ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS
School of Information Sciences and Technology
Department of Informatics

Congestion and Error Control for Multi-Party Content Distribution

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Charilaos Stais

Athens
October 2016

EPIGRAPH

The best way

to predict the future

is to create it.

—Abraham Lincoln

TABLE OF CONTENTS

Epigraph . . . iii

Table of Contents . . . iv

List of Figures . . . vi

List of Tables . . . vii

Acknowledgements . . . viii

Vita . . . xi

Abstract of the dissertation . . . xiii

Chapter 1 Introduction . . . 1
- 1.1 Motivation . . . 1
- 1.2 Contributions . . . 5
- 1.3 Dissertation outline . . . 7

Chapter 2 Background review . . . 8
- 2.1 Information-Centric Networking . . . 8
  - 2.1.1 The PSI architecture . . . 8
  - 2.1.2 Wide Area Multicast in PSI . . . 11
  - 2.1.3 Reliable Multicast . . . 13
- 2.2 Wireless Sensor Networks . . . 16
  - 2.2.1 Reliable Transport for Disaster Recovery . . . 16
- 2.3 BitTorrent . . . 21
  - 2.3.1 The Tracker Protocol . . . 21
  - 2.3.2 The Peer-wire Protocol . . . 22

Chapter 3 Reliable Multicast Transport for Information-Centric Networks . . . 29
- 3.1 Introduction . . . 29
- 3.2 Multicast Error Control for PSI . . . 32
  - 3.2.1 Overview . . . 32
  - 3.2.2 Communication setup . . . 32
  - 3.2.3 Initial content distribution . . . 35
  - 3.2.4 Recovery . . . 36
  - 3.2.5 Comparison with PGM . . . 37
- 3.3 Experimentation and Evaluation . . . 38
  - 3.3.1 Simulator Setup . . . 38
  - 3.3.2 Experimental Results . . . 39
- 3.4 Conclusion . . . 44

Chapter 4 Sink-Driven Reliable Transport for Wireless Sensor Networks . . 46
4.1 Introduction 46
4.2 Rationale and Related Work 48
4.3 Protocol Description 50
4.3.1 Connection establishment 51
4.3.2 Data exchange and idle 52
4.3.3 Connection control and release 54
4.3.4 Error recovery 55
4.3.5 Bandwidth and Congestion management 56
4.4 Performance evaluation 62
4.4.1 Experimental setup 62
4.4.2 Experimental results 64
4.5 Conclusion 72

Chapter 5 Streaming with BitTorrent 74
5.1 Introduction 74
5.2 Streaming Extensions 76
5.2.1 Fixed Size Window 76
5.2.2 High Priority Set 76
5.2.3 Stretching Window 77
5.2.4 An example of policies operation 78
5.3 Experimental Setup 79
5.3.1 BitTorrent Simulator 79
5.3.2 Network and content setup 80
5.4 Evaluation with player skipping 82
5.4.1 Assumptions and metrics 82
5.4.2 Piece loss 83
5.4.3 Prefetch time 84
5.4.4 Download time 86
5.4.5 Overall Evaluation 87
5.5 Evaluation with player stalling 87
5.5.1 Assumptions and metrics 87
5.5.2 Number of Stall Periods 89
5.5.3 Average Stall Duration 90
5.5.4 Download Time 90
5.5.5 Overall Evaluation 91
5.6 Conclusion 92

Chapter 6 Conclusions and future work 94

Appendix A Acronyms 97

Bibliography 99

LIST OF FIGURES

Figure 2.1: An example communication in PSI. 11
Figure 2.2: An example of a multicast tree with relay points. 12

Figure 3.1: An example of Bloom filter distribution. 34
Figure 3.2: Number of packets sent downstream by all network elements. . . 40
Figure 3.3: Number of recovery packets sent downstream by all network elements. 40
Figure 3.4: Number of recovery packets sent downstream by the publisher. . 41
Figure 3.5: Total number of packets transmitted by all network elements. . . 42
Figure 3.6: Total NAKs sent by subscribers and received by publisher. 42

Figure 4.1: Connection establishment and sensor information exchange. . . . 51
Figure 4.2: Start of data exchange stage. 53
Figure 4.3: Data exchange stage. 53
Figure 4.4: Data exchange and connection control stage. 54
Figure 4.5: Event sensor scenario - Mean Bandwidth Allocation and Usage (2% loss). 65
Figure 4.6: Event sensor scenario - Total Bandwidth Usage (2% loss). 66
Figure 4.7: Event sensor scenario - Rate control messages sent (2% loss). . . 66
Figure 4.8: Event sensor scenario - Bandwidth Allocation and Usage (5% loss). 67
Figure 4.9: Event sensor scenario - Rate control messages sent (5% loss). . . 68
Figure 4.10: Continuous sensor scenario - Mean Bandwidth Allocated to sensors. 68
Figure 4.11: Continuous sensor scenario - Total Bandwidth Used by sensors. . 69
Figure 4.12: Mixed sensor scenario - Mean Bandwidth Allocated to continuous sensors (2% loss). 71
Figure 4.13: Mixed sensor scenario - Mean Bandwidth Allocated to event sensors (2% loss). 71
Figure 4.14: Mixed sensor scenario - Mean Bandwidth Used by event sensors (2% loss). 72

Figure 5.1: Example of windows in the different policies. 79
Figure 5.2: Piece loss for 2% window size (%). 83
Figure 5.3: Piece loss for 8% window size (%). 84
Figure 5.4: Prefetch time for 2% window size (sec). 85
Figure 5.5: Prefetch time for 8% window size (sec). 85
Figure 5.6: Download time for 2% window size (sec). 86
Figure 5.7: Download time for 8% window size (sec). 86
Figure 5.8: Number of Stall Periods per download. 89
Figure 5.9: Average Stall Duration per download (sec). 90
Figure 5.10: Average download and seeding time per download (sec). 91

LIST OF TABLES

Table 4.1: Experimentation parameters 63

Table 5.1: Simulation parameters 81

ACKNOWLEDGEMENTS

A long time ago, when I started my studies in the Department of Informatics at the Athens University of Economics and Business, I could not imagine that I would become a PhD Candidate. My internship in the Laboratories of the Department of Informatics and also my involvement in the Athens Wireless Metropolitan Network gave me a sense of identity. At that point, I said to myself "If you want a career that fulfills you, you need to focus not only on your qualifications, but on your interests." That is what I did. After my graduation I started studying for a MSc in Information Systems. Furthermore, I started planning on having a PhD in Computer Networking. When I completed my postgraduate studies, I decided to talk to Mr. George Xylomenos about my plans and ask him whether I could get actively involved in researching. There was a very positive response to my ideas – Mr. George Xylomenos seemed to be pleased with it. My educational journey towards a greater understanding of Computer Network methodologies and technologies had just begun.

During the course of my research in the university, my family, my friends, and also my colleagues and my professors were a great support. First of all, I would like to thank Associate Professor George Xylomenos, Professor George Polyzos, and Assistant Professor Giannis Marias. Mr. George Xylomenos, my supervisor, always guided me through the more complicated issues I had to deal with. I really admire him for his responsiveness to questioning. Mr. George Polyzos encouraged me to figure out my "Why" – the real problem I want to solve. His observations on my research activities helped me to answer the fundamental question "Why?" and thus to increase my motivations. Lastly, Mr. Giannis Marias contributed to our effort from a Security perspective, improving the overall quality.

Moreover, I am grateful to all my colleagues in Mobile Multimedia Laboratory for all that we have done together the last years. Dr. Konstantinos Katsaros, Dr. Pantelis Frangoudis, Dr. Nikos Fotiou, Xenofon Vasilakos, Dr. Evangelos Douros, Dr. Christos Tsilopoulos, and Yannis Thomas significantly contributed to my work by brainstorming, cooperation, and constructive feedback. Konstantinos Katsaros was a mentor on my first steps in research, while Pantelis Frangoudis was the best person to discuss almost every topic in Computer Science. Nikos Fotiou, Evangelos Douros and Xenofon Vasilakos were a great company as lab roommates. With Christos Tsilopoulos and Yannis Thomas, both colleagues and friends, we spent many hours on networking problems, brain-storming and relaxing.

I would like to thank also Nausika Kokkini, Lina Kanelopoulou, and Kostas Makedos for their enjoyable company and the pleasant days.

I must also express my thanks for all the help my close friends gave me. Yannis Sazonof, Elena Tchejourova, George Papaioannou, Maria-Lida Menexi, Danai Argyrakopoulou, George Arethas, Eleni Paflia and Panagiotis Stampernas are always on my side.

Of course, I would like to thank my family. My parents, Aristidis and Eleni, always made me feel proud of my accomplishments and gave me the confidence I needed to be out in the world on my own. My aunts, Panagiota and Ligeri Zacharaki, were there for me when I needed them although we have not always been close.

Last, but most valuable person, I am especially grateful to my partner, Christina Aretha, with whom we share many common interests. She always listens to me patiently and encourages me in everything I want to do. She believed in me even at times I had lost my faith. This dissertation would not be the same without Christina's contribution. Christina, thank you for having great faith in me!

VITA

2006	B. Sc. in Computer Science, Athens University of Economics and Business, Greece
2008	M. Sc. in Information Systems, Athens University of Economics and Business, Greece
2016	Ph. D. in Computer Science, Athens University of Economics and Business, Greece

PUBLICATIONS

Journal Papers

C. Stais, G. Xylomenos and A. Voulimeneas, **"A Reliable Multicast Transport Protocol for Information-Centric Networks"**, *Journal of Network and Applications*, Elsevier, Vol. 50, April 2015, pp. 92-100.

Refereed International Conference and Workshop Papers

C. Stais, G. Xylomenos and E. Zafeiratos, **"RT-SENMOS: Sink-Driven Congestion and Error Control for Sensor Networks"**, in *IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, 2016.

C. Stais and G. Xylomenos, **"RT-SENMOS: Reliable Transport for Sensor Networks with Mobile Sinks"**, in *DMSS Workshop at the IEEE Symposium on Computers and Communications (ISCC)*, pp. 105-110, 2015.

C. Stais, G. Xylomenos and G. Marias, **"Sink Controlled Reliable Transport for Disaster Recovery"**, in *ACM International Conference on Pervasive Technologies Related to Assistive Environments (PETRA)*, pp. 61:1-61:4, 2014.

C. Stais, A. Voulimeneas and G. Xylomenos, **"Towards an Error Control Scheme for a Publish/Subscribe Network"**, in *IEEE International Conference on Communications (ICC)*, pp. 3743-3747, 2013.

C. Stais, Y. Thomas, G. Xylomenos and C. Tsilopoulos, **"Networked Music Performance over Information-Centric Networks"**, in *IEEE ICC Workshop on Immersive & Interactive Multimedia Communications over the Future Internet (IIMC)*, pp. 647-651, 2013.

C. Stais and G. Xylomenos, **"Realistic Media Streaming over BitTorrent"**, in *Future Network and Mobile Summit*, pp. 1-8, 2012.

C. Stais, G. Xylomenos and A. Archontovasilis, **"A comparison of streaming extensions to BitTorrent"**, in *IEEE Symposium on Computers and Communications (ISCC)*, pp. 1068-1073, 2011.

V. Giannaki, X. Vasilakos, C. Stais, G.C. Polyzos and G. Xylomenos, **"Supporting Mobility in a Publish-Subscribe Internetwork Architecture"**, in *IEEE Symposium on Computers and Communications (ISCC)*, pp. 1030-1032, 2011.

C. Stais, D. Diamantis, C. Aretha and G. Xylomenos, **"VoPSI: Voice over a Publish-Subscribe Internetwork"**, in *Future Network and Mobile Summit*, pp. 1-8, 2011.

K. Katsaros, C. Stais, G. Xylomenos and G.C. Polyzos, **"On the incremental deployment of overlay information centric networks"**, in *Future Network and Mobile Summit*, pp. 1-8, 2010.

K. Katsaros, V.P. Kemerlis, C. Stais and G. Xylomenos, **"A BitTorrent Module for the OMNeT++ Simulator"**, *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 1-10, 2009.

Refereed Demos

G. Parisis, B. Tagger, D. Trossen, C. Tsilopoulos, Y. Thomas, C.Stais and G. Xylomenos, **"Demonstrating Usage Diversity Over an Information-centric Network"**, in *IEEE INFOCOM*, pp. 27-28, 2013.

G. Parisis, B. Tagger, D. Trossen, D. Syrivelis, P. Flegkas, C. Stais, C. Tsilopoulos and G. Xylomenos, **"Demonstrating an Information-Centric Network in an International Testbed"**, in *International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities (TRIDENTCOM)*, pp. 400-402, 2012.

ABSTRACT OF THE DISSERTATION

Congestion and Error Control for Multi-Party Content Distribution

by

Charilaos Stais

Doctor of Philosophy in Computer Science

Athens University of Economics and Business, Athens, 2016

Associate Professor George Xylomenos, Chair

We have recently experienced a considerable research drive targeting new architectures for the next generation of the Internet, which drop the current design in order to improve the efficiency of large-scale content delivery, as well as other emerging applications, such as machine to machine communications. These efforts have brought to the forefront alternative communication paradigms: some future Internet architectures are based on native one-to-many (multicast) communication to make content distribution more efficient, most sensor network architectures focus on many-to-one communication from the sensors to the sink, while file exchange applications like BitTorrent make use of many-to-many communication. The design and implementation of the omnipresent Internet architecture however leans on the traditional telephone network idea, where there are only two parties wishing to communicate. Extending the Internet model to multiparty communication requires considerable engineering effort and costs, as such a radical redesign brings to the forefront the need to design new protocols at all layers of the networking stack. This dissertation focuses on the design of transport and application layer protocols that can support different types of multi-party content distribution applications.

The first part of this dissertation deals with the issue of error control for multicast communication, as applied to the Information Centric Networking (ICN) context. Despite the fact there is a lot of work on reliable transport for IP Multicast, none has managed to fully solve the problem of feedback implosion towards the sender as group size grows, while existing solutions are not amenable to the special characteristics of ICN and its native multicast mechanisms. We have proposed a suitable solution for ICN (RMTPSI) and we compared it against one of the most well-known mechanisms for IP Multicast, PGM. Our simulations show that RMTPSI is more efficient than PGM, while requiring the same time to complete a reliable transfer as PGM; specifically, RMTPSI requires 2.9% to 10.2% fewer downstream transmissions and 3.5% to 12.1% fewer total transmissions than PGM.

In the second part of this dissertation we move from one-to-many communication to the reverse problem, reliable many-to-one communication, a very common problem in the sensor networks area, where large numbers of sensors transmit data to a single sink node. As sensors become cheaper and more powerful, sensor networks face the challenge of controlling all these devices in a efficient manner, which naturally leads to a sink-driven transport scheme. We have proposed a sink-driven reliable transport protocol for sensor networks which places the responsibility for transmission rate allocation, congestion avoidance and error control to the sink. The protocol is implemented at the application level, which allows it to be used without supervisor privileges, as well as enabling the application at hand to fine tune its operational parameters. Our protocol was implemented and tested over a real network with emulated losses and compared against a well-known sink driven approach, RCRT. The results show that our approach fully exploits the available bandwidth in all cases, while RCRT only manages to use 60% to 90% of it. Finally, the overhead of protocol operation, in terms of required computation resources and control messages, is much lower for our approach.

Finally, in the third part we investigate the many-to-many communication model for use in streaming multimedia services. We approach the problem by evaluating extensions to the well-known BitTorrent application, in order to meet the requirements of video streaming. We implemented a number of proposed solutions to this problem to our own comprehensive packet level BitTorrent Simulator and evaluated their behavior with streaming video from both the network and the user experience points of view, considering players that either skip over missing pieces or stall until they arrive. We concluded that the simplest proposed solution is no worse than the others under a very wide range of parameters, and that its user-level performance is actually quite acceptable for real use, when properly configured.

Chapter 1

Introduction

This chapter is a general introduction to the dissertation. Section 1.1 describes the context of this work, identifying the major issues motivating us, with Section 1.2 highlighting the original contributions of this work on congestion and error control for content distribution. Section 1.3 provides an outline of the dissertation structure.

1.1 Motivation

When computing moved from a stand-alone machine to a connected model, a new era began along with many hard problems and challenges. In the late 1950s a military funded research program by U.S.A. and Canada succeeded in interconnecting radar systems. This was the first instance of an interconnected computational system. In the next few years, great effort was spent on developing a more generic computer network. The result was the famous ARPANET. Having ARPANET as a starting point, many extensions and optimizations were introduced, among which some of the most important steps were towards providing transmission reliability and handling of network congestion. These were critical issues in a network which, unlike the telephony network, had unpredictable traffic demands and consisted of relatively unreliable hosts and links.

The organic growth of the Internet is largely due to the ubiquitous adoption of the TCP/IP protocol stack. TCP/IP based networks were designed to simply forward traffic between pairs of communicating end hosts, following the prevailing communication pattern of previous networks, including the telephone network. However, communication patterns have evolved since then, and the use of the Internet has shifted towards *information-centric* services and applications, such as content delivery networks (CDNs), cloud computing services and peer-to-peer (P2P) file sharing applications like BitTorrent. In these services, in sharp contrast to the underlying Internet model, the focus is on the information itself rather than on the specific end hosts providing or consuming it. Hence, these services are implemented as overlay solutions on top of the information-agnostic network substrate [1]. BitTorrent was one of the first widely adopted and used overlay schemes, but we can find many other solutions in the bibliography that add some structure in the overlay to improve its performance, as well as decoupling the communication substrate from the actual application used. The most famous of them are Pastry [2], Tapestry [3], Scribe [4] and Canon [5].

While the development of this overlay functionality has in many cases yielded important benefits to end users, at the same time it has revived the problems incurred by the mismatch between the prevailing communication patterns and the underlying network model. This is especially evident in the case of P2P file-sharing applications, in which end users benefit by the peer-assisted nature of content-distribution in terms of perceived download times and content providers decrease their resource requirements.

However, due to the lack of a suitable underlying network model, these benefits have come at the cost of excessive traffic load for network operators. Indeed, by following the end-to-end communication model, these applications have plagued the Internet with redundant unicast transmissions, e.g. when many nearby nodes individually download the exact same data from a faraway node, since they make their choices without any co-ordination [6].

While the research community was focusing on development of overlay networks, a new trend came up. At first, it was expressed through the desire to decouple host identities from host locations, as in the Routing on Flat Labels (ROFL) scheme [7]. The next steps were more aggressive and included the partial or total redesign of existing Internet infrastructure, from redesigning the naming in the current Internet infrastructure in the Data-Oriented Network Architecture (DONA) [8] to the more radical Publish-Subscribe Internet Routing Paradigm (PSIRP) [9], Content-centric Networking (CCN) [10] and Named-Data Networking (NDN) [11] projects. Many research projects followed that idea, e.g. 4WARD [12], SAIL [13], COMET [14] and CONVERGENCE [15]. While previous overlay solutions tried to improve network efficiency on top of the IPv4 network stack, these so-called “clean slate” approaches took all the benefits of overlay solutions and tried to introduce new network elements with native support of these characteristics. All these initiatives produced different network architectures, but all were based on the same main idea: information is in the center of attention. This idea formed the Information-Centric Networking (ICN) research area, which encompasses all those projects.

The PSIRP project and its successor, Publish Subscribe Internet Technology (PURSUIT) [16], introduced the Publish-Subscribe Internetwork (PSI) architecture. This new architecture promotes the information-centric point of view and redesigns the network stack and its elements, including routing and forwarding functionality. The functional model of PSI focuses on information retrieval through a network-based mechanism providing privacy, security and isolation as its main aspects, relying on native multicast support for content distribution.

As the number of connected devices to the Internet dramatically increases (smart devices are expected to reach 15 billion until 2020) and bandwidths increase, traffic constantly grows (nearly doubled every year from 2010 to 2014), leading to a constant battle with network congestion. In addition to congestion losses, the proliferation of wireless and mobile devices and bandwidth hungry applications, along with the continuous arrival of new subscribers, introduce additional loss sources. The emerging field of the Internet of Things (IoT) aims to integrate huge numbers of autonomous devices to the Internet, thus connecting Wireless Sensor Networks (WSNs) to the Internet at large. WSNs with their unusual communication model, where many sensors push data to a single sink, bring their own transport and error control issues.

As a result of these developments, reliable data transfer is more important than ever for the Internet. In current networks, we may encounter different setups based on the needs of each application; one-to-many, many-to-one and many-to-many. All of them share the same congestion control and reliability needs, but the implementation of appropriate schemes differs for each one. In other words, there is no common mechanism able to adapt to each use case without any changes. Each situation has to be studied independently, taking into account its special characteristics. This thesis therefore attacks these problems in three different environments, adapting existing solutions to new settings.

1.2 Contributions

The evolution of networking from the simple communication between two machines to a telephony-like computer network and more complex Internet-like solutions that involve billions of devices and network entities raises the open problem of efficient content delivery at high speeds with the number of errors kept as low as possible. Content delivery may take place in different scenarios; one-to-many, many-to-one and many-to-many. The main goal of this dissertation is to investigate potential approaches and propose a proper solution for each content delivery setup. In order to be more specific in our work, we approached each such case from the point of view of one of the most representative applications.

First, the one-to-many case can be handled by multicasting data from one sender to many recipients. Multicast has not seen widespread use in the current Internet infrastructure, due to technical and business obstacles to its deployment, but it is the main aspect of almost every new architecture proposal. In Chapter 3 we rely on the newly introduced PSI architecture and propose a suitable solution for error control management for on-demand applications and services. There is a lot of previous work on the field of reliable multicast ([17], [18], [19]), but none of it is able to operate over a PSI network out-of-the-box. For this reason, we have designed an appropriate mechanism, named RMTPSI [20] and implemented it on top of PSI. Our mechanism uses negative acknowledgements to manage feedback to the sender. We added aggregation by using relay points (RPs) and exploited their position in the network topology to create smaller multicast trees and avoid false positive decisions in the forwarding mechanism. Regarding packet recovery, we perform recovery in rounds, again via multicast. We compare and contrast our RMTPSI approach against a modified for PSI version of Pragmatic General Multicast (PGM) [19], a similar approach for IP Multicast, and compare their performance. Our simulations show that RMTPSI is more efficient than PGM, while requiring the same time to complete a reliable transfer; specifically, RMTPSI requires 2.9% to 10.2% fewer downstream transmissions and 3.5% to 12.1% fewer total transmissions than PGM.

Second, in Chapter 4 we deal with the reliability problem in the case of WSNs, where multiple sensors attempt to transmit data to a single sink, thus leading to traffic convergence around the sink. In the emerging IoT field we expect having multiple types of sensors and a more heterogeneous traffic mix than in the past, with each sensor type requiring custom handling. As the demands of different IoT applications may be totally different, we use as an example an emergency response scenario, in which we distinguish two main types of sensors; event and continuous. Event sensors are triggered by an event and send a specific piece of information, while continuous ones send data in predefined time intervals. In addition, we handle sensors according to the application they serve, e.g. a video camera is able to transmit only at a predefined set of rates. Our mechanism, called RT-SENMOS [21], takes into account all these parameters and tries to provide the best possible user experience under the current circumstances of the network. Our protocol is implemented and tested over a real network with emulated losses and it is compared against a well-known sink driven approach, Rate Controlled Reliable Transport (RCRT) [22]. The results show that our approach fully exploits the available bandwidth in all cases, while RCRT only manages to exploit 60% to 90% of it. Finally, the overhead of protocol operation, in terms of computing resource and control message overhead, is much lower for our approach, while our protocol also adapts much faster to prevailing network conditions.

Third, in Chapter 5 we move to the many-to-many paradigm, in which we study the suitability of BitTorrent to serve streaming applications. BitTorrent divides content into pieces and searches for the rarest one among the connected peers. It is obvious that this technique will not produce a reproducible content stream, unless all content is downloaded. As a result, the piece selection mechanism of BitTorrent has to be modified to take into account the sequencing requirements of streaming media. However, the basic aspects of BitTorrent cannot be totally removed, as the selection of the rarest piece increases the probability for a peer to receive the entire content, even if some peers leave the swarm. We implemented a number of proposed solutions to this problem using our own BitTorrent Simulator [23] and evaluated their behavior from the user experience point of view, considering players that either skip over missing pieces or stall until they arrive. We conclude that the simplest solution is no worse than the others, and that its user-level performance is actually quite acceptable.

1.3 Dissertation outline

The remainder of the dissertation is organized as follows. Chapter 2 provides background and related work regarding the PSI architecture, transport protocols for WSNs and a description of BitTorrent functionality. In Chapter 3 we present our reliable one-to-many transport mechanism for the PSI architecture and compare it against an alternative. In Chapter 4 we move to sensor network transport and present a reliability protocol providing congestion management and error control for many-to-one communication and compare it against an alternative. In Chapter 5 we focus on extensions to BitTorrent that enable it to support video streaming and compare them against each other. Finally, we conclude in Chapter 6.

Chapter 2

Background review

This chapter sets the necessary background for each of the following chapters. Specifically, in Section 2.1 we discuss the PSI architecture, its support for native multicast and past work in reliable multicast transport. In Section 2.2 we outline past work in congestion and error control for wireless sensor networks. Finally, in Section 2.3 we describe the BitTorrent protocol and its mechanisms.

2.1 Information-Centric Networking

2.1.1 The PSI architecture

A publish/subscribe architecture consists of three main elements: publishers, subscribers, and an event notification service, also known as a Rendez-Vous network, consisting of Rendez-Vous Points (RVPs) [24]. The publishers are the content owners who offer their content to potential consumers. To announce the availability of the content, the publishers advertise it to the responsible RVP by issuing publication messages. The subscribers are the content consumers who express their interest in specific content items by issuing subscription messages. Information indicating the desired content items is included in the publication and subscription messages.

PSI is an instantiation of such a public/subscribe architecture in a networking context: publishers and subscribers are located at network nodes and exchange data via publish and subscribe primitives which are facilitated by a distributed rendezvous function. Data items are identified by a Scope Identifier (SId) in combination with a Rendez-Vous Identifier (RId). The SId identifies a collection of content items and is mapped to the RVP responsible for this particular collection, possibly using a Distributed Hash Table [25]. The RId identifies a specific content item in the collection and is determined by the application issuing the publication message. The scoping mechanism in PSI has been developed with the aim of limiting access to content. In order for a subscriber to obtain a desired content item, he needs to have access to the scope in which it is located. Each scope may have different governance and access control rules. Moreover, a hierarchical structure, where parent-children relationships exist, is employed [26]. In PSI, scopes can be either physical, such as a corporate network, or logical, such as a social network.

A subscriber needs to be aware of the SId and RId of a desired content item in order to issue a subscription message for it. When a subscription message arrives at the responsible RVP, based on the SId included in the subscription, the RVP checks whether the subscriber is allowed access to the scope. If so, it determines which publishers can satisfy the subscriber's request and then communicates with the Topology Manager (TM) to request a suitable forwarding path from a publisher to the subscriber. The TM, which is either a service in the same machine or a stand-alone server, maintains network topology information, i.e. the interconnection between routers and hosts, discovered via a link-state routing protocol. The TM can thus calculate a path between the publisher and the subscriber; when multiple subscribers are interested in the same content item, a multicast tree containing the publisher and all subscribers is calculated instead.

The path calculated by the TM is described by a Bloom filter, adopting the approach of Line-speed Publish Subscribe Internetworking (LIPSIN) [27]. Bloom filters constitute probabilistic representations of sets where each element is encoded as a string of zeroes and ones using a set of hash functions. A set is represented as the logical OR of all its elements. A Bloom filter is included in the header of each packet to describe the labels of all the links that are part of the path, whether it is a unicast route or a multicast one. When a packet arrives at a router, the router looks at the Bloom filter and determines to which of its outgoing links (possibly, more than one) it will have to forward the packet, by performing a logical AND between the label of each link and the in-packet Bloom filter. This technique supports native multicast, since the Bloom filter in the packet header may represent an entire multicast tree, rather than a linear unicast path; the Bloom filter is simply a set of link labels. Note that these link labels *must* be unidirectional, as otherwise packets would loop. As a result, the encoded paths, whether unicast or multicast, are also unidirectional, from the publisher towards the subscriber(s).

Figure 2.1 summarizes the above procedures, starting with the announcement of data availability and ending with data reception. First, a publisher issues a publication under a certain SId/RId to the corresponding RVP (step 1). A subscriber that is aware of this SId/RId pair subscribes to it (step 2). After the RVP corresponding to the requested SId receives the subscription, it communicates with the TM in order

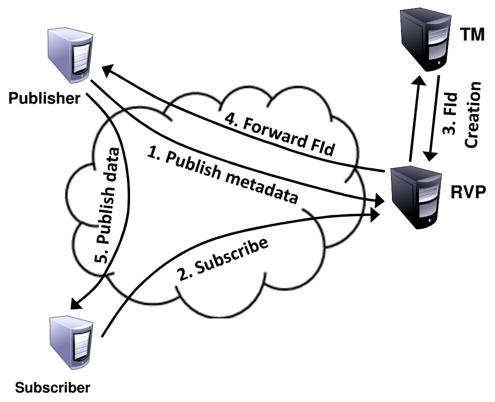


Figure 2.1: An example communication in PSI.

to retrieve a suitable Bloom filter for data dissemination from the publisher to the subscriber (step 3). Once the RVP gets the Bloom filter, it forwards it to the publisher (step 4). Finally, data are delivered to the subscriber using the Bloom filter (step 5).

2.1.2 Wide Area Multicast in PSI

The main disadvantage of Bloom filters is *false positives*: since Bloom filters are probabilistic representations of sets, as more elements are added to a Bloom filter, it becomes more likely that the Bloom filter will match elements that were not really added to it. When Bloom filters are used to encode routes as in PSI, as more links are added to a route, it is more likely that random links will match the Bloom filter. If such a link happens to be on the path taken by a packet, the packet will be needlessly transmitted over that link. In some cases, a packet may even end up making loops inside the network. The extent of this problem depends on how many links are encoded into the set. In [28], the authors propose that the number of ones in the Bloom filter should not exceed 40% of the total bits, meaning that with reasonably sized Bloom filters (since they must fit within packet headers) we cannot represent very large groups or very long paths.

In order to scale this scheme to larger multicast groups, we exploit the concept of Bloom filter switching at designated Relay Points (RPs) [29]. RPs are routers that replace the Bloom filter inside a packet with a new one before forwarding the packet. When a packet arrives, the RP checks in its database if an entry for the combination of the SId and the RId of the packet exists and, if so, replaces the Bloom filter in the packet with a stored one. When the TM constructs the initial Bloom filter, it pays attention to the ratio of ones in it. If the ratio exceeds 40%, it resorts to RPs, which are selected during a breadth-first traversal of the entire multicast tree. The TM therefore constructs Bloom filters from the publisher to the RPs and from these RPs to the subscribers or to new RPs, if needed. The RPs and the Bloom filters are returned to the RVP, which informs the selected RPs how to switch filters when they receive a packet with a specified SId/RId.

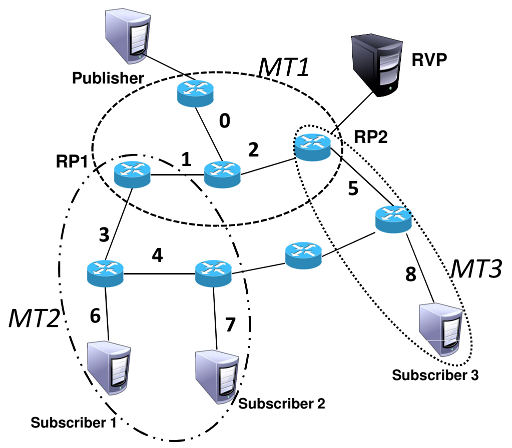


Figure 2.2: An example of a multicast tree with relay points.

Figure 2.2 presents an example of this approach with one Publisher and three Subscribers. The multicast tree is broken in three subtrees, MT1 to MT3, connected via two RPs, RP1 and RP2. MT1 starts from the Publisher and ends at the two RPs, MT2 starts at RP1 and ends at Subscribers 1 and 2, and MT3 starts at RP2 and ends at Subscriber 3. Messages are initially transmitted by the Publisher using the Bloom filter for MT1. Upon arrival at RP1 (RP2), this Bloom filter is replaced by the corresponding one for MT2 (MT3). As a result, each RP needs to store a mapping from a SId/RId pair to a new Bloom Filter. Note that we are using the SId and RId instead of the Bloom filter itself to determine the next Bloom filter at RPs, so as to allow the Bloom filters between two RPs to evolve due to link additions and deletions, without updating the switching information at all RPs.

2.1.3 Reliable Multicast

Even though IP multicast never got widely deployed, considerable work was dedicated to transport layer protocols for reliable multicast. The Reliable Multicast Transport Protocol (RMTP) [17] is a receiver-driven reliable transport scheme for non-real-time multicast content delivery. It relies on selective acknowledgments (ACKs), possibly indicating multiple lost packets, which are periodically sent from each receiver towards the sender. In order to avoid sender implosion due to ACKs, a set of receivers, called Designated Receivers (DR), aggregate the ACK state and forward it upstream, to higher level DRs or to the sender. Subsequently, the retransmissions from the sender are forwarded by each DR towards its group members and lower level DRs. Retransmissions may be either multicasted or unicasted, depending on the number of receivers that have lost the corresponding packet. In order to map this scheme to PSI, the TM should use the DRs as RPs in order to extend the reach of the multicast tree, but this would lead to suboptimal routing, since packets would not use the best path in either direction due to the need to select actual receivers as RPs; additional RPs would be needed if there were no receivers at appropriate points in the multicast tree.

The Scalable Reliable Multicast (SRM) [18] approach on the other hand relies on the fact that in IP multicast anyone can send to a multicast group, hence receivers can multicast their Negative Acknowledgments (NAKs) for missing data so as to reach other nodes that could retransmit them. By limiting the scope of NAKs to a few hops, ideally only nearby receivers will attempt to respond with the missing data. By randomizing the response time to these NAKs and locally multicasting the missing data, the first receiver to respond will silence all others. This approach does not guarantee reliability and is very hard to map to PSI, since it relies on bidirectional multicast trees to allow each receiver to multicast NAKs and retransmissions. Since in PSI multicasting is unidirectional, this would require creating separate Bloom filters from each subscriber to all other nodes within a subtree, which would place a much heavier load to the TM.

Finally, in the Pragmatic General Multicast (PGM) [19] approach, a receiver in the multicast group is guaranteed to either receive all data packets (from their original transmission or a retransmission), or to be able to detect unrecoverable data packet loss. PGM relies on NAKs to report missing packets and a hierarchy of PGM-enabled routers, called Network Elements (NEs), deployed throughout the multicast tree to aggregate feedback from the receivers towards the sender. Essentially, each NE is responsible for the subtree rooted at itself and extending downstream up to either the receivers or the downstream NEs. When a packet loss is detected by a receiver in PGM, it unicasts a NAK towards its parent NE after a random waiting period. The parent NE on reception of the NAK multicasts a NAK Confirmation (NCF) to its subtree, so as to suppress NAKs for the same lost packet from other receivers, and notes the packet number requested by the NAK. The NE then pushes the NAK towards its own parent, which in turn multicasts an NCF to its own tree, and so on, until the NAK reaches the source, in which case the missing packet is retransmitted downstream. Only NEs that have noted that they received a NAK for this packet forward it downstream, therefore retransmissions only reach the subtrees where at least one receiver has reported that particular packet to be lost. To further reduce the number of NAKs, an NE can create NAK packets which indicate multiple missing packets, thus aggregating individual NAKs.

PGM can be easily mapped to the PSI architecture, by simply using the RPs as the NEs. The only additional requirement is to provide each receiver and RP with a Bloom filter for reaching its parent RP or the publisher, so as to transmit NAKs; NCFs and retransmissions rely on the multicast Bloom filter used for the original packet transmissions. While PGM in its pure form is most appropriate for semi-reliable continuous data transmission, with minor modifications it can be made to operate in fully reliable mode. Since PGM is, essentially, a more sophisticated version of RMTP, in that it adds NAK suppression via NCFs to the hierarchical control scheme of RMTP, in this work we focus on comparing our approach with PGM only. PGM has several other features, such as support for retransmissions from local caching nodes, recovery based on forward error correction and congestion control. These features will not concern us further, as they can be integrated to our own error control scheme in the same manner as with PGM.

2.2 Wireless Sensor Networks

Since the early 2000's, a significant amount of research has been devoted to Wireless Sensor Networks (WSNs). Congestion control, error recovery and routing in various types of networks (e.g. ad-hoc, sink-driven) has been exhaustively studied, as wireless sensor networks introduce numerous novel challenges due to the nature of the underlying network: the topology may vary over time due to mobility, entities may connect and disconnect at unpredictable times and any predictions of the network status are, if not totally infeasible, very difficult. A very interesting aspect of WSNs is that their typical use involves a large number of sensor nodes communicating with a single sink node, either directly or via other nodes (sensors or not). This many-to-one communication model requires different assumptions than those made for one-to-one transport protocols, such as TCP. In this section we will discuss existing work on reliable transport for WSNs, also assessing their suitability for disaster recovery scenarios.

2.2.1 Reliable Transport for Disaster Recovery

In disaster areas with pre-existing WSNs, the network may only be partially connected, as some sensors may have failed. Therefore, a rescuer may have to roam around the disaster area to receive information from the surviving sensors. As speed is essential in disaster recovery scenarios, transport protocols for WSNs in such scenarios must be able to quickly connect to sensors, get as much information as possible, and adapt to changing conditions in the network as the rescuer roams.

According to the taxonomy in [22], transport protocols for WSNs can be classified based on their approach to reliability and congestion control: a protocol may offer unreliable or reliable service and a protocol may offer no congestion control, distributed congestion control or centralized congestion control. Reliability can be further subdivided to hop-by-hop reliability via retransmissions, as in RMST [30], end-to-end reliability via retransmissions, as in RCRT [22] and STCP [31], and forward reliability without retransmissions, as in ReInForM [32]. While the hop-by-hop reliability of RMST is useful for a wireless environment with losses due to bad link conditions, most wireless networks can retransmit lost packets at the link layer, if needed [30]. STCP concentrates on congestion induced losses, which are handled end-to-end, coupled with congestion control. The use of multiple transmissions without NAKs, as in ReInForM, requires a well-connected sensor network, which is unlikely in cases of disaster recovery applications. Considering the fact that end-to-end retransmissions do not require any co-operation from intermediate nodes, we will focus only on this kind of recovery scheme.

We can also distinguish WSN transport protocols based on who is responsible for congestion control. In sensor-driven protocols, each sensor sending data is responsible for congestion detection and error correction in its own data stream. In this approach, the sink simply receives data and reports packet losses (through ACKs or NAKs); the sink is not involved in congestion management decisions. The advantage of this technique is that the sink is very simple. However, this means that the sensors must be more complex as they must make all the decisions. The other option is to make the sink drive congestion control, using its awareness of all simultaneous transport sessions to co-ordinate the sensors. In this case, the sink is complex and the sensors are simple; in addition to its cost benefits, this choice allows different applications to be supported by suitably modifying the sink. This is of particular interest to disaster recovery situations, where existing sensors that may have been intended for a different application (e.g., security) may be reused to assist rescuers. For this reason, we will focus only on transport protocols where the sink is in control.

Another way to classify transport protocols is according to the method they use for congestion detection. The main methods are observing packet losses, queue lengths and packet delays, either in absolute terms or in relative terms (i.e., how these metrics evolve over time). Packet loss measurements require an ACK or NAK-based mechanism for feedback, while delay measurements require taking Round-Trip Time (RTT) samples; queue length is the only measurement that does not require feedback. All of these techniques are in principle suitable for disaster recovery applications.

Rather than handling congestion and error control on an end-to-end basis, an alternative is to use hierarchical aggregation, as in multicast transport protocols. For example, ESRT [33] chooses some nodes and assigns them the responsibility to manage those events, by collecting data from nearby sensors, merging them and delivering them to the sink. This approach comes with the open problem of suitable node selection, which requires some kind of network topology discovery. Furthermore, a hierarchical approach may cause delays in case of emergency, exactly when we need quick responses.

Priority-based Congestion Control Protocol (PCCP) [34] detects congestion using a metric derived from the ratio of packet inter-arrival time and packet service time. Packet inter-arrival time is the time interval between two consecutive packets, arriving either from a source or from transit traffic, while packet service time is the time interval between a packet's arrival at the MAC layer and the succesfull transmission of its last bit. The purpose of this metric is to improve QoS via better link utilization with respect to packet losses and delays. The Adaptive Rate Control (ARC) [35] protocol does not enforce a specific method of congestion detection and management; each node acts alone and tries to adapt its transmission rate. This approach does not burden the network with control overhead, but speed and efficiency cannot be guaranteed. Siphon [36] reacts to congestion by redirecting traffic to other nodes, which are upgraded to act like virtual sinks. The goal is to unload congested links and perform load balancing through multipath transmission. Of course, this approach assumes a multi-hop and dense network, where we can asily find alternative paths towards the sink.

Multipath transmission and load balancing are combined in [37], aiming to proactively handle congestion via a primary and a secondary path for load balancing. The mechanism monitors the usage level of sensor buffers for congestion detection. In the same direction, CODA [38] takes advantage of buffer monitoring to detect congestion. The sink examines the buffer usage of nearby buffers based on sensor reports and uses this information to detect congestion. Once congestion appears in the network, the sink starts sending signals and all other nodes push them via a backpressure mechanism towards the source. In a similar way, Fusion [35] relies on the queue lengths of intermediate nodes for congestion detection and uses hop-by-hop rate adjustment. Buffer usage is a good indicator of network capacity, but in case of increased packet collisions and after several unsuccessful Medium Access Control (MAC) retransmissions, packets are dropped. Consequently, the decrease in buffer occupancy due to these drops may falsely indicate an absence of congestion.

In [39] the authors combine fairness and congestion control. They assume that fairness is achieved via the allocation of equal rates, as each node sends the same number of packets. This solution is not suitable for WSNs with different types of sensors, where each type demands a different rate. The use of an equal rate allocation policy may not exploit all the available bandwidth, leaving some nodes with less bandwidth than what they asked for, while wasting bandwidth when nodes do not need it.

Rate-Controlled Reliable Transport (RCRT) [22] presents a centralized approach where the sink is responsible for network monitoring and congestion management, leaving sensors without any functionality related to reliable transfer (except recovery) as in [31]. RCRT employs a negative acknowledgment mechanism, where only losses are reported by the sink to the sensors. RCRT can employ three different rate allocation policies to assign the available transmission capacity to the sensors. In the *fair* policy, the same rate is allocated to all sensors, regardless of their requirements. In the *demand proportional* policy, each sensor is assigned the same fraction of its desired rate as all other sensors. In the *demand limited* policy the same rate is allocated to all sensors, except that sensors that asked for a lower rate are only assigned that rate, leaving more bandwidth for the others. RCRT relies on the time needed to recover from a packet loss to detect congestion and adapt the overall transmission rate to be assigned to the sensors. If this time is above a high watermark, the network is congested and the sink reduces the rates to all connected nodes. If this time is below a low watermark, then the network is underutilized and the sink increases the rates to all connected nodes. If this time is between the two watermarks, the network is considered stable and rates do not change. RCRT is a good candidate for disaster recovery applications as the sink is in control of the network, meaning that sensors are simple, all control loops are end-to-end, meaning there is no reliance on intermediate nodes for packet processing, and the protocol is relatively simple to operate, meaning that it is easy to re-establish connectivity when the sink moves.

2.3 BitTorrent

BitTorrent is a distributed application that allows a set of nodes, called *peers* to exchange files broken in fixed size *pieces*. While file exchange starts from a special set of peers which hold the entire file, the *seeds*, any peer can send and receive data from any other peer. This allows the file exchange to take advantage of far more bandwidth than that available to the seeds. As BitTorrent peers act independently, we can expect them to operate according to their own interest rather than the common good. However, BitTorrent is designed to induce behavior that maximizes the chances of each peer receiving the desired file, even in the face of node disconnections and failures, which means focusing on the download of those pieces that are harder to find. This behavior does not lend itself naturally to streaming applications, where the focus is on downloading pieces (mostly) in order, so as to allow playback to commence and continue as the file is downloaded. We provide below an outline of regular BitTorrent operation, as implemented in our own BitTorrent simulator [23].

2.3.1 The Tracker Protocol

Typically, the distribution of a new file[1] with BitTorrent starts by publishing a `.torrent` metafile; this metafile is distributed to peers using an out-of-band channel,

[1]Since BitTorrent organizes the set of files to be distributed as a linear sequence of bytes, akin to a single file, we use the terms file and files interchangeably.

usually by posting the metafile on a web page. Among other information, a `.torrent` metafile contains the tracker address, file size, piece size, and the hashes for the file pieces. Trackers are responsible for helping peers discover each other so as to form a swarm. In most cases, each `.torrent` metafile is served by a single tracker, but extensions to the protocol allow multiple trackers for one file or even no trackers at all [40]; the *trackerless* approach employs Distributed Hash Tables for decentralized peer discovery.

Clients communicate with the tracker via a simple text-based protocol, layered on top of HTTP/HTTPS, using the tracker's URL stored inside the metafile. During the download phase, each client communicates with the tracker and publishes its progress (in terms of total bytes downloaded/uploaded), as well as its *contact details* (e.g., IP address, TCP port, identification info. These parameters are passed from the client to the tracker using the standard HTTP GET method. Note that most of the information *announced* by the client is for statistical purposes; only the IP address and TCP port of a client are crucial for the tracker. After such message, called a *tracker request*, the tracker randomly selects a set of peers and returns their contact details in a *bencoded* dictionary [41]. Since each tracker request provides the contact details of a client to the tracker, the tracker can return such details back in its replies. In this manner, over time the peers discover increasing subsets of the swarm.

2.3.2 The Peer-wire Protocol

The peer-wire protocol provides the core BitTorrent functionality: interaction with remote peers. In the following we first present an overview of the peer-wire protocol and then proceed with the details of its operation, focusing on the most important features, which are available in our simulator implementation.

Protocol overview

After contacting the tracker, a client attempts to establish TCP connections with the peers listed in the tracker response. Upon connection establishment, the two peers exchange HANDSHAKE messages in order to verify their peer identities and ensure that they are interested in the same torrent metafile. This handshake is then followed by an exchange of BITFIELD messages that contain the *bitfield* of each client, which is a bitmap denoting the availability of each piece at the client. Based on that information a client can determine whether it is interested in one or more pieces of the remote peer. Note that this exchange is optional when the client has no pieces, since it would result in the exchange of useless information, and is therefore avoided in our implementation.

By following the above procedure over multiple peer connections, a client collects information regarding the availability of the pieces that it is still missing in the subset of the swarm that has explored using that information. Based on this information, it then decides which pieces to request from each peer. In general, if a peer does not hold any pieces that the client does not already hold, a NOT INTERESTED message is sent to that peer to indicate the lack of interest for its data. At the beginning of a connection, peers are assumed not to be interested in each other's pieces by default.

Although at this stage a client knows the peers it is interested in, it cannot make any requests yet as data cannot be exchanged until the remote peer actively permits this by sending an UNCHOKE message. This means that each client is by default blocked, or, in BitTorrent parlance, *choked* by the corresponding remote peer. The decision to choke or unchoke a client is made based on several criteria embodied in the *choking algorithm* [41] of the protocol:

Reciprocation: Peers unchoke the clients providing the best upload rates.

TCP performance: TCP behaves better when the number of simultaneous uploads is capped.

Fibrillation avoidance: Frequent choking/unchoking causes data transfer interruptions that deteriorate protocol performance.

Optimistic unchoking: New peers are occasionally unchoked so as to discover potentially better connections. Moreover, peers are thus given the chance to acquire their first pieces.

When a client is unchoked by a peer, it starts sending REQUEST messages, each asking for a specific *block* of the selected piece. The peer sends back the requested data using PIECE messages. Upon completing the downloading of a piece, a client informs via HAVE messages the peers that it has established connections to. These peers update the bitfield for that client and may then, potentially, express their interest for that piece with an INTERESTED message.

Connections

A client learns about other peers by employing the Tracker protocol and parsing the peer list returned by the tracker. The client then joins the swarm by establishing connections with some peers. However, as noted in [41], each connection incurs an increase in signaling traffic, especially for bitfield maintenance via the exchange of

HAVE messages. Thus, in most implementations, including our simulator, there are configurable lower and upper bounds for the number of established connections.

Piece downloading strategy

The piece downloading strategy refers to the policy followed in the selection of the pieces that will be requested from a peer. It is an important aspect of BitTorrent as it heavily affects the diversity of the pieces available at each peer. A low degree of diversity would result in low interest for a peer's pieces, thus causing degraded application performance. We have implemented the two most prevalent piece downloading strategies, *Rarest First* and *Random First*. Based on the information gathered during the BITFIELD – HAVE message exchange, the *Rarest First* strategy selects those pieces that appear less frequently in a client's set of connected peers. This selection is randomized among several of the less common pieces, according to a configuration parameter, in order to avoid multiple peers converging on the same piece. This way, peers download pieces that most other peers probably want, therefore facilitating data exchange. However, rare pieces are present only in a few peers, and it is possible that downloading them may be interrupted due to a choking decision. Clients with no pieces in their possession would therefore have to wait for an optimistic unchoking event from a peer holding the same rare piece in order to continue downloading. The *Random First* strategy avoids this problem by selecting a random piece which is more likely to be available from multiple peers, so that a choking decision would not have such an adverse effect.

Queueing

As mentioned above, Request messages refer to specific blocks of a piece. This facilitates fine-grained data exchange by enabling the queuing of data requests. As common piece sizes vary from 256 KB to 1 MB [41] or even larger, per piece requests would result in a multitude of redundant packet retransmissions in the event of a choking decision during piece transfer. A window-based queuing mechanism is employed for these requests, otherwise propagation delays would dominate the total download time.

Since the exact nature of the queueing policy is under dispute, we implemented a generic queueing mechanism in which the user can specify the exact size of the queue. In this mechanism a client may send to a peer up a configurable number of Request messages for blocks. Once a Piece message has been received, the client may send the next Request message. Once a piece has been requested in its entirety, if the request queue is not full, the client chooses another desired piece from that peer's bitfield according to the piece selection strategy (see Section 2.3.2) and starts sending Request messages for its blocks.

Choking algorithm

For the choking algorithm we followed the guidelines presented above,long with the ability to tune the choking algorithm as desired. The user is able to select appropriate values for the time between (optimistic) choking decisions, using configuration parameters for the intervals between decisions and between optimistic unchoking decisions. In order to facilitate the deployment of content providers with advanced seeding capabilities (see [23]), these parameters can be separately configured for such nodes.

Super Seeding

The *super seed* feature is especially useful for content distribution as it helps the initial seeder of a file avoid excessive bandwidth consumption while fostering data exchange between participating peers. A super seeder does not inform its peers that it has all pieces available, masquerading as an ordinary client. Initially it pretends to possess no pieces and only later informs it about the availability of an individual piece with a Have message, as if it had just completed downloading it. The seeder either selects a piece it has never uploaded before or, if all pieces have already been uploaded at least once, a piece that has been uploaded only a few times. After the piece has been downloaded by the peer, the seeder will not inform it of another one until it sees this piece marked as available in the bitfield of another peer, implying that the first peer has in turn uploaded this piece.

Our simulator implements this feature at all clients, but only enables it at the initial seeder via a configuration parameter, since super seeding is not recommended for ordinary peers [41]. These peers provide their pieces as requested by their clients, acting as regular seeders after downloading all pieces. The duration of this seeding phase can be set via another configuration parameter.

Endgame mode

The *endgame* mode addresses the problem of slow transfers for the last data blocks of an exchange, since at that stage most pieces have been downloaded, therefore the degree of parallelization is low. In this mode the client sends Request messages for each missing block to all peers that are not choking it, as opposed to a single peer. While this is not clarified in the specification [41], our implementation does not send these messages to all peers in its current peer set since a peer choking the client will simply discard the request. Another unclarified aspect of the endgame mode regards the entry condition. In our implementation, the client enters this mode when the number of missing blocks equals the number of requested blocks, meaning that all missing blocks have been requested.

Chapter 3

Reliable Multicast Transport for Information-Centric Networks

3.1 Introduction

In the past few years, considerable research has been dedicated to a set of novel future Internet architectures which drop the current design in order to improve the efficiency of large-scale content delivery. Many of these efforts are based on native multicast support, which has long been considered the key for efficient content distribution, but was never widely adopted on the Internet, due to the limitations of the current Internet architecture and the lack of adequate business models for its adoption [42]. The design and implementation of the omnipresent Internet architecture leans on the traditional telephone network idea where there are only two parties wishing to communicate. Extending this model to multiparty communication requires considerable engineering effort and costs for the network operators. Unfortunately, there is no clear path to adoption aligned with the business models of network operators, hence IP multicast is prevalent only inside private networks for specific applications, e.g. IPTV over ADSL networks.

Recent research efforts have tried to move the center of attention from *where* the desired information is located to the *information* itself, since in most applications users are only interested in getting the desired content, not in the location from where it can be retrieved. This direction has manifested itself in many developments, including *Peer-to-Peer* (P2P) file sharing, *Content Delivery Networks* (CDNs) and cloud computing services, which generally operate as an overlay to the existing Internet. A more radical approach is to introduce a clean-slate *Information-Centric Networking* (ICN) architecture, which focuses on the content rather than the endpoints hosting and consuming it. By designing a suite of network protocols around information itself, these proposals aim to better satisfy the requirements of current and future content distribution applications [7, 8, 43, 26].

A commonly claimed advantage of the ICN paradigm is that since content is explicitly requested, the network can easily aggregate requests for the same content and then serve them via multicast, thus boosting the efficiency of content delivery. While most ICN architectures offer native support for multicast at the network level, they have not yet addressed the issue of designing efficient reliable transport protocols for multicast, even though considerable work has been performed in this area for IP multicast. In general, reliable multicast can be achieved in two ways: sender-driven with acknowledgements as feedback, and receiver-driven with negative acknowledgements. In sender-driven protocols the sender will eventually become a bottleneck due to acknowledgement implosion as the number of receivers grows, thus affecting the sender's throughput [44]. Therefore, most reliable multicast protocols are receiver-driven, an approach that we also adopt.

In this chapter, we present Reliable Multicast Transport for PSI (RMTPSI) [45, 20], a transport layer protocol for the Publish Subscriber Internet architecture, which was designed and implemented during the FP7 EU project PURSUIT [16]. We compare RMTPSI with Pragmatic General Multicast (PGM) [19], a reliable multicast transport protocol designed for IP Multicast, and evaluate their performance, when both are implemented over the PSI architecture. Our results show that RMTPSI is more efficient than PGM, while requiring the same time to complete a reliable transfer as PGM; specifically, RMTPSI requires 2.9% to 10.2% fewer downstream transmissions and 3.5% to 12.1% fewer total transmissions than PGM.

It should be noted that the target application for RMTPSI is fully reliable multicast delivery, for example, the distribution of firmware, operating system patches or antivirus updates over the network. In these applications each recipient must receive all data correctly, regardless of how long this may take, otherwise the received data are useless. The reason for focusing on such applications is that they offer a natural synchronization between senders and receivers: as updates become available, it makes sense to transmit them immediately to all available recipients. In contrast, in applications such as media distribution, either it is hard to ensure receiver synchronization (e.g. in video on demand) or mostly reliable delivery is sufficient (e.g. in live video streaming).

The structure of this chapter is as follows. In Section 3.2 we present the design details of RMTPSI. Section 3.3 first provides a description of the experimentation environment and then presents the performance results obtained. We conclude in Section 3.4.

3.2 Multicast Error Control for PSI

3.2.1 Overview

RMTPSI is most similar to PGM, in that it uses selected routers *on the multicast tree*, as opposed to the designated receivers in RMTP, to aggregate feedback and control the propagation of retransmissions. In order to minimize the nodes that need to manage state for multicast, we reuse the multicast Relay Points (RPs) which are required by the forwarding architecture of PSI when trees grow large. As a result, each multicast subtree created by the PSI Topology Manager (TM) independently manages the error recovery process, thus avoiding feedback implosion at the original receiver.

RMTPSI operates in three phases: communication setup, where the multicast subtrees are created and the appropriate Bloom filters are distributed, initial content distribution, where the publisher sends the entire content and collects aggregate feedback, and recovery, which may involve one or more cycles of initiating retransmissions and collecting new feedback, until all the subscribers have completed the download successfully. We elaborate on these phases in the following subsections.

3.2.2 Communication setup

In the communication setup phase, when the PSI Rendezvous Point (RVP) matches a publisher that has made available data under a SId/RId pair with a set of subscribers for these data, it notifies the TM to create a multicast delivery tree. The TM first calculates an overall tree and then splits it into subtrees, creates the appropriate Bloom filters for each subtree and distributes them to the RPs. As already mentioned, Bloom filters operate only in one direction, therefore we need separate Bloom filters in order to return feedback from the subscribers towards the publisher. We therefore modified the TM to calculate not only downstream Bloom filters from the root to the leaves of each multicast subtree, but also upstream filters. This is achieved by simply ORing the link labels for the *reverse* direction of the tree; these upstream filters represent a tree from all leaves towards the root, therefore they can be used to send upstream messages from *any* leaf to the root. The advantage of this approach is that only a single Bloom filter needs to be created and communicated to all the receivers in the subtree. The creation of the subtrees is performed via a breadth-first traversal of the overall tree; when the Bloom filter for a subtree contains enough 1's, the subtree stops there and an RP is created. Since the same number of links is entered in both the forward and reverse Bloom filters, the number of 1's and the false positive probability is roughly the same in both directions. Algorithm 1 elaborates upon the recursive procedure of Bloom filter creation.

Algorithm 1 RMTPSI Bloom Filter Calculation.

```
 1: procedure CALCULATE_BLOOMFILTERS(topology, publisher, subscribers)
 2:     $roots.push(publisher)$
 3:     while $Subscriber\ without\ BF\ exists$ do
 4:         $current\_root := roots.pop()$
 5:         $new\_roots := run\_BFS\_until\_BF\_full(topology, current\_root, NULL)$
 6:         $roots.push(new\_roots)$
 7: procedure RUN_BFS_UNTIL_BF_FULL(topology, root, prev_link)
 8:     $BF := null$
 9:     $BFS\_graph = run\_BFS(topology, root)$
10:     while $Ones\ in\ BF\ and\ RevBF\ under\ limit$ do
11:         $new\_link := BFS\_graph.get\_next\_link()$
12:         $add\_link\_to\_BF(new\_link)$
13:         $add\_link\_to\_RevBF(new\_link)$
```

The set of forward and reverse Bloom filters for the publisher and each RP are then sent to the RVP, which in turn sends to the publisher and *each* RP both a downstream Bloom filter, used for data forwarding, and an upstream Bloom filter, used for feedback *within its own subtree.* The publisher then sends a setup message to initiate the content distribution; this message includes the upstream Bloom filter that should be used for feedback within its subtree and a generation counter set to zero. Upon reception of this message, each RP stores the upstream Bloom filter used to reach its parent, switches the Bloom filter in the packet and forwards the message, encapsulating inside it the upstream Bloom filter that should be used by its own children for feedback. At the end of this process, each RP knows the Bloom filter needed to reach its children (from the original message of the RVP) and each RP and receiver knows the Bloom filter needed to reach its parent (from the setup message received from the parent) for a specific multicast group, denoted by a SId/RId pair.

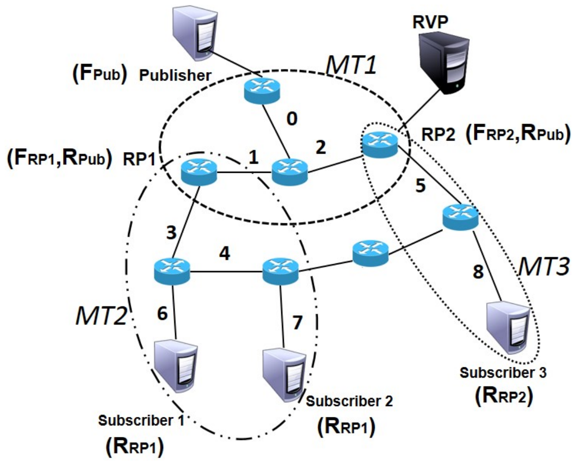


Figure 3.1: An example of Bloom filter distribution.

In Figure 3.1 the RVP first sends a pair of filters (F_{Pub}, R_{Pub}) to the publisher, a pair (F_{RP1}, R_{RP1}) to RP1 and another pair (F_{RP2}, R_{RP2}) to RP2. The setup message from the publisher is forwarded using F_{Pub} and encapsulating R_{Pub}. When it reaches

RP1 through the path {0,1} and RP2 through the path {0,2}, each RP stores R_{Pub} as its feedback filter to the publisher. Then the message is forwarded from RP1 using F_{RP1} and encapsulating R_{RP1} and from RP2 using F_{RP2} and encapsulating R_{RP2}. Each receiver, upon reception of this message, stores the encapsulated filter for feedback. In this way the multicast tree from publisher to subscribers is divided into three smaller ones (MT1-MT3), and each node knows the downstream and upstream Bloom filters needed for protocol operation. Figure 3.1 shows the final state of Bloom filter distribution and the list of Bloom filters s that are stored in each node.

3.2.3 Initial content distribution

After setup completes, the publisher starts sending the data packets, inserting in each header the generation counter from the setup message. At each RP, the SId/RId in the packet is looked up and the Bloom filter is replaced with the one needed to reach the next RPs and/or subscribers. For example, in Figure 3.1 RP1 would replace F_{Pub} with F_{RP1}.

When a subscriber detects that a packet has been lost (based on sequence numbers) or corrupted (based on checksums), it uses the upstream Bloom filter to return a Negative Acknowledgment (NAK) message to its next upstream RP, also inserting the generation counter in the header. The RP holds the NAK for a specified interval waiting for more NAKs to come, in case more subscribers have missed the same or other packets. If additional NAKs, either for the same or for different packets, arrive at the RP during the waiting period, they are combined into a single NAK which is forwarded upstream, by using the corresponding upstream Bloom filter.

Each RP tracks the NAKs arriving from its subtree, in order to later forward recovery data only where needed, as well as to avoid relaying NAKs for packets that have already been NAKed. For example, say that in Figure 3.1 RP1 receives a NAK from one of the receivers in its subtree. After the waiting interval, this NAK will be forwarded to the publisher and RP1 will note that it has reported that packet as missing. When the publisher later retransmits the missing packet using its forward Bloom filter, the retransmission will reach both RP1 and RP2, but only RP1 will forward it in its subtree, while RP2 will drop it.

3.2.4 Recovery

When the publisher finishes the initial content distribution, it waits for a specified period of time in order to allow nodes that have received the entire transmission to leave the multicast group. When those nodes notify the RVP that they do no longer wish to subscribe to the multicast group, the RVP will notify the TM to update the distribution tree accordingly, by issuing new Bloom filter pairs wherever needed. The publisher will then send a new setup message with the generation counter increased by one, so as to distinguish packets from different recovery cycles. This message lets subscribers know that the current round has finished, allowing them to detect any packets that were lost at the end of the current round; these packets will be requested as part of the *next* round. In addition, the setup message updates the upstream Bloom filters throughout the multicast tree, as in the original transmission round.

Then, a retransmission round begins, with the publisher transmitting all packets for which NAKs have been received; each RP only forwards in its subtree the packets for which it has noted that NAKs were received and forwarded. Again, receivers send NAKs for missing packets, which are aggregated as previously explained.

Both data and NAK packets use the generation counter from the most recent setup message. This procedure (setup, transmission, feedback) is repeated at the end of every retransmission round, until the download completes at all receivers. Since subscribers leave the multicast group after correctly receiving all data, the tree will eventually be torn down, thus concluding the transfer.

3.2.5 Comparison with PGM

As mentioned above, RMTPSI is most similar to PGM, in that feedback is aggregated in tree nodes, effectively splitting the multicast distribution tree into subtrees. While in PGM a single IP multicast address is used for the entire tree and the subtrees are used only for feedback aggregation purposes, in our approach we reuse the RPs mandated by the forwarding architecture for this purpose. Also, while in PGM NAKs are sent via unicast to the upstream aggregation point, we instead use reverse Bloom filters to achieve the same result, since our forwarding architecture is based on source routing. Essentially, these modifications are required to map from IP multicast to the PSI forwarding architecture.

A more significant difference is the way feedback is aggregated. In PGM, the RP attempts to suppress further NAKs for a packet by multicasting a Negative Acknowledgment Confirmation (NCF) message, while in RMTPSI each receiver sends NAKs for all missing packets to its upstream RP; there is no NAK suppression within a subtree. If the loss probability is similar for all links in a subtree, it is unlikely that many receivers will lose the same packet, therefore it is wasteful to multicast an NCF to the entire subtree for each loss; even if another NAK for the same packet is sent, the multicast NCF may still be more costly, as it crosses all links in the subtree.

Another difference is that PGM sends new data interspersed with retransmissions, eventually giving up on lost packets, while RMTPSI retransmits packets in rounds until all have been received, expecting that in each round some group members will depart, hence reducing the number of links that each retransmitted packet will cross at every round. This is due to the fact that RMTPSI is designed for fully reliable transmission, while PGM is geared towards mostly reliable transfers.

3.3 Experimentation and Evaluation

3.3.1 Simulator Setup

In order to evaluate the performance of our protocol against PGM, we used extensive simulations over NS-3 [46], where the entire PSI architecture was implemented, including Bloom filter based forwarding and wide-area multicasting based on RPs. We simulated a scenario where a single publisher distributes the desired content (e.g. an update for an operating system or an antivirus update) to a large set of subscribers. The content size is 20 MB, composed of 20.000 data packets with a payload of 1 KB each. We used randomly generated scale-free network topologies of 200 and 500 routers (generated with the Barabasi-Albert algorithm [47]) with 50 and 100 subscribers attached to randomly chosen routers, leading to an average of 15.6 and 28.6 RPs, respectively; smaller topologies can be handled without RPs. Each scenario was executed 7 times, with different random positions of attachment to the network for the publisher, resulting in a different tree being generated and different RPs being chosen each time.

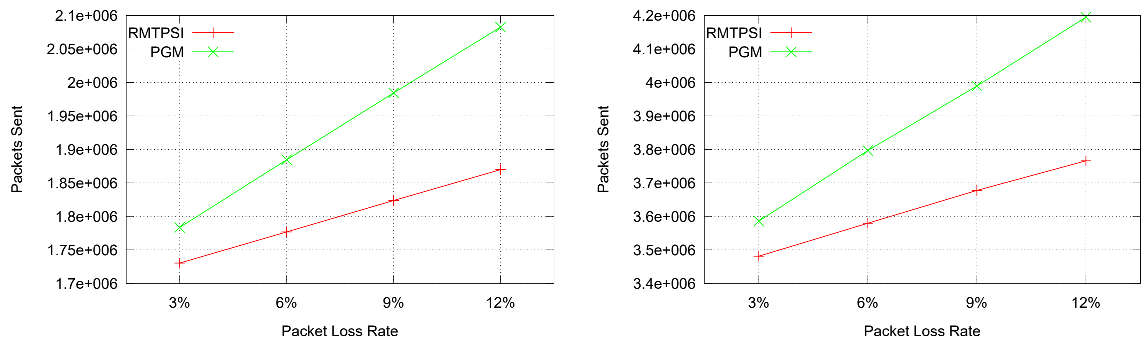
We assumed that all losses were random, i.e. packets were independently lost with the same probability in each link; we did not model the correlated losses usually associated with congestion. The values for the link loss probability were chosen experimentally, so that in each topology the average loss rate reported to the publisher would be 3, 6, 9 or 12%. Packet losses affected all packets, with the exception of setup packets.

Concerning the PGM protocol, all its features described above have been implemented, using the RPs as NEs and the same multicast trees for both PGM and RMTPSI. When a PGM subscriber notices a loss, it picks a random interval from 0 to 1000 ms before sending a NAK, while RMTPSI subscribers send their NAKs immediately. When a NAK is received, the RP in both PGM and RMTPSI waits for 70 ms before passing it upstream, aggregating all NAKs received in the meantime into a single NAK. In order to make PGM fully reliable, each subscriber starts a 500 ms timer upon sending a NAK, waiting for an NCF. If the timer expires before an NCF arrives, the NAK is retransmitted and the timer is restarted. Upon arrival of an NCF, the subscriber starts a new 500 ms timer, waiting for the missing packet. If the timer expires before the packet arrives, the NAK is retransmitted and the timer is restarted, until the packet is eventually received.

3.3.2 Experimental Results

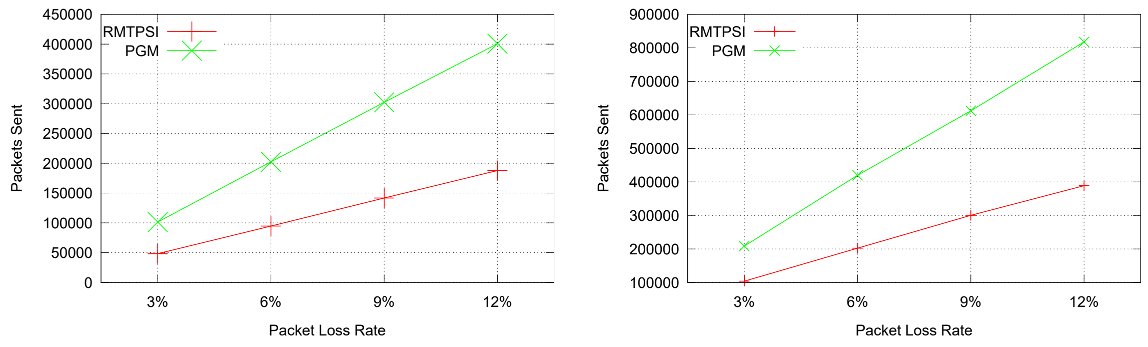
Although the protocols were also tested without losses, we do not show here any results from these runs, as in that case the only overhead to the network comes from the communication setup and BF distribution phase, where the behavior of both protocols is identical, as we use the same communication setup in both cases.

Our first metric deals with the total number of packets transmitted in the publisher to subscriber direction, for both the initial data distribution and the recovery phases; we count all single hop packet transmissions, that is, a multicast transmission from the publisher or an RP that crosses a tree with n links counts as n transmissions. Figure 3.2 presents this metric for RMTPSI and PGM, for the 200 and 500 router topologies. The size of the network changes the number of total packet transmissions required, as there are both more receivers and more intermediate routers in the larger topology, but the overall trends are the same: RMTPSI requires 2.9% to 10.2% fewer downstream transmissions than PGM, with higher benefits as the loss rate grows.



(a) 200 routers (b) 500 routers

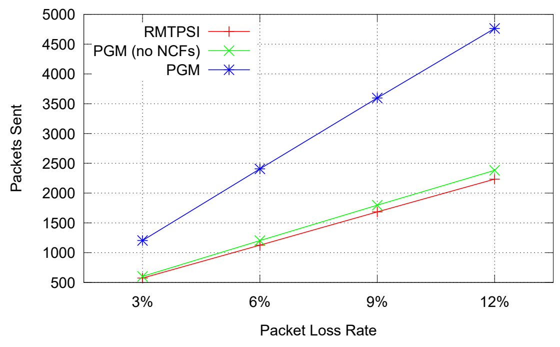
Figure 3.2: Number of packets sent downstream by all network elements.



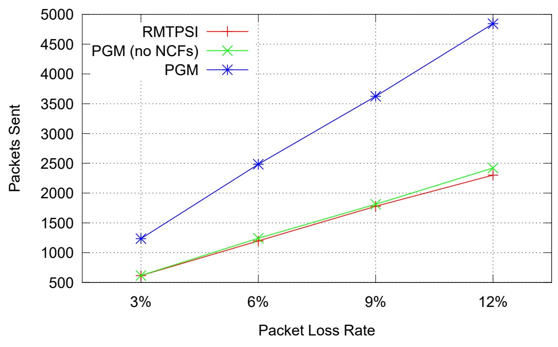
(a) 200 routers (b) 500 routers

Figure 3.3: Number of recovery packets sent downstream by all network elements.

In order to isolate the performance of the recovery mechanism, in Figure 3.3 we show the total number of packets transmitted in the publisher to subscriber direction only for error recovery purposes, that is, the retransmissions and, in the case of PGM, the NCFs. Again the trends are very similar for both topologies: RMTPSI requires 50.1% to 53.2% fewer recovery packets in the downstream direction than PGM, a nearly constant reduction across the range of loss rates tested.



(a) 200 routers

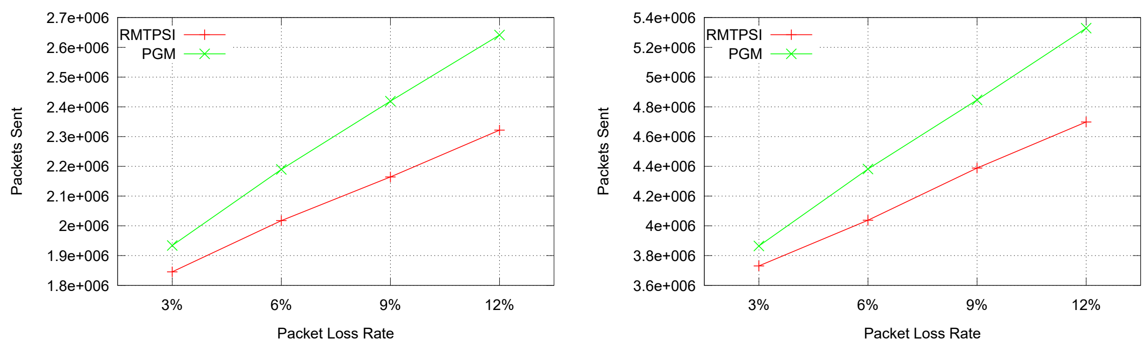


(b) 500 routers

Figure 3.4: Number of recovery packets sent downstream by the publisher.

The reason for this big gap can be understood by Figure 3.4, which shows the number of packets sent directly by the publisher for recovery purposes only. While PGM retransmits roughly the same number of data packets as RMTPSI, the NCFs roughly double the downstream recovery traffic. In PGM, the first NAK for each packet in a subtree triggers an NCF; in most cases, only one leaf of the subtree will have lost that packet, therefore the number of NCFs is roughly equal to the number of NAKs and the number of retransmissions, thus the number of packets sent for recovery purposes doubles in PGM. The same holds for all routers, thus explaining the large difference in favor of RMTPSI in Figure 3.2.

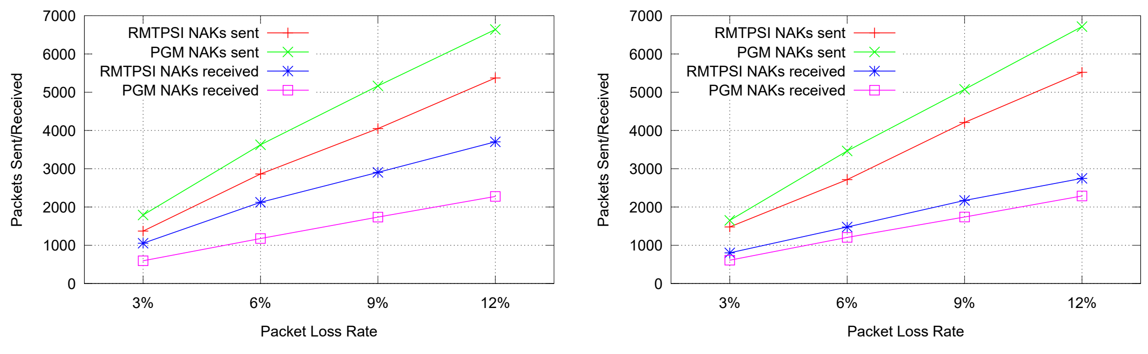
Since error recovery also requires traffic in the upstream direction, in Figure 3.5, we show the total number of packets transmitted in both the downstream and upstream



(a) 200 routers (b) 500 routers

Figure 3.5: Total number of packets transmitted by all network elements.

directions; again, we count all single hop packet transmissions, as in Figure 3.2. Upstream traffic is 6.7% to 24.8% of the downstream traffic for RMTPSI (7.8% to 27.0% for PGM), representing a considerable overhead, especially as loss rates grow. In total, RMTPSI requires 3.5% to 12.1% fewer packet transmissions than PGM in both directions.



(a) 200 routers (b) 500 routers

Figure 3.6: Total NAKs sent by subscribers and received by publisher.

The increased savings with RMTPSI when both directions are taken into account can be explained by Figure 3.6, which shows how NAK aggregation performs in PGM and RMTPSI; specifically, the figure shows how many NAKs are transmitted by all subscribers and how many NAKs are finally received by the publisher. The reason for the larger number of NAKs generated by the subscribers in PGM is the NAK retransmissions due to timer expirations when either the NCF or the missing packet is not received on time. Since in PGM NCF and retransmitted data packets compete with regular data packet transmissions, these delays can be quite large. To reduce NAK retransmissions as much as possible, we increased the timeout delay up to the point where the download duration started to increase. On the other hand, in RMTPSI there is no need for timeouts: if a missing packet has not been received by the end of the round in which the corresponding NAK was sent, then a new NAK is sent for that packet, thus NAKs are rarely retransmitted. In contrast, PGM is more effective than RMTPSI in aggregating these NAKs, since fewer NAKs reach the publisher, although this does not make up for the increased overhead due to the original NAKs. We also notice that aggregation is more efficient in the larger network topology. This is because a larger network leads to larger trees and the insertion of more RPs. By increasing the points where NAKs are aggregated, both mechanisms work more efficiently.

Finally, we measured the time needed for both mechanisms to complete the transmission to all subscribers, including recovery. While we expected that our approach would need more time to complete than PGM due to our recovery round structure, we observed that both approaches were ending almost simultaneously in all cases. This happens because the publisher in PGM has to interrupt the data transmission in order to send NCFs and retransmissions in parallel with new data; in addition, as noted above, in PGM many NAKs are retransmitted due to timeouts, thus further reducing the useful throughput of PGM as packets are needlessly retransmitted. RMTPSI in contrast always sends at full speed, since NAKs are gathered during one round and their retransmissions are taking place in the next round; NAKs are retransmitted only after the end of a round. In all our tests, full recovery in RMTPSI required two rounds of retransmissions, with three rounds needed only in a few of the repetitions at the highest loss rate.

3.4 Conclusion

In this chapter we presented an approach for multicast error control for the reliable, on-demand, delivery of information over a network supporting native multicast, using relay points to extend the reach of the source-routing mechanism used. Our RMTPSI scheme is based on feedback aggregation towards the sender via the relay points and multicast retransmissions of lost data. We explored the performance of RMTPSI through simulations using detailed message exchanges, focusing on its feedback aggregation features. Our results indicate that the aggregation mechanism effectively prevents feedback implosion, especially in larger graphs.

RMTPSI is broadly similar to PGM, originally proposed for IP multicast, in that both protocols use hierarchical aggregation to handle NAKs. Rather than selecting arbitrary routers for feedback aggregation as in PGM, RMTPSI uses the forwarding relays required to scale Bloom filters to large multicast groups for this role; the same adaptation was made to PGM for testing in PSI. One difference is that RMTPSI targets fully reliable distribution, thus operating in a series of transmission and retransmission phases, unlike PGM which targets mostly reliable distribution, thus mixing new data and retransmissions. Another difference is that RMTPSI does not rely on feedback suppression via multicasting NAK confirmations, as PGM does, thus making the protocol simpler: the receivers do not need timeouts to detect lost

NCFs, they just send their NAKs in each round. Most of the extensions proposed to PGM however [19] are compatible with our work.

Our experimentation results show that RMTPSI is more efficient than PGM, while requiring the same time to complete a reliable transfer as PGM. To be more specific, RMTPSI requires 2.9% to 10.2% fewer downstream transmissions and 3.5% to 12.1% fewer total transmissions than PGM. In terms of recovery related transmissions, RMTPSI requires 50.1% to 53.2% fewer recovery packets in the downstream direction than PGM as it does not depend on NAK confirmations as PGM. Although RMTPSI is not as capable as PGM in suppressing NAKs, it generates far fewer NAKs, leading to decreased control overhead compared to PGM in both directions. Finally, the use of recovery rounds makes the protocol less complex and multicast easier to exploit, without penalizing performance, as both RMTPSI and PGM take the same time to complete a similar transmission.

Chapter 4

Sink-Driven Reliable Transport for Wireless Sensor Networks

4.1 Introduction

A Wireless Sensor Network (WSN) may serve many applications and situations, from simple data collection and reporting, e.g. smart home monitoring or securing a controlled area, all the way to assisting human or robotic rescuers in disaster recovery situations. Disaster recovery is one of the most demanding use cases, since the sensor network may be partially connected and the need for quick and reliable data transmission is paramount. The rescuer cannot count on sensor redundancy for gathering critical information, hence the sensor network must rely on a transport layer that can transfer data quickly without losing any packets due to congestion around the sink, where data from many sources naturally converge. As the network environment in WSNs in general, and in disaster recovery situations in particular, is very unpredictable, transport protocols should be quick to adapt to changes in prevailing conditions, including congestion levels and network topology.

Initially motivated by the DIstributed Sensor systems For Emergency Response (DISFER) project [48], we designed the Reliable Transport protocol for SEnsor Networks with MObile Sinks (RT-SENMOS) [49, 50], a purely sink-controlled protocol, in the sense that the sink allocates transmission rates to all reachable sensors and manages the error recovery process depending on application objectives. A crucial aspect of our approach is that sensor handling depends on the application at hand, since sensors only need to implement a generic and very simple behavior, with the sink implementing the complex behavior required by that application. This allows the same sensors to be used for different application scenarios, by simply modifying the behavior of the sink. Furthermore, RT-SENMOS uses UDP/IP for packet transmission, therefore it can be fully embedded into the application, operating without kernel/superuser access.

In this chapter we present the detailed design of the latest version of our scheme, as well as a detailed performance evaluation of a real RT-SENMOS implementation over an actual network. We compare our mechanism against the Rate-Controlled Reliable Transport (RCRT) [22] protocol, another sink-controlled scheme for sensor networks, via experimentation over real implementations and emulation of real networks. We apply the general design of RT-SENMOS to a disaster recovery scenario in a public building, in which different types of sensors attempt to communicate with a mobile rescuer that moves around the disaster area.

The structure of this chapter is as follows. In Section 4.2 we present our assumptions, motivate our design choices and show how RCRT compares to our work. In Section 4.3 we describe the protocol's operation. Section 4.4 contains our performance evaluation of RT-SENMOS against RCRT. We conclude and discuss our plans for future work in Section 4.5.

4.2 Rationale and Related Work

The communication environment assumed by RT-SENMOS is a multi-hop network, where neighboring nodes are connected via a broadcast wireless technology, such as WiFi, Bluetooth or ZigBee. Some of the nodes in the network (possibly all of them) host sensors which attempt to transmit their data to a destination node, the sink. The sink may be mobile, for example, a human or robot equipped with a computer. We assume that all nodes know the network address of the sink, that is, they are pre-programmed with a specific sink address which they periodically try to connect to. Sensors are relatively dumb, in the sense that they are not aware of a specific application scenario, they just try to send their data to the sink. On the other hand, the sink is customized for a specific application, employing the RT-SENMOS mechanisms to control the sensors as needed. Different applications can be used at different times over the same network, by simply having the appropriate sink take over the pre-programmed sink address for that network.

Due to mobility and sensor outages, the sink may only be able to communicate with a subset of the sensors at any given time, therefore it must be able to adapt to rapid changes in the network environment. We assume that a dynamic routing protocol is used, ensuring connectivity between the sink and any reachable sensors [51]. Connectivity may be lost at any time though, due to node failures or sink mobility.

Finally, we assume that the network is deployed over an area where a disaster has taken place. As a result, we do not expect to see a complicated network topology, due to the possible loss of many sensors; instead, we expect that sensors will be only a few (probably only one) hop away from the sink, with the sink having to move across the area to communicate with additional sensors. This precludes the use of backpressure mechanisms for congestion management, as there are not many hops to control and the movement of the sink means that the situation around it changes constantly. We instead rely on end-to-end congestion and error control, as it only requires co-operation between the sink and each individual sensor.

To allow RT-SENMOS to serve diverse applications, we assume multiple classes of sensors, which can be treated differently. In general, the sink first allocates bandwidth to each class and then it assigns a portion of that bandwidth to each individual sensor. Different algorithms can be used for each of these allocation schemes, depending on the application, although we generally expect congestion to be concentrated around the sink, as all transmissions converge there. Since RT-SENMOS is purely sink-controlled, the sensors are unaware of these policies. In our test scenarios, sensors are divided into two classes: continuous and event sensors. A *continuous sensor* collects point data, e.g. a temperature value or a camera snapshot, and sends them periodically to the sink. An *event sensor* is triggered by an event, e.g. motion detection, to send some data, e.g. a 30-sec live video. We assume that event sensor data are delay sensitive, since they are correlated to a specific event in time. In contrast, we assume that continuous sensor data are not that delay sensitive as they are periodically refreshed.

Our protocol executes at the application-layer over UDP/IP, allowing it to be implemented in any device with IP connectivity and a UDP socket interface, without any kernel modifications or privileges. Our implementation is written in Java, therefore it can be used on all kinds of devices, including Android smartphones and tablets. While most transport protocols are implemented as libraries, with application calling their primitives to exchange data, RT-SENMOS was designed to be embedded within an application, thus allowing the application to directly control all protocol parameters depending on its needs. In addition to the rate allocation policies between and within each class, the application can even control the level of reliability desired, by explicitly asking for only those retransmissions it deems sufficient for its goals.

In [50] we compared a number of transport protocols for sensor networks to RT-SENMOS [22, 31, 32, 51]. The closest to our approach is RCRT [22], as congestion and error control are handled by the sink, with sensors having a more passive role. RT-SENMOS adds many new features, focusing on the differentiated handling of different sensor types in terms of congestion and error control, so as to be able to serve a wider variety of applications.

4.3 Protocol Description

In this section we describe the behavior of the current version of RT-SENMOS[1]. In general, the protocol operates in three phases. First, the sensor makes periodic attempts to connect to the sink; when the sink responds, the sensor sends it class and its desired rate(s), and the sink responds with an initial rate allocation. Second, one or more data transmissions take place, with retransmissions taking place either alongside data transmission or in recovery rounds; the sink may update the rate allocation of a sensor at any time with a rate update message. Finally, either side can drop the

[1]Older versions of the protocol use a simpler congestion control scheme and only one error control method; see [49, 50] for details.

connection or detect (via timeouts) a connection failure. A more detailed description of the protocol is provided in the following subsections.

4.3.1 Connection establishment

The RT-SENMOS model defines two separate communication channels per sensor, one for control data and one for user data. An additional common control channel attached to a well-known UDP port is used for connection requests. This approach allows the sink to dedicate one thread to listening for connection requests and a set of threads for exchanging control plane messages with sensors, adding data channel threads on demand. As a result, there is no need for multiplexing control messages from multiple sensors over a single channel.

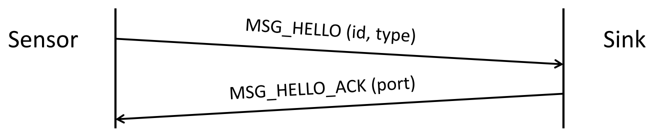


Figure 4.1: Connection establishment and sensor information exchange.

When the sink begins to operate, it listens to a well-known UDP port for connection requests from the sensors. The sensors wait for the sink to become reachable before trying to connect to it. As shown in Figure 4.1, each sensor may periodically send a probe message, `MSG_HELLO`, to the well-known IP address and UDP port of the sink until it receives a response. Alternatively, the routing protocol used may notify the sensor when the sink becomes reachable. The `MSG_HELLO` message includes the sensor identifier and its type, i.e. event or continuous sensor. The sink responds to the `MSG_HELLO` message with a `MSG_HELLO_ACK` message. The `MSG_HELLO_ACK` message indicates a UDP port dedicated to that sensor for its control messages. If the sink does not have a pre-configured sensor identifier, then it leaves the corresponding field empty in the `MSG_HELLO` message and the sink assigns it one in the `MSG_HELLO_ACK` message.

After receiving a response from the sink, the sensor prepares an `MSG_INFO` message which includes the delay between receiving the message from the sink and sending the response, the data rate requested by the sensor, whether the sensor is ready to send data at this point in time or not, the data packet size to be used and the total size of the data to send. When the sink receives the `MSG_INFO` message it calculates the time elapsed since sending the `MSG_HELLO_ACK`, subtracting the delay in the message to get the Round-Trip Time (RTT) to that sensor. At this point, the connection has been established and the sink is aware of the sensor's bandwidth requirements.

4.3.2 Data exchange and idle

If the sensor has indicated that it is not ready to send data in its `MSG_INFO` message, the sink moves to an idle state, waiting until the sensor sends a `MSG_CTRL_DATA` with no parameters through the control channel for that sensor, as shown in Figure 4.2. Then, the sink starts the data exchange by sending an `MSG_CTRL_START` message to the sensor, indicating the data rate to use and a UDP data port to use for the data transmission. If the sensor indicated in the `MSG_INFO` message that it was ready to send data, the `MSG_CTRL_START` message is sent immediately as a response. The initial rate allocated to the sink is set as explained in Section 4.3.5.

After the sink sends the `MSG_CTRL_START`, the actual data transfer begins, using

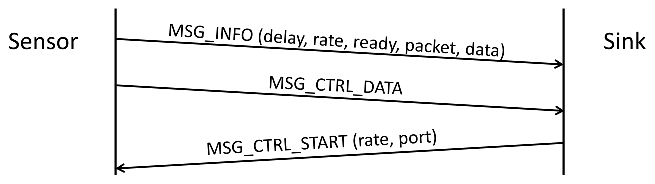

Sensor
Sink
MSG_INFO (delay, rate, ready, packet, data)
MSG_CTRL_DATA
MSG_CTRL_START (rate, port)

Figure 4.2: Start of data exchange stage.

the UDP data port assigned for the transfer; control messages, such as NAKs and rate updates, are exchanged out of band over the control channel, without being rate controlled. This allows control messages to be sent without waiting behind a, possibly long, queue of data messages. The sensor breaks down its transmission into packets of the size indicated in the `MSG_INFO` message, until the data indicated in the `MSG_INFO` message are exhausted. Data packets only have a single header field, a segment number used to sequentially number all data packets.

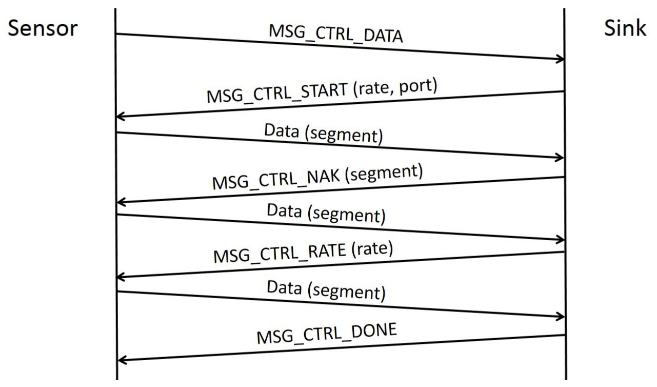


Figure 4.3: Data exchange stage.

When a missing packet is detected, the sink sends a `MSG_CTRL_NAK` to the sensor over the control channel, as shown in Figure 4.3. The response from the sensor depends on its type, as explained in Section 4.3.4. Once the data transmission, including any recovery messages, is complete, the sink sends a `MSG_CTRL_DONE` message over the control channel to indicate a successfully completed transfer. Both endpoints then move to an idle state, until the sensor generates a new `MSG_CTRL_DATA` message. If needed, the sink may send a `MSG_CTRL_RATE` message to the sensor indicating its new rate allocation, as explained in Section 4.3.5.

4.3.3 Connection control and release

Since the path between the sink and the sensor may become disconnected due to sink mobility, the connection may fail between data transfers, without either side noticing. For this reason, the sink periodically sends an `MSG_CTRL_ALIVE` message to the sensor, as shown in Figure 4.4, which is acknowledged by an `MSG_CTRL_ACK` message from the sensor that includes the delay incurred between receiving the `MSG_CTRL_ALIVE` and responding with the `MSG_CTRL_ACK`. In addition to confirming that the connection is still alive, this procedure allows the sink to periodically measure the RTT of the connection.

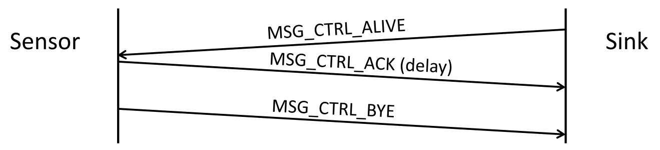


Figure 4.4: Data exchange and connection control stage.

If any side wishes to complete the connection, they can send a `MSG_CTRL_BYE` message, which does not need to be acknowledged, as after either side drops the connection, the other one will eventually timeout: the sink times out if no responses are received to its `MSG_CTRL_ALIVE` messages, while the sensor times out if no `MSG_CTRL_ALIVE` messages arrive.

4.3.4 Error recovery

The basic error recovery mechanism of RT-SENMOS relies on the sink issuing Negative Acknowledgements (NAKs) for data packets with the sensors retransmitting the lost data. However, while most protocols retransmit lost data immediately, RT-SENMOS can differentiate its behavior based on sensor class. As the basic retransmission policy must also be implemented at the sensors, we have chosen two schemes which cover a wide range of applications, while allowing the sink to modify its behavior depending on the application.

For the continuous sensor class, and in general for any transmissions that are not delay-sensitive, retransmissions are performed in *recovery rounds*, as follows. First, all the data packets are transmitted, with the sink issuing NAKs. Next, the data packets for which NAKs were received are retransmitted, then the ones for which NAKs were received again are retransmitted, and so on, as in RMTPSI [20]. The sink can modify this policy by stopping the recovery process whenever it deems appropriate, for instance, when enough packets have been received to reconstruct the content, by sending a `MSG_CTRL_DONE` message. Furthermore, if a source loses connectivity with the sink before the transmission completes, the sink can approximately reconstruct the content, as it will have received incomplete data from beginning to end, rather than complete data only from the beginning.

For the event sensor class, and in general for any transmissions that are delay-sensitive, recovery has to be completed in a limited time frame. For this reason, packets are immediately retransmitted by the sensors when a NAK is received, as in RCRT [22]. The sink can modify this policy by choosing which packets to NAK, depending on their value to the application. For example, in MPEG video, packets containing I-frames are more valuable than those containing P-frames. In our experimental evaluation we have implemented a policy where the sink measures the recovery time for the past 10 retransmissions (a configurable parameter) to determine whether a retransmitted packet will arrive soon enough to be played out, based on the current position of the video player. If not, the packet is not NAKed at all.

4.3.5 Bandwidth and Congestion management

The congestion management algorithm relies on two observations: first, congestion will most likely be concentrated around the sink, where all transmissions converge and, second, due to sink movement, the algorithm must converge quickly in order to be stable. For these reasons, while TCP and many other transport protocols use an Additive Increase - Multiplicative Decrease (AIMD) algorithm to probe the (unknown) available bandwidth on the end-to-end path, RT-SENMOS starts from the (known) available bandwidth around the sink, adding fixed and symmetric adaptation steps.

Starting from the bandwidth available around the sink, RT-SENMOS first allocates a fixed portion of the remaining bandwidth to each sensor class, depending on application requirements. Then, it performs rate allocation separately for each class, possibly using an entirely different algorithm per class. Note again that since RT-SENMOS is rate-controlled, the sensors are not aware of the actual scheme used, they just follow the sink's instructions. Rate allocation takes place whenever the sensor mix changes, that is, when a new sensor is connected or when an existing one is disconnected. Between these events, the rates are modulated by a congestion management scheme that relies on measuring the loss rate from each sensor in order to adapt its transmission rate.

In our original scheme, each sensor announced its desired rate during connection establishment [50]; we have extended this to allow either indicating a single target rate, or a set of target rates. In our implementation, continuous sensors ask for a single target rate, while event sensors ask for a set of target rates, which could match different video qualities. The interpretation of the single target rate is that this is the minimum rate that the sensor would need to offer good service, so the rate allocation algorithm tries to match or exceed that rate allocation. The interpretation of the multiple target rates is that the sensor will use the highest one of these rates that is allowed by its rate allocation, meaning that ideally the rate allocation algorithm should exactly match one of these rates, otherwise bandwidth may remain unused.

Rate Allocation

Whenever a new sensor is connected or an existing one is disconnected, the system is checked to see whether global adjustments need to be made, separately in each sensor class. First, we calculate the total rates requested (not assigned) by the sensors of each class. If these are below the available bandwidth, new sensors will get their requested rate, existing sensors that were previously rate-limited will increase their rate by 20%, while all other existing sensors will get their requested rate. If, however, the requested bandwidth is higher than the available one, the available rate is shared *equally* among all sensors of that class. The detailed procedure is given in Algorithm 2, as applied to continuous sensors.

On the other hand, for event sensors which only support specific rates, the rates initially assigned must be adjusted to the highest supported rate that is less than or equal to their initial allocation. As a result, some of the initially allocated

Algorithm 2 RT-SENMOS Rate Allocation - Continuous sensors.

```
1: procedure RATE_ALLOCATION - PART 2
2:     $continuous\_total\_bw := calculate\_total\_requested\_BW(cont\_sensors\_list)$
3:     $total\_BW := get\_total\_BW()$
4:     $continuous\_BW\_share := total\_BW * (1 - configuration.get\_event\_share())$
5:     if $continuous\_total\_bw < continuous\_BW\_share$ then
6:         for $sensor\ in\ cont\_sensors\_list$ do
7:             if $sensor\ left\ AND\ sensor.current\_rate() > sensor.requested\_rate$ then
8:                 $continue$
9:             else
10:                $assign\_rate(sensor.requested\_rate)$
11:    else
12:        $equal\_share := continuous\_total\_bw/num\_of\_cont\_sensors$
13:        for $sensor\ in\ cont\_sensors\_list$ do
14:            if $sensor\ left\ AND\ sensor.current\_rate() > equal\_share$ then
15:                $continue$
16:            else
17:                $assign\_rate(equal\_share)$
```

bandwidth may remain unused. In this case, we allocate the remaining bandwidth to those event sensors whose allocated bandwidth is smaller than their desired bandwidth and their transmission is not congested. The rate allocation scheme for event sensors is elaborated upon in Algorithm 3; the algorithm for assigning the remainder is explained below, as it is reused in the congestion management algorithm.

Note that in both algorithms, if the event that triggered the algorithm was a sensor leaving the mix, the algorithm first checks if the current rate of a sensor is higher than its requested or equal share. If so, there is no need to change it (i.e., reduce it), since the overall available bandwidth must have increased.

After all bandwidth is allocated, new sensors are notified of their assigned rate in the start transmission message, while existing sensors are notified by a rate update message [50]; the same procedure is used for both event and continuous sensors.

Algorithm 3 RT-SENMOS Rate Allocation - Event sensors.

```
1: procedure Rate_Allocation - Part 1
2:     $event\_total\_bw := calculate\_total\_requested\_BW(event\_sensors\_list)$
3:     $total\_BW := get\_total\_BW()$
4:     $event\_BW\_share := total\_BW * configuration.get\_event\_share()$
5:     $unused\_BW := 0$
6:     if $event\_total\_bw < event\_BW\_share$ then
7:         for $sensor\ in\ event\_sensors\_list$ do
8:             if $sensor\ left\ AND\ sensor.current\_rate() > sensor.requested\_rate$ then
9:                 $continue$
10:            else
11:                $assign\_rate(sensor.requested\_rate)$
12:        $unused\_BW := event\_BW\_share - event\_total\_bw$
13:    else
14:        $equal\_share := event\_total\_bw/num\_of\_event\_sensors$
15:        for $sensor\ in\ event\_sensors\_list$ do
16:            if $equal\_share\ not\ in\ list\ of\ supported\ rates\ by\ sensor$ then
17:                $new\_rate := find\_suitable\_bitrate(equal\_share)$
18:                $unused\_BW + := equal\_share - new\_rate$
19:            else
20:                $new\_rate := equal\_share$
21:            if $sensor\ left\ AND\ sensor.current\_rate() > new\_rate$ then
22:                $continue$
23:            else
24:                $assign\_rate(new\_rate)$
25:    $exploit\_unused\_BW(event\_sensors\_list,\ unused\_BW)$
```

Congestion detection and management

While the sensor mix remains static, RT-SENMOS runs Algorithm 4 to detect congestion and adapt the assigned rates to prevailing conditions. Congestion detection is based on the assumption that the congestion losses experienced by the sink in each connection are more akin to those occurring in a network with Random Early Detection (RED) [52] rather than drop-tail gateways. To be more exact, due to the large number of sensors and the convergence of their data around the sink, congestion losses are expected to be randomly distributed among connections, with the loss probability increasing with the level of congestion. Furthermore, we assume that each sensor class has some minimum requirements in order to provide an acceptable service level, expressed by a maximum acceptable loss rate max_rate; this loss rate accounts for wireless losses that do not indicate congestion. The congestion management scheme monitors the current loss rate, $loss$, for each sensor, and uses the difference between max_rate and $loss$ to make its decisions. Specifically, when $max\_rate < loss$, the sink decreases the assigned rate in proportion to the current loss rate: $newRate = currentRate * (1 - loss)$. When however $max\_rate > loss$, the sink increases the assigned rate in proportion to the difference between the threshold and the current loss rate: $newRate = currentRate * (1 + max\_rate - loss)$. As an example, if $max\_rate = 3\%$, when $loss = 1\%$ the rate will be increased by 2%, while when $loss = 5\%$ the rate will be decreased by 5%. To allow the network to react to these changes, we do not modify the rates for a small period after each change, unless if the sensor pool changes, in which case we apply the previous (global) algorithm. The event sensor rates are modified in the same way as described above, that is, we adjust their allocations to their actually desired rates.

Algorithm 4 RT-SENMOS Congestion Detection and Management.

```
1: procedure CONGESTION_HANDLER
2:     $sorted\_event\_sensors\_list\_decr := sort\_loss\_rate\_decrease(event\_sensors\_list)$
3:     $sorted\_event\_sensors\_list\_incr := sort\_loss\_rate\_increase(event\_sensors\_list)$
4:     $sorted\_cont\_sensors\_list\_decr := sort\_loss\_rate\_decrease(cont\_sensors\_list)$
5:     $sorted\_cont\_sensors\_list\_incr := sort\_loss\_rate\_increase(cont\_sensors\_list)$
6:     for $sensor\ in\ sorted\_event\_sensors\_list\_decr$ do
7:         if $sensor.current\_loss\_rate > sensor.loss\_threshold$ then
8:             $new\_rate := sensor.current\_rate * (1 - sensor.current\_loss\_rate)$
9:             $assign\_rate(find\_suitable\_bitrate(new\_rate))$
10:    for $sensor\ in\ sorted\_cont\_sensors\_list\_decr$ do
11:        if $sensor.current\_loss\_rate > sensor.loss\_threshold$ then
12:            $new\_rate := sensor.current\_rate * (1 - sensor.current\_loss\_rate)$
13:            $assign\_rate(new\_rate)$
14:    if $changes\ performed$ then
15:        Wait for changes to take effect
16:    for $sensor\ in\ sorted\_event\_sensors\_list\_incr$ do
17:        if $sensor.current\_loss\_rate < sensor.loss\_threshold$ then
18:            $new\_rate := sensor.current\_rate * (1 + sensor.loss\_threshold - sensor.current\_loss\_rate)$
19:            $assign\_rate(find\_suitable\_bitrate(new\_rate))$
20:    for $sensor\ in\ sorted\_cont\_sensors\_list\_incr$ do
21:        if $sensor.current\_loss\_rate < sensor.loss\_threshold$ then
22:            $new\_rate := sensor.current\_rate * (1 + sensor.loss\_threshold - sensor.current\_loss\_rate)$
23:            $assign\_rate(new\_rate)$
```

Congestion control and rate management is performed in two steps, as explained in Algorithm 4. In the first step we examine all sensors that may be congested, starting from the sensors with the highest loss rate and proceeding towards those with the lowest loss rate. Any sensors found to be in a congested state, i.e. exhibiting a loss rate higher than the threshold, are assigned reduced rates, as described above. After that, we wait for a little while for the network to settle, and then proceed in the second step, where we examine all sensors that may not be congested, from lowest to highest loss rate. Any sensors found to be in an uncongested state, are assigned increased rates, again as described above.

Note that the second part of Algorithm 4 is also used in the global rate allocation algorithm to assign any leftover bandwidth from the event sensor pool to other event sensors: we cycle through the sensors and only assign the leftover rate to a sensor if it allows the sensor to upgrade its rate to the next possible one, e.g. the next highest video quality.

4.4 Performance evaluation

4.4.1 Experimental setup

To evaluate the performance of RT-SENMOS, we relied on three different scenarios. The main concept of all scenarios is the emulation of a disaster area with different setup of connected sensors. In the *event sensor* and *continuous sensor* scenarios we have 14 sensors of the corresponding type, while in the *mixed sensor* scenario we have 14 event and 14 continuous sensors. In all scenarios an additional node acts as the sink, and all sensors are connected to the sink over a single physical

Table 4.1: Experimentation parameters

Parameter	**Value**
Continuous Sensor content size (MB)	8
Continuous Sensor bit rate (KBps)	82
Event Sensor content duration (sec)	30
Event Sensor bit rates (KBps)	50, 87, 187 and 312
Chunk size (bytes)	512
Loss samples stored per sensor	10
Bandwidth available at the sink (MBps)	2
Continuous Sensor share	10%, 30% and 50%
Target Loss rate	2% or 5%

hop using a shared wireless channel. The connection sequence of sensors emulates the movement of sink in the disaster area, with each scenario following a different pattern of sensors connection. We used our own implementations of RT-SENMOS and RCRT written in Java, adding a loss injection module to emulate congestion losses at the sink. Data was transmitted in 512 byte chunks. The emulated loss rate rather than being random, as in losses induced by wireless errors, was actually proportional to the current bandwidth usage, matching our assumption of a RED-like congestion loss model. Specifically, the loss rate at any given time was calculated by multiplying the fraction of the total bandwidth allocated by the sink with the target loss rate, which was either 2% or 5%, thus ranging from zero to the target loss rate.

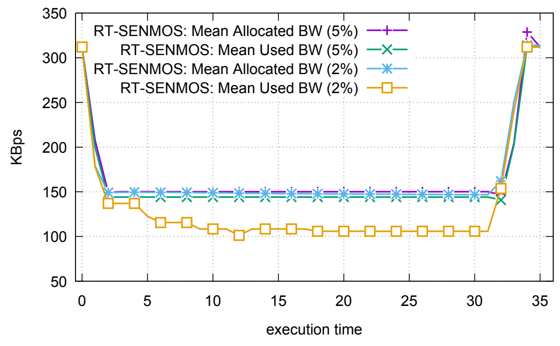
We assumed that the continuous sensors were security cameras that sent individual 8 MB frames at a minimum desired rate of 82 KBps, while the event sensors when triggered sent 30 sec of live video at 50, 87, 187 or 312 KBps, depending on their rate allocation. The total bandwidth available at the sink was 16 Mbps (or, 2 MBps). The *event sensor* scenario simulates a disaster recovery incident where a rescuer enters the disaster area and tries to get a short video from each camera. During the first 2 sec of the experiment the event sensors gradually connect to the mobile rescuer, and then we gather data until all sensors have completed their transmissions. In the *continuous sensor* scenario we keep the same setup as in the *event sensor* scenario, but continuous sensors transmit a fixed-size screenshot each; since in both scenarios each sensor only makes a single transmission, the main difference between the scenarios is in the way each type of sensor operates: event sensors can only use specific rates and recover from loses in parallel with data transmissions; continuous sensors can use any assigned rate (ideally, highest than the target rate) and recover from losses in rounds.

In the *mixed sensor* scenario we have a 100 m corridor, with event sensors deployed every 6.8 m, with the rescuer connecting to them while moving along the corridor. The continuous sensors are all connected at the beginning of the experiment. Again, we wait until all transmissions are complete. The experimental parameters are listed in Table 4.1. Regarding RCRT, we implemented its functionality, including all three rate allocation policies, on top of our message exchange scheme, using the same parameters values as in [22]. However, we only show results from the fair (all sensors get the same rate) and demand-proportional (each sensor gets the same proportion of its desired rate) policies, using the highest desired bit rate for event sensors. The rate-limited policy does not allow continuous sensors to exceed their minimum bandwidth, therefore it does not make sense to compare it with our scheme.

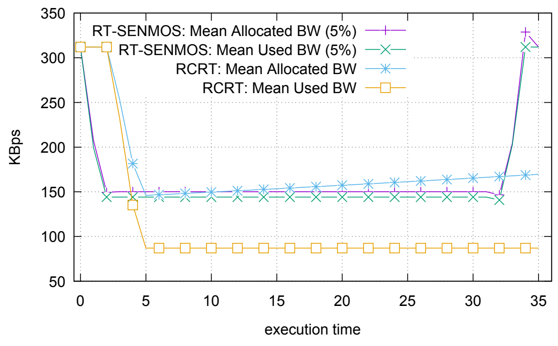
4.4.2 Experimental results

Event sensor scenario

We first discuss the results from the *event sensor* scenario. In Figure 4.5a we show the mean bandwidth allocated and used per sensor for acceptable loss rates of 2% and 5%, and a target loss rate of 2%. Note that since losses are tracked on a per sensor basis, even though the average loss rate peaks at 2%, individual connections can see much higher loss rates, thus triggering congestion control actions. Although in both cases RT-SENMOS quickly adapts as sensors enter and leave the network, the match between the assigned and used rates is much better when the acceptable loss rate is set to 5%.



(a) RT-SENMOS 2% and 5%



(b) RT-SENMOS vs RCRT

Figure 4.5: Event sensor scenario - Mean Bandwidth Allocation and Usage (2% loss).

In Figure 4.5b we compare RCRT[2] and RT-SENMOS with a 5% acceptable loss rate, when the actual loss rate is 2%. As in the previous figure, the allocated and used bandwidth in RT-SENMOS is almost the same, meaning that RT-SENMOS allocates rates which can actually be used, while RCRT exhibits a remarkable difference between the allocated and used bandwidth, as it strives to allocate as much bandwidth as possible to each sensor, rather than the actual rate that the sensor can use, thus wasting the available bandwidth. Note also that RCRT slowly increases the allocated bandwidth, since there is no congestion (as some of the allocated bandwidth remains unused), without of course affecting the bandwidth actually used.

This is also reflected in Figure 4.6, where we show the total bandwidth used

[2]We used the demand-proportional policy for this figure, but all policies have exactly the same behavior in this scenario.

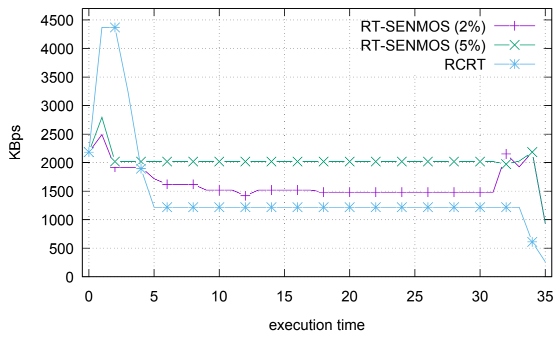


Figure 4.6: Event sensor scenario - Total Bandwidth Usage (2% loss).

by RCRT and RT-SENMOS with a 2% and a 5% acceptable loss threshold. RCRT is slow to detect the actual bandwidth available, as it relies on slow RTT measurements, causing an initial overallocation (it allocates 4.5 MBps, where only 2 MBps are available) which then leads to congestion. Even in steady state, RCRT exploits less of the available bandwidth than RT-SENMOS with a 2% acceptable loss threshold.

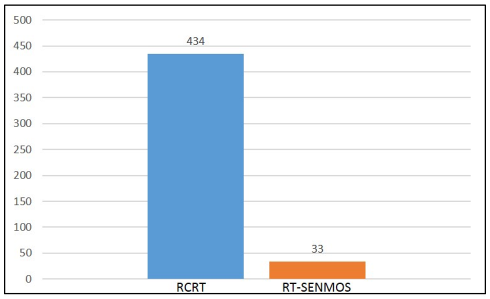
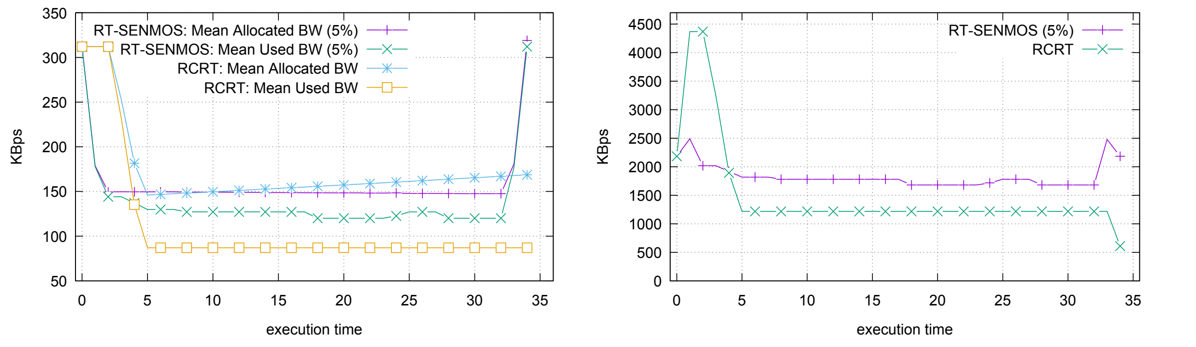


Figure 4.7: Event sensor scenario - Rate control messages sent (2% loss).

Finally, Figure 4.7 shows that RCRT generates twelve times as many control packets for rate allocation as RT-SENMOS[3] as RCRT sends rate allocation change messages to *all* nodes, in contrast to RT-SENMOS which controls each node separately. During the steady state period shown in Figure 4.5b, RCRT detects that the network is under-utilized and tries to slowly increase the rates assigned to nodes. As it is not

[3]All other control messages are the same in both protocols.

aware of the desired rates for the event sensors, these rate changes do not impact the real data transfers, but add a large traffic overhead. RCRT will only manage to upgrade the bit rate usable by those nodes after a long time and a huge number of rate control messages.



(a) Mean Bandwidth Usage - RT-SENMOS (b) Total Bandwidth Usage - RT-SENMOS vs RCRT

Figure 4.8: Event sensor scenario - Bandwidth Allocation and Usage (5% loss).

Figure 4.8 shows the same metrics as Figures 4.5b and 4.6, with a target loss rate of 5% rather than 2%. The gap between allocated and used bandwidth per sensor is much higher for RCRT, as shown in Figure 4.8a, even though the used bandwidth for RT-SENMOS has dropped due to the higher loss rate used. Note again the gradual increase in the rates allocated by RCRT, which has no tangible benefit to actual bandwidth used. The total bandwidth used across all sensors, shown in Figure 4.8a, is much higher for RT-SENMOS, due to its quick adaptation to the environment and its awareness of the rate allocation requirements of the application; again, RCRT overallocates rates at the beginning, leading to congestion.

In Figure 4.9, the control messages sent are again far more for RCRT than for RT-SENMOS, but at this setup the difference is reduced to seven times. The RT-SENMOS rate control messages are increased in comparison to Figure 4.7, since

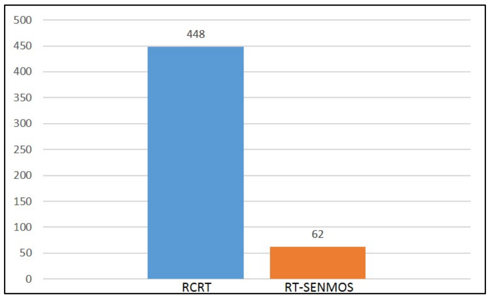
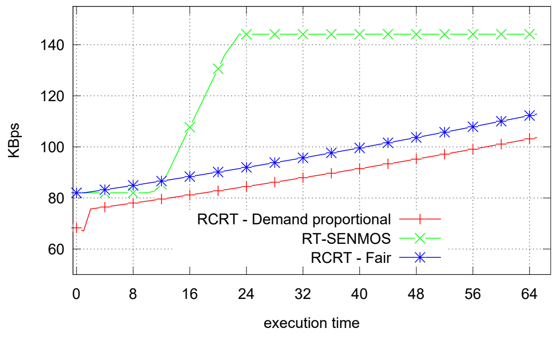


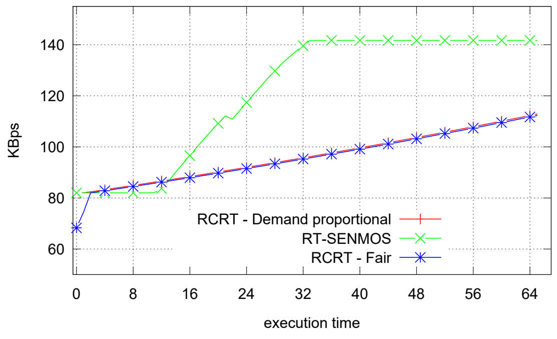
Figure 4.9: Event sensor scenario - Rate control messages sent (5% loss).

RT-SENMOS reaches the maximum acceptable loss rate and reconfigures the rates assigned to nodes. In other words, the control messages are ramped up due to the fact that RT-SENMOS tries to find a balance between the maximum utilization of available bandwidth and not exceeding the maximum acceptable loss rate. RCRT on the other hand always sends rate control messages to all sensors, ending up with a much higher control overhead. Finally, note that there is no appreciable difference in experiment completion time between the different protocols and their variants.

Continuous sensor scenario



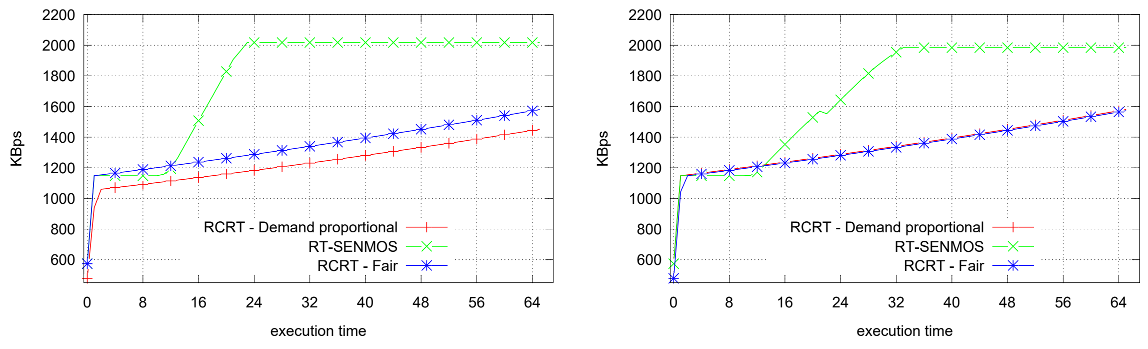
(a) 2% loss



(b) 5% loss

Figure 4.10: Continuous sensor scenario - Mean Bandwidth Allocated to sensors.

In the second set of experiments we examine the case where only continuous sensors exist. These sensors can use any bandwidth made available to them, starting at their indicated rate, unlike event sensors which operate only at specific rates, thus allowing us to compare the way each protocol exploits the available bandwidth without rate quantization effects. Figure 4.10 illustrates the rates allocated by RT-SENMOS with an acceptable loss rate of 5% and the two different RCRT policies (fair and demand-proportional) at actual loss rates of 2% and 5%. At the desired continuous sensor rate, the system is uncongested, therefore all policies gradually increase the bandwidth allocations. While both RCRT policies operate similarly, since all sensors have asked for the same rate, making fair the same as demand-proportional, they only increase the rates slowly, as also seen in the previous scenario; in contrast, RT-SENMOS quickly reaches the maximum available bandwidth.



(a) 2% loss (b) 5% loss

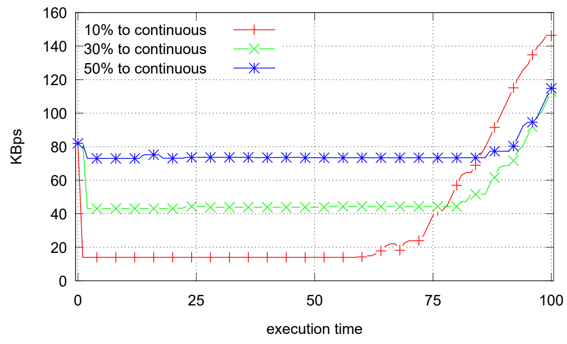
Figure 4.11: Continuous sensor scenario - Total Bandwidth Used by sensors.

In Figure 4.11 we show total allocated bandwidth to all sensors for the same experiment. It is clear that RT-SENMOS can quickly reach the maximum available bandwidth of 2 MBps, unlike the RCRT policies that do not manage to get even close until the end of the experiment. The change in the loss rate emulated does not make a big difference for the RCRT policies, since the system remains uncongested until the end of the run; as mentioned previously, the actual losses emulated are proportional to the congestion in the network. In contrast, RT-SENMOS quickly reaches the saturation point of the network, hence congestion does induce losses. The result is that in the 5% loss rate case RT-SENMOS takes slightly more time to reach its equilibrium state.

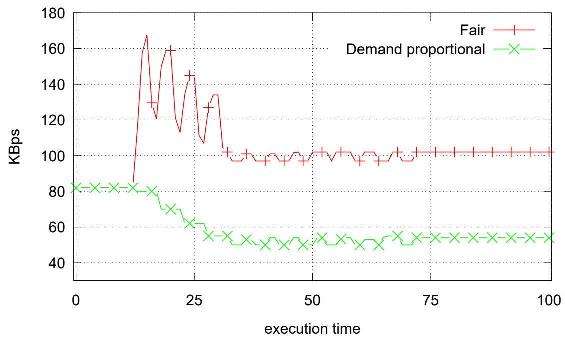
Mixed sensor scenario

We finally discuss the results from the *mixed* sensor scenario, where both sensor classes co-exist. In RT-SENMOS we assign a fixed fraction of the available bandwidth to each class, so Figures 4.12a and 4.13a show the mean bandwidth allocated to continuous and event sensors, respectively, when continuous sensors take up 10%, 30% and 50% of the bandwidth. The fraction allocated to each class balances their performance against each other, allowing the application to determine which sensor class it values more. We can also see that each sensor class quickly converges to a fair allocation of rates: event sensors are slowly connected, hence they only need to reduce their rates when a large number of them are in the pool, while continuous sensors are all connected at the beginning, hence they quickly reach their fair shares, which increase as some of them complete their transmissions.

Figures 4.12b and 4.13b show the same metrics for RCRT, using the two allocation policies available, namely, fair and demand-proportional. As the system quickly becomes congested as event sensors join, the two policies clearly show their differences. With the fair policy, RCRT attempts to equally share the rate among all sensors, so they all converge around 100 KBps, which is more than enough for the continuous sensors, but too low for the event sensors. On the other hand, the demand-proportional policy attempts to allocate to each sensor class the same fraction

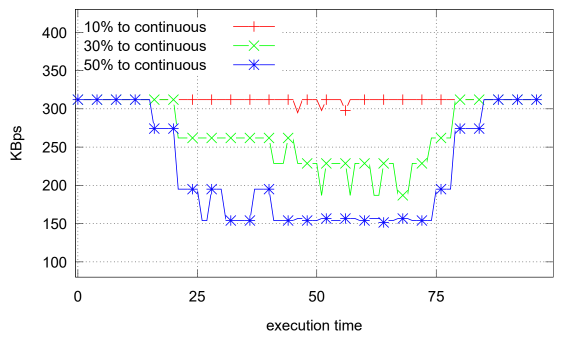


(a) RT-SENMOS

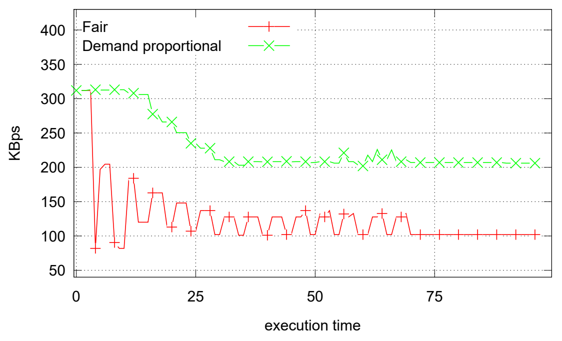


(b) RCRT

Figure 4.12: Mixed sensor scenario - Mean Bandwidth Allocated to continuous sensors (2% loss).



(a) RT-SENMOS

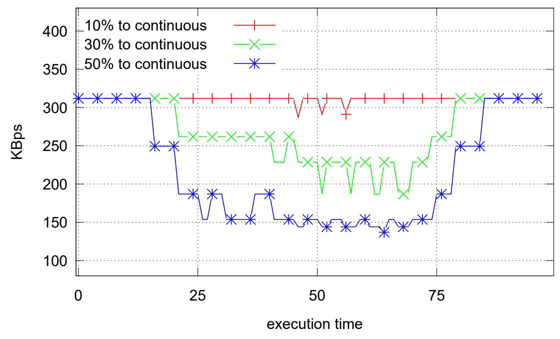


(b) RCRT

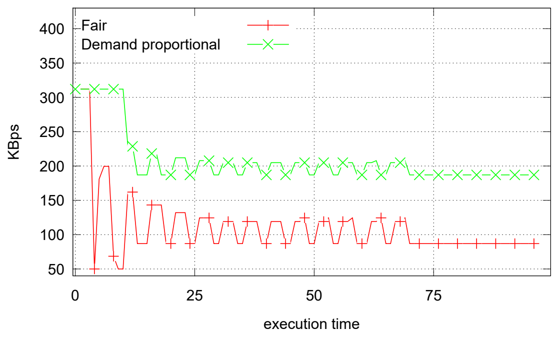
Figure 4.13: Mixed sensor scenario - Mean Bandwidth Allocated to event sensors (2% loss).

of their desired rates, hence ending up with 50 KBps for continuous sensors (less than the desired 82 KBps) and 200 KBps for event sensors (less than the highest rate, more than the second highest rate). As a result, even though the demand-proportional policy is better adjusted to a heterogeneous sensor mix, the allocations it produces are not satisfactory to either class.

As discussed above, while continuous sensors adapt to any rate allocation offered, meaning that the available and used bandwidths are generally the same, event sensors can only send at specific rates. As shown in Figure 4.14a, the mean



(a) RT-SENMOS



(b) RCRT

Figure 4.14: Mixed sensor scenario - Mean Bandwidth Used by event sensors (2% loss).

bandwidth used by event sensors with RT-SENMOS is nearly exactly the same as the one allocated, shown in Figure 4.13a, while with RCRT the bandwidth used, shown in Figure 4.14b, is less than the one allocated, shown in Figure 4.13b, exactly as in the event sensor scenario above. Specifically, with the fair policy the event sensors only use 87 KBps out of the 100 KBps allocated, while with the rate-proportional policy they only use 187 KBps out of the more than 200 KBps allocated. Again, there is no appreciable difference in the experiment completion time between the different protocol and their variants.

4.5 Conclusion

We have presented and evaluated a reliable transport protocol for sensor networks, RT-SENMOS, especially suitable for disaster recovery applications with a rescuer as the sink. RT-SENMOS is purely sink driven and implemented at the application layer, thus allowing application policies to be set at the sink, using a basic toolkit of error and congestion control policies that we have implemented. This allows the sink to split the bandwidth between different classes of sensors and within each sensor class depending on application preferences, without modifying the sensors. Finally, RT-SENMOS controls each sensor individually, depending on the level of congestion, as estimated by measuring packet loss.

We have also provided a performance evaluation of an actual implementation of our protocol over a real network with emulated losses against RCRT. The results show that our approach better exploits available bandwidth, unlike RCRT which is unaware of sensor classes in general and their specific target rates in particular. RT-SENMOS allocates all available bandwidth and tries to meet the actual requirements of each class, while RCRT utilizes 60% to 90% of the available bandwidth, depending on the rate allocation policy used. The experiments show that the gain from our approach is more visible in scenarios with sensors that can operate using a set of rates rather than any available rate, either in isolation or with other sensor types. Furthermore, RT-SENMOS requires a much smaller number of control messages than RCRT, as it sends rate control messages only to individual nodes and only when these are needed, unlike RCRT which constantly sends new rates to all nodes. Finally, RT-SENMOS is much quicker to adapt to network conditions than RCRT, reaching full bandwidth utilization much faster than RCRT which only gradually adapts to the available bandwidth.

Chapter 5

Streaming with BitTorrent

5.1 Introduction

The BitTorrent behavior outlined in Section 2.3 leads to a seemingly random piece download order, as each peer attempts to download the rarest pieces available at any given time; this allows the peer to become a more popular partner, while at the same time reducing the probability that the exchange will fail due to nodes leaving the swarm, as random pieces are quickly replicated. The problem with this behavior is that it hinders streaming, as even a file downloaded by 99% may be missing the first few pieces that need to be played out first. To help streaming, peers would ideally download pieces in their playing order, that is, sequentially. Although peers operate independently, which means that they can indeed implement this strategy, this would not only undermine the operation of the swarm at large, it would also lead to peers being idle when the next piece is not made available to them, while the pieces right after that one are available. For this reason, all proposed extensions to BitTorrent for streaming purposes try to balance out the desire of a streaming peer to download pieces sequentially, with the practical needs of downloading available pieces to maintain the proper BitTorrent operation.

In this chapter we evaluate three different policies which attempt to modify BitTorrent operation so as to better support streaming [53, 54, 55], without changing the fundamental design of the application. Although all three policies have been evaluated by their authors, these evaluations are not comparable to each other as they focus on different metrics and rely on different simulation platforms which implement only parts of the BitTorrent protocol. To ensure a fair and accurate comparison, we have implemented their functionality in our own comprehensive BitTorrent simulator [23], which allows us to compare them under the exact same settings and parameters. In addition, we have evaluated two different behaviors for the streaming player when it encounters a piece that has not yet been downloaded: it either skips the missing pieces, leading to playout gaps, or it waits until the missing pieces are downloaded, leading to stalls.

The structure of this chapter is as follows. In Section 5.2 we describe and compare the proposed streaming extensions to BitTorrent using an illustrated example. In Section 5.3 we explain the setup used for experimentation. Sections 5.4 and 5.5 present and discuss simulation results using the two different streaming player behaviors mentioned above. Finally, we conclude in Section 5.6.

5.2 Streaming Extensions

5.2.1 Fixed Size Window

The Fixed Size Window (FSW) [53] method uses a sliding window (of a fixed size) for piece selection. The FSW scheme proposes two modifications that allow us to deliver multimedia data on time. The first modification concerns the replacement of the rarest-first chunk downloading policy of BitTorrent by a policy requiring peers to download first the chunks that they will watch in the near future. The second modification is a new randomized tit-for-tat peer selection policy that gives free tries to a larger number of peers and lets them participate sooner in the media distribution.

The window in FSW covers k consecutive pieces starting from the first non available piece (k is a parameter of the program). The window contains both downloaded pieces, pieces that are currently being downloaded and not yet requested pieces. Inside the window, the selection of a piece follows the rarest-first policy, so that the BitTorrent client will receive rare pieces to exchange later, so to avoid being choked by the other peers. Pieces for downloading are chosen only inside this window. Note that the window slides to the right when its first piece has been downloaded. We did not consider the second modification (randomized tit-for-tat) in our implementation.

5.2.2 High Priority Set

A possible problem with FSW is that it does not consider rare pieces outside the window, which may become useful in the tit-for-tat exchange. Another problem is that by fixing the window size it limits the choice of pieces as the window fills up, thus wasting available downlink bandwidth. These problems are solved by the High

Priority Set (HPS) or BiTOS approach [54], in which a fixed size set holds the next pieces in sequence that have not been download already. In contrast to the FSW method where a window of size k covers exactly k consecutive pieces, out of which at most k pieces have not been downloaded, in the HPS method a set of size k covers at least k pieces in the piece sequence space, out of which exactly k pieces need to be downloaded or are currently being downloaded.

In addition to the window being elastic, in HPS we can also download pieces outside the window, using again the rarest first piece selection policy, so that we do not waste downlink bandwidth and have the opportunity to download rare pieces outside the window. This increases the probability that the client will not be chocked in the future (due to the Tit-for-Tat policy), by having enough rare pieces to exchange with the other peers. We choose to download from inside the window with a probability p (which is a parameter of the program), and outside the window with a probability $1 - p$, in both cases choosing a piece with the rarest-first policy.

The HPS is modified whenever any of its pieces completes downloading, as this piece must be removed from the HPS and added to the Received Pieces Set. This is the main difference from FSW, as in HPS the window only holds pieces that are currently being downloaded or have not been requested yet.

5.2.3 Stretching Window

Finally, the Stretching Window (SW) approach [55] is a mix of the FSW and HPS approaches. The SW resembles HPS in the fact that it contains (up to) k non downloaded pieces, but pieces are only requested from within the SW. In addition, the distance between the first and last piece in the SW in terms of the piece sequence space is bounded by a bound $b > k$; hence, the SW may contain up to k pieces, provided these pieces do not cover more than b consecutive slots in the piece sequence. Other than that, SW operates exactly as HPS. The differentiation with FSW is only that the size of the window can be stretched rather than being constant. A parameter of this policy is the b to be used for bounding the growth of the window. By stretching the window, SW tries to avoid getting stuck due to a missing piece (unlike FSW); by limiting the stretch, it tries to maintain focus on the upcoming pieces (unlike HPS).

5.2.4 An example of policies operation

The functionality of these three protocol extensions is explained in Figure 5.1. Pieces marked with an X were not downloaded in time, pieces marked OK have already been downloaded, pieces marked D are currently being downloaded and unmarked pieces have not started downloading yet. Arrows indicate the current playback position, as well as the window limits (for FSW) or HPS limits (for HPS and SW).

The first part of the figure explains the FSW extension. The player is in second piece, the window starts from the first in progress piece and covers $k = 4$ consecutive pieces, including pieces three to six. The window includes non downloaded pieces, currently downloading and already downloaded pieces. This is the main difference from the other extensions, as we include in the window the pieces that have already been downloaded.

In the second part with *HPS* we denote the window of the corresponding extension. As we describe in section 5.2.2, this window contains only the k pieces that are currently being downloaded or have still not been requested; downloaded pieces are removed from the HPS. In addition, pieces may be requested outside the HPS.

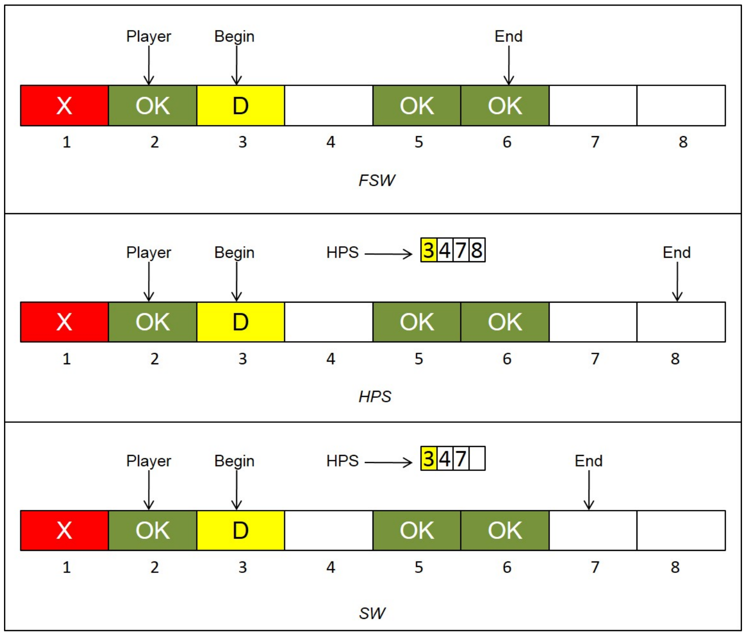


Figure 5.1: Example of windows in the different policies.

In the third part of the picture we have the SW extension. In this extension the window contains (as in HPS) only pieces that are currently being downloaded or have still not been requested, but the window can grow up to a bound in the piece sequence space (in this example the bound is $b = 5$). As a consequence, the window may be smaller than with HPS, as shown in the figure. In addition, pieces are only requested from within the SW.

5.3 Experimental Setup

5.3.1 BitTorrent Simulator

For the simulation-based comparison, we modified our own detailed OMNeT++ Simulator for BitTorrent [23], in order to add the necessary functionality for the three streaming extensions discussed above. Our simulator models not only the detailed protocol messages exchanged by BitTorrent peers and the tracker, it also models all the TCP/IP messages exchanged, using the INET framework [56], a fact that makes the results more realistic and stresses protocol performance to its limits. We have used transit-stub topologies generated by the GT-ITM module [57], including both core and access routers. Each scenario was executed 10 times with different random seeds, which means that peers were deployed in different ways in the network.

5.3.2 Network and content setup

Our results are based on a scenario with one initial seeder and 120 peers (*leechers*) which join the swarm at random times, starting from scratch. This means that peer joins are incremental, instead of a flash crowd join. The topology we used consists of four Autonomous Systems (AS) and 192 access routers in total. Peer access links have asymmetric uplink and downlink bandwidths: 20% have 1/4 Mbps capacity (uplink/downlink), 40% have 1/8 Mbps, 25% have 2/12 Mbps and 15% have 2/24 Mbps.

The streaming application modeled was a video player attempting to playback a 256 Kbps video stream. We assumed that the video was encoded in an MPEG like manner, where each Group of Pictures (GOP) was mapped to exactly one BitTorrent piece. We set the GOP/piece size to 192 KB, which translates to 6 seconds of video. In this manner, a piece that has not been downloaded on time, will cause a 6 second gap in the video, but it will not prevent the next piece from decoding[1]. The block size was set to 16 KB. The entire video size was 200 MB, which corresponds to around 106 minutes of playtime, or 1067 pieces in total.

[1]This is actually a slight simplification, since the I-frame in each GOP is also used by B-frames in the previous GOP.

Table 5.1: Simulation parameters

Parameter	**Value**
Video size (in MB)	200
Video Bit rate (in Kbps)	256
Piece size (in KB)	192
Block size (in KB)	16
Number of pieces for prefetch buffering	1 or 5
Window size k (% of total num of pieces)	2% or 8%
Probability p (only for HPS)	80% or 90%
Bound b in pieces (only for SW)	30 or 100

We kept all the default BitTorrent settings unchanged e.g. optimistic unchoke interval, number of connections per peer, as given in [23]. To make our model relevant to live streaming, we assumed that each peer will remain in the swarm and seed other peers until the video playback reaches its end, even if the peer has already completed the video download, i.e. that the peer leaves the swarm at playback completion. The initial seeder on the other hand remains permanently in the swarm, thus there is always at least one source for every piece, modeling a video source that exploits BitTorrent extensions to offload some of the traffic to the peers.

In addition, we assumed that each peer starts by prefetching a few initial pieces (either 1 or 5), as in most media players, in order to provide a satisfactory buffer to the player before starting playback. During the prefetch period, we use the rarest-first policy only for these first pieces, until they are all completely downloaded. We set the base window size k to 2% or 8% of the entire number of pieces for all algorithms, which in our case translates to 21 or 85 pieces, respectively. The probability p was set to either 80% or 90% for HPS. The bound b for SW was set to 30 pieces for $k = 21$ and to 100 pieces for $k = 85$, therefore SW can grow its window more than FSW, but not as much as HPS. Table 5.1 summarizes these parameters.

5.4 Evaluation with player skipping

5.4.1 Assumptions and metrics

The main problem with streaming is that the player may run out of data; in BitTorrent terms, it may reach a piece that has not been downloaded yet. In this case, it has two options: either skip the piece, or stall until the piece is downloaded. In this section we examine the skipping approach, while in the next section we examine the stalling approach. For the skipping approach, we used the following metrics to evaluate the various proposed BitTorrent extensions [58]:

- *Piece loss*: The most important metric in our case is piece loss. A piece is characterized as lost if the player reaches it and the piece is not completed, leading to a gap in the video; this includes the case where the piece is currently being downloaded.

- *Prefetch time*: We measure the necessary time for prefetching the first pieces of the video file. In other words, the time the user of the video application should wait for the player to start.

- *Download duration*: This metric shows how much time the entire download took; it is interesting to compare this with the (fixed) playing time, to see how far ahead the download is from the player.

In the following subsections, we present simulation results for these metrics with the three protocol modifications presented above.

5.4.2 Piece loss

In Figure 5.2 we show the piece loss for each protocol modification, with the window set to 2% of the video size and either 1 or 5 pieces prefetched. It is obvious that with more prefetched pieces the performance slightly improves with all protocols; the penalty is an increased delay before starting playback, as shown in the next section. While all protocols exhibit acceptable performance, HPS works best, with piece losses of around 1-1.5%, while FSW and SW suffer from losses of around 1.5-2.5%. Among the two different probabilities in HPS for selecting pieces within the window, the highest one (90%) works slightly better. On the other hand, FSW and SW have nearly identical performance, despite the fact that SW can grow its window to nearly 50% more than FSW.

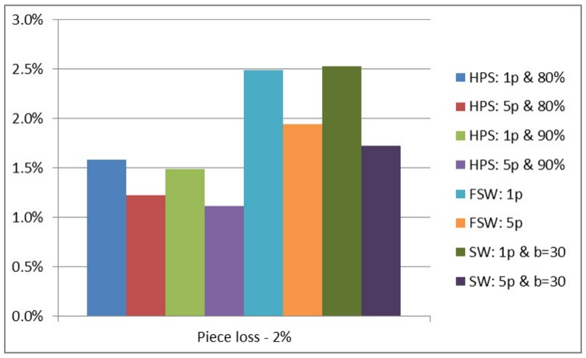


Figure 5.2: Piece loss for 2% window size (%).

Figure 5.3 shows the corresponding data for a window size of 8%. In this case HPS becomes the worst performer, with losses of roughly 3.5-5%, while FSW and SW actually work better than in the previous case, with losses of 1.5%. Again prefetching more pieces leads to better performance in all schemes. For HPS, the higher probability for selecting pieces within the window works noticeably better, while FSW and SW have again nearly identical performance.

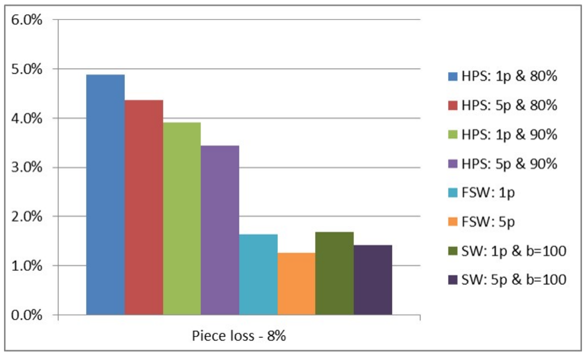


Figure 5.3: Piece loss for 8% window size (%).

Looking at both scenarios, it is clear that the window size is an important parameter. With a 2% window (21 pieces), FSW and SW cannot keep up with HPS which can grow its window more. With a 8% window however, FSW and SW have enough pieces to download; there is no need to grow the window too much. Regarding the HPS option of downloading outside the window, in both scenarios HPS works better when it is less likely to do so, while FSW and SW that do not offer this option work even better when the window is not too small. Finally, note that FSW and SW are more predictable than HPS, since their performance changed only slightly from one scenario to the other.

5.4.3 Prefetch time

The buffering time at the beginning of a video fluctuates between 21 and 33 seconds (average values), depending on the number of pieces we have chosen to download; Figure 5.4 shows the data for a 2% window size, while Figure 5.5 presents the case for a 8% window size. Note that when only one piece is prefetched, there is essentially no buffering: it is necessary for the first piece to be downloaded in order for the player to begin anyway, therefore these experiments show the minimum buffering time. With five pieces prefetched, the user must wait a little bit longer, around 50% more, hoping for better performance later on, although the improvements in loss rates are low, as discussed in the previous section.

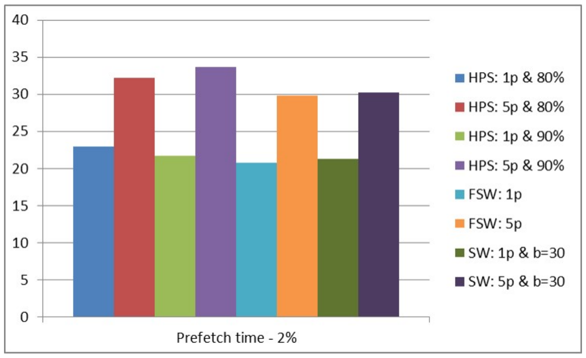


Figure 5.4: Prefetch time for 2% window size (sec).

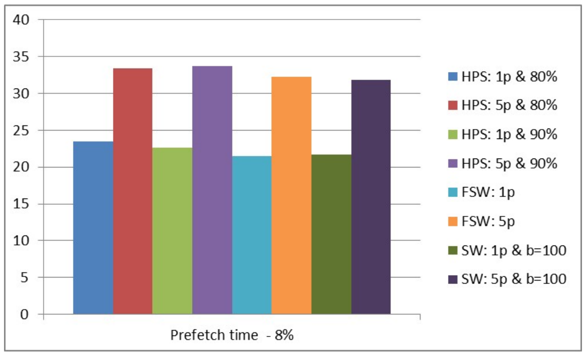


Figure 5.5: Prefetch time for 8% window size (sec).

These waiting times seem to be large, but this is due to the latency from the first joined peers in the swarm, when there are not many sources available, except for the initial seeder, therefore peers have to wait in order to receive these first pieces via opportunistic unchokes. Note that there are hardly any differences between the various protocols in this metric, since the protocol modifications have not started operating yet. The same is true of the window size, which is disregarded during the prefetch period.

5.4.4 Download time

The time required to complete the video download, as shown in Figures 5.6 (2% window size) and 5.7 (8% window size), is lower for HPS and higher for FSW and SW. While SW and FSW are nearly identical, in HPS it seems that a higher probability to download within the window leads to worse download times (recall that it also led to lower loss rates, so there is a tradeoff here). On the other hand, the number of prefetched pieces only has a minor impact on completion time, as the prefetch period is dwarfed by the rest of the download.

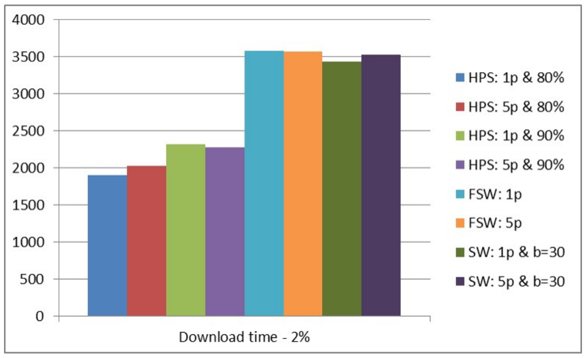


Figure 5.6: Download time for 2% window size (sec).

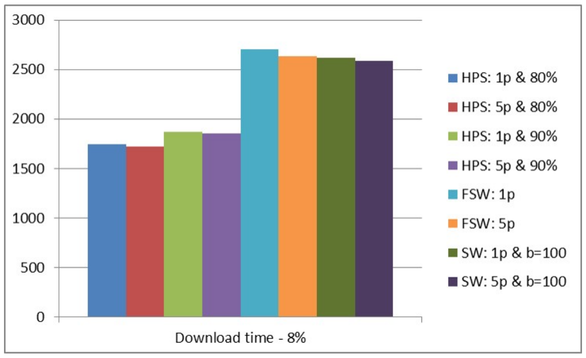


Figure 5.7: Download time for 8% window size (sec).

We point out that since the playback time is 6400 sec, all schemes manage to complete the download well before the playback ends. With a larger window, the downloads complete even faster. Since clients remain in the swarm until their download completes, this means that there are always plenty of peers to exchange pieces with, except in the very beginning of the exchange, when everyone has to rely on the initial seeder; we surmise that most of the observed losses are due to these first peers.

5.4.5 Overall Evaluation

Based on the above evaluation, it is clear that there is no dominant scheme; HPS exhibits lower loss rates than FSW and SW for smaller windows, but with larger windows the situation is reversed. Since HPS works slightly better with a lower probability to request pieces outside the window, the ability of HPS to download outside the window does not seem beneficial. FSW and SW perform nearly identically, therefore the extra complexity of SW does not seem to be worthwhile. Prefetching does lower the loss rates slightly, but at the cost of adding nearly 10 extra seconds of startup delay until the pieces are downloaded. Regarding download times, HPS is clearly the winner, but since all protocols complete the download well before the end of playback, this may not be as important as a reduced loss rate.

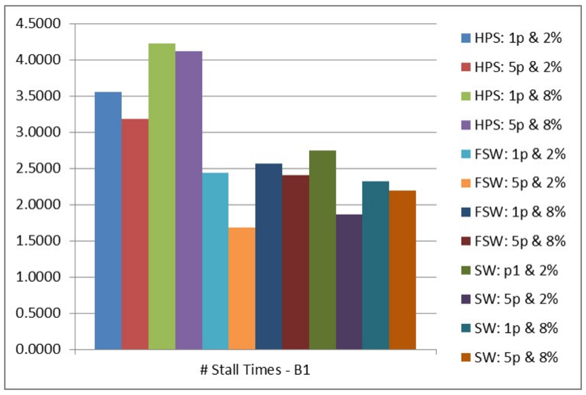
5.5 Evaluation with player stalling

5.5.1 Assumptions and metrics

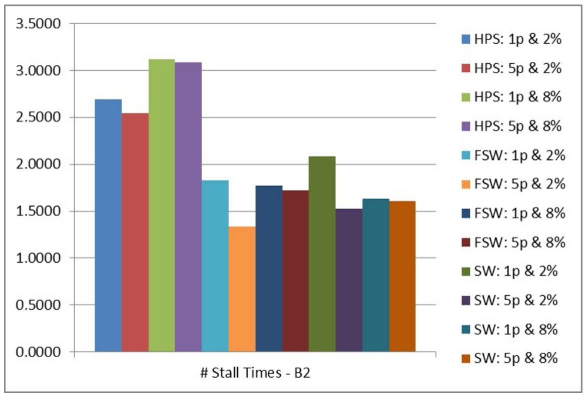
In this section we turn to the case where the player stalls until the missing pieces have been downloaded. We have used nearly the same setup as in the previous tests, with a couple of changes to make the experiments more realistic. First, the piece size was set to 112 KB, equivalent to 3.5 seconds of video, assuming again that the video was encoded in an MPEG like manner, with each GOP mapped to exactly one piece; this is closer to what many MPEG decoders use as the default interval for I-frames. Second, when a peer completes the download, it remains in the swarm as a seed only with a 50% probability; this means that there are not so many seeders as the downloads complete. We only show results for the HPS scheme with p set to 80% (as originally proposed in [54]) to avoid cluttering the figures. In this scenario, there is no loss to measure; instead, in addition to the download time, we also measure the following:

- *Stall periods*: The most important metric in our case is the number if stall periods per playback, that is, the number of times the video player stalled due to the next piece not having finished downloading when its playback time is up.
- *Average stall duration*: The second metric of interest is the duration of each stall period, that is, the time the player remains blocked whenever it has to stall.
- *Seeding time*: Since now 50% of the peers leave the swarm after their download completes, we measured the average time a peer remains in the swarm as a seeder.

We tested two different buffering strategies after a stall occurs with each protocol. In mode B1 the player stalls only until the next piece has completed downloading, while in mode B2 the player stalls until the next three pieces have been downloaded; the former strategy tries to minimize the average stall duration, while the latter tries to minimize the number of stall periods by prefetching more pieces (which may extend the stall duration, of course).



(a) B1

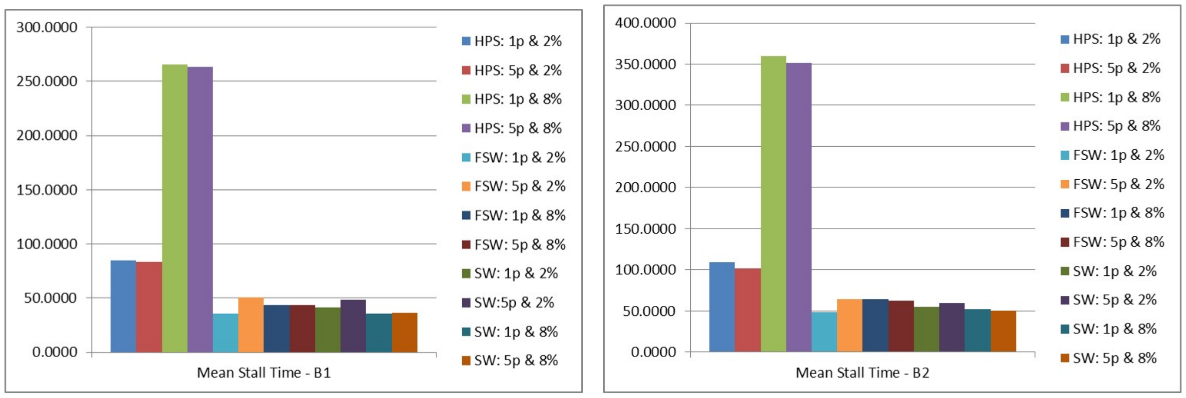


(b) B2

Figure 5.8: Number of Stall Periods per download.

5.5.2 Number of Stall Periods

Figures 5.8a and 5.8b shows the average number of stall periods during playback, that is, how many times the user will experience a stall during playback. As expected, the number of stall periods in Mode B1 is significantly higher (around 30% on average) than that in Mode B2 for all protocol extensions, since in Mode B1 the player only buffers the next piece before continuing. However, the absolute number of stalls is quite small in all cases (from 1.5 to 4) despite the large playback duration (more than 100 minutes), which is good news for user experience. For each individual protocol and for both buffering modes, the best performance is achieved by prefetching more pieces (5 rather than 1). The effect of the window size (2% or 8%) depends on the scheme. Among the different protocols, HPS has the worst behavior, as it has the largest effective window and it also downloads pieces outside the window, thus paying less attention to the pieces that are nearing their playback time. FSW and SW have very similar performance; FSW is slightly better with a larger window, while SW is slightly better with a smaller window.



(a) B1 (b) B2

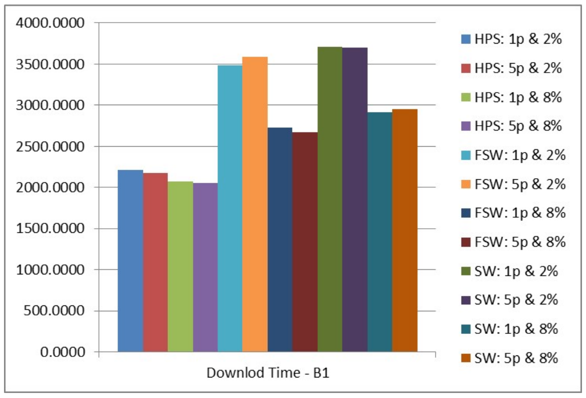
Figure 5.9: Average Stall Duration per download (sec).

5.5.3 Average Stall Duration

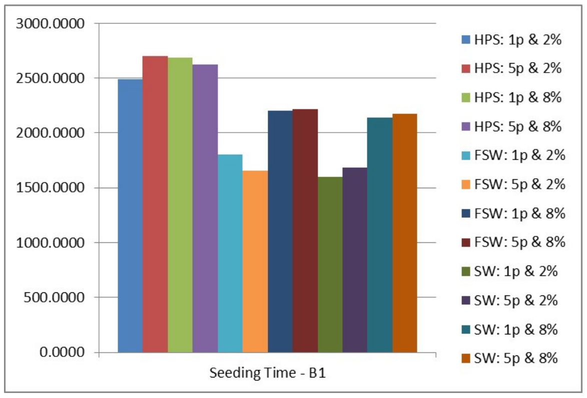
Figures 5.9a and 5.9b show the average stall duration during playback, that is, how long the user will have to wait in each stall period during playback. As expected, the average stall duration in Mode B2 is significantly higher (again around 30% on average, exactly counterbalancing the stall period metric) than that in Mode B1, since in Mode B2 the player waits until the next three pieces are buffered. The absolute average stall duration ranges from noticeable (from 40 to 90 sec for FSW and SW) to quite long (from 80 to 350 sec for HPS). In this metric, the effects of window size and initial buffering are insignificant. The FSW and SW protocols exhibit very similar performance, as in the previous metric, with FSW slightly better with a larger window and SW slightly better with a smaller window, but this time the HPS scheme is clearly unacceptable with the larger window size.

5.5.4 Download Time

Figures 5.10a and 5.10b shows the average download time and the average seeding time per download for buffering Mode B1; the results for both are nearly



(a) Download time (B1)



(b) Seeding time (B1)

Figure 5.10: Average download and seeding time per download (sec).

identical in Mode B2, as the buffering mode only changes the behavior of the player and not the protocol. Note that only 50% of the peers become seeders after they complete the download and until they finish playback. HPS offers the lowest download times due to the fact that its larger effective window and occasional downloads outside the window provide it with more opportunities for downloading rare pieces. In addition, the faster a peer completes the download, the more time it will remain as a seeder if it chooses to do so, since the playback duration is fixed, therefore HPS also exhibits higher seeding times. FSW and SW exhibit very similar performance, as in the previous metrics. However, since all protocols complete the download well before the end of the playback duration (which is 6400 sec), emphasis should be placed on the user visible performance metrics discussed above, where HPS has clear problems, rather than on the download time.

5.5.5 Overall Evaluation

The more realistic stall-based evaluation model adopted in this set of simulations clearly shows that HPS does not work well: compared to FSW and SW, it suffers from both more stalls and higher delays per stall, thus leading to visibly worse user experience, which is especially noticeable in the average stall delay metric. On the other hand, FSW and SW perform similarly, with SW better with smaller windows and FSW better with larger windows, indicating (as in previous section) that the extra complexity of SW is not worthwhile, provided the window is large enough. With FSW, the simplest but best performing approach, the worst case behavior is less than 2.5 stalls per download each lasting no more than 60 sec, which is reasonable for such large downloads. The two buffering modes tested led to the expected results, i.e. with less buffering per stall (mode B1) we have shorter stall periods, while with more buffering per stall (mode B2) we have fewer stall periods.

5.6 Conclusion

In this chapter we presented a simulation based performance comparison of three protocol extensions that add streaming capabilities to BitTorrent. We explored, through simulations using detailed TCP/IP message exchanges, the factors that impact protocol performance and user-visible behavior, using the exact same setup for each protocol extension, either with player skipping or with player stalling when the next piece is not available.

In experiments using skipping, we focused on three metrics: *piece loss*, *prefetch time* and *download time*. The collected data showed that the HPS approach presents the best performance in terms of download time, but its loss rate heavily depends on the window size. The SW and FSW approaches provided nearly identical performance, therefore there seems to be no point in the additional complexity of SW over FSW. While SW and FSW led to higher download times than HPS, they offered lower loss rates when the window size was large enough.

In the most realistic stalling mode, we evaluated the number of *stall periods* and their *average duration* when playing back a large video file, as well as the *download time*. The results indicate that the simplest approach, that is, FSW, works as good as any other, and that the overall user experience that it offers is quite reasonable, i.e. 1 to 2.5 stalls of 40 to 60 seconds each, for a video lasting 106 minutes. On the other hand, a more complex approach, like HPS, even though it offers markedly shorter download times, leads to much worse user visible performance.

Based on the results of both experiments, we conclude that that the simplest strategy of downloading within a fixed size window (FSW) works best overall, taking also into account its simplicity and stability, and that it offers reasonable performance at the user level when configured properly.

Chapter 6

Conclusions and future work

This dissertation addressed the need for effective content distribution for every communication setup; one-to-many, many-to-one and many-to-many. For one-to-many communication we chose reliable downloads over the PSI architecture, a promising candidate for the architecture of the Future Internet which offers support for native multicast. For the many-to-one case, we chose a Wireless Sensor Network scenario where multiple sensors communicate with a single sink. Finally, we used BitTorrent for the many-to-many case, studying the problem of adapting BitTorrent to media streaming. In all cases, we employed both network-level and user-level metrics, so as to show what the user-visible performance is and understand how it is produced by network-visible effects.

For the one-to-many communication case, we proposed an error control scheme suitable for the PSI architecture. Our mechanism, presented in Chapter 3, operates on top of PSI and solves the problem of feedback implosion from the subscribers to the publisher via hierarchical feedback aggregation. As a point of reference, we selected the most similar approach proposed for IP multicast, PGM, and adapted it to operate over the PSI network, comparing it against our RMTPSI scheme. We showed through simulations that our RMTPSI approach outperforms PGM in terms of control feedback required, without stretching the time needed to complete a download. Future work in this area includes an investigation of the delay interval that a relay point should wait for NAK aggregation, which represents a tradeoff between feedback traffic and completion time. Another direction is to study the time and resources needed for the first phase of the mechanism (setup). We also plan to move further and couple the error control scheme with an efficient solution for multicast congestion control over the PSI architecture. Finally, we are planning to examine the effectiveness of caching at relay points in order to enable local retransmissions of lost data, another idea borrowed from PGM.

For the many-to-one communication case, we proposed a sink-driven congestion and error control scheme in Chapter 4. The proposed scheme, RT-SENMOS, provides a reliable solution for sensor to sink communication in wireless sensor networks that exhibit a RED-like loss model. We compared our approach against RCRT, a sink-driven approach that is very similar to RT-SENMOS. We showed via a real implementation and network emulation that RT-SENMOS exploits all available bandwidth and adapts quicker to network changes than RCRT in a simple network topology. Future work in this area includes the study of multi-hop sensor topologies, along with an investigation of the benefits of exploiting intermediate nodes for NAK aggregation, thus enabling sensors to act as relay points, as in the RMTPSI approach presented in Chapter 3. Using sensors as relay points will boost performance if data caching and NAK aggregation are combined, causing intermediate nodes to act as local recovery points. Finally, we plan to make an accurate comparison of the computational resources required by

RT-SENMOS versus RCRT.

Finally, for the many-to-many communication case, in Chapter 5 we studied proposed extensions to the BitTorrent protocol in order to make it applicable to streaming applications. We implemented all proposed approaches over our detailed BitTorrent simulator, allowing a fair and accurate comparison of all proposed approaches under different application scenarios. We evaluated user-level performance either with a player that skips pieces that have not been downloaded on time, or with a player that stalls in such cases until the missing pieces have been downloaded, using user-level metrics to evaluate the applicability of each solution. Our simulation results indicate that the simplest strategy of focusing piece selection and downloads on a fixed size sliding window, performs better or equally well as any other one, without any adverse effects on network-level performance metrics. Indeed, we have evaluated in unpublished work many other variants on the proposed schemes, which have not shown any tangible benefits for the user, thus concluding that in this case, the simplest idea works best.

Appendix A

Acronyms

ACK	ACKnowledgement
AIMD	Additive Increase - Multiplicative Decrease
AS	Autonomous System
CDN	Content Delivery Network
DISFER	DIstributed Sensor systems For Emergency Response
DONA	Data-Oriented Network Architecture
DR	Designated Receiver
FSW	Fixed-Size Window
GOP	Group Of Pictures
GT-ITM	Georgia Tech Internet Topology Model
HPS	High-Priority Set
ICN	Information-Centric Networking
IoT	Internet of Things
LIPSIN	Line-speed Publish Subscribe Internetworking
MAC	Medium Access Control

NAK Negative Acknowledgment

NCF NAK Confirmation

NDN Named-Data Networking

NE Network Element

PGM Pragmatic General Multicast

P2P Peer to Peer

PSI Publish-Subscribe Internetwork/Infrastructure

RCRT Rate-Controlled Reliable Transport

RId Rendez-Vous Identifier

RMTP Reliable Multicast Transport Protocol

RMTPSI Reliable Multicast Transport for PSI

ROFL Routing On Flat Labels

RP Relay Point

RT-SENMOS Reliable Transport protocol for SEnsor Networks with MObile Sinks

RTT Round-Trip Time

RVP Rendez-Vous Point

SId Scope Identifier

SRM Scalable Reliable Multicast

SW Stretching Window

TM Topology Manager

WSN Wireless Sensor Network

Bibliography

[1] D. Clark, B. Lehr, S. Bauer, P. Faratin, R. Sami, and J. Wroclawsk, "Overlay networks and the future of the internet," *Communication & Strategies*, vol. 63, 2006.

[2] A. Rowstron and P. Druschel, *Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems.* Springer Berlin Heidelberg, 2001, pp. 329–350.

[3] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: a resilient global-scale overlay for service deployment," *IEEE Journal on Selected Areas in Communications*, pp. 41–53, 2004.

[4] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel, *Scribe: The Design of a Large-Scale Event Notification Infrastructure.* Springer Berlin Heidelberg, 2001, pp. 30–43.

[5] P. Ganesan, K. Gummadi, and H. Garcia-Molina, "Canon in G major: designing DHTs with hierarchical structure," in *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, 2004, pp. 263–272.

[6] T. Karagiannis, P. Rodriguez, and K. Papagiannaki, "Should internet service providers fear peer-assisted content distribution?" in *Proc. of the Internet Measurement Conference*, 2005, pp. 63–76.

[7] M. Caesar, T. Condie, J. Kannan, K. Lakshminarayanan, I. Stoica, and S. Shenker, "ROFL: routing on flat labels," in *Proc. of the 2006 ACM SIGCOMM*, 2006, pp. 363–374.

[8] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica, "A data-oriented (and beyond) network architecture," in *Proc. of the 2007 ACM SIGCOMM*, 2007, pp. 181–192.

[9] FP7 PSIRP: Publish-Subscribe Internet Routing Paradigm project. (2016) Home page. http://www.psirp.org/.

[10] Content-Centric Networking project. (2016) Home page. https://www.parc.com/work/focus-area/content-centric-networking/.

[11] NSF Named-Data Networking project. (2016) Home page. http://nameddata.net/.

[12] FP7 4WARD project. (2016) Home page. http://www.4ward-project.eu/.

[13] FP7 SAIL project. (2016) Home page. http://www.sail-project.eu/.

[14] FP7 COMET project. (2016) Home page. http://www.comet-project.org/.

[15] FP7 CONVERGENCE project. (2016) Home page. http://www.ictconvergence.eu/.

[16] FP7 PURSUIT: Pursuing a Pub/Sub Internet project. (2016) Home page. www.fp7-pursuit.eu/.

[17] J. Lin and S. Paul, "RMTP: a reliable multicast transport protocol," in *Proc. of the 1996 IEEE INFOCOM*, 1996, pp. 1414–1424.

[18] S. Floyd, V. Jacobson, S. McCanne, C.-G. Liu, and L. Zhang, "A reliable multicast framework for light-weight sessions and application level framing," in *Proc. of the 1995 ACM SIGCOMM*, 1995, pp. 342–356.

[19] J. Gemmell, T. Montgomery, T. Speakman, and J. Crowcroft, "The PGM reliable multicast protocol," *IEEE Network*, vol. 17, no. 1, pp. 16 – 22, 2003.

[20] C. Stais, G. Xylomenos, and A. Voulimeneas, "A reliable multicast transport protocol for information-centric networks," *Journal of Network and Computer Applications*, vol. 50, pp. 92 – 100, 2015.

[21] C. Stais, G. Xylomenos, and E. Zafeiratos, "RT-SENMOS: Sink-driven congestion and error control for sensor networks," in *Proc. of the IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, 2016.

[22] J. Paek and R. Govindan, "RCRT: Rate-controlled reliable transport for wireless sensor networks," in *Proc. of SenSys '07*, 2007, pp. 305–319.

[23] K. Katsaros, V. Kemerlis, C. Stais, and G. Xylomenos, "A BitTorrent module for the OMNeT++ simulator," in *Proc. of the IEEE MASCOTS*, 2009.

[24] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Computing Surveys*, vol. 35, pp. 114–131, 2003.

[25] K. V. Katsaros, N. Fotiou, X. Vasilakos, C. N. Ververidis, C. Tsilopoulos, G. Xylomenos, and G. C. Polyzos, "On inter-domain name resolution for information-centric networks," in *Proc. of the 2012 IFIP Networking*, 2012, pp. 13–26.

[26] G. Xylomenos, X. Vasilakos, C. Tsilopoulos, V. A. Siris, and G. C. Polyzos, "Caching and mobility support in a publish-subscribe Internet architecture," *IEEE Communications*, vol. 50, no. 7, pp. 52–58, 2012.

[27] P. Jokela, A. Zahemszky, S. Arianfar, P. Nikander, and C. Esteve, "LIPSIN: line speed publish/subscribe internetworking," in *Proc. of the 2009 ACM SIGCOMM*, 2009, pp. 195–206.

[28] M. Sarela, C. E. Rothenberg, T. Aura, A. Zahemszky, P. Nikander, , and J. Ott, "Forwarding anomalies in Bloom filter-based multicast," in *Proc. of the 2011 IEEE INFOCOM*, 2011, pp. 2399 –2407.

[29] C. Tsilopoulos and G. Xylomenos, "Scaling Bloom filter-based multicast via filter switching," in *Proc. of the 2013 IEEE ISCC*, 2013.

[30] F. Stann and J. Heidemann, "RMST: reliable data transport in sensor networks," in *Proc. of the IEEE SNPA Workshop*, 2003, pp. 102–112.

[31] Y. G. Iyer, S. Gandham, and S. Venkatesan, "STCP: a generic transport layer protocol for wireless sensor networks," in *Proc. of ICCCN*, 2005, pp. 449–454.

[32] B. Deb, S. Bhatnagar, and B. Nat, "ReInForM: reliable information forwarding using multiple paths in sensor networks," in *Proc. of the IEEE LCN*, 2003, pp. 406–415.

[33] Özgür B. Akan and I. F. Akyildiz, "Event-to-sink reliable transport in wireless sensor networks," *IEEE/ACM Transactions on Networking*, vol. 13, no. 5, pp. 1003–1016, Oct. 2005.

[34] C. Wang, K. Sohraby, V. Lawrence, B. Li, and Y. Hu, "Priority-based congestion control in wireless sensor networks," in *IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC'06)*, 2006.

[35] A. Woo and D. E. Culler, "A transmission control scheme for media access in sensor networks," in *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '01, 2001.

[36] C.-Y. Wan, S. B. Eisenman, A. T. Campbell, and J. Crowcroft, "Siphon: Overload traffic management using multi-radio virtual sinks in sensor networks," in *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, ser. SenSys '05, 2005.

[37] J. Y. Teo, Y. Ha, and C. K. Tham, "Interference-minimized multipath routing with congestion control in wireless sensor network for high-rate streaming," *IEEE Transactions on Mobile Computing*, vol. 7, no. 9, pp. 1124–1137, 2008.

[38] C.-Y. Wan, S. B. Eisenman, and A. T. Campbell, "CODA: Congestion detection and avoidance in sensor networks," in *Proc. of the 2003 ACM SenSys*, 2003, pp. 266–279.

[39] C. T. Ee and R. Bajcsy, "Congestion control and fairness for many-to-one routing in sensor networks," in *Proc. of the 2004 ACM SenSys*, 2004, pp. 148–161.

[40] BitTorrent.org. (2013) DHT protocol. [Online]. Available: http://www.bittorrent.org/beps/bep_0005.html

[41] ——. (2009) BitTorrent Protocol Specification. [Online]. Available: http://www.bittorrent.org/beps/bep_0003.html

[42] C. Diot, B. Levine, B. Lyles, H. Kassem, and D. Balensiefen, "Deployment issues for the IP multicast service and architecture," *IEEE Network*, pp. 78 –88, 2000.

[43] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *Proc. of the 2009 ACM CoNEXT*, 2009, pp. 1–12.

[44] S. Pingali, D. Towsley, and J. F. Kurose, "A comparison of sender-initiated and receiver-initiated reliable multicast protocols," in *Proc. of the 1994 ACM SIGMETRICS*, 1994, pp. 221–230.

[45] C. Stais, A. Voulimeneas, and G. Xylomenos, "Towards an error control sceme for a publish/subscribe network," in *Proc. of the 2013 IEEE ICC*, 2013, pp. 3743 –3747.

[46] NS-3 Simulator, "Home page," www.nsnam.org, 2013.

[47] A. Barabasi and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, pp. 509 –512, 1999.

[48] DIstributed Sensor systems For Emergency Response (DISFER) Project. (2016) Home page. http://www.aueb.gr/disfer.

[49] C. Stais, G. Xylomenos, and G. F. Marias, "Sink controlled reliable transport for disaster recovery," in *Proc. of the International Conference on Pervasive Technologies Related to Assistive Environments*, 2014.

[50] C. Stais and G. Xylomenos, "RT-SENMOS: Reliable transport for sensor networks with mobile sinks," in *Proc. of the IEEE Symposium on Computers and Communication (ISCC)*, 2015, pp. 105–110.

[51] K. Karenos and V. Kalogeraki, "Traffic management in sensor networks with a mobile sink," *IEEE Trans. on Parallel and Distributed Systems*, vol. 21, no. 10, pp. 1515–1530, 2010.

[52] S. Floyd and V. Jacobson, “Random early detection gateways for congestion avoidance,” *IEEE/ACM Transactions on Networking*, vol. 1, pp. 397–413, 1993.

[53] P. Shah and J. Paris, “Peer-to-peer multimedia streaming using BitTorrent,” in *Proc. of the IEEE International Performance, Computing, and Communications Conference*, 2007.

[54] A. Vlavianos, M. Iliofotou, and M. Faloutsos, “BiToS: Enhancing BitTorrent for supporting streaming applications,” in *Proc. of the IEEE INFOCOM*, 2006.

[55] P. Savolainen, N. Raatikainen, and S. Tarkoma, “Windowing BitTorrent for video-on-demand: Not all is lost with tit-for-tat,” in *Proc. of the IEEE GLOBECOM*, 2008.

[56] I. Baumgart, B. Heep, and S. Krause, “OverSim: A flexible overlay network simulation framework,” in *Proc. of the IEEE Global Internet Symposium*, 2007.

[57] E. Zegura, K. Calvert, and S. Bhattacharjee, “How to model an internetwork,” in *Proc. of the IEEE INFOCOM*, vol. 2, 1996, pp. 594–602.

[58] C. Stais, G. Xylomenos, and A. Archontovasilis, “A comparison of streaming extensions to BitTorrent,” in *Proc. of the IEEE ISCC*, 2011.