Universitetet i Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study program/specialization: *Computer Science*	Spring semester, 2016 Open / ~~Confidential~~
Author: *Fredrik Wæhre Severinsen*	[illegible] (signature author)
Instructor: *Hein Meling* Supervisor(s):	
Title of Master's Thesis: *Goxos Reliable Datagram Protocol: A new Transport Layer for Goxos*	
ECTS: *30*	
Subject headings: *Transport Layer Protocols* *Flow Control – Sliding Window* *Network Communication* *Connection Handling* *Congestion Avoidance*	Pages: *68* + attachments/other: *Code attached in pdf* Stavanger, 15 June 2016

Goxos Reliable Datagram Protocol

A new Transport Layer for Goxos

University of Stavanger

Fredrik Wæhre Severinsen

June 2016

Department of Electrical Engineering and Computer Science
Faculty of Science and Technology
University of Stavanger

Abstract

Multiple computers working together as a single distributed system is becoming increasingly common. State machine replication is an approach used to ensure fault tolerance in such distributed systems. Goxos is a framework, based on the consensus algorithm Paxos, which can be used to builds such fault tolerant replicated state machine systems.

In this thesis, we aim to leverage the fact that Paxos does not depend on reliable communication to ensure consistency among the replicated state machines, to implement a new datagram-based communications protocol. A new protocol with low overhead and fast transmission, intended to be used by the Goxos framework.

Paxos requires some form of retransmission to ensure that progress can be made. To this end, the new protocol features a sliding window protocol with selective retransmission. In addition, the new protocol features various other transport layer mechanisms. The mechanisms were selected by first examining some of the functionality commonly provided by other communication protocols. Then, different variations of the mechanisms were compared in experiments and a final incarnation of the protocol was compared to the original Goxos implementation.

Though the new protocol does not surpass the original Goxos implementation when it comes to performance, it delivers a viable throughput, not far from the original's. In addition, by being implemented in user space, and having a modular, layered structure, the new protocol is open and accessible for further modifications and improvements.

Dedication

This thesis is dedicated to my grandmother Johanne Emilie Wæhre, and to the memory of my grandfather Rolf Henry Wæhre.

Acknowledgments

I would first like to thank my advisor Hein Meling. His weekly meetings, helpful guidance and support kept me poised throughout the semester. Secondly i would like to extend my gratitude to Tormod Lea for helping to clear up some roadblocks encountered during the implementation and testing of the new protocol.

Next i give thanks to my fellow students, for proofreading, feedback on the thesis and for all the good times.

I would also like to thank my parents, Magne Severinsen and Turid Wæhre, and my brothers, Joakim and Daniel, for their love, support and encouragement. Lastly i thank Marte Engeland Reisæter, for all her love and support.

Contents

Chapter 1

Introduction

The use of cloud computing for various tasks and services is becoming increasingly prevalent. In a distributed cloud computing system, it is important that a failure in one node does not bring down the whole system. Typically costly techniques are used to provide an acceptable level of fault tolerance in a system. This can involve backup hardware which kicks in at the event of a failure or multiple transmissions of the same data.

There is a desire to keep costs low, while at the same time preserving fault tolerance. To this end, we have the replicated state machine (RSM) approach. Fault tolerance is achieved by having each machine in the system, contain a copy of the system's state. Updates to the state has to be agreed upon, through consensus. The most widely used algorithm for reaching such a consensus is know as Paxos.

The Paxos protocol itself does not depend on reliable communication to ensure safety, i.e. consistency among the replicas, and so it can be implemented using unreliable communication. In this thesis, we aim to leverage this fact to implement a new datagram-based communications protocol with low overhead and fast transmission. A new protocol intended to be used by a Paxos-based RSM framework.

While Paxos retains safety even when using unreliable communication, it may experience lack of progress, and so to make progress, our protocol needs to implement some form of retransmission. In addition to retransmission, our protocol provides a handful of other transport layer services, with the added bonus of being located in user space. This means the protocol is accessible for modifications and extensions. Different variants of transport layer mechanisms will be compared and selected through experimentation. Comparisons between the final incarnation and the original framework, using TCP for communication, will be conducted. Hopefully the new protocol has achieved performance and efficiency which surpasses the original.

Chapter 2 provides the background information, such as transport layer mechanisms and terminology, needed to understand the thesis. Chapter 3 describes the current implementation of the Paxos-based RSM framework Goxos. In chapter 4, the architecture and design of the protocol is presented, while the implementation is presented in chapter 5. Chapter 6 shows the results of the experiments. Possible future work is mentioned in chapter 7, and chapter 8 concludes the thesis.

Chapter 2

Background

This chapter describes some of the functionality commonly provided by the transport layer for network communication. The problems these services solve will be discussed and we take a look at some specific variations of flow and congestion control to see how those problems are met. To start, there will be some details about Go. Go is the programming language which is used for the Goxos implementation. It is also the language which will be used for the implementation work in this thesis.

2.1 Go

This section gives a description of the programming language used in this thesis. A rudimentary understanding of the terminology in Go is expected to be able to understand the implementation work done in this thesis.

Go [7] is a programming language initially developed by Google in 2007, later released to the public as an open source project in 2009. It was designed to be a systems programming language which was compiled, statically typed, had good support for networking and multiprocessing, as well as being textually light, in the fashion of dynamically typed languages. Go is related to C and C++, which are also statically typed and compiled, but Go differs from them by being memory safe and by including garbage collection.

Go was also designed with concurrency in mind, it features built-in concurrency facilities, specifically goroutines, channels and the "select" statement. A goroutine is a light-weight process. It can be viewed as Go's take on threads. Starting a function in a new goroutine can be achieved simply by prefixing the function call with the "go" keyword. While common concurrency control structures such as mutually exclusive locks are available in go, synchronization and message passing between goroutines are normally achieved with the use of channels, which are synchronized FIFO buffers.

Go's types are similar to objects of other languages. A type can be a struct, meaning it is a type which has a collection of fields. Basically this means that it is a type which is made up of multiple other types. In listing 2.1 you can see a type called person. It is a struct consisting of a string called name and an int called age.

String and int are two basic types of Go. A function tied to a specific struct is called a method of that struct.

```go
type person struct {
  name string
  age  int
}
```

Listing 2.1: A type called person

Go features a slice type, which is an abstraction built on top of Go's array type. Slices are analogous to arrays in other languages.

Go's channels are type specific. A channel of type T can only pass messages of type T. They are operated by special arrow syntax; say a goroutine includes the expression <-ch, the goroutine will be blocked until a message is passed over the channel ch, on the other hand the expression ch <- x, sent the value x over the channel ch. The built-in switch-like select statement can be used to implement non-blocking communication on multiple channels.

Listing 2.2 shows a simple program implemented in Go, using channels and goroutines. A channel which passes ints is created at line 1. On line 4 a call to an anonymous function is made. The call is prefixed with the "go" keyword, this means it will be executed in a separate goroutine (thread). On line 5 The anonymous function is blocked from proceeding until something is passed to the ch channel. The main goroutine continues and prints the string "Hello ". After the print, it passes an int to ch. The goroutine, blocked at line 5, can now read a value from the channel and proceed to print "world". Resulting output: Hello world.

```go
 1  func main() {
 2    ch := make(chan int)
 3
 4    go func() {
 5      <-ch
 6      fmt.Print("world")
 7    }()
 8
 9    fmt.Print("Hello ")
10    ch <- 1
11  }
```

Listing 2.2: Hello World implementation in Go, using goroutines and a channel

2.2 The Transport Layer

The Transport Layer is the fourth layer of the Open Systems Interconnection (OSI) model [5] which is a standardization of computer networking. The Transport layer provides services such as flow control, reliability, congestion avoidance and multiplexing based on port numbers. The most known transport layer protocols are Transmission Control Protocol (TCP) [1] and User Datagram Protocol (UDP) [2].

The main differences between the two is reliability. UDP is a simple unreliable protocol which can be seen as a "fire and forget"[11] protocol, sending independent datagrams without expecting a response. TCP on the other hand is a connection-oriented protocol, providing a reliable stream of data, with guaranteed order of messages [13].

TCP also include many other services which UDP does not, such as flow control and congestion avoidance. Due to the extra overhead associated with the TCP connection and its various other services, the protocol is much heavier than the lightweight UDP. The benefit of UDP is the low overhead associated with it. UDP provides unreliable but quick data transmission.

2.3 Flow Control

This section gives a description of flow control, a transport layer service provided by protocols such as TCP and Google's QUIC [6]. Flow control is not included in UDP, and as such it is one of the services which will be added on top of UDP in this thesis.

Flow Control is the process of determining the rate of data transmission in an effort to avoid congestion at a data receiver. When two nodes are communicating, the transmitting node might transmit faster than the receiver can receive and process the data, causing the receiver's buffer to fill up and overflow. This can happen either if the receiver is experiencing heavy traffic load, or if it simply has less processing power than the transmitter. An overflowing buffer will lead to new frames being lost while the receiver is still trying to process old ones. To avoid this, some form of flow control is implemented. Flow control provides some mechanism which allows the receiver to determine the rate of transmission when communication is initiated. Flow control schemes are often divided into two classes, those which provide feedback, and those which do not send any kind of feedback. Below are descriptions of two of the most common forms of flow control. Both forms of flow control will be included in the implementation of the new protocol.

2.3.1 Stop-and-Wait

This section gives a description of the simplest and most basic form of flow control, known as Stop-and-Wait (also known as Send-and-Wait) [12].

Stop-and-Wait works by having the transmitter stop and wait for an acknowledgment (ACK) for each individual packet it sends, before it sends the next packet. Large messages are split into smaller packets. The packets have to be smaller in size than the max buffer size of the receiver.

After splitting a message, a first packet is sent, and an ACK is returned from the receiver which confirms that the packet was successfully received. The transmitter then sends the next packet after having received the ACK. This simple communication is shown in figure 2.1. A timer is also used in case the packet or ACK is lost during transmission. After sending a packet, the transmitter waits for the ACK until some specified time has passed. If it does not receive the ACK within that time, a timeout occurs and the packet is retransmitted.

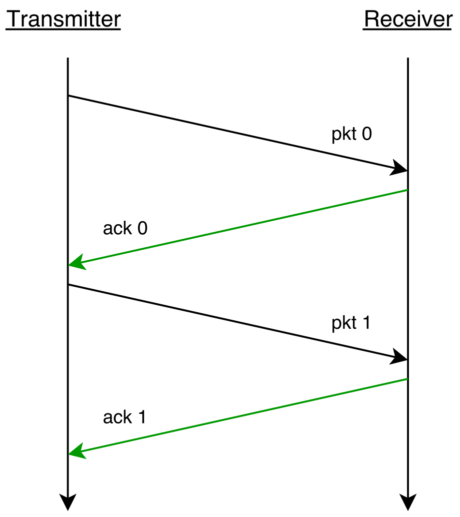

Figure 2.1: Packet exchange over a network for a simple Stop-and-Wait protocol. Time proceeds down the figure.

This method of flow control includes a large amount of waiting. The need to wait for an ACK after each packet is sent, means that transmission will often be slow and inefficient [12].

2.3.2 Sliding Window

The most common type of flow control is known as sliding window. It is a much more efficient flow control method than Stop-and-Wait. By using sliding window multiple packets can be in transit at the same time. The flow control included in TCP is a variation on the general sliding window mechanism described below.

When using sliding window, communication between two nodes start with the receiver allocating space in its buffer for N packets. This space is the receivers window. The transmitter also has a window (sliding window) of the same size. The transmitter can then send N packets without the need to wait for an ACK for the individual packets. The packets are given individual sequence numbers by the transmitter and after the packets are received, the receiver returns an ACK which includes the sequence number of the packet it received from the transmitter. This ACK indicates to the transmitter that the receiver has received the packet with that sequence number. If the acknowledged packet is the first entry in the transmitters sliding window, the window can slide pased that slot. This means that a new sequence number is available for transmission.

Figure 2.2 shows an example of communication when the sliding window protocol is in use. In Figure 2.2 the transmitter is able two send two packets before it receives an acknowledgment of the first. The window size is 5. That means the transmitter can, at the start, potentially send packets with sequence number 0 through 4 without receiving an ACK. For some reason ACKs for packet 3, 4 and 0 are

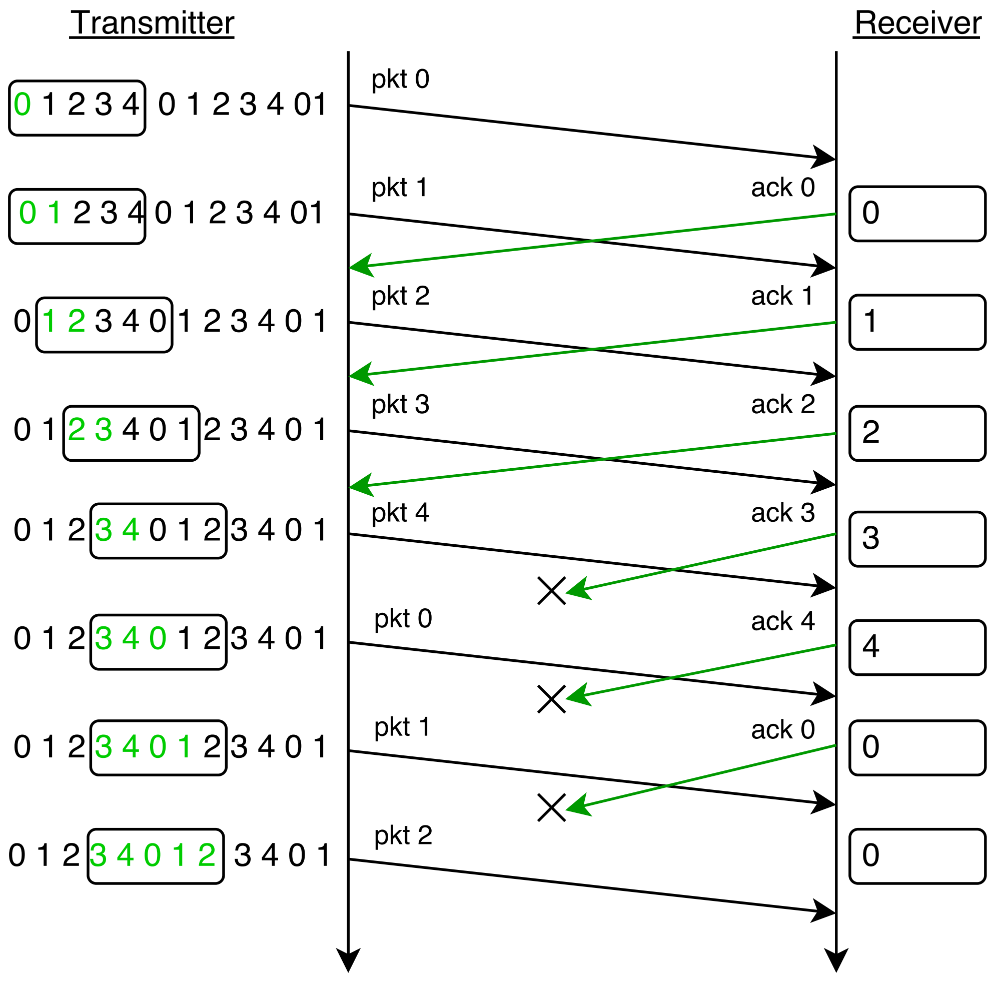

Figure 2.2: An example of the sliding window protocol with window size of 5. A green sequence number indicates that the packet is awaiting an acknowledgment. Time proceeds down the figure.

lost. The transmitter can still send packets until the window is full of unacknowledged packets. If the window of a sliding window protocol is filled up due to packet or ACK loss, some form of retransmission must occur. If not, the communication between transmitter and receiver will break down. Retransmission is discussed in the next section. The receiver instantly passed the received messages to the application due to the fact that all packets are received in order.

sliding window performs much better than Stop-and-Wait due to the fact that it allows multiple packets to be in transit at the same time [12]. A sliding window protocol with window size set to 1 is the equivalent of a Stop-and-Wait protocol.

2.3.3 Sequence Numbers and Integer Overflow

In figure 2.2, the sequence numbers are shown to "wrap-around", iterating from 4 back to 0. The "wrap-around" can be achieved by doing a modulo operation on the original sequence number, using the window size as modulo. This prevents the sequence numbers from getting too large. An integer can only represent numbers up to a certain value, depending on how many bits the integer consists of. Exceeding this value will lead to an integer overflow, e.g. a signed integer represented by 8 bits will overflow if it gets larger than 2^7 -1 = 127. Another way of achieving a "warp-around" effect, is to use unsigned integers. When an unsigned integer overflows, it will simply "wrap-around" on its own. The output of listing 2.3 shows what happens in Go, when a 8 bit signed or unsigned integer overflows.

```go
var i8 int8 = 124 // overflows if > 127
var u8 uint8 = 252 // overflows if > 255

fmt.Println("int8 : uint8")
fmt.Println("------------")
for i := 1; i < 7; i++ {
  i8 += 1
  u8 += 1
  fmt.Println(i8, " : ", u8)
}
// Output:
//
// int8 : uint8
// ------------
// 125  :  253
// 126  :  254
// 127  :  255
// -128  :  0
// -127  :  1
// -126  :  2

```

Listing 2.3: Overflowing int8 and uint8 in Go

2.4 Automatic Repeat Request

Automatic Repeat Request (ARQ) is a method of error control for data transmission. An ARQ protocol usually works by retransmitting a packet if it has not received an ACK for that packet within a specified timeout duration [12]. ARQ protocols may

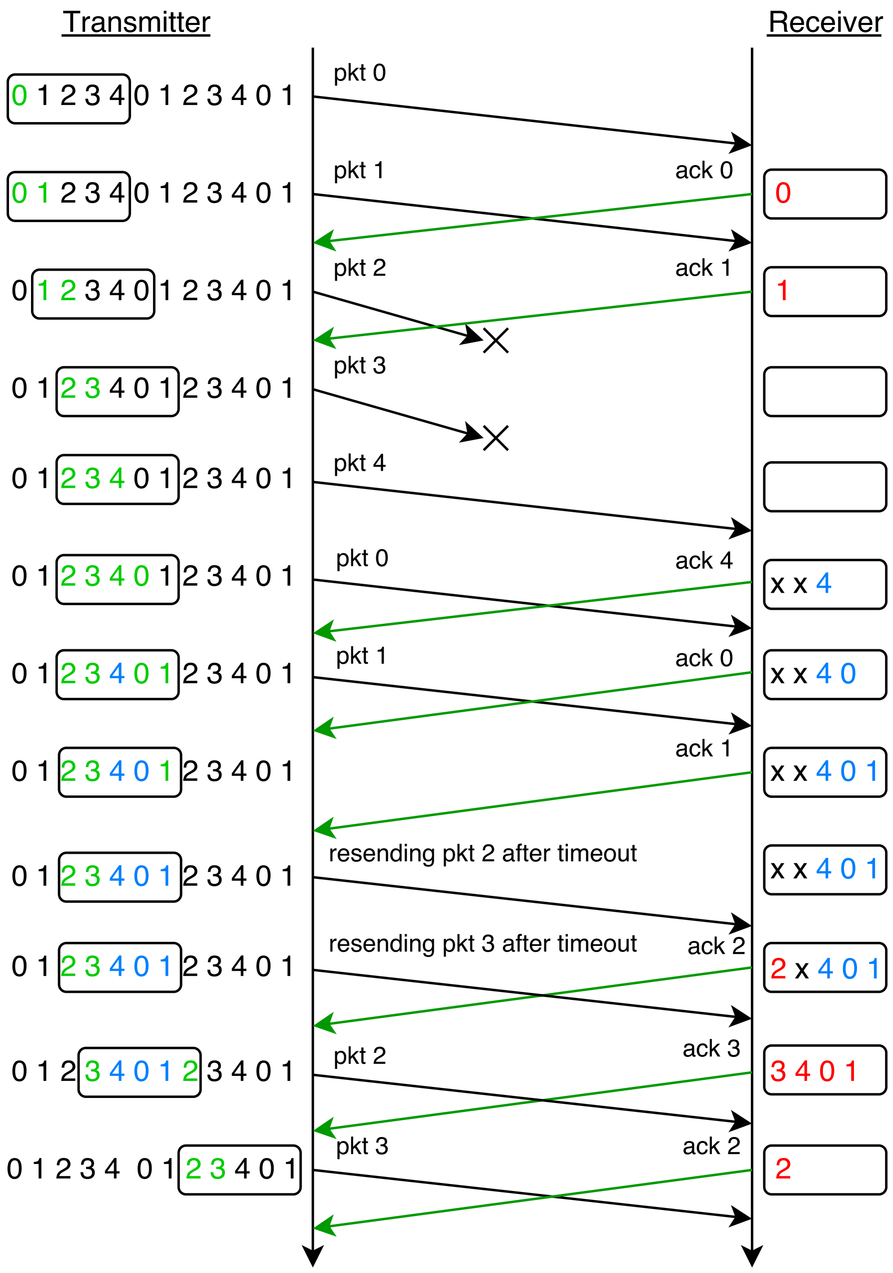

Figure 2.3: An example of the sliding window protocol with Selective Repeat ARQ. Time proceeds down the figure.

be characterized by their persistency. Persistence is the willingness of the protocol to retransmit lost frames to ensure reliable delivery of traffic across the link [3].

2.4.1 Selective Repeat ARQ

Most ARQ protocols are extensions of sliding window protocols. One such version is Selective Repeat ARQ. Selective Repeat ARQ differentiates itself from many other ARQ protocols by only retransmitting the packets which are actually lost. In other words, the receiver can buffer packets with sequence numbers higher than the lost packet. Other ARQ protocols, such as Go-Back-N [12], retransmits the lost packet, and all packets which were sent after it. Even though those packets might have been received.

In figure 2.3 a scenario is shown where two packets are lost during communication through a sliding window with Selective Repeat ARQ. The green numbers indicate packets which are awaiting acknowledgment. The blue numbers are packets which have been received by the receiver, but can not be passed to the application until some packet or multiple packets with lower sequence numbers are received. In the case of the transmitter, the blue numbers indicates that the packet has been acknowledged, but the sliding window can not slide passed it until some packet(s) with a lower sequence number(s) has been acknowledged. The red numbers indicate packets which have been received and are ready to be passed to the application.

Even though packets 2 and 3 are lost during transmission, the transmitter continues and sends packets 4, 0 and 1. Now the receiver has packets 4, 0 and 1 in its buffer, but it cannot pass them to the application before it receives packets 2 and 3. The transmitters window is full, so it stops sending packets. After a certain amount of time, packet 2 times out and is resent. Shortly thereafter, the same happens with packet 3. Packet 2 is received by the receiver and gets passed to the application. The buffer still contains packets 4, 0 and 1. The receiver is waiting for packet 3, which is next in line to be passed to the application. When it is received, all the packets in the buffer gets passed to the application. Likewise on the transmitter side, its window slides one slot after packet 2 has been acknowledged. After packet 3 has been acknowledged, it can slide passed 3, 4, 0 and 1.

2.5 Congestion

Congestion is a condition which occurs when a node in a network becomes overwhelmed by incoming traffic. The node will start dropping packets when its buffer is filled up. This can lead to a state known as congestive collapse, in which useful communication over the network becomes limited or prevented.

Congestion is easy to understand when considering a simple example taken from [13]: Let us say we have a network consisting of an Ethernet switch, with only three computers attached. Suppose two computers send data to a third. Assume each of the connections to the switch operates at 1 Gbps. Now consider the scenario where computers A and B are sending data to C continuously. A and B will forward data at an aggregate rate of 2 Gbps. Because the connection to C can only handle half that rate, the link to C will become congested. The switch has no way of informing A and B of the congestion or stopping incoming traffic. After some amount of time the buffer space of the switch will be used up, forcing the switch to drop arriving packets.

2.6 Congestion Avoidance

This section describes the methods and mechanisms used to avoid sending more data than a network is capable of transmitting, that is, to avoid causing congestion.

Congestion Avoidance regulates the traffic entering a network, in an effort to avoid congestive collapse. This is achieved through a reduction of the transmission rate at the transmitter, in response to some indication that congestion is occurring. Such indications may be larger delay times, round trip times (RTTs) or packet loss. In addition to mechanisms which deal with congestion as it occurs, there are also mechanisms used to avoid congestion in the first place.

2.6.1 Slow-Start

This section describes a part of the congestion avoidance strategy used by TCP, called slow-start.

When communication over a network one way of determining the rate of which new packets are sent into the network is to use the returning ACKs as triggers. ACKs cannot be generated by the receiver faster than the packets can get through the network. This system will dynamically adjust to variations in delay and bandwidth in the network. There is a problem with this system though, it requires data in circulation, meaning that starting up after a packet loss or otherwise is hard, as stated by the creators of slow-start [14]: to get data flowing there must be ACKs to strobe out packets but to get ACKS there must be data flowing.

It is to solve this problem slow-start was developed, and the algorithm is quite simple [14]:

- Add a congestion window, *cwnd*, to the per-connection state.
- When starting or restarting after a loss, set cwnd to one packet.
- On each ACK for new data, increase cwnd by one packet.
- When sending, send the minimum of the receiver's advertised window (receiving window, mentioned in 2.3.2) and cwnd

Despite what the name implies, this algorithm does not actually give a slow start. As illustrated by figure 2.4, even though it starts at a single packet, it increases the congestion window exponentially, ensuring that the maximum transmission rate, bounded by the receiving window, is quickly reached.

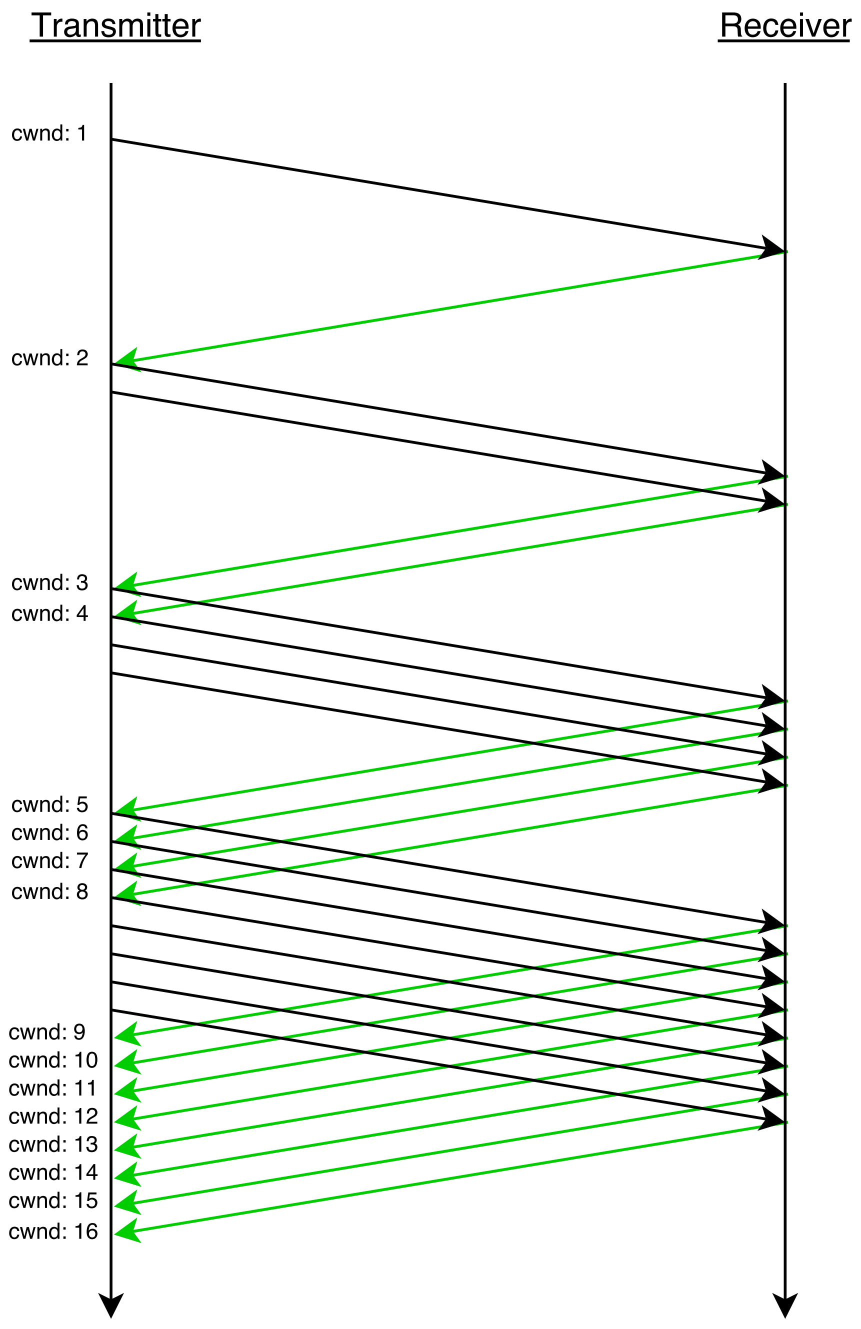

Figure 2.4: An example of the slow start protocol. The black arrow represent packets and the green arrows represents acknowledgments. Time proceeds down the figure.

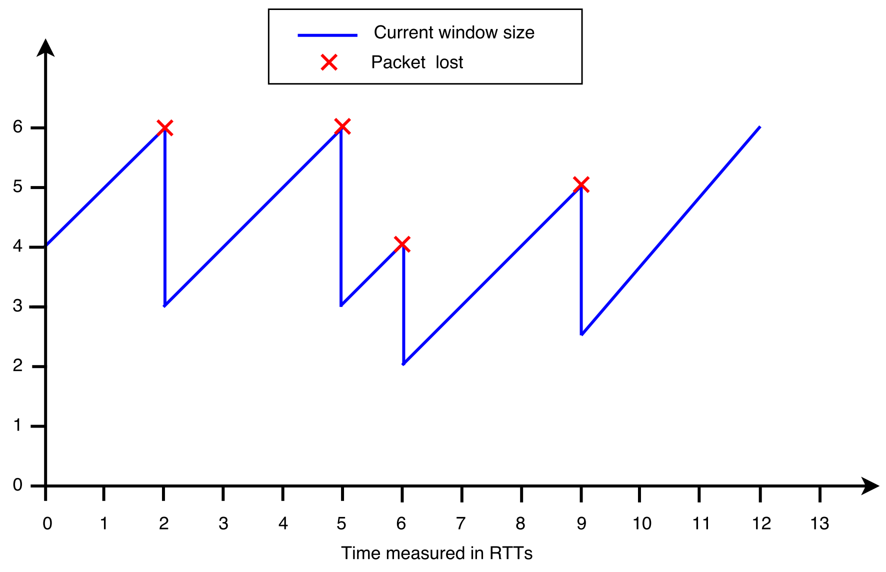

Figure 2.5: Typical behavior of the window size when using AIMD. Unless there has been a packet loss, an ACK is received at each RTT point.

2.6.2 Additive-increase/Multiplicative-decrease

This section describes the Additive-increase/multiplicative-decrease (AIMD) algorithm used for congestion control. AIMD is included in TCP congestion avoidance. One benefit of AIMD over the related algorithms multiplicative-increase/multiplicative-decrease (MIMD) and additive-increase/additive-decrease (AIAD), shown by Chiu and Jain in 1989 [15], is that multiple data flows using AIMD congestion control will eventually converge to use equal amounts of a contended link.

The name of AIMD refers to how modification of the congestion window is done. The algorithm works by increasing the congestion window until loss occurs. The window is additively increased with some fixed amount each time some duration of time has past. This can be either a statically given or dynamic duration, for example RTT. When congestion is signaled, often by a timeout or duplicate ACKs, the congestion window is decreased by a multiplicative factor. It is common to half the window after a loss. Such is the case in TCP. The typical behavior of the cwnd size when using AIMD is illustrated in figure 2.5, while figure 2.6 shows a scenario where AIMD is combined with the maximum window size of e.g. a sliding window protocol. The saw-tooth graph is characteristic of communication when AIMD is implemented.

2.7 Paxos

Paxos was proposed by Leslie Lamport in 1998 [4] as an algorithm used for the purpose of reaching consensus in distributed systems. Paxos ensures that decisions can be made if more than half of the nodes in a distributed system are functioning

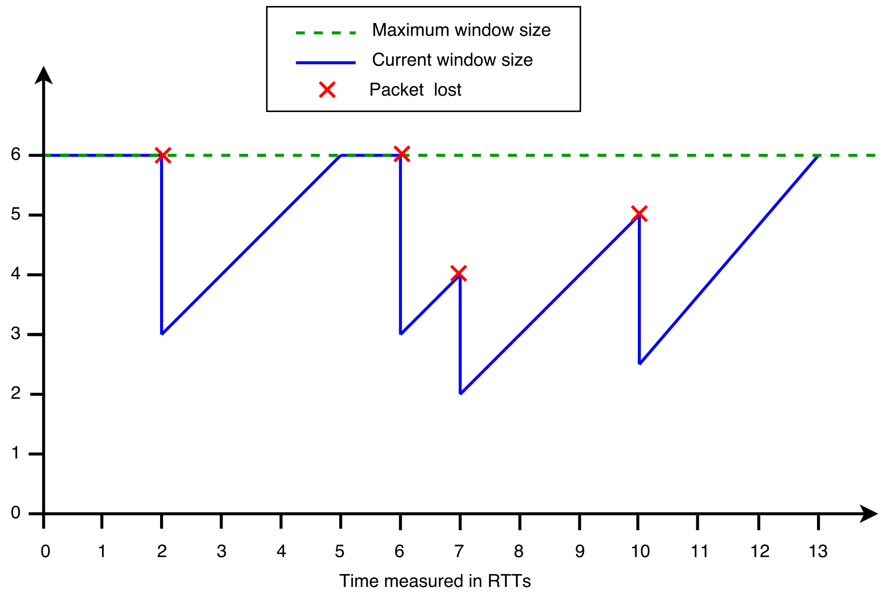

Figure 2.6: Behavior of the window size when using AIMD combined with sliding window. Unless there has been a packet loss, an ACK is received at each RTT point.

properly. The Paxos algorithm is a central part of the Goxos implementation. Most of the messages passing through the new transport layer will be carrying one of the message types of the Paxos algorithm as payload. Even so, understanding the Paxos algorithm in detail is not required to understand the implementation work done in this thesis.

Chapter 3

Goxos

Most large web sites and web applications today usually consist of several distributed subsystems. These distributed systems need to be fault tolerant, a single node failing should not stop the whole system from functioning. This fault tolerance is often achieved by using the Replicated State Machine (RSM) approach. RSMs are built on a consensus algorithm, one of the most commonly used algorithms being Paxos.

Goxos is a Paxos-based RSM framework developed by the Distributed Systems Group at the University of Stavanger, which applications can use to implement a consistent stateful service. The main foundation of goxos was implemented by Jothen and Lea while taking the course Project in Computer Science [8], with new functionality and improvements added later through multiple master theses done by various students at the University of Stavanger, most notably Lea's own master's thesis from 2013 [9].

The implementation work done in this thesis builds upon Goxos. This chapter will give an overview of the Goxos framework and a more in-depth look at the structure of Goxos's network module, which is the focal point of this thesis.

3.1 Overview

One of the design decisions made during the planning done before the initial Goxos implementation, was for Goxos to be modularized [8]. The modular design was included as a way to manage the complexity of Goxos's code base. This modularization leads to a better debugging and testing environment, in which tests can focus on single modules. Another benefit is that modules can be replaced with relative ease, something which was very important for the earlier work done on Goxos, as multiple versions of the Paxos algorithm was implemented and evaluated [8]. In this thesis, we will also benefit from this modularization, as it allows us to focus mainly on a single module; the Network module. Below is a description of the main modules and submodules of Goxos. The structure and interconnectivity of these main modules are shown in figure 3.1.

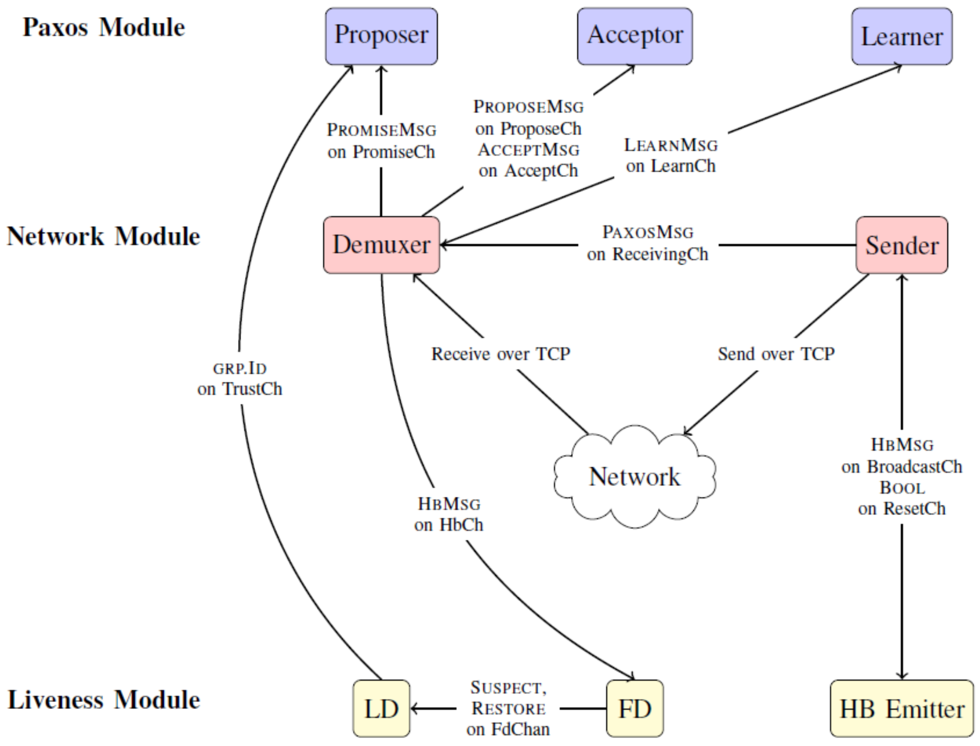

Figure 3.1: A partial view of the architecture of a single node of Goxos where each separately colored layer is a module (Go package) and each entity is a goroutine sending and receiving messages. Each arrow represents a Go channel. This figure and its caption is taken from the original report on the Goxos implementation [8].

The Paxos module is the first of the main modules. The Goxos framework is based on communication between multiple nodes in a distributed system. Each node implements three submodules representing the three actors required by the Paxos algorithm: a Proposer, an Acceptor and a Learner. These three submodules, as a whole, are referred to as one of the three main modules of Goxos; the Paxos module [8]. It is responsible for all logic associated with the Paxos algorithm.

The Liveness module is one of the main module of Goxos, is responsible for leader election and failure detection. It is divided into multiple submodules: Leader Detector; which hold information about the current leader, as well as being responsible for leader election and replacement handling. Failure Detector; which is responsible for informing other modules about which nodes are believed to have failed. It assumes failure if the node has not received any messages from a specific node for some given time. Even though any message type indicates liveness to the failure detector a third submodule is included in the Liveness module to compliment the failure detector, namely the Heartbeat Emitter. This submodule broadcasts heartbeats at regular, specified intervals unless another message type has recently been broadcast.

The Network Module is the last of main module of Goxos. This is where most of the implementation work of this thesis will be done, and as such it is described in greater detail in the section 3.2. Here I will only give a short overview.

The Network module is responsible for all communication within and between nodes. Communication with clients is handled by another, separate module called the Client Handler. The Network module is comprised of two main parts, a Demuxer and a Sender. All incoming messages are handles by the Demuxer. This includes both messages originating from other modules internally in the specific node and messages sent by other nodes. The Sender is responsible for the transmission of messages to other nodes.

Other important modules include the group manager. It holds a map of all the nodes in the system. Another module which is significant in relation to the work in this thesis is the Server module. It is one of the most central parts of the Goxos framework. It is the representation of a server node. It is the Server module which combines the other modules to form a Goxos implementing server replica. It is responsible for creating, initializing, starting and stopping the other modules of the node.

3.2 The Network Module

Goxos relies heavily on network and internal communication between modules. The Network module handles all communications that are not related to clients. Currently, the implementation of the Network module relies on TCP connections between nodes. The use of TCP means that flow control, error detection and correction, retransmission and congestion control are included in the Goxos implementation.

In the original report on the Goxos implementation it was noted that the choice of TCP was made to simplify many aspects of the implementation. Use of UDP for some parts of the network communication was mentioned as a possible improvement [8]. In addition to the main submodules, Demuxer and Sender, the Network module includes definitions of a few message and connection types, with related functions such as Read and Write. Messages sent between nodes are encoded with Go's built-in gob encoding format. Gob is a binary encoding which has very good encoding and decoding performance [8].

3.2.1 Demuxer

The Demuxer handles incoming messages and distributes them to the different modules. Connections from other nodes are handled by the Demuxer and messages received on those connections are distributed to the appropriate modules for further processing. Communication between modules are done over Go's channels. The Demuxer passes a messages of a certain type to a channel corresponding to that type. Other modules can subscribe the channels with message types relevant to them, e.g. the Proposer subscribes to a channel which the Demuxer passes messages of type Promise to. These channel can also be used by the modules to send messages internally.

3.2.2 Sender

The Sender is responsible for sending messages to other nodes. It includes functionality for both broadcast and unicast. It holds a map of the connections received by and shared with the Demuxer. Other modules can request sending of messages by passing them to one of two channels held by the Sender, one for broadcast and one for unicast, along with the unique id of the node it wishes to send to. The Sender takes care of addressing and matching the id with the corresponding IP address and port.

Chapter 4

Architecture

This chapter aims to give an overview of the design and architecture of the new network module. It describes the responsibilities and functions of the different components of the net package.

4.1 Three-Layered Approach

The main structure of the network package is based on the implementation done by Knut Helleland in connection to his 2014 thesis "Paxos Optimizations" [10]. He proposed a three layered structure with the Demuxer and Sender on top, a Connection Manager at the middle level and a protocol layer on the bottom. Some of the responsibilities of the Demuxer and Sender have been moved to the other two layers. This three layered approach suites my implementation well. The bottom layer provides a clean code space. The protocol implementation does not have to concern itself with any application specific factors. The main bulk of transport layer features will be implemented in the bottom layer.

The top layer interacts with other submodules of Goxos. It receives send requests and passes received messages to the correct channels. All interaction with the network is done through the bottom layer. Between the two layers is a third, middle, layer which is responsible for the initial connect procedure, among other things. The main components of the net package and their main responsibilities are shown in figure 4.1

4.2 Top Layer

The top layer consists of the Sender and Demuxer. The Sender is responsible for handling send requests from other modules. Unicast send requests passed to the Sender consists of some kind of message (e.g. an Accept or Learn from the Paxos modules) and a target node id. The message gets encoded and the Sender retrieves the IP and Port number of the target node. The Sender can then pass the message on to the lower layers. A broadcast send requests is just the message itself. It gets

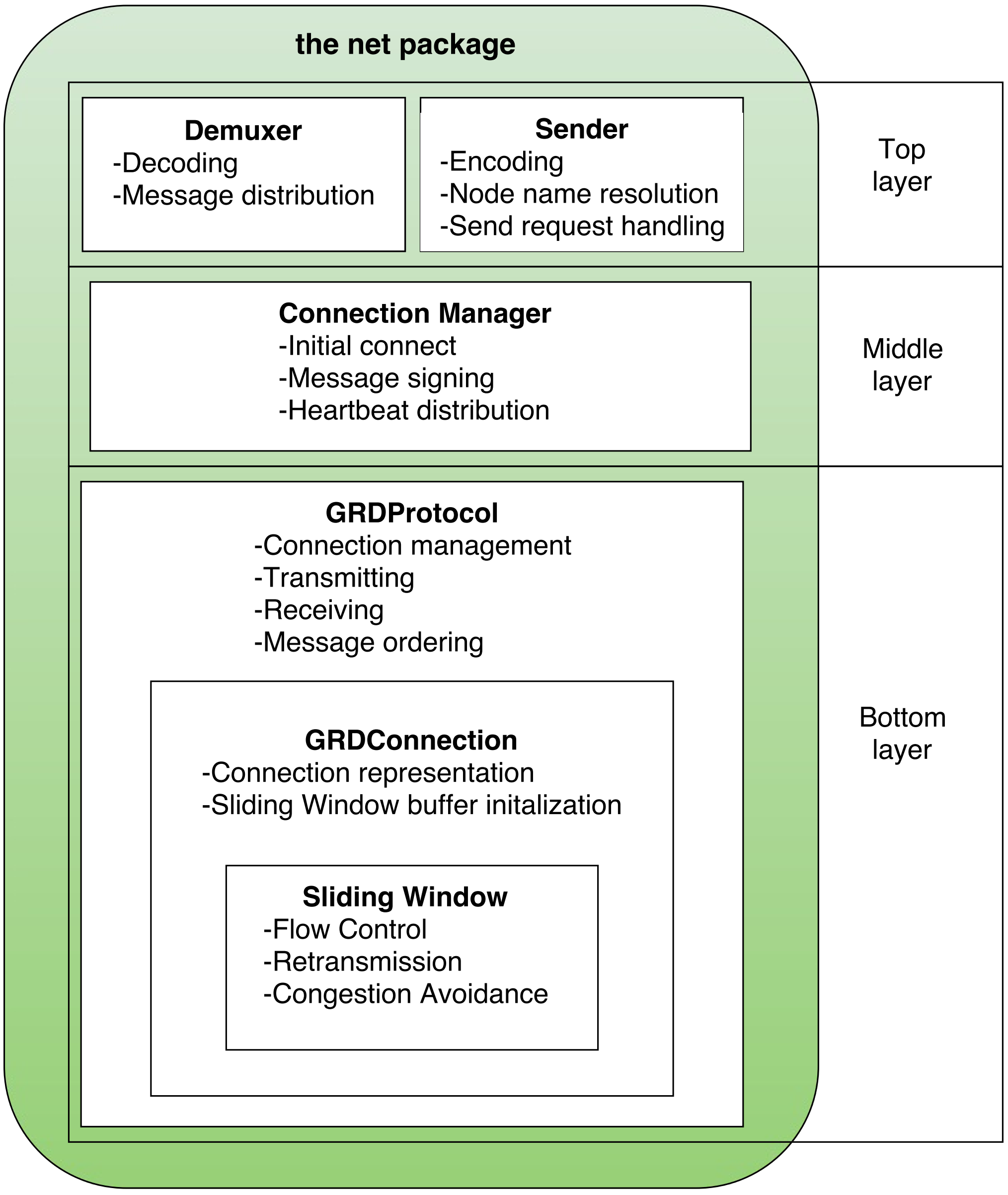

Figure 4.1: Overview of the net package.

encoded and passed on to the lower layers. If a message is to be sent through unicast or broadcast is determined by which of the Senders channels it was passed through.

The Demuxer is constantly reading messages from the lower layers. Whenever a message is received, it gets decoded and its type gets asserted. It will then be passed to a channel corresponding to its type. The various other modules of Goxos subscribe to channels of the Demuxer which corresponds to types they are interested in, e.g. the Paxos module's Learner submodule subscribes to the Demuxers Learn channel.

4.2.1 Encoding

The switch from TCP to UDP based communication meant that the use of Go's native Gob encoding was no longer as optimal. This is due the fact that Gob is designed to work on streams of data. Even so, four encoding schemes have been implemented, including one using Gob. Two are JSON based. One using Go's own "encoding/json" package, and the other using gobson (release r2015.12.06). Created by Gustavo Niemeyer in 2011, gobson is Binary JSON (BSON) encoding for Go [19]. The function calls to the "encoding/json" and "gopkg.in/mgo.v2/bson" packages which are used in this implementation are identical. They both use the same Sender and Demuxer, called XsonSender and XsonDemuxer respectively. Due to the structural similarity of the JSON and BSON scheme, they will henceforth collectively be referred to as XSON.

The XSON scheme has some drawbacks as well. As a result of the Gob and XSON schemes being deemed unsatisfactory, a fourth solution was implemented. This scheme is based on MessagePack. MessagePack was originally created by Sadayuki Furuhashi[20]. The Go implementation of MessagePack which is used in this thesis was created by Vladimir Mihailenco [21]. With the MessagePack scheme, the main drawbacks of the other schemes were avoided. More information about the different encoding schemes will be given in the implementation chapter.

4.3 Middle Layer

The two lower layers both implement the same interface called PacketProtocol, with methods for reading and sending packets, both unicast and broadcast. It also has a method for adding new connections and one which returns the maximum size of a packet. Helleland explained and justified the inclusion of the connection manager (CM) by stating "The connection manager is a layer on top of the PacketProtocol interface to manage connections according to our GroupManager. This separation is made because we want to have our PacketProtocols to be as low-level as possible, without depending on GroupManager." [10]. This desire to have the PacketProtocol be as low-level as possible holds true for our implementation as well. It is the CM who initiates the initial connect procedure and provides the PacketProtocol with necessary information concerning the other nodes in the system.

The CM intercepts sends to the lower layers. It prefixes the messages being sent with a signature comprised of the id of the node and an epoch value, before passing

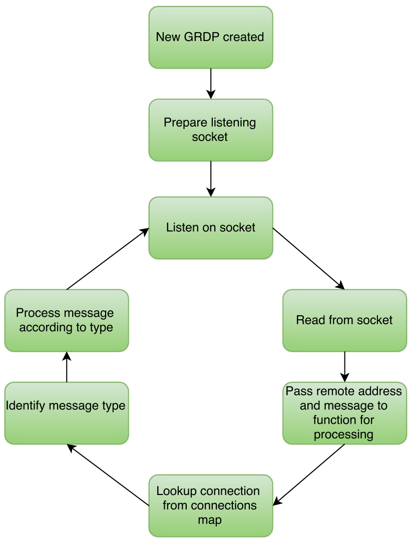
New GRDP created
Prepare listening socket
Listen on socket
Read from socket
Pass remote address and message to function for processing
Lookup connection from connections map
Identify message type
Process message according to type

Figure 4.2: The listening loop of GRDP.

it to the bottom layer. When a message is being passed from the bottom layer to the Demuxer, it will intercept this as well. It reads and removes the prefixed signature. It then passes the id to the failure detector of the liveness module. This informs the failure detector that the remote node is alive. The CM then passes the message, stripped of its prefix, to the Demuxer.

4.4 Bottom Layer

The bottom layer contains most of the algorithms and logics of the Goxos Reliable Datagram Protocol (GRDP). It handles communication over the network.

4.4.1 Goxos Reliable Datagram Protocol

Even though the whole net package is a part of the work being presented in this thesis, its at the bottom layer we find most of the transport layer mechanisms and protocols of GRDP. The two upper layers are Goxos specific and concern themselves with application level tasks. As such, the term GRDP will henceforth refer to this layer, specifically the GRDProtocol struct and the protocols connected to it and its sub elements (i.e. the GRDPconnection and SlidingWindow structs).

When a new instance of GRDP is created (i.e. a Go new variable of the GRDProtocol struct type is created), it spawns a loop in which it listens to the network on a socket specified at its creation. This loop will run until a signal to stop is received from the Demuxer. A simplified version of the processing steps of the listening loop is shown in figure 4.2. Each node holds one instance of GRDP, which holds a map of GRDPconnections. Each connection holds two sliding window buffers. In addition to its listening loop, GRDP has a transmission loop for each GRDPconnection.

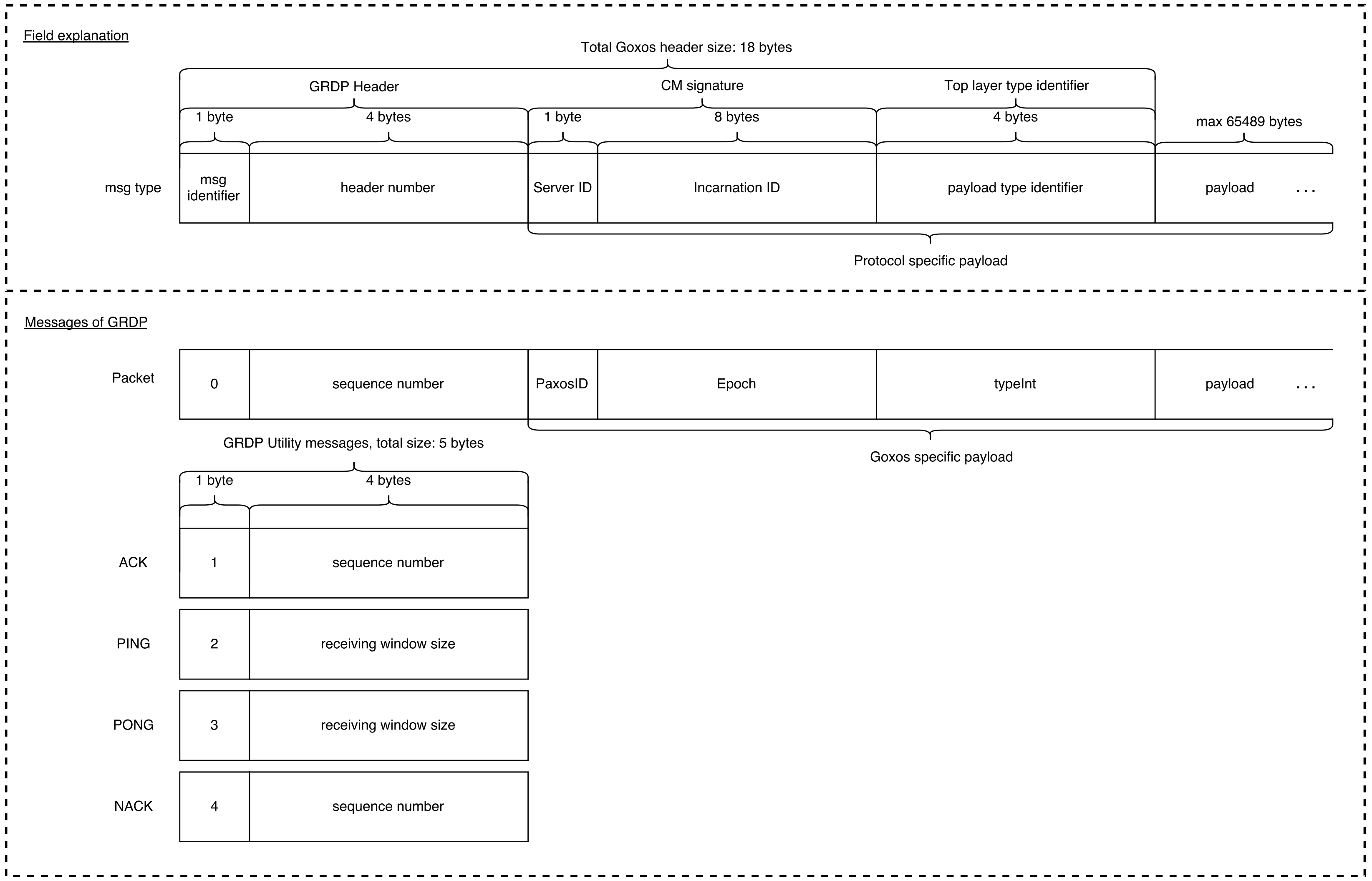

Figure 4.3: Structure of the different message types of GRDP.

4.4.2 Message Types and Headers

GRDP has 5 different message types. All messages are 5 bytes large, except packets, which also has a payload. The first field of every header is a message type identifier. It is an unsigned integer, represented by 8 bits. The different types are shown in figure 4.3.

The second to fifth bytes of each message represents a unsigned integer. For a regular packet, those bits represent the packet's sequence number. In the case of an ACK or NACK, it signifies what packet it is a response to. NACKs will be discussed further in the implementation chapter. Ping and Pong messages use the 32 bits to represent the receiving window size of the node which is transmitting them. This means that the maximum sequence number of a packet is the same as the maximum window size of a buffer. The theoretical limit being: 2^{32} - 1 = 4,294,967,295. The only message larger than 5 bytes is, as mentioned, a regular packet. All bytes after the 5 header bytes of the packet, is its payload.

Goxos Specific Fields

Only the five first bytes are strictly counted as GRDP's header. The fields featured in figure 4.3 after the head number field are specific to Goxos. These fields are not required for GRDP to function.

When using the XSON encoding sheme, the sender prepends a payload type identifier to the encoded message before passing it to the CM. The type identifier is an unsigned integer made of 4 bytes, which is used by the Demuxer to decode the payload to the correct type. More information on this will be given in section 5.2.

When the CM receives a message from the Sender it prepends it with a signature. This signature consists of a server ID which is 1 byte large and an incarnation ID, which is represented by an 8 integer. The incarnation ID is renewed if a server is shut down (properly or crash) and restarted. The CM passes heartbeats to the failure detector based on this signature, thus updating the liveness status of the remote node.

This means that the maximum payload of a packet, sent through GRDP, is the maximum payload of a UDP datagram minus the GRDP header, CM signature and payload type identifier. This gives a maximum of 65,507 - (5 + 9 + 4) = 65,489 bytes.

4.4.3 Ping-Pong Handshake

When the CM starts the initial connect procedure, it will ask GRDP to add all the nodes of the nodemap to GRDPs connections map. One by one, the nodes will be added and the connections will be confirmed through the GRDPs handshake protocol, called Ping-Pong. The 1 RTT Ping-Pong Handshake is illustrated in figure 4.4 which starts with the initiating node sending a Ping message to the remote node. A Ping message only contains the receiving window size of the initiating node and an prefixed byte which indicates that it is a Ping. When the remote node receives the Ping, it adds the initiating node to its connections map, and returns a Pong.

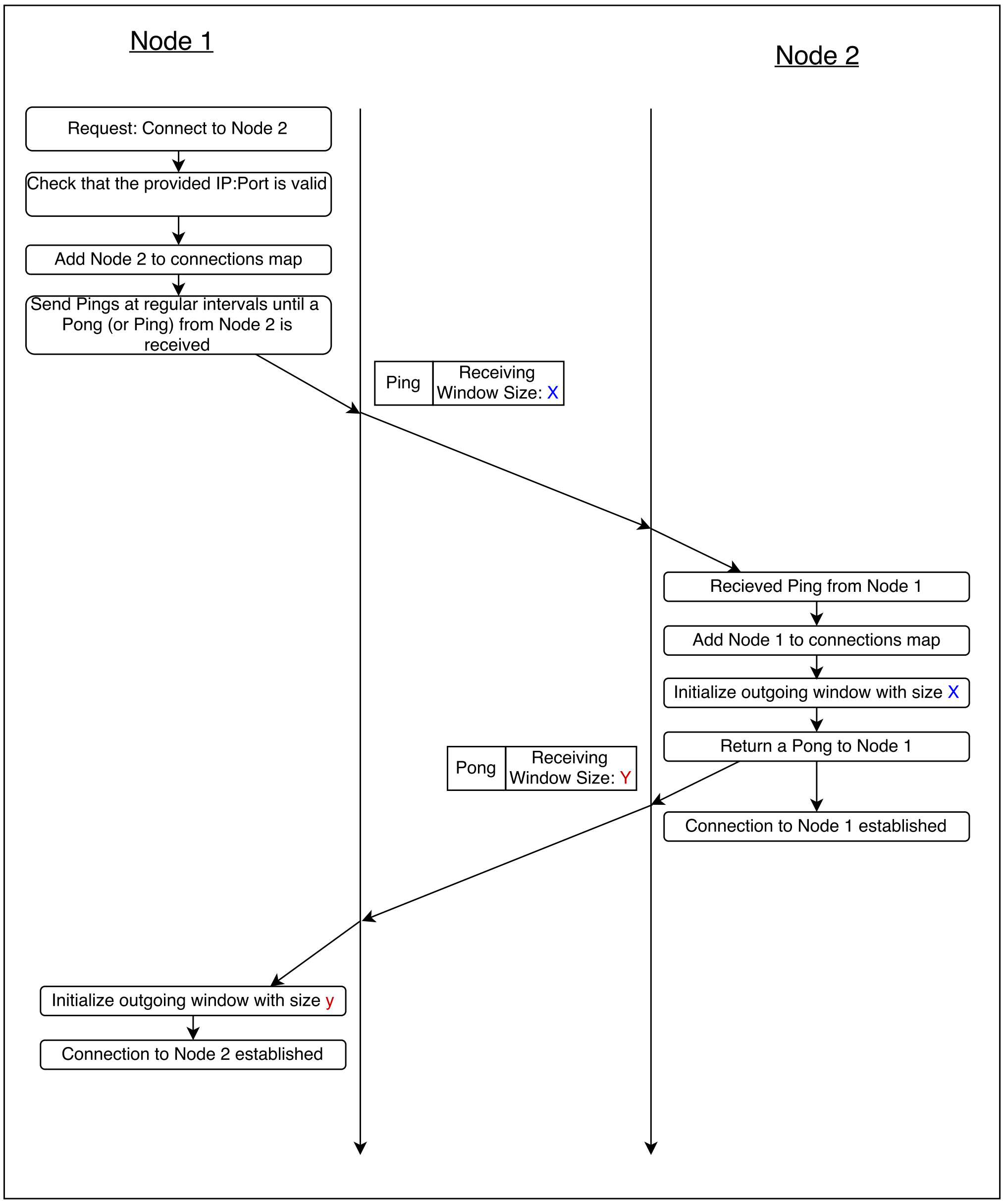
Node 1
Node 2
Request: Connect to Node 2
Check that the provided IP:Port is valid
Add Node 2 to connections map
Send Pings at regular intervals until a Pong (or Ping) from Node 2 is received
Ping
Receiving Window Size: X
Recieved Ping from Node 1
Add Node 1 to connections map
Initialize outgoing window with size X
Return a Pong to Node 1
Pong
Receiving Window Size: Y
Connection to Node 1 established
Initialize outgoing window with size y
Connection to Node 2 established

Figure 4.4: The Ping-Pong handshake. Time proceeds down the figure.

The Pong consists of the remote nodes receiving window size and a prefixed byte indicating its a Pong. Now that they have each others receiving window size, they can initialize their outgoing sliding window buffers. When the buffers are ready, the GRDP starts a transmission loop for the connection. The Ping confirms for the remote node that the initiating node exists and is alive, while the Pong confirms the same about the remote node for the initiating node.

As explained in section 2.6.2, the window size included in a Ping or Pong is represented by 4 bytes or, in other words, 32 bits. Therefore, the largest window that can be used is 2^{32} -1 B = 4,294,967,295. The TCP header has a 16 bit field which specifies window size [17], this means the maximum window size of TCP is 2^{16} -1 = 65,535. Comparatively, the maximum window size of GRDP, at roughly 4.3 billion, may seem vast compared to the meager 65 thousand maximum window size of TCP. However, as shown by the IPv4 address exhaustion [18], numbers which seem vast today, might be considered small in the future. So GRDP has a maximum window size of 4.3 billion and a maximum of 65,489 bytes of payload data per packet. That means that the theoretical maximum amount of unacknowledged payload data which can be in transit at the same time, when using GRDP, is 4,294,967,295 * 65,489 bytes = 281,273,113,182,255 bytes. That is roughly 255 Terabytes of data.

4.4.4 GRDP Connections

GRDP connections are virtual connections representing the link between two nodes. Hereafter, GRDP connections will be referred to as connections. The GRDP keeps track of all its connections in a single map. The socket address of the connection is used as the map key. The connection itself also contains this socket address. Whenever GRDP receives a packet in its listening loop, it looks up a connection from it is map, using the remote address returned from the network read as key. It can then pass the packet to the connections incoming buffer. The socket address contained within the connection is used each time something is to be transmitted over the network.

4.4.5 Flow Control

Each connection holds two sliding window buffers, which was introduces in section 2.3.2. One for outgoing data and one buffer for incoming data. The sliding window algorithm for incoming packets is a method of the GRDP, which is responsible for receiving from the network and passing messages to the upper layers. The algorithm ensures that messages are passed in the correct order. The sliding window algorithm for outgoing packets is a method of the sliding window buffer, which is responsible for passing messages, which are ready to be sent over the network, to the GRDP. It ensures that there are never more packets in transit than the receivers incoming window can handle. By setting the window size of the sliding window to one, we get an implementation of the Stop-and-Wait protocol, described in section 2.3.1. Comparisons of Stop-and-Wait and sliding window will be shown in chapter 6.

4.4.6 Retransmission

GRDP includes a take on the Selective Repeat ARQ protocol. The protocol will be persistent, meaning it will keep retransmitting, without backing of, until the message is acknowledged. As explained in section 2.4, retransmission will occur if a packet is not acknowledged within a certain timeout duration. After each acknowledgment is received, RTT is recorded. The RTT is used to calculate the new timeout duration. The calculation is based on TCP's smoothed timeout duration calculation, defined in RFC 6298 [16]. The following page is, in its entirety, excerpted from RFC 6298:

The rules governing the computation of SRTT, RTTVAR, and RTO are as follows:

(2.1) Until a round-trip time (RTT) measurement has been made for a segment sent between the sender and receiver, the sender SHOULD set RTO <- 1 second, though the "backing off" on repeated retransmission discussed in (5.5) still applies.

Note that the previous version of this document used an initial RTO of 3 seconds [PA00]. A TCP implementation MAY still use this value (or any other value > 1 second). This change in the lower bound on the initial RTO is discussed in further detail in Appendix A.

(2.2) When the first RTT measurement R is made, the host MUST set

```
SRTT <- R
RTTVAR <- R/2
RTO <- SRTT + max (G, K*RTTVAR)
```

where K = 4.

(2.3) When a subsequent RTT measurement R' is made, a host MUST set

```
RTTVAR <- (1 - beta) * RTTVAR + beta * |SRTT - R'|
SRTT <- (1 - alpha) * SRTT + alpha * R'
```

The value of SRTT used in the update to RTTVAR is its value before updating SRTT itself using the second assignment. That is, updating RTTVAR and SRTT MUST be computed in the above order.

The above SHOULD be computed using alpha=1/8 and beta=1/4 (as suggested in [JK88]).

After the computation, a host MUST update
RTO <- SRTT + max (G, K*RTTVAR)

(2.4) Whenever RTO is computed, if it is less than 1 second, then the RTO SHOULD be rounded up to 1 second.

GRDP follows rule 2.2 and 2.3. For clarification, the retransmittion timeout (RTO) is the actual timeout duration. SRTT is the smoothed RTT and RTTVAR is the RTT variance. K is a constant set to 4. G is the clock granularity. The max function of G and K*RTTVAR is included to ensure that e.g. if K*RTTVAR is 0.2 seconds, and the clock only increments in full seconds, the system will not be waiting for a time (0.2 seconds) which the clock never hits.

GRDP does not follow rule 2.4, which specifies that the minimum RTO should be 1 second. Goxos is meant for both wide area networks and computer cluster environments where computers are connected through a local area network. In the latter case, RTTs would be much smaller than 1 second. Therefor, it is recommended that minimum RTO in GRDP be set depending on the server setup and connections.

4.4.7 Congestion Avoidance

The sliding window buffer sizes can be considered maximum window sizes. In addition to these maximum window sizes, there is a current window size, determined by the congestion avoidance algorithm. It is this window size which actually limits the number of unacknowledged packets which can be in transit at the same time. At startup the current window is the same as the maximum size, but each time a packet is lost, and retransmission takes place, the current window is cut in half. Whenever a packet is acknowledged, the current window is increased by one. This is a variation of AIMD, which was described in section 2.6.2.

The algorithm can easily be converted to Slow-Start, shown in section 2.6.1. This is done by setting current window to one at startup and each time a packet is lost. AIMD, Slow-Start as well as MIMD will be compared in chapter 6,

Chapter 5

Implementation

This chapter will present and explain significant pieces of the implementation of GRDP. First the two variations of the top layer, the Gob and XSON schemes, will be described. After this, the rest of the chapter will focus on the bottom layer, GRDP. The CM will not be described here, as it has seen minimal change from the CM implementation from 2014, made by Helleland [10].

5.1 The Gob Scheme

As previously mentioned, Gob is not well suited for datagram-based communication. Listing 5.1 shows the unicast method of the Gob Sender. The method gets called each time the sender receives a send request through its unicast channel. The detrimental part of this method is located at line 6. For each unicast message the sender want to send, a new Gob encoder has to be created. Similarly, a new Gob encoder is created with each call to the Senders broadcast method as well.

```
1  func (snd *GobSender) unicast(msg interface{}, id grp.ID) {
2    if id == snd.id {
3      snd.dmx.HandleMessage(msg)
4    } else {
5      var buf bytes.Buffer
6      enc := gob.NewEncoder(&buf)
7      err := enc.Encode(&msg)
8      if err != nil {
9        glog.Fatalln("encode error:", err)
10     }
11     node, _ := snd.grpmgr.NodeMap().LookupNode(id)
12     err = snd.protocol.SendPacket(node.PaxosAddr(), buf.Bytes())
13     if err != nil {
14     glog.Error(err)
15     return
16     }
17   }
18 }
```

Listing 5.1: The Gob Senders unicast method

Listing 5.2 shows that the same problem is present at the receiving side. On line 11 a new Gob decoder is being created. This happens each time a message is read from the lower layers. One benefit of this scheme is that when the rawBytes byte slice is decoded into msg on line 12, it will be decoded to the correct type. This is thanks to Gob's ability to record types with its register method, at compile time. This convenience is not available for the XSON scheme, which will be shown in section 5.2.

```go
1  func (dmx *GobDemuxer) Start() {
2    glog.Infoln("Starting")
3    dmx.closed = false
4    go func() {
5      defer dmx.stopCheckIn.Done()
6      for !dmx.closed {
7        var rawBytes = make([]byte, dmx.protocol.GetPacketMaxSize())
8        rawBytes = dmx.protocol.ReadPacket()
9        var msg interface{}
10       buf := bytes.NewBuffer(rawBytes)
11       dec := gob.NewDecoder(buf)
12       err := dec.Decode(&msg)
13       if err != nil {
14         glog.Warningln("decode error:", err)
15       } else {
16         dmx.HandleMessage(msg)
17       }
18     }
19   }()
20 }
```

Listing 5.2: The Gob Demuxers main loop

Fruitless attempts were made at a Gob scheme in which only a single encoder and decoder was created. This involved making the GRDP struct implement the Go's Writer and Reader interfaces, so that the encoder and decoder could write and read directly to and from GRDP. Unfortunately, this solution had no way for the Sender to tell GRDP which connection to send the message over. On the decoder side, various decoding errors where encountered. Another possible solution would be to have an encoder and decoder for each connection held by the GRDP.

5.2 The XSON Scheme

Even though XSON was first chosen for its convenience, there were some unfortunate problems which appeared with this new scheme. XSON does not have an equivalent to Gob's register function for recording struct types, which meant that structs being decoded at the Demuxer would only be identified as plain interfaces. Any simple solution involving the Demuxer decoding the messages into the correct struct types based on an identifying int did not work. The problem was that the demuxer could not get access to the types of the paxos package (Paxos module), because the paxos package imported the net package, and trying to import the paxos package anywhere inside the net package only lead to a disallowed circular import. This meant that some other mechanism was needed to build the received bytes into their proper types, so that they could, in turn, be distributed to the other modules. Unfortunately the solution involves hard coding in type information and decoding functions for each custom type which is to be transmitted over GRDP.

5.2.1 Encode Function

The solution involved several components. First of which is a new function, called encode, which is shown in listing 5.3. The function takes as parameter any type which implements the empty interface (i.e. all types). This parameter is called i. The function also takes a bool, useBson, as parameter, this bool is used to select JSON or BSON encoding. If the type of i is recognized as one of the 12 types being sent through Goxos, the typeInt variable will be set to some value which is specific to that type. If i is some other type, typeInt remain 0, which it was initialized to be at line 5. The function makes a call to either json's or bson's Marshal function depending on the useBson parameter, which returns the JSON or BSON encoding of i respectively. This encoding of the i is represented by a byte slice called b. Lastly, typeInt gets converted to a byte slice and the encoding of i is appended to it, and the resulting byte slice is returned together with a nil error value, if no error occurred during the call to Marshal.

The byte slice which gets returned has already been shown in figure 4.3 as part of the packet message's composition. The byte slice representation of typeInt is referred to as the top layer type identifier or payload type identifier and the byte slice b is referred to as the payload.

```
1  func encode(i interface{}, useBson bool) ([]byte, error) {
2    var typeInt uint32 = 0
3    switch reflect.ValueOf(i).Type().String() {
4      case "paxos.Prepare":
5        typeInt = 1
6      case "paxos.Promise":
7        typeInt = 2
8      case "paxos.Accept":
9        typeInt = 3
10     case "paxos.Learn":
11       typeInt = 4
12     case "paxos.CatchUpRequest":
13       typeInt = 5
14     case "paxos.CatchUpResponse":
15       typeInt = 6
16     case "paxos.RangeTuple":
17       typeInt = 7
18     case "paxos.ResponseTuple":
19       typeInt = 8
20     case "liveness.Heartbeat":
21       typeInt = 9
22     case "liveness.FdMsg":
23       typeInt = 10
24     case "*paxos.Accept":
25       typeInt = 11
26     case "*paxos.CatchUpRequest":
27       typeInt = 12
28     }
29   if useBson {
30     b, err := bson.Marshal(i)
31     return append(intToBytes(typeInt), b...), err
32   } else {
33     b, err := json.Marshal(i)
34     return append(intToBytes(typeInt), b...), err
35   }
36 }
```

Listing 5.3: The encode function

Every time the Sender want to send a unicast or broadcast message, it first makes a call to the encode function. This call can be seen on line 7 in listing 5.4, which shows the XSON Senders unicast method. After the encoded packet is returned, it gets passed to GRDP together with the socket address of the target node in the call to SendPacket on line 14.

```go
1  func (snd *XsonSender) unicast(msg interface{}, id grp.ID) {
2    if id == snd.id {
3      snd.dmx.HandleMessage(msg)
4      return
5    }
6
7    packet, err := encode(msg, snd.useBson)
8    if err != nil {
9      glog.Warning("xson.Marshal():", err)
10     glog.Infoln("xson packet:", string(packet))
11     return
12   }
13   node, _ := snd.grpmgr.NodeMap().LookupNode(id)
14   err = snd.protocol.SendPacket(node.PaxosAddr(), packet)
15   if err != nil {
16     glog.Error(err)
17     return
18   }
19 }
```

Listing 5.4: The XSON Senders unicast method

5.2.2 Unmarshallers Variable

The next component of the solution is a variable which is defined in the server package (Server module). There are actually two versions of this variable, one for the JSON and for the BSON scheme. The variables are practically identical. The only difference is the term json/bson. The JSON related variable is shown in listing 5.5. The variable is a map of anonymous functions. The key of the map is a unsigned integer made of 4 bytes. The functions of the map takes in a byte slice which it decodes into a specific type, related to the value of the key.

```go
var json_um = map[uint32]func([]byte) (interface{}, error){
  1: func(raw []byte) (interface{}, error) {
    var f paxos.Prepare
    err := json.Unmarshal(raw, &f)
    return f, err
  },
  2: func(raw []byte) (interface{}, error) {
    var f paxos.Promise
    err := json.Unmarshal(raw, &f)
    return f, err
  },
  3: func(raw []byte) (interface{}, error) {
    var f paxos.Accept
    err := json.Unmarshal(raw, &f)
    return f, err
  },
  4: func(raw []byte) (interface{}, error) {
    var f paxos.Learn
    err := json.Unmarshal(raw, &f)
    return f, err
  },
  5: func(raw []byte) (interface{}, error) {
    var f paxos.CatchUpRequest
    err := json.Unmarshal(raw, &f)
    return f, err
  },
  6: func(raw []byte) (interface{}, error) {
    var f paxos.CatchUpResponse
    err := json.Unmarshal(raw, &f)
    return f, err
  },
  7: func(raw []byte) (interface{}, error) {
    var f paxos.RangeTuple
    err := json.Unmarshal(raw, &f)
    return f, err
  },
  8: func(raw []byte) (interface{}, error) {
    var f paxos.ResponseTuple
    err := json.Unmarshal(raw, &f)
    return f, err
  },
  9: func(raw []byte) (interface{}, error) {
    var f liveness.Heartbeat
    err := json.Unmarshal(raw, &f)
    return f, err
  },
  10: func(raw []byte) (interface{}, error) {
    var f liveness.FdMsg
    err := json.Unmarshal(raw, &f)
    return f, err
  },
  11: func(raw []byte) (interface{}, error) {
    var f *paxos.Accept
    err := json.Unmarshal(raw, &f)
    return f, err
  },
  12: func(raw []byte) (interface{}, error) {
    var f *paxos.CatchUpRequest
    err := json.Unmarshal(raw, &f)
    return f, err
  },
  0: func(raw []byte) (interface{}, error) {
    var f interface{}
    err := json.Unmarshal(raw, &f)
    return f, err
  },
}
```

Listing 5.5: The JSON unmarshallers variable

The unmarshallers variable is passed to the Demuxer through its factory function. By doing this, access to the paxos and liveness packages' types has been achieved in the net package.

Whenever the Demuxer reads a message from the lower levels, it passes the message to its processBytes method. The method is shown in listing 5.6. The GRDP header and the CM signature has already been stripped from the message, meaning that the 4 first bytes of the message is the payload type identifier which was added by the encode function, which we described in section 5.2.1. The rest of the message is the actual payload. On line 2, the payload type identifier is converted to its int form and passed to the Demuxer's unmarshallInnerData method, together with the payload.

```
1   func (dmx *XsonDemuxer) processBytes(rawBytes []byte) {
2     val, err := dmx.unmarshalInnerData(bytesToInt(rawBytes[0:4]), rawBytes[4:])
3     if err == nil {
4       dmx.HandleMessage(val)
5     } else if bytesToInt(rawBytes[0:4]) == 99 {
6       glog.Infoln(string(rawBytes[4:]))
7     } else {
8       glog.Infoln("xson.Unmarshal:", err)
9     }
10  }
```

Listing 5.6: The XSON Demuxer's processBytes function

On line 2 of listing 5.7 the type identifier is used as key on the Demuxers unmarshallers map. If the key is valid, a function will be returned. On line 6 the payload is fed to the function and the decoded payload which the function returns, is returned together with a nil error value, if the decoding went successfully. Back in processBytes, listing 5.6, line 2, val is the decoded payload. If there were no error during decoding, val has a proper type (e.g. the paxos package's Learn or Prepare type) and can be passed to HandleMessage on line 4. HandleMessage distributes the message according based on type, to the other modules of Goxos, as explained in sections 3.2.1 and 4.2.

```go
func (dmx *XsonDemuxer) unmarshalInnerData(i uint32, b []byte) (interface{}, error) {
  unmarshaller, found := dmx.unmarshallers[i]
  if !found {
    return nil, ErrUnmarshallerNotFound
  }
  return unmarshaller(b)
}
```

Listing 5.7: The XSON Demuxer's unmarshalInnerData method

5.3 the MessagePack Scheme

MessagePack is a binary serialization format similar to BSON [20]. The go implementation of MessagePack has the same Marshal and Unmarshal function signatures as the XSON packages. This means MessagePack could be included in the Goxos implementation by simply swapping the package references to one of the XSON packages with the MessagePack package. This solution does not avoid the hard coding problem of the XSON scheme though. Thankfully, MessagePack features a function for registering type extensions, akin to Gob's register function. This means MessagePack, at compile time, records all the different types which will be sent through GRDP. This means that instead of making a call to the encode function, a simple call to MessagePack's Marshal function is made. This can be seen on line 6 of listing 5.8, which show's the Sender's unicast method.

```
1  func (snd *MpSender) unicast(msg interface{}, id grp.ID) {
2    if id == snd.id {
3      snd.dmx.HandleMessage(msg)
4      return
5    }
6    packet, err := msgpack.Marshal(msg)
7    if err != nil {
8      glog.Warning("msgpack.Marshal():", err)
9      return
10   }
11   node, _ := snd.grpmgr.NodeMap().LookupNode(id)
12   err = snd.protocol.SendPacket(node.PaxosAddr(), packet)
13   if err != nil {
14     glog.Error(err)
15     return
16   }
17 }
```

Listing 5.8: The MessagePack Senders's unicast method

The need for both the type identifying int and the unmarshallers variable has been removed, thanks to the type registration of MessagePack. When the Demuxer receives a message from the lower levels, the bytes can be decoded directly into an empty interface, as seen on line 10 of listing 5.9. The resulting variable is of the correct type and can be passed to the HandleMessage method for distribution to the other Goxos modules.

```
1  func (dmx *MpDemuxer) Start() {
2    glog.Infoln("Starting Demuxer loop")
3    dmx.closed = false
4    go func() {
5      defer dmx.stopCheckIn.Done()
6      for !dmx.closed {
7        var msg interface{}
8        var rawBytes = make([]byte, dmx.protocol.GetPacketMaxSize())
9        rawBytes = dmx.protocol.ReadPacket()
10       err := msgpack.Unmarshal(rawBytes, &msg)
11       if err == nil {
12         dmx.HandleMessage(msg)
13       }
14     }
15   }()
16 }
```

Listing 5.9: The MessagePack Demuxers's main loop

5.4 Data Structures of the Goxos Reliable Datagram Protocol

In this section the main struct types of GRDP is presented. Most of the transport layer protocols included in GRPD involves all of them in one way or another. The main struct type is the GRDProtocol struct, which holds a map of pointers to GRDP-connections. Each GRDPconnection holds two SlidingWindows, which in turn holds a slice of pointers to swPackets.

```
1 type GRDProtocol struct {
2   connections           map[string]*GRDPConnection
3   packetsIncomingBuffer chan []byte
4   nullconn              *net.UDPConn
5   mu                    sync.Mutex
6   stopped               bool
7   wndSize_Max           uint32
8   stopChan              chan int
9 }
```

Listing 5.10: The GRDProtocol struct

The GRDProtocol struct, shown in listing 5.10, contains a map called connections, as mentioned in sections 4.4.1 and 4.4.4. In addition to the connections map, the struct holds a channel, called packetsIncominBuffer, used to pass byte slices to the Demuxer. The UDPConn, on line 4 of listing 5.10, called nullconn, is a a UDPConn with IP address 0.0.0.0 and Port number 0000. It is the node's route to the Internet. All connections in the connections map write to nullconn when they wish to send something over the network. The Mutex mu is included to ensure that only one goroutine accesses the connections map at a time. wndSize_Max is the window size of the receiving window. The bool, stopped, and the channel, stopChan, are utilities used during the shutdown procedure.

```go
type GRDPConnection struct {
  addr      string
  Connected bool
  sw_out    SlidingWindow
  sw_in     SlidingWindow
}
```

Listing 5.11: The GRDPConnection struct

The GRDPConnection struct, shown in listing 5.11, includes a socket address and two SlidingWindow structs. One for incoming data and one for outgoing. The struct also contains a bool, called Connected, which is used during Ping-Pong.

```go
1  type SlidingWindow struct {
2    // Thread safety related variable
3    Mu sync.Mutex
4    // Data storing and passing related variables
5    Buffer  []*swPacket
6    outChan chan []byte
7    // Flow Control related variables
8    seqChan   chan uint32
9    NextWrite uint32
10   NextAck   uint32
11   maxSeqNr  uint32
12   // FC & Congestion Avoidance related variable
13   WindowSize uint32
14   // Time-out related variables
15   TimeoutDur time.Duration
16   varRtt     float64
17   sRtt       float64
18   alpha      float64
19   beta       float64
20   firstRtt   bool
21 }
```

Listing 5.12: The SlidingWindow struct

As, shown on line 5 of listing 5.12, the SlidingWindow struct contains a slices of pointers to swPackets (struct definition shown in listing 5.13). This slice is simply called Buffer. The outChan channel is connected to a transmission loop held by the GRDP. Anything which gets passed to this channel will be written to the GRDP's nullconn. The Mutex Mu is used to ensure that only one goroutine can access the SlidingWindow's Buffer at a time. The remaining fields of the sliding window are mostly numbers used by the different transport layer protocols of GRDP. Listing 5.12 includes comments which details which protocol the various variables are connected to.

```go
1 type swPacket struct {
2   timeStamp time.Time
3   msg       []byte
4   acked     bool
5   sending   bool
6   signaler  chan int
7 }
```

Listing 5.13: The swPacket struct

A swPacket contains a byte slice called msg. Seen on line 3 of listing 5.13. If the swPacket is connected to an outgoing SlidingWindow, msg is a network ready byte slice. It can be any of the 5 message types of GRDP, shown in section 4.4.2. If the swPacket is connected to an receiving SlidingWindow, msg will be a regular packet, stripped of it's GRDP header. The timeStamp field, of type time.Time, is used to record RTTs. The acked bool is a vital part of the flow control and message ordering protocols. The signaler channel is used to inform the retransmission protocol that a message has been acknowledged, thus stopping retransmission.

5.5 Receiving

It was mentioned in section 4.4.1 that when a GRDProtocol struct is created, it spawns a loop in which it listens for messages from the network. Specifically, a call to the receive method, shown in listing 5.14, is made in the factory function of GRDProtocol. The call is prefixed with the "go" keyword, meaning it is ran in a separate goroutine.

The method first readies a UDPConn, on line 7, to receive messages on. It then enters the listening loop on line 12. On the next line a byte slice with maximum capacity equal the maximum size of a UDP datagram is prepared. The process then tries to read from the UDPConn named sock. This read will timeout after half a second, as defined on line 14. This timeout is included to ensure that the loop regularly checks if the GRDP has been told to stop.

After a message has been received, the receivedDatagram byte slice gets resliced. On line 19, so that it is no larger than it has to be. The read returns r, which is the socket address of the remote node. However, the returned port number is a wildcard port number. This means that it does not represent the port which the remote node is actually listening to. All the Goxos nodes will be listening on the same port number, so on line 20, GRDP builds a remote address by appending its own port number to r's IP address. The remote address and the byte slice is then passed to another method for processing. When the method returns, the loop starts over.

```go
1   func (grdp *GRDProtocol) receive(listenAddr string) {
2     addr, err := net.ResolveUDPAddr("udp", listenAddr)
3     if err != nil {
4       glog.Fatalf("Couldnt resolve udpAddr (%v)", err)
5     }
6
7     sock, err := net.ListenUDP("udp", addr)
8     if err != nil {
9       glog.Fatalf("Couldnt listen on udp addr (%v)", err)
10    }
11
12    for !grdp.stopped {
13      receivedDatagram := make([]byte, headerSize+datagramSize)
14      sock.SetReadDeadline(time.Now().Add(500 * time.Millisecond))
15      length, r, err := sock.ReadFromUDP(receivedDatagram)
16      if err != nil {
17        glog.V(3).Infoln(err)
18      } else {
19        receivedDatagram = receivedDatagram[:length]
20        remoteAddr := r.IP.String() + ":" + strconv.Itoa(addr.Port)
21        grdp.processReceivedDatagram(remoteAddr, receivedDatagram)
22
23      }
24    }
25  }
```

Listing 5.14: The listening loop of GRDP

The listen loop passes the remote address and the received bytes to the processReceivedDatagram method. First, the remote address is used as key to retrieve a connection from GRDP's connections map. This retrieval is shown on line 3 of listing 5.15. If the corresponding connection is not found on the map, a check is done on line 24 to see if the message is a Ping. If it indeed is a Ping, a new connection is created and added to the map. The connections outgoing window is initialized and its transmission loop is initiated. Lastly, on line 31, a Pong is returned to the remote node.

If a connection was retrieved from the map on line 3, a switch on the message identifier is performed on line 6. If the message is a NACK, it will simply lead to the message with sequence number equaling the header number contained by the NACK, being acknowledged. This behavior is pretty naive, but there is only one scenario which has currently been encountered, in which an NACK is received. It happens when a packet is received by the receiving node, and the returning ACK is lost. The transmitting node will then retransmit the already received packet. If the sliding window of the receiver has already moved passed the packets sequence number, the packet will be deemed outside the current receiving window. This triggers the sending of a NACK.

If the message is a Ping, it most likely means the remote node had time to send multiple Pings before it received a Pong. It could also mean that the two nodes initiated Ping-Pong at the same time. In the case of the first scenario nothing will be done. In second scenario the node will initialize its outgoing window with the size specified by the received message's header number (second to fifth byte) and deem the remote node connected. This is also what happens if a Pong is received.

```
1  func (grdp *GRDProtocol) processReceivedDatagram(remoteAddr string, receivedDatagram []byte)
       {
2    grdp.mu.Lock()
3    c, found := grdp.connections[remoteAddr]
4    grdp.mu.Unlock()
5    if found {
6      switch uint8(receivedDatagram[0]) {
7        case 4:
8        c.sw_out.ackMsg(bytesToInt(receivedDatagram[1:5]))
9        case 3:
10       if !c.Connected {
11         c.initOutWindow(bytesToInt(receivedDatagram[1:5]))
12       }
13       case 2:
14       if !c.Connected {
15         c.initOutWindow(bytesToInt(receivedDatagram[1:5]))
16         c.sw_out.outChan <- append([]byte{3}, intToBytes(grdp.wndSize_Max)...)
17       }
18       case 1:
19       c.sw_out.ackMsg(bytesToInt(receivedDatagram[1:5]))
20       case 0:
21       grdp.processMessage(c, bytesToInt(receivedDatagram[1:5]), receivedDatagram[5:])
22     }
23   } else {
24     if uint8(receivedDatagram[0]) == 2 {
25       c := NewGRDPConnection(remoteAddr, grdp.wndSize_Max)
26       grdp.mu.Lock()
27       grdp.connections[remoteAddr] = c
28       grdp.connections[remoteAddr].initOutWindow(bytesToInt(receivedDatagram[1:5]))
29       grdp.mu.Unlock()
30       go grdp.transmit(c)
31       c.sw_out.outChan <- append([]byte{3}, intToBytes(grdp.wndSize_Max)...)
32     }
33   }
34 }
```

Listing 5.15: Processing and distributing the datagram

As with the NACK, if the message is of type ACK, the sequence number it contains will be sent to SlidingWindow's ackMsg method. If the message is a regular packet, a call will be made to GRDP's processMessage method. The method receives the connection, the sequence number and the payload of the packet as parameters. The call to processMessage can be seen on line 21.

5.6 Transmitting

Shortly after adding a new connection to it's map, GRDP makes a call to its transmit method, passing the conncetion as a parameter. The method is ran in its own goroutine. On line 8, of listing 5.16, the transmission loop is blocked until a message is passed to it through outChan. When a message is received through outChan, it is written to the network through GRDP's nullconn. This loop keeps repeating until a signal to stop is received from GRDP's stopChan.

```
1  func (grdp *GRDProtocol) transmit(c *GRDPConnection) error {
2    udpaddr, err := net.ResolveUDPAddr("udp", c.addr)
3    if err != nil {
4      return err
5    }
6    for {
7      select {
8        case msg := <-c.sw_out.outChan:
9        _, err = grdp.nullconn.WriteToUDP(msg, udpaddr)
10       if err != nil {
11         fmt.Printf("Error while transmitting (%v)", err)
12       }
13       case <-grdp.stopChan:
14       return nil
15     }
16   }
17 }
```

Listing 5.16: Transimssion loop, G

5.7 Adding Connections

The AddConnection method of GRDP, shown in listing 5.17, is called by the CM during the initial connection procedure. On line 2 a UDPAddr is resolved, using the address supplied through the method call. This is done to confirm that the address is valid. If the address is valid, GRDP checks if a connection with that address has already been added to it's map. If a connection is found the method simply returns. If not found, a new connection is created and added to the map. Then, on line 16, a Ping message is constructed. The Ping includes GRDP's wndSize_Max, which is its receiving window size. The connections outgoing window has not yet been initialized, meaning its outChan is unavailable. Therefor the Ping is sent through a call to the singleTransmit method. The singleTransmit method writes a single message to the GRDP's nullconn and return. Pings will be sent each half second until a Pong (or Ping) has been received from the remote node. When a Ping or Pong has been received, the connections outgoing window and outChan is ready to be used. GRDP can now spawn the connections transmission loop on line 22 of listing 5.17, and return.

```
 1  func (grdp *GRDProtocol) AddConnection(addr string) error {
 2    udpAddr, err := net.ResolveUDPAddr("udp", addr)
 3    if err != nil {
 4      return err
 5    }
 6    _, found := grdp.connections[udpAddr.String()]
 7    if !found {
 8      c := NewGRDPConnection(udpAddr.String(), grdp.wndSize_Max)
 9      grdp.mu.Lock()
10      grdp.connections[udpAddr.String()] = c
11      grdp.mu.Unlock()
12      var message []byte
13      message = append([]byte{2}, intToBytes(grdp.wndSize_Max)...)
14      for !c.Connected {
15        grdp.singleTransmit(c, message)
16        time.Sleep(500 * time.Millisecond)
17      }
18      go grdp.transmit(c)
19    }
20    return nil
21  }
```

Listing 5.17: GRDP adding a new connection

5.8 Outgoing Sliding Window

The implementation of a sliding window protocol, with Selective Repeat ARQ, involves multiple interconnected methods of the GRDP and SlidingWindow structs. This section will give a description of those methods.

5.8.1 SlidingWindow's seqChan

The SlidingWindos' seqChan is a channel which can store a number of sequence numbers equal to the outgoing window size. When the outgoing window is initialized a call is made to initSeqChan. The initSeqChan method fills seqChan with sequence numbers, as can been seen on line 2 - 4 of listing 5.18. Listing 5.18 also shows the increaseSeq, which provides a new sequence number for seqChan, given seqChan isn't already full. Lastly, the getSeq method is shown. It is used to request a sequence number from seqChan.

```
 1  func (s *SlidingWindow) initSeqChan() {
 2    for s.currentWindowContains(s.NextWrite) {
 3      s.seqChan <- s.NextWrite
 4      s.NextWrite = (s.NextWrite + 1) % s.maxSeqNr
 5    }
 6  }
 7
 8  func (s *SlidingWindow) increaseSeq() {
 9    s.seqChan <- s.NextWrite
10    s.NextWrite = (s.NextWrite + 1) % s.maxSeqNr
11  }
12
13  func (s *SlidingWindow) getSeq() uint32 {
14    nr := <-s.seqChan
15    return nr
16  }
```

Listing 5.18: seqChan related methods

Each time a call is made to GRDP's SendPacket method, which is shown in listing 5.19, a call is made on line 12 to the SlidingWindow's getSeq method after retrieving a conncetion from the connections map. If seqChan is empty, it will block the method until a new number is passed to it. How this happens will be explained in section 5.8.2. If seqChan has an available sequence number, the method can proceed to build a packet, using the number it got from seqChan as its sequence number. The packet is then passed to the SlidingWindow's addToBufferAndSend method together with the sequence number, in a new goroutine. This happens on line 15 of listing 5.19

```
1   func (grdp *GRDProtocol) SendPacket(addr string, data []byte) error {
2     udpAddr, err := net.ResolveUDPAddr("udp", addr)
3     if err != nil {
4       return err
5     }
6     grdp.mu.Lock()
7     c, found := grdp.connections[udpAddr.String()]
8     grdp.mu.Unlock()
9     if !found {
10      return errConnectionNotFound
11    }
12    nr := c.sw_out.getSeq()
13    var message []byte
14    message = append(append([]byte{0}, intToBytes(nr)...), data...)
15    go c.sw_out.addToBufferAndSend(nr, message)
16    return nil
17  }
```

Listing 5.19: GRDP's SendPacket method

On line 2 of listing 5.20, addToBufferAndSend creates and initializes a new swPacket, which wraps the packet. A pointer to the swPacket is added to the SlidingWindow's Buffer using the sequence number of the packet as index. Then on line 8, a call is made to the send method, passing the sequence number as parameter.

```
1   func (s *SlidingWindow) addToBufferAndSend(nr uint32, message []byte) {
2     s.Buffer[nr] = &swPacket{
3       signaler: make(chan int, 1),
4       msg:      message,
5       acked:    false,
6       sending:  false,
7     }
8     s.send(nr)
9   }
```

Listing 5.20: SlidingWindo's addToBufferAndSend method

5.8.2 Recursive Retransmission

The first thing that happens when a call is made to send, is that a time-stamp is added to the swPacket, in the Buffer index specified by the sequence number. Then, on line 3 of listing 5.21, the packet, contained by the swPacket, is passed to a transmission loop through outChan. The method then waits for the signaler to indicate that the message has been acknowledged. When it receives a signal, the method returns. If a signal is not received within the duration, calculated in listing 5.25, a timeout has occurred. in the case of a timeout, a call will first be made to the decrease the outgoing window size. Then a recursive call will be made to send, using the same sequence number as parameter.

```go
func (s *SlidingWindow) send(nr uint32) {
  s.Buffer[nr].timeStamp = time.Now().UTC()
  s.outChan <- s.Buffer[nr].msg
  s.Buffer[nr].sending = true
  select {
    case <-s.Buffer[nr].signaler:
      s.Buffer[nr].sending = false
    case <-time.After(s.TimeoutDur):
      s.decreaseWindow()
      s.send(nr)
  }
}
```

Listing 5.21: The recursive send method

When a packet is sent and an ACK is successfully returned. The header number of the ACK, representing the sequence number of the packet its acknowledging, is passed to the SlidingWindow's ackMsg method. Line 2 of listing 5.22 shows the RTT being recorded by subtracting the time-stamp of the swPacket, in the SlidingWindow's Buffer, from the current time. The RTT is passed to updateTimeoutDur, which updates the timeout duration. Next, the outgoing window size is increased and an int is passed to the swPacket's signaler, stopping retransmission.

Line 7 and 9 - 13 of listing 5.22 shows the actual algorithm which slides the outgoing window. First, the swPacket at the nr index of Buffer is deemed acknowledged on line 7. If it is not next in line to be slid passed, nothing more happens and the method return. However, if it is indeed the next in line, line 11 will be reached. The packet has now played its part and the swPacket at the nr index of Buffer is deemed unacknowledged. NextAck, which indicates the number next in line to be slid by, is increased and a call is then made to increaseSeq. If the swPacket at the next index in the Buffer is also acknowledged the same steps from line 11 to 13 will be repeated. This will repeat until an unacknowledged swPacket is encountered.

```go
1   func (s *SlidingWindow) ackMsg(nr uint32) {
2     rtt := time.Now().UTC().Sub(s.Buffer[nr].timeStamp)
3     if rtt != 0 {
4       s.updateTimeoutDur(rtt)
5     }
6     s.increaseWindow()
7     s.Buffer[nr].acked = true
8     s.Buffer[nr].signaler <- 1
9     if nr == s.NextAck {
10      for s.Buffer[s.NextAck].acked {
11        s.Buffer[s.NextAck].acked = false
12        s.NextAck = (s.NextAck + 1) % s.maxSeqNr
13        s.increaseSeq()
14        if s.Buffer[s.NextAck] == nil {
15          break
16        }
17      }
18    }
19  }
```

Listing 5.22: Processing an ACK

If a goroutine is being blocked on line 12 of the SendPacket method, shown in listing 5.19, it now receives the sequence number newly provided to seqChan by the call to increaseSeq on line 13 of listing 5.22. It can now proceed to send its packet.

5.9 Incoming Sliding Window

The algorithm implemented by GRDP's processMessage method is pretty similar to the algorithm of ackMsg. As explained in section 5.5, when a packet is received, its sequence number and payload gets passed to processMessage. First the method confirms that the sequence number, represented by the nr parameter, is within the current receiving window of the SlidingWindow. A scenario where this check returns false was described in section 5.5.

If the sequence number is within the window, the payload is added to the incoming Buffer, at index nr. As seen on line 4 of listing 5.23, the payload is wrapped by a swPacket struct with its acked bool set to true. In the case of the incoming SlidingWindow, the swPacket's acked bool does not signify that a packet has been acknowledged. The naming is a consequence of the same bool field of swPacket being used to signify different things depending on which type of SlidingWindow its used in (outgoing or incoming). In processMessage the acked bool is used to indicate that a packet has been received, and that the packets payload, not yet passed to the Demuxer, is in the Buffer at that specific index.

```
1  func (grdp *GRDProtocol) processMessage(c *GRDPConnection, nr uint32, data []byte) {
2    c.sw_in.Mu.Lock()
3    if c.sw_in.currentWindowContains(nr) {
4      c.sw_in.Buffer[nr] = &swPacket{msg: data, acked: true}
5      if nr == c.sw_in.NextAck {
6        for c.sw_in.Buffer[c.sw_in.NextAck].acked {
7          grdp.packetsIncomingBuffer <- c.sw_in.Buffer[c.sw_in.NextAck].msg
8          c.sw_in.Buffer[c.sw_in.NextAck] = &swPacket{acked: false}
9          c.sw_in.NextAck = (c.sw_in.NextAck + 1) % c.sw_in.maxSeqNr
10         if c.sw_in.Buffer[c.sw_in.NextAck] == nil {
11           break
12         }
13       }
14     }
15     message := append([]byte{1}, intToBytes(nr)...)
16     c.sw_out.outChan <- message
17   } else {
18     message := append([]byte{4}, intToBytes(nr)...)
19     c.sw_out.outChan <- message
20   }
21   c.sw_in.Mu.Unlock()
22 }
```

Listing 5.23: Processing a packet

If nr is not the next in line to be passed to the Demuxer, an ACK containing the sequence number of the packet is returned to the packets transmitter. In the case where nr is the next in line to be passed to the Demuxer, the payload gets passed to GRDP's packetsIncomingBuffer channel. A channel which the Demuxer is constantly reading from, through calls to GRDP's readPacket method. The readPacket method is shown in listing 5.24. It should be noted that before the Demuxer receives the payload, the CM reads and strips the CM signature from the payload. After the payload has been passed to packetsIncomingBuffer, an almost empty swPacket, containing only acked, initialized to false, is placed in the Buffer slot which previously contained the payload. Then on line 9 of listing 5.23, the NextAck variable, representing the sequence number which is next in line to be passed to the Demuxer is increased. If the next packet is already in the Buffer, the steps will repeat. This happens until an empty Buffer slot is encountered. After this an ACK is returned for the received packet, and the method returns.

```go
func (grdp *GRDProtocol) ReadPacket() []byte {
  select {
    case <-grdp.stopChan:
    var s []byte
    s = make([]byte, 9, 9)
    return append(append(s, intToBytes(99)...), []byte("GRDP stopping")...)
    case b := <-grdp.packetsIncomingBuffer:
    return b
  }
}
```

Listing 5.24: GRDP's ReadPacket method

5.10 Timeout Calculation

The time which GRDP wait before retransmission is based on TCP's timeout calculation rules [16], which was explained in section 4.4.6. Each time a message is acknowledged, the RTT gets recorded and passed to the method shown in listing 5.17. The if block from line 8 to line 10, ensures that we get the absolute value of the difference between the SRTT and RTT.

The calculations is done with float64 representations of seconds. On line 10, the resulting number is multiplied with 10^9 to get a number which represents the duration in nanoseconds. Nanoseconds are the basic units of Go's time.Time, in other word, the finest granularity possible in Go. The check on line 14 ensures that the RTO does not get smaller than the specified minimum RTO.

```go
 1  func (s *SlidingWindow) updateTimeoutDur(rtt time.Duration) {
 2    if s.firstRtt {
 3      s.sRtt = float64(rtt.Seconds())
 4      s.varRtt = float64(rtt.Seconds() / 2)
 5      s.firstRtt = false
 6    }
 7    abs := s.sRtt - rtt.Seconds()
 8    if abs < 0 {
 9      abs = -abs
10    }
11    s.varRtt = ((1 - s.beta) * s.varRtt) + (s.beta * abs)
12    s.sRtt = ((1 - s.alpha) * s.sRtt) + (s.alpha * rtt.Seconds())
13    s.TimeoutDur = time.Duration((s.sRtt + (4 * s.varRtt)) * 1000000000)
14    if s.TimeoutDur < minimumRTO {
15      s.TimeoutDur = minimumRTO
16    }
17  }
```

Listing 5.25: Calculating the timeout duration

5.11 Congestion Avoidance Algorithms

The congestion avoidance works by increasing and decreasing the outgoing window based on some specific rules. The call to increaseWindow is made when a packet is acknowledged, as shown on line 6 of listing 5.22. The call to decreaseWindow is made when a timeout occurs, shown on line 9 of listing 5.21. Though only the AIMD algorithm is included in the final code base, three variations of congestion avoidance have been implemented and tested. First we have the slow start algorithm, shown at line 2 to 10 in listing 5.26. The rules are simple; when decreaseWindow is called, set window size to 1. When increase window is called, increase window size by 1. In addition to this, a call to decreaseWindow is made when the outgoing buffer is first initialized.

```
1  //------------------SLowStart---------------------
2  func (s *SlidingWindow) decreaseWindow() {
3    s.WindowSize = 1
4  }
5  func (s *SlidingWindow) increaseWindow() {
6    s.WindowSize += 1
7    if s.WindowSize > s.maxSeqNr/2 {
8      s.WindowSize = s.maxSeqNr / 2
9    }
10 }
11 //--------------------AIMD------------------------
12 func (s *SlidingWindow) decreaseWindow() {
13   s.WindowSize = s.WindowSize / 2
14   if s.WindowSize <= 0 {
15     s.WindowSize = 1
16   }
17 }
18 func (s *SlidingWindow) increaseWindow() {
19   s.WindowSize += 1
20   if s.WindowSize > s.maxSeqNr/2 {
21     s.WindowSize = s.maxSeqNr / 2
22   }
23 }
24 //--------------------MIMD------------------------
25 func (s *SlidingWindow) decreaseWindow() {
26   s.WindowSize = s.WindowSize / 2
27   if s.WindowSize <= 0 {
28     s.WindowSize = 1
29   }
30 }
31 func (s *SlidingWindow) increaseWindow() {
32   if s.WindowSize == 1 {
33     s.WindowSize += 1
34   }
35   s.WindowSize += s.WindowSize / 2
36   if s.WindowSize > s.maxSeqNr/2 {
37     s.WindowSize = s.maxSeqNr / 2
38   }
39 }
```

Listing 5.26: Slow Start

Next we have AIMD, shown at line 12 to 23 in listing 5.26. In this implementation of AIMD, window size is cut in half each time a call to decreaseWindow is made. When a call to increaseWindow is made, window size is increased by 1.

Lastly we have MIMD, shown at line 25 to 39 in listing 5.26. When a call is made to decreaseWindow, the window size is cut in half. When a call is made to increaseWindow, the window is increased by half its current size. If the current window size is 1, the window will be increased by 1. This is to prevent the window size getting stuck at 1. Which can happen because 1 divided by 2 returns a 0 (integer).

Chapter 6

Experiments

This chapter details the experiments which were ran to test GRDP, and select some of its features. We will first describe the test setup, before moving on to display and discuss the results of the experiments.

By running the experiments, we wish to find which version of the different mechanism provides the optimal throughput for GRDP. Throughput is measured as the number of client request which are processed and completed each second. Such measurements will be denoted by req/sec.

6.1 setup

The Goxos framework has been used for various other research experiments. It therefor includes an, already implemented, key-vale-store application, called kvs, for the purpose of running experiments. The kvs application is a RSM, using Goxos.

There is also an application included called kvsc. This is a client for kvs, which can be setup to e.g. generate and send a certain amount of randomly generated write requests to kvs.

Throughput is recorded by the Server module of Goxos. For our experiments this happens every 25 milliseconds.

There is also a script included for running the experiments. This script had to be modified slightly, as it had previously been used for experiments involving replica failure and recovery.

The script first prepares 3 kvs replicas on 3 physical machines. It then prepares 5 physical machines with 26 kvsc-clients each. Each of the 130 clients proceed to send 3000 randomly generated write requests to the kvs cluster. When a 60 seconds has passed, the script checks if the states of the replicas are identical. If not, the test run is deemed invalid. After this, the script generates a HTML-report, containing the measured throughput of the run. This whole process is repeated for some specified number of times. In our case 10 or 20 times.

6.1.1 Hardware

The experiments are run on a cluster of identical machines. The machines are located in on of the labs of the University of Stavanger. The machines have the following specifications:

CPU Intel Xeon E5606 @ 2.13 Ghz. (4 cores)

Memory 16 GB

Operating System CentOS 6.8 (6-8.el6.centos.12.3.x86_64)

Network Gigabit

6.1.2 Parameters

In this section we list testing parameters which are unchanged throughout the experiments.

Alpha The alpha variable determents the amount of pipelining used. More information about alpha and the pipelining in Goxos can be found in Helleland's thesis from 2014 [10].

Batch Size Batch Size determines how many messages can be batched together before being passed through the Paxos module. As with alpha, batch size has also been set to a value which makes the experiments conform to earlier experiments. The batch size is set to 10. For more information about batch size and the batching in Goxos, see [10] .

Key Size and Value Size The key and value are the data payload of the experiments. For the experiments the key and value size are set to 8 bytes. This means the total payload size is 16 bytes.

Paxos Implementation All the tests use the Multipaxos implementation of the Paxos algorithm.

Replicas The number of computers acting as servers. Out experiment use 3 Replicas.

Clients The number of clients sending requests to the replicas. For the experiments there are 5 physical machines. Each physical machine runs 26 instances of the client program. This means there are a total of 130 virtual clients.

Number of Requests A request is read or write request sent by a client to the replicas. Each client in the setup sends 3000 requests. This gives a total of 390000 per 60 second run.

Run duration and Total Number of Runs Each experiment run is 60 seconds. Each experiment, except the final comparison between the original implementation and GRDP, consists of 10 runs.

6.2 Results

In this section the results obtained through the experiments will be presented and discussed. First some details around failed runs will be given. Next the results of experiments done to select and tune the various features of GRDP is presented. Lastly, the results of the final experiment, which compares the original Goxos with the final incarnation of GRDP will be presented and discussed.

6.2.1 Failed Runs

For all the experiments testing GRDP, the number of valid runs ranged from 7 to 10. The average being somewhere around 8.5 valid runs per 10 runs. A run is deemed valid if no clients or replicas crash, and the replica states are identical at the end of the run. Table 6.1 shows that the average number of valid runs per 10 runs, during the window size experiment in section 6.2.2, is 8.6429. Most failed runs are due to an unresolved bug connected to the methods of the test applications MapRequest struct. Paxos messages sometimes reach their resend limit, it is suspected that this is also connected to the bug in the test application.

Window Size	Runs	Valid Runs
1	10	7
2	10	9
3	10	8
4	10	7
5	10	10
6	10	7
7	10	10
8	10	8
9	10	9
10	10	10
25	10	8
50	10	10
500	10	10
1000	10	8
Average:	10	8.6429

Table 6.1: Table of valid runs from the window size experiment in section 6.2.2

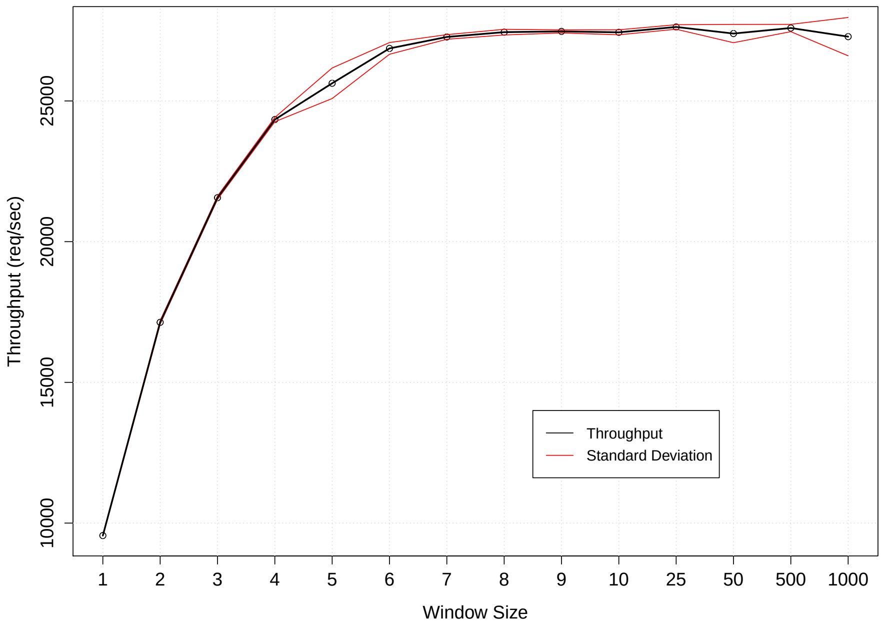

Figure 6.1: A graph showing the result when testing for throughput with different window sizes.

6.2.2 Window Size

First we ran experiments using different window sizes. For this experiment we use the BSON encoding scheme and AIMD for congestion avoidance.

When the window size is set to 1, we are, in effect, using Stop-and-Wait, which was presented in section 2.3.1. As figure 6.1 shows, simply increasing the window size from 1 to 2 almost doubles the throughput, going from 9000 to 17000 req/sec. Increasing the window size further yields diminishing increases in throughput until the graph flattens out at around 27000 req/sec, after reaching a window size of 8. This means that the increase in window size from 1 to 8 produced a 200% increase in throughput.

To ensure that the window size does not become a bottleneck in the other experiments, only window sizes larger than 10 will be used. As a default, we set window size to 100.

6.2.3 Congestion Avoidance

Next we tested the different congestion avoidance algorithms. The results are presented in figure 6.2. For this experiment we used the BSON encoding scheme and set the maximum window size to 500.

Unfortunately, congestion conditions were not encountered. This could be because of some bottleneck limiting the throughput, that the experiment was ran in a

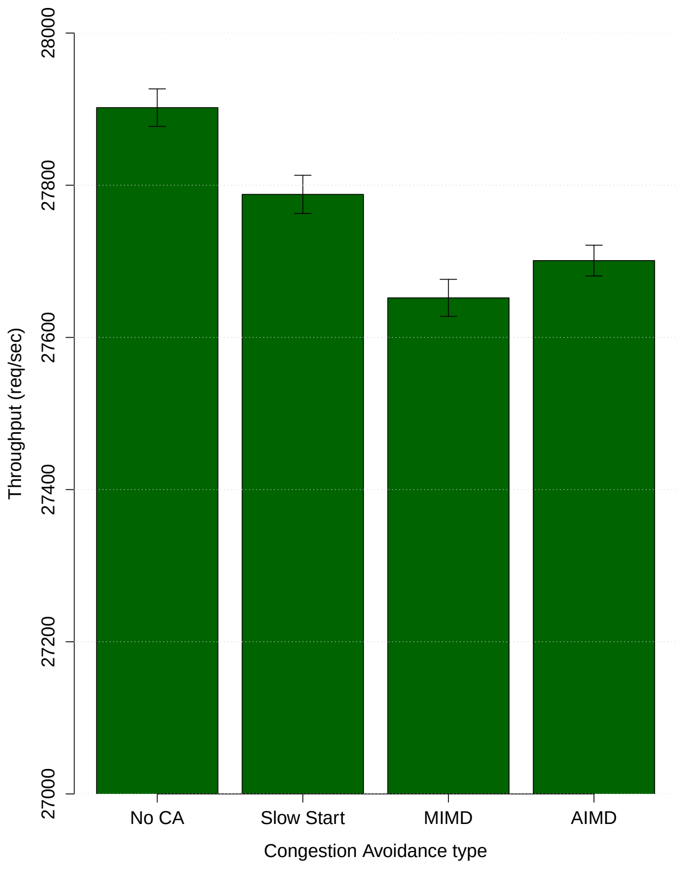

Figure 6.2: A bar graph showing the result when testing for throughput with different congestion avoidance algorithms. The graph includes standard deviation.

cluster environment, or a combination of the two. Due to this, the best results were attained by not using any congestion avoidance. This makes intuitive sense. In a scenario where congestion never occurs, removing the overhead associated with avoiding it should give better throughput. Even so, the algorithms performed almost identically, with only around 300 req/sec separating the lowest results from the highest. When recorded throughput is at around 27000 req/sec, 300 req/sec is almost negligible.

As the experiment does not indicate a clear best-performer, we fall back on the research done by Chiu and Jain [15]. As such we select AIMD as the default congestion avoidance for GRDP.

6.2.4 Encoding Schemes

For the experiments ran to test the different encoding schemes we use AIMD for congestion control and a window size of 50.

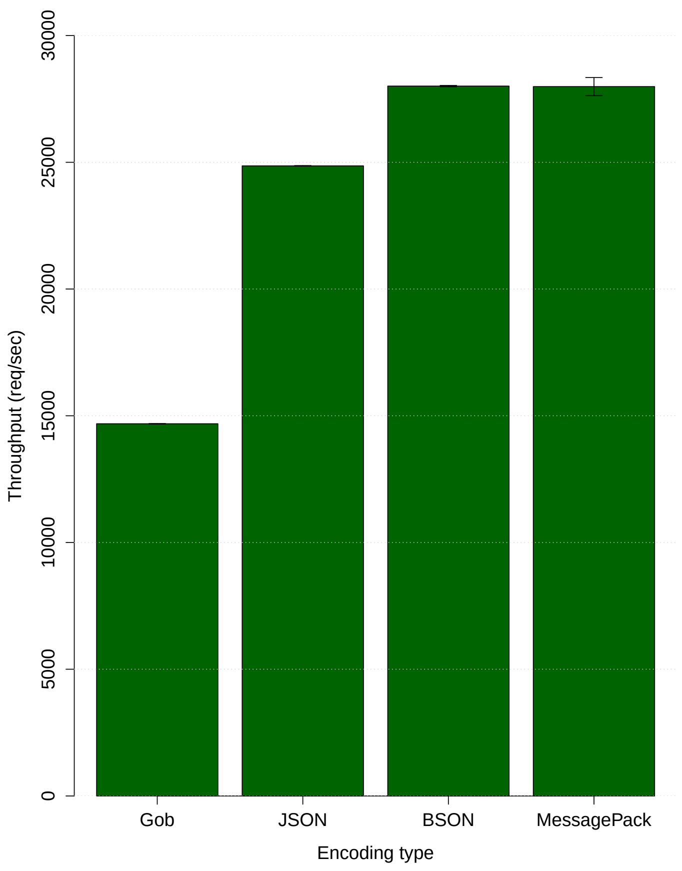

Figure 6.3: A bar graph showing the result when testing for throughput with different encoding schemes. The graph includes standard deviations.

As shown in figure 6.3, the Gob scheme delivers the lowest throughput. In actuality, Gob is the fastest serialization method among the four included in the GRDP [22]. However, as explained in section 5.1, Gob is meant for stream-based communication, and our implementation has to create a new Gob encoder and decoder for each packet it sends. This results in very poor throughput while using the Gob scheme.

the XSON scheme performs significantly better than the Gob scheme. The JSON version achieves a throughput at around 25000 req/sec, while the BSON version surpass this with around 3000 req/sec, ending up with a throughput at around 28000 req/sec.

Lastly we have the MessagePack scheme. At 28000 req/sec, its performance is equal that of the BSON scheme. This makes selecting MessagePack as GRDP's default encoding scheme easy. It is tied for highest performance, it is the implementation with lowest amount of code lines and adding new custom types is simple and easy, requiring only a single line of code.

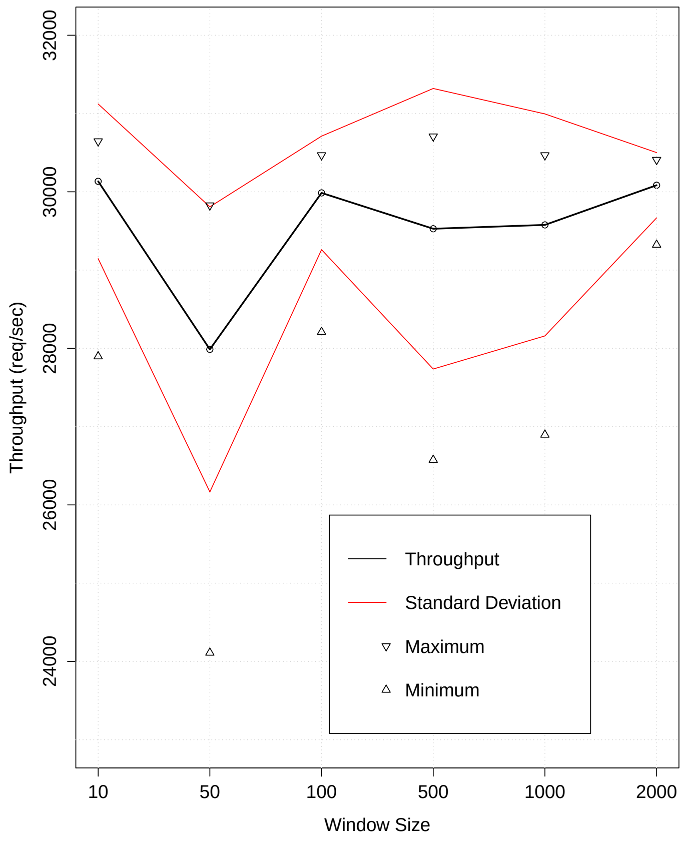

Figure 6.4: A graph showing the result when testing for throughput with MessagePack encoding and different window size.

A closer look at MessagePack

Seeing as the MessagePack scheme was selected as the default encoding scheme in GRDP, we decided to take a closer look at its performance. For this experiment we used AIMD congestion avoidance, and tested throughput at different window sizes.

The results show that throughput achievable by the MessagePack scheme is actually around 2000 req/sec higher than that of the BSON scheme. The graph in figure 6.4 shows that a throughput closer to 30000 req/sec is actually achieved.

There is some standard deviation observed. To explain this, figure 6.4 also includes the best and worst run from each test. The average throughput is close to the maximum throughput achieved for each test, while the minimum is quite far from the average. This signifies that a few runs with significantly lower throughput than the average contributes to a lower average and higher standard deviation, while most runs actually performs close to the maximum.

Upon closer examination of the individual runs, it was observed that in a few runs, a single client out of the 130 clients does not connect to the replicas until all the other 129 clients are done sending all their requests. When only this one client is being served, its suspected that not enough requests are supplied for the batcher to have any effect. As such a lower throughput is observed. These cases are rare, and as such the average throughput is expected to approach around 30000 req/sec in the long run.

6.3 Final Experiment

Finally, we compare GRDP with the original implementation of Goxos. For this experiment GRDP uses the mechanisms and configurations which has been selected as default. AIMD is used for congestion avoidance. Window size is set to 100 and we use the MessagePack encoding scheme.

For the last experiment we use 20 valid runs for each protocol to make the comparison. This meant running different amount of runs for the two protocols, as the original failed around 50% of the time, thanks to the bug in the testing application, which was mentioned in 6.2.1.

Most of GRDP's runs were valid, but two runs where a client connects after all others have finished, were removed. The runs in question are not representative of the actual throughput, as not enough packets are being supplied to the replicas for a certain period of time, resulting in throughput recordings as low as 11000 req/sec. Below, in table 6.2 the results of the 20 valid runs are displayed.

Run	Throughput (Original)	Throughput (GRDP)
1	39897	30528
2	39897	30891
3	40310	30891
4	39897	27036
5	39694	30891
6	39897	30952
7	39593	30769
8	40102	31075
9	39897	30952
10	39897	30891
11	39795	27177
12	39593	30891
13	40414	30891
14	39593	30830
15	40102	30648
16	35944	30830
17	39593	30708
18	39897	24111
19	39897	31075
20	39795	30708

Table 6.2: Table of throughputs from the final experiment. Throughput is measured in request per second

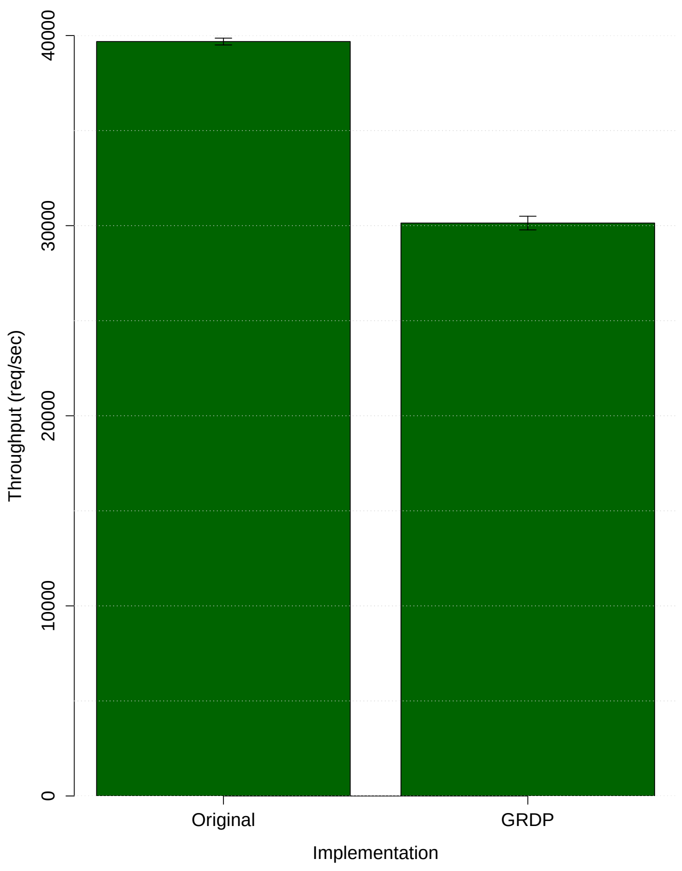

Figure 6.5: The final comparison between the original implementation and GRDP.

Final Comparison

The average throughputs and standard deviations of the final experiment are shown in figure 6.5. The original Goxos implementation achieves a throughput of 39685 req/sec, with a standard deviation of 908, while GRDP achieved a throughput of 30137 req/seq with a standard deviation of 1829. This means GRDP delivers 75,94% of the originals throughput. In other word, we have unfortunately not achieved our goal of increasing the performance of Goxos. Even so, this is a decent result, which could be improved in various ways in the future. To this end, some suggestions will be given in chapter 7

We suspect the major bottleneck of GRDP to be the encoding scheme. Some benchmarking test show that when implemented as intended, Gob, which the original uses, is approximately 35% faster than MessagePack [22].

The results also indicates that the assumption stated in section 6.2.4, concerning the throughput approaching 30000 req/sec in the long run, is correct, as the GRDP's throughput is actually over 30000 req/sec.

Chapter 7

Future Work

This chapter describes some possible future work related to improving the performance and functionality of GRDP. All three recommendations could significantly increase the performance and efficiency of GRDP.

7.1 Encoding

Encoding and decoding are suspected of being the major bottlenecks of GRDP. Benchmark tests of different encoding methods, performed by Alec Thomas, shows that multiple methods outperforms MessagePack, using less then half the time of MessagePack to encode and decode data [22]. A new encoding scheme would only involve creating a new Sender and Demuxer at the top level. A possible substitute for the current MessagePack encoding scheme could be one using Google's protocol buffers or the extended version called gogoprotobuf [24].

7.2 Multicast

By being based on UDP, it is possible to extend GRDP with multicast. The larger a cluster of RSM replicas get, the more they could benefit from the use of multicast. Currently GRDP uses unicast, meaning transmission is one-to-one. In the current implementation of GRDP, when a node issues a broadcast request, a packet is written to the network for each connection. With multicast, there would always be just a single write, as multicast is a one-to-many method of transmission.

7.3 Batching

Goxos already has Batching for the messages passing through its Paxos module. In his thesis from 2014, Helleand show the great effect this had on the performance of Goxos [10]. Implementing a similar mechanism for GRDP is possible. GRDP could potentially gain a large increase in throughput by batching a number of packets together, before sending them over the network.

Chapter 8

Conclusion

While being fault tolerance is important for a cloud computing system, it is also vital that the system is able to make progress. This thesis has provided a new communications protocol which ensures that a Paxos-based RSM framework, specifically Goxos, can continue to make progress. The protocol achieves this by selectively retransmits lost packets based on timeouts, calculated with the use of round trip time measurements. The protocol also provides mechanisms for ordering out-of-order packets before passing them to the application, as well as a mechanism for avoiding congestion. In other words, the protocol is prepared for harsh network conditions. All this is achieved with the use of a small, 5 byte, header. Which means that bandwidth is mainly being used to transmit what is actually important, the payload.

Various experiments were performed, testing and comparing different mechanisms. Through the experiments, an encoding scheme based on MessagePack was selected, and additive-increase/multiplicative-decrease was included to provide congestion avoidance. We saw the great benefit of implementing a sliding window protocol, and we saw the great drawback of using Gob for datagram-based communication.

Though the new protocol does not surpass the original Goxos framework when it comes to performance, it delivers a viable throughput, not far from the original's. In addition, by being implemented in user space, and having a modular, layered structure, the new protocol is open and accessible for further modifications and improvements.

Bibliography

[1] J. Postel, [RFC793] *Transmission Control Protocol*, Internet Engineering Task Force, 1981.

[2] J. Postel, [RFC768] *User Datagram Protocol*, Internet Engineering Task Force, 1980.

[3] G. Fairhurst, L. Wood , [RFC3366] *Advice to link designers on link Automatic Repeat reQuest (ARQ)*, Internet Engineering Task Force, 2002.

[4] Leslie Lamport, *The Part-Time Parliament*, ACM Transactions on Computer Systems 16 (2)(p.133-p.169), 1998.

[5] International Organization for Standardization, ISO/IEC 7498-1, *Open Systems Interconnection - Basic Reference Model: The Basic Model*, 1994.

[6] Jim Roskind, *QUIC: Design Document and Specification Rationale*, `https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit`, 2013 [Online; accessed 15-February-2016]

[7] The Go Project, *The Go Programming Language*, `https://www.golang.org`, 2015 [Online; accessed 15-February-2016]

[8] Stephen Michael Jothen, Tormod Erevik Lea, *Goxos: A Paxos implementation in the Go Programming Language*, Department of Electrical Engineering and Computer Science, University of Stavanger, Norway 2012.

[9] Tormod Erevik Lea, *Implementation and Experimental Evaluation of Live Replacement and Reconfiguration*, Department of Electrical Engineering and Computer Science, University of Stavanger, Norway 2013.

[10] Knut Helleland, *Parallellisering og andre optimaliseringer på en Paxos-implementasjon*, Department of Electrical Engineering and Computer Science, University of Stavanger, Norway 2014.

[11] Nicolai M. Josuttis, *SOA in Practice, The Art of Distributed System Design*, O'Reilly, Cologne, Germany 2007.

[12] William Stallings, *DATA AND COMPUTER COMMUNICATIONS*, 8th edition, Pearson Education, Inc., Upper Saddle River, NJ, 2007.

[13] Douglas E. Comer, *Internetworking With TCP/IP Vol I: Principles, Protocols, and Architecture*, 6th edition, Pearson Education, Inc., New Jersey, NJ, 2000.

[14] Van Jacobson, Congestion avoidance and control. In *Proceedings of SIGCOMM*, ACM, Stanford, CA, 1988.

[15] Dah-Ming Chiu, Raj Jain *Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks*, Digital Equipment Corporation, Littleton, MA, 1989.

[16] V. Paxson, M. Allman, J. Chu, M. Sargen [RFC6298] *Computing TCP's Retransmission Timer*, Internet Engineering Task Force, 2011.

[17] D. Borman, B. Braden, V. Jacobson, R. Scheffenegger [RFC7323] *TCP Extensions for High Performance*, Internet Engineering Task Force, 2014.

[18] IEEE-USA *Next Generation Internet: IPv4 Address Exhaustion, Mitigation Strategies and Implications for the U.S.* , 2009.

[19] Gustavo Niemeyer, *gobson BSON serialization for Go*,
`https://labix.org/gobson`,
2011 [Online; accessed 20-may-2016]

[20] Sadayuki Furuhashi, *MessagePack*,
`http://msgpack.org/`,
2013 [Online; accessed 26-may-2016]

[21] Vladimir Mihailenco, *MessagePack encoding for Golang*,
`https://github.com/vmihailenco/msgpack`,
commit: 6915de5f21af6f5bcd2517da00d99c5131bb1da7
2013 [Online; accessed 26-may-2016]

[22] Alec Thomas, *Benchmarks of Go serialization methods*,
`https://github.com/alecthomas/go_serialization_benchmarks`
2016 [Online: accessed 11-june-2016]

[23] Google, *Protocol Buffers*
`https://developers.google.com/protocol-buffers/`
2016 [Online: accessed 11-june-2016]

[24] Walter Schulze, *gogoprotobuf, Protocol Buffers in Go with Gadgets*
`http://gogo.github.io/`
2016 [Online: accessed 11-june-2016]

List of Figures

Listings

Appendix A

Running the Code

1. Make sure ‘$GOPATH‘ is set locally.
2. Uncompress the code attached to this pdf into the "$GOPATH/src" directory.
3. Install BSON with: $ go get gopkg.in/mgo.v2/bson
4. Install MessagePack with: $ go get gopkg.in/vmihailenco/msgpack.v2
5. Install Fabric (http://fabfile.org/) and its Python dependencies.
6. Navigate to "$GOPATH/src/github.com/relab/goxos/exp/exprun".
7. Make sure the prerequisites defined in the README.md file are met.
8. Create an experiement description, defined in an INI file. See example-exp.ini for available options.
9. Run the scrip using either the default or a custom output path:
   (a) Using the default output path (‘/tmp/kvs-experiments‘):
   $ fab run_exp:example-exp.ini
   (b) Using a custom output path:
   $ fab run_exp:example-exp.ini,local_out=/tmp/somedir

Appendix B

Attachments

Source code: attached to pdf