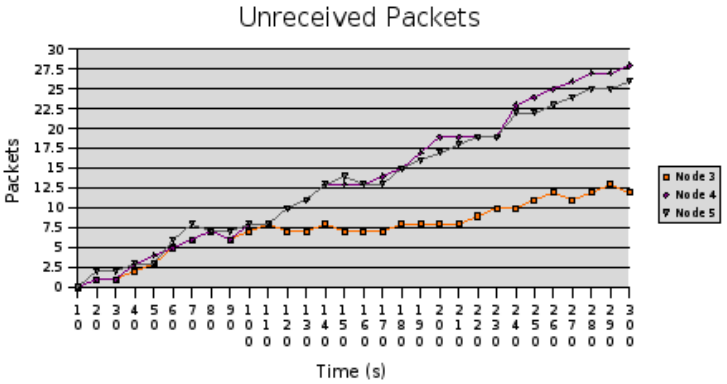
Figure 4.10.A clearly shows that the fairness feature correctly complies with the differentiated weights, as expected, while also still maintaining congestion under control, since the core queues never reach their limits. Unfortunately, since the nodes farther away from the sink are the ones with the highest weight, they ultimately generate more data, thus being also more susceptible to link layer packet collisions.

The following simulation, just as before, adds the use of the congestion-control feature to weighted-fairness.

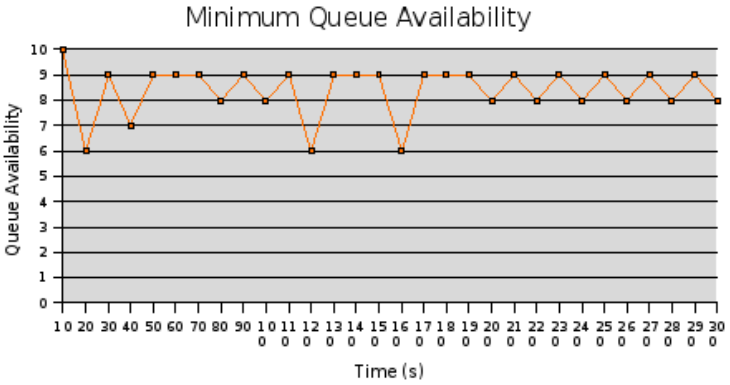
A.



B.



C.



D.

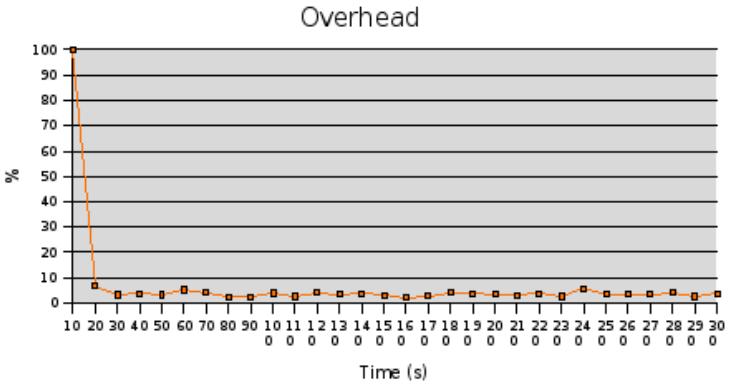


Figure 4.11: Congestion-Control and Weighted-Fairness Simulation Results

Unlike in the previous test case, where congestion control was used with simple fairness, Figure 4.11.A shows that the congestion control feature doesn't induce a conservative response from weighted fairness. On the other hand, Figure 4.11.C shows a slight decrease in overall congestion, which ultimately leads to a smaller end-to-end delay. Additionally, the use of congestion control, in this case, led to a slight decrease on the number of lost packets, thus also slightly decreasing the associated overhead.

The above simulation results indicate that congestion control and fairness features can coexist peacefully. On the other hand, the simultaneous use of both features generally adds only a slight improvement to their individual performances, if any at all. This being the case, it would probably be a wiser decision to include only one of these features, thus freeing up resources for an otherwise useful purpose. As for comparing each feature against the other, fairness was shown to also contribute to mitigating congestion but, on the other hand, congestion control, once adequately tweaked, allows the nodes to generate their data at a higher, yet unfair, rate.

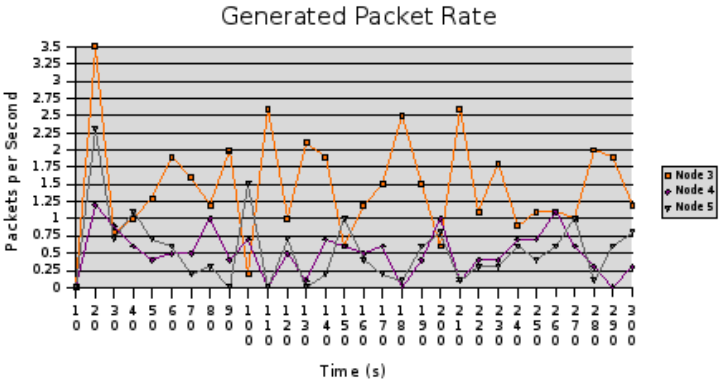
4.3.4 WMTP Reliability

Unlike the previous features, WMTP reliability goes beyond mere traffic shaping. The methodology behind WMTP reliability implies retaining cached versions of packets within the core queue of, not only the source nodes, but also any forwarding nodes

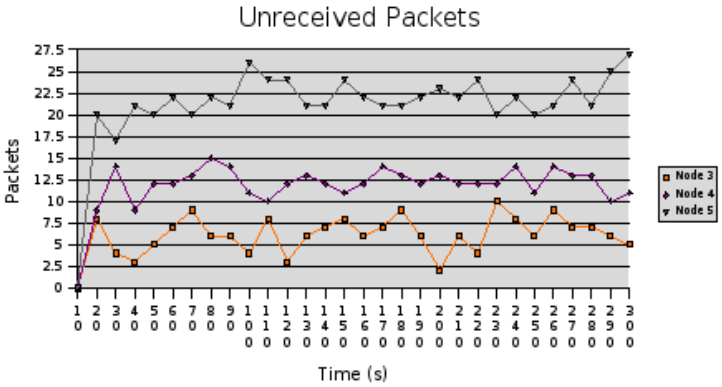
along the way. These cached versions must persist after the packet has been sent in order to successfully recover from its eventual loss but, on the other hand, this also implies that the core queues will tend to take longer to free up, leaving the bottleneck nodes even more vulnerable to congestion. This being the case, it is understandable that the use of reliability semantics may involve a high performance penalty.

The following simulations assess how WMTP reliability performs as a stand-alone feature, as well as how the congestion-control and the fairness features affect its performance. These simulations are useful not only to make sure that WMTP reliability operates as expected, but also to quantify the performance toll that it entails.

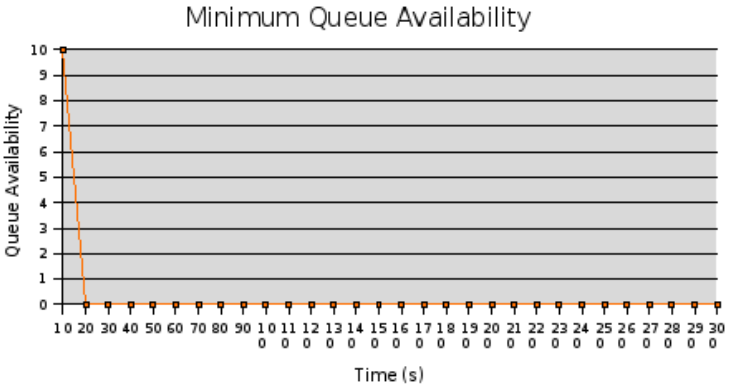
A.



B.



C.



D.

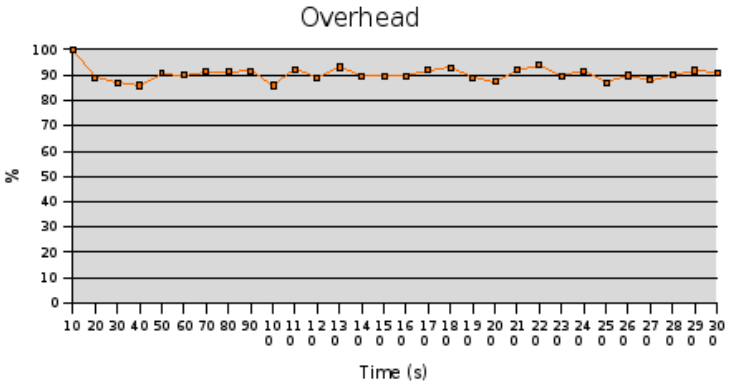
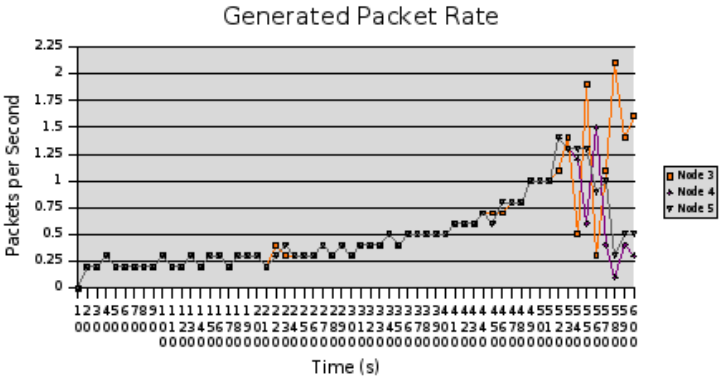


Figure 4.12: WMTP Reliability Simulation Results

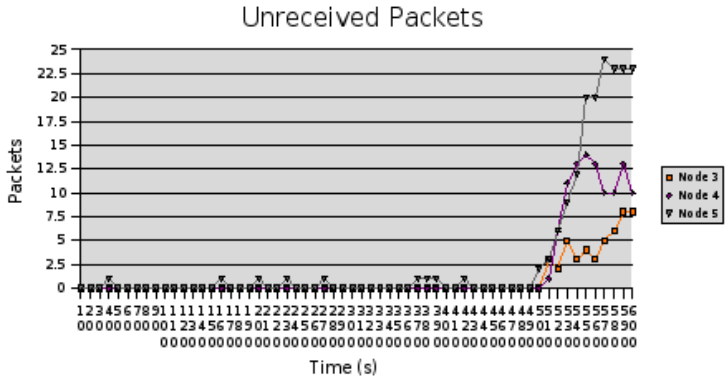
This simulation shows how WMTP reliability works on its own. Since Figure 4.12.B doesn't have a growth tendency, it is clear that the feature does, in fact, provide full packet reliability. On the other hand, the generated packet rate suffered a great fall when compared to the base-line scenario, achieving approximately a sixth of the base-line rates. As for Figure 4.12.C, since neither congestion-control, nor fairness was used in this simulation, the network sustains a constant state of severe congestion, which not only causes a certain level of burstiness in the generated rates, but also leads to the severe overheads.

The previous simulation shows how WMTP reliability operates under extreme load, since each source is generating its data at the highest rate that it physically can. In order to show how this feature works under a less stressing environment, an additional simulation was executed where each source also varied the rate at which it throttled its data generation. This way, each source started off by generating a packet every five seconds and then reduced this packet period by 250 ms, every 30 seconds, thus ending up by generating its data at the highest rate it physically could, by the end of this special 10 minute simulation.

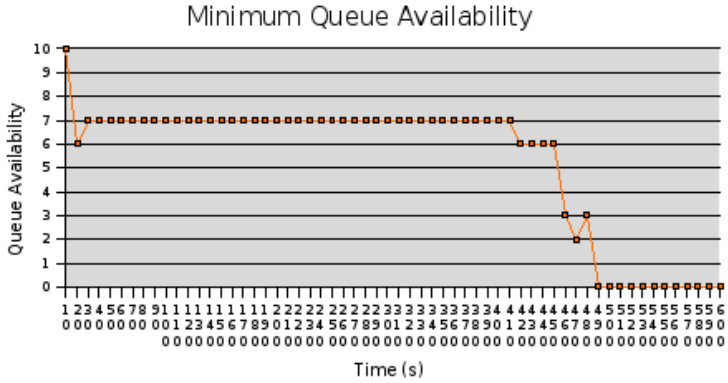
A.



B.



C.



D.

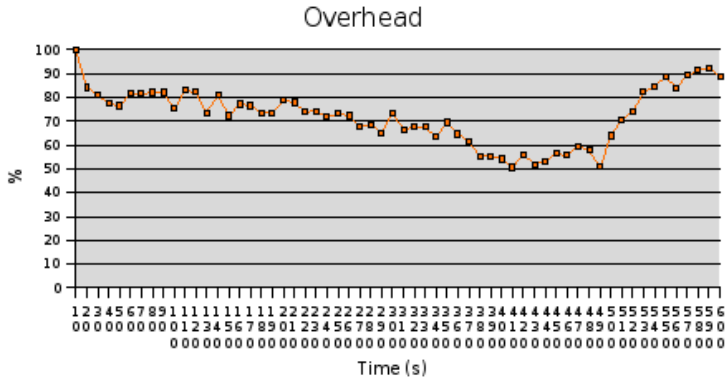


Figure 4.13: WMTP Reliability Operating Under Varying Load Conditions

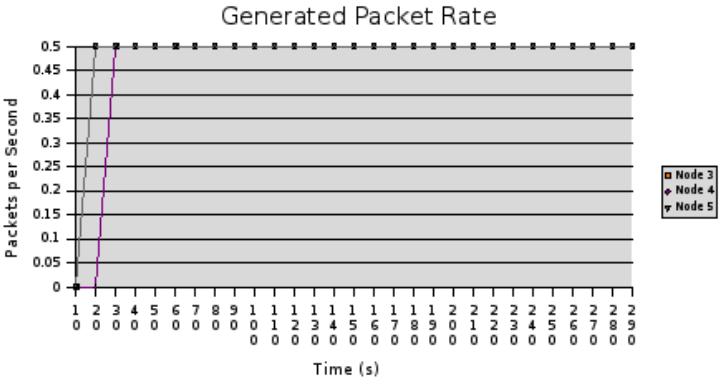
Since Figure 4.13.C is initially stable, it is shown that, for lighter loads, the WMTP reliability feature can operate without congesting the network. This completely stops the packet generation burstiness, as well as reduces end-to-end delay to a minimum, as can be seen through Figure 4.13.B.

As for Figure 4.13.D, it is clear that, if packets are generated at too slow a rate, the network will spend most of its time sending out periodic empty availability maps, thus leading to an additional increase in protocol overhead. On the other hand, generating the packets at to high a rate congests the network, also increasing protocol overhead. This situation leads to the existence of an optimum value that not only assures a better

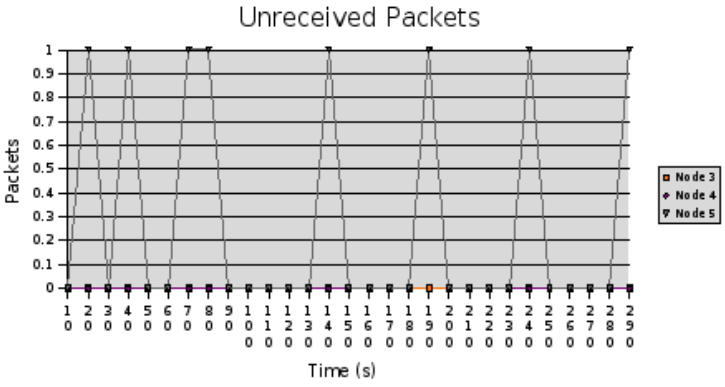
usage of network resources, but also avoids wasteful idle periods. This rate was determined, experimentally, to be between a packet every two seconds and one every second, for this scenario.

Since WMTP reliability is supposed to provide full packet reliability, even in the worst of conditions, a special simulation was prepared using a reasonable packet generation rate (one packet every two seconds) with a higher packet loss probability for each radio link. In the following simulation, approximately 10% of all packets sent are lost, and thus must be repeated.

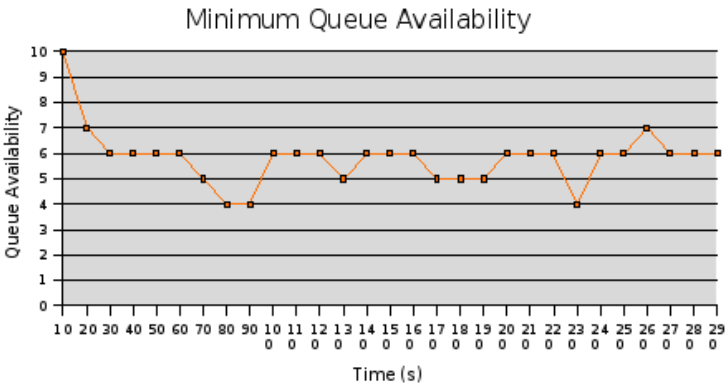
A.



B.



C.



D.

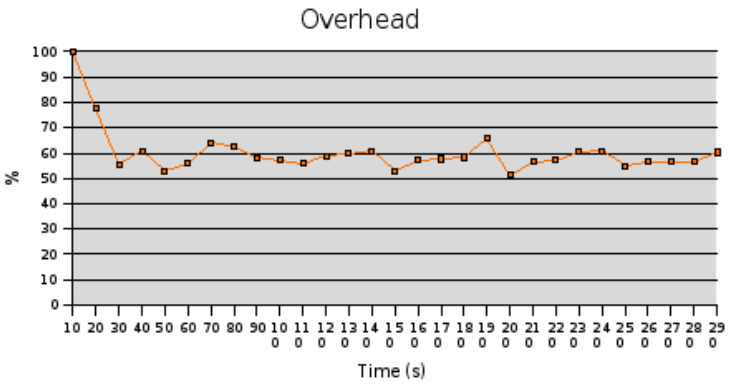
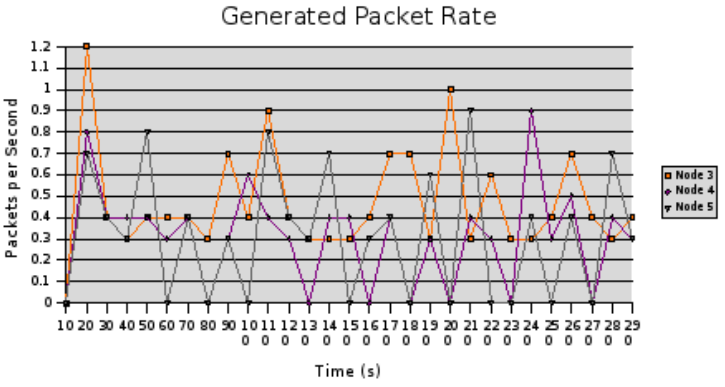


Figure 4.14: WMTP Reliability Simulation Results (Lossy Scenario)

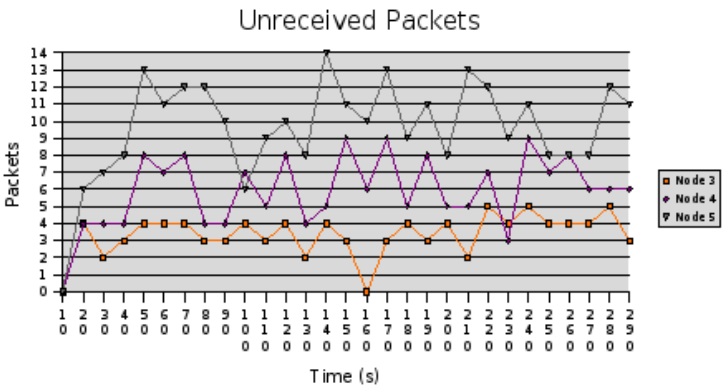
Just as before, Figure 4.14.B doesn't present a growth tendency, thus making it clear that even under these harsher conditions, WMTP reliability still holds up, albeit with some additional delay. On the other hand, Figure 4.14.C shows a minor increase in congestion and there is also a slight increase in the overall protocol overhead, thus showing that although WMTP reliability does still function, as expected, in a lossy environment, its performance is still mildly degraded.

The following simulation shows how WMTP reliability is affected when congestion control is used to regulate packet generation.

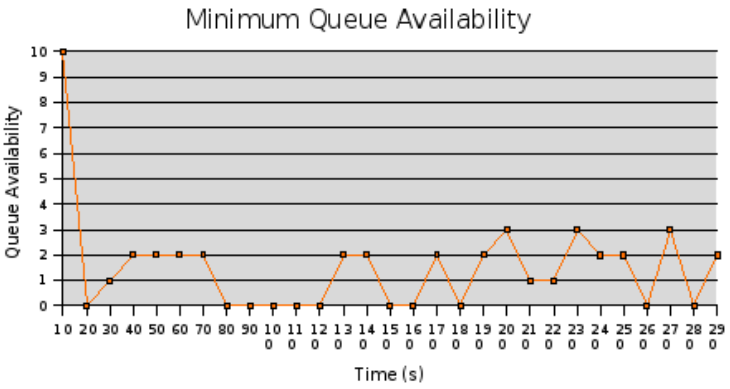
A.



B.



C.



D.

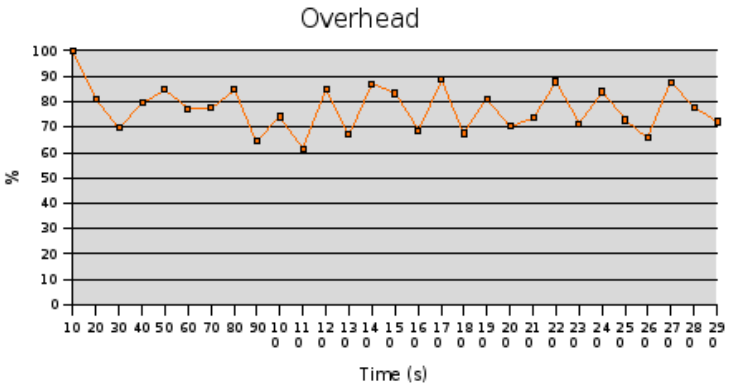
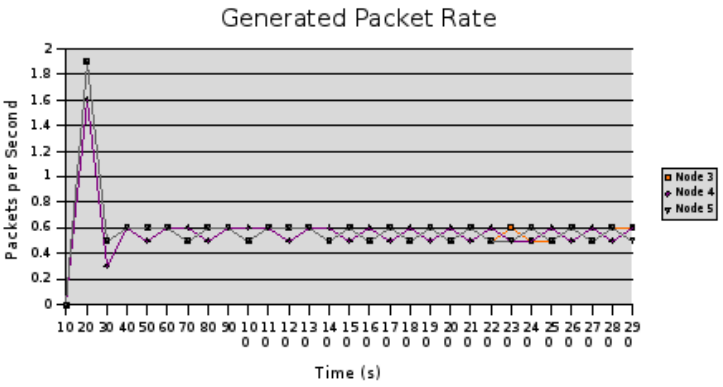


Figure 4.15: WMTP Reliability and Congestion Control Simulation Results

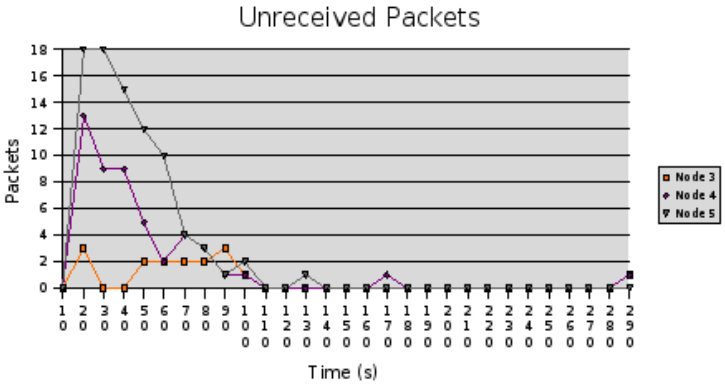
Just as before, there is no growth tendency in Figure 4.15.B, thus the use of congestion control does not interfere with WMTP reliability's main functionality. On the other hand, Figure 4.15.C indicates that the general state of congestion is slightly alleviated, thus showing some improvements on that front. Unfortunately, this slightly improved congestion state comes at the price of an even lower packet generation rate than in the unregulated scenario (approximately half of the values attained with just WMTP reliability). These factors ultimately contribute to a better network resource utilization, thus slightly cutting down the protocol overhead.

The following simulation shows how WMTP reliability is affected under the use of fairness.

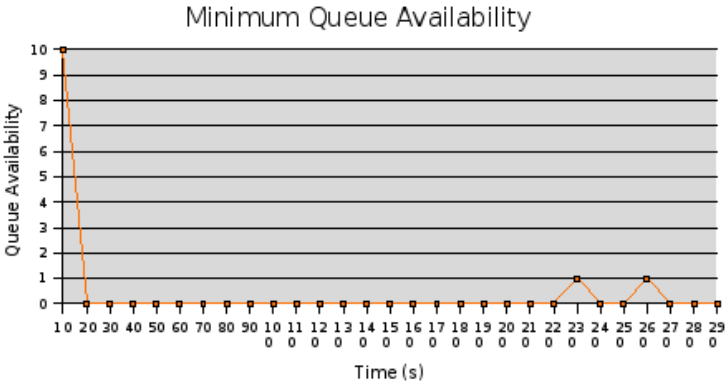
A.



B.



C.



D.

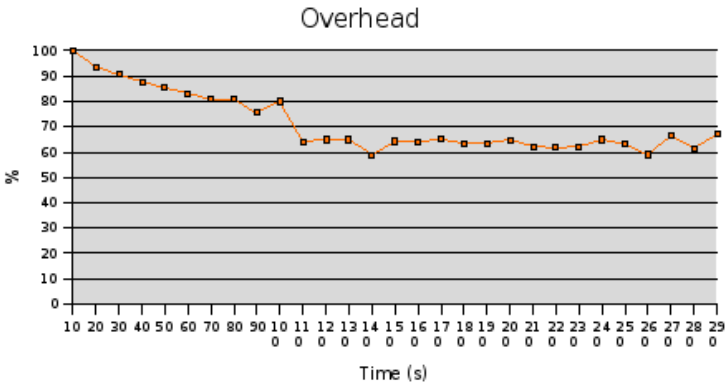
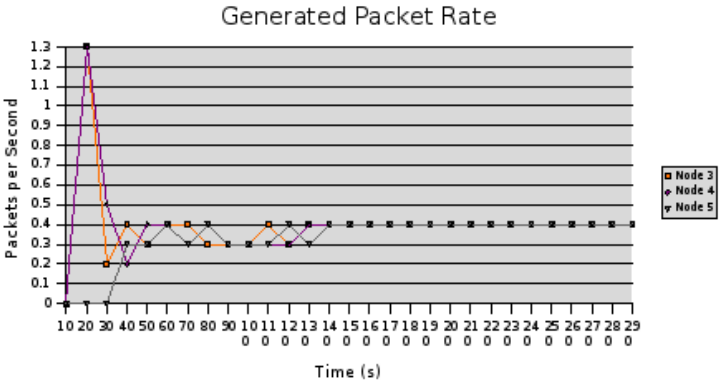


Figure 4.16: WMTP Reliability and Fairness Simulation Results

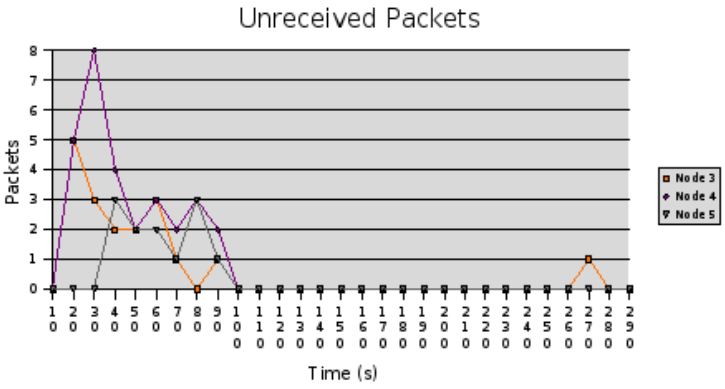
Unlike the previous case, where the congestion-control feature was used with WMTP reliability, Figure 4.16.C shows that the overall state of severe congestion is only very slightly mitigated. On the other hand, the received packet rates do converge towards a common rate, without any burstiness, which is approximately the same value that the nodes farthest away from the sink managed to achieve in the initial scenario, with just WMTP reliability. This shows that, on one hand, the existence of WMTP reliability does not affect the fairness feature's ability to function, and, on the other, that fairness does not entail a considerable performance penalty on WMTP reliability. Additionally, not only does Figure 4.16.B not show a growth tendency, but after an initial period of instability, it decrease to near-zero values, meaning that, although the network nodes appear to be congested, the delay suffered by packets along their route towards the sink is minimal. As for the protocol overhead, the use of fairness is shown to regulate packet generation at a rate that approximates the ideal level that maximizes the utilization of network resources.

The following simulation shows how all three of these features work simultaneously.

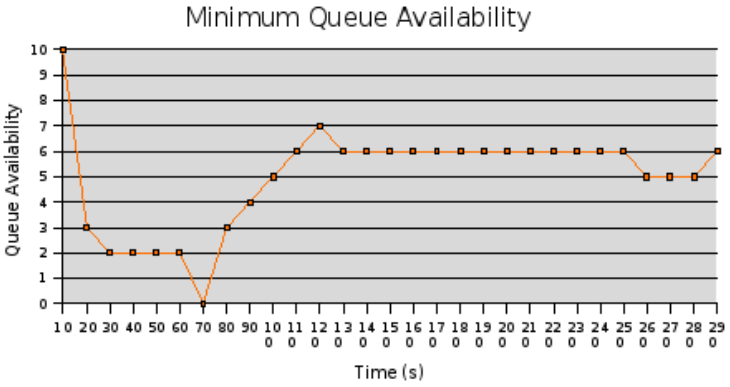
A.



B.



C.



D.

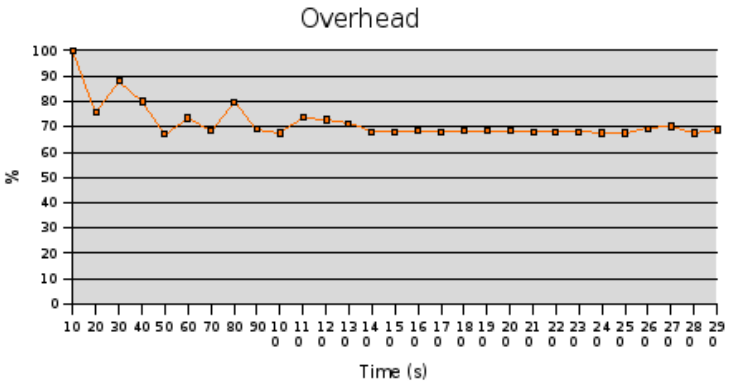


Figure 4.17: WMTP Reliability, Congestion Control, and Fairness Simulation Results

This simulation shows that the simultaneous use of WMTP reliability with congestion-control and fairness manages to pick up some advantages, as well as some of the disadvantages associated to each individual feature. The congestion-control feature is successful in mitigating the overall congestion state, as Figure 4.17.C does not tend towards zero but, as before, it entails a performance penalty, since packet generation rates are slightly decreased. Additionally, the fairness feature is also shown to work in this scenario, since packet generation rates still converge towards a common value. Unfortunately, the overall protocol overhead is not as good as what has already been achieved in previous scenarios.

4.3.5 Quality-of-Service

As previously mentioned, the quality-of-service feature was simulated under a slightly different scenario from the other features (see Figure 4.2). This scenario was purposely selected to be extremely harsh in the absence of quality-of-service, in order to prove this feature's worth.

To precisely show the advantages of using this feature, two simulations were created, one with quality-of-service enabled and one with it disabled. Both simulations use the special quality-of-service scenario previously described and, in both situations, the source nodes generate packets at the same rate; in other words, the quality-of-service

disabled sources, nodes 1 and 4, generate packets at the highest rate that they physically can, and the quality-of-service enabled node, node 5, generates a new packet every 200 ms. This particular value was determined to be approximately the highest rate the quality-of-service connection could withstand, while still ensuring its service levels. If a faster rate were to be requested, then the connection would fail to be established, as the reservation system would deny the request.

For each simulation, the received packet rate and the unreceived packets are shown only for the packets that originate from the single source that, in the quality-of-service enabled simulation, actually uses this feature, node 5. This way, it is easier to assess the effects directly associated with the use of this feature.

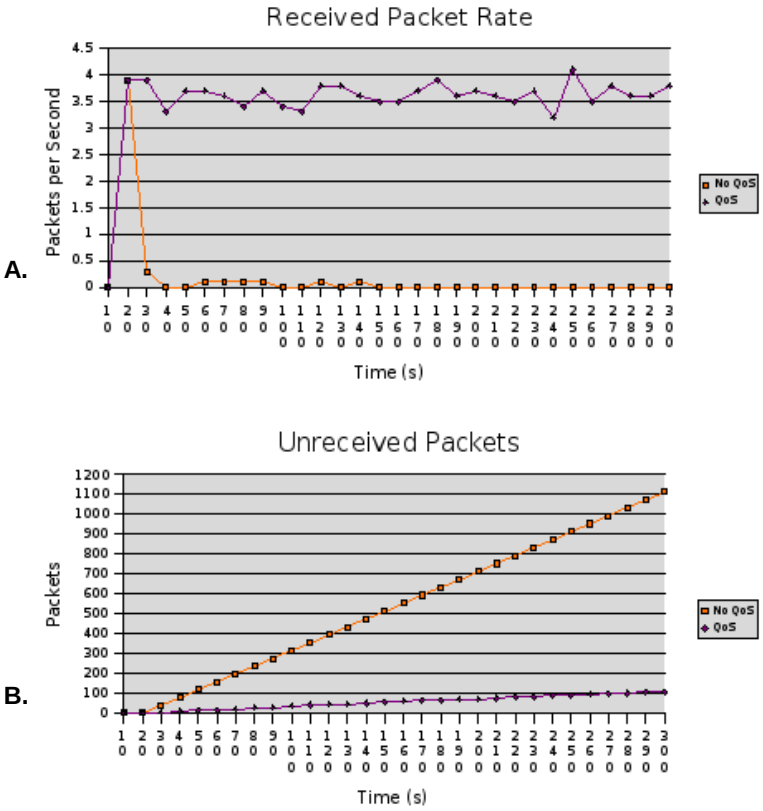


Figure 4.18: Quality-of-Service Simulation Results

As predicted, in the absence of quality-of-service, node 1, manages to almost entirely consume all of the network resources. This is clearly visible since, after a short while, all but a few of node 5's packets are lost. This obviously leads to a linear growth of the unreceived packets, in Figure 4.18.B.

On the other hand, when quality-of-service is activated, node 5 not only is able to send its data along the network, but it also manages to sustain its reserved rate with minimal packet losses (the few packets that are lost are due to MAC layer collisions, since the neighboring nodes are still generating packets at the highest rate that they physically

can). These simulations clearly demonstrate that the quality-of-service feature does, in fact, operate as expected, even under the direst of conditions.

5 Conclusions

This dissertation proposes the Wireless Modular Transport Protocol (WMTP), a new transport layer protocol for WSNs that provides its functionality through the use of a novel modular architecture. This protocol not only allows the simultaneous use of all of the main features commonly found in WSN transport protocols, namely congestion control, fairness, and reliability, but also does so in a modular fashion, thus allowing the application layer to use exactly the features that it requires without having to deal with the inevitable trade-offs associated with the ones that it doesn't. Additionally, WMTP provides its own unique set of uncommon features such as throttling, flow-control, transport layer quality-of-service, and optional integration with service-discovery.

The use of this specialized modular architecture also allows WMTP to support environments with heterogeneous applications, thus allowing different applications to use different features and still coexist in the same network. Additionally, this also allows the network administrator to build stripped down versions of WMTP that don't support the features that will never be used during the network's life-time. This way, the additional resources associated with any unused features may be freed up to be used for an otherwise more useful purpose.

In order to provide all of this functionality, WMTP's architecture was developed around the concept of a central core that, in itself, provides very little functionality. This core, in turn, provides a specialized framework that can then be used by one or more pluggable feature modules that actually implement the true transport layer functionality. On the other hand, the core is also used as the central coordinator that communicates with the upper and lower layers. In order to do this, WMTP uses an unconventional protocol stack that, rather than sitting in between the network layer and the application layer, as would be expected, it sits directly above the link layer and uses a specialized interface to communicate with the network layer.

The application layer interface, in turn, was also designed to support the advanced functionality that WMTP provides. This interface, aside from exchanging packet data with the application layer, also provides the tools that it needs to manage WMTP's features. Additionally, WMTP uses an efficient event-based approach to regulate application data generation, thus providing an alternative to the traditional blocking system calls, commonly found in most protocol stacks.

Once the design phase of this architecture and the development of the reference implementation were completed, the protocol was evaluated, through simulation, with TOSSIM. Since WMTP's reference implementation was developed for the TinyOS platform, this simulator is able to perform a complete assessment of WMTP's functionality, while using the same source code that runs on the sensor nodes.

In order to successfully evaluate how each of WMTP's features performs, either individually or in a combined form, a common simulation scenario was designed, thus allowing each simulation run to be executed under similar conditions. This way, not only can each feature's performance be compared to that of the others, but the effect that each feature has when used in combination with others can also be assessed.

Using this basic approach, each of WMTP's features was shown to work as expected when used individually. Additionally, all of the relevant combinations of features were also put to the test, in order to assess the existence of any adverse interactions between them. This time, the simulated results showed that, although the combination of certain features may not bring a significant benefit over the use of just one of them (e.g. the use of congestion control and fairness was shown to be redundant, since fairness already mitigated congestion to a certain extent), no significant adverse interactions were found to occur.

Aside from validating its functionality, the simulated tests were also used to evaluate WMTP's performance. Since WMTP was developed for WSNs, this performance assessment not only encompasses the usual throughput and/or delay metrics, but also takes into account energy consumption aspects. Since transmitting data over the radio is one of the operations that consumes more energy on wireless sensor nodes (see [3] for reference values), a basic protocol overhead metric was used to measure each feature's energy efficiency. This overhead value is calculated as the percentage of packets that are transmitted over the radio but aren't directly associated with the successful arrival of a data packet at its destination. This being the case, the overhead metric not only takes into account the wasteful use of explicit management packets or unnecessary retransmissions, but also any lost data packets that never reach their destination.

Using these basic performance attributes, most of WMTP's features were shown to perform their functionality in an adequate manner, while also avoiding too much additional overhead. The main exception to this was WMTP reliability. Since the use of reliability semantics involves holding data packets within the core queue of each node along the data's path, until the reliability module is sure that they won't be needed, the use of this feature generally comes with a severe performance penalty. The simulated results show that, although the use of additional features alongside reliability (e.g. fairness), may mitigate this effect, WMTP reliability will always entail a relatively large performance toll.

5.1 Future Work

Since WMTP uses a modular architecture, it is, by nature, easily extensible to provide new features. On the other hand, WMTP provides a set of specialized interfaces that may be used by external modules to provide additional functionality that is not directly associated to the transport layer, such as integrated service discovery or the use of advanced routing techniques. All of these factors contribute to the creation of a series of open issues and

future work suggestions that may be further explored. The following open issues are but some the ones that come to mind for further development:

- WMTP provides the option to integrate with service discovery, but the system that ultimately supplies this functionality is not a part of WMTP itself. The design and implementation of such a system for use through the WMTP architecture, or the integration of an already preexisting one, would be a considerable enhancement that is well worth the effort;
- Although WMTP already integrates with TinyOS's current routing layer, as well as provides its own source routed variant for connection oriented data, the development of additional routing modules to integrate WMTP with other, more advanced, routing systems would also be an interesting research effort to follow;
- While WMTP uses a conventional interface to exchange packets with the link layer, a specialized interface is used to obtain its quality-of-service characteristics. Since the link layer currently used by TinyOS doesn't provide quality-of-service semantics, these characteristics are currently inferred statistically. This being the case, it would be interesting to integrate WMTP with a true quality-of-service enabled link layer and to evaluate how its transport layer quality-of-service model would perform under these conditions;
- Even though the WMTP reference implementation was developed solely for TinyOS, nothing prevents that alternative implementations be developed for different platforms. One such platform where this would be particularly useful would be the computer to which the sink node is connected. Once a computer based WMTP implementation is developed, traditional applications could interact with the WSN directly, thus benefiting from all of the WMTP's functionality;
- Although extensive simulations have already been run to evaluate WMTP, further testing is needed to assess whether or not WMTP would actually perform similarly when used in real-word sensor networks;
- While several tests have been executed to show how WMTP's features perform, an additional effort would have to be made to compare WMTP's implementation of each individual feature with that of other existing protocols that also provide the same feature.

Although the further improvement of WMTP provides several open issues for future development, one must not forget that this protocol was designed to be used by applications. This being the case, WMTP also creates the opportunity to development all new WSN applications that can benefit from its advanced functionality. Under these circumstances, the possibilities are only limited by ones own imagination.

6 References

1. I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, E. Cayirci, “Wireless Sensor Networks: A Survey”, Computer Networks, 38(4): 393-422, March, 2002.
2. Atmel Corporation, “ATmega128(L) Datasheet”, Document Number Rev. 2467P-AVR-08/07.
3. T. Camilo, P. Melo, A. Rodrigues, L. Pedrosa, J. S. Silva, R. Neves, R. Rocha, F. Boavida; "Wireless Sensor Network Deployment: an Experimental Approach", Book Chapter, forthcoming in "Handbook of Wireless Mesh and Sensor Networking", McGraw-Hill International, New York
4. D. Chen, P. K. Varshney, “QoS Support in Wireless Sensor Networks: A Survey”, in proceedings of the International Conference on Wireless Networks (ICWN '04), June, 2004.
5. Chipcon AS, “Chipcon AS SmartRF® CC2420 Preliminary Datasheet (rev 1.2)”, June, 2004.
6. Crossbow Technology, Inc., “MICAz Datasheet”, Document Part Number 6020-0060-04 Rev. A.
7. A. Dunkels, “Distributed TCP Caching for Wireless Sensor Networks”, in proceedings of 3rd Annual Mediterranean Ad Hoc Net., June, 2004.
8. C. Ee, R. Bajcsy, “Congestion Control and Fairness for Many-to-One Routing in Sensor Networks”, in proceedings of ACM Sensys ’04, November, 2004.
9. D. Gay, P. Levis, D. Culler, E. Brewer, “nesC 1.1 Language Reference Manual”, May, 2003.
10. J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, D. Ganesan, “Building Efficient Wireless Sensor Networks with Low-Level Naming”, in proceedings of the Eighteenth ACM Symposium on Operating Systems Principles, October, 2001.
11. B. Hull, K. Jamieson, H. Balakrishnan, “Mitigating Congestion in Wireless Sensor Networks”, in proceedings of ACM Sensys ’04, November, 2004.
12. C. Intanagonwiwat, R. Govindan, D. Estrin, “Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks”, in proceedings of the Sixth Annual International Conference on Mobile Computing and Networking, August, 2000.
13. Y. Iyer, S. Gandham, S. Venkatesan, “STCP: A Generic Transport Layer Protocol for Wireless Sensor Networks” in proceedings of IEEE ICCCN, October, 2005.
14. P. Levis, N. Lee, “TOSSIM: A Simulator for TinyOS Networks”, September, 2003.

15. P. Levis, N. Lee, M. Welsh, D. Culler, “TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications”, in proceedings of the 1st international Conference on Embedded Networked Sensor Systems, November, 2003.

16. S. Park, “A Scalable Approach for Reliable Downstream Data Delivery in Wireless Sensor Networks”, in proceedings of ACM MobiHoc ’04, May, 2004.

17. Y. Sankarasubramaniam, O. B. Akan, I. Akyildiz, “ESRT: Event-to-Sink Reliable Transport in Wireless Sensor Networks”, in proceedings of ACM Mobihoc’03, June, 2003.

18. F. Stann, J. Heidemann, “RMST: Reliable Data Transport in Sensor Networks”, in proceedings of IEEE SNPA ’03, May, 2003.

19. K. Sundaresan, V. Anantharaman, H. Hsieh, R. Sivakumar, “ATP: A Reliable Transport Protocol for Ad-hoc Networks”, in proceedings of 4th ACM International Symposium on Mobile Ad-hoc Networking & Computing, 2003.

20. A. Tanenbaum, C. Gamage, and B. Crispo, “Taking Sensor Networks from the Lab to the Jungle”, In IEEE Computer, Volume 39, Issue 8, Aug. 2006 Page(s): 98-100.

21. C. Wan, “Siphon: Overload Traffic Management Using Multi-Radio Virtual Sinks in Sensor Networks”, in proceedings of ACM SenSys ’05, November, 2005.

22. C. Wan, A. Campbell, “PSFQ: A Reliable Transport Protocol for Wireless Sensor Networks”, in proceedings of ACM WSNA ’02, September, 2002.

23. C. Wan, S. Eisenman, A. Campbell, “CODA: Congestion Detection and Avoidance in Sensor Networks”, in proceedings of ACM Sensys ’03, November, 2003.

24. C. Wang, “Priority-Based Congestion Control in Wireless Sensor Networks”, in proceedings of IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing, June 2006.

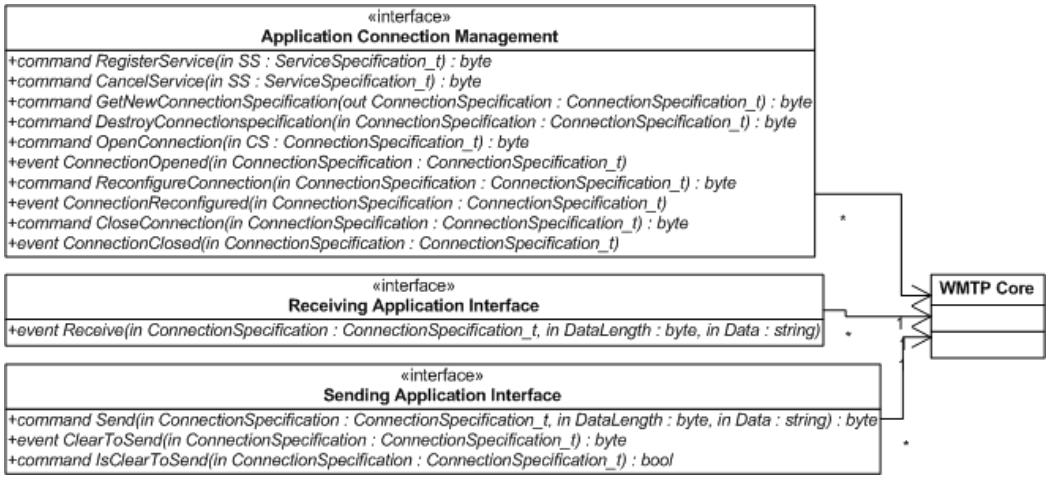
25. C. Wang, K. Sohraby, B. Li, M. Daneshmand, Y. Hu, “A Survey of Transport Protocols for Wireless Sensor Networks”, IEEE Network, 20(3): 34-40, June 2006.

26. A. Woo, D. Culler, “A Transmission Control Scheme for Media Access in Sensor Networks”, in proceedings of ACM Mobicom ’01, July, 2001.

27. H. Zhang, “Reliable Bursty Convergecast in Wireless Sensor Networks”, in proceedings of ACM Mobihoc ’05, May, 2005.

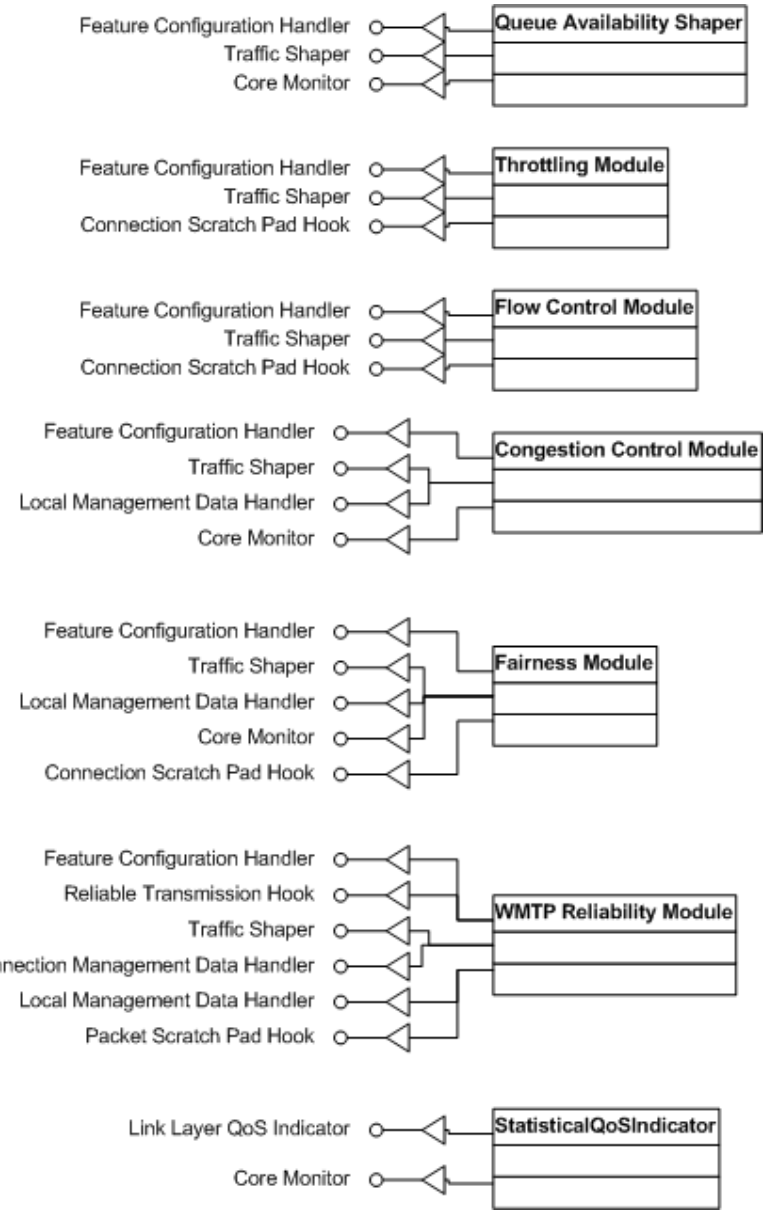
7 Annexes

Annex 1: WMTP Application Interface UML Static Structure Diagram

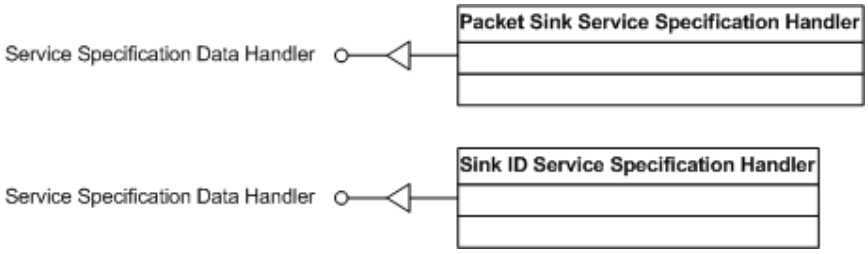


Annex 2: WMTP Core Interfaces UML Static Structure Diagram

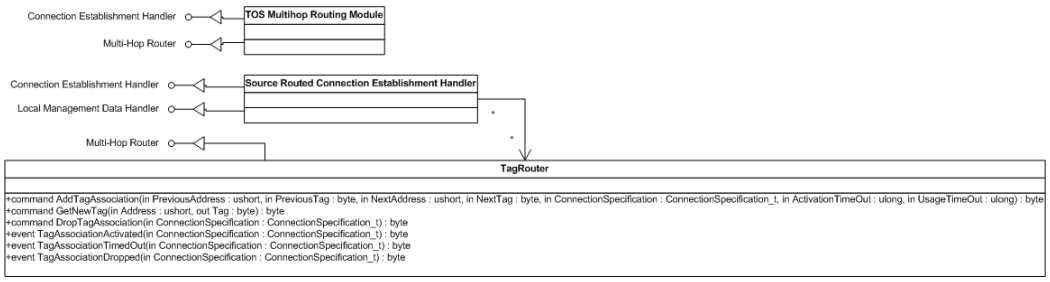
Annex 3: WMTP Feature Module UML Static Structure Diagram



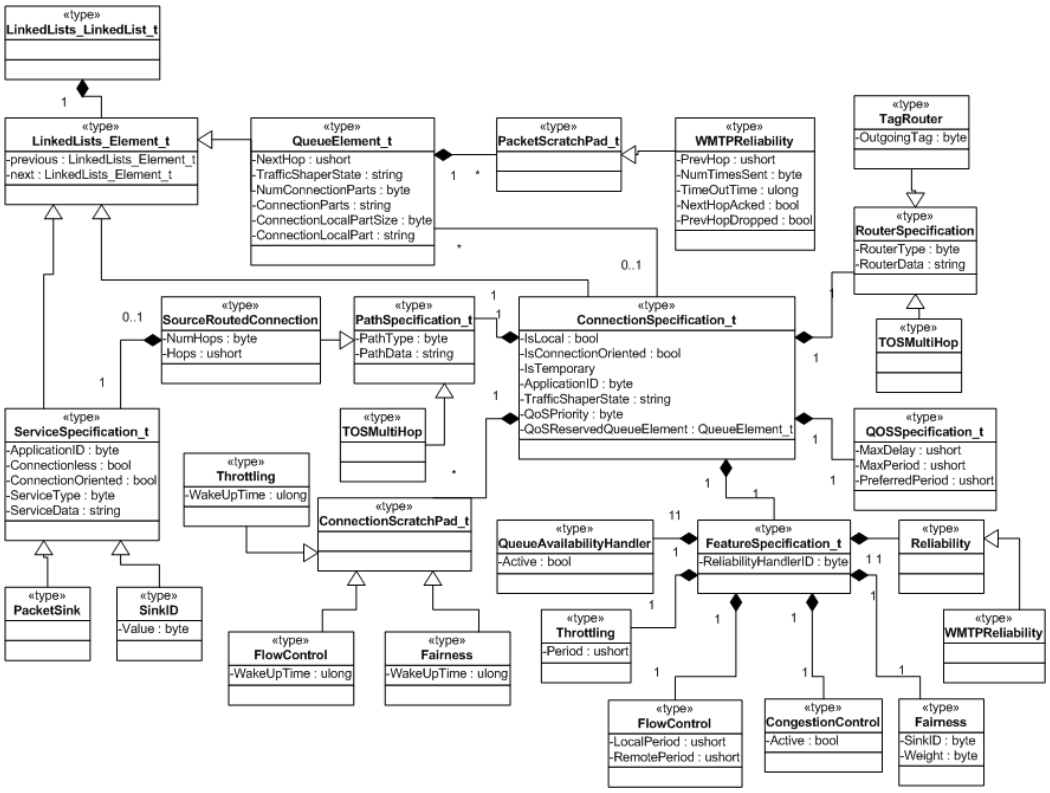
Annex 4: WMTP Service Specification Handlers UML Static Structure Diagram



Annex 5: WMTP Multi-Hop Routers UML Static Structure Diagram



Annex 6: WMTP Data Types UML Static Structure Diagram



Annex 7: WMTP Message Formats

	TOS_Msg					
Field:	Address	Type	Group	Data Length	Data	CRC
Size (Bytes):	2	1	1	1	<Data Length>	2

This is TinyOS's native message format. Any messages sent over the radio must be created through this structure.

	WMTP_Msg					
Field:	Src Addr	Local Part 0 Type	Local Part 0 Data	Local Part 1 Type	Local Part 1 Data	(...)
Size (Bytes):	2	1	*	1	*	(...)

This is the basic message structure that WMTP uses to convey all of its information. This includes local and connection management data as well as router data and the data payloads. The WMTP_Msg is, in essence, a collection of Local Parts. This being the case, the following Local Parts are defined:

- WMTP_LocalPart_CongCtrl
- WMTP_LocalPart_Fairness
- WMTP_LocalPart_Reliability
- WMTP_LocalPart_SrcRoutedConn
- WMTP_LocalPart_Conn

	WMTP_LocalPart_CongCtrl	
Field:	Reserved	Congestion Notification Bit
Size (Bits):	7	1

This structure holds the congestion control local management headers.

	WMTP_LocalPart_Fairness				
Field:	Last Sink	Sink ID	Normalized Period	Limiting Node	(...)
Size (Bits):	1	7	16	16	(...)

This structure holds the fairness local management headers. Basically, this pattern is repeated for each sink, with the last one containing the Last Sink flag set.

	WMTP_LocalPart_Reliability			
Field:	Originating Address	Last Packet	Packet ID	(...)
Size (Bytes):	16	1	15	(...)

This structure holds the WMTP reliability local management headers. Basically, this pattern is repeated for each packet within the availability map, with the last one containing the Last Packet flag set. Each packet is identified by the address of the node that generated it and an incremental 15 bit identifier.

	WMTP_LocalPart_SrcRoutedConn					
Field:	Next Hop	Next Tag	QoS Max Delay	QoS Max Period	QoS Preferred Period	QoS Accumulated Delay
Size (Bytes):	2	4	2	1	1	2
Field:	Num Hops	Hops	Configuration Data	Service Specification Data		
Size (Bytes):	1	[Num Hops x 2]	*	*		

This structure is used to establish source routed connections. The Configuration Data and Service Specification Data fields have a variable length, since they contain a WMTP_ConnPart_Config and a WMTP_SvcSpec, respectively.

	WMTP_LocalPart_Conn				
Field:	Router Type	Router Data	Connection Part 0 Type	Connection Part 0 Data	(...)
Size (Bytes):	1	*	1	*	(...)

This structure holds all of the data that is associated to a data packet, including router data, connection management data, and the packet payload itself. Basically this message is built by appending multiple Connection Parts, after the router data, with the last Connection Part being a WMTP_ConnPart_Data, WMTP_ConnPart_Close, or WMTP_ConnPart_Last structure. This being the case, the following routing headers are defined:

- WMTP_RouterData_Tag

Additionally, the following Connection Parts are defined:

- WMTP_ConnPart_Reliability
- WMTP_ConnPart_Data
- WMTP_ConnPart_Close
- WMTP_ConnPart_Last
- WMTP_ConnPart_Config

The WMTP_ConnPart_Close and WMTP_ConnPart_Last structures are special in that they have zero length. In other words, they are used solely as place markers and don't hold any additional data.

	WMTP_SvcSpec	
Field:	Type	Service Specification Data
Size (Bytes):	1	*

This structure holds service specification data, representing an interest.

	WMTP_SvcSpec_SinkID	
Field:	Reserved	Sink ID
Size (Bits):	1	7

This structure holds the sink ID service specification.

	WMTP_RouterData_Tag
Field:	Tag
Size (Bits):	8

This structure holds the routing data used by the tag router.

	WMTP_ConnPart_Reliability		
Field:	Originating Address	Packet ID	Reserved
Size (Bytes):	16	15	1

This structure holds the WMTP reliability connection management headers.

	WMTP_ConnPart_Data	
Field:	Data Payload Size	Data Payload
Size (Bytes):	1	*

This structure holds a packet's payload.

	WMTP_ConnPart_Config		
Field:	Configuration Part 0 Type	Configuration Part 0 Data	(...)
Size (Bytes):	1	*	(...)

This structure holds the configuration data for connections or individual packets. Basically, this pattern is repeated for each Configuration Part, with the last one being a WMTP_ConfPart_Last structure. This being the case, the following Configuration Parts are defined:

- WMTP_ConfPart_QueueAvailabilityShaper
- WMTP_ConfPart_Throttling
- WMTP_ConfPart_FlowCtrl
- WMTP_ConfPart_CongCtrl
- WMTP_ConfPart_Fairness
- WMTP_ConfPart_WMTPReliability
- WMTP_ConfPart_Last

The WMTP_ConfPart_QueueAvailabilityShaper, WMTP_ConfPart_Throttling, WMTP_ConfPart_CongCtrl, WMTP_ConfPart_WMTPReliability, and WMTP_ConfPart_Last structures are special in that they have zero length. In other words, they are used solely as place markers and don't hold any additional data.

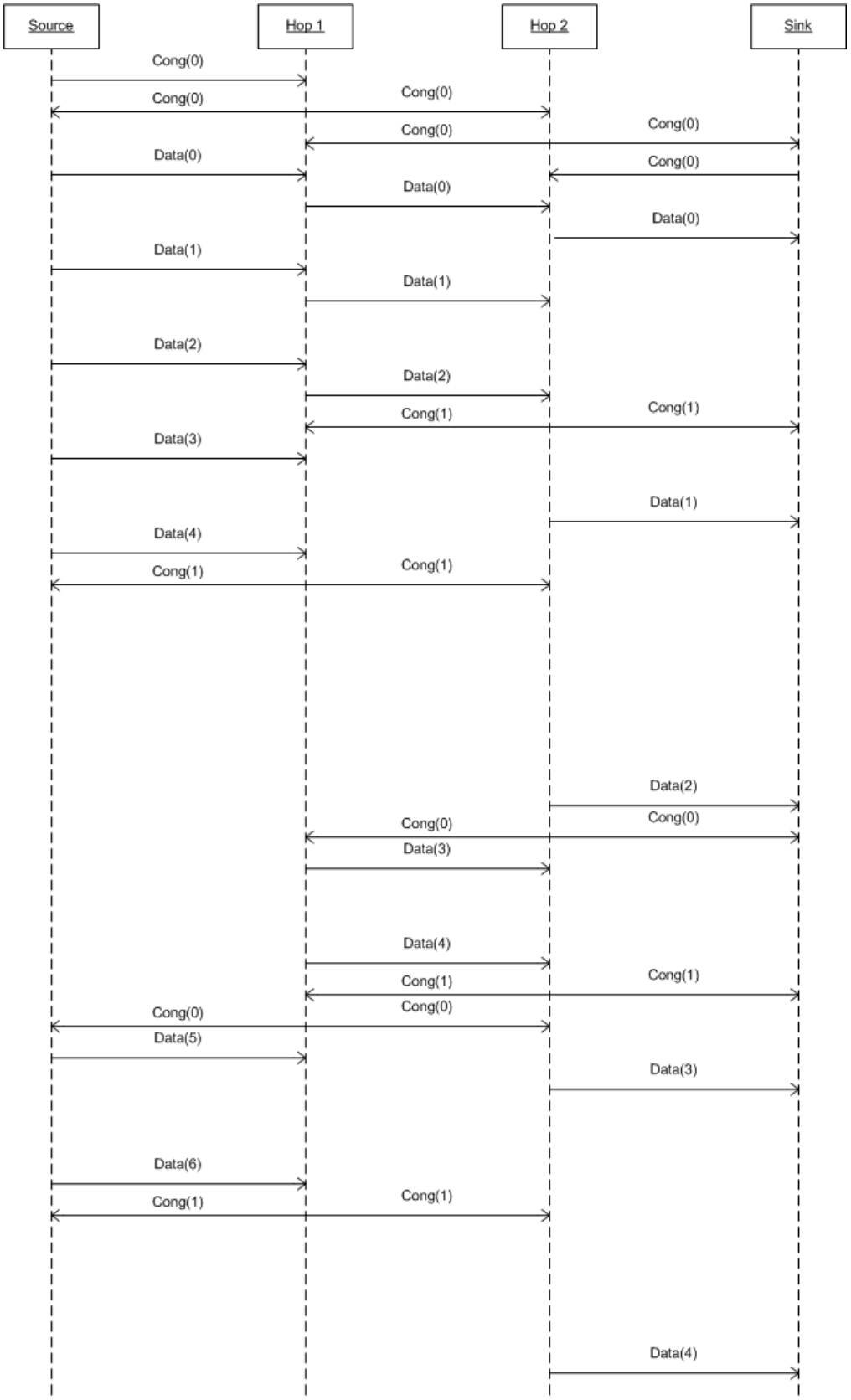
	WMTP_ConfPart_FlowCtrl
Field:	Period
Size (Bits):	16

This structure holds the flow control configuration.

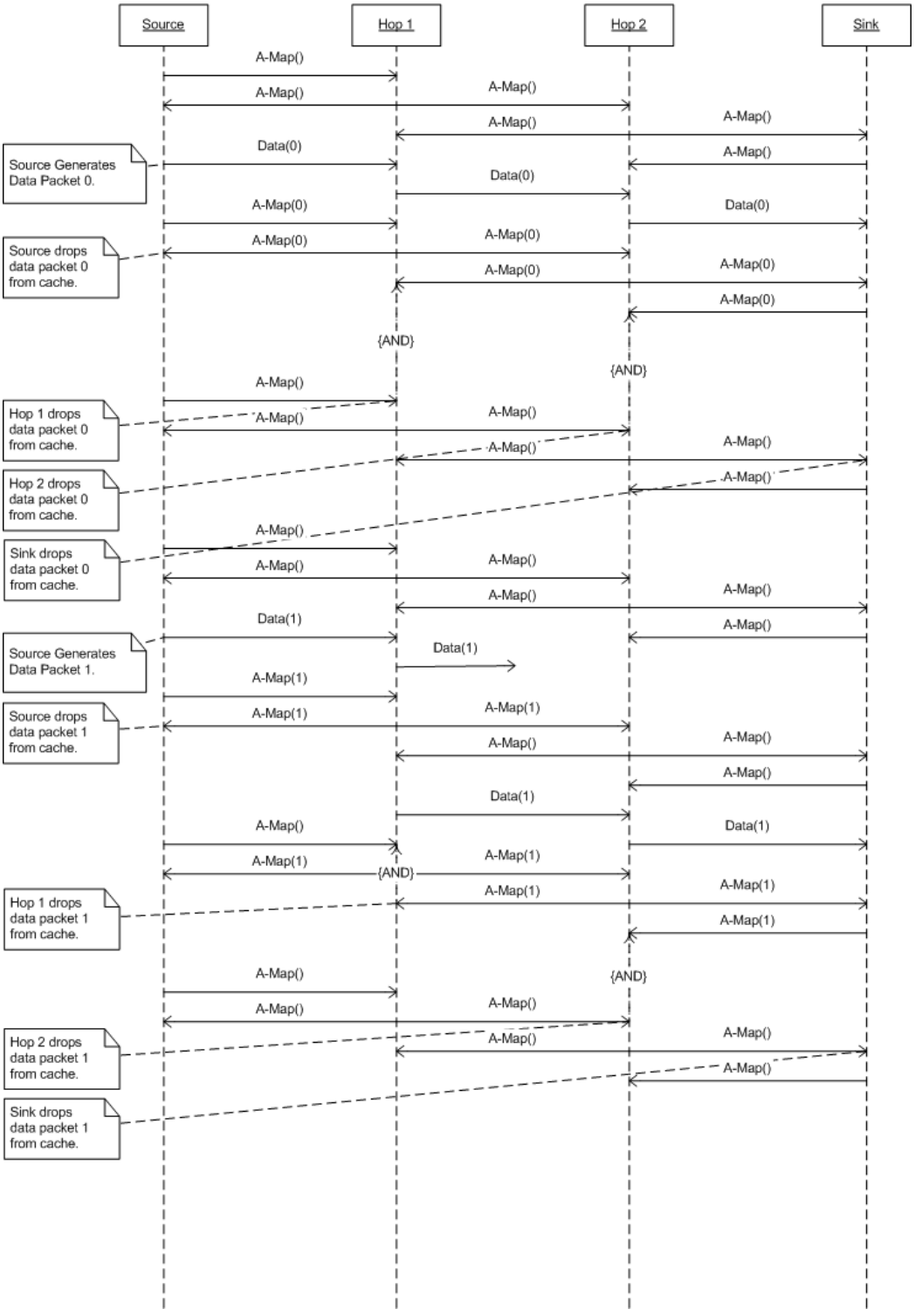
	WMTP_ConfPart_Fairness		
Field:	Reserved	Sink ID	Weight
Size (Bits):	1	7	8

This structure holds the fairness configuration.

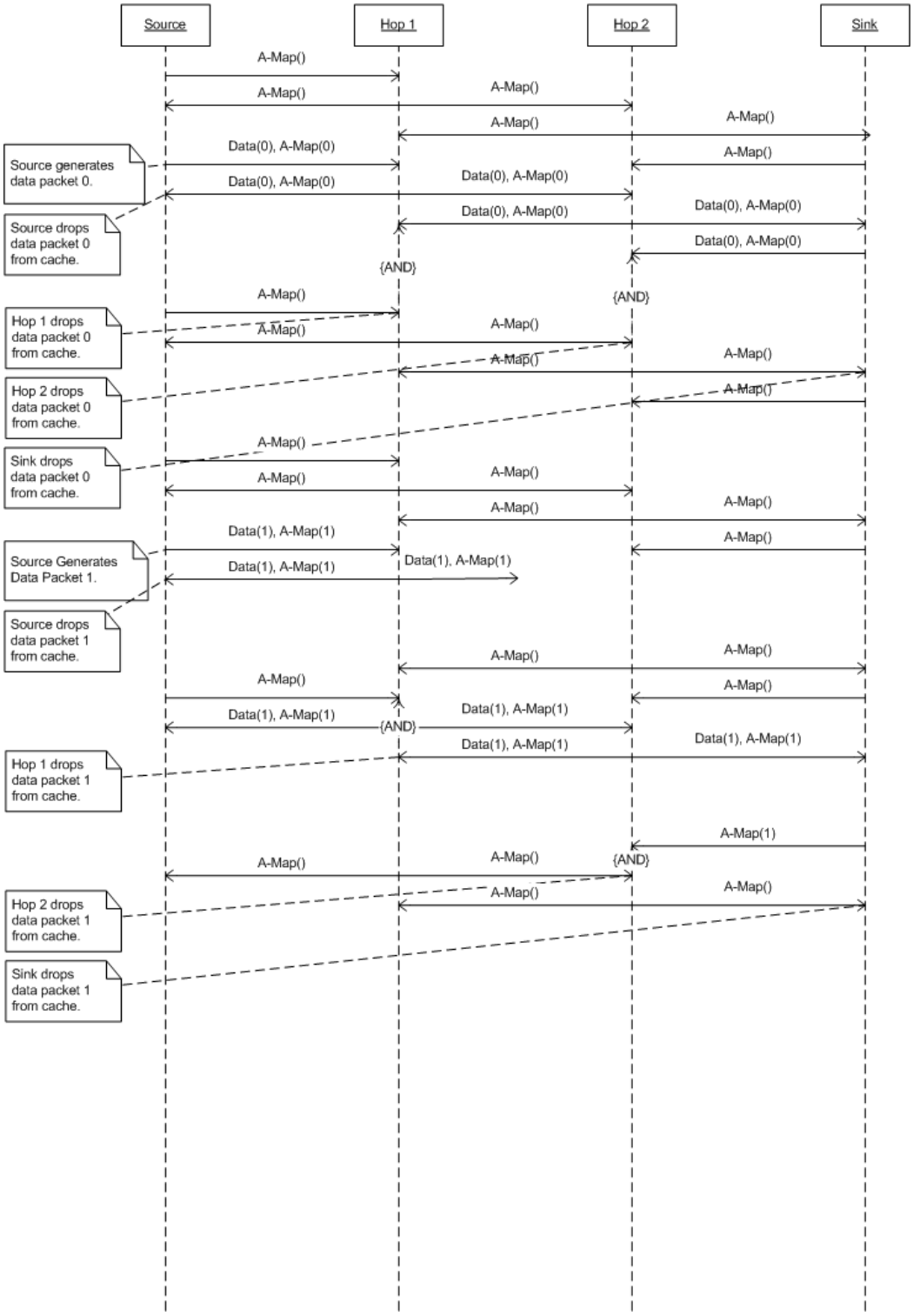
Annex 8: Congestion Control Message Sequence Chart



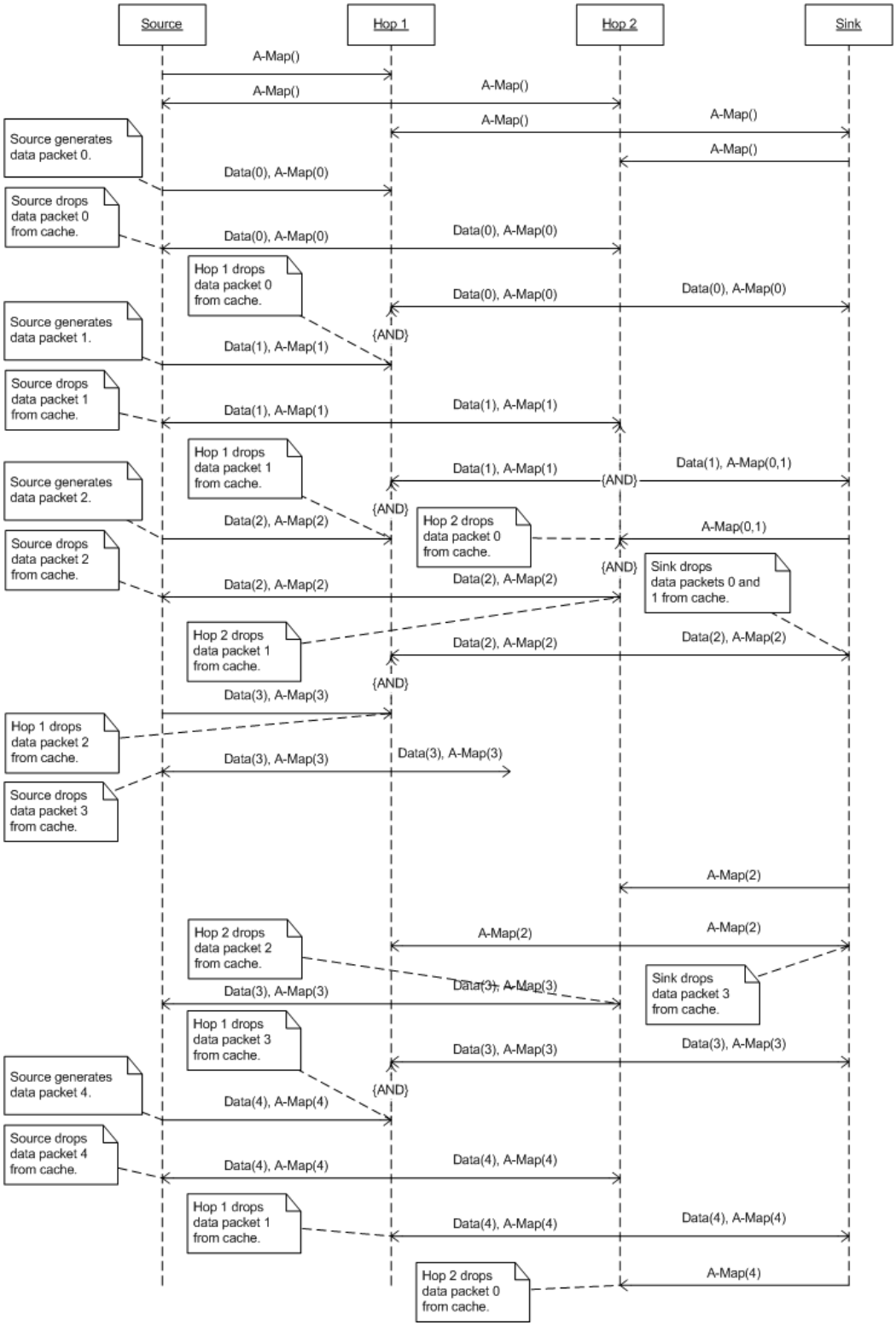
Annex 9: Reliability Message Sequence Chart (Simple Scenario)



Annex 10: Reliability Message Sequence Chart (with Piggybacking)



Annex 11: Reliability Message Sequence Chart (High Data Rate Scenario)



Annex 12: Example Sending Application Source Code

```
includes WMTP;

module SendingApplicationM {
  provides {
    interface StdControl;
  }
  uses {
    interface StdControl as WMTPControl;
    interface WMTPConnectionManager;
    interface WMTPSendMsg;
  }
} implementation {
  command result_t StdControl.init() {
    return call WMTPControl.init();
  }

  command result_t StdControl.start() {
    WMTPConnectionSpecification_t *CS;
    if ( call WMTPControl.start() != SUCCESS )
      return FAIL;
    if ( call WMTPConnectionManager.GetNewConnectionSpecification(
      &CS )
      != SUCCESS )
      return FAIL;
    CS->PathSpecification.PathType = WMTP_PATHTYPE_TOSMULTIHOP;
    CS->FeatureSpecification.QueueAvailabilityShaper.Active = TRUE;
    CS->FeatureSpecification.Throttling.Period = 1000;
    return call WMTPConnectionManager.OpenConnection( CS );
  }

  command result_t StdControl.stop() {
    return call WMTPControl.stop();
  }

  event result_t WMTPConnectionManager.ConnectionOpened(
    WMTPConnectionSpecification_t *CS ) {
    static WMTPPayload_t Payload;
    // Fill in first outgoing message.
    return call WMTPSendMsg.Send( CS, 0, &Payload );
  }

  event result_t WMTPConnectionManager.ConnectionReconfigured(
    WMTPConnectionSpecification_t *OldCS,
    WMTPConnectionSpecification_t *NewCS ) {
    return SUCCESS;
  }
  event result_t WMTPConnectionManager.ConnectionClosed(
    WMTPConnectionSpecification_t *CS ) {
    return SUCCESS;
  }

  event result_t WMTPSendMsg.ClearToSend(
    WMTPConnectionSpecification_t *CS ) {
    static WMTPPayload_t Payload;
    // Fill in outgoing message.
    return call WMTPSendMsg.Send( CS, 0, &Payload );
  }
}
```

Annex 13: Example Receiving Application Source Code

```
includes WMTP;

module ReceivingApplicationM {
  provides {
    interface StdControl;
  }
  uses {
    interface StdControl as WMTPControl;
    interface WMTPConnectionManager;
    interface WMTPReceiveMsg;
  }
} implementation {
  command result_t StdControl.init() {
    return call WMTPControl.init();
  }

  command result_t StdControl.start() {
    static WMTPServiceSpecification_t SS;
    if ( call WMTPControl.start() != SUCCESS )
      return FAIL;
    SS.Connectionless = TRUE;
    SS.ConnectionOriented = FALSE;
    SS.ServiceType = WMTP_SERVICETYPE_PACKETSINK;
    return call WMTPConnectionManager.RegisterService( &SS );
  }

  command result_t StdControl.stop() {
    return call WMTPControl.stop();
  }

  event result_t WMTPConnectionManager.ConnectionOpened(
    WMTPConnectionSpecification_t *CS ) {
    return SUCCESS;
  }

  event result_t WMTPConnectionManager.ConnectionReconfigured(
    WMTPConnectionSpecification_t *OldCS,
    WMTPConnectionSpecification_t *NewCS ) {
    return SUCCESS;
  }
  event result_t WMTPConnectionManager.ConnectionClosed(
    WMTPConnectionSpecification_t *CS ) {
    return SUCCESS;
  }

  event WMTPPayload_t *WMTPReceiveMsg.Receive(
    WMTPConnectionSpecification_t *CS,
    uint8_t Length,
    WMTPPayload_t *Msg ) {
    // Handle received messages.
    return Msg;
  }
}
```

Annex14: Quality-of-Service Reservation Log Excerpt

```
(...)
5:   WMTPSourceRoutedConnectionEstablishmentHandlerM: Opening connection
     for application 0.
5:   WMTPCoreM: Calculating QoS shortest delay for connection
     establishment handler 0 (Max Send Delay = 91).
5:   WMTPCoreM: The requested minimum delay of 200 maps into a level of 0
     but policy dictates a level of 2.
5:   WMTPCoreM: Shortest available delay is at level 2 (delay = 455).
5:   WMTPSourceRoutedConnectionEstablishmentHandlerM: QoS reservation
     delay: 200.
(...)
5:   WMTPCoreM: Reserving QoS resources for connection establishment
     handler 0 (Max Send Delay = 91).
5:   WMTPCoreM: The requested delay of 200 maps into a level of 0.
5:   WMTPCoreM: Made a reservation at level 0, slot 0.
(...)
2:   WMTPSourceRoutedConnectionEstablishmentHandlerM: Opening new non-
     local connection.
2:   WMTPCoreM: Calculating QoS shortest delay for connection
     establishment handler 0 (Max Send Delay = 83).
2:   WMTPCoreM: The requested minimum delay of 200 maps into a level of 0
     but policy dictates a level of 2.
2:   WMTPCoreM: Shortest available delay is at level 2 (delay = 415).
(...)
2:   WMTPSourceRoutedConnectionEstablishmentHandlerM: QoS reservation
     delay: 200; Accumulated delay: 400.
(...)
2:   WMTPCoreM: Reserving QoS resources for connection establishment
     handler 0 (Max Send Delay = 83).
2:   WMTPCoreM: The requested delay of 200 maps into a level of 0.
2:   WMTPCoreM: Made a reservation at level 0, slot 0.
(...)
1:   WMTPSourceRoutedConnectionEstablishmentHandlerM: Opening new non-
     local connection.
1:   WMTPCoreM: Calculating QoS shortest delay for connection
     establishment handler 0 (Max Send Delay = 81).
1:   WMTPCoreM: The requested minimum delay of 200 maps into a level of 0
     but policy dictates a level of 2.
1:   WMTPCoreM: Shortest available delay is at level 2 (delay = 405).
(...)
1:   WMTPSourceRoutedConnectionEstablishmentHandlerM: QoS reservation
     delay: 200; Accumulated delay: 600.
(...)
1:   WMTPCoreM: Reserving QoS resources for connection establishment
     handler 0 (Max Send Delay = 81).
1:   WMTPCoreM: The requested delay of 200 maps into a level of 0.
1:   WMTPCoreM: Made a reservation at level 0, slot 0.
(...)
0:   WMTPSourceRoutedConnectionEstablishmentHandlerM: Opening new local
     connection.
0:   WMTPSourceRoutedConnectionEstablishmentHandlerM: Sending keep-alive
     to confirm connection.
(...)
4:   WMTPSourceRoutedConnectionEstablishmentHandlerM: Opening connection
     for application 0.
4:   WMTPCoreM: Calculating QoS shortest delay for connection
     establishment handler 0 (Max Send Delay = 73).
```

```
4:   WMTPCoreM: The requested minimum delay of 200 maps into a level of 0
     but policy dictates a level of 2.
4:   WMTPCoreM: Shortest available delay is at level 2 (delay = 365).
4:   WMTPSourceRoutedConnectionEstablishmentHandlerM: QoS reservation
     delay: 200.
(...)
4:   WMTPCoreM: Reserving QoS resources for connection establishment
     handler 0 (Max Send Delay = 73).
4:   WMTPCoreM: The requested delay of 200 maps into a level of 0.
4:   WMTPCoreM: Made a reservation at level 0, slot 0.
(...)
1:   WMTPSourceRoutedConnectionEstablishmentHandlerM: Opening new non-
     local connection.
1:   WMTPCoreM: Calculating QoS shortest delay for connection
     establishment handler 0 (Max Send Delay = 70).
1:   WMTPCoreM: The requested minimum delay of 200 maps into a level of 0
     but policy dictates a level of 2.
1:   WMTPCoreM: No available slots left.
(...)
1:   WMTPSourceRoutedConnectionEstablishmentHandlerM: Failed to open
     connection.
(...)
4:   WMTPCoreM: Freeing QoS resources for connection establishment handler
     0.
(...)
```