Proving correctness of the stop-and-wait protocol

Ethan Leba

April 21, 2021

Contents

1 Introduction

As networks facilitate communication across the globe in any range of conditions, the connections must be robust to various disruptions; packets may be lost, intercepted, or delayed. One protocol for dealing with lost packets is the stop-and-wait ARQ (Automatic Repeat Request) protocol. Stop-and-wait is used in Bluetooth communication, and can be seen as a special case of TCP with a sliding window of size 1.

In this paper, we describe an implementation of a model network using stop-and-wait in ACL2(s), and demonstrate a mechanical proof of the following theorem:

The data contained by the receiver will always be a prefix of the data being sent by the sender, regardless of network conditions.

1.1 Approach

After a brief investigation, we found that there are no pre-existing proofs of this protocol in ACL2. Furthermore, there are many variations on the stop-and-wait protocol, so we suspect that it's unlikely this specific model has been implemented and proven elsewhere.

1.2 Simplifications of the model

For the simplicity of the proof, we omit the following details.

Two-way communication. We use a one-way communication model, where the receiver cannot respond with any data.

Handshakes and teardowns. No handshakes and teardowns take place.

Packet delay. We assume that an ack/packet will either be sent, or dropped. No packet delay is modelled.

End of file. The simulation completes when the sender's data has been fully transmitted, so no EOF handling is needed.

1.3 Explanation of the protocol

Here we describe a high level overview of how the protocol functions, which will map to the ACL2 code provided in the implementation section. Both the sender and the receiver start with a sequence number starting at zero, representing the last packet sent or received, respectively.

1.3.1 Sender behavior

The sender will send out the (sequence number)'th chunk of data in a packet, along with the sequence number at the beginning of each round. The sender will wait for an ACK and respond in one of the following ways.

1. If no ACK is received, do nothing.

2. If an ACK is received with the expected number, increment the sequence number.

3. If an ACK is received with a greater number, set the sequence number to the one received.

1.3.2 Receiver behavior

The receiver will wait for a packet and respond one of the following ways.

1. If a packet is received with a sequence number one greater than the current sequence number, then store the packet data and set the current sequence number to the number received.

2. Otherwise, do nothing.

In either case, the receiver will respond with an ACK containing the current sequence number after potentially updating it.

2 Implementation

2.1 Model

Firstly, we will show the datatypes used to represent the model.

```
;; Represents the state of the sender, holding the data to send and
;; the current sequence number.
(defdata sender-state `(sendstate ,tl ,nat))

;; Represents the state of the receiver, holding the data received so
;; far.
(defdata receiver-state `(recvstate ,tl))

;; An event is one of:
;; - OK           both the packet and ack are transmitted.
;; - DROP-ACK     the packet is transmitted but the ack is dropped.
;; - DROP-PACKET  the packet is dropped, no packet to ack.
(defdata event (enum '(ok drop-ack drop-packet)))

;; Represents a sequence of events that occur in the simulated network
;; environment.
```

```
(defdata event-deck (listof event))

;; Represents the state of the simulation, holding the state of sender
;; and receiver.
(defdata sim-state `(sim-state ,sender-state ,receiver-state))
```

We opted for a tagged-list representation of the datatypes, to increase readability over untagged lists. We also implemented a macro, `match-let*`, which parallels racket's implementation and allows us to work with the tagged lists more easily. The `receiver-state` datatype does not contain an explicit sequence number. Instead, we infer the receiver's sequence number from the length of the data it has received.

Next, we will show `simulator-step`, which performs the progression of one round of the simulation.

```
(definec simulator-step (sim :sim-state event :event) :sim-state
  "Performs one round of the simulation with the given event."
  (if (simulator-completep sim) sim
    (match-let* ((('sim-state ('sendstate sdata sseq)
       ('recvstate rdata)) sim))
      (cond
       ;; Packet dropped
       ((== event 'drop-packet) sim)
       ;; OK packet, sender up-to-date
       ((and (== event 'ok) (== sseq (len rdata)))
 `(sim-state (sendstate ,sdata ,(1+ sseq))
     (recvstate ,(app rdata (list (nth sseq sdata))))))
       ;; OK packet, sender is behind
       ((and (== event 'ok) (!= sseq (len rdata)))
 `(sim-state (sendstate ,sdata ,(len rdata))
     (recvstate ,rdata)))
       ;; Ack dropped, sender up-to-date
       ((and (== event 'drop-ack) (== sseq (len rdata)))
 `(sim-state (sendstate ,sdata ,sseq)
     (recvstate ,(app rdata (list (nth sseq sdata))))))
       ;; Ack dropped, sender behind
       ((and (== event 'drop-ack) (!= sseq (len rdata))) sim)))))
```

The behavior of the protocol described in Section 1.3 is coalesced into one algorithm, which contains each of the possible outcomes of a round. For a precise description of the mapping, see Appendix A. We have also defined a predicate `simulator-completep`, which checks if the sender has sent all of it's data.

Finally, we see `simulator`, which performs the entire simulation with the given simulation state and network conditions.

```
(definec simulator (sim :sim-state steps :event-deck) :sim-state
  "Repeatedly applies simulator-step with the events specified."
  (cond
   ((lendp steps) sim)
   (T (simulator-step (simulator sim (cdr steps)) (car steps)))))
```

2.2 Proof

Firstly, in order to state the theorem programmatically we define two functions to create a predicate for the receiver-sender prefix property, as defined in `rs-prefix-of-ssp`.

```
(definec prefixp (x :tl y :tl) :bool
  "Checks if X is a prefix of Y."
  (cond
   ((lendp x) T)
   ((lendp y) (lendp x))
   (T (and (equal (car x) (car y))
    (prefixp (cdr x) (cdr y))))))

(definec rs-prefix-of-ssp (sim :sim-state) :bool
  "Check if the receiver's data is a prefix of the sender's."
  (match-let* ((('sim-state ('sendstate ss &)
      ('recvstate rs)) sim))
    (prefixp rs ss)))
```

The theorem we stated in the introduction is represented by the following `defthm`:

```
(defthm simulator-prefix-property
  (implies (and (tlp d)
 (event-deckp evts))
    (rs-prefix-of-ssp
     (simulator `(sim-state (sendstate ,d 0)
    (recvstate ())) evts))))
```

The lemma states that given an reasonable initial simulator state (where the sender's sequence number is zero, and the receiver has not collected any information), the receiver-sender prefix property holds regardless of the data being sent or network conditions. This is a specific case of a general property that we will prove:

```
(defthm simulator-maintains-prefix-property
  (implies (and (sim-statep sim)
 (event-deckp evt)
 (rs-prefix-of-ssp sim))
    (rs-prefix-of-ssp (simulator sim evt)))
  :hints (("Goal"
    :induct (simulator sim evt)
    :in-theory (disable simulator-step-definition-rule))))
```

This lemma states that given any starting simulation state where the receiver-sender prefix property holds, and any set of events to occur during the simulation, the prefix property holds after applying the simulation to the provided state.

We can prove this inductively, by showing that:

1. The base case maintains the prefix property.
2. The inductive step, which is applying `simulator-step` to the recursion, maintains the property.

The base case is trivial, as the simulator returns the state of the simulator if there are no events left. We now show that each simulator step maintains the prefix property.

```
(defthm simulator-step-prefix-property
  (implies (and (sim-statep sim)
 (rs-prefix-of-ssp sim)
 (eventp evt))
    (rs-prefix-of-ssp (simulator-step sim evt))))
```

With no helper lemmas, ACL2 is unable to prove the above. We introduce a more general lemma which can be applied in the proof.

```
(defthm prefix-nth
  (implies (and (tlp x)
  (tlp y)
  (prefixp x y)
  (< (len x) (len y))
  (== index (len x)))
     (prefixp (app x (list (nth index y))) y)))
```

This lemma shows that given a list X that is smaller than, and a prefix of Y, adding the next element of Y maintains the prefix property. This parallels the behavior of `simulator-step` when a packet is received with an expected sequence number.

```
 (defthm simulator-step-prefix-property
   (implies (and (sim-statep sim)
  (rs-prefix-of-ssp sim)
  (eventp evt))
     (rs-prefix-of-ssp (simulator-step sim evt)))
   ;; Applying the prefix-nth lemma to the OK and DROP-ACK subgoals
   :hints (("Subgoal 5'5'" :use
     (:instance prefix-nth
        (y sim8)
        (x sim9)
        (index (len sim9))))
    ("Subgoal 2'5'" :use
     (:instance prefix-nth
        (y sim8)
        (x sim9)
        (index (len sim9))))))
```

We can now return to Lemma `simulator-step-prefix-property`. ACL2 performs a proof by cases, and we can now apply an instance of this lemma to Subgoals 5 and 2, the cases when the receiver accepts and appends a packet. These are the only two cases where the receiver's data is extended, so the other cases hold trivially.

Now with Lemma `simulator-step-prefix-property`, ACL2 is able to prove `simulator-maintains-prefix-property`, and the `simulator-prefix-property` corollary follows from this.

3 Conclusion

3.1 Results

In this paper, we meet our planned criteria for success: proving the the receiver-sender prefix property of the stop-and-wait protocol. The network protocols used in practice are far more complex than the protocol we have reasoned about, but the behavior of our model is the foundation that protocols such as TCP have built off of.

3.2 Personal Progress

The proof for this paper was constructed incrementally, slowly increasing the complexity of the model. However, the original model proved to be too complex to prove correctness with packet loss, so it had to be reworked. One major issue with that model was the fact that the sender would discard any data that was confirmed to be received, by replacing the data with it's `cdr`, and sending the `car` instead of using `nth`. This at first seemed easier for ACL2 to reason about, but proving the receiver-sender prefix property while the contents of the sender's data was changing made the inductive proof far more challenging. In addition, the first model had dedicated functions for the receiver and sender, but we opted to coalesce the behavior for the final model into `simulator-step` in order to simplify the proof by cases.

Another challenge was representing the datatypes for the model. The record type for `defdata` seemed to be a great fit, but using records in proofs seemed to cause great difficulty for ACL2. Instead we opted for using a tagged list structure in the final model.

3.3 Summary

We have now shown a mechanized proof in ACL2 proving the correctness of the prefix property of the stop-and-wait protocol, a fundamental property of network communication that countless applications rely on. Looking forward, this model has potential to be used as a foundation for reasoning about more complex network properties and protocols in ACL2.

4 Appendix A

The `simulator-step` function describes 5 possible outcomes for each set of events and simulation state. Here we state precisely how the ACL2 function maps to the description in Section 1.3.

Packet dropped The receiver does not receive a packet, so it does not respond, and the sender performs step 1.

OK packet, sender up-to-date The receiver performs step 1, and the sender performs step 2.

OK packet, sender is behind The receiver performs step 2, and the sender performs step 3.

Ack dropped, sender up-to-date The receiver performs step 1, and the sender performs step 1.

Ack dropped, sender behind The receiver performs step 2, and the sender performs step 1.

5 Source Code

The source code for the project can be viewed at `https://github.com/ethan-leba/stop-and-wait-arq-proof`.