2007

A Multiple-Copy Scheme for Multi-Channel Stop-and-Wait HARQ

Yucheng Shih
San Jose State University

A Multiple-Copy Scheme for Multi-Channel Stop-and-Wait HARQ

A Project Report

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

By

Yucheng Shih

August 2007

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Professor Teng Moh, Computer Science Department

Professor Melody Moh, Computer Science Department

Dr. Wei-Peng Chen, Fujitsu Laboratory of America

ABSTRACT

A Multiple-Copy Scheme for Multi-Channel Stop-and-Wait HARQ

By Yucheng Shih

HARQ (Hybrid Automatic Repeat Request) combines ARQ (Automatic Repeat Request) with FEC (Forward Error Correction) to provide a reliable way to ensure that data are received correctly and in sequence. A multiple-copy HARQ scheme is proposed for WiMAX (Worldwide Interoperability for Microwave Access) to reduce the waiting time of erroneously received data in the receivers' buffer. In this project, the performances of WiMAX multi-channel stop-and-wait HARQ scheme are compared with that of the proposed multiple-copy HARQ scheme. The multiple-copy HARQ can send the same copy of a data burst on contiguous channels during noisy channel conditions so that the required time for an unsuccessfully received data burst to recover is shortened. It is beneficial for time sensitive data to have a shorter waiting time in a SS's (Subscriber Station) buffer. The simulation results show that the multiple-copy HARQ scheme requires only 56% of the time that the original WiMAX HARQ needs to successfully deliver erroneous data bursts in the SS's buffer. The throughput of the multiple-copy HARQ scheme can still reach more than 73% of the original WiMAX HARQ scheme's throughput.

ACKNOWLEDGEMENTS

I would like to express my appreciation to Professor Melody Moh for her advising and encouragement that make this project possible. I would also like to thank Professor Teng Moh for his assistance along the way and Dr. Wei-Peng Chen for his instruction with WiMAX. Lastly, thanks also to my parents and friends for their continued support.

TABLE OF CONTNETS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ACID: HARQ Channel Identifier
AES: Advanced Encryption Standard
AI_SN: ARQ Identifier Sequence Number
ARQ: Automatic Repeat Request
BER: Bit Error Rate
BLER: Block Error Rate
BS: Base Station
BSN: Block Sequence Number
CC: Chase Combining
CQI: Channel Quality Indicator
CRC: Cyclical Redundancy Check
DCD: Down Link Channel Descriptor
DL: Down Link
EAP: Extensible Authentication Protocol
FDD: Frequency Division Duplex
FEC: Forward Error Correction
GBN: Go Back N
HARQ: Hybrid Automatic Repeat Request
HSDPA: High Speed Downlink Packet Access
IE: Information Element
IP: Internet Protocol
IR: Incremental Redundancy
IrDA: Infrared Data Association
MIMO: Multiple Input Multiple Output
OFDMA: Orthogonal Frequency Division Multiple Access
PDU: Protocol Data Unit
QoS: Quality of Service
SR: Selective Repeat
SS: Subscriber Station
SW: Stop and Wait
TDD: Time Division Duplex
UL: Up Link
WiMAX: Worldwide Interoperability for Microwave Access

1. Project Overview

1.1 Introduction

Broadband wireless networks have been increasingly used for services such as data, video, voice, and internet traffic. Wireless communication provides users with tremendous convenience compared with communicating in a wired environment. However, the wireless data traffic is more likely to suffer from signal interference and attenuation; therefore, maintaining a high QoS (Quality of Service) is quite challenging for all wireless technologies. WiMAX (Worldwide Interoperability for Microwave Access) is a wireless digital communication system that can provide a whole metropolitan area with broadband wireless services. Furthermore, the mobile WiMAX supports not only fixed subscribers but also subscribers that are moving at vehicular speed, which further demonstrates the marvelous potential of WiMAX as next generation broadband wireless technology. ARQ (Automatic Repeat Request), due to its implementation simplicity, is commonly used to provide a reliable way to ensure that packets are successfully received at receivers and in sequence. HARQ (Hybrid Automatic Repeat Request) combining ARQ scheme and FEC (Forward Error Correction) functionality further improves the performance of wireless communication systems. HARQ/ARQ relies on feedbacks (ACK/NAK) to determine whether retransmissions are required. The WiMAX uses multi-channel stop-and-wait HARQ scheme. Each HARQ channel is independent of each other, and a packet can only be retransmitted by the HARQ channel that initially sends the packet. During a noisy channel condition, a successful transmission might take quite a long time to deliver because more retransmissions are needed. The proposed multiple-copy HARQ scheme sends multiple

copies of the same data burst on contiguous HARQ channels so that it dramatically reduces the time to successfully recover an erroneously received packet in a receiver's buffer. During noisy channel conditions, a time sensitive packet that needs more retransmissions can be retransmitted quicker and recovered faster as well.

This report is organized as follows: In the rest of the section 1, the proposed area of study, project requirements and academic contributions are addressed. In section 2, the background of the project is covered, including ARQ, HARQ, WiMAX, WiMAX HARQ and related works. In section 3, a summary of CS297 project is included. In section 4, the proposed multiple-copy HARQ scheme is explained in detail. In section 5, simulation settings and simulation results for both original WiMAX HARQ scheme and proposed multiple-copy HARQ scheme are included. In addition, performance comparisons of these two HARQ schemes are also provided. Finally, in section 6, conclusions are made.

1.2 Proposed Area of Study

Seven ARQ schemes, including three basic ARQ schemes, were studied and their throughput performances were analyzed. In addition, a guideline of designing an ARQ scheme with high throughput performance is provided, please refer to section 3. ARQ mechanisms control the way a transmitter transmits and retransmits data to a receiver. Generally speaking, an efficient ARQ mechanism should be able to deliver a great throughput performance; however, there are other issues that an ARQ mechanism should also consider in order to suit any particular need. For example, even though the stop-and-wait ARQ scheme has low throughput performance when propagation delay is too long, it is very suitable for a transmitter that only sends data to nearby receivers that only have small buffer.

HARQ makes use of both ARQ mechanism and FEC functionality, so the ARQ mechanism still plays a crucial role in designing a good HARQ scheme. WiMAX currently uses multi-channel stop-and-wait HARQ scheme which is simple and has good throughput performance; however, a data packet might take a long time to be successfully transmitted due to each HARQ channel is independent of each other. The WiMAX HARQ was studied, aiming to come up with a brand new mechanism that is compatible with the current WiMAX protocol and solves the issue of unsuccessfully transmitted data busts' long waiting time in the SS buffer.

1.3 Project Requirements

1.3.1 Project Scope

The purpose of this project, in the first phase, is to study and analyze existing ARQ schemes by using a network simulator to mimic ARQ schemes running in different channel conditions. As a result, a guideline of designing an ARQ scheme with high throughput performance is derived from analyzing simulation results. Moreover, the WiMAX HARQ that combines stop-and-wait ARQ scheme and Chase Combining functionality is studied. In the second phase, the multiple-copy HARQ scheme is proposed to reduce the waiting time that an erroneously received data burst has to spend in the SS buffer while the channel experiences high block error rate (BLER) conditions. A simulator is programmed to simulate both the original WiMAX HARQ scheme and the proposed multiple-copy HARQ scheme on a wireless channel with varying channel conditions. Both HARQ schemes' throughput performance and average waiting time of data bursts in the SS buffer are compared and analyzed.

1.3.2 Project Goals

The primary objective of the project is to propose a HARQ scheme that can be used in the current WiMAX protocol. The HARQ scheme currently used in the WiMAX is extremely simple; however, it lacks flexibility because HARQ channels are independent of each other. Retransmissions can only be done by the same sending channel. It would be desirable to have HARQ channels without aforementioned restriction so that HARQ channels could work together and quickly react to bad channel conditions. The proposed HARQ scheme should be compatible with the current WiMAX implementation and the fewer changes needed to be made, the better. The throughput performance of the original HARQ scheme is good enough in a good channel condition; therefore, it would be better that the proposed HARQ scheme remains the same as the original HARQ scheme to keep the design simple in a good channel condition and dynamically triggers the new mechanism when channel's condition becomes noisy. The way of deciding how the proposed HARQ scheme triggers or turns off the new mechanism should be efficient and match the channel's condition as closely as possible.

1.4 Academic Contributions

A multiple-copy HARQ scheme is proposed for WiMAX to reduce the long wait of erroneously received data packets in the receiver's buffer. It is most beneficial for data packets containing time sensitive information. Even though the throughput performance of the multiple-copy HARQ scheme is lower than that of the multi-channel stop-and-wait HARQ scheme, the multiple-copy HARQ scheme dramatically reduces packets' waiting time in the receiver's buffer when channel condition is noisy.

The proposed multiple-copy HARQ scheme can be further studied to further accurately estimate channel conditions so that the throughput performance could be even better, especially when a channel has high packet error rate and more packets need to be retransmitted. The multiple-copy HARQ scheme, using chase combining, can also be modified to support using incremental redundancy. Furthermore, with a simple modification as proposed in [27], the MIMO (Multiple Input Multiple Output) technology that has been proven to improve throughput performance can runs multiple-copy HARQ scheme in its multiple antennas separately.

2. Background

2.1 ARQ

In wireless network communications, ARQ schemes have long been used to enhance the reliability of packet-based data transmissions. Requesting needed retransmissions, ARQ scheme ensures that lost or erroneous packets are eventually received at the receiver side without errors and sent to the upper layer in sequence. The three basic ARQ protocols are stop-and-wait (SW), go-back-N (GBN), and selective repeat (SR) [12]. The SW protocol has the advantage of implementation simplicity; nevertheless, its inherently huge idle time (propagation delay) spent on waiting for an acknowledgement from a receiver for every single transmission hurts its throughput performance. The GBN protocol that continuously transmits data within the ARQ window has a more efficient data transmission result than the SW protocol. The performance of the GBN protocol degrades dramatically in both high packet error rate and long round-trip delay situations. The reason is that when receiving an erroneous packet, a receiver not only discards this erroneous packet but also discards all subsequent

packets even though they are successfully received. There is a chance that correctly received packets that were discarded might be corrupted during the retransmission and then trigger another retransmission. Lastly, the SR protocol states that a transmitter performs retransmissions only for erroneously received packets, so it is the most efficient protocol among these three. Theoretically, a receiver should have an infinite buffer so that a transmitter can keep sending data all the time while retransmitting erroneous ones. In practice, the SR protocol usually sends packets within a finite window-size buffer. The SR protocol's drawback is that the buffer requirement is higher and is more complicated to implement than the other two protocols. In addition to these three basic ARQ schemes, there are a number of variant ARQ schemes, as in [3], [6], [14], and [17], that are proposed to suit a variety of situations.

2.2 HARQ

Using wireless channels increases chances of suffering from interference of other channels using the same frequency or surrounding noises. A successful packet transmission sometimes requires several retransmissions if packet error rate becomes high. The performance of ARQ schemes suffers quickly because more retransmissions are required. In order to reduce the frequency of retransmissions, a system can adopt a FEC functionality to correct erroneously received packets. Combining ARQ scheme (with CRC) and FEC functionality to correct errors first and then detect uncorrectable errors for retransmission is called HARQ. In [21], a HARQ scheme adopts stop-and-wait ARQ scheme is a good example. A more general example was proposed in [22].

In general, there are three types of HARQ. In type I HARQ, the receiver simply discards erroneous packets after it fails to correct errors, and then sends NAK to the

transmitter to request a retransmission. There is no need to have extra buffer to store erroneous packets for type I HARQ. Fixed code rate is used for error correction so that the type I HARQ cannot effectively adapt to changing channel conditions. Using a code rate too high may cause too many retransmissions in high packet error rate conditions; on the other hand, using a code rate too low may cause too much redundancy in low packet error rate conditions. In previously mentioned situations, the throughput is degraded by either a high frequency of retransmission or a number of redundant data in transmissions. Choosing a suitable code rate is crucial for type I HARQ. Type I HARQ is best suited for a channel with a consistent level of SNR (signal to noise ratio). In type II HARQ, packets are coded with ARQ and FEC just as type I HARQ does except the receiver keeps erroneously received packets in the buffer in order to combine them with retransmitted packets. There are two major FEC categories of coding for type II HARQ: chase combining (CC) [9] and incremental redundancy (IR). In chase combining, the receiver combines received copies of the same packet to get diversity gain [10] [13]. All redundant bits for each retransmitted packet are the same as the first transmission; therefore, it is relatively simple but less adaptive to the channel condition because the decoder may just need a smaller number of redundant bits. The buffer needed is the number of coded symbols of one coded packet. In incremental redundancy, it adapts to changing channel conditions by retransmitting redundant bits gradually to a receiver until all redundancies are sent. At the beginning, a transmitter using IR sends coded packets with a small number of FEC redundant bits or even without any. If a retransmission is needed, only different redundant bits, derived from different puncturing patterns, are retransmitted depending on the base coding rate. The information data will not be

retransmitted except a receiver still cannot successfully decode the packet after a transmitter has sent all the redundant bits. This approach increases the receiver's coding gain one retransmission at a time; therefore, the IR scheme allows the system to adapt channel encoder rates to the channel quality. A bigger buffer is required to store all retransmitted data, including the first transmitted data as well. Type II HARQ needs a larger buffer size than type I HARQ, but it has higher performance in terms of throughput. The drawback of type II IR HARQ is that a receiver has to receive the first transmitted data in order to combine it with subsequently received redundant bits. To overcome this drawback, a type III HARQ was proposed [20]. Type III HARQ can be said as a special case of type two HARQ except each (re)transmitted packet is self-decodable. A receiver can either correctly decode information data by combining the first transmitted packet with retransmitted packets or just use only one of retransmitted packets.

2.3 WiMAX

WiMAX (Worldwide Interoperability for Microwave Access) [15] [16] is a broadband wireless communication technology also known as IEEE 802.16, WirelessMAN. WiMAX can provide broadband wireless services that cover a larger area than WiFi does due to its more efficient bandwidth usage, interference avoidance, and so forth. Therefore, it has been defined as a "last mile" broadband wireless access alternative to cable modem service, Digital Subscriber Line, and T1/E1 services.

One WiMAX base station (BS) can serve hundreds of WiMAX subscriber stations (SS), point-to-multipoint, and it also can reach many SSs among high-rise buildings in city, non-line-of-sight (using 2 GHz – 11 GHz channel). WiMAX has a tradeoff between having high bandwidth and reaching long distance, so a SS initializes

connection by choosing a BS with a stronger reception signal and, if necessary, performs handoff to a different BS that provides better reception signal when moving to a different location. When Time Division Duplex (TDD) mode is used, a BS schedules bandwidth for transmitting downlink data on downlink subframes (from BS to SS) and uplink data on uplink subframes (from SS to BS). WiMAX also supports Frequency Division Duplex (FDD) duplexing mode (best suited for voice service), but TDD can flexibly handle both symmetric and asymmetric traffic between BSs and SSs, which is more suitable for data traffic.

With intelligent technologies integrated in WiMAX such as scalable OFDMA (Orthogonal Frequency Division Multiple Access), all IP (Internet Protocol) architecture, MIMO (Multiple Input Multiple Output), smart antennas, etc, an always-connected environment could be a reality. Supporting security mechanism such as EAP (Extensible Authentication Protocol) and AES (Advanced Encryption Standard) helps WiMAX overcome security shortcomings that the WiFi has faced. Five service classes of QoS (Quality of Service) also enhance WiMAX to better serve different applications needs such as streaming audio or video, VoIP, Data transfer, and so forth. When the costs of WiMAX chipsets decrease and Mobile WiMAX solutions are fully delivered, WiMAX will completely change the notion of broadband services.

2.4 WiMAX HARQ

HARQ is supported in WiMAX that uses OFDMA (Orthogonal Frequency Division Multiple Access) physical layer. The HARQ scheme used in WiMAX is a basic stop-and-wait protocol. Using multiple HARQ channels can compensate the propagation delay of the stop-and-wait scheme, that is, one channel transmits data while others are

waiting for feedbacks. Therefore, multi-channel stop-and-wait HARQ using a small number of channels (e.g. 6) is an efficient and simple protocol that minimizes the memory required for HARQ and stalling [24]. HARQ channels are distinguished by a channel identifier (ACID). Data reordering at a SS is done by referring to PDU (Protocol Data Unit) sequence numbers that are enabled when HARQ operation is used. When Chase combining is used, the total buffer capability is determined by multiplying the number of bits that each HARQ channel is allowed to send with the number of channels used. HARQ DL MAP IE (Information Element) contains information about DL HARQ Chase sub-burst IE that specifies the location of HARQ sub-bursts, ACID, AI_SN, etc. By referring to MAP IEs, a SS can retrieve a correct data burst that is given to it. HARQ feedbacks (ACK and NAK) are sent by the SS after a fixed delay (synchronous feedbacks). To specify the start of a new transmission on each HARQ channel, one-bit HARQ identifier sequence number (AI_SN) is toggled on each successful transmission. [15] [16]

2.5 Related Work

In addition to three basic ARQ schemes, a number of other variant ARQ schemes have been proposed to suit different network situations. An efficient selective-repeat ARQ scheme was proposed for infrared links under a high bit error rate condition [4]. The efficient selective-repeat ARQ scheme is a window-based ARQ scheme operating with a finite receiver buffer and a finite range of sequence numbers. A novel selective-repeat stop-and-wait ARQ was proposed for operating in half-duplex channels [8]. Unlike the normal stop-and-wait ARQ scheme, the novel stop-and-wait ARQ scheme sends a set of packets that belongs to the same frame of an upper layer datagram and resends packets

negatively ACKed. The sender sends the next set of packets after all packets of the current set are successfully received. A block window retransmission ARQ scheme was proposed for next generation high speed IrDA (Infrared Data Association) links [5]. The block window ARQ scheme modifies the method of acknowledging packets and the way of retransmitting packets by using concept of blocks that contain several packets. Lastly, an optimum go-back-N ARQ strategy was proposed to send n copies of the same packet continuously [7] [19]. The optimum go-back-N ARQ dynamically increases and decreases n according to feedbacks by using an equation m = ceil[K/(n-1)] (K is about 5 to 15 times greater than n and m is used to decide when to decrease n). These four ARQ schemes are compared in Table 1, and last two rows are based on simulation results. The simulation results of these four ARQ schemes are covered in section 3.2.

	Efficient SR	**Optimum GBN**	**SR SW**	**Block Window reTX**
Type of ARQ	Selective Repeat	Go-Back-N	Combination of selective Repeat and Stop-and-Wait	Selective Repeat
ACK scheme	Following the first negative ACK, receiver also sends both positive and negative ACKs back.	Positive ACKs and the first negative ACK are sent back.	Only negative ACKs will be sent back.	Send both positive and negative ACKs back except ACKs for the first and last blocks.
Buffer usage	Store correctly received ARQ blocks which have sequence numbers within the window.	No buffer needed.	Store correctly received ARQ blocks which have sequence numbers within the window.	Store correctly received ARQ blocks which have sequence numbers within the window.

	Efficient SR	**Optimum GBN**	**SR SW**	**Block Window reTX**
Pros	1. Only retransmit erroneously received ARQ blocks.	1. Sending multiple copies of ARQ block reduces the retransmission need in high error rate environment. 2. Receiver does not need buffers to store out of order ARQ blocks: discards them.	1. Simple implementation. 2. Only retransmit erroneously received ARQ blocks.	1. Using one bit to acknowledge a block-window ARQ block is beneficial if the size of ACK bitmap is limited.
Cons	1. More complex to implement. 2. In high bit error rate environment, throughput may decrease because of always sending one window-size ARQ block instead of blocks inside the window.	1. Unnecessary retransmission of ARQ blocks. 2. Low through at high bit error rate environment, especially when propagation delay is long.	1. No new ARQ block can be sent before prior set of ARQ blocks are all successfully received.	1. More complex to implement. 2. Some unnecessary retransmissions have to occur.
Normalized throughput in high BER environment: BER = 10^{-4} (*)	**Second place (**53%)**	**Forth place (**22%)**	**First place (**100%)**	**Third place (**50%)**
Normalized throughput in low BER environment: BER = 10^{-5} (*)	**First place (**100%)**	**Forth place (**68%)**	**Second place (**94%)**	**Third place (**68%)**

(*) in the situation where propagation delay is short (0.002 ms).

(**) taking first place value as the base value.

Table 1. Comparison of four variant ARQ schemes

One famous example of using multiple-channel stop-the-wait is the ARPAnet (Advanced Research Projects Agency Network), which supported multiplexing of 8 logical channels over a single link, and ran stop-and-wait ARQ on each logical channel [18]. Using multiple-channel stop-and-wait HARQ with chase combining as a link adaptation HARQ technique has been widely used in wireless networks such as HSDPA (High-Speed Downlink Packet Access), refer to [2] for protocol description and simulation results. HSDPA uses HARQ channels to transmit data inflexibly as WiMAX, that is, each channel is independent of each other. On the other hand, the proposed multiple-copy HARQ scheme makes data transmissions more flexible and adaptive to channel conditions. Timing of N-channel stop-and-wait HARQ operation can be in fully asynchronous, partial asynchronous and synchronous mode, as discussed in [1]. The transmitter can (re)transmit data at any time in fully asynchronous mode. In partial asynchronous mode, retransmissions can only be done at i + nN frame intervals (i is the frame in which the first transmission takes place, n is a positive integer, and N is the feedback delay in frames). In synchronous mode, retransmissions can take place only at a fixed time interval. The proposed multiple-copy scheme is designed to work in synchronous mode: each HARQ channel receives feedback and transmits data at every fixed time interval. Multiple-channel stop-and-wait HARQ with chase combining can also be used with MIMO (Multiple Input Multiple Output) technology. As in [7], the authors proposed a new MIMO scheme to improve throughput by attaching each substream separate CRC (Cyclic Redundancy Check) and employing one HARQ entity with 3 processes in each transmit antenna. The concept of the proposed multiple-copy

HARQ scheme can be easily used with MIMO to not only improve throughput but also reduce waiting time of erroneously received data bursts in receivers' buffer.

3. Summary of CS297 Project --- Evaluation of ARQ Schemes

3.1 Simulation Topology and Settings

In this section, simulation results of seven different automatic repeat request (ARQ) schemes are examined. In order to mimic a metropolitan WiMAX network environment, half-duplex is chosen as a transmission mode and a short propagation delay is configured in the simulation. When changing the bit error rate from 10^{-3} to 10^{-8}, simulation results show that ARQ schemes which only need to retransmit erroneously received ARQ blocks have better overall throughput performance. Both evaluating and analyzing the ability of handling data retransmission efficiently are the primary goals of the project; therefore, a list of guidelines for designing an efficient ARQ scheme is provided according to simulation results. Being aware of network status and reacting to it effectively may be challenging when it comes to designing an efficient ARQ scheme. There is no one particular ARQ scheme that can be the best for all kinds of channel situations and network topologies. In this project, by mimicking a metropolitan WiMAX network environment in the simulation, it paves a road to further research on HARQ used in WiMAX.

The simulation tool used for all ARQ protocol simulations in this section is JSim [23]. The simulation topology consists of two nodes, a transmitter and a receiver, and a link that connects these two nodes, as depicted in Fig. 1. The network configuration includes link bandwidth, propagation delay, and bit error rate. The transmission is set to

be half-duplexing, which means that no simultaneous transmissions from both transmitter and receiver. The transmitter node and the receiver node send packets or acknowledgement to one another according to the ARQ protocol applied in the simulation. The link between two nodes has a predefined bandwidth at 10 Mbps and a predefined propagation delay at 0.002 ms (a distance of 600 meters away). The bit error rate in the simulation ranges from 10^{-3} to 10^{-8}. The packet payload size is 512 bytes and the acknowledgement packet payload size is 10 bytes, both of these values are fixed throughout the simulation. If there is a window size in use, the window size is 64 ARQ blocks. The major measurement of performance is throughput. Packets are considered successfully delivered when they are successfully received and sent to the upper layer in-sequence.

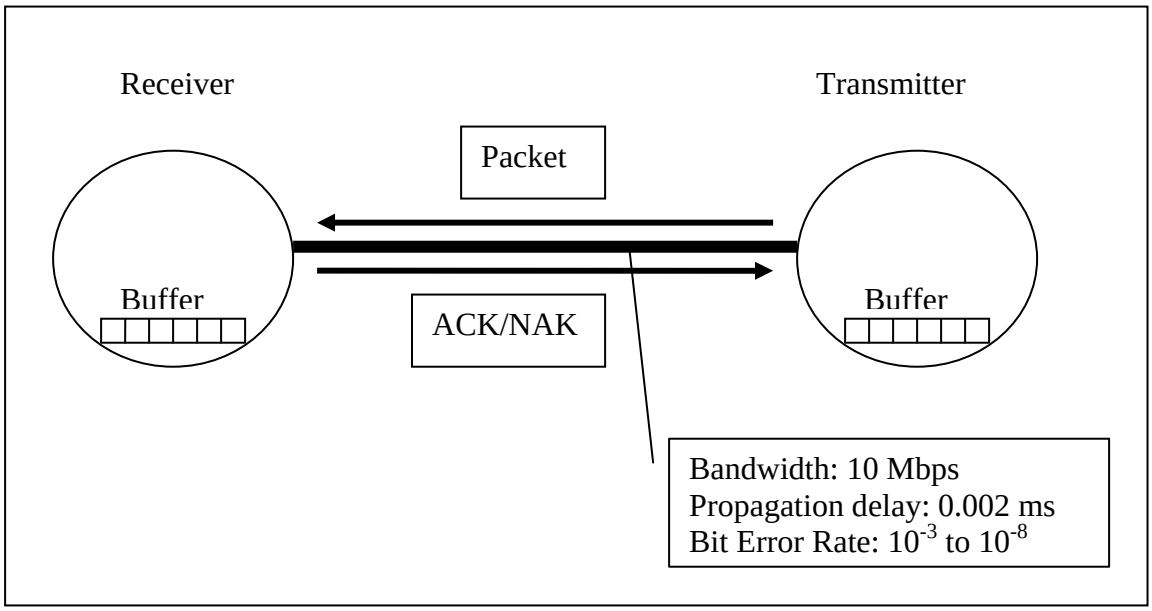

Figure 1. Network topology and configuration for ARQ simulation.

3.2 Simulation Results

In this section, four variant ARQ schemes and three basic ARQ schemes mentioned previously are compared.

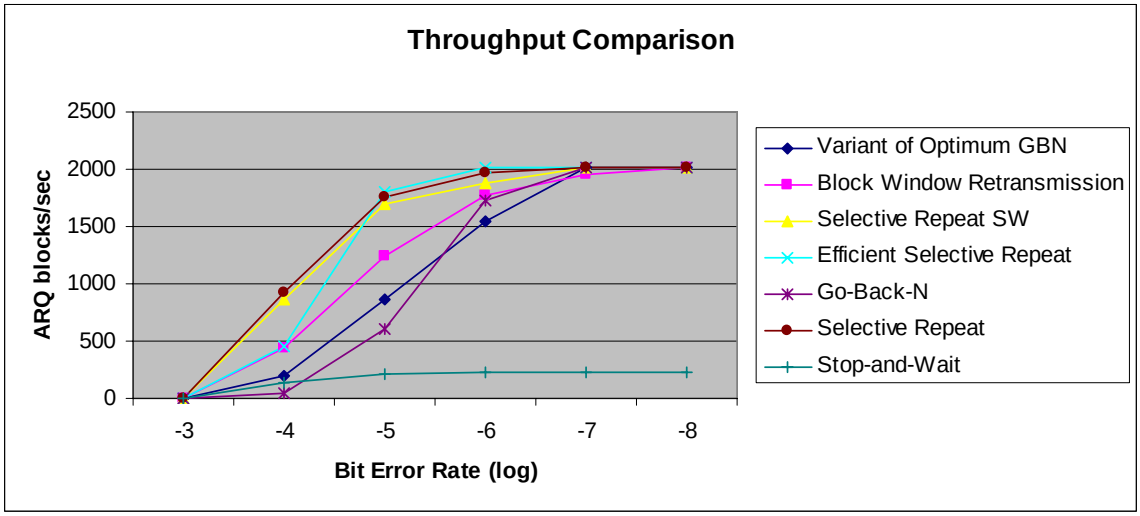

Figure 2. Throughput comparison of seven different ARQ schemes

The throughput is one of the crucial factors in determining whether an ARQ protocol is desirable or not. The throughput performance of seven different ARQ protocols operating in different bit error rates situations is shown in Fig. 2. Undoubtedly, the stop-and-wait ARQ has the lowest throughput due to long waiting time for each acknowledgement (long propagation delay). The selective Repeat scheme and its variants have relatively good throughput performance in high bit error rate situations because they follow one important concept: a transmitter only sends ARQ blocks that are erroneously received or missing. The throughput performance of the Go-Back-N scheme and its variant sit right in the middle because a number of unnecessary retransmissions take place when just one packet is incorrectly received or missing. Sending duplicate ARQ blocks, a variant of optimum GBN scheme, helps improve the throughput of GBN in high bit error rate situations.

By analyzing the simulation results, a list of suggested guidelines for designing an efficient ARQ protocol is as follows:

- Retransmitting only erroneous received ARQ blocks.

  As the mechanism used in selective repeat scheme and its variants, retransmission is only required for erroneously received ARQ blocks. Having fewer retransmissions saves propagation delay significantly in a half-duplexing environment.

- Dynamically sending more than one copy of an ARQ block in a high bit error rate situation.

  By comparing throughput performance between GBN scheme and variant of Optimum GBN scheme, dynamically increasing or decreasing number of ARQ block copies sent at a any moment can improve throughput performance in a high hit error rate situation.

- Sending more new ARQ blocks that are outside the ARQ window in a low bit error situation.

  As shown in Fig. 2, the efficient selective repeat scheme has higher throughput in low bit error rate situation because it sends another whole window-size ARQ blocks and lets the receiver decide whether to discard or accept them. If most of the ARQ blocks are correctly received, the receiver can accept more ARQ blocks immediately without having to wait for the next transmission opportunity.

- Selective repeat SW could be considered in a low bit error rate situation for simplicity reason.

  If propagation delay is short, the selective repeat SW scheme can be a good candidate to be used in a low bit error rate situation. It is simple, yet has good throughput performance as a regular selective repeat scheme. However, it is not like

the regular selective repeat scheme in which the buffer restriction prevents new ARQ blocks from being transmitted.

- Block window retransmission concept can be considered when the number of available bits in a bitmap for acknowledging ARQ block is too small.

  Block window retransmission scheme treats a block-window-size ARQ block as a unit to acknowledge ARQ blocks. This means that, for example, using 10 bits, 50 ARQ blocks can be acknowledged if there is an average number of 5 ARQ blocks in one block window. There is no need to use 50 bits to individually acknowledge each ARQ block.

4. Design of CS 298 Project --- Multiple-copy HARQ

4.1 The BS-side Implementation

4.1.1 Parameter MC and Parameter ICN

Each HARQ channel of the original multi-channel stop-and-wait HARQ scheme operates independently of each other. A data burst that is unsuccessfully received can only be retransmitted by its initial sending channel. The proposed multiple-copy HARQ scheme eliminates aforementioned restriction by adding two parameters in the DL HARQ Chase sub-burst IE that is contained in the downlink map in the same downlink subframe as data bursts [16]. One of the parameters is MC (Multiple Copy) that specifies whether the data bust is the first transmission or not – “false” represents first transmissions and “true” represents retransmissions. The other one is ICN (Initial Channel Number) that specifies what the initial sending channel of the data burst is. ICN can be skipped if MC = false (SS just refers to ACID for the channel number). Using parameters MC and ICN, a

SS can correctly choose erroneous data bursts to combine with retransmitted data bursts that are received on HARQ channels different from their initial sending channels.

4.1.2 Information Stored in the BS

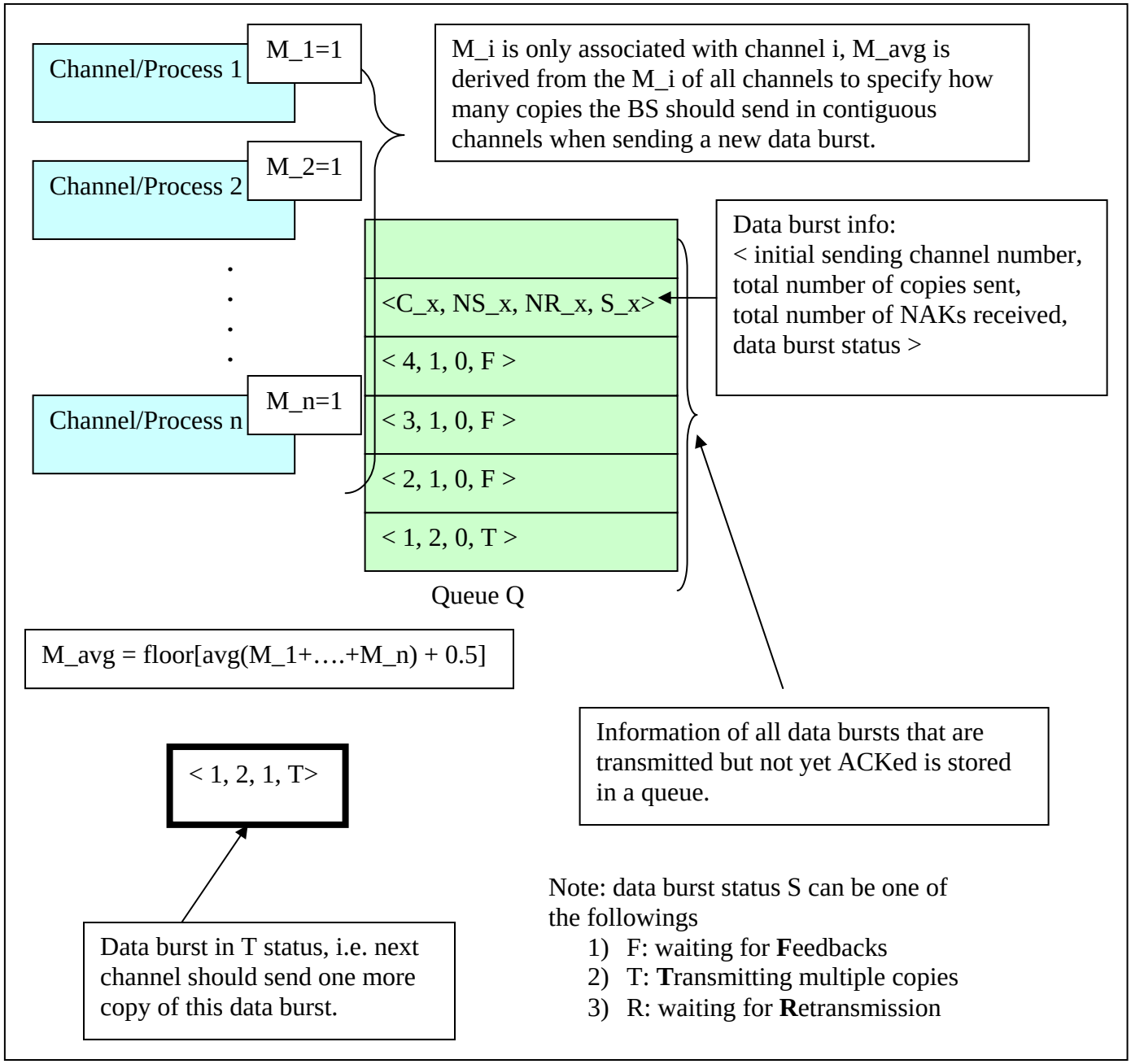

Figure 3. Information of unACKed data bursts stored in the BS

As shown in Fig. 3, the BS stores information of all unACKed data bursts in a queue Q and deletes the data bursts' information when receiving corresponding ACKs. The maximum size of the queue is the number of HARQ channels used since all channels of the multiple-copy HARQ scheme are still using basic stop-and-wait scheme, i.e., a HARQ channel will not send any data burst before receiving a feedback.

4.1.2 Data Burst Information (C_x, NS_x, NR_x, S_x)

Because a unACKed data burst can be retransmitted on different HARQ channels, each unACKed data burst is associated with a set of four parameters of its own so that the BS can choose a correct data burst to retransmit. These four parameters are listed and explained in Table 2. Parameter C_x for a data burst is set and fixed after the data burst is transmitted for the first time. The BS refers to C_x to set ICN when it retransmits a data burst D_x. The NS_x, NR_x, and S_x are changed according to received feedbacks. The BS refers to NS_x to set MC when it (re)transmits a data burst D_x. How these three parameters are changed is described as follows (Supposing a data burst D_x has parameters NS_x, NR_x, and S_x):

i. The NS_x is increased by one when the D_x is either transmitted for the first time or retransmitted.

ii. The NR_x is increased by one when a channel receives a negative feedback (NAK) of the D_x.

iii. The data burst status S_x is set to T when sending multiple copies is needed; the S_x is set to F when the D_x has not received all the feedbacks (NS_x > NR_x) after sending required multiple copies (or

sending 1 copy if M_avg = 1); the S_x is set to R when the D_x has received all the feedbacks but still has not been successfully transmitted (NS_x = NR_x).

C_x	Initial sending channel's channel number of the data burst (fixed after being set)
NS_x	Number of copies of the data burst that have been transmitted (initially is 0; changeable)
NR_x	Number of NAKs of the data burst that have been received (initially is 0; changeable)
S_x	Data burst status. (T, **T**ransmitting multiple copies; F, waiting for **F**eedback; R, waiting for **R**etransmission; changeable)

Table 2. Parameters for a unACKed data burst.

4.1.3 Parameter M_i and Parameter M_avg

The proposed multiple-copy HARQ scheme sends the same data burst through contiguous HARQ channels when channel conditions become noisy. In order to decide how many copies to send, two parameters are introduced: M_i and M_avg. The M_avg is calculated from all M_i's as shown in equation (1). Each HARQ channel_i is associated with a parameter M_i of its own. M_i is initially set to 1 for each channel; therefore, M_avg is also equal to 1 initially, which means that the subsequent channel of a channel, say channel 1, does not need to send one more copy of the same data burst that was sent by channel 1. Each HARQ channel's M_i is adjusted according to received feedbacks, ACK and NAK. How the M_i is adjusted is described as follows:

i. When a BS receives a NAK of a data bust, D_x, the M_i of the D_x's initial sending channel is adjusted to NR_x + 1.

ii. When a BS receives an ACK of the first copy of a data burst, D_x, sent by channel i, the M_i is adjusted to NR_x + 1. In this case, the M_i always

becomes to 1 since NR_x is always equal to 0 when the feedback of the D_i's first copy is an ACK.

$$M\_avg = Floor[Avg(M\_1 + M\_2 + \ldots + M\_N) + 0.5] \quad (1)$$

Note: N is the number of HARQ channels.

When a HARQ channel is eligible to send a new data burst, the channel checks M_avg to decide how to send data bursts and how to set the data bursts' status afterward. The following describes how.

i. If M_avg is greater than 1, the status of a new data burst D_x sent by current channel is set to T (The BS has not sent enough copies of the data burst D_x after current transmission).

ii. If M_avg is 1 or the parameter NS_x of the data burst D_x sent by current channel is equal to M_avg, the status of the data burst is set to F (The BS has sent enough copies of the data burst D_x after current transmission).

If the status of a data burst, D_x, is set to T, the subsequent HARQ channel should send another copy of D_x. After sending a copy of the data burst, D_x, on the subsequent channel, the BS checks M_avg to decide the status of the D_x again. The same procedure repeats until the data burst's status is set to F. Whenever a M_i is adjusted, the M_avg is also updated before being used by the BS to make decisions on transmitting data bursts and updating parameters.

The operation of the proposed multiple-copy HARQ scheme that runs at BS side is described in Fig. 5. Data busts with different statuses have different priorities of being sent. Data bursts with status T have the highest priority of being sent; data bursts with

status R have second priority of being sent; data bursts have status F have the lowest priority of being sent. The BS sends data bursts with status F only when there is no data burst with status T, no data burst with status R, and an outstanding data burst of the current channel existing (in this case, a new data burst can not be transmitted).

4.1.4 Operation of HARQ Channel i in the BS

Generally speaking, the BS firstly changes information of unACKed data bursts and updates M_i, and M_avg after receiving a feedback. Secondly, if there is a data burst with status T, the BS sends one more copy of the data burst with status T and updates information of this data burst. Thirdly, if there is no data burst with status T, the BS sends one more copy of data burst with status R, if any, and updates information of this data burst. Fourthly, if there is no data bust with status R, the BS can send a new data burst if the following tow conditions are satisfied. First condition is that there should be no unACKed data burst initially sent by the current channel; otherwise, the BS sends a unACKed data burst whose information entry is the first element stored in the queue. Second condition is that the SS buffer capability has not been reached; otherwise, the BS does not send any data burst. The flow chart of the proposed multiple-copy HARQ scheme's operation on channel i in the BS is illustrated in Fig. 4, and the corresponding description is shown in Fig. 5.

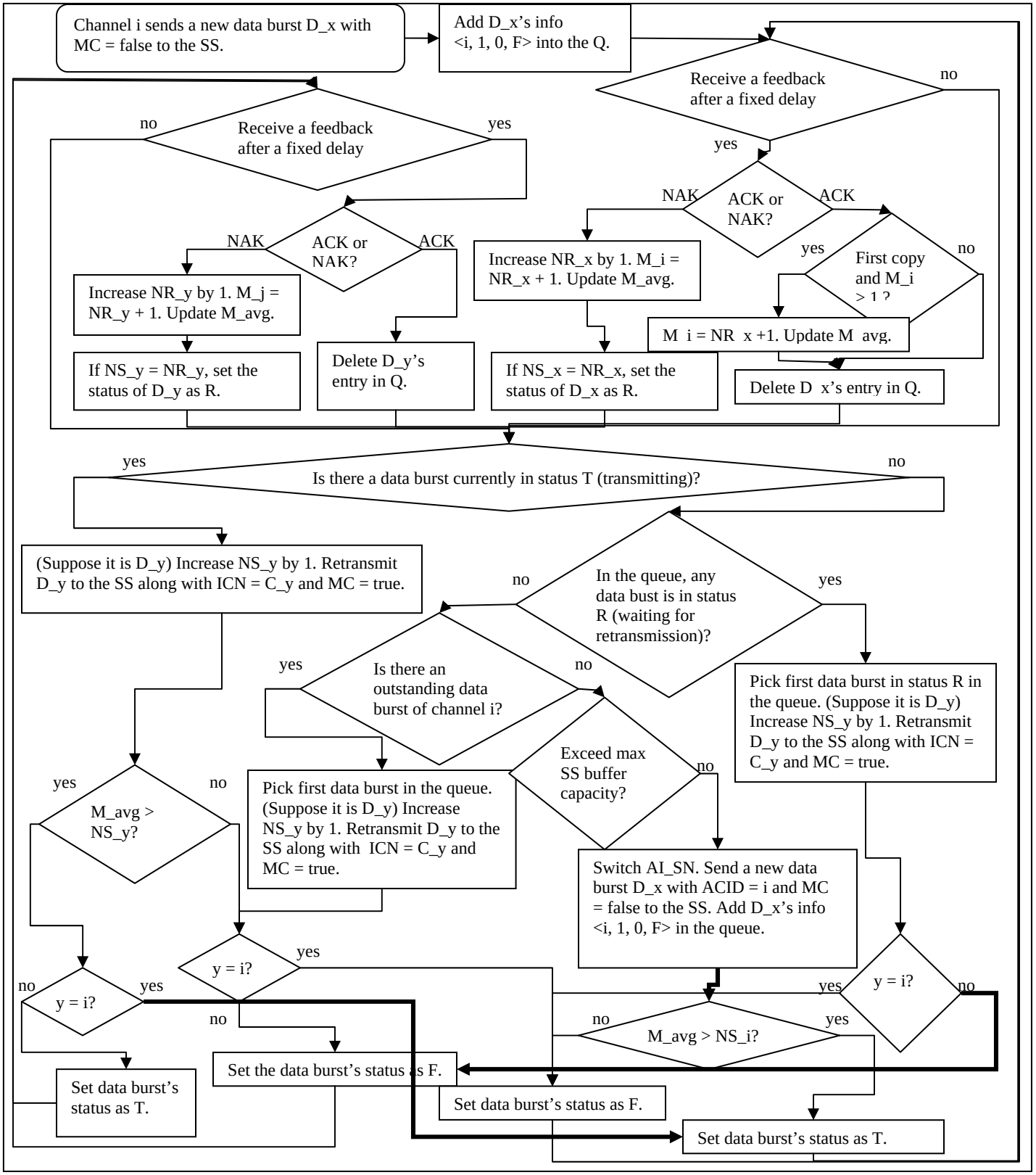

Figure 4. Flow chart of multiple-copy HARQ operation – BS sends data bursts on channel i.

```
(I).Initially for channel i := 0 to N { // N is the number of channels
    (1)Send a new data burst D_x to the SS along with MC = false and wait for the feedback
                                                        // first copy of the data burst
    (2).Store entry of the data burst in the queue} end of (I)
(II).For channel i := 0 to N {
     (1)If channel i receives a feedback of a data burst D_x from the SS
          (a). Data burst D_x is initially sent by channel i
               i. If receiving an ACK
                    1.If D_x is the first copy and M_i is not equal to 1
                         a.Set M_i = NR_x + 1 //set M_i back to 1
                         b. Update M_avg = floor[avg(M_1+M_2+….+M_N) + 0.5]
                    2.Delete D_x's entry from the queue
               ii.Else // receiving a NAK
                    1.NR_i++, M_i = NR_x + 1
                    2.Update M_avg = floor[avg(M_1+M_2+….+M_N) + 0.5]
                    3.If all feedbacks for D_x are NAKs // NS_x = NR_x
                         a.set S_i as R
          (b). Data burst D_x is not initially sent by channel i
               i.If receiving an ACK
                    1.Delete D_x's entry from the queue
               ii.Else // receiving a NAK
                    1.NR_x++, M_j = NR_x + 1
                    2.Update M_avg = floor[avg(M_1+M_2+….+M_N) + 0.5]
                    3.If all feedbacks for D_x are NAKs // NS_x = NR_x
                         a.set S_x as R
   (2).If there is a data burst with status T (supposed it is D_y)
       (a).Retransmit one more copy of D_y to the SS along with ICN = C_y and MC = true
       (b).NS_y++
       (c).If M_avg > NS_y, set S_y = T; otherwise, set S_y = F
   Else // no data burst in status T
       (3).If there any data burst in status R
            (a).Retransmit the first data burst (suppose it is D_y) that is in status R in the
                queue along with ICN = C_y and MC = true
            (b).NS_y++
            Else // no data burst in status R
                 (4).If there is any data burst in status F initially sent by channel i in the queue
                      (a).Retransmit the first data burst (suppose it is D_y) in the queue
                          along with ICN = y and MC = true
                       (b).NS_y++
                  Else
                      (5).If maximum SS buffer has not reached
                        (a).Send a new data burst D_x to the SS along with  MC = false
                        (b).Adding data burst's entry in the queue
                        (c).If M_avg > NS_x, set S_x as T; otherwise, set S_x as F } end of (II)
```

Figure 5. Description of multiple-copy HARQ operation in the BS

4.2 The SS-side Implementation

4.2.1 Operation of HARQ Channel i in the SS

When receiving a data burst, the SS first looks at the data burst's parameter MC. MC indicates whether the data burst is the first copy or not. If the data burst is the first copy, MC = false, the one-bit AI_SN is checked to see if it is a new transmission. The SS ignores data bursts sent from the same channel with MC = false and also with the same AI_SN. If the AI_SN is different and MC = false, the SS starts to decode the data burst. The SS sends an ACK back to the BS if the decoding is successful, otherwise it sends a NAK back to the BS and stores the erroneously received data burst in the buffer to combine with retransmissions. Successfully decoded data bursts are sent to the upper layer in sequence. When the SS receives a retransmission, MC = true, the ICN of the data burst is checked to identify the initial sending channel of the data burst so that the SS can correctly combine the retransmitted data burst with the one that is erroneously received previously. If there is no erroneously received data burst that has the ACID equal to the retransmitted data burst's ICN, the erroneously received data burst must have been recovered by combining with previously retransmitted data bursts, so the SS simply ignores the retransmitted data burst. If a data burst with error is successfully decoded after combining, the SS sends the data burst to the upper layer in sequence and sends back an ACK to the BS. The description of the operation is in Fig. 6, and the flow chart of the operation is illustrated in Fig. 7.

(I).When the SS receives a data burst D_x // receiving one by one
(1)If receiving the first copy of D_x// MC = false.
(a).If same AI_SN for sending channel i// old data burst
i.Ignore D_x
(b).Else // new data burst
i.Switch AI_SN for channel i
ii.If successfully decode D_x
1.Send an ACK back to the BS at a pre-scheduled time
2.If D_x is in-sequence, then send to the upper layer; otherwise store D_x
iii.Else// not successfully
1.Send a NAK back to the BS at a pre-scheduled time
2.Store D_x for later combining
(2).Else// second or later copies
(a).If D_x has already been successfully decoded with previous copies
i.Ignore D_x
(b).Else
i.Combine D_x with the previously stored data burst
ii.If successfully decode D_x after combining
1.Send an ACK back to the BS at a pre-scheduled time
2.Send in-sequence data bursts to the upper layer, otherwise store D_x
iii.Else// not successfully
1.Send a NAK back to the BS at a pre-scheduled time
2.Store D_x for later combining

Figure 6. Description of multiple-copy HARQ operation in the SS

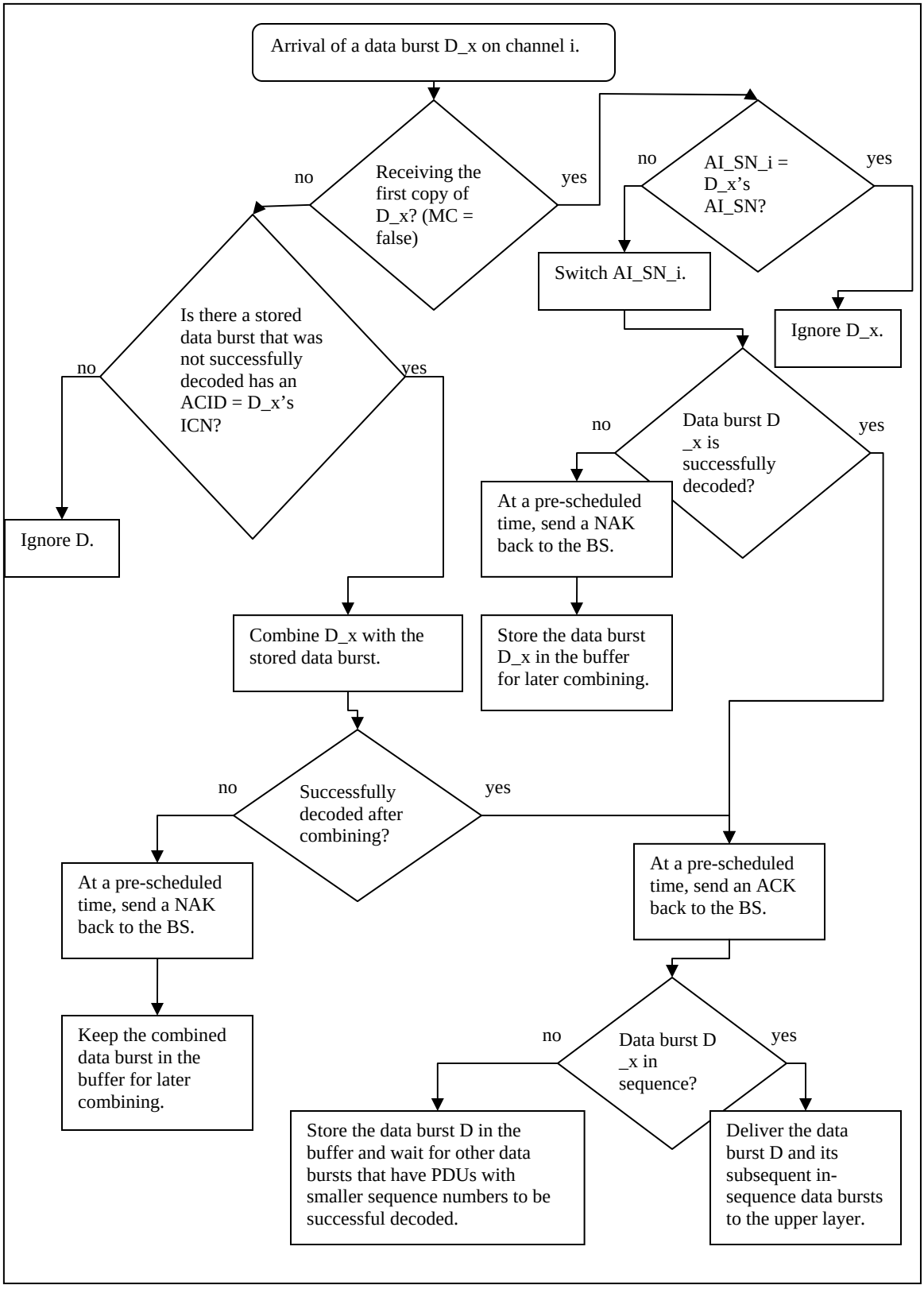

Figure 7. Flow chart of HARQ operation – SS receives data bursts from channel i.

4.3 Formal Analysis of the Multiple-Copy HARQ Scheme

4.3.1 Correctness Analysis

The most important thing in an ARQ/HARQ mechanism is to transmit data correctly to receivers and receivers can send data to the upper layer in sequence. We do the correctness analysis by using mathematical induction.

Firstly, we prove that the BS transmits data bursts correctly and in sequence, using N channels. Secondly, we prove that the SS sends data bursts to the upper layer in sequence, using N channels.

1.) The BS sends new data bursts and resends erroneously received data bursts correctly so that the data bursts sent are in sequence.

1.1) There is only one HARQ channel:

1.1.1) The BS sends a data burst and receives an ACK. The BS sends a new data burst, so data bursts are sent in sequence.

1.1.2) The BS sends a data burst and receives a NAK. The status of the NAKed data becomes R, and it is recent again. After receiving an ACK, the BS then sends a new data burst as in 1.1.1), so data bursts are sent in sequence.

1.2) Suppose that the BS can send data bursts in sequence when using N channels

1.3) Using N + 1 HARQ channels can send data bursts in sequence as well

1.3.1) Initially, the BS sends a new data burst on (N + 1)th channel, then wait for the feedbacks.

1.3.2) Using N channels preserves the correctness as 1.2) supposes, when (N + 1)th channel receives a feedback, the situations can be one of the followings:

1.3.2.1) Receiving a feedback for a data burst that was initially sent by itself

After receiving an ACK/NAK: (N + 1)th channel can: 1. send one more copy of the data burst sent by Nth channel; 2. send one copy of the oldest waiting-for-retransmission data burst; 3. send one copy of the oldest unACKed data burst; 4. send a new data bust; 5. do not send any because of buffer limitation. In situation 1, (N + 1)th channel does not send a new data burst, so data bursts sent are still correct and in sequence as the situation where only N

channels exist. In situation 2, sending one more copy of a unACKed data burst only increases the chance of a successful decoding. Data bursts sent are still correct and in sequence. In situation 3, the same reason as in situation 2. In situation 4, only when no waiting-for-retransmission data bursts exist, (N + 1)th channel sends a new data burst. So, data bursts sent are still correct and in sequence when there are N + 1 channels existing. In situation 5, not sending data burst has no effect.

1.3.2.2) Receiving a feedback for a data burst that was initially sent by other channel

After receiving an ACK/NAK: the only difference from 1.3.2.1) is that the BS updates other channel's M, so data burst are still sent correctly and in sequence when there are N + 1 channels existing.

2.) The SS stores out-of-sequence data bursts and waits for those with smaller sequence number to be successfully recovered so that it can send data bursts to the upper layer in sequence.

2.1) There is only one HARQ channel:

2.1.1) The SS successfully decodes a data burst and sends an ACK back to the BS. Then the SS sends the data burst to the upper layer. So data bursts are sent to the upper layer in sequence.

2.1.2) The SS does not successfully decode the data burst and sends a NAK back to the BS. Then the SS stores the data burst in the buffer. The BS resends the same data burst to the SS to combine with the stored data burst. Abovementioned procedure repeats until the data burst is successfully decoded, so data bursts are sent to the upper layer in sequence.

2.2) Suppose that the SS can deliver data bursts to the upper layer in sequence using N channels

2.3) Using N + 1 HARQ channels can deliver data bursts to the upper layer in sequence as well

2.3.1) Initially, the SS receives a new data burst on (N + 1)th channel. If the data burst is successfully decoded, the SS sends the data burst to the upper layer if

there is no erroneously received data burst with smaller sequence numbers, or it stores the data burst and waits for those erroneously received data bursts to be recovered. If the data burst is not successfully decoded, the SS stored the data burst for later combining. So data bursts are sent to the upper layer in sequence.

2.3.2) Using N channels preserves the correctness as 2.2) supposes, when (N + 1)th channel receives a data bust, the situations can be one of the followings:

2.3.2.1) Receiving the first copy of a data burst

If the data burst is successfully decoded, the SS stores it if it is not in sequence; otherwise, the SS sends it to the upper layer. If the data burst is not successfully decoded, the SS stores it. So data bursts are still sent to the upper layer in sequence.

2.3.2.2) Receiving any following copies of a data burst

The only difference from 2.3.2.1) is that the SS combines it with the same previously received bad data burst to see if the data burst can be successfully decoded after combining. If it can, the SS sends in-sequence data bursts to the upper; otherwise, the SS stores it and asks for retransmission again. So data bursts are still sent to the upper layer in sequence.

3.) Knowing from 1.) and 2.), by mathematical induction, data bursts can be received in sequence using N HARQ channels.

4.3.2 Time Analysis of the BS Operation

The timing of sending data bursts on the BS side is the major difference between the original HARQ scheme and the multiple-copy HARQ scheme. Suppose that a BS can give a SS as many as N HARQ channels to compensate the delay. Suppose that the number of retransmission times is T for chase combining scheme to successfully correct errors. Since retransmissions of a data burst can only be done by the same HARQ channel, the original HARQ scheme needs T*N + 1 frames (the 1 means the frame in which the BS sends the last needed copy of data burst) to transmit all needed data bursts to successfully deliver a correct data burst.

On the other hand, the Multiple-copy HARQ can estimate how many retransmissions may be needed so that the BS transmits copies of the same data burst through contiguous HARQ channels. Suppose that the BS estimates S copies are needed. The actually required copies are 1 + T.

If S is greater or equal to 1 + T (overestimated), there are two situations may happens:

1) If the number of given channels, n, is equal or greater than 1 + T. The time required to transmit all needed data bursts to successfully deliver a correct data burst is as little as (1+T) frames.

2) If the number of given channels, n, is less than 1 + T. The time required to transmit all needed data bursts to successfully deliver a correct data burst is floor[(1+T)/n]*N + remainder of (1+T)/n.

If S is smaller than 1 + T (underestimated), there are also two situations may happens:

1) If the number of given channels, n, is equal or greater than S. The minimum time required is S + (1+T-S)*N + 1 frames (the 1 means the frame in which the BS sends the last needed copy of data burst)

2) If the number of given channels, n, is less than S. The minimum time required is floor[S/n]*N + (remainder of S/n) + (1+T-S)*N + 1.

A summary of the time analysis of both HARQ schemes is shown in Table 3. (Unit of time is frame).

Conditions / HARQ Scheme	1+T	S >= 1 + T (overestimated)	S < 1 + T (underestimated)
Original WiMAX HARQ	T*N + 1	N/A	N/A
Mulitple-copy HARQ	T*N + 1 (Note 1)	(1+T) or floor[(1+T)/n]*N + remainder of (1+T)/n	S + (1+T-S)*N + 1 or floor[S/n]*N + (remainder of S/n) + (1+T-S)*N + 1 (Note 2)

Note 1. The mechanism of sending multiple copies has not been triggered.
Note 2. Minimum time required

Table 3. A summary of time analysis of both HARQ schemes

4.3.3 Space Analysis of the BS Operation

In addition to parameters that are required in current WiMAX HARQ implementation such as ACID and AI_SN, the proposed multiple-copy HARQ requires additional parameters that are either stored in the BS or transmitted with data bursts to accomplish the task of sending multiple copies correctly.

A set of four parameters <C_x, NS_x, NR_x, S_x> is associated with a unACKed data burst. All sets of four parameters are stored in a queue Q in the BS. Space required for each parameter is list as follows.

- C_x: ceil[$\log_2 N$] bits (N is the number of channels)
- NS_x: ceil[$\log_2(R+1)$] bits (R is the maximum allowable retransmission time, +1 is for the first transmission)

- NR_x:ceil[$\log_2$(R+1)] bits (R is the maximum allowable retransmission time, +1 is for the first transmission)
- S_x: 2 bits (00 for status T, 01 for status F, 10 for status R, 11 is reserved)

A queue Q stores all sets of four parameters for unACKed data bursts. There is only one unACKed data burst for one HARQ channel because each HARQ channel uses stop-and-wait scheme. The queue Q is stored in the BS. The following is the upper bound of the queue Q.

- Queue Q: N*(size of <C_x, NS_x, NR_x, S_x>) bits (N is the number of channels)

The parameters MC and ICN are sent in the DL Map within the same DL subframe where the associated data burst locates.

- MC: 1 bit (1 for true; 0 for false)
- ICN: ceil[$\log_2$N] bits (N is the number of channels)

One parameter M_i is associated with one HARQ channel_i, which means that N HARQ channels need N*M_i; there is only one M_avg that derived from all M_i. All the M_i and the M_avg are stored in the BS.

- All M_i: N*ceil[$\log_2$(R+1)] bits (N is the number of HARQ channels, R is the maximum allowable retransmission time, +1 is for the first transmission)
- M_avg: ceil[$\log_2$(R+1)] bits (R is the maximum allowable retransmission time, +1 is for the first transmission)

5. Performance Evaluations of Multiple-Copy HARQ

5.1 Synchronous DL and UL

The multiple-copy HARQ scheme and the original stop-and-wait HAQR scheme are implemented by using Java. The timings of sending and receiving data bursts are carefully organized so that a synchronous operation for both UL (Up Link) feedbacks and DL (Down Link) data transmissions can be accurately simulated [2]. As shown in Fig. 8, operations of one data burst transmitted to the SS (in downlink subframe) and one feedback received from the SS (in uplink subframe) are done during one frame, and each data bust's feedback is received after a fixed delay (there are 5 frames delay in Fig. 8) . The feedback delay is the only delay we consider in the simulation.

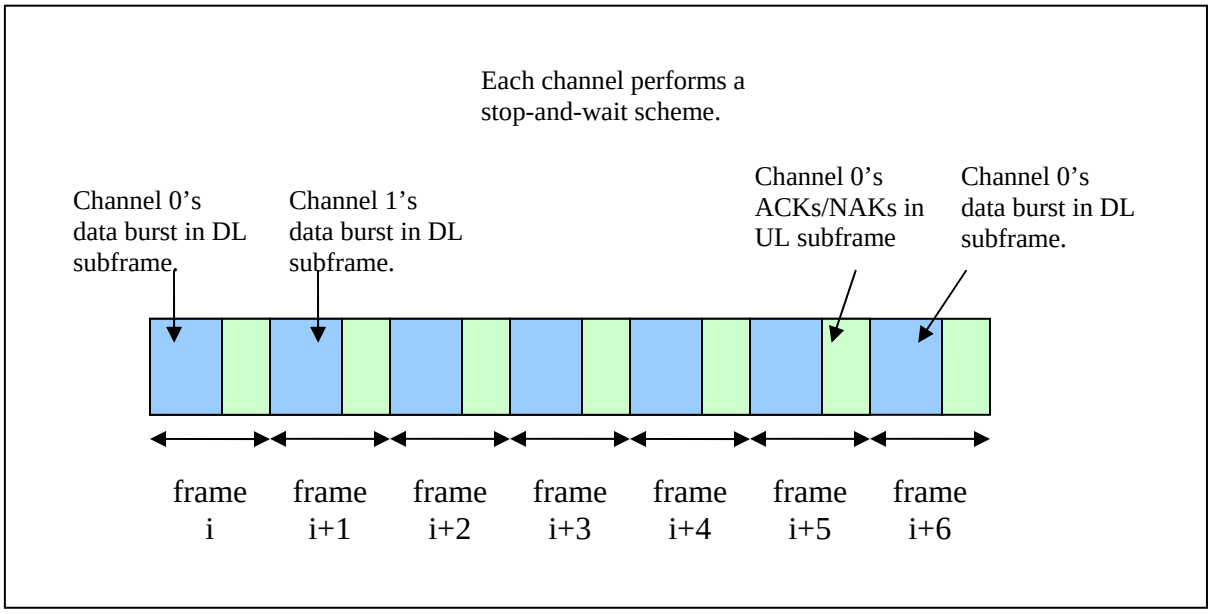

Figure 8. HARQ operation with six HARQ channels to compensate five frames feedback delay

The number of HARQ channels given to the SS is decided by the BS. If the BS has more available resources, it can give the SS enough HARQ channels to compensate the delay caused by using simple stop-and-wait ARQ scheme. Sometimes the SS may get

less HARQ channels than it needs to keep the pie full. Generally speaking, the delay is composed of propagation delay, SS processing time, feedback delay, and BS processing time [2].

5.2 Simulation Topology and Models

The simulation demonstrates one HARQ operation running between one BS and one SS. The uplink feedback channel is assumed to be error-free, and the BS is assumed to have a full traffic load for the SS. Both original multi-channel stop-and-wait HARQ scheme and proposed multiple-copy HARQ scheme are simulated. The simulation topology is illustrated in Fig. 9.

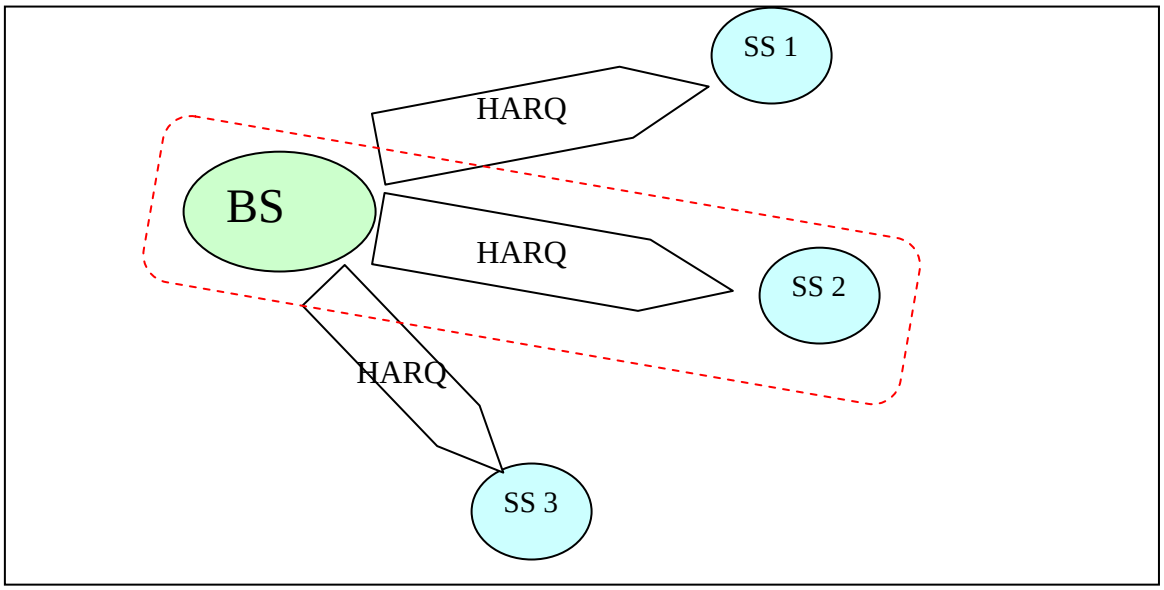

Figure 9. Simulation topology of HARQ schemes

5.3 Chase Combining

The type of HARQ considered in this project is the Chase Combining. The erroneously received data bursts are stored in the SS buffer for later combining with

retransmissions. The size of the data busts are the same for both first transmission and retransmissions.

Reference [11] mentions when the Chase Combining HARQ is applied for a simulation, a valid approach is to simply add the Es/No (symbol energy to noise ratio) values of each individual packet to get a resulting or combined Es/No after the combining. Reference [26] says that in a low BLER (Block Error Rate) situation, a successful transmission would have happened after one retransmission. In our simulation, we supposed that one retransmission is required in a normal channel condition (10% BLER [24]) as suggested in [26]. As for relatively higher BLER (> 60% BLER) channel conditions during noise bursts, the retransmission time required for the chase combining is reasonably assumed. We assume that the maximum number of retransmissions for a successful data burst transmission is 4 times. When the chase combining is used, the number of required retransmissions should be lower than simply discarding packets and asking for a retransmission. Therefore, a reasonably defined retransmission time required for a successful transmission under different BLER channel conditions is shown in Table 4.

Retransmissions required for chase combining	**BLER**
1 time	10% BLER
80% 1 time, 20% 2 times	60% BLER
75% 4 times, 25% 3 times	90% BLER

Table 4. Chase Combining modeling

4.4 Noise Burst Conditions

Noise bursts make channel condition of wireless communications unpredictable so that the AMC (Adaptive Modulation and Coding) scheme chosen by a BS cannot always achieve a targeted low BLER, causing a higher BLER for data transmissions. To simulate the noise burst situations, three factors are considered: one is the occurrence rate of noise bursts, NO; another one is the duration of a noise burst (in frames), ND; and the last one is the BLER during noise bursts. The occurrence rate of noise bursts specifies how often a noise burst can happen, and the noise burst duration specifies how long a noise burst is maintained. Different combinations of these three factors can result in different overall block error rates, as shown in equation (2). These three factors can be adjusted to analyze their impacts on the performance of HARQ schemes. Four combinations of the three factors used in the simulation are listed in Table 5. During a noise burst condition, for simplicity, the BLER stays the same.

$$\text{Overall BLER} = (1-NO*ND)*BLERn + (NO*ND)*BLERb \quad (2)$$

Note: BLERn means the BLER during the normal channel condition.
BLERb means the BLER during noise bursts.

Occurrence Rate of Noise Bursts	**Duration of a Noise Burst**	**BLER During Noise Bursts**	**Overall BLER**
1/1000	100	60%	15%
1/1000	100	90%	18%
5/1000	100	60%	35%
5/1000	100	90%	50%

Table 5. Noise burst modeling

5.5 Simulation Settings

Simulation settings		**note**
Transmission Timing	Synchronous	
HARQ type	Chase Combing	
Frame Size	5	ms
DL Peak Data Rate	64	Mbps (DL/UP = 1)
Number of SS	100	
Duplexing Mode	TDD	Time Division Duplex
Number of Channels	4, 6, 8, 12, 16	
Simulation Duration	100000	frames
Normal Channel Condition	10%	BLER
Channel Condition During Noise Bursts	60% and 90%	BLER
Noise Burst Occurrence Rate	1/1000 and 5/1000	
Retransmission Time	1, 2, 3, or 4	Also refer to Table 4.
Data Burst Size	1536	bits
Buffer Size	4 to 19	Size of data burst
Noise Burst Duration	100	frames

Table 6. Simulation settings

5.6 Simulation Criteria

1) Throughput

The system throughput is defined as the total bits of successfully received data bursts per second. Each frame length is fixed as 5 ms and one frame contains one data burst that has 1536 bits for a particular SS.

2) Average Data Bursts Waiting Time in the SS Buffer

A successfully transmitted data burst has to wait for other unsuccessfully transmitted data bursts that have PDUs with smaller sequence number in order to be delivered to the upper layer in sequence. The data burst waiting time in the SS buffer is

defined as the time it takes for a data burst from being received by the SS to the moment the SS sends the data burst to the upper layer.

3) Maximum Buffer Occupancy

Data bursts are buffered if they are erroneously received or out-of-sequence. Buffer occupancy will grow bigger and bigger if more out-of-sequence data bursts are waiting for a data burst with the smallest PDU sequence number to be successfully decoded. The maximum buffer occupancy is the biggest value of how many data bursts buffered in the SS throughout the simulation.

4.7 Simulation Results

In this section we present simulation results of the proposed multiple-copy HARQ scheme and the original HARQ scheme. The principal comparisons of these two schemes are throughput (kbit/s) performance and average waiting time (ms) performance. Performance evaluations are done under four different channel conditions. For the purpose of easier referrals, each channel condition used in the simulation is assigned an upper-case letter to it. Channel conditions and their associated letters are listed as follows. (Noise burst duration is 100 frames for all channel conditions).

1. A --- BLER = 60 % during noise bursts, noise burst occurrence rate = 1/1000.
2. B --- BLER = 90 % during noise bursts, noise burst occurrence rate = 1/1000.
3. C --- BLER = 60 % during noise bursts, noise burst occurrence rate = 5/1000.

4. D --- BLER = 90 % during noise bursts, noise burst occurrence rate = 5/1000.

The ratio of MC/OR is to compare the performance of two HARQ schemes (simulation result of multiple-copy HARQ scheme over original HARQ scheme). As shown in Fig. 12, 13, 17, and 18, throughput performance and average waiting time are compared under the situation where the buffer size is equal to the number of HARQ channels given to a SS, i.e., a SS has buffer size of 4 data bursts when giving 4 HARQ channels to this SS.

1) The effects of SS buffer space

Configuration: 4 HARQ channels; buffer space increases from 4 to 7; simulated in channel condition A, B, C, and D.

In addition to knowing that the original HARQ scheme inherently has higher throughput than the multiple-copy HARQ scheme, we also investigate how the size of the SS buffer can influence both HARQ schemes' performances.

The throughput performance of both schemes, using 4 HARQ channels, with a buffer sizes ranging from 4 to 7 is shown in Fig. 10. Generally speaking, lower throughput performance occurs when the channel condition has high overall BLER, as in condition D. The throughput performance of the multiple-copy HARQ scheme is improved only slightly by the increase of the SS buffer size because the multiple-copy scheme encounters high buffer occupancy less frequently; on the other hand, the throughput performance of the original HARQ has a noticeable increase when buffer size

increases. As the buffer size increases, the variation of average waiting time for both schemes using 4 HARQ channels can be seen in Fig. 11. In condition A and B, the multiple-copy HARQ scheme and the original HARQ scheme have very little difference in average waiting time due to similar operations between two HARQ schemes in low overall BLER channel conditions, about 4 ms, despite the buffer size increases from 4 to 7. A bigger difference in average waiting time can be seen in condition C and D, 21 ms and 30 ms respectively; nevertheless, the increase in buffer size still does not have too much influence on average waiting time.

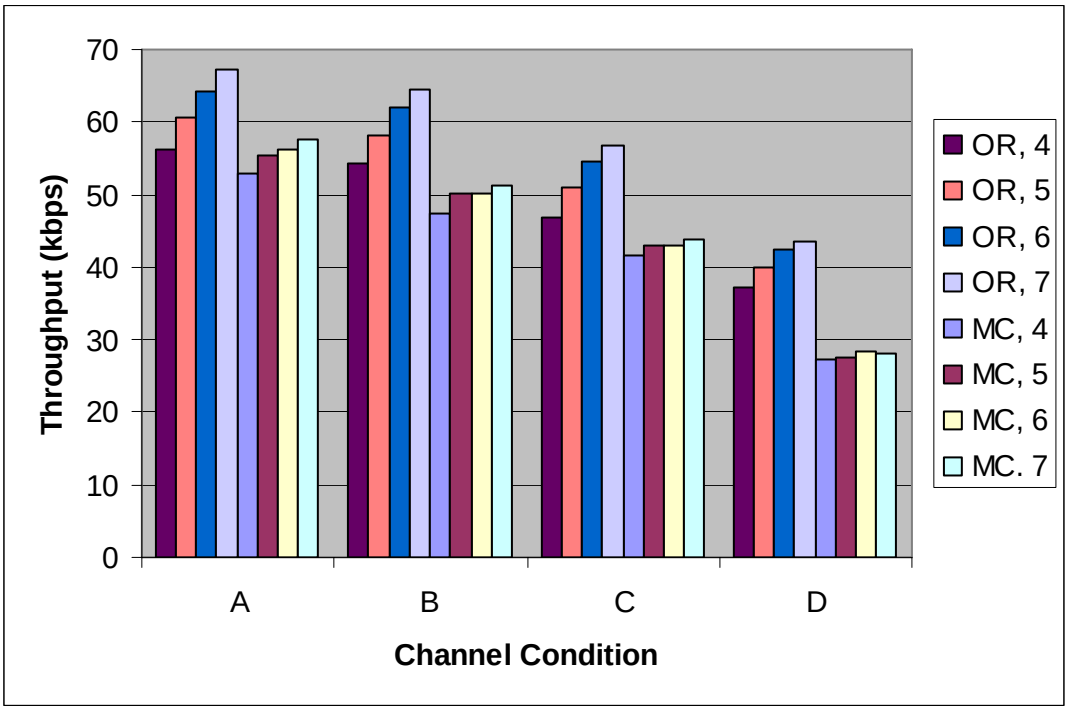

Figure 10. Throughput comparison of both HARQ schemes with different numbers of buffer size.

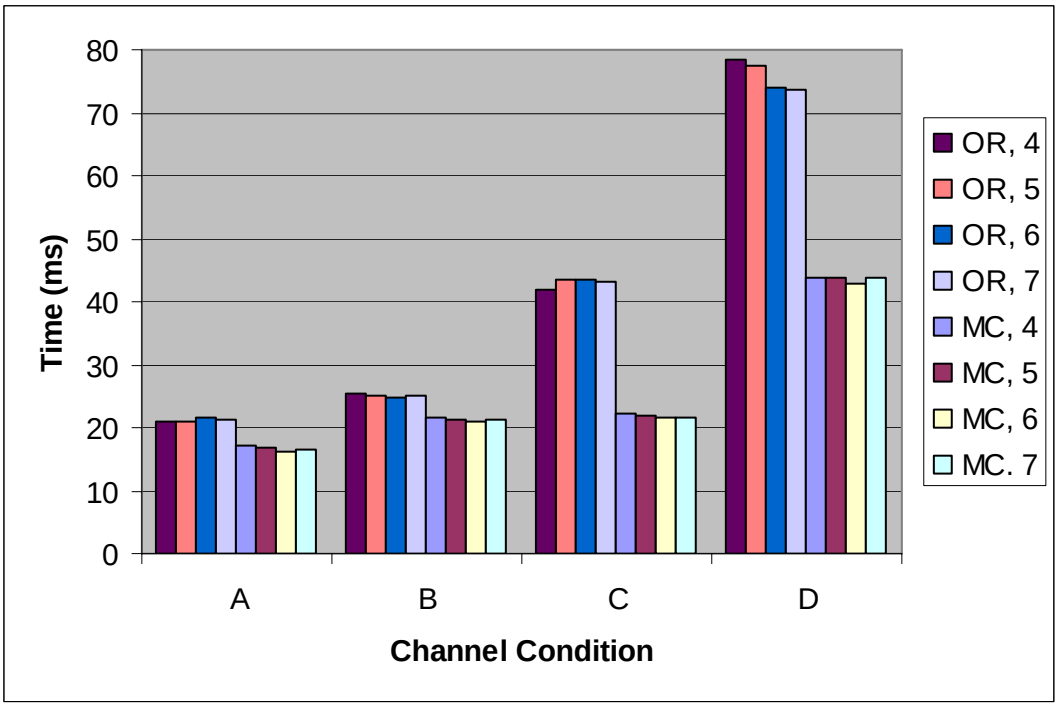

Figure 11. Average waiting time comparison of both HARQ schemes with different number of buffer size.

2) The effects of number of HARQ channels used

Configurations: HARQ channel increases from 4 to 16; buffer space is equals to the number of HARQ channels; simulated in channel condition A, B, C, and D.

A SS can be given more HARQ channels if a BS has more bandwidth resource for that SS; therefore, we investigate how throughput performances of both HARQ schemes are affected when HARQ channels increases from 4 to 16. The multiple-copy HARQ scheme applies the same (re)transmission mechanism as the original HARQ scheme when the channel condition has low overall BLER, as in channel condition A. Therefore, the multiple-copy HARQ scheme's throughput performance is almost as high as the original HARQ scheme's (94% to 99%), as shown in Fig. 12. Also, using multiple-copy

HARQ scheme makes the average waiting time becomes shorter (81% to 90%) compared with the original HARQ scheme, as shown in Fig. 13.

The mechanism of sending multiple copies of the same data burst through contiguous HARQ channels is more likely to be triggered for the multiple-copy HARQ scheme when the channel's overall BLER becomes higher, as in condition D. Therefore, the throughput performance of the multiple-copy HARQ scheme decreases (73% to 80% of the original HARQ scheme's throughput) due to sending some unnecessary copies of data bursts, as shown in Fig. 12. However, the average waiting time is dramatically decreased (52% to 59% of the original HARQ scheme's average waiting time), as shown in Fig. 13.

In Fig. 12, MC/OR ratio shows that increasing channels helps multiple-copy HARQ scheme gain relatively higher throughput compared with the original HARQ scheme because the M_avg is updated more frequently to reflect channel conditions when more HARQ channels are used. When fewer HARQ channels are used, the multiple-copy HARQ scheme reduces relatively more average waiting time compared with the original HARQ scheme because sending multiple copies is triggered more quickly when a small number of HARQ channels are used, as shown in Fig. 13.

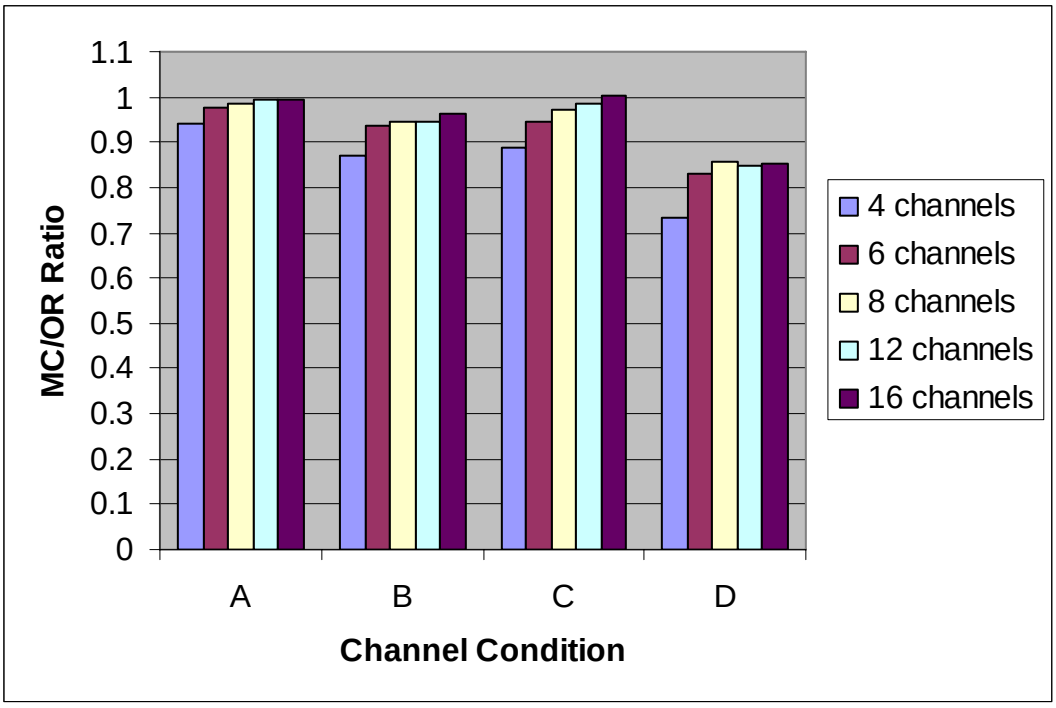

Figure 12. Throughput comparison of both HARQ schemes (MC/OR ratio) when using different number of HARQ channel.

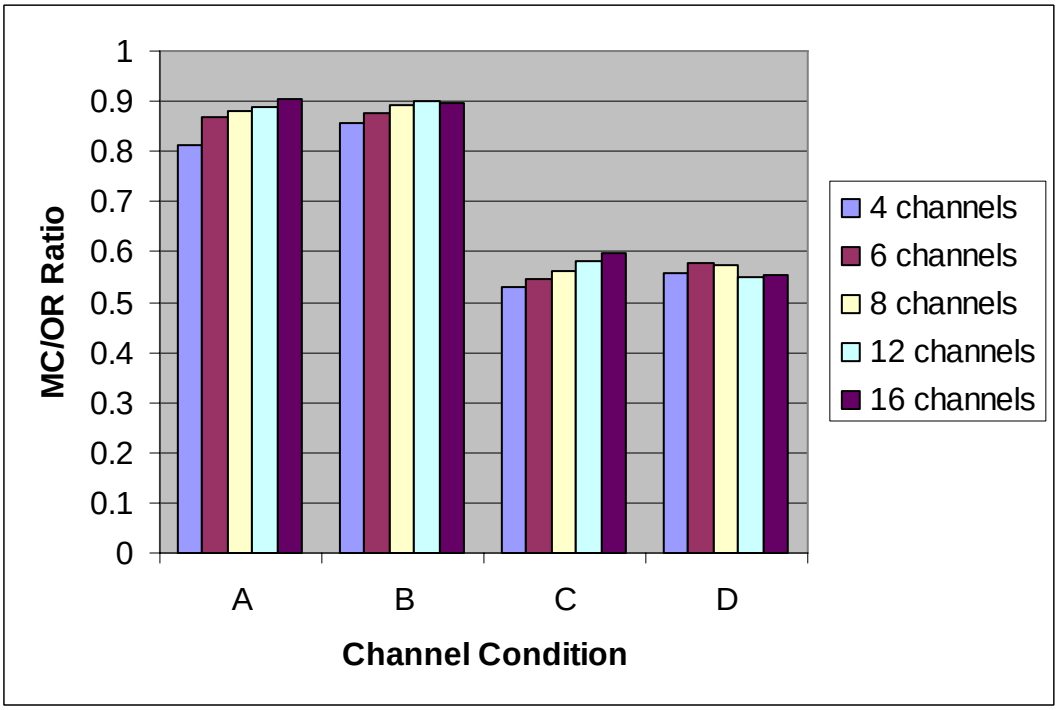

Figure 13. Average waiting time comparison of both HARQ schemes (MC/OR ratio) when using different numbers of HARQ channel.

3) Maximum buffer occupancy

Configurations: HARQ channel increases from 4 to 16; buffer space is unlimited; simulated in channel condition A, B, C, and D.

The maximum buffer occupancy of both HARQ schemes is examined (when buffer is unlimited), simulation results are shown in Fig. 14. Both HARQ schemes have similar maximum buffer occupancy when retransmission times of data burst are low (1 time), as in channel condition A and C. When the retransmission time is high (4 times), as in channel condition B and D, the multiple-copy HARQ scheme sends more copies on contiguous HARQ channels so that unsuccessfully received data bursts are recovered quicker and sent to upper layer with other in-sequence data bursts; therefore, the buffer occupancy is reduced.

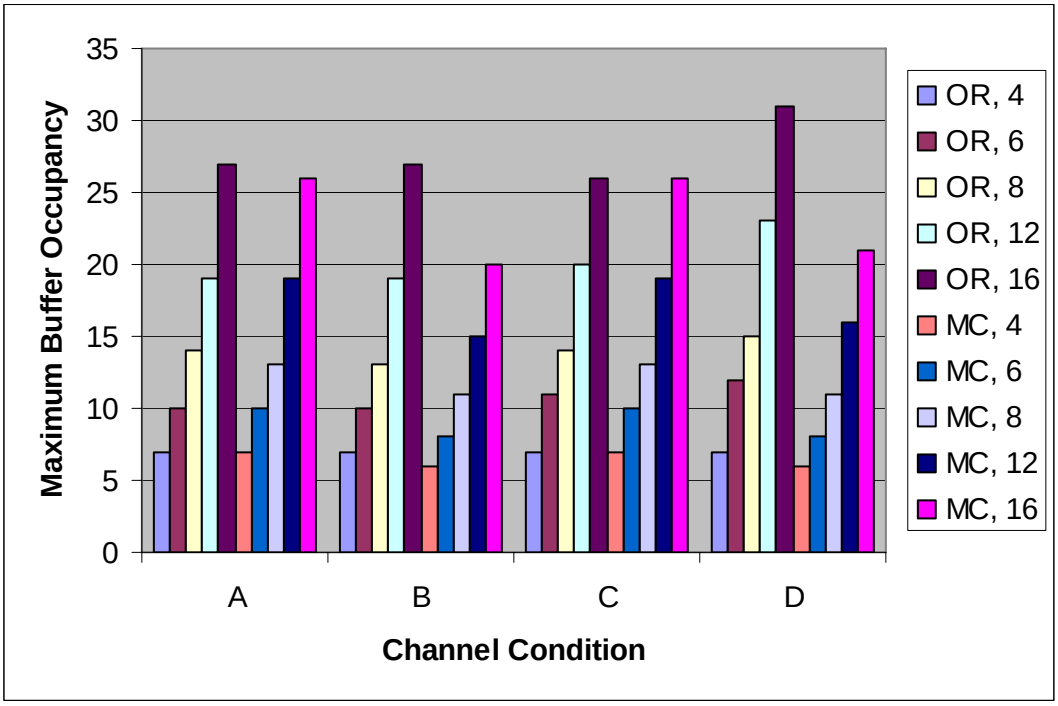

Figure 14. Maximum buffer occupancy comparison of both HARQ schemes.

4) The effects of unlimited buffer space

Configurations: HARQ channel increases from 4 to 16; buffer space is unlimited; simulated in channel condition A, B, C, and D.

We compare throughput performance and average waiting time when the buffer size is unlimited to see the full performance of both HARQ schemes in different channel conditions, simulation results are shown in Fig. 15 and Fig. 16.

The ratio of MC/OR becomes higher when the given number of HARQ channels increases, as shown in Fig. 15, which means that increasing channels helps multiple-copy HARQ scheme gain relatively higher throughput compared with the original HARQ scheme because the M_avg is updated more frequently to reflect channel conditions when more HARQ channels are used. Undoubtedly, the multiple-copy HARQ scheme tremendously reduces average waiting time when the channel condition is noisy, as in condition C and D. How the average waiting time of both HARQ schemes is affected when using more HARQ channels is shown in Fig. 16. When fewer HARQ channels are used, the multiple-copy HARQ scheme reduces relatively more average waiting time compared with the original HARQ scheme because sending multiple copies is triggered more quickly when a small number of HARQ channels are used.

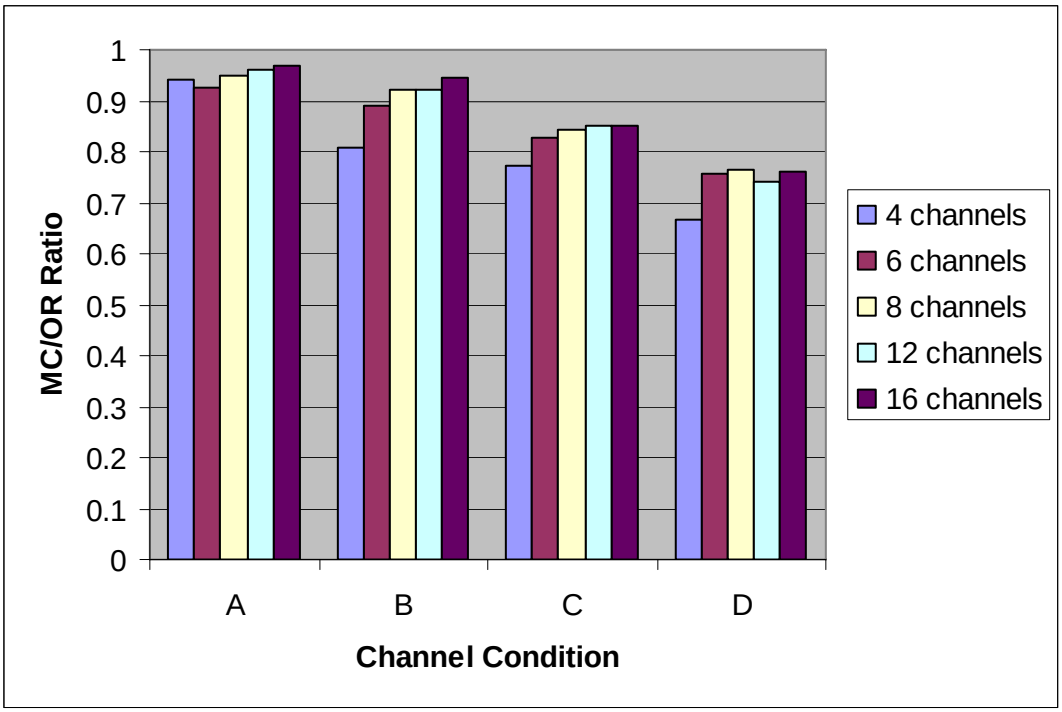

Figure 15. Throughput comparison of both HARQ schemes (MC/OR ratio) when using different numbers of HARQ channel (no buffer limitation).

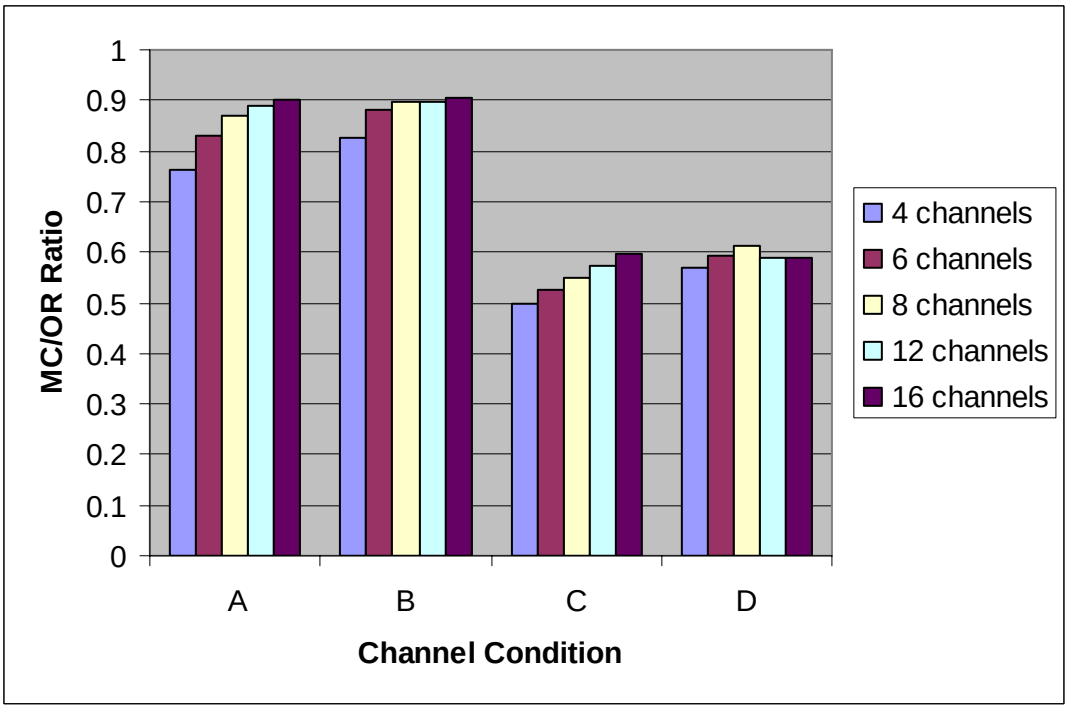

Figure 16. Average waiting time comparison of both HARQ schemes (MC/OR ratio)when using different numbers of HARQ channel (no buffer limitation).

5) A non-IEEE approach

Configurations: HARQ channel increases from 4 to 16; buffer space is equals to the number of HARQ channels; simulated in channel condition A, B, C, and D.

We also investigate the throughput performance and the average waiting time for a situation where the buffer control is done in the SS, i.e., the BS keeps sending data bursts and the SS sends back a NAK if it encounters a buffer overflow situation, simulation results are shown in Fig. 17 and Fig. 18. Basically, both HARQ schemes' throughput performances become higher and average waiting times become lower because the SS can accept data bursts right away if any data bursts get sent to upper layer. As we can see from the MC/OR ratios (in Fig. 12, 13, 17, and 18), the non-IEEE approach has a very similar simulation results as the IEEE approach. The subtle difference in that the multiple-copy HARQ scheme's throughput performance becomes slightly less efficient in all channel conditions, and the reduction of average waiting time becomes slightly less efficient as well in condition D. However, using multiple-copy HARQ scheme still significantly reduces the average waiting time at a relatively small cost of throughput decreasing.

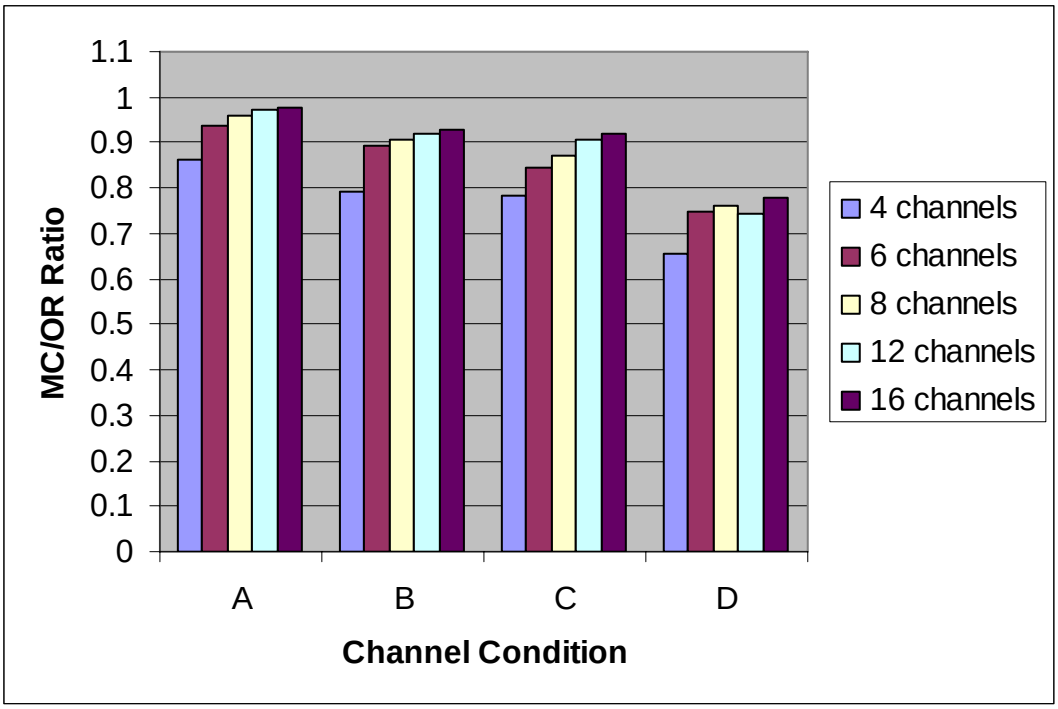

Figure 17 Throughput comparison of both HARQ schemes (MC/OR ratio) when using different number of HARQ channel (non-IEEE approach)

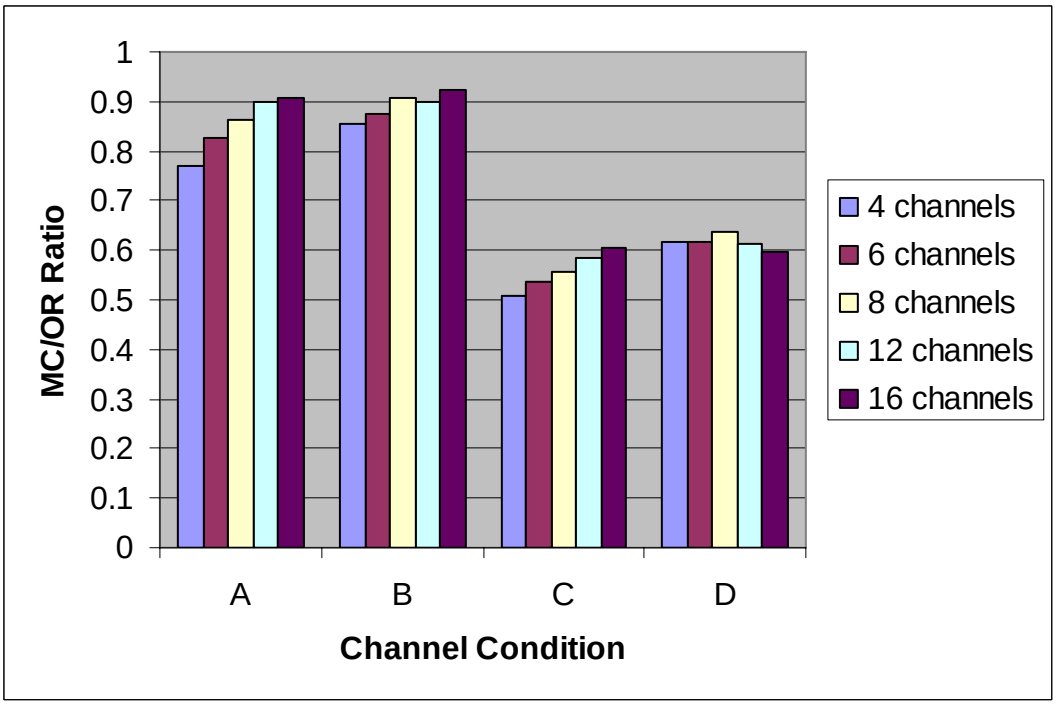

Figure 18. Average waiting time comparison of both HARQ schemes (MC/OR ratio) when using different numbers of HARQ channel (non-IEEE approach)

The following is the summary of the simulation results mentioned above, regarding throughput performance, average waiting time, and maximum buffer occupancy.

- If noise bursts' occurrence rate is low and BLER is also low during noise bursts, the multiple-copy HARQ scheme can have throughput as high as the original HARQ scheme's and also have slightly lower average waiting time than the original HARQ scheme's.

- If noise bursts' occurrence rate is high and BLER is also high during noise bursts, the multiple-copy HARQ scheme's average waiting time is dramatically decreased compared with the original HARQ scheme's; in the same situation, the throughput of the multiple-copy HARQ scheme decreases, but not as much as its average waiting time.

- When giving more SS buffer, the original HARQ scheme has a noticeable increase in throughput in all channel conditions; on the other hand, the multiple-copy HARQ scheme's throughput only increases a little bit when giving more SS buffer.

- Generally, the average waiting time of both multiple-copy HARQ scheme and original HARQ scheme is barely affected by the increase of SS buffer (SS buffer >= number of HARQ channels). Except for a channel condition where noise bursts' occurrence rate is high and BLER is high during noise bursts, the average waiting time of the original HARQ scheme decreases when giving more SS buffer.

- If more HARQ channels are given, the throughput performance of the multiple-copy HARQ scheme gets closer to that of the original HARQ scheme in almost all channel conditions except in a channel condition where noise bursts' occurrence rate is high and BLER is high during noise bursts.

- If more HARQ channels are given, using the multiple-copy HARQ scheme reduces less average waiting time in almost all channel conditions except in a channel condition where noise bursts' occurrence rate is high and BLER is high during noise bursts.

- In channel conditions where BLER is high during noise bursts (more retransmissions are needed), using multiple-copy HARQ scheme has lower maximum buffer occupancy. In other channel conditions, both HARQ schemes have similar maximum buffer occupancy.

- Simply dropping data when encountering buffer overflows at the SS side has higher throughput and lower average waiting time than conducting buffer control at the BS side; however, using multiple-copy HARQ scheme in this condition becomes less efficient in throughput performance (MC/OR of throughput is lower).

6. Conclusions

6.1 Project Achievements

The original stop-and-wait HARQ scheme uses HARQ channels that are independent of each other; therefore, retransmission delay may become large because retransmissions can only be done by the initial sending channel. The proposed multiple-

copy HARQ scheme has the mechanism of sending multiple copies of the same data burst through contiguous channels when channel conditions become noisy, so data bursts that need retransmissions have a shorter waiting time in the SS buffer. In this project, we examine the performances of the multiple-copy HARQ scheme and the original stop-and-wait HARQ scheme. Based on simulation results, the multiple-copy HARQ scheme can have a low of about 56% average waiting time of the original HARQ scheme's average waiting time, yet the throughput of the multiple-copy HARQ scheme can still reach 73% of the original HARQ scheme's throughput. Therefore, the proposed multiple-copy HARQ scheme can be very beneficial to wireless communication networks that easily suffer from noise bursts and require more retransmissions.

6.2 Future Enhancements

The proposed multiple-copy HARQ scheme uses a simple scheme, derived from ACK/NAK feedbacks sent back to the BS, to estimate channel conditions in order to decide how many multiple copies are required to handle bad channel conditions. The throughput performance of the proposed multiple-copy HARQ scheme could decline when sending more copies than necessary. To estimate changing channel conditions precisely is difficult and not realistic; therefore, a scheme that can quickly adapt to channel conditions is truly desirable. Each SS may be given a different number of HARQ channels based on available resources in the BS. If a HARQ scheme purely relies on ACK/NAK feedbacks to adapt channel conditions, having fewer HARQ channels may reduce the correctness of estimating a channel's condition due to receiving fewer feedbacks. Hence, it would be better to reply on some existing functions that constantly examine channel conditions and provide feedbacks to the BS. The channel quality

indicator (CQI) reports channel condition regularly. A BS may make use of CQI information to get sufficient channel condition feedbacks regardless of how many HARQ channels are given to the SS. Lastly, the proposed multiple-copy HARQ scheme currently works on synchronous mode; therefore, it can also be further enhanced to work on partially asynchronous mode or fully asynchronous mode. In conclusion, future enhancements can be done probably in two major aspects: being more adaptive to channel conditions and relying on regularly received feedback information.

6. References

[1] 3rd Generation Partnership Project, TSG-RAN Working Group 1, "Text Proposal for the TR 25.848", January 15 – 19, 2001. Retrieved July 10, 2007, from http://www.3gpp.org/ftp/tsg_ran/WG1_RL1/ TSGR1_18/ Docs/PDFs/R1-01-0124.pdf

[2] 3rd Generation Partnership Project, "Technical Specification Group Radio Access Network; Physical Layer Aspects of UTRA High Speed Downlink Packet Access", V 4.0.0, 2001 – 2003. Retrieved Aug. 1, 2007, from http://www.3gpp.org/ftp/Specs/html-info/25848.htm

[3] M. Aghadavoodi Jolfaei, K. Aghadavoodi Jolfaei, S. Baucke and J. Kaltwasser, "Improved Selective Repeat ARQ Schemes for Data Communication", Proceedings of IEEE Vehicular Technology Conference, vol.3, pp.1407 - 1411, June 8-10, 1994

[4] S. S. Ara, A. M. Shah and M. Matsumoto, "An Efficient Selective-repeat ARQ Scheme for Half-duplex Infrared Links under High Bit Error Rate Conditions", Proceedings of IEEE Consumer Communications and Networking Conference, Vol. 2, pp.1088 - 1092, Jan. 8-10, 2006

[5] S. S. Ara, A. M. Shah and M. Matsumoto, "Block Window Retransmission ARQ Scheme for Next Generation High Speed IrDA Links", Proceedings of IEEE International Symposium on Wireless Pervasive Computing, pp.1 - 4, Jan. 16-18, 2006

[6] G. Benelli, "Some ARQ protocols with finite receiver buffer", Proceedings of IEEE Transactions on Communications, Vol. 41, Issue 4, pp.513 - 523, April 1993

[7] H. Bruneel and M. Moeneclaey, "On the Throughput Performance of Some Continuous ARQ Strategies with Repeated Transmissions", Proceedings of IEEE Transactions on Communications, Vol. 34, Issue 3, pp. 244 - 249, Mar. 1986

[8] J. Chen and J. Wang, "A novel selective repeat stop-wait ARQ for half-duplex channels", Proceedings of IEEE Region 10 Conference on Computers, Communications, Control and Power Engineering, Vol. 2, pp.1238 – 1241, Oct. 28-31, 2002

[9] K. D. Chase, "Code Combining: A maximum-likelihood Decoding Approach for Combining an Arbitrary Number of Noisy Packets," IEEE Transactions on Communications, vol. 33, pp. 593–607, May 1985

[10] M. W. El Bahri, H. Boujernaa and M. Siala, " Performance Comparison of type I, II, and III Hybrid ARQ Schemes over AWGN Channels", IEEE International Conference on Industrial Technology, Volume 3, pp. 1417-1421, Dec. 8-10, 2004

[11] F. Frederiksen and T. E. Kolding, "Performance and Modeling of WCDMA/HSDPA Transmission/H-ARQ Schemes", Proceedings of 2002 IEEE 56th Vehicular Technology Conference, Volume 1, pp. 472-476, Sept. 24-28, 2002

[12] R. Fantacci, “Performance evaluation of efficient continuous ARQ protocols”, IEEE Transactions on Communications, Vol. 38, Issue 6, pp. 773 – 781, June 1990

[13] S. Falahati, T. Ottosson, A. Svensson and Z. Lin, “Convolutional Coding and Decoding in Hybrid Tyep-II ARQ Schemes on Wireless Channels”, IEEE 49th Vehicular Technology Conference, Volume 3, pp. 2219-2224, May 16-20, 1999

[14] O. Gurbuz and E. Ayanoglu, “A Transparent ARQ Scheme for Broadband Wireless Access”, Proceedings of IEEE Wireless Communications and Networking Conference, Vol. 1, pp.423 - 429, 21-25 March 2004

[15] IEEE, 802.16 IEEE Standard for Local and Metropolitan Area Networks, Part 16: Air Interface for Fixed Broadband Wireless Access Systems, 2004

[16] IEEE, 802.16 IEEE Standard for Local and Metropolitan Area Networks, Part 16: Air Interface for Fixed and Mobile Broadband Wireless Access Systems, 2005

[17] W. S. Jeon and D. G. Jeong, “Improved Selective Repeat ARQ Scheme for Mobile Multimedia Communications”, IEEE Communications Letters , Vol. 4, Issue 2, pp. 46 – 48, Feb. 2000

[18] L. Kleinrock and H. Opderbeck, "Throughput in the ARPANET—Protocols and Measurement", IEEE Transactions on Communications, Vol. 25, Issue 1, pp. 95-104, Jan. 1977

[19] P. Kosut and J. Polec, “Investigation into Optimum Go-Back-N ARQ Strategy of Bruneel and Moeneclaey”, IEEE Electronics Letters, Vol. 36, Issue 4, pp. 381 - 382, Feb. 17, 2000

[20] S. Kallel, “Complementary Punctured Convolutional (CPC) Codes and Their Applications”, IEEE Transactions on Communication, Volume 43, Issue 6, pp. 2005-2009, June 1995

[21] S. Kallel, “Efficient Stop-and-Wait Type II hybrid ARQ scheme”, IEEE Electronics Letters, Volume 28, Issue 12, pp. 1097 – 1098, June 4, 1992

[22] S. Kallel and C. Leung, “Efficient ARQ Schemes with Multiple Copy Decoding”, IEEE Transactions on Communications, Volume 40, Issue 3, pp. 642 – 650, March 1992

[23] The Illinois Network Design and EXperimentation (INDEX) Group, Referred September 3, 2006, from http://www.j-sim.org/

[24] WiMAX Forum, “WiMAX System Evaluation Methodology”, version 1.0, 1/20/2007

[25] WiMAX Forum, “Mobile WiMAX – Part II: A comparative Analysis”, May 2006

[26] WINNER, “Test Scenarios and Calibration Cases Issue 2”, IST-4-027756 WINNER II, D6.13.7 v1.00, Dec. 31, 2006, Retrieved July 15, 2007, from https://www.ist-winner.org/WINNER2-Deliverables/D6.13.7.pdf

[27] H. Zheng, A. Lozano and M. Haleem, ”Multiple ARQ Processes for MIMO Systems”, The 13th IEEE International System Symposium on Personal, Indoor and Mobile Radio Communication, Vol. 3, pp. 1023-1026, Sept. 15-18, 2002

Appendix A: Source Code

- **Multiple-copy/Original HARQ Scheme (BS operation)**

```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.Queue;
import java.util.Vector;
/**
 * This class simulates HARQ's operations. Each HARQ process runs one after another, receiving
 * feedbacks and then sending data bursts to the Subscriber Station.
 * @author Yucheng
 */
public class HARQ_SIM {
    private int process_num;   //number of HARQ process
    private int SimDuration;  //duration of simulation in frames
    private double M = 1;   //M'
    private int[] AI_SN;    //AI_SN for each HARQ process
    private SS_HARQ SS;     //HARQ handler in the SS
    private int SN = 0;     //sequence number of current data burst being sent
    private boolean initial = true;     //for initializing purpose
    private Queue<String> feedbackQueue;//a queue that stores feedbacks from the SS
    private Vector<DataInfo> queue = new Vector<DataInfo>();  //data bursts information queue
    private Vector<HARQ_process> process = new Vector<HARQ_process>();// processes queue
    private boolean multipleCopy; //true, mulitple-copy HARQ is enabled; false, original HARQ is enabled
    private boolean limit_buffer = false;//if limiting buffer is enabled
    private int buffer_limit = 4; //given buffer size
    private int maximum_process = 16;//max number of process, also used as feedback delay
    private boolean SSlimit_buffer = false;//if SS-side buffer limit is enabled
        /**
         * HARQ_SIM constructor
         *@param simDur duration of the simulation
         *@param proNum number of processes
         *@param MC multiple-copy is enabled or not
         *@param limit_buffer buffer is limited or not; 1-->limited on BS side, 2-->limited on SS side
         *@param Channel_condition control setting of channel condition
         */
      public HARQ_SIM(int simDur, int proNum, boolean MC, int limit_buffer, int BL_num, int
Channel_condition){
          SimDuration = simDur;
          process_num = proNum;
          multipleCopy =  MC;
          if(limit_buffer == 1){
            this.limit_buffer = true;
          }else if(limit_buffer == 2){
            SSlimit_buffer = true;
          }
          buffer_limit = BL_num;
          feedbackQueue = new LinkedList<String>();
           AI_SN = new int[maximum_process];
           for(int i = 0; i < maximum_process; i++){
                AI_SN[i] = 0;//initialize AI_SN for each process
                if(i<process_num){
                 process.add(new HARQ_process(i, 1));//initialize processes
                }else{
                 process.add(new HARQ_process(i, 0));//initialize processes (not used)
```

```
            }
    }
  SS = new SS_HARQ(maximum_process, SSlimit_buffer, BL_num, Channel_condition);
}
/**
 * Start the simulation
 */
public Results start(){
  Results r = null;
  int channel = 0;//rotate processes (0 - process_num)
  int pn;//process number, for updating AI_SN
  int seq_num; //first pdu sn in the data burst being sent
  while(SimDuration > 0){
            if(channel == maximum_process){
                        initial = false;
                        channel = 0; //rotating to the fist process
            }
            HARQ_process p = process.get(channel);//one process after another gets to operate
            if(initial){
              if(channel < process_num){
                        p = process.get(channel);
                        p.addInfo(queue, SN, M);
                 SS.processDataBursts(channel, channel, 0, 0, feedbackQueue, SN);
                        SN += 10;//10 PDUs in a data burst
              }else{
                        SS.processDataBursts(channel, channel, 1, 0, feedbackQueue, -1);
              }
              channel++;//next channel
              SimDuration--;
            }else{
               String feedback = feedbackQueue.remove();
            //multiple copy HARQ mechanism
            if(multipleCopy){
              if(channel >= process_num){
                SS.processDataBursts(channel, channel, 1, AI_SN[channel], feedbackQueue, -1);
              }else{
                   if(limit_buffer){//buffer is limited
                       if(feedback.equalsIgnoreCase("ACK")){
                            boolean update = p.ACKupdate(queue);
                            if(update){
                                  calculateM();
                             }
                      }else if (feedback.equalsIgnoreCase("NAK")){
                                p.NAKupdate(queue, process);
                                calculateM();
                       }
                       if(p.checkT(queue)){//data burst in status T
                            seq_num = p.sendMC(queue, M);
                        }else{//no data burst in status T
                              if(p.checkR(queue)){//any data burst in status R
                                        seq_num = p.resendDataBurst(queue);
                              }else{//NO data burst in status R
                                        if(p.outstandingACID(queue)){
                                          seq_num = p.sendExtraDataBurst(queue);
                                        }else{
                                             if(okToSend()){//check max buffer
```

```
                                        pn = p.sendNewDataBurst(queue, SN, M);
                                        seq_num = SN;
                                        SN += 10;
                                        if(AI_SN[pn] == 0){
                                          AI_SN[pn] = 1;
                                        }else{
                                          AI_SN[pn] = 0;
                                        }
                                    }else{
                                        p.notSendNewDataBurst();
                                        seq_num = SN;
                                    }
                                }
                            }
                        }
                    //SS operations
    SS.processDataBursts(channel, p.getICN(), p.getMC(), AI_SN[p.getICN()], feedbackQueue, seq_num);
            }else{//buffer is not limited
                if(feedback.equalsIgnoreCase("ACK")){
                 boolean update = p.ACKupdate(queue);//dataInfo queue
                    if(update){
                            calculateM();
                    }
                  }else if (feedback.equalsIgnoreCase("NAK")){
                 p.NAKupdate(queue, process);
                    calculateM();
                  }
                  if(p.checkT(queue)){//data burst in status T
                   seq_num = p.sendMC(queue, M);
                  }else{//no data burst in status T
                       if(p.checkR(queue)){//any data burst in status R
                          seq_num = p.resendDataBurst(queue);
                       }else{//NO data burst in status R
                            if(p.outstandingACID(queue)){
                                 seq_num = p.sendExtraDataBurst(queue);
                       }else{
                            pn = p.sendNewDataBurst(queue, SN, M);
                            seq_num = SN;
                            SN += 10;
                            if(AI_SN[pn] == 0){
                              AI_SN[pn] = 1;
                            }else{
                              AI_SN[pn] = 0;
                            }
                       }
                  }
              }
          //SS operations
SS.processDataBursts(channel, p.getICN(), p.getMC(), AI_SN[p.getICN()], feedbackQueue, seq_num);
        }
    }
}else{// Original SW HARQ operation
    if(channel >= process_num){
          SS.processDataBursts(channel, channel, 1, AI_SN[channel], feedbackQueue, -1);
    }else{
          if(limit_buffer){//limit buffer at BS side
```

```
                if(feedback.equalsIgnoreCase("ACK")){
                  p.deleteInfo(queue);
                    if(okToSend()){//ok to send a new data burst
                          if(AI_SN[channel] == 1){
                            AI_SN[channel] = 0;
                          }else{
                            AI_SN[channel] = 1;
                          }
                          p.addInfo(queue, SN, 0);// 0 is for M, not used here
                          SS.processDataBursts(channel, channel, 0, AI_SN[channel], feedbackQueue, SN);
                          SN += 10;
                     }else{//reach max buffer
                          SS.processDataBursts(channel, channel, 0, AI_SN[channel], feedbackQueue,
SN);//simulate not sending
                        }
              }else if (feedback.equalsIgnoreCase("NAK")){
                        seq_num = p.getOSN(queue);//get original sequence number
                        SS.processDataBursts(channel, channel, 1, AI_SN[channel], feedbackQueue, seq_num);
               }else{//receive null because buffer was full, it did not send out a data burst
                    if(okToSend()){
                          if(AI_SN[channel] == 1){
                                  AI_SN[channel] = 0;
                          }else{
                                  AI_SN[channel] = 1;
                          }
                          p.addInfo(queue, SN, 0);// 0 is for M, not used here
                          SS.processDataBursts(channel, channel, 0, AI_SN[channel], feedbackQueue, SN);

                          SN += 10;
                    }else{
                           SS.processDataBursts(channel, channel, 0, AI_SN[channel], feedbackQueue,
SN);//simulate not sending
                      }
                   }
              }else{// buffer is not limited
                if(feedback.equalsIgnoreCase("ACK")){
                    p.deleteInfo(queue);
                      if(AI_SN[channel] == 1){//switch AI_SN
                                  AI_SN[channel] = 0;
                       }else{
                                  AI_SN[channel] = 1;
                         }
                         p.addInfo(queue, SN, 0);
                         SS.processDataBursts(channel, channel, 0, AI_SN[channel], feedbackQueue, SN);
                          SN += 10;
                    }else if (feedback.equalsIgnoreCase("NAK")){
                      seq_num = p.getOSN(queue);
                      SS.processDataBursts(channel, channel, 1, AI_SN[channel], feedbackQueue, seq_num);
                      }
                    }
                }
              }
               channel++;
               SimDuration--;
              }
    }
```

```
  if(SimDuration - 1 ==-1){
    r = SS.getThroughput(multipleCopy);
  }
  return r;
}
        /**
         * Check buffer limitation
         * @return true, max buffer has not reached; otherwise, false.
         */
        private boolean okToSend(){
                if(queue.size() == 0){//data info queue
                        return true;
                }else{
                        DataInfo dataInfo = queue.firstElement();
                        if(SN - dataInfo.getSN() >= buffer_limit*10){
                                return false;
                        }else{
                                return true;
                        }
                }
        }
        /**
         * Calculate M' according to all Ms in processes
         */
        public void calculateM(){
                double M_total = 0.0;
                for (Iterator it = process.iterator(); it.hasNext(); ) {
                        HARQ_process p = (HARQ_process) it.next();
                        M_total += p.getM();
                }
                M = Math.floor(M_total/process_num + 0.5);
        }
        /**
         * Main method
         * @param args arguments
         */
        public static void main(String args[]){
                int numOfRuns = 1;//how many time the simulation runs
                boolean MC;//multiple-copy scheme
                LinkedList<Results> results = new LinkedList<Results>();
                if(args[2].equals("0")){
                        MC = false;
                }else{
                        MC = true;
                }
                for(int i = 0; i < numOfRuns; i++){
                        HARQ_SIM sim = new HARQ_SIM(Integer.valueOf(args[0]).intValue(),
            Integer.valueOf(args[1]).intValue(),
            MC, Integer.valueOf(args[3]), Integer.valueOf(args[4]),Integer.valueOf(args[5]) );
                        results.add((Results)sim.start());
                }
                double throughput = 0.0;
                double average_waiting_time = 0.0;
                for(int i = 0; i< numOfRuns; i++){
                        Results r = results.get(i);
                        throughput += r.getTE();
```

```
                        average_waiting_time += r.getAWT();
                }
        System.out.println("*********************************************************************");
        System.out.println("Throughput: " + throughput/numOfRuns +" kbps");
        System.out.println();
        System.out.println("Average Waiting Time: " + average_waiting_time/numOfRuns+ " ms");
        System.out.println("*********************************************************************");
        }
}
```

****
****

- **Multiple-copy/Original HARQ Scheme (SS operation)**

```
import java.util.Collections;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.ArrayList;
import java.util.Queue;
import java.util.Random;
/**
 * This class simulates the operations in the Subscriber Station when receiving data bursts.
 * @author Yucheng
 */
public class SS_HARQ {
    private Integer[] UDDB;   //unsuccessfully decided data bursts
    private Integer[] AI_SN;  //recording AI_SN for each channel
    private ChannelSim cs;    //channel condition simulation class
    private int SN = 0;//smallest sequence number for PDU that is waiting to be sent to the upper layer
    private int size = 10;    //supposing ten PDUs per data burst
    private int totalSDDB = 0;//total successfully decoded data bursts
    private int totalTDB = 0;//total transmitted data bursts
    private int totalWaitingTime = 0;
    private int timer = 0; //a timer, for calculating data burst waiting time in SS
    private int[] combiningTime;//recording time of combining
    private LinkedList<Integer> buffer = new LinkedList<Integer>();//out of order data bursts (successfully
decoded)
    private ArrayList<BufferedDB> bufferedDB = new ArrayList<BufferedDB>();// buffered data bursts
    private int frameSize = 5;// 5 ms frame
    private int buffersize = 0;//current buffer size
    private int[] combiningTimeRequired; //store combining time required
    private boolean limit_buffer = false; // SS-side buffer limitation enabled or not
    private boolean enabled = false; // if SS-side buffer limitation enabled or not
    private int buffer_limit = 16;//buffer size
    private int bit_per_databurst = 1536;// size data burst in bits
        /**
         * HARQ operation in the Subscriber Station
         * @param numOfChannel number of channels
         * @param BL if SS-side buffer limit is enabled
         * @param BL_num value of limited buffer
         * @param Channel_Concition setting of channel condition
         */
        public SS_HARQ(int numOfChannel, boolean BL, int BL_num, int Channel_Condition){
                AI_SN = new Integer[numOfChannel];
                UDDB = new Integer[numOfChannel];
                combiningTime = new int[numOfChannel];
```

```
                combiningTimeRequired = new int[numOfChannel];
                limit_buffer = BL;
                enabled = BL;
                buffer_limit = BL_num;
                cs = new ChannelSim(Channel_Condition);
                for(int i=0; i<numOfChannel;i++){
                        AI_SN[i]= new Integer(1);
                }
        }
        /**
         * Process data bursts information
         * @param ACID_ ID of sending channel
         * @param ICN initial sending channel; 0 if MC = 0
         * @param MC mulitple copy
         * @param AI_SN_ HARQ identifier sequence number
         * @param feedbackq feedback queue
         * @param sn sequence number of this data burst
         */
        public void processDataBursts(int ACID_, int ICN, int MC, int AI_SN_, Queue<String>
feedbackq, int sn){
                timer++;
                totalTDB++;//total data burst received
                if(limit_buffer){
                        if(buffersize < buffer_limit && sn < (SN + buffer_limit*10)){
                                enabled = true;
                        }else{
                                enabled = false;
                        }
                }else{
                        enabled = true;
                }
                if(MC == 0 && enabled){//receiving initial copy of data burst
                    if((AI_SN[ACID_]).intValue() != AI_SN_){//new data burst, change AI_SN_
                        AI_SN[ACID_] = new Integer(AI_SN_);
                          if(cs.decodeDB(ACID_)){//first copy is successfully decoded
                                bufferedDB.add(new BufferedDB(timer, sn));
                                reordering(sn);
                                feedbackq.add("ACK");
                        }else{//not successfully decoded
                                UDDB[ACID_] = new Integer(ACID_);
                                bufferedDB.add(new BufferedDB(timer, sn));
                                feedbackq.add("NAK");
                            }
                        }else{//sles, ignore same AI_SN
                              cs.isburstNoise();
                              feedbackq.add("NULL");
                        }
                }else{//receiving second or later copies
                        if(sn == -1){//simulate not sending
                              cs.isburstNoise();
                              feedbackq.add("NULL");
                        }else{
                             if(UDDB[ICN] != null){//unsuccessfully data burst exists
                                  if(cs.decodeDB(ICN)){//combinined, and successfully decoded
                                      feedbackq.add("ACK");
                                      reordering(sn);
```

```
                                }else{//not successfully decoded
                                  feedbackq.add("NAK");
                                }
                            }else{
                                boolean temp = false;
                                for (Iterator it = buffer.iterator(); it.hasNext(); ) {
                                    Integer seq = (Integer) it.next();
                                    if(seq.intValue() == sn){
                                        temp = true;
                                     }
                                 }
                                 if(limit_buffer){
                                    if (sn < SN || temp){
                                       cs.isburstNoise();
                                       feedbackq.add("Null");
                                     }else{
                                        if(buffersize < buffer_limit && sn < (SN + buffer_limit*10)){
                                                if((AI_SN[ICN]).intValue() != AI_SN_){
                                                    AI_SN[ICN] = new Integer(AI_SN_);
                                                     if(cs.decodeDB(ICN)){
                                                        bufferedDB.add(new BufferedDB(timer, sn));
                                                        reordering(sn);
                                                        feedbackq.add("ACK");
                                                      }else{
                                                        UDDB[ICN] = new Integer(ICN);
                                                        bufferedDB.add(new BufferedDB(timer, sn));
                                                         feedbackq.add("NAK");
                                                       }
                                                  }else{
                                                        //ignore same AI_SN
                                                  }
                                            }else{
                                                  cs.isburstNoise();
                                                   feedbackq.add("NAK");
                                              }
                                        }
                                    }else{//buffer is not limited
                                            cs.isburstNoise();
                                            feedbackq.add("Null");
                                        }
                                    }
                            }
                    }
                    int temp = 0;
                    for(int i = 0; i < UDDB.length; i++){
                            if(UDDB[i] != null){
                                    temp +=1;
                            }
                    }
                    buffersize = buffer.size()+ temp;
            }
            /**
             * Reording stored data bursts to send to the upper layer
             * @param i first sequence number of a successfully decoded data burst
             */
            private void reordering(int sn_){
```

```
		int seq_num = sn_;
		if(seq_num == SN){//this data burst is the next expected data burst
			totalSDDB++;
			calculateBufferedTime(seq_num);
			seq_num += size;
			for (Iterator it = buffer.iterator(); it.hasNext(); ) {
				Integer sn = (Integer) it.next();
				if(sn.intValue() == seq_num){//sent in order PDUs to the upper layer
					it.remove();
					calculateBufferedTime(seq_num);
					totalSDDB++;
					seq_num += size;
				}else{
					break;
				}
			}
			SN = seq_num;//set smallest sequence number
		}else{// not in order, store the data burst
			buffer.add(new Integer(seq_num));
			Collections.sort(buffer);//sorting the buffer
		}
	}
	/**
	 * Calculate the time that a buffer has been stored in the SS before sending to the upper layer
	 * @param sn sequence number of the first PDU in the data burst
	 */
	private void calculateBufferedTime(int sn){
		for(int i = 0; i < bufferedDB.size(); i++){
			BufferedDB db = (BufferedDB) bufferedDB.get(i);
			if(db.getSN() == sn){
				totalWaitingTime += (timer - db.getTime());
				bufferedDB.remove(i);
				break;
			}
		}
	}
	/**
	 * This class simulates channel conditions
	 */
	public class ChannelSim{
		Random generator = new Random();
		private int BLER = 0;	//normal block error rate (percentage)
		private boolean burstNoise = false;//is burst noise happening?
		private int noiseDuration;// (changeable) burst noise duration time (in frames)
		private int noiseDuration_ = 100;// used for assigning a fixed value (in frames)
		private int burstNoiseFrequency = 1;//(out of 1000)occurrence rate of noise bursts
		private int burstBLER = 90;//block error rate when noise burst is happening (percentage)
		private boolean burstNoiseEnabled = true;
		public ChannelSim(int channel_setting){
			if(channel_setting == 0){
				burstNoiseFrequency = 1;
				burstBLER = 90;
			}else if(channel_setting == 1){
				burstNoiseFrequency = 1;
				burstBLER = 60;
			}else if(channel_setting == 2){
```

```
                burstNoiseFrequency = 5;
                burstBLER = 90;
        }else if(channel_setting == 3){
                burstNoiseFrequency = 5;
                burstBLER = 60;
        }
}
/**
 * simulate burst noises
 */
public boolean isburstNoise(){
        if(!burstNoise){
                int value = generator.nextInt(1000);
                if(value < burstNoiseFrequency){
                        burstNoise = true;
                        noiseDuration = noiseDuration_;
                }
        }else{//during a noise burst
                int value = generator.nextInt(1000);
                if(value < burstNoiseFrequency){
                        burstNoise = true;
                        noiseDuration += noiseDuration_;
                }
                noiseDuration--;
                if(noiseDuration == 0){
                        burstNoise = false;
                }
        }
        return burstNoise;
}
/**
 * Simulate decoding of a data burst, including combining
 * @param acid ACID of the data burst
 * @return true, successful; false, not successful
 */
public boolean decodeDB(int acid){
        Integer uddb = UDDB[acid];
        if(burstNoiseEnabled){
                if(isburstNoise()){
                    int value = generator.nextInt(100);
                    int combineControl = 0;
                    if(burstBLER == 60){ combineControl = 80;}
                    if(burstBLER == 90){ combineControl = 75;}
                    if(uddb == null){//check if combining time needs to be changed
                     if(value < combineControl){
                            if(burstBLER == 60){
                                combiningTimeRequired[acid] = 1;
                            }else{
                                combiningTimeRequired[acid] = 4;
                             }
                       }else{
                             if(burstBLER == 60){
                                   combiningTimeRequired[acid] = 2;
                             }else{
                                   combiningTimeRequired[acid] = 3;
                              }
```

```
                                        }
                                    }
                                        BLER = burstBLER;
                                }else{//normal channel condition
                                    combiningTimeRequired[acid] = 1;
                                    BLER = 10;
                                }
                            }
                            //if there is a unsuccessfully decoded data burst exists
                            //check combining time required
                            if(uddb != null){
                                int time = combiningTime[acid] + 1;
                                if(time < combiningTimeRequired[acid]){
                                    combiningTime[acid] = time;
                                    return false;
                                }else{//successful after combining
                                    combiningTime[acid] = 0;
                                    UDDB[acid] = null;// take it out after successfully decoding
                                    return true;
                                }
                            }else{//null, no unsuccessfully decoded data burst that has this acid
                            //if it is a new data burst
                            //do BLER check
                                int value = generator.nextInt(100);
                                if(value < BLER){
                                    return false;//unsuccessful
                                }
                                return true;//successful
                            }
                        }
                    }
                    /**
                     * This class represents a data burst stored in the Subscriber Station
                     * @author Yucheng
                     */
                    private class BufferedDB{
                        private int time;//the time when SS buffers this data burst
                        private int SN;  //sequence number of this data burst
                        public BufferedDB(int time, int SN){
                            this.time = time;
                            this.SN = SN;
                        }
                        /**
                         * Retrieve time
                         * @return time
                         */
                        public int getTime(){
                            return time;
                        }
                        /**
                         * Retrieve sequence number
                         * @return SN
                         */
                        public int getSN(){
                            return SN;
                        }
```

```
	}
	/**
	 * performance statistic
	 */
	public Results getThroughput(boolean MC){
		double throughput = (double)totalSDDB*bit_per_databurst/(totalTDB*0.005*1000);
		Results r = new Results(throughput, (double)totalWaitingTime*frameSize/totalSDDB);
		return r;
	}
}
```

****
****

- **Multiple-copy/Original HARQ Scheme (HARQ Process)**

```
import java.util.Iterator;
import java.util.Vector;
/**
 * This class specifies how each HARQ process operates according to information of data bursts stored
 * in the Base Station and feedbacks received from the Subscriber Station.
 * @author Yucheng
 */
public class HARQ_process {
	private final int ACID; // HARQ channel ID
	private int MC = 0;     // multiple copies indicator; 0 -> fist copy, 1 -> second or later
	private double M;       // number of multiple copies, needed for estimating M'
	private int ICN;        // data burst's initial sending channel number
	/**
	 * HARQ_process constructor
	 * @param ACID HARQ channel ID
	 * @param M initial value of M
	 */
	public HARQ_process(int ACID, int M){
		this.ACID = ACID;
		ICN = ACID;
		this.M = M;
	}
	/**
	 * Retrieve M of this process
	 * @return M number of multiple copies, for extimating M'
	 */
	public double getM(){
		return M;
	}
	/**
	 * Add info of data bursts into the queue
	 * @param queue A queue that stores unacked data bursts' info
	 * @param sn Sequence number of the first PDU in the data burst
	 * @param M M'
	 */
	public void addInfo(Vector<DataInfo> queue, int sn, double M){
		DataInfo di = new DataInfo(ACID);
		di.setSN(sn);
		MC = 0; //first copy of the data burst
		ICN = ACID;
		queue.add(di);
```

```
		if(M > di.getNS()){
			di.setStatus("T");
		}
	}
	/**
	 * Delete data burst information from the queue
	 * @param queue A queue that stores data bursts' information
	 */
	public void deleteInfo(Vector<DataInfo> queue){//only used by original SW HARQ
			for (Iterator info = queue.iterator(); info.hasNext(); ) {
				DataInfo di = (DataInfo) info.next();
				if(di.getC() == ACID){//itself
				  info.remove();
				  break;
				}
			}
	}
	public void decreaseM(){
		if(M > 1){
			M--;
		}
	}
	/**
	 * Retrieve initial channel number (ICN)
	 * @return ICN initial channel number
	 */
	public int getICN(){
		return ICN;
	}
	/**
	 * Retrieve indicator of sending multiple copy
	 * @return MC
	 */
	public int getMC(){
		return MC;
	}
	/**
	 * Update M
	 * @param m M
	 */
	public void updateM(double m){
		M = m;
	}
	/**
	 * Update parameters when receiving a NAK
	 * @param queue a queue that stores data bursts' information
	 * @param p all processes
	 */
	public void NAKupdate(Vector<DataInfo> queue, Vector<HARQ_process> p){
		if(MC == 0){// last time, first data burst sent by this HARQ process
			for (Iterator info = queue.iterator(); info.hasNext(); ) {
				DataInfo di = (DataInfo) info.next();
				if(di.getC() == ACID){//according to current process's ACID
				  di.updateNR();//+1
				  M = di.getNR() + 1;//update M of this process
```

```
                                if(di.getNS() == di.getNR()){//has received all feedback
                                    di.setStatus("R");
                                }
                                break;
                            }
                        }
                }else{ // last time, second or later data burst sent by this HARQ process
                        for (Iterator info = queue.iterator(); info.hasNext(); ) {
                                DataInfo di = (DataInfo) info.next();
                                if(di.getC() == ICN){//according to initial process's ACID
                                    di.updateNR();
                                    HARQ_process pi = p.get(ICN);
                                    pi.updateM(di.getNR() + 1);//update M of initial sending process
                                    if(di.getNS() == di.getNR()){//has received all feedback
                                        di.setStatus("R");
                                    }
                                break;
                                }
                        }
                }
}
/**
 * Update parameters when receiving a NAK (for original HARQ operation)
 * @param queue a queue that stores data bursts' information
 */
public int getOSN(Vector<DataInfo> queue){
        int seq_num = 0;
                for (Iterator q = queue.iterator(); q.hasNext(); ) {
                        DataInfo di = (DataInfo) q.next();
                        if(di.getC() == ACID){//using channel ACID to get sequence num
                                seq_num = di.getSN();
                            break;
                        }
                }
                return seq_num;
}
/**
 * Update parametes when receives a ACK feedback
 * @param queue a queue that stores data bursts' information
 * @param p initial sending channel
 */
public boolean ACKupdate(Vector<DataInfo> queue){
        boolean updateMprime = false;
        if(MC == 0){// last time, first data burst was initially sent by this HARQ process
                for (Iterator info = queue.iterator(); info.hasNext(); ) {
                        DataInfo di = (DataInfo) info.next();
                        if(di.getC() == ACID){
                                if(M != 1){
                                        M = di.getNR() + 1 ;//update M back to 1
                                        updateMprime = true;
                                }
                            info.remove();//info already removed here
                            break;
                        }
                }
        }else{ //second or later data burst sent by this process
```

```
                    for (Iterator info = queue.iterator(); info.hasNext(); ) {
                            DataInfo di = (DataInfo) info.next();
                            if(di.getC() == ICN){//based on ICN remembered by the process
                                    //M has been updated when receiving NAKs
                               info.remove();//info already removed here
                               break;
                            }
                    }
            }
            return updateMprime;
}
/**
 * Check if there is a data burst in status T (Transmitting)
 * @param queue a queue that stores data bursts' info
 * @return true, there is a unACKed data burst in status T; false, there is not
 */
public boolean checkT(Vector<DataInfo> queue){
            for (Iterator info = queue.iterator(); info.hasNext(); ) {
                        DataInfo di = (DataInfo) info.next();
                        if(di.getS().equalsIgnoreCase("T")){
                          return true;
                        }
            }
            return false;
}
/**
 * check if there is any data burst in status R (waiting to be resent)
 * @param queue a queue that stores data bursts' information
 * @return true, there is a unACKed data burst in status R; false, there is not
 */
public boolean checkR(Vector<DataInfo> queue){
     for (Iterator info = queue.iterator(); info.hasNext(); ) {
            DataInfo di = (DataInfo) info.next();
    if(di.getS().equalsIgnoreCase("R")){// the data burst in waiting to be retransmitted status
                          return true;
                        }
            }
            return false;
}
/**
 * Resend this data burst when there a data burst in status R
 * @param queue a queue that stores data bursts' information
 * @return sequence number of the burst being resent
 */
public int resendDataBurst(Vector<DataInfo> queue){
            int sn = 0;//sequence number
            for (Iterator info = queue.iterator(); info.hasNext(); ) {
                        DataInfo di = (DataInfo) info.next();
                        if(di.getS().equalsIgnoreCase("R")){
                                di.updateNS();//+ 1
                                di.setStatus("F");
                                sn = di.getSN();
                                MC = 1;
                                ICN = di.getC();
                          break;//only find the first data burst that is in status R
                        }
```

```
		}
		return sn;
	}
	/**
	 * Check if current channel has an unACKed data burst
	 * @param queue a queue that stores data bursts' info
	 * @return true, it has; false, it doesn't have
	 */
	public boolean outstandingACID(Vector<DataInfo> queue){
		for (Iterator info = queue.iterator(); info.hasNext(); ) {
			DataInfo di = (DataInfo) info.next();
			if(di.getC() == ACID){
			  return true;
			}
		}
		return false;
	}
	/**
	 * Send an extra copy of data burst when there is an outstanding ACID
	 * @param queue a queue that stores data bursts' info
	 * @return sequence number of the data burst being sent
	 */
	public int sendExtraDataBurst(Vector<DataInfo> queue){
		int sn = 0;//sequence number
		DataInfo di = (DataInfo) queue.firstElement();
		di.updateNS();// + 1
		sn = di.getSN();
		ICN = di.getC();
		MC = 1;
		//already in status F
		return sn;
	}
	/**
	 * Send a new data burst
	 * @param queue a queue that stores data bursts' information
	 * @param sn a sequence number
	 * @param M M'
	 * @return this channel's ACID
	 */
	public int sendNewDataBurst(Vector<DataInfo> queue, int sn, double M){
		addInfo(queue, sn, M);
		return ACID;
	}
	/**
	 * Simulate the situation that the max buffer has reached
	 */
	public void notSendNewDataBurst(){
		MC = 0;
		ICN = ACID;
	}
	/**
	 * Send multiple copy when there is a data burst in status T
	 * @param queue a queue that stores data bursts' information
	 * @param M_prime number of how many extra copies have to be sent
	 * @retrn first PDU's sequence number in the data burst being sent
	 */
```

```
	public int sendMC(Vector<DataInfo> queue, double M_prime){
		int sn = 0;//sequence number
		for (Iterator info = queue.iterator(); info.hasNext(); ) {
		  DataInfo di = (DataInfo) info.next();
		  if(di.getS().equalsIgnoreCase("T")){//check if a data burst in waiting to be retransmitted
			MC = 1;//second or later copies
			ICN = di.getC();//give ICN of the data burst to current process
			sn = di.getSN();
			di.updateNS();// + 1;
			if(di.getNS() < M_prime){
				di.setStatus("T");//transmit the same data burst on the next process
			}else{
				di.setStatus("F");//set status as waiting for feedback
			 }
			break;
		  }
		}
		return sn;
	}
}
```

****
****

- **Multiple-copy/Original HARQ Scheme (unACKed Data Burst)**

```
/**
 * This class specifies information of unACKed data bursts that are stored in the Base Station.
 * HARQ processes refer to this information to perform transmissions/retransmission.
 * @author Yucheng
 */
public class DataInfo {
	private final int C;  //initial sending channel's ACID
	private int NR = 0;   //number of NAKs received
	private int NS = 1;   //number of data burst copy sent
	private String S = "F";   //status of the data burst; T, F(initially), or R
	private int SN;  //sequence number of the first PDU in the data burst for recordering in the
Subscriber Station
	/**
	 * DataInfo calss constructor
	 * @param ACID initial sending channel
	 */
	public DataInfo(int ACID){
		C = ACID;
	}
	/**
	 * Set sequence number of first PDU
	 * @param sn sequence number
	 */
	public void setSN(int sn){
		SN = sn;
	}
	/**
	 * Retrieve sequence number of first PDU
	 * @return
	 */
	public int getSN(){
```

```
		return SN;
}
/**
 * Set status of the data burst
 * @param status status of the data burst
 */
public void setStatus(String status){
		S = status;
}
/**
 * Update number of data burst copy sent by one
 */
public void updateNS(){
		NS += 1;
}
/**
 * Update number of NAK received by one
 */
public void updateNR(){
		NR += 1;
}
/**
 * Check if NS equals NR
 * @return true, equal; false, unequal
 */
public boolean NSEqualsNR(){
		if(NS == NR){
			return true;
		}else{
			return false;
		}
}
/**
 * Retrieve status of the data burst
 * @return status
 */
public String getS(){
		return S;
}
/**
 * Retrieve NS
 * @return NS
 */
public int getNS(){
		return NS;
}
/**
 * Retrieve NR
 * @return NR
 */
public int getNR(){
		return NR;
}
/**
 * Retrieve C
 * @return C
```

```
	 */
	public int getC(){
		return C;
	}
}
```

****
****

- **Multiple-copy/Original HARQ Scheme (Simulation Results)**

```
/**
 * This class stores simulation results for final display
 * @author Yucheng
 */
public class Results {
	private double throughput;
	private double average_waiting_time;
	public Results(double t, double awt){
		throughput = t;
		average_waiting_time = awt;
	}
	/**
	 * Retrieve value of throughput
	 * @return throughput
	 */
	public double getTE(){
		return throughput;
	}
	/**
	 * Retrieve value of average waiting time
	 * @return average waiting time
	 */
	public double getAWT(){
		return average_waiting_time;
	}
}
```

****
****

- **TCL source code**

```
# Topology:
#
# h0 ---------- h1
#

cd [mkdir -q drcl.comp.Component /test]

# create the topology
puts "create topology..."
set link [java::new drcl.inet.Link]
$link setPropDelay 0.002; # ms
set adjMatrix [java::new {int[][]} 2 {{1} {0}}]
java::call drcl.inet.InetUtil createTopology [! .] $adjMatrix $link
```

```
puts "create node builder..."
# NodeBuilder:
set nb [mkdir drcl.inet.NodeBuilder .nodeBuilder]
$nb setBandwidth 1.0e7; #10Mbps

puts "build..."
$nb build [! h0] {
	udp                                drcl.inet.transport.UDP
	transmitter	101/udp	Transmitter
}

$nb build [! h1] {
	udp                                drcl.inet.transport.UDP
	receiver 101/udp Receiver
}

! h?/udp setTTL 3

puts "set up simulator..."
set sim [attach_simulator .]

puts "Done!"

*****************************************************************************************
*****************************************************************************************

import java.util.*;
	/**
	 * A acknowledgement message sent from the receiver.
	 */
	public class Acknowledgement{
		public LinkedList bitmap;
		public boolean ack = true;
		public int next_expected_packet;
		public double time;
		public boolean done = false;
		public int sequence_num;
		public boolean forLastOne = false;
		public boolean next;

		public Acknowledgement(){
			bitmap = new LinkedList();
		}
		/**
		 * Set next expected number.
		 * @param next a number.
		 */
		public void next(int next){
			next_expected_packet = next;
		}
		/**
		 * Set a bitmap in the acknowledgement packet.
		 * @param bitmap a bitmap.
		 */
		public void setBitmap(LinkedList bitmap){
```

```
                this.bitmap = (LinkedList) bitmap.clone();
        }
        /**
         * Set sequence Number.
         * @param num sequence number.
         */
        public void setNum(int num){
                sequence_num = num;
        }
}
/**
 * An ARQ block.
 */

public class ARQ_Block{
        public int sequence_num = 0;
        public boolean isLast = false;

        public ARQ_Block(){
        }
        /**
         * Set packet's sequence number.
         *@param num sequence number.
         /
        public void setNum(int num){
                sequence_num = num;
        }
        /**
         * Set this packet as a last packet sent.
         */
        public void setLast(){
                isLast = true;
        }
        /**
         * Unset this packet as a last packet sent.
         */
        public void unSet(){
                isLast = false;
        }
}
```

****
****

- **Efficient Selective-Repeat ARQ Scheme**

o **Transmitter**

```
import drcl.comp.Port;
import drcl.comp.Contract;
import java.util.*;
/**
 *  Implementing the Efficient Selective-Repeat ARQ Scheme, transmitter side.
 */
public class Transmitter extends drcl.inet.application.SUDPApplication{
        long dst;
```

```
int dport;
int packet_num;
LinkedList buffer = new LinkedList();
int window_size = 64;
int max_sequence_num = 2*window_size + 2;
int current_sequence_num = 0;
LinkedList bitmap = new LinkedList();
int resend_num = 0;
double start_time;
int total_packets;
boolean stop = false;

public Transmitter() {
	super();
	dst = 1;
	dport = 101;
}
/**
 * Start the program.
 * @param total_packets total packets needed for calculating throughput.
 */
public void start(int total_packets){
	this.total_packets = total_packets;
	start_time = getTime();
	send_packets(dst, dport);
}
/**
 * Send a window-size packets to the receiver.
 * @param dst_ the receiver.
 * @param dport_ the port at the receiver.
 */
public void send_packets(long dst_, int dport_){
	for(int i=0; i<window_size; i++){
		ARQ_Block p = new ARQ_Block();
		p.setNum(getSequenceNum());
		if(i == window_size-1){
			p.setLast();
			ARQ_Block p1 = new ARQ_Block();
			p1.setNum(p.sequence_num);
			buffer.add(p1);
		}else{
			buffer.add(p);
		}
		sendmsg(p , 512/*size*/, dst_, dport_);
	}
}
/**
 * Implement the protocol when an acknowledgement arrives.
 */
protected synchronized void dataArriveAtDownPort(Object data_, Port downPort_){
	LinkedList temp = new LinkedList();
	long src_ = getPeerAddress(data_);
	int sport_ = getPeerPort(data_);
	Acknowledgement pkt_ = (Acknowledgement) getContent(data_);
	if (pkt_.done == 0 && !stop){
		if(pkt_.ack){
```

```
                        bitmap = pkt_.bitmap;
                        if(bitmap.size()!=0){

                                resend(pkt_.next_expected_packet,src_,sport_, temp);
                        }else{
                                if(pkt_.next_expected_packet!=current_sequence_num){
                                resend(pkt_.next_expected_packet,src_,sport_, temp);
                                }
                        }
                                for(int i = 0; i < window_size - bitmap.size(); i++){
                                        buffer.removeFirst();
                                }
                        for(int i = 0; i < bitmap.size(); i++){
                                String value = (String) bitmap.get(i);
                                if(value.equals("0")){
                                        ARQ_Block packet = (ARQ_Block) buffer.get(i);
                                        resend_num++;
                                        if(resend_num == 64){
                                                packet.setLast();
                                        }
                                        sendmsg(packet, 512, src_, sport_);
                                        if(resend_num == 64){
                                                ARQ_Block p1 = new ARQ_Block();
                                                p1.setNum(packet.sequence_num);
                                                temp.add(p1);
                                        }else{
                                        temp.add(packet);}
                                }
                        }
                        int new_packet_num = window_size - resend_num;
                        for(int i=0; i<new_packet_num; i++){
                                ARQ_Block packet = new ARQ_Block();
                                int temp1 = getSequenceNum();
                                packet.setNum(temp1);
                                if(i == new_packet_num -1){
                                        packet.setLast();
                                        sendmsg(packet , 512/*size*/, src_, sport_);
                                        ARQ_Block p1 = new ARQ_Block();
                                        p1.setNum(temp1);
                                        temp.add(p1);
                                }else{
                                        sendmsg(packet , 512/*size*/, src_, sport_);
                                        packet.unSet();
                                        temp.add(packet);
                                }
                        }
                        }
                        buffer = temp;
                        resend_num=0;
                }else{
                        if(pkt_.done==1){
                        stop = true;
                        System.out.println("Time spent: "+pkt_.time);
                        System.out.println("Througput of sending "+ total_packets +" packets is "+
total_packets/pkt_.time+" packets/s");
                        }
```

```
                }
        }
        /**
         * Resend packets to the receiver.
         * @param block block size.
         * @param num   current sequence number.
         * @param src_  the receiver.
         * @param sport_ the port at the receiver.
         */
        private void resend(int num, long src_, int sport_, LinkedList temp){
                ARQ_Block packet = new ARQ_Block();
                packet.setNum(num);
                sendmsg(packet, 512, src_, sport_);
                resend_num++;
                temp.add(packet);
        }
        /**
         * Assign sequence number.
         */
        private int getSequenceNum(){
                if(current_sequence_num <= max_sequence_num - 1 ){
                        return current_sequence_num++ ;
                }else{
                        current_sequence_num = current_sequence_num - max_sequence_num;
                         return current_sequence_num++;
                }
        }
}
```

- **Receiver**

```
import drcl.comp.Port;
import drcl.comp.Contract;
import java.util.*;

/**
 * Implementing the Efficient Selective-Repeat ARQ Scheme, receiver side.
 */
public class Receiver extends drcl.inet.application.SUDPApplication{
        long dst = 0;
        int dport = 101;
        LinkedList buffer = new LinkedList();
        int window_size = 64;
        int next_expected_num = 0;
        LinkedList bitmap = new LinkedList();
        boolean firstError=true;
        boolean normalState=true;
        int total_packets = 0;
        int total_packets_;
        boolean stop = false;

        public Receiver() {
                super();
        }
        /**
         * Implement the protocol when a packet arrives.
```

```
 * @param data_ a packet
 * @param downPort_ a port
 */
protected synchronized void dataArriveAtDownPort(Object data_, Port downPort_){
	long src_ = getPeerAddress(data_);
	int sport_ = getPeerPort(data_);
	ARQ_Block pkt_ = (ARQ_Block)getContent(data_);
	boolean b = isError();
	if(!stop){
	if (!b/*!isError()*/ && forwardDistanceOK(pkt_)){
		if(normalState){
			addToBuffer(pkt_);
			if(pkt_.isLast){
				sendAckBack();
			}
		}else{
			addToBuffer(pkt_);
			bitmap.add("1");
			if(pkt_.isLast){
				sendAckBack();
			}
		}
	}else if(firstError){
		firstError=false;
		normalState=false;
		if(pkt_.isLast){
			sendAckBack();
		}
	}else{
		bitmap.add("0");
		if(pkt_.isLast){
			sendAckBack();
		}
	  }
	}
}
/**
 * Add received packets into the buffer or send to the upper layer.
 * @param pkt_ a packet.
 */
public void addToBuffer(ARQ_Block pkt_){
	if(pkt_.sequence_num == next_expected_num){
		if(buffer.size()==0){
			next_expected_num++;
			sequenceCycle();
			finish();
		}else{
			finish();
			next_expected_num++;
			sequenceCycle();
			int b = buffer.size();
			for(int i = 0; i<b;i++){
				int b2 = buffer.size();
				int next = next_expected_num;
				for(int j=0; j<b2; j++){
					ARQ_Block temp = (ARQ_Block) buffer.get(j);
```

```
                                        if(temp.sequence_num == next){
                                                finish();
                                                buffer.remove(j);
                                                next_expected_num++;
                                                sequenceCycle();
                                                break;
                                        }
                                }
                        }
                }
        }else{
                buffer.add(pkt_);
        }
}
/**
 * Determine if this packet has error based on bit error rate.
 */
private boolean isError(){
        Random r = new Random();
        int i = r.nextInt(100000000) + 1;
        if(i > 540*8*10000){
                return false;
        }else{
                return true;
        }
}
/**
 * Send acknowledgement containing a bitmap to the receiver.
 */
private Acknowledgement sendAck(){
        Acknowledgement ack = new Acknowledgement();
        ack.next(next_expected_num);
        ack.setBitmap(bitmap);
        return ack;
}
/**
 * Make sequence number.
 */
private void sequenceCycle(){
        if(next_expected_num == (64*2 + 2)){
                next_expected_num = 0;
        }
}
/**
 * Calcuate the forward distance of received packet.
 * @param p a packet
 * @return  ture:ok to store the packet; false:otherwise.
 */
private boolean forwardDistanceOK(ARQ_Block p){
        int distance = p.sequence_num - next_expected_num;
        if( distance > 0){
                if(distance < window_size){
                        return true;
                }
                        return false;
        }else if(distance <0){
```

```
                        int newDistance = distance + (2*window_size + 2);
                        if(newDistance < window_size){
                                return true;
                        }
                                return false;
                }
                return true;
        }
        /**
         * Set totoal numbr of packets, for calculating throughput purpose.
         * @param value number of packets
         */
        public void setPacketNum(int value){
                total_packets_ = value;
        }
        /**
         * Check if the totoal numbers of packets sent have reached.
         */
        private void finish(){
                total_packets += 1;
                if(total_packets == total_packets_){
                        stop = true;
                        Acknowledgement ack = new Acknowledgement();
                        ack.done = 1;
                        ack.time = getTime();
                        sendmsg(ack, 10, 0, 101);}
        }
        /**
         * Send acknowledgement back to the transmitter.
         */
        private void sendAckBack(){
                firstError=true;
                normalState=true;
                sendmsg(sendAck(), 10, dst, dport);
                bitmap.clear();
        }

}
```

****

- **Variant of Optimum Go-Back-N ARQ Strategy by Bruneel and Moeneclaey**

o **Transmitte**r

```
import drcl.comp.Port;
import drcl.comp.Contract;
import java.util.*;
/**
 * Implementing the Variant of Optimum Go-Back-N ARQ Scheme, transmitter side.
 */
public class Transmitter extends drcl.inet.application.SUDPApplication{
        long dst;
        int dport;
        int packet_num = 0;
        LinkedList buffer = new LinkedList();
```

```
	int window_size = 64;
	int number = 0;
	double start_time;
	int total_packets;
	boolean stop = false;
	int K = 64;
	int numberOfCopy = 1;
	boolean continue_ = true;

	public Transmitter() {
		super();
		dst = 1;
		dport = 101;
	}
	/**
	 * Start the program.
	 * @param total_packets total packets needed for calculating throughput.
	 */
	public void start(int total_packets){
		start_time = getTime();
		this.total_packets = total_packets;
		send_packets(dst, dport);
	}
	/**
	 * Send a window-size packets to the receiver.
	 * @param dst_ the receiver.
	 * @param dport_ the port at the receiver.
	 */
	public void send_packets(long dst_, int dport_){
		for(int i=0; i<window_size; i++){
		ARQ_Block p = new ARQ_Block();
		p.setNum(i);
		number++;
		if(i== window_size -1){
			p.isLast = true;
		}
		sendmsg(p , 512/*size*/, dst_, dport_);
		}
	}
	/**
	 * Implement the protocol when an acknowledgement arrives.
	 */
	protected synchronized void dataArriveAtDownPort(Object data_, Port downPort_){
		long src_ = getPeerAddress(data_);
		int sport_ = getPeerPort(data_);
		Acknowledgement ack_ = (Acknowledgement) getContent(data_);
		if(ack_.ack && !stop){
			if(ack_.forLastOne == true){
				stop = true;
				double time_spent = ack_.time - start_time;
				System.out.println("time spent: "+ time_spent);
				System.out.println("Througput of sending "+ total_packets +" packets
is "+ total_packets/ack_.time +" packets/s");
			}else if(ack_.next && !stop){
				adjust_copy(ack_.sequence_num);
				retransmit(ack_.sequence_num + 1);
```

```
                }
        }else{
                if(!stop){
                        continue_ = false;
                        if(ack_.sequence_num!= 0){adjust_copy(ack_.sequence_num-1);}
                        else{adjust_copy(ack_.sequence_num);}
                        continue_ = true;
                        numberOfCopy += 1;
                        retransmit(ack_.sequence_num);
                }
        }
}
/**
 * Retransmit a window-sized packets.
 * @param num a packet sequence number.
 */
private void retransmit(int num){
        number = num;
        for(int i = 0; i < window_size; i++){ //send another block-size packets
                for(int j=0; j<numberOfCopy; j++){
                        ARQ_Block p = new ARQ_Block();
                        p.setNum(number + i);//from next expected number;
                        int temp = number +i;
                        if(i == (window_size -1) && j == (numberOfCopy -1)){
                        p.isLast = true;
                        }
                        sendmsg(p , 512/*size*/, dst, dport);
                }
        }
}
/**
 * Adjust number of packet's copies.
 */
private void adjust_copy(int seqNum){
        if(seqNum == 0) return;
        int temp = seqNum - packet_num;
        while(true){
                int temp2 = temp - generate_m();
                if(temp2 >= 0){
                        packet_num += generate_m();
                        numberOfCopy -= 1;
                        if(numberOfCopy == 0){ numberOfCopy = 1;}
                        temp = temp2;
                }else{
                        if(!continue_){
                        packet_num = seqNum;}
                        break;
                }
        }
}
/**
 * Generate number M to decide if increasing number of copy is needed.
 */
private int generate_m(){
        if(numberOfCopy - 1 == 0){
                return 1;
```

```
		}else{
			return (int) Math.ceil(K/(numberOfCopy -1 ));
		}
	}
}
```

- **Receiver**

```
import drcl.comp.Port;
import drcl.comp.Contract;
import java.util.*;

/**
 *  Implementing the Variant of Optimum Go-Back-N ARQ Scheme, receiver side.
 */
public class Receiver extends drcl.inet.application.SUDPApplication{
	long dst = 0;
	int dport = 101;
	int pkt_num;
	LinkedList buffer = new LinkedList();
	int next_expected_num = 0;
	boolean normalState=true;
	LinkedList ACK = new LinkedList();
	int total_packets = 0;
	int total_packets_ = 1000;
	int first_neg_seq = -1;
	boolean stop = false;

	public Receiver(){
		super();
	}
	/**
	 * Implement the protocol when a packet arrives.
	 * @param data_ a packet
	 * @param downPort_ a port
	 */
	protected synchronized void dataArriveAtDownPort(Object data_, Port downPort_){
		long src_ = getPeerAddress(data_);
		int sport_ = getPeerPort(data_);
		ARQ_Block pkt_ = (ARQ_Block) getContent(data_);
		boolean b = isError();
		if(!stop){
		if (!b){
			if(addToBuffer(pkt_)){
				finish(pkt_.sequence_num);
			if(pkt_.isLast && !stop){
				Acknowledgement ack = new Acknowledgement();
				ack.setNum(pkt_.sequence_num);
				ack.time = getTime();
				ack.next = true;
				sendmsg(ack, 10/*size*/, src_, sport_);
			}
			}else{
				if(!normalState && pkt_.isLast){
					sendAck();
				}
				if(normalState && pkt_.isLast){//duplicate no-error packet
```

```
                                        Acknowledgement ack = new Acknowledgement();
                                        ack.setNum(pkt_.sequence_num);
                                        ack.time = getTime();
                                        ack.next = true;
                                        sendmsg(ack, 10/*size*/, src_, sport_);
                                }
                        }
                }else{
                        if(normalState && pkt_.sequence_num != (next_expected_num-1)){
                                normalState = false;
                                first_neg_seq = pkt_.sequence_num;
                        }
                        if(normalState && pkt_.sequence_num == (next_expected_num-1)){
                                if(pkt_.isLast){
                                        Acknowledgement ack = new Acknowledgement();
                                        ack.setNum(pkt_.sequence_num);
                                        ack.time = getTime();
                                        ack.next = true;
                                        sendmsg(ack, 10/*size*/, src_, sport_);
                                }
                        }
                        if(!normalState && pkt_.isLast){
                                sendAck();
                        }

                }
                }
        }
        /**
         * Add received packets into the buffer and send to the upper layer.
         * @param pkt_ a packet.
         */
        public boolean addToBuffer(ARQ_Block pkt_){
                if(pkt_.sequence_num == next_expected_num){
                        next_expected_num++;
                        normalState = true;
                        total_packets += 1;
                        return true;
                }else{
                        return false;
                }
        }
        /**
         * Determine if this packet has error based on bit error rate.
         */
        private boolean isError(){
                Random r = new Random();
                int i = r.nextInt(100000000) + 1;
                if(i > 540*8*10){
                        return false;
                }else{
                        return true;
                }
        }
        /**
         * Send ack when all window-size ARQ blocks are received.
```

```
	 */
	private void sendAck(){
		Acknowledgement ack = new Acknowledgement();
		ack.setNum(first_neg_seq);
		ack.setAck(false);
		sendmsg(ack, 10, dst, dport);
	}
	/**
	 * Check if required number of packets is reached
	 */
	private void finish(int lastOne){
		if(total_packets == total_packets_){
			System.out.println("send ack back to notify");
			stop = true;
			Acknowledgement ack = new Acknowledgement();
			ack.time = getTime();
			ack.setNum(lastOne);
			ack.forLastOne = true;
			sendmsg(ack, 10/*size*/, dst, dport);
			}
	}

}
```

****

****

- **Selective Repeat SW-ARQ Scheme**

o **Transmitter**

```
import drcl.comp.Port;
import drcl.comp.Contract;
import java.util.*;
/**
 * Implementing the Selective Repeat SW ARQ scheme, transmitter side.
 */
public class Transmitter extends drcl.inet.application.SUDPApplication{
	long dst;
	int dport;
	int window_size = 64;
	int number = 0;
	LinkedList bitmap = new LinkedList();
	int total_packets;
	double start_time;
	double time_spent;
	boolean stop = false;

	public Transmitter(){
		super();
		dst = 1;
		dport = 101;
	}
	/*
	 * Start the program.
	 * @param run total runs needed for calculating throughput.
```

```
	 */
	public void start(int total_packets){
		this.total_packets = total_packets;
		start_time = getTime();
		send_packets(dst, dport);
	}
	/*
	 * Send a window-sized packets to the receiver.
	 */
	public void send_packets(long dst_, int dport_){
		for(int i=0; i<window_size; i++){
			ARQ_Block p = new ARQ_Block();
			p.setNum(number);
			number += 1;
			if(i == window_size - 1){
				p.setLast();
			}
		   sendmsg(p , 512/*size*/, dst_, dport_);
		}
	}
	/*
	 * Implement the protocol when an acknowledgement arrives.
	 */
	protected synchronized void dataArriveAtDownPort(Object data_, Port downPort_){
		long src_ = getPeerAddress(data_);
		int sport_ = getPeerPort(data_);
		Acknowledgement pkt_ = (Acknowledgement) getContent(data_);
			bitmap = pkt_.bitmap;
			if(!stop){
			if(pkt_.ack){
				number = 0;
				if(pkt_.forLastOne == true){
					stop = true;
					time_spent = pkt_.time - start_time;
					System.out.println("Time spent: " + time_spent);
					System.out.println("Througput of sending "+ total_packets +"
packets is "+ total_packets/time_spent+" packets/s");
				}else{
					send_packets(dst, dport);
				}
			}else {
				for(int i=0; i<bitmap.size(); i++){
					int m = ((Integer)(bitmap.get(i))).intValue();
					ARQ_Block p = new ARQ_Block();
					p.setNum(m);
					if(i == bitmap.size()-1){
						p.isLast = true;
					}
					sendmsg(p, 512, src_, sport_);
				}
			}
		}
	}
}
```

- **Receiver**

```
import drcl.comp.Port;
import drcl.comp.Contract;
import java.util.*;

/**
 * Implementing the Selective Repeat SW ARQ scheme, receiver side.
 */
public class Receiver extends drcl.inet.application.SUDPApplication{
	long dst = 0;
	int dport = 101;
	LinkedList buffer = new LinkedList();
	int next_expected_num = 0;
   LinkedList bitmap = new LinkedList();
	boolean normalState = true;
	int total_packets = 0;
	int total_packets_ = 1000;
	boolean stop = false;

	public Receiver() {
		super();
	}
	/*
	 * Implement the protocol when a packet arrives.
	 * @param data_ a packet
	 * @param downPort_ a port
	 */
	protected synchronized void dataArriveAtDownPort(Object data_, Port downPort_){
		long src_ = getPeerAddress(data_);
		int sport_ = getPeerPort(data_);
		ARQ_Block pkt_ = (ARQ_Block)getContent(data_);
		boolean b = isError();
		if(!stop){
		if (!b/*!isError()*/){
			if(normalState){
				addToBuffer(pkt_);
				if(pkt_.isLast){
					for(int i=0; i<bitmap.size(); i++){
						int m = ((Integer)(bitmap.get(i))).intValue();
					}
					next_expected_num=0;
					Acknowledgement ack = new Acknowledgement();
					ack.ack = true;
					ack.setBitmap(bitmap);
					ack.time = getTime();
					sendmsg(ack, 10/*size*/, src_, sport_);
					bitmap.clear();

				}
			}else{
				if(pkt_.sequence_num == next_expected_num){
					normalState = true;
				}
				addToBuffer(pkt_);
				if(pkt_.isLast){
					if(normalState){
```

```
                                        next_expected_num = 0;
                                        Acknowledgement ack = new Acknowledgement();
                                        ack.ack = true;
                                        ack.setBitmap(bitmap);
                                        ack.time = getTime();
                                        sendmsg(ack, 10/*size*/, src_, sport_);
                                        bitmap.clear();
                                }else{
                                        for(int i=0; i<bitmap.size(); i++){
                                                int m = ((Integer)(bitmap.get(i))).intValue();
                                        }
                                        Acknowledgement ack = new Acknowledgement();
                                        ack.ack = false;
                                        ack.setBitmap(bitmap);
                                        sendmsg(ack, 10, src_, sport_);
                                        bitmap.clear();
                                }
                        }
                }
        }else{
                normalState = false;
                bitmap.add(new Integer(pkt_.sequence_num));
                if(pkt_.isLast){
                        for(int i=0; i<bitmap.size(); i++){
                                int m = ((Integer)(bitmap.get(i))).intValue();
                        }
                        Acknowledgement ack = new Acknowledgement();
                        ack.ack = false;
                        ack.setBitmap(bitmap);
                        sendmsg(ack, 10, src_, sport_);
                        bitmap.clear();
                }
        }
  }
}
/*
 * Add received packets into the buffer or send to the uppper layer.
 * @param pkt_ a packet.
 */
public void addToBuffer(ARQ_Block pkt_){
        if(pkt_.sequence_num == next_expected_num){
                if(buffer.size()==0){
                        total_packets += 1;
                        finish(pkt_.sequence_num);
                        next_expected_num++;
                }else{
                        total_packets += 1;
                        finish(pkt_.sequence_num);
                        next_expected_num++;
                        int b = buffer.size();
                        for(int i = 0; i<b;i++){
                                int b2 = buffer.size();
                                int next = next_expected_num;
                                for(int j=0; j<b2; j++){
                                        ARQ_Block temp = (ARQ_Block) buffer.get(j);
                                        if(temp.sequence_num == next){
```

```
                                                    total_packets += 1;
                                                    finish(temp.sequence_num);
                                                    buffer.remove(j);
                                                    next_expected_num++;
                                                    break;
                                                }
                                            }
                                        }
                                    }
                                }else{
                                    buffer.add(pkt_);
                                }
                        }
                        /*
                         * Determine if this packet has error based on bit error rate.
                         */
                        private boolean isError(){
                                Random r = new Random();
                                int i = r.nextInt(100000000) + 1;
                                if(i > 540*8*10000){
                                        return false;
                                }else{
                                        return true;
                                }
                        }
                        /*
                         * Send acknowledgement containing a bitmap to the receiver.
                         */
                        private Acknowledgement sendAck(){
                                Acknowledgement ack = new Acknowledgement();
                                ack.setBitmap(bitmap);
                                return ack;
                        }
                        /**
                         * Check if required number of packets is reached.
                         */
                        private void finish(int lastOne){
                                if(total_packets == total_packets_){
                                        stop = true;
                                        Acknowledgement ack = new Acknowledgement();
                                        ack.ack = true;
                                        ack.time = getTime();
                                        ack.setNum(lastOne);
                                        ack.forLastOne = true;
                                        sendmsg(ack, 10/*size*/, dst, dport);
                                        }
                        }
}
```

****

- **Block Window Retransmission ARQ Scheme**

o **Transmitte**r

```
import drcl.comp.Port;
import drcl.comp.Contract;
import java.util.*;
/**
 * Implementing the Block Window Retransmission ARQ scheme, transmitter side.
 */
public class Transmitter extends drcl.inet.application.SUDPApplication{
	long dst;
	int dport;
	LinkedList buffer = new LinkedList();
	int window_size = 64;
	int max_sequence_num = 64;
	int current_sequence_num = 0;
	LinkedList bitmap = new LinkedList();
	int resend_num = 0;
	int b1 = (window_size /*+ 1*/)/4;
	int b2 = (window_size /*+ 1*/)/8;
	int b3 = (window_size /*+ 1*/)/8;
	int b4 = (window_size /*+ 1*/)/8;
	int b5 = (window_size /*+ 1*/)/8;
	int b6 = ((window_size /*+ 1*/)/4)/* - 1*/;
	boolean final_block = false;
	double start_time;
	int total_packets;
	boolean stop = false;

	public Transmitter() {
		super();
		dst = 1;
		dport = 101;
	}
	/**
	 * Start the program.
	 * @param total_packets total packets needed for calculating throughput.
	 */
	public void start(int total_packets){
		this.total_packets = total_packets;
		start_time = getTime();
		send_packets(dst, dport);
	}
	/**
	 * Send a window-size packets to the receiver.
	 */
	public void send_packets(long dst_, int dport_){
		for(int i = 0; i < window_size; i++){
			ARQ_Block p = new ARQ_Block();
			p.setNum(getSequenceNum());
			buffer.add(p);
			if(i == window_size - 1){
				p.setLast();
			}
			sendmsg(p , 512, dst_, dport_);
		}
	}
	/*
	 * Implement the protocol when a acknowledgement arrives.
```

```
	 */
	protected synchronized void dataArriveAtDownPort(Object data_, Port downPort_){
		long src_ = getPeerAddress(data_);
		int sport_ = getPeerPort(data_);
		Acknowledgement pkt_ = (Acknowledgement) getContent(data_);
		if (pkt_.done == 0 && !stop){
			bitmap = pkt_.bitmap;
			resend(b1, pkt_.next_expected_packet,src_,sport_);
			for(int i = 0; i < bitmap.size(); i++){
				String value = (String) bitmap.get(i);
				if(value.equals("0")){
					if(i==0){
						resend(b2 , pkt_.next_expected_packet +
b1 ,src_,sport_);
					}else if(i==1){
						resend(b3 , pkt_.next_expected_packet + b1 +
b2 ,src_,sport_);
					}else if(i==2){
						resend(b4 , pkt_.next_expected_packet + b1 + b2 +
b3 ,src_,sport_);
					}else if(i==3){
						resend(b5 , pkt_.next_expected_packet + b1 + b2 +
b3 + b4 ,src_,sport_);
					}
				}
			}
			final_block = true;
			resend(b6, pkt_.next_expected_packet + b1 + b2 + b3 + b4 + b5, src_, sport_);
			resend_num=0;
		}else if(pkt_.done == 1){
			stop = true;
			double time_spent = pkt_.time - start_time;
			System.out.println("time spent: "+ time_spent);
			System.out.println("Througput of sending "+ total_packets +" packets is "+
total_packets/time_spent+" packets/s");
		}
	 }
	/**
	 * Resend packets to the receiver.
	 * @param block block size.
	 * @param num   current sequence number.
	 * @param src_  the receiver.
	 * @param sport_ the port at the receiver.
	 */
	private void resend(int block, int num, long src_, int sport_){
		int num_ = num % (window_size +1)/*16*/;
		for(int i = 0; i < block; i++){
			ARQ_Block packet = new ARQ_Block();
			if(final_block && i == block -1){
				packet.isLast = true;
				final_block = false;
			}
			packet.setNum(num_++);
			if(num_> window_size/*15*/){
				num_ -= (window_size + 1)/*16*/;
			}
```

```
                        sendmsg(packet, 512, src_, sport_);
                }
        }
        /**
         * Assign sequence number.
         */
        private int getSequenceNum(){
                if(current_sequence_num < max_sequence_num){
                        return current_sequence_num++ ;
                }else{
                        current_sequence_num = current_sequence_num - window_size/*15*/;
                         return current_sequence_num++;
                }
        }
}
```

- **Receiver**

```
import drcl.comp.Port;
import drcl.comp.Contract;
import java.util.*;

/**
 * Implementing the Block Window Retransmission ARQ scheme, receiver side.
 */
public class Receiver extends drcl.inet.application.SUDPApplication{
        long dst = 0;
        int dport = 101;
        int pkt_num;
        LinkedList buffer = new LinkedList();
        int window_size = 64;
        int max_sequence_num = 15;
        int next_expected_num = 0;
        LinkedList bitmap = new LinkedList();
        boolean firstError = true;
        boolean normalState = true;
        LinkedList bitmapSend = new LinkedList();
        int b1 = (window_size /*+ 1*/)/4;
        int b2 = (window_size /*+ 1*/)/8;
        int b3 = (window_size /*+ 1*/)/8;
        int b4 = (window_size /*+ 1*/)/8;
        int b5 = (window_size /*+ 1*/)/8;
        int b6 = ((window_size /*+ 1*/)/4)/* - 1*/;
        LinkedList sentToUpper = new LinkedList();
        int packets_sent = 0;
        int total_packets;
        boolean stop = false;

        public Receiver() {
                super();
        }
        /**
         * Implement the protocol when a packet arrives.
         * @param data_ a packet
         * @param downPort_ a port
         */
```

```
protected synchronized void dataArriveAtDownPort(Object data_, Port downPort_){
	long src_ = getPeerAddress(data_);
	int sport_ = getPeerPort(data_);
	ARQ_Block pkt_ = (ARQ_Block)getContent(data_);
	boolean b = isError();
	if(!stop){
	if (!b){
		if(normalState){
			addToBuffer(pkt_);
			if(pkt_.isLast){
				sendAckBack();
			}
		}else{
			addToBuffer(pkt_);
			bitmap.add("1");
			if(pkt_.isLast){
				sendAckBack();
			}
		}
	}else if(firstError){
		firstError = false;
		normalState = false;
		if(pkt_.isLast){
			sendAckBack();
		}
	}else{
		bitmap.add("0");
		if(pkt_.isLast){
			sendAckBack();
		}
	}
  }
}
/**
 * Add received packets into the buffer or send to the uppper layer.
 * @param pkt_ a packet.
 */
public void addToBuffer(ARQ_Block pkt_){
	if(pkt_.sequence_num == next_expected_num){
		if(buffer.size()==0){
			packets_sent +=1;//records total packets sent to upper layer
			finish(pkt_.sequence_num);
			next_expected_num++;
			sequenceCycle();
			sentToUpper.add(new Integer(pkt_.sequence_num));
		}else{
			packets_sent +=1;
			finish(pkt_.sequence_num);
			next_expected_num++;
			sequenceCycle();
			int b = buffer.size();
			for(int i = 0; i<b;i++){
				int b2 = buffer.size();
				int next = next_expected_num;
				for(int j=0; j<b2; j++){
					ARQ_Block temp = (ARQ_Block) buffer.get(j);
```

```
                                        if(temp.sequence_num == next){
                                                buffer.remove(j);
                                                sentToUpper.add(new Integer(next));
                                                packets_sent +=1;
                                                finish(temp.sequence_num);
                                                next_expected_num++;
                                                sequenceCycle();
                                                break;
                                        }
                                }
                        }
                }
        }else{
                boolean packet_exist = false;
                for(int i = 0; i < buffer.size(); i++){
                        ARQ_Block p = (ARQ_Block)buffer.get(i);
                        if(p.sequence_num == pkt_.sequence_num){
                                packet_exist = true;
                        }
                }
                for(int i = 0; i < sentToUpper.size(); i++){
                        int n = ((Integer) sentToUpper.get(i)).intValue();
                        if(n==pkt_.sequence_num){
                                packet_exist = true;
                        }
                }
                if(!packet_exist){
                        buffer.add(pkt_);
                }
        }
}

/**
 * Determine if this packet has error based on bit error rate.
 */
private boolean isError(){
        Random r = new Random();
        int i = r.nextInt(100000000) + 1;
        if(i > 540*8*100000){
                return false;
        }else{
                return true;
        }
}

/**
 * Send acknowledgement containing a bitmap to the receiver.
 */
private Acknowledgement sendAck(){
        Acknowledgement ack = new Acknowledgement();
        ack.next(next_expected_num);
        ack.setBitmap(bitmapSend);
        return ack;
}
/**
 * Construct next expected window that will be received.
```

```
 */
private void constructNextWindow(){
	if(addBitmap(b2, next_expected_num + b1)){
		bitmapSend.add("1");
	}else{
		bitmapSend.add("0");
	}
	if(addBitmap(b3, next_expected_num + b1+b2)){
		bitmapSend.add("1");
	}else{
		bitmapSend.add("0");
	}
	if(addBitmap(b4, next_expected_num + b1+b2+b3)){
		bitmapSend.add("1");
	}else{
		bitmapSend.add("0");
	}
	if(addBitmap(b5, next_expected_num + b1+b2+b3+b4)){
		bitmapSend.add("1");
	}else{
		bitmapSend.add("0");
	}
}
/**
 * Determine bitmap value for constructing next expected window use.
 * @param num block size
 * @param n   expected sequence number of fist position in the block
 */
private boolean addBitmap(int num, int n_){
	int n = n_% (window_size + 1)/*16*/;
	int temp = 0;
	for(int i = 0; i < num; i++){
		n = n + i;
		if(n > window_size/*15*/){
			n=0;
		}
		for(int j = 0; j < buffer.size(); j++){
			ARQ_Block p = (ARQ_Block)buffer.get(j);
			if(p.sequence_num == n){
				temp += 1;
			}
		}
	}
	if(temp == num){
		return true;
	}else{
		return false;
	}
}
/**
 * Make sequence number.
 */
private void sequenceCycle(){
	if(next_expected_num == window_size + 1/*16*/){
		next_expected_num=0;
	}
```

```
	}

	/**
	 * Set totoal numbr of packets, for calculating throughput purpose.
	 * @param vale number of packets
	 */
	public void setPacketNum(int value){
		total_packets = value;
	}
	/**
	 * check if required number of packets has reached, then finish.
	 */
	private void finish(int seqNum){
		if(packets_sent == total_packets){
			stop = true;
			Acknowledgement ack = new Acknowledgement();
			ack.time = getTime();
			ack.setNum(seqNum);
			ack.done = 1;
			sendmsg(ack, 10/*size*/, 0, 101);//send back to nofity sender
		}
	}
	/**
	 * send acknowledgement back to the transmitter.
	 */
	private void sendAckBack(){
		firstError = true;
		normalState = true;
		constructNextWindow();
		sendmsg(sendAck(), 10, dst, dport);
		bitmap.clear();
		bitmapSend.clear();
		sentToUpper.clear();
	}
}
```