NASA/CR-2009-215703
NIA Report No. 2008-09

Design and Verification of a Distributed Communication Protocol

César A. Muñoz and Alwyn E. Goodloe
National Institute of Aerospace, Hampton, Virginia

April 2009

NASA/CR-2009-215703
NIA Report No. 2008-09

Design and Verification of a Distributed Communication Protocol

César A. Muñoz and Alwyn E. Goodloe
National Institute of Aerospace, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Cooperative Agreement NNX08AE37A

April 2009

DESIGN AND VERIFICATION OF A DISTRIBUTED COMMUNICATION PROTOCOL[*]

César Muñoz [†] and Alwyn E. Goodloe [‡]

CONTENTS

1 INTRODUCTION

AirSTAR [17, 2] is an integrated flight test infrastructure which utilizes remotely piloted, powered subscale models for flight testing developed at NASA's Langley Research Center (LaRC). When flying, commands from the ground-based pilot are broadcast to the aircraft and telemetry data from the aircraft are broadcast to the ground station. The goal of this

[*]This work was supported by the National Aeronautics and Space Administration under NASA Cooperative Agreement NNX08AE37A.

[†]Senior Staff Scientist, National Institute of Aerospace (NIA), 100 Exploration Way, Hampton VA, 23666, Email: `munoz@nianet.org`

[‡]Postdoctoral Associate, National Institute of Aerospace (NIA), 100 Exploration Way, Hampton VA, 23666, Email: `Alwyn.Goodloe@nianet.org`

work is to design and verify a communication protocol that satisfies AirSTAR's safety and operational requirements:

- It is of paramount importance that commands from the pilot be treated by the communication system as urgent. That is, they are to be broadcast to the aircraft as soon as possible and should be processed at the aircraft with all due speed. This means that there should be no buffering of these data at either the sender or receiver. This requirement is called as the *weak delivery requirement* since data lost in transmission should not be retransmitted as they would be considered stale by the time they arrived at the pilot.

- Engineers on the ground need to receive all telemetry data produced by the aircraft in order to analyze aircraft performance as well as to plan future aircraft flights. In contrast to the previous case, the protocol should guarantee that all of telemetry data are eventually delivered. This requirement is called as the *guaranteed delivery requirement.*

Since the requirements of weak and guaranteed delivery are in some sense orthogonal to each other, we can structure the solution as two different protocols: the *weak delivery protocol* (WDP) and the *guaranteed delivery protocol* (GDP). In addition to these two protocols, other protocols are needed to support communication between the aircraft and ground station. As usual in protocol design, this collection of protocols is structured in a *protocol stack* (see Figure 1).

This document contains a description of a protocol that satisfies both the weak delivery and the guaranteed delivery requirements. This protocol is intended for the AirSTAR remotely operated vehicle. In addition to the high-level description of the protocol stack, we provide an overview of its formal specification and verification. For readability purposes, we use standard mathematical notation as much as possible. However, the mathematical development presented here has been formally specified and verified in the Prototype Verification System (PVS) [19]. This development is electronically available from `http://research.nianet.org/fm-at-nia/AirSTAR`.

Section 2 gives a brief overview of the protocol stack. The layers of the protocol stack, i.e., application layer, WDP, GDP, and link layer, are specified in Section 3. Section 4 shows how sender and receiver processes, that conform to the stack protocol, are specified. The formal verification that the proposed protocol satisfies the weak and guaranteed delivery requirements is described in Section 5.

2 PROTOCOL STACK

Protocols are generally structured in layers, where each layer handles a different aspect of message processing [24]. As a message moves down the stack, each layer performs some processing and adds packet headers. As a message moves up the stack, the corresponding packet headers are removed. The classic ISO seven layers model partitions the stack into a physical layer, link layer, network layer, transport layer, session layer, presentation layer, and application layer. Protocols for embedded systems used in the aerospace arena are usually not sophisticated enough to warrant but a few layers. For instance, in our case, the protocol simply broadcasts information so there is no need for a network layer, which performs routing. The layers of our protocol stack roughly correspond to the application

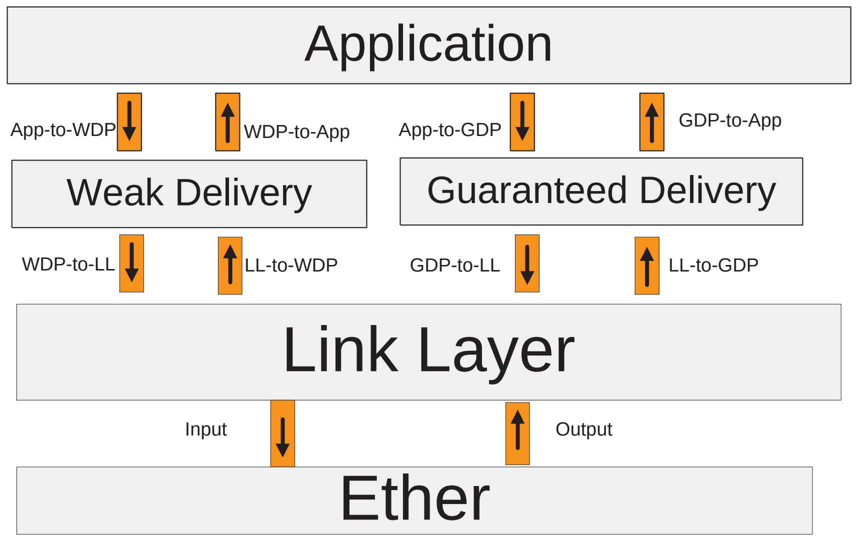

Figure 1: *Protocol stack*

layer, transport layer, link layer, and physical layer. Given that we do not currently have any details about the physical layer, it will not be explicitly modeled in this document. However, we do model an abstract communication medium that we call ether.

The proposed protocol stack is composed of the four layers illustrated in Figure 1. At the top is the application layer. The AirSTAR flight-computer software as well as the ground-control software are presumed to reside here. All messages sent from the application layer are sent via either WDP or GDP depending on whether weak or guaranteed delivery is required. The application is assumed to decide which protocol is used to send which type of message. The next layer down corresponds to the transport layer and it is here that the WDP and GDP protocols reside. WDP simply sends a message, but provides no guarantee that the message ever arrived at its destination. Hence, messages may be lost or corrupted in transit and are never resent. GDP is designed to provide its user with a guarantee that any message sent is eventually received, and that they are received in the same order they are sent. The differences between WDP and GDP are similar to the differences between UDP and TCP, but both are considerably simpler. The link layer is the next layer in our protocol stack. Note that the WDP and GDP protocols directly interface with the link layer as there is no network layer. The link layer performs error detection and multiplexes outgoing messages from the WDP and GDP layers and demultiplexes the incoming messages. The last layer is the ether, which correspond to the communication medium.

3 SPECIFICATION OF PROTOCOL LAYERS

In our model, the protocol layers are connected using First-In First-Out (FIFO) queues. In Figure 1, each queue is represented by a small rectangle with an arrow pointing in the direction of the information flow. The queues are named `x_to_y` indicating that the data moves from layer `x` to layer `y`. The names of the queues are given as follows:

`App_to_WDP`	`WDP_to_App`	`App_to_GDP`	`GDP_to_App`
`WDP_to_LL`	`LL_to_WDP`	`GDP_to_LL`	`LL_to_GDP`

The `WDP_to_LL`, `LL_to_WDP`, `GDP_to_LL`, and `LL_to_GDP` queues form the link interface (see 3.4).

In PVS, these queues are defined as *finite sequences*, i.e., a record with two fields: `length`, which contains the length of the queue, and `seq` an array indexed by $0 \ldots \texttt{length} - 1$ containing the data. The top of the queue is indexed by 0. Although an implementation would most likely adapt a more efficient scheme than FIFO queues, the behavior of frames moving from one layer to another would remain the same.

The ether, which represents the communication medium, interacts with the protocol stack through two unidirectional communication channels: the `input` channel that flows from the protocol stack to the medium, and the `output` channel that flows from the medium to protocol stack. The `input` and `output` channels form the ether interface (see 3.5). In PVS, these channels are defined as *bags*, i.e., multisets reflecting the possibility of duplicate messages and out of order delivery.

Before examining each layer in some depth, let us illustrate how a message sent via WDP flows down the stack, over the ether, and back up the stack. The node sending the message is denoted as the *sender* and the node receiving the message is denoted the *receiver*. The sender's application layer places the message into the `App_to_WDP` queue. The WDP layer processing removes it and places it into the `WDP_to_LL` queue. The link layer will remove this and place it into ether `input`. The link layer at the receiver will remove this message from the ether `output` and place it into the receiver's `LL_to_WDP` queue. The receiver's WDP layer will process the message placing it in the `WDP_to_App` queue from which the application on the receiving side removes it for processing.

3.1 Application Layer

The application layer is where all user and flight computer applications reside. We do not know what specific applications currently exist or what applications may be created in the future so this layer is treated quite abstractly as is traditional in the networking community. This layer may be composed of many processes, but we assume that at both the ground station and the aircraft, only a single process handles the processing of incoming messages. This simplification is realistic based on the current operation of the aircraft and allows us to forgo a complex socket-like mechanism.

In PVS, the application layer is modeled as two pairs of queues. The `App_to_WDP` queue at the sender that contains the data that must be sent in a timely fashion, e.g., via WDP, and the `APP_to_GDP` queue containing data that must be sent reliably, e.g., via GDP. The `WDP_to_APP` and `GDP_to_APP` queues hold data that have been received from WDP and GDP, respectively. It is assumed that the initial state of `App_to_WDP` and `APP_to_GDP` at the sender contain all of the data that will be sent and the `WDP_to_APP` and `GDP_to_APP` queues at the receiver are empty. This allows us to formulate a correctness criteria for the weak delivery protocol and the guaranteed-delivery protocol in terms of relations stating that the `WDP_to_APP` queue in the receiver side is a subset of the `App_to_WDP` queue when the sender was in its initial state and that the `GDP_to_APP` queue in the receiver side is a prefix of the `App_to_GDP` queue when the sender was in its initial state. The use of the initial state of the sender in this formulation is necessary because the queues will have data removed as the protocol executes.

3.2 Weak Delivery Protocol (WDP)

WDP is designed to satisfy the weak delivery requirement. The goal is to send and deliver messages to and from the application level as promptly as possible. In this application domain, stale data is useless so if a message is corrupted or lost in transmission it should not be resent. Nor should messages be buffered at either the sender or the receiver. This is a very basic protocol that simply moves data from the application layer to the link layer without really performing any processing. The sender and receiver are modeled as separate processes.

The state of WDP sender is defined as

$$\begin{aligned}\texttt{WDPSender} &= \texttt{App\_to\_WDP} : \texttt{fifo[Data]} \times \\ &\quad \texttt{link} : \texttt{LinkInterface} \times \\ &\quad \texttt{ether} : \texttt{EtherInterface} \times \\ &\quad \texttt{nop} : \texttt{Boolean},\end{aligned}$$

forming a tuple of the queue containing data to be sent, an interface to the link layer holding the `WDP_to_LL` and `LL_to_WDP` queues, an interface to the ether, and a Boolean that is true if no action is taken.

The WDP sender protocol is defined as a state transition function that maps the current WDP sender state to the next state:

$$\texttt{WDPSenderNext:}\quad \texttt{WDPSender} \longrightarrow \texttt{WDPSender}.$$

The function behaves as follows. If the `App_to_WDP` queue is nonempty, then remove the next message from `App_to_WDP` and add it to the `WDP_to_LL` queue in the `link` interface.

The state of the WDP receiver is defined as

$$\begin{aligned}\texttt{WDPReceiver} &= \texttt{WDP\_to\_App} : \texttt{fifo[Data]} \times \\ &\quad \texttt{link} : \texttt{LinkInterface} \times \\ &\quad \texttt{ether} : \texttt{EtherInterface} \times \\ &\quad \texttt{nop} : \texttt{Boolean},\end{aligned}$$

forming a tuple of the queue where the received data will be placed, an interface to the link layer holding the `WDP_to_LL` and `LL_to_WDP` queues, an interface to the ether, and a Boolean that is true if no action is taken.

The WDP receiver is defined as a state transition function that maps the current WDP receiver state to the next state:

$$\texttt{WDPReceiverNext:}\quad \texttt{WDPReceiver} \longrightarrow \texttt{WDPReceiver}.$$

The function behaves as follows. If the `LL_to_WDP` queue in the `link` interface is nonempty, then remove the next message from that queue and add it to the `WDP_to_App` queue.

3.3 Guaranteed Delivery Protocol (GDP)

GDP is designed to satisfy the guaranteed delivery requirement. Intuitively, GDP should ensure that messages are delivered in the same order as they are sent. In order for the sender to know that a message has been received, the recipient must send back an acknowledgment. In protocols such as TCP, this acknowledgment is often piggybacked on a message sent to the original sender rather than using a dedicated acknowledgment. Given that a significant percentage of the traffic flow will be from the aircraft to the ground station, this does not seem like the best design choice. In addition, rather than acknowledging a single message, it seems more appropriate to acknowledge receipt of a contiguous block of data since this keeps down the number of acknowledgment packets, which is desirable in this application domain. Hence, GDP is a sliding-window protocol [24] with block acknowledgment [9].

3.3.1 Sliding-Window Protocol

In general, sliding-window protocols have the following characteristics:

- Each message has a sequence number that acts as an identifier.
- The protocol receiver process acknowledges the receipt of data messages by sending an acknowledgment message to the sender.
  - If the sender has not received an acknowledgement that a message has been received in a predefined time, then a timeout will occur and the protocol will resend that message.
  - If a receiver has already received and acknowledged a message, but the same message (defined as having same sequence number) is received again, then the system will resend the acknowledgement, but nothing is done with the data since it has already been processed. This covers the situation where an acknowledgement message is lost or corrupted in transit and the sender resends a message.
- There is an upper bound sw on the number of data messages that can be sent without receiving acknowledgment for any of them. There is also an upper bound rw on the number of data messages that can be received without sending an acknowledgment. The value of rw should be chosen so that $rw \leq sw$. The value sw is called the sender's window size and the value rw is called the receiver's window size.

The protocol sender maintains a bounded buffer called `ackd`. This buffer has two fields: a data field and a Boolean mask field. The GDP protocol moves data from the `App_to_GDP` queue to the `ackd` buffer's data field when it is to be broadcast and initializes the mask field to false. The buffer index indicates the sequence order in which messages are sent. Each data entry is broadcast to the destination, which, at some point in time, sends a response acknowledging the receipt of some contiguous block of sequence numbers. The mask values are set to true for those data entries that have been acknowledged.

The variable ns is a pointer to the sequence number of the next data item to be sent and the variable na is a pointer to the first sequence number that has yet to be acknowledged. That is, sequence numbers $0, \ldots, na - 1$ have all been acknowledged as received by the

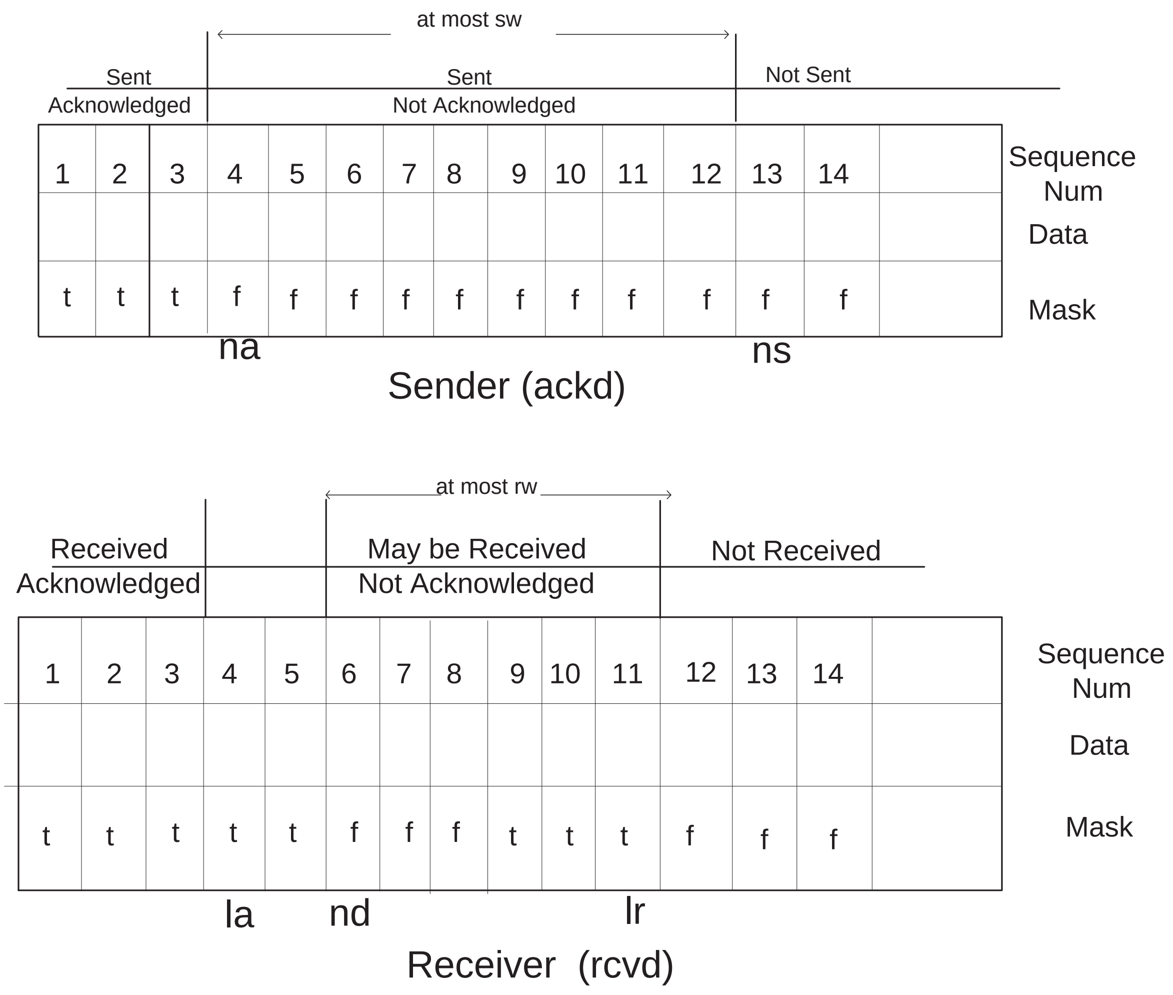

Figure 2: *Sliding window*

sender, but sequence number na has not yet been acknowledged. An invariant $na \leq ns \leq na + sw$ is maintained by the sender saying that the window of sent but not acknowledged data is of size at most sw. The sender will not send messages with a sequence number greater than $na + sw$ until data message na is acknowledged. The sender may receive acknowledgments for sequence numbers s, where $na \leq s < ns$, in any possible order; yet, only when acknowledgments for the contiguous sequence numbers $na, \ldots, x$, where $x < ns$, have been received, is the value of na slid forward to $x + 1$.

The receiver also maintains a bounded buffer `rcvd` containing data received from the sender. The protocol moves data sent by the sender from the `LL_to_GDP` queue into the `rcvd` buffer and sets the mask value at that position to true. The data is indexed by the sequence numbers. The variable la points to the last acknowledged sequence number, i.e., acknowledgment messages have been sent for sequence numbers $0, \ldots, la-1$. The variable nd points to the lowest sequence number that has yet to be delivered to the `GDP_to_App` queue. The variable lr points the highest sequence number that has been received and $lr \leq nd+rw$. The receiver ignores messages with sequence numbers greater than $nd + rw$. When the receiver has received the contiguous block of sequence numbers $nd, \ldots, x$, the pointer nd is slid forward to $x + 1$. Note that messages $la, \ldots, nd - 1$ have been received, but not yet acknowledged. Periodically, GDP sends an acknowledgment message that acknowledges the receipt of messages $la, \ldots, nd - 1$ and la is reset to nd.

Let us illustrate how the protocol works using the example from Figure 2. Assume that the sequence numbers $1, 2$, and 3 have been sent and acknowledged so $na = 4$ and $la = 4$. From the diagram we see that sequence numbers $4, \ldots, 12$ have been sent, but have not necessarily been delivered to the receiver. Assume that sequence numbers 4 and 5 have been sent and received, but not yet acknowledged. Let us also assume that $9, 10$, and 11 are delivered to the GDP receiver process, but $6, 7$, and 8 were lost in transit. At some point the sender times out and resends these messages. This situation is illustrated in Figure 2. When sequence numbers $4, \ldots, 11$ have been received by the receiver, the pointer nd slides forward to 12. When the acknowledgment message is sent by the receiver acknowledging the receipt of the sequence numbers $4, \ldots, 11$ the receiver now sets la to 12. If the acknowledgment is lost in transit, the sequence numbers $4, \ldots, 11$ would eventually be resent and, although the messages would be ignored, acknowledgments would be resent.

3.3.2 Bounded Buffers

In PVS, we model the windows `ackd` and `rcvd` as bounded buffers. The parameter `maxsize` defines the upper-bound on the length of the buffer. The type definition for our buffer is as follows:

$$\begin{aligned}\texttt{Window} \;=\; & \texttt{length:}0 \leq \texttt{length} \leq \texttt{maxsize} \\ \times\; & \texttt{mask:}\{\texttt{s:}\;\; \texttt{ARRAY}[(0, \ldots, \texttt{maxsize} - 1) \rightarrow \texttt{Boolean}] \\ & \quad \mid \forall (i : (\texttt{length}, \ldots, \texttt{maxsize} - 1)) : \neg\, \texttt{s}(i)\} \\ \times\; & \texttt{data:ARRAY}[(0, \ldots, \texttt{maxsize} - 1) \rightarrow \texttt{Data}]\end{aligned}$$

where mask is a finite sequence of Boolean values and data is a finite sequence of data. Both sequences are indexed by an integer value of at most $\texttt{maxsize} - 1$. The protocol only cares about data indexed by $0, \ldots, \texttt{length} - 1$. In particular, we assume that $\texttt{mask}(i)$ is equal to false for $i \geq \texttt{length}$. In the case of the ackd buffer, $\texttt{mask}(i)$ is true if and only if an acknowledgment has been received by the sender for sequence number $na + i$. In the case of the rcvd buffer, $\texttt{mask}(i)$ is true if and only if a message has been received by the receiver with the sequence number $nd + i$.

Recall that when acknowledgments for sequence numbers $na, \ldots, x$ have been received, na is slid forward to $x + 1$ and similarly when messages with sequence numbers $nd, \ldots, y$ have been received, nd is slid forward to $y + 1$. The process of readjusting the window is handled by the two functions first_false and slide. The function first_false returns f such that value of the buffer's mask at $0, \ldots, f - 1$ are all true, but the mask value at f is false. The lower pointer, either nd or na, of the buffer is moved to the first-false value f. The function slide takes as parameters a buffer of length n and an index to the first false in that buffer f and returns a new buffer of size $n - f$ where index i in the new buffer holds the contents of index $i + f$ in the old buffer. So as long as $f > 0$, the new buffer has a length smaller than the original.

In an actual implementation, the bounded nature of the buffer would be handled somewhat differently because sequence numbers must also be bounded. The traditional solution is to use modulo arithmetic, but this greatly complicates the complexity of the model and we feel that our design decision to model the finite buffer as we did is a reasonable modeling trade-off.

3.3.3 Sender

The sender process implements the procedure for the sliding window protocol sender outlined above. The window ackd is implemented as an instance of Window. The state of the sender is defined as

$$\begin{aligned} \texttt{GDPSender} \; = \; & \texttt{App\_to\_GDP} : \texttt{fifo[Data]} \times \\ & \texttt{winsender} : \texttt{WinSenderPrivate} \times \\ & \texttt{link} : \texttt{LinkInterface} \times \\ & \texttt{ether} : \texttt{EtherInterface} \times \\ & \texttt{nop} : \texttt{Boolean}, \end{aligned}$$

forming a tuple of the queue holding data to be sent, the local pointers and bounded buffers of the sender side of the sliding-window protocol, the interface to the link layer containing the two queues GDP_to_LL and LL_to_GDP, the interface to the ether, and a Boolean that is set to true if no action is performed.

The sender process is modeled as a state transition function that takes the current state of the GDP sender and an action to be performed, and returns the next state:

$$\texttt{GDPSenderNext:} \quad \texttt{GDPSender} \times \texttt{GDPSenderAction} \longrightarrow \texttt{GDPSender}.$$

The possible actions performed by the sender are send a message, process an acknowledgment, and timeout due to the fact that an acknowledgment has not been received in a predefined time. The actions are formally defined by the enumeration type:

$$\texttt{GDPSenderAction} = \{\texttt{Send,}\quad \texttt{GetAck,}\quad \texttt{Timeout}\}.$$

The following describes GDP sender transition to the next state. In each case, the next state is the same as the current state except for the changes described below.

`Send`. If there is data to be sent and space in the sender window, i.e.,

$$\neg\ \texttt{empty\_fifo?}(\texttt{App\_to\_GDP})\ \wedge\ ns - na < sw,$$

then

- If there is no space in the `ackd` sender window, then the next state is the same as the current state except that `nop` is set to true.
- The data is removed from the `App_to_GDP` queue.
- A GDP frame is formed from the value removed from `App_to_GDP` and the sequence number ns and added to the `GDP_to_LL` queue to send it to the link layer for further processing.
- ns is incremented by 1.
- `ackd` is updated, i.e., its length is incremented by 1.
- `data`(n) is assigned the value of the data removed from the queue and `mask`(n) is assigned the value false, where n is the current length of the buffer.

`GetAck`. An acknowledgment message contains two fields, lb and ub, that denote the lower bound and upper bound on the sequence numbers being acknowledged. If the message being acknowledged falls outside of the window

$$lb < na \vee ub \geq ns,$$

then ignore the message removing it from the `LL_to_GDP` queue.

If

$$na \leq lb \wedge ub < ns,$$

then the next state is the same as the current state with the acknowledgment message removed from the `LL_to_GDP`. The `ackd` mask entries $lb - na, \ldots, ub - na$ are set to true. The function `slide` is then invoked to change `ackd` as described above.

Otherwise, `nop` is set to true.

`Timeout`. If a timeout has occurred, because an acknowledgement has not been received within the predetermined limit, and $ns > na$, then retransmit `data(0)`, from `ackd`, which corresponds to the message with the sequence number na. Otherwise, the next state is the same as the current state except that `nop` is set to true.

3.3.4 Receiver

The receiver process implements the procedure for the sliding window protocol receiver outlined above. The window `rcvd` is also an instance of `Window`. The state of the sender is defined as

$$\begin{aligned}\texttt{GDPReceiver} \;=\; & \texttt{GDP\_to\_App} : \texttt{fifo[Data]} \times \\ & \texttt{winreceiver} : \texttt{WinReceiverPrivate} \times \\ & \texttt{link} : \texttt{LinkInterface} \times \\ & \texttt{ether} : \texttt{EtherInterface} \times \\ & \texttt{nop} : \texttt{Boolean}.\end{aligned}$$

This forms a tuple of the queue that will hold the data once it has been received, the local pointers and bounded buffers of the receiver side of the sliding-window protocol, an interface to the link layer holding the queues `GDP_to_LL` and `LL_to_GDP`, an interface to the ether, and a Boolean that is set to true if no operations are performed.

The receiver process is modeled as a state transition function that takes the current state of the GDP receiver and an action to be performed, and returns the next state:

$$\texttt{GDPReceiverNext:}\quad \texttt{GDPReceiver} \times \texttt{GDPReceiverAction} \longrightarrow \texttt{GDPReceiver}.$$

The possible actions performed by the receiver are to receive a message or to send an acknowledgment. The actions are formally defined by the type enumeration:

$$\texttt{GDPReceiverAction} = \{\texttt{Receive, Sendack}\}.$$

The following describes GDP receiver transition to the next state. In each case, the next state is the same as the current state except for the changes described below.

`Receive`. If the message on the top of the `LL_to_GDP` queue is not a data message, then `nop` is set true. If a data message is on the top of the `LL_to_GDP` queue, then set a local variable *idx* to the value of this message's sequence number. Depending on the value of the sequence number, the protocol takes the following action:

- If $idx \geq nd + rw \;\vee\; la \leq idx \;\wedge\; idx < nd$, then the message is removed from the `LL_to_GDP`.
- If $idx < la$, which means that the message has already been acknowledged, but for whatever reason the sender has resent it, then send an acknowledgment back. That is, the message removed from the `LL_to_GDP` queue and the acknowledgment added to the `GDP_to_LL` queue.
- If $nd \leq idx < nd + rw$, then the sequence number is within the window and so the data is placed in the `rcvd` buffer at location $idx - nd$ and the mask set to true.
  - The message is removed from the `LL_to_GDP` queue.
  - The message is added to the `GDP_to_App` queue.

– nd is set to the index of the first mask in `rcvd` that is false.
– lr is set to the maximum between lr and $idx + 1$.
– `rcvd` is slid as explained above.

`SendAck`. If $nd > la$, then form an acknowledgment message acknowledging $la, \ldots, nd - 1$. The next state is the same as the current state except that

- la is set to nd
- The new acknowledgment message is added to the `GDP_to_LL` queue.

3.4 Link Layer

The link layer is intended to serve as an interface between the WDP and GDP layers and the physical layer. It provides common services needed by WDP and GDP such as error detection. Although the details are elided in our specification, the assumption is that a function is applied to an outbound message generating a checksum or some other such value. The message is then wrapped in a link layer header containing the error-detection code. The link layer also multiplexes messages sent from the WDP and GDP layers wrapping them in the common header and demultiplexes them on the receiving side removing this header and sending the unwrapped frame to the appropriate protocol for processing. The communication medium is assumed to be unreliable so just because a message was sent by the link layer does not mean that it will arrive. Also note that the communication medium may corrupt a message, hence the need for the checksum field. A message that is corrupted in transit will be dropped.

A link layer frame is composed of a checksum and a disjoint sum of a GDP or a WDP frame:

$$\begin{aligned}\texttt{LinkFrame} : \; &= \texttt{cs} : \texttt{CheckSum} \times \\ &\quad \texttt{frame} : \texttt{GDPFrame} + \texttt{WDPFrame},\end{aligned}$$

where + denotes disjoint sum. The type `CheckSum` is defined as a nonempty uninterpreted type.

`LinkInterface` represents the interface that the link layer provides to the higher layer and is defined as a tuple of FIFO queues holding `LinkFrame` data. The structure is defined as follows:

The state of the link layer is defined as the following triple:

$$\begin{aligned}\texttt{Link} \; &= \texttt{link} : \texttt{LinkInterface} \times \\ &\quad \texttt{ether} : \texttt{EtherInterface} \times \\ &\quad \texttt{nop} : \texttt{Boolean}.\end{aligned}$$

The link layer processing is modeled as a state transition function that given the current state and an action to perform, returns the next state for the link layer

$$\texttt{LinkNext} : \texttt{Link} \times \texttt{LinkAction} \longrightarrow \texttt{Link}.$$

The transition function performs the following actions: send a GDP message, send a WDP message, and receive a message. Formally, the actions are defined by the type enumeration:

$$\texttt{LinkAction} = \{\texttt{SendWDP},\ \texttt{SendGDP},\ \texttt{Receive}\}.$$

Note that the

`SendWDP`. Broadcast the frame located on the top of the `WDP_to_LL` queue to the receiver, i.e.,

- Remove the WDP frame, say `frame`, from the `WDP_to_LL` queue.
- Create a link frame as the product of `frame` and the frame's checksum.
- Place the linkframe in the ether.

`SendGDP`. Broadcast the fame located on the top of the `GDP_to_LL` queue to the receiver, i.e.,

- Remove the GDP frame, say `frame`, from the `GDP_to_LL` queue.
- Create a link frame as the product of `frame` and the frame's checksum.
- Place the linkframe in the ether.

`Receive`. Let `linkframe` be the link layer frame removed from the ether,

- If the checksum on the `linkframe` is not valid, then the next state is the same as the current state except that `nop` is set to true.
- If the `linkframe` is a GDP message, then add the GDP frame to `LL_to_GDP` queue and remove `linkframe` from the ether.
- If the `linkframe` is a WDP message, then add the WDP frame to `LL_to_WDP` queue and remove `linkframe` from the ether.

3.5 Ether

The ether is an unreliable communication medium where messages can sometimes be duplicated, dropped, or corrupted by noise. Our model reflects the unreliable nature of the medium.

The `EtherInterface` is specified as follows:

$$\begin{aligned}\texttt{EtherInterface} &= \texttt{input} : \texttt{bag[LinkFrame]} \times \\ &\ \texttt{output} : \texttt{bag[LinkFrame]}.\end{aligned}$$

The link layer sends messages by placing link frames into its ether input channel and receives frames on its ether output channel. In practice, a node acting as a sender places information on its input channel so that same channel must be the output channel at the receiver.

The state of the ether layer is defined as the following triple:

$$\begin{aligned}\texttt{Ether} &= \texttt{ether} : \texttt{EtherInterface} \times \\ &\ \texttt{nop} : \texttt{Boolean}.\end{aligned}$$

Ether processing is structured as a state transition function that given the current state and an action to be performed, returns the next state:

$$\texttt{EtherNext} : \texttt{Ether} \times \texttt{EtherAction} \longrightarrow \texttt{Ether}.$$

The transition function may drop, duplicate, or corrupt messages. The actions are formally defined by the following enumeration type:

$$\texttt{EtherAction} \;=\; \{\, \texttt{DropIn},\ \texttt{DropOut},\ \texttt{DupIn},\ \texttt{DupOut},\ \texttt{NoiseIn},\ \texttt{NoiseOut} \,\}.$$

`DropIn`. Drop a specified frame in the input channel by removing that frame from the input multiset.

`DropOut`. Drop a specified frame in the output channel by removing that frame from the output multiset.

`DupIn`. Duplicate a specified frame in the input channel by adding an additional copy of the frame to the input multiset.

`DupOut`. Duplicate a specified frame in the output channel by adding an additional copy of the frame to the output multiset.

`NoiseIn`. A noise corrupted frame is added to the input channel.

`NoiseOut`. A noise corrupted frame is added to the output channel.

4 SPECIFICATION OF SENDER AND RECEIVER PROCESSES

Thus far, we have described each layer of the protocol stack individually. In this section, we show how these layers are composed to form a protocol stack. First, we will look at the WDP and GDP protocols in isolation and then we shall see how these two protocols can be composed asynchronously.

For each one of the WDP and GDP protocols, we will assume that we have two processes: a sender process and a receiver process. The WDP sender and receiver processes are called `WDPSender?` and `WDPReceiver?`, respectively. Similarly, the GDP sender and receiver processes are called `GDPSender?` and `GDPReceiver?`, respectively. These processes behave in a non-deterministic way. Hence, each one of them is defined as relation between the current state and one of the possible next states.

For instance, `GDPSender?`, which relates the current state of the GDP sender process and a possible next state, is defined as either a `GDPSenderNext` transition, a `LinkNext` transition, or a `EtherNext` transition, where the fields that are not modified by the transitions remain unchanged. As explained in Section 3, each one of these transitions depends upon a particular action. In order to model the non-deterministic nature of actions, we use existential quantifiers to generate actions for each transition. The relation `GDPSender?` is formally expressed as follows:

$$\begin{aligned}\texttt{GDPSender?}(s, n : \texttt{GDPSender}) \;=\; & (\,\exists\, a : \texttt{GDPSenderAction}.\ n = \texttt{GPDSenderNext}(s, a) \\ & \quad \wedge \neg s`\texttt{nop} \;\wedge\; \neg n`\texttt{nop}\end{aligned}$$

$$
\begin{array}{l}
\bigvee \\
\quad (\exists\, a : \texttt{LinkAction}.\ n_l = \texttt{LinkNext}(s_l, a) \\
\quad\ \wedge \neg s_l\text{`}\texttt{nop} \ \wedge\ \neg n_l\text{`}\texttt{nop} \\
\quad\ \wedge\ n\text{`}\texttt{App\_to\_GDP} = s\text{`}\texttt{App\_to\_GDP} \\
\quad\ \wedge\ s\text{`}\texttt{winsender} \ = \ n\text{`}\texttt{winsender}\ ) \\
\bigvee \\
\quad (\exists\, a : \texttt{EtherAction}.\ n_e = \texttt{EtherNext}(s_e, a) \\
\quad\ \wedge \neg s_e\text{`}\texttt{nop} \ \wedge\ \neg n_e\text{`}\texttt{nop} \\
\quad\ \wedge\ n\text{`}\texttt{link} = s\text{`}\texttt{link} \\
\quad\ \wedge\ n\text{`}\texttt{App\_to\_GDP} = s\text{`}\texttt{App\_to\_GDP} \\
\quad\ \wedge\ s\text{`}\texttt{winsender} \ = \ n\text{`}\texttt{winsender}\ ),
\end{array}
$$

where s and n stand for the current and next `GDPSender` state, respectively, and the back-quote symbol is the field access operator. We denote by sub-indices l and e the projections of states s, n into `Link` and `Ether` states, respectively.

The relations `GDPReceiver?`, `WDPSender?`, and `WDPReceiver?` are defined in a similar way.

$$
\begin{array}{lcl}
\texttt{GDPReceiver?}(s, n : \texttt{GDPReceiver}) & = & (\exists\, a : \texttt{GDPReceiverAction}.\ n = \texttt{GPDReceiverNext}(s, a) \\
& & \quad \wedge \neg s\text{`}\texttt{nop} \ \wedge\ \neg n\text{`}\texttt{nop} \\
& \bigvee & \\
& & \quad (\exists\, a : \texttt{LinkAction}.\ n_l = \texttt{LinkNext}(s_l, a) \\
& & \quad\ \wedge \neg s_l\text{`}\texttt{nop} \ \wedge\ \neg n_l\text{`}\texttt{nop} \\
& & \quad\ \wedge\ n\text{`}\texttt{GDP\_to\_App} = s\text{`}\texttt{GDP\_to\_App} \\
& & \quad\ \wedge\ s\text{`}\texttt{winreceiver} \ = \ n\text{`}\texttt{winreceiver}\ ) \\
& \bigvee & \\
& & \quad (\exists\, a : \texttt{EtherAction}.\ n_e = \texttt{EtherNext}(s_e, a) \\
& & \quad\ \wedge \neg s_e\text{`}\texttt{nop} \ \wedge\ \neg n_e\text{`}\texttt{nop} \\
& & \quad\ \wedge\ n\text{`}\texttt{link} = s\text{`}\texttt{link} \\
& & \quad\ \wedge\ n\text{`}\texttt{GDP\_to\_App} = s\text{`}\texttt{GDP\_to\_App} \\
& & \quad\ \wedge\ s\text{`}\texttt{winreceiver} \ = \ n\text{`}\texttt{winreceiver}\ ).
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{WDPSender?}(s, n : \texttt{WDPSender}) & = & (n = \texttt{WDPSenderNext}(s) \\
& & \quad \wedge \neg s\text{`}\texttt{nop} \ \wedge\ \neg n\text{`}\texttt{nop} \\
& \bigvee & \\
& & \quad (\exists\, a : \texttt{LinkAction}.\ n_l = \texttt{LinkNext}(s_l, a) \\
& & \quad\ \wedge \neg s_l\text{`}\texttt{nop} \ \wedge\ \neg n_l\text{`}\texttt{nop}
\end{array}
$$

$$\begin{array}{l} \quad\quad \wedge\ n`\texttt{App\_to\_WDP} = s`\texttt{App\_to\_WDP}\ ) \\ \bigvee \\ \quad (\ \exists\ a : \texttt{EtherAction}.\ n_e = \texttt{EtherNext}(s_e, a) \\ \quad\quad \wedge\ \neg s_e`\texttt{nop}\ \wedge\ \neg n_e`\texttt{nop} \\ \quad\quad \wedge\ n`\texttt{link} = s`\texttt{link} \\ \quad\quad \wedge\ n`\texttt{App\_to\_WDP} = s`\texttt{App\_to\_WDP}\ ). \end{array}$$

$$\begin{array}{rcl} \texttt{WDPReceiver?}(s, n : \texttt{WDPReceiver}) & = & (\ n = \texttt{WDPReceiverNext}(s) \\ & & \quad \wedge\ \neg s`\texttt{nop}\ \wedge\ \neg n`\texttt{nop} \\ & & \bigvee \\ & & \quad (\ \exists\ a : \texttt{LinkAction}.\ n_l = \texttt{LinkNext}(s_l, a) \\ & & \quad\quad \wedge\ \neg s_l`\texttt{nop}\ \wedge\ \neg n_l`\texttt{nop} \\ & & \quad\quad \wedge\ n`\texttt{WDP\_to\_App} = s`\texttt{WDP\_to\_App}\ ) \\ & & \bigvee \\ & & \quad (\ \exists\ a : \texttt{EtherAction}.\ n_e = \texttt{EtherNext}(s_e, a) \\ & & \quad\quad \wedge\ \neg s_e`\texttt{nop}\ \wedge\ \neg n_e`\texttt{nop} \\ & & \quad\quad \wedge\ n`\texttt{link} = s`\texttt{link} \\ & & \quad\quad \wedge\ n`\texttt{WDP\_to\_App} = s`\texttt{WDP\_to\_App}\ ). \end{array}$$

We define a sender process, which conforms to the protocol stack, as the asynchronous composition of the WDP and GDP sender processes. Formally, the state of the sender process is the union of the fields in `WDPSender` and `GDPSender`:

$$\begin{array}{rcl} \texttt{Sender} & = & \texttt{App\_to\_GDP} : \texttt{fifo[Data]} \times \\ & & \texttt{App\_to\_WDP} : \texttt{fifo[Data]} \times \\ & & \texttt{winsender} : \texttt{WinSender} \times \\ & & \texttt{link} : \texttt{LinkInterface} \times \\ & & \texttt{ether} : \texttt{EtherInterface}. \end{array}$$

The relation between the current state of the sender and a possible next state is as either a `WDPSender?` transition or a `GDPSender?` transition. As in the previous relations, fields that are not modified by the transitions remain the same:

$$\begin{array}{rcl} \texttt{Sender?}(s, n : \texttt{Sender}) & = & \texttt{WDPSender?}(s_{ws}, n_{ws}) \\ & & \wedge\ s`\texttt{App\_to\_GDP} = n`\texttt{App\_to\_WGDP} \\ & & \wedge\ s`\texttt{windsender} = n`\texttt{winsender} \\ & \bigvee & \\ & & \texttt{GDPSender?}(s_{gs}, n_{gs}) \\ & & \wedge\ s`\texttt{App\_to\_WDP} = n`\texttt{App\_to\_WDP}. \end{array}$$

We denote by sub-indices ws and gs the projections of states s, n into `WDPSender` and `GDPSender` states, respectively.

In a similar way, the receiver process is defined as the asynchronous composition between the WDP and GDP receiver processes. Formally,

$$\begin{aligned}\texttt{Receiver} \;=\; & \texttt{GDP\_to\_App} : \texttt{fifo[Data]} \times \\ & \texttt{WDP\_to\_App} : \texttt{fifo[Data]} \times \\ & \texttt{winreceiver} : \texttt{WinReceiver} \times \\ & \texttt{link} : \texttt{LinkInterface} \times \\ & \texttt{ether} : \texttt{EtherInterface}.\end{aligned}$$

$$\begin{aligned}\texttt{Receiver?}(s, n) \;=\; & \texttt{WDPReceiver?}(s_{wr}, n_{wr}) \\ & \wedge\; s`\texttt{GDP\_to\_App} = n`\texttt{GDP\_to\_App} \\ & \wedge\; s`\texttt{windreceiver} = n`\texttt{winreceiver} \\ \bigvee \; & \\ & \texttt{GDPReceiver?}(s_{gr}, n_{gr}) \\ & \wedge\; s`\texttt{WDP\_to\_app} = n`\texttt{WDP\_to\_app},\end{aligned}$$

where sub-indices wr and gr denote the projections of states s, n into `WDPReceiver` and `GDPReceiver` states, respectively.

5 PROTOCOL VERIFICATION

In this work, we focus on the functional correctness of a distributed system that consists of a sender node and receiver node communicating via our protocol stack through the ether. Each node has both the GDP and WDP processes, but for the sake of analysis we consider a sender and receiver and assume that they are located at different nodes. We denote that system by `WDP` $\|$ `GDP`. The system configuration is illustrated by Figure 3.

5.1 Distributed System

We formally specify the distributed system `WDP` $\|$ `GDP` as a state transition system using a set of PVS theories developed by Rusu [21]. In those theories a state transition system is defined by an initial set of states and a state transition relation. The set of reachable states is the set of states in the reflexive-transitive closure of the transition relation starting from the set of initial states. The functional correctness of a system specified this way is expressed by *invariant properties*, i.e., predicates that hold in every reachable state of the system.

Formally, the state of `WDP` $\|$ `GDP`, namely `WDP_GDP`, is composed of the union of fields in `Sender` and `Receiver`, where the ether interface is shared between the two processes:

$$\begin{aligned}\texttt{WDP\_GDP} \;=\; & \texttt{App\_to\_GDP} : \texttt{fifo[Data]} \times \\ & \texttt{App\_to\_WDP} : \texttt{fifo[Data]} \times \\ & \texttt{GDP\_to\_App} : \texttt{fifo[Data]} \times \\ & \texttt{WDP\_to\_App} : \texttt{fifo[Data]} \times\end{aligned}$$

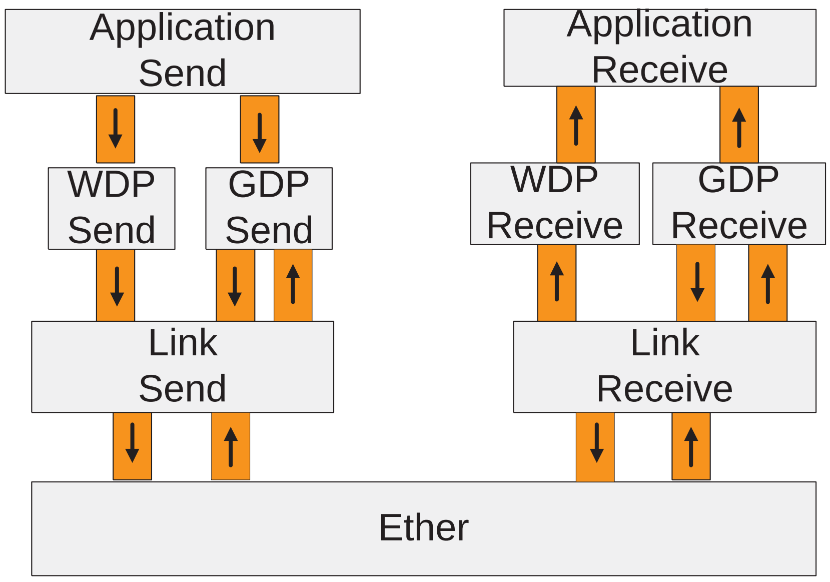

Figure 3: *Sender/Receiver processes*

$$\begin{aligned}&\texttt{winsender} : \texttt{WinSender} \times\\ &\texttt{winreceiver} : \texttt{WinReceiver} \times\\ &\texttt{linksender} : \texttt{LinkInterface} \times\\ &\texttt{linkreceiver} : \texttt{LinkInterface} \times\\ &\texttt{ether} : \texttt{EtherInterface}.\end{aligned}$$

At the initial states, the queue `App_to_GDP` contains the data to be sent using GDP and the queue `App_to_WDP` contains the data to be sent using WDP. All the other queues and the ether channels are empty.

The transition relation of `WDP` || `GDP` is defined as either a `Sender?` transition or a `Receiver?` transition, where the fields that are no modified by the transitions remain the same:

$$\begin{aligned}\texttt{WDP\_GDP?}(s, n : \texttt{WDP\_GDP}) \quad = \quad & \texttt{Sender?}(s_s, n_s)\\ & \wedge\ s`\texttt{GDP\_to\_App} = n`\texttt{GDP\_to\_App}\\ & \wedge\ s`\texttt{WDP\_to\_App} = n`\texttt{WDP\_to\_App}\\ & \wedge\ s`\texttt{winreceiver} = n`\texttt{winreceiver}\\ & \wedge\ s`\texttt{linkreceiver} = n`\texttt{linkreceiver}\\ \bigvee \quad & \\ & \texttt{Receiver?}(s_r, n_r)\\ & \wedge\ s`\texttt{App\_to\_GDP} = n`\texttt{App\_to\_GDP}\\ & \wedge\ s`\texttt{App\_to\_WDP} = n`\texttt{App\_to\_WDP}\\ & \wedge\ s`\texttt{winsender} = n`\texttt{winsender}\\ & \wedge\ s`\texttt{linksender} = n`\texttt{linksender}.\end{aligned}$$

The sub-indices s and r denote the projections of states s, n into `Sender?` and `Receiver?` states, respectively. In the case of s_r and n_r the input and output channels of the ether are exchanged, i.e.,

$$\begin{aligned} s_r\text{`}\texttt{ether`input} &= s_s\text{`}\texttt{ether`output}, \\ s_r\text{`}\texttt{ether`output} &= s_s\text{`}\texttt{ether`input}, \\ n_r\text{`}\texttt{ether`input} &= n_s\text{`}\texttt{ether`output}, \\ n_r\text{`}\texttt{ether`output} &= n_s\text{`}\texttt{ether`input}. \end{aligned}$$

The invariant predicate that expresses the correctness property of WDP is defined as follows:

$$\texttt{WDP\_sound}(s, s_0 : \texttt{WDP\_GDP}) = s\text{`}\texttt{WDP\_to\_App} \subseteq s_0\text{`}\texttt{App\_to\_WDP},$$

where s refers to a reachable state of the distributed system and s_0 refers to an initial state. This invariant states that WDP data that the receiver process delivers to the application layer were indeed sent by the sender's application layer.

The invariant predicate that expresses the correctness property of GDP is defined as follows:

$$\texttt{GDP\_sound}(s, s_0 : \texttt{WDP\_GDP}) = s\text{`}\texttt{GDP\_to\_App} \preceq s_0\text{`}\texttt{App\_to\_GDP},$$

where $\preceq$ is the prefix relation between sequences. This invariant states that GDP data are delivered by the receiver to the application layer in the same order as they were sent by the sender's application layer.

Our verification objective is to formally prove that these two predicates, `WDP_sound` and `GDP_sound`, are indeed invariants of the distributed system `WDP` $\|$ `GDP`. A direct proof of these properties can be accomplished by strengthening the invariant to a form that can be proved by induction on the length of reachable traces, for example using the PVS theories provided in [21]. However such a proof is extremely cumbersome since it requires a case analysis of more than 600 cases, which correspond to all possible interleavings between sender and receiver processes of WDP and GDP.

Instead of a direct proof, we propose a compositional approach where each invariant is independently proved for its respective system, i.e., `WDP_sound` is an invariant of WDP and `GDP_sound` is an invariant of GDP, and then we provide a general framework that enables to lift these invariants to the distributed system `WDP` $\|$ `GDP`.

5.2 Proving Invariants on `WDP` and `GDP`

We first consider the case of a system of a sender and receiver WDP processes. The state of this system, called `WDP`, is a tuple composed of the union of the fields in `WDPSender` and `WDPReceiver` where the ether interface is shared between the two processes.

$$\begin{aligned} \texttt{WDP} = \; & \texttt{App\_to\_WDP} : \texttt{fifo[Data]} \times \\ & \texttt{WDP\_to\_App} : \texttt{fifo[Data]} \times \\ & \texttt{linksender} : \texttt{LinkInterface} \times \\ & \texttt{linkreceiver} : \texttt{LinkInterface} \times \\ & \texttt{ether} : \texttt{EtherInterface}. \end{aligned}$$

The WDP? transition relation is defined as follows:

$$\begin{array}{rcl}\texttt{WDP?}(s, n : \texttt{WDP}) & = & \texttt{WDPSender?}(s_s, n_s) \\ & & \wedge\ s`\texttt{WDP\_to\_App} = n`\texttt{WDP\_to\_App} \\ & & \wedge\ s`\texttt{linkreceiver} = n`\texttt{linkreceiver} \\ & \vee & \\ & & \texttt{WDPReceiver?}(s_r, n_r) \\ & & \wedge\ s`\texttt{App\_to\_WDP} = n`\texttt{App\_to\_WDP} \\ & & \wedge\ s`\texttt{linksender} = n`\texttt{linksender}.\end{array}$$

As in the case of WDP ‖ GDP, the projections from a WDP state into a WDPSender state and into a WDPReceiver state are defined such that the input and output channels of the ether interface in the sender process are connected, respectively, to the output and input channels of the ether interface in the receiver process:

$$\begin{array}{rcl}s_r`\texttt{ether}`\texttt{input} & = & s_s`\texttt{ether}`\texttt{output}, \\ s_r`\texttt{ether}`\texttt{output} & = & s_s`\texttt{ether}`\texttt{input}, \\ n_r`\texttt{ether}`\texttt{input} & = & n_s`\texttt{ether}`\texttt{output}, \\ n_r`\texttt{ether}`\texttt{output} & = & n_s`\texttt{ether}`\texttt{input}.\end{array}$$

The system GDP is defined in a similar way:

$$\begin{array}{rcl}\texttt{GDP} & = & \texttt{App\_to\_GDP} : \texttt{fifo[Data]} \times \\ & & \texttt{GDP\_to\_App} : \texttt{fifo[Data]} \times \\ & & \texttt{winsender} : \texttt{WinSender} \times \\ & & \texttt{linksender} : \texttt{LinkInterface} \times \\ & & \texttt{winreceiver} : \texttt{WinReceiver} \times \\ & & \texttt{linkreceiver} : \texttt{LinkInterface} \times \\ & & \texttt{ether} : \texttt{EtherInterface}.\end{array}$$

The GDP? transition relation is defined as follows:

$$\begin{array}{rcl}\texttt{GDP?}(s, n : \texttt{GDP}) & = & \texttt{GDPSender?}(s_s, n_s) \\ & & \wedge\ s`\texttt{GDP\_to\_App} = n`\texttt{GDP\_to\_App} \\ & & \wedge\ s`\texttt{winreceiver} = n`\texttt{winreceiver} \\ & & \wedge\ s`\texttt{linkreceiver} = n`\texttt{linkreceiver} \\ & \vee & \\ & & \texttt{GDPReceiver?}(s_r, n_r) \\ & & \wedge\ s`\texttt{App\_to\_GDP} = n`\texttt{App\_to\_GDP} \\ & & \wedge\ s`\texttt{winsender} = n`\texttt{winsender} \\ & & \wedge\ s`\texttt{linksender} = n`\texttt{linksender}.\end{array}$$

Proving invariants on transition systems, such as `WDP` and `GDP`, are routine in the theorem proving community. It usually entails the transformation of the initial invariant to an inductive form, i.e., a stronger invariant that can be proved by induction.

The main difficulty remains the finding of auxiliary invariants that enable the inductive proof of the original invariant. For `WDP` and `GDP` the problem is further complicated by the fact that we have to consider the full protocol stack and all possible interleavings between the sender and receiver processes. The number of interleavings for each system is significantly lower than for the composed system `WDP` || `GDP`, 21 cases for `WDP` and 31 cases for `GDP`, but it is still a tedious task. For each one of these cases we have to prove that if an invariant is satisfied at step n, it is also satisfied at step $n + 1$. This is a considerable amount of work even though most of the cases can be easily discharged by using general properties of bags, queues, and bounded buffers.

To automate the verification task, we have defined a set of proof strategies that unfold the transition relations `WDP?` and `GDP?` and discharge the easy cases of inductive proofs. Even in the cases where the strategies do not succeed, they generate enough information to assist a developer in finding weaker invariants. The strategies can be applied to discrete transition systems defined in PVS using Rusu's theories, but are particularly suitable for protocols that use data structures such as bags, FIFO queues, and bounded buffers.

Theorem 1. *`WDP_sound` is an invariant on WDP.*

Proof Sketch. The proof that `WDP_sound` is an invariant of `WDP` only requires two proof commands. The first command is our strategy `discharge-inv`, which automatically proves 20 of the 21 inductive cases. The unproven case suggests an auxiliary invariant that states that all WDP frames in the link layer and in the ether belongs to s_0‘`App_to_WDP`. Once this invariant is added as a lemma to the theory, the proof is finished by using our strategy `use-inv`. To prove the auxiliary invariant, we follow the same approach, which suggests the new invariant:

$$s\text{‘}\mathtt{WDP\_to\_App} \subseteq s_0\text{‘}\mathtt{WDP\_to\_App}.$$

This new invariant is automatically discharged by `discharge-inv`. □

Theorem 2. *`GDP_sound` is an invariant on GDP.*

Proof Sketch. The soundness proof of `GDP` is considerably more complicated than the soundness proof of `WDP`, but the general method is the same. We use our strategy `discharge-inv` to eliminate the easy cases and we add new invariants to discharge the unproven cases via `use-inv`. We iterate this approach on the new invariants. In total we have added 6 auxiliary invariants as lemmas, including the following relations between the sender's and receiver's windows:

- The counter of received messages is less than or equal to the counter of sent messages:

$$s\text{‘}\mathtt{winreceiver}\text{‘}\mathtt{lr} \leq s\text{‘}\mathtt{winsender}\text{‘}\mathtt{ns}$$

- The counter of delivered messages is less than or equal to the counter of sent messages:

$$s\text{‘}\mathtt{winreceiver}\text{‘}\mathtt{nd} \leq s\text{‘}\mathtt{winsender}\text{‘}\mathtt{ns}$$

- The largest sequence number for which an acknowledgment has been received is less than or equal to the counter of the sent acknowledgments:

$$s`\texttt{winsender}`\texttt{na} + \texttt{last\_true}(s`\texttt{winsender}`\texttt{ackd}) \leq s`\texttt{winsender}`\texttt{la},$$

  where the function `last_true` returns the difference between s‘`winsender`‘`na` and the largest sequence number for which an acknowledgment has been received.

□

As a point aside, we note that the invariants we have discussed in this section involve relationships between the sender and the receiver processes. There are many relationships that are local to either the sender or the receiver. For instance, the property that states that the index of the next message to be sent is greater than or equal to the index of the next message waiting to be acknowledged, i.e., $s`\texttt{winsender}`\texttt{na} \leq s`\texttt{winsender}`\texttt{ns}$, only concerns the sender process, and the property that states that the index of the next message to be delivered is greater than or equal to the index of the last message to be acknowledged, i.e., $s`\texttt{winreceiver}`\texttt{la} \leq s`\texttt{winreceiver}`\texttt{nd}$, only concerns the receiver process. As these properties can be described solely in terms of either `GDPSender` or `GDPReceiver`, they can be easily encoded using the PVS's dependent type system and automatically discharged by the type checker.

5.3 Proving Liveness for `GDP`

The soundness property for GDP that was proved above corresponds to a safety property. We now consider a liveness property for GDP. While a soundness property states something bad does not happen, liveness states that something good will eventually happen. In the case of a sliding window protocol, liveness means that messages sent from the sender will eventually arrive at the receiver. Yet there are some complications to resolve. If some data is never sent, then it would be difficult to see what liveness property for GDP could be formulated under such conditions. Consequently, liveness properties are formulated under the assumption that the system behaves in a ‘fair’ manner. For GDP, the fairness property says that all messages in the `App_to_GDP` queue are eventually sent. That is, for every message in `App_to_GDP`, it is eventually the case that a state is recorded where each message has been sent. Since it is an invariant that `ns` always points to the next item to be sent in r_0‘`App_to_GDP`, we can state the fairness property as saying that given any run of the protocol, for every sequence number $m < r_0`\texttt{App\_to\_GDP}`\texttt{length},$ the run records a state where $\texttt{ns} > m$. This is stated formally as follows:

$$\begin{aligned}\text{fair}[(\text{run})] \quad = \quad & \lambda(r : (run)) : \forall(m : \text{below}(r_0`\texttt{App\_to\_GDP}`\texttt{length}) : \\ & \exists(n : \text{Nat}) : r_n`\texttt{winsender}`\texttt{ns} > m.\end{aligned}$$

In principle, the liveness property should state that all messages that are sent are eventually received. Yet this is not possible to prove given our assumption that `App_to_GDP` is of finite length. To see why, recall that the sender maintains a sliding window with size

sw and that the protocol ensures that all data with sequence numbers below `na` have been delivered. The invariant $\texttt{na} \leq \texttt{ns} \leq \texttt{na} + \texttt{sw}$ relates the values of `na` and `ns`. If the fairness condition is satisfied, that is all of the data in the `App_to_GDP` queue has been sent and `ns` is equal to the length of the queue and the value of `na` is undetermined (except for the relation governed by the invariant), then based on the knowledge of the sender, we can only prove that the first $0, \ldots, r_0$`App_to_GDP`‘`length` $-$ `sw` items eventually arrive at the receiver. Consequently, our liveness property says that all data with a sequence number less than r_0‘`App_to_GDP`‘`length` $-$ `sw` eventually arrive at the receiver. Formally, this is expressed as a predicate on the runs of the protocol as follows:

$$\begin{aligned}\text{live}[(\text{run})] &= \lambda(r : (run)) : \forall(m : \text{below}(r_0\text{‘}\texttt{App\_to\_GDP‘length - sw}) : \\ &\qquad \exists(n : \text{Nat}) : r_n\texttt{‘winreceiver‘nd} > m.\end{aligned}$$

Note that it is an invariant that

$$\texttt{nd} = r_n\text{‘}\texttt{GDP\_to\_App‘length}$$

so the predicate expresses the desired property that for a given run it is eventually the case that all the messages with sequence numbers less than r_0‘`App_to_GDP`‘`length - sw` are placed in the `GDP_to_App` queue.

Several lemmas aide in the task of showing liveness follows from fairness. The predicate fair_aux1 says that for all data with a sequence number less than r_0‘`App_to_GDP`‘`length` $-$ `sw` it is eventually the case that the sender knows that these sequence numbers have been delivered.

$$\begin{aligned}\text{fair\_aux1}[(\text{run})] &= \lambda(r : (run)) : \forall(m : \text{below}(r_0\text{‘}\texttt{App\_to\_GDP‘length - sw}) : \\ &\qquad \exists(n : \text{Nat}) : r_n\texttt{‘winsender‘na} > m.\end{aligned}$$

The following lemma says that fair_aux1 follows from fair:

Lemma 1. $\forall(r : (run)) : \textit{fair} \Rightarrow \textit{fair\_aux1}.$

Proof Sketch. The crux of the proof is to skolemize the variable m in the consequent, which has a type $0 \leq m < r_0$‘`App_to_GDP`‘`length - sw`, and then instantiate m in the antecedent, which has a range $0 \leq m < r_0$‘`App_to_GDP`‘`length`, to $m + \texttt{sw}$. Applying the invariant $\texttt{na} \leq \texttt{ns} \leq \texttt{na + sw}$ yields $m + \texttt{sw} < \texttt{ns} \leq \texttt{na + sw}$ from which we can conclude $m < \texttt{na}$. □

Recall that the receiver maintains a pointer `la` indicating that all sequence numbers below `la` have been acknowledged. The predicate fair_aux2 says that for any given run, for all sequence numbers less than r_0‘`App_to_GDP`‘`length - sw` there is a state where these sequence numbers have been acknowledged.

$$\begin{aligned}\text{fair\_aux2}[(\text{run})] &= \lambda(r : (run)) : \forall(m : \text{below}(r_0\text{‘}\texttt{App\_to\_GDP‘length - sw}) : \\ &\qquad \exists(n : \text{Nat}) : r_n\texttt{‘winreceiver‘la} > m.\end{aligned}$$

The next lemma relates the sender and receiver saying the fair_aux2 follows from fair_aux1.

Lemma 2. $\forall (r : (run)) : \mathit{fair\_aux1} \Rightarrow \mathit{fair\_aux2}$

Proof Sketch. The lemma follows from the antecedent and the invariant that `na` $\leq$ `la` $\square$

The liveness theorem states that liveness follows from fairness

Theorem 3. $\forall (r : (run)) : \mathit{fair} \Rightarrow \mathit{live}$

Proof Sketch. The result follows from the above lemma. $\square$

5.4 Proving Invariants on `WDP` $\|$ `GDP`

We have seen that `WDP_sound` is an invariant of `WDP` and that `GDP_sound` is an invariant of `GDP`. However, our verification objective is to show that both of them are also invariants of `WDP` $\|$ `GDP`. This goal could be trivially achieved if `WDP` and `GDP` were completely independent. They are not. As illustrated by Figure 3, `WDP` and `GDP` share the link layer and ether interfaces.

In order to prove that a predicate is an invariant of a composed system, such as `WDP` $\|$ `GDP`, we develop a general theory of asynchronous composition of transition systems where invariants on one system can be lifted to the composed system. To this end, we consider that the state of a transition system consists of a private state and a shared state. The state of the composed system has a copy of the private states of each transition system but only one shared state common to both of them. When the composed system performs a transition in one of the constituent parts of the system, the private state of the other system remains unchanged.

We define an *abstraction* α of a transition system T as a function that maps states into states such that

1. if s_0 is an initial state in T, then $\alpha(s_0)$ is also an initial state of T, and

2. if (s_n, s_{n+1}) is a transition in T, then $(\alpha(s_n), \alpha(s_{n+1}))$ is also a transition in T.

Given this definition, we formally prove the following theorem:

Theorem 4 (Invariant Left-Lifting)**.** *Let P be an invariant of a transition system T_1. The predicate P is an invariant of the transition system $T_1 \| T_2$ if there is an* abstraction *α of T_1 such that the following conditions are met:*

1. *P is orthogonal to α, i.e., $P(\alpha(s))$ implies $P(s)$, and*

2. *under the abstraction α, T_2 does not interfere with T_1, i.e., if (s_n, s_{n+1}) is a transition in T_2, then $(\alpha(s_n), \alpha(s_{n+1})$ is a transition in T_1.*

Proof Sketch. Consider an arbitrary trace $s_0, \ldots, s_n$ in $T_1 \| T_2$. We will show that P holds in s_n. First, we show that $\alpha(s_0), \ldots, \alpha(s_n)$ is a trace in T_1. There are two cases:

1. The transition (s_i, s_{i+1}) is a T_1 transition. In this case, $(\alpha(s_i) \rightarrow \alpha(s_{i+1}))$ is also a T_1 transition since α is an abstraction.

2. The transition (s_i, s_{i+1}) is a T_2 transition. In this case, $(\alpha(s_i), \alpha(s_{i+1}))$ is also a T_1 transition since T_2 does not interfere with T_1.

Therefore, $\alpha(s_0), \ldots, \alpha(s_n)$ is a trace in T_1. Since P is an invariant on T_1, P holds in $\alpha(s_n)$. Since P is orthogonal to α, P holds in s_n as well. □

A symmetric theorem for the right transition system can be proved in a similar way.

Theorem 5 (Invariant Right-Lifting). *Let P be an invariant of a transition system T_2. The predicate P is an invariant of the transition system $T_1 \parallel T_2$ if there is an* abstraction *α of T_2 such that P is orthogonal to α, and under the abstraction α, T_1 does not interfere with T_2.*

For the case of the distributed system `WDP` || `GDP`, the queues `App_to_WDP` and `WDP_to_App` are private to `WDP`. The queues `App_to_GDP` and `GDP_to_App`, and the fields `winsender` and `winreceiver` are private to `GDP`. All the other fields. i.e., the link and the ether interfaces, are shared.

Theorem 6 (`WDP` Soundness). *`WDP_sound` is an invariant on `WDP` || `GDP`.*

Proof Sketch. We consider an abstraction $\alpha_w(s : \texttt{WDP})$ such that $\alpha_w(s) = s$ in all fields but:

$$\begin{aligned}
\alpha_w(s\text{`}\texttt{link}\text{`}\texttt{GDP\_to\_Link}) &= \texttt{empty},\\
\alpha_w(s\text{`}\texttt{link}\text{`}\texttt{Link\_to\_GDP}) &= \texttt{empty},\\
\alpha_w(s\text{`}\texttt{ether}\text{`}\texttt{input}) &= \texttt{remove\_gdp}(s\text{`}\texttt{ether}\text{`}\texttt{input}),\\
\alpha_w(s\text{`}\texttt{ether}\text{`}\texttt{output}) &= \texttt{remove\_gdp}(s\text{`}\texttt{ether}\text{`}\texttt{output}),
\end{aligned}$$

where `empty` is the empty queue and `remove_gdp` removes all GDP frames from a multiset. Then, we prove that α_w is indeed an abstraction of `WDP`, that `WDP_sound` is orthogonal to α_w, and that, under α_w, `GDP` does not interfere with `WDP`. Therefore, by theorems 1 and 4, `WDP_sound` is an invariant on `WDP` || `GDP`. □

Theorem 7 (`GDP` Soundness). *`GDP_sound` is an invariant on `WDP` || `GDP`.*

Proof Sketch. We consider an abstraction $\alpha_g(s : \texttt{GDP})$ such that $\alpha_g(s) = s$ in all fields but:

$$\begin{aligned}
\alpha_g(s\text{`}\texttt{link}\text{`}\texttt{WDP\_to\_Link}) &= \texttt{empty},\\
\alpha_g(s\text{`}\texttt{link}\text{`}\texttt{Link\_to\_WDP}) &= \texttt{empty},\\
\alpha_g(s\text{`}\texttt{ether}\text{`}\texttt{input}) &= \texttt{remove\_wdp}(s\text{`}\texttt{ether}\text{`}\texttt{input}),\\
\alpha_g(s\text{`}\texttt{ether}\text{`}\texttt{output}) &= \texttt{remove\_wdp}(s\text{`}\texttt{ether}\text{`}\texttt{output}),
\end{aligned}$$

where `remove_wdp` removes all WDP frames from a multiset. Then, we prove that α_g is indeed an abstraction of `GDP`, that `GDP_sound` is orthogonal to α_g, and that, under α_g, `WDP` does not interfere with `GDP`. Therefore, by theorems 2 and 5, `GDP_sound` is an invariant on `WDP` || `GDP`. □

As in the case of invariants, we have developed proof strategies to prove that a given function is an abstraction, and that the orthogonality and noninterference conditions are satisfied.

6 RELATED WORK

Numerous variations of the basic sliding window protocol have been subjected to formal verification. Stenning [23] is likely to have been the first to discuss the correctness of such protocols. Virtually every study has focused on proving safety properties just as we have. Several case studies have been produced deriving sliding window protocols using the methodology originated by Dijkstra's school. Van de Snepscheut [8] transforms a sequential program while preserving its correctness and Hoogerwoord [12] applies a methodology for deriving multiprograms to derive a sliding window protocol. In both cases, the protocol assumes unbounded window size. Process algebras have also been used to manually verify one-bit sliding window protocols [25, 3].

Model checking has been applied to a number of sliding window protocols. Holzmann [10, 11] verified both safety and liveness properties for a protocol with a window size of five and [14] did the same for a protocol with a window size of seven. Applying abstraction and model checking, Sthal was able to verify the a protocol with a window size of sixteen [22].

Others have also applied automated theorem provers to verify sliding window protocols. Cardell-Oliver used HOL to verify safety properties [4]. A timed model was given in [7] and a safety property is verified using PVS. Rusu [21] proved safety and liveness of a protocol with unbounded window size in PVS. Safety and liveness properties of a protocol with arbitrary finite window size employing modulo-arithmetic were verified using process algebra techniques with the assistance of the PVS prover in [1].

The sliding window protocols verified in aforementioned efforts were considerably simpler than the sliding window protocol with block acknowledgment response that we have presented here. Only [1] also considers a protocol with arbitrary, but finite window size. Previous work considered the sliding window protocol acting in isolation rather than as a component in a protocol stack, which added considerable complexity to the proofs, but is required to accurately model the communication process.

Concurrently executing programs are complex artifacts making it difficult to reason about their correctness. For parallel programs with shared variables, the classical theory of Owicki and Gries [18] was the first breakthrough for reasoning about the correctness of parallel programs having shared variables, but the theory is not compositional. Assume-Guarantee methods modify the theory to be compositional [16, 13, 15, 26]. Rushby [20] has developed a version of an assume-guarantee rule for use in the verification of timed reactive systems. Charpentier [6, 5] has recently explored the composition of invariants for concurrent systems. In this research the authors explored both invariants satisfied by every component of the composed system as well as situations similar to the one we explored where an invariant in the composed system is satisfied by one component of the composed system. Our approach is not as powerful as assume-guarantee methods, but is largely mechanizable as we have shown here. We believe that our method is suitable for composition of protocol stacks that share the lower communication layers.

7 CONCLUSION

In this paper, we have presented the formal verification of a communication protocol between an airborne vehicle and a ground station. The protocol stack is structured as an application layer, transport layer, and link layer. Separate protocols have been presented that satisfy the safety requirements of weak delivery and guaranteed delivery. The model for each layer is described in some depth as is the composition of the layers. In addition, we have presented the theorems that characterize the functional correctness of the proposed protocol.

We aim to support an iterative protocol design process and a rapid prototyping environment. To this end, we propose a hierarchical verification approach where safety properties are first proved for the individual components of the protocol and then lifted to the composed system. This approach is largely mechanizable and we provide several proof strategies that automate most of the verification burden.

Our formal development in PVS consists of 28 theories and 129 lemmas. In total, there are 1758 lines of specification, 4987 lines of proofs, and 712 lines of strategy code. Most of these theories concern the specification and verification of the proposed protocol. However, we also provide a general theory for the asynchronous composition of transition systems and general strategies to prove invariants, abstractions, and the orthogonality and noninterference conditions. We believe that this verification framework can be applied to a family of distributed protocols, particularly those protocols that use data structures such as bags, FIFO queues, and bounded buffers.

REFERENCES

[1] B. Badban, W. Fokkink, J Groote, J. Pang, and J. van de Pol. Verification of a sliding window protocol in μCRL and PVS. *Formal Aspects of Computing*, 17:342–388, 2005.

[2] R. Bailey, R. Hostetler, K. Barnes, C. Belcastro, and C. Belcastro. Experimental validation subscale aircraft ground facilities and integrated test capability. In *Proceedings of the AIAA Guidance Navigation, and Control Conference and Exhibit 2005*, San Francisco, California, 2005.

[3] J. Brunekreff. Sliding window protocols. In *Algebraic Specification of Protocols*, number 36 in Cambridge Tracts in Theoretical Computer Science, pages 71–112. 1993.

[4] Rachel Mary Cardell-Oliver. *The Formal Verification of Hard Real-Time Systems.* PhD thesis, University of Cambridge, 1992.

[5] M. Charpentier. Composing invariants. *Science of Computer Programming*, 60:221–243, 2006.

[6] M. Charpentier and K. M. Chandy. Specification transformers: A predicate transformer approach to composition. *Act Informatica*, 40:265–301, 2004.

[7] D. Chkliaev, J. Hooman, and E. de Vink. Verification and improvement of the sliding windonw protocol. In *Proceedings of the 9th Conference on Tools and Algorithms for the Construction of Analysis of Systems (TACAS'03)*, Lecture Notes in Computer Science 2619, pages 113–127. Springer-Verlag, 2003.

[8] J.L.A. Van de Snepscheut. The sliding-window protocol revisited. *Formal Aspects of Computing*, 7:3–17, 1995.

[9] M. Gouda. *Elements of Network Protocols.* Wiley-Interscience, 1998.

[10] G. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1997.

[11] G. Holzmann. The model checker spin. *IEEE Transactionsactions of Software Engineerng*, 23(4):279–295, 1997.

[12] R. Hoogerwoord. A formal derviation of a sliding window protocol. Technical University of Eindhoven, 2006.

[13] C. Jones. Tentative steps toward a method for interfering programs. *ACM Transactions of Programming Languages and Systems (TOPLAS)*, 5(4):596–619, 1983.

[14] R. Kaivola. Using compositional preorders in the verification of a sliding window protocol. In *Proceedings of the 9th Conference on Computer Aided Verification*, Lecture Notes in Computer Science 1254, pages 48–59. Springer-Verlag, 1997.

[15] R. Manohar and P. Sivilotti. Composing processes using modified rely-guarantee. Technical report, California Institute of Technology, 1996.

[16] J. Misra and K. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, 1981.

[17] A. Murch. A flight control system architecture for the nasa airstar flight test infrastructure. AIAA Guidance, Navigation and Control Conference and Exhibit, 18-21 August 2008.

[18] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[19] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Proc. 11th Int. Conf. on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, June 1992.

[20] J. Rushby. Formal verification of McMilian's compositional assume-guarantee rule. Technical report, SRI, 2001.

[21] V. Rusu. Verifying a Sliding-Window Using PVS. In *Formal Techniques for Networked and Distributed Systems (FORTE01)*, pages 251–266. Kluwer Academic, 2001.

[22] K. Stahl, K. Baukus, K Lakhnech, and Y Steffen. Divide, abstract, and model check. In *Proceedings of the 6th International SPIN Workshop*, Lecture Notes in Computer Science 1680, pages 57–76, 1999.

[23] N. Stenning. A data transfer protocol. *Computer Networks*, 1(2):99–110, 1976.

[24] A. Tannenbaum. *Computer Networks.* Prentice Hall, third edition, 1996.

[25] F. Vaandrager. Verification of two communication protocol by means of proces algebra. Technical report, CWI, 1986.

[26] Q. Xu, W. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* 01- 04 - 2009	**2. REPORT TYPE** Contractor Report	**3. DATES COVERED** *(From - To)*

4. TITLE AND SUBTITLE Design and Verification of a Distributed Communication Protocol	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER NNX08AE37A
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Muñoz, César A.; and Goodloe, Alwyn E.	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER 645846.02.07.07.07
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) NASA Langley Research Center Hampton, VA 23681-2199	**8. PERFORMING ORGANIZATION REPORT NUMBER** NIA Report No. 2008-09
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, DC 20546-0001	**10. SPONSOR/MONITOR'S ACRONYM(S)** NASA
	11. SPONSOR/MONITOR'S REPORT NUMBER(S) NASA/CR-2009-215703

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 62
Availability: NASA CASI (443) 757-5802

13. SUPPLEMENTARY NOTES
Langley Technical Monitor: Paul S. Miner

14. ABSTRACT
The safety of remotely operated vehicles depends on the correctness of the distributed protocol that facilitates the communication between the vehicle and the operator. A failure in this communication can result in catastrophic loss of the vehicle. To complicate matters, the communication system may be required to satisfy several, possibly conflicting, requirements. The design of protocols is typically an informal process based on successive iterations of a prototype implementation. Yet distributed protocols are notoriously difficult to get correct using such informal techniques. We present a formal specification of the design of a distributed protocol intended for use in a remotely operated vehicle, which is built from the composition of several simpler protocols. We demonstrate proof strategies that allow us to prove properties of each component protocol individually while ensuring that the property is preserved in the composition forming the entire system. Given that designs are likely to evolve as additional requirements emerge, we show how we have automated most of the repetitive proof steps to enable verification of rapidly changing designs.

15. SUBJECT TERMS
Distributed protocols; Software verification; System analysis

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			STI Help Desk (email: help@sti.nasa.gov)
					19b. TELEPHONE NUMBER ***(Include area code)***
U	U	U	UU	33	(443) 757-5802