nanoarq

Charles Nicholson <charles.nicholson@gmail.com>

Abstract

This document introduces `nanoarq`, a single-file C library that provides reliability over an unreliable communications channel. `nanoarq` implements the Selective Repeat ARQ algorithm and provides basic functionality for establishing and gracefully destroying connections. `nanoarq` is meant to be suitable for embedded systems, with a design focus on simplicity, flexibility, and ease of integration. The `nanoarq` implementation is released into the public domain.

Cite ARQ

Contents

1 Introduction

Many communications channels in embedded systems, such as UART lines between multiple CPUs, or between a target and host system, provide an unreliable transport for transmitting and receiving data. Bits can be altered in flight on the wire, in isolation or in bursts. Crosstalk and signal degradation can occur when the routing of critical signals is too long, or the signals are transmitted over cables. Even bytes that are transmitted without errors can be lost due to infrequent servicing of the transport layer, overwritten in a hardware register by the arrival of the next byte. Finally, application backpressure can cause valid incoming bytes to be discarded, as the system runs out of space to store them.

All of these problems speak to the necessity of a reliability layer in software. Sliding window protocols are the ubiquitous solution, and are used as the foundation for more complex protocols like TCP. Fundamentally, sliding window protocols guarantee that the application layer will be presented with all data that was sent to it, in order, with integrity and without duplicates.

`nanoarq` is an implementation of the "Selective Repeat ARQ" protocol, and uses an explicit ACK mechanism to retransmit lost or corrupted data. Additionally, `nanoarq` provides connectivity services; a standard handshake and FIN/ACK disconnect strategy can be optionally used to establish and statefully manage a connection.

1.1 Goals

Reliability
First and foremost, `nanoarq` provides reliability to unreliable communications. As long as the physical link still exists between the two endpoints, `nanoarq` ensures that data will be transmitted in-order, without corruption, and without duplication of data.

Simplicity
`nanoarq` does as little work as possible to achieve its functionality, intentionally eschewing complex and powerful behavior that is present in TCP. `nanoarq` is not a general-purpose networking toolkit; it exists only to provide reliability over a predictable link with reasonable

performance between two endpoints.

Transparency

nanoarq does not hide implementation details from the user, or enforce a notion of encapsulation. nanoarq is not object-oriented; all internal data structures are accessible to users. In the case the provided API does not offer sufficient functionality, it should be as easy as possible to access, manipulate, and extend nanoarq's state.

Ease of Integration

It can be difficult to integrate third-party C libraries into a user application. The C language does not have a unified build system, which can have a "Tower of Babel" effect on attempts to reuse code. While packages like CMake and Ninja are growing in popularity, assuming their presence puts a significant burden on a would-be user. Additionally, deploying libraries on Windows with Visual Studio has the extra complexity of having to provide support for the static (/MT) and dynamic (/MD) versions of the Microsoft C Runtime. While some C libraries prefer to provide support for as many build systems as possible, nanoarq aims to be as easy to integrate by providing no build system, instead existing in a single header file. This approach is prevalent and proven in Sean Barrett's stb_* libraries.

A similar challenge exists at runtime. nanoarq is intended to be used in embedded systems, which have no standard operating system. Some systems run on so-called "bare metal" with a static task scheduling algorithm, while others may use sophisticated pre-emptive RTOS's that provide concurrency primitives, conditional waits, and work queues. nanoarq is designed to work in any environment, as long as the user can provide the amount of time that has elapsed since the last polling call was made.

Finally, the nanoarq implementation and tests are released into the public domain, which means there are no licesnses or fees for use.

1.2   Non-goals

Implementing a Large Standard

nanoarq does not aim to be a full TCP, IP, PPP, or POSIX-compatible sockets implementation. nanoarq has no concept of routing, endpoints, ports, addresses, sockets, or names. nanoarq clients do not

statefully listen for connections; if a connection request arrives, it is serviced.

Security

    `nanoarq` strives to be resilient against malformed input, but provides no encryption or authentication services. If security is required, it is up to the user to ensure that all data transmitted via `nanoarq` is properly secured.

Dynamic Configuration

    `nanoarq` assumes that both endpoints are compatibly configured. `nanoarq` provides no services for communicating connection options during connection establishment.

Congestion Control

    `nanoarq` is oblivious to the concept of congestive collapse, and performs no dynamic throttling of data in response to data loss. `nanoarq` relies on the application to address the problems of congestion and backpressure.

Physical Link Management

    `nanoarq` does not assume control over any specific communications hardware or OS resources. The user is responsible for the actual low-level transmission of bytes into and out of `nanoarq`. This allows `nanoarq` to function on embedded systems without requiring intimate knowledge of a given communication peripheral block, as well as on larger systems like desktop computers, across multiple operating systems, etc.

Error Correction

    `nanoarq` does not currently support any form of forward error correction (e.g. Reed-Solomon, Turbocodes, etc.). A FEC encoding and reconstruction phase would be an interesting future addition to support environments that have an extremely high retransmission cost.

2   Protocol

`nanoarq` implements a connection-oriented stream-based protocol; both endpoints maintain connection state and the internal delimiting and segmenting of data is not exposed to the application layer. As with the POSIX sockets

API, clients call high-level functions similar to `send()` and `recv()`. There is no parity guarantee between the number of send and receive calls between the transmitter and receiver.

Internally, `nanoarq` breaks up transmissions into fixed-size blocks so that they can be selectively acknowledged. The atomic unit of transmitted data is the `frame`. The portion of the frame that carries the user data is called the `segment`. A `message` is an aggregation of frames, and the unit of acknowledgement (`ACK`).

An entire message worth of frames is transmitted before the receiver sends an ACK. The receiver's ACK response contains a bitfield of which segments were successfully received. This serves as both a positive and negative (`NAK`) acknowledgement, and allows the sender to only retransmit the failed frames.

`nanoarq` uses sequence numbers to identify messages instead of bytes. Similarly, ACK numbers and window advertisements refer are measured in message units. A zero-based index is used to identify which frame is being transmitted, and a bitfield is used for ACKing which frames in a message were received.

It is the user's responsibility to configure `nanoarq` to find a balance between message size and window size to maximize bandwidth, minimize unnecessary ACK traffic, and make aggressive forward progress in poor environments.

2.1 Frames

Frames contain a header, a segment, and an optional checksum footer. The frame header carries enough information to uniquely identify which message it belongs to. It also carries the sender's receive window size for flow control purposes. The frame layout is visible in figure 1. Frames are encoded using the COBS algorithm to ensure that frame delimiting is unambiguous, but COBS does impose a maximum frame size of 254 bytes. Frame headers and footers consume a maximum of 16 bytes, so the maximum payload size per segment is 238 bytes. Accordingly, `nanoarq` imposes a constant 6.7% overhead on transmission size.
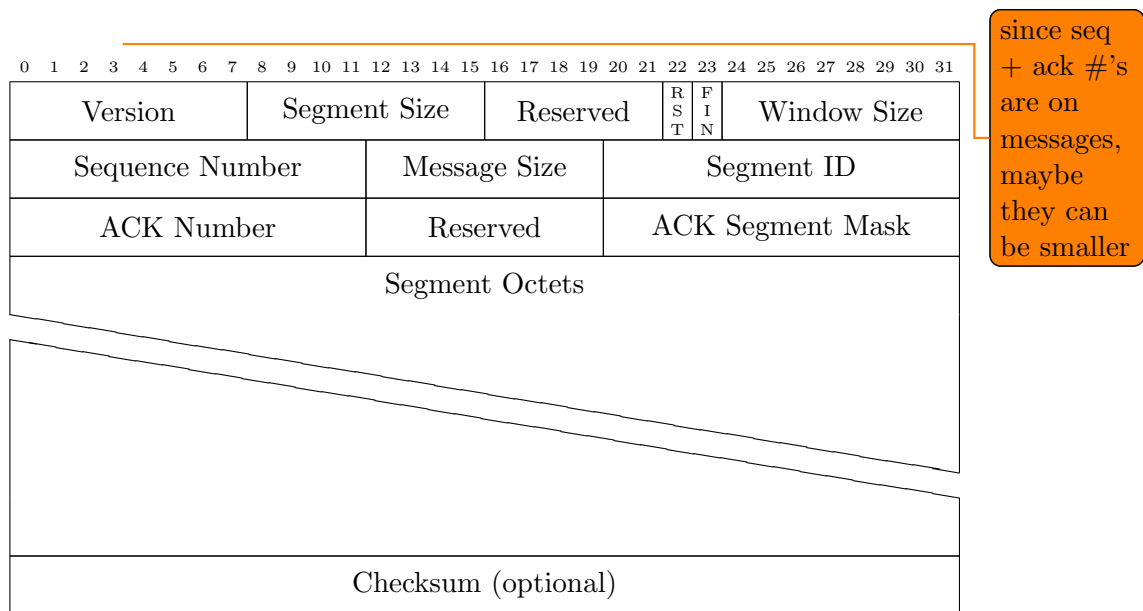
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Version	Segment Size	Reserved	R S T	F I N	Window Size
Sequence Number		Message Size		Segment ID	
ACK Number		Reserved		ACK Segment Mask	
Segment Octets					
Checksum (optional)					

since seq + ack #'s are on messages, maybe they can be smaller

Figure 1: `nanoarq` frame layout

A description of each field follows.

Version

Internal `nanoarq` protocol version number.

Segment Size

Size, in bytes, of the segment contained in the current frame.

RST Flag

Set to indicate a new connection or the resetting of an existing connection.

FIN Flag

Set to initiate or cooperate in a graceful disconnection.

Window Size

The number of messages the sender is currently capable of receiving.

Sequence Number

The identity of the current message being sent. Sequence numbers identify messages and not segments.

Message Size
 Size, in segments, of the current message being sent.

Segment ID
 Zero-based index in the current message of the segment being sent.

ACK Number
 The identity of the message being acknowledged by the sender.

ACK Segment Mask
 A bitfield identifying which segments of the message identified by the ACK Number field have been received.

Payload Octets
 The user data carried by the `nanoarq` protocol.

Checksum
 An optional field used to validate the integrity of the current frame.
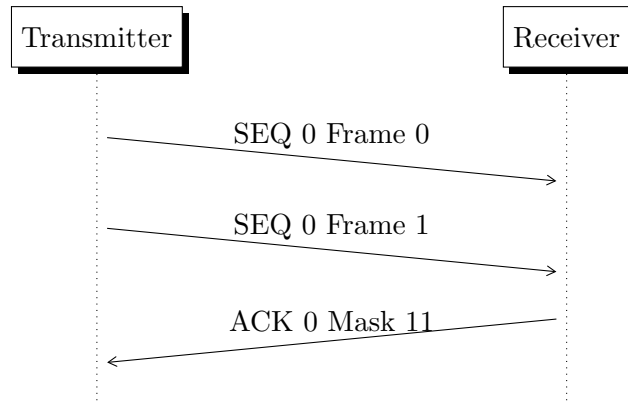
2.2   Messages

Messages are aggregations of segments. `nanoarq` acknowledgements can be both positive (ACK) or negative (NAK), depending on the contents of the ACK Segment Mask field in the frame header. When a message is acknowledged, the ACK Segment Mask bitfield will contain 1's for all successfully received segments, and 0's for segments that were not received. This allows `nanoarq` to make as much progress as possible over faulty channels, since only failed frames are retransmitted.

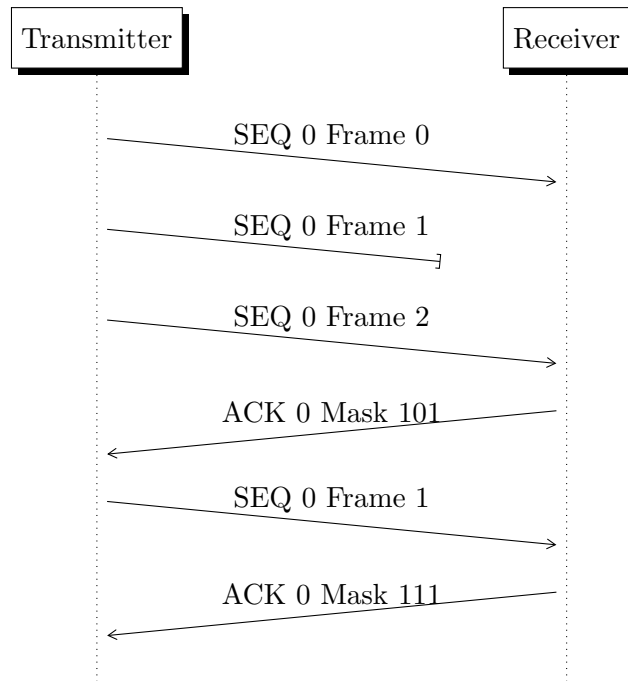 The following examples display how various scenarios are handled in `nanoarq`.
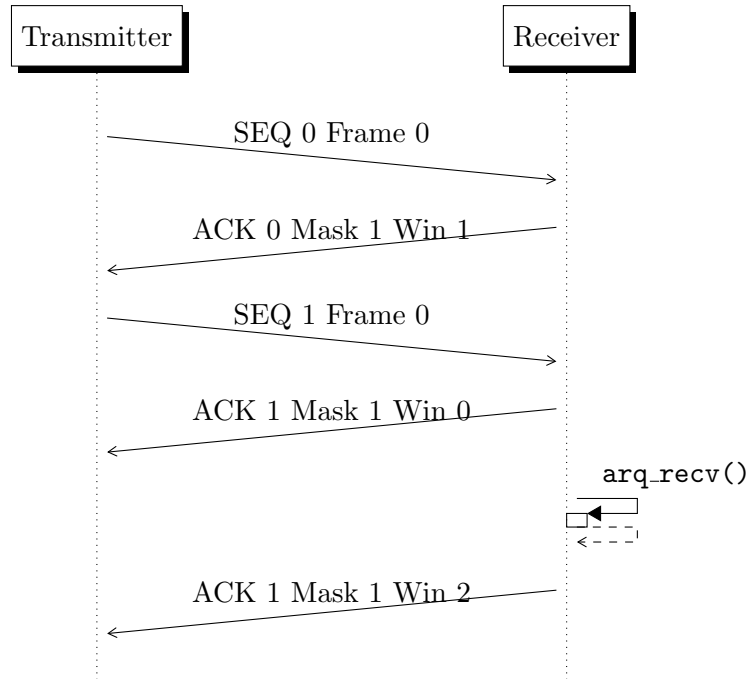
Example 1 *Message is one frame, successfully received*
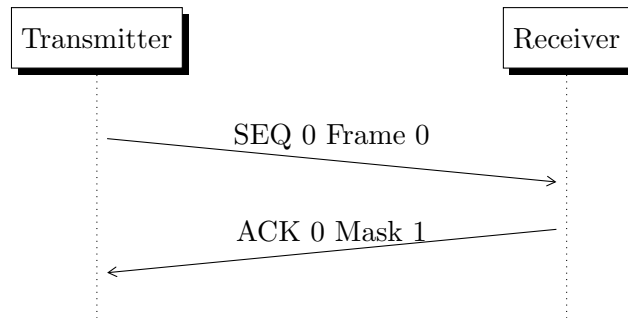
Example 2 *Message is two frames, successfully received*

Example 3 *Message is three frames, middle frame fails, retransmitted*

Example 4 *Message is one frame, two messages sent, window size of two*

Transmitter   Receiver

SEQ 0 Frame 0

ACK 0 Mask 1 Win 1

SEQ 1 Frame 0

ACK 1 Mask 1 Win 0

arq_recv()

ACK 1 Mask 1 Win 2

2.3 Flow Control

nanoarq supports basic flow control by using window advertisements and message-level acknowledgement. All flow control is static; nanoarq does not dynamically throttle data based on sampling round-trip times. nanoarq does attempt to avoid Silly Window Syndrome, but these mechanisms can be overridden.

Each frame carries a window advertisement. This field contains the number of messages the sender has capacity for and is willing to receive. Since fully-received and acknowledged messages remain in the receiver's window until the application has retrieved them via the public API, it's necessary to communicate the window size explicitly to avoid transmitting data that has no hope of being received.

Silly Window Syndrome is avoided during transmission through a cite SWS user-configurable "tinygram" transmission delay, which specifies the amount of time to wait before sending less than one message worth of data. This delay can be explicitly overridden via the nanoarq public API, in the case

where there are high-priority small messages to send.

Receive-side Silly Window Syndrome manifests when the application receives very few bytes at a time from the network stack, causing the network layer to transmit a large number of ACK messages, driving the link efficiency towards zero. `nanoarq` only ACKs messages and not bytes, so it is naturally immune to receive-side SWS. Unfortunately, this immunity does not come for free! The tradeoff is that `nanoarq` is not maximally efficient with respect to flow control: if all but the final segment in a message has been received up to the application layer, `nanoarq` will not advertise the empty space in its receive window as available.

2.4 Connection State Machine

`nanoarq` optionally provides stateful connection services for establishing, maintaining, and closing a connection to the peer endpoint. The state machine, detailed in figure 2, is similar to TCP but differs slightly. `nanoarq` has no concept of a client or a server, so the `LISTEN` state present in TCP does not exist. There is also no distinction between connecting and resetting, so the difference between `RST` and `SYN` is uninteresting. Additionally, `nanoarq` sequence numbers start at zero, so the standard TCP 3-way handshake can be simplified to a 2-way handshake.
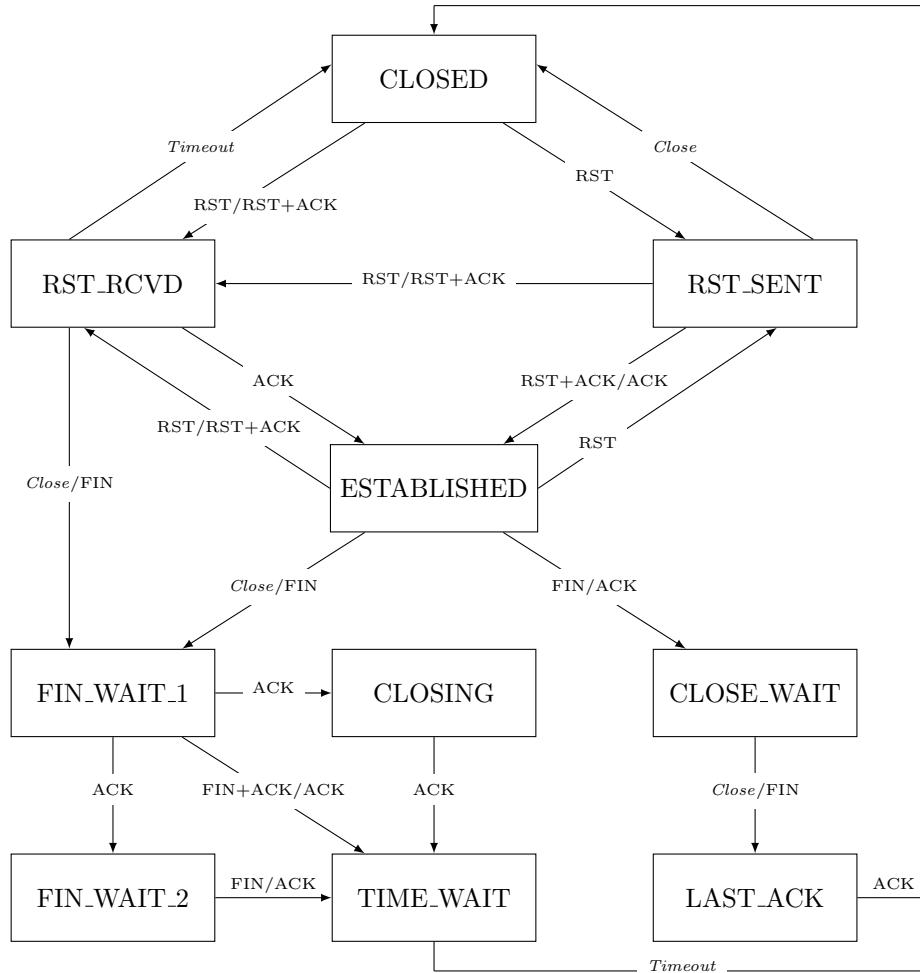
fix diagram to be 2-way

Figure 2: `nanoarq` state machine

3 Integration

3.1 Architecture

3.2 Physical Organization