A Wireless Link Protocol: Design by Refinement

M. G. Gouda
Department of Computer Science
The University of Texas
Austin, TX 78712
gouda@cs.utexas.edu

Sanjoy Paul
AT&T Bell Laboratories
Holmdel, New Jersey 07733
sanjoy@research.att.com

Abstract

In this paper, we develop an asymmetric protocol for wireless communication in a step-by-step manner. We start with a very simple protocol and prove its correctness. Then we relax the assumptions of the simple protocol one by one, verifying the correctness of the protocol at each step as we relax the assumptions. This process is continued in a systematic manner until no assumptions are left. The novelty of the paper lies in the way the assumptions are relaxed without violating the correctness properties of the protocol while at the same time making the protocol efficient. The final result is a provably correct protocol which is also efficient for wireless channels.

1 Introduction

With the increasing popularity of wireless communication, there is a growing need for new protocols to accomodate the specific properties of a wireless channel in an efficient way so that a mobile terminal with better performance, lower power, and smaller size can be designed. In addition, because of the effects of fading, interference and mobility, the error rate incurred in a mobile wireless channel is often very high. This has two effects. First, in current systems, end-to-end protocols must recover from these errors, often using retransmission timers. These timers are typically set to values on the order of seconds to allow for variable network delays. This causes the recovery time of an error that occurs on a wireless channel to be long. Secondly, losses incurred on the wireless channel trigger the end-to-end transmitters to decrease the window size and increase the duration of the retransmission timers as observed by Caceres and Iftode in [CI93][CI94]. This leads to lower throughput and higher latency. Therefore, correcting errors at the link-layer in addition to end-to-end correction will result in better performance compared to only end-to-end correction.

We envision a network architecture which has a wired network as its backbone with base stations on it acting as access points for the mobile terminals (Figure 1). Mobile terminals communicate through the base stations with other hosts. In the context of this architecture, a *wireless channel* refers to the channel between a mobile terminal and a base station.

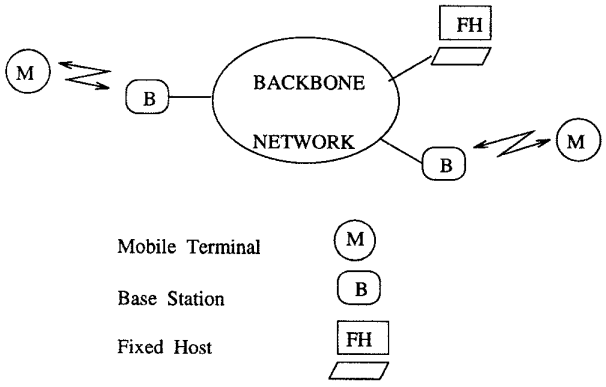

Figure 1: Wireless Network Architecture

The configuration with a wireless channel between a base station (which is wired to the backbone network) and a mobile terminal (which has no wired connection) is inherently *asymmetric* in nature. The asymmetry can be attributed to the fact that the mobile terminal has *limited power* and *lower processing capability* compared to the base station. In order to accomodate this asymmetry, we propose to put as much *intelligence* as

0-8186-7216-1/95 $04.00

possible in terms of processing in the base stations and make the mobile terminals *relatively* dumb.

In addition to the asymmetric nature of the configuration, the medium by itself has certain properties.

1. The bandwidth (Hz) is limited and bps (bits per second) needs are constantly increasing. Therefore, bandwidth needs to be conserved.
2. Bit error rate is variable, but can be very high ($\approx 10^{-2}$) at times.
3. There is a *fixed processing delay* of at most 100 ms. at each end of a wireless channel associated with processing of the packets before sending a response. This delay defines a window in terms of the number of packets that can be in the pipeline between a transmitter and the corresponding receiver at the two ends of a wireless channel.

The above mentioned properties of a wireless channel lead to conflicting requirements in the design of a protocol. For example, since the bandwidth is expensive, it is necessary to minimize retransmissions. On the other hand, the error rate being high, it is desirable to make redundant transmissions with the assumption that the probability of a packet being delivered correctly is higher if it is transmitted more than once. This tradeoff must be taken into consideration while designing a protocol.

Keeping the above considerations in mind, we present the design of an asymmetric protocol for a wireless link. The key ideas involved in the design are as follows:

1. Timers are *always* at the base station regardless of whether it is transmitting or receiving. Thus the *intelligence* in terms of maintaining timers, processing complex status messages and most importantly, making decisions *resides* in the base station.
2. The base station receiver sends its status to a mobile transmitter *periodically*, like the SNR protocol [NRS90]. The justification for using periodic status messages is to reduce the dependence on the error prone medium. In particular, if the wireless channel is bad and the base station does not receive packets, it can still send status messages, because sending of status messages is triggered by the timeout signal of a local timer and not by the event of receiving a packet. Also, if a status message gets lost, there is always a next one to bank on.

   Note that the period of sending status messages can be adjusted so as not to waste much bandwidth for control information while at the same time offsetting the effect of a noisy channel.
3. The mobile receiver *does not* send its status to the base transmitter periodically because of its power constraint. These status messages are event driven.
4. The mobile receiver *combines* several status messages into one status message to conserve power. That is, the mobile terminal does not send a status message after receiving each packet. Rather it sends a status message after receiving a block (a set of packets). Since transmitting status messages consumes battery power, the mobile can conserve battery power by transmitting its status message after receiving a block (set of packets) rather than after receiving every packet.
5. The protocol uses *selective repeat* retransmission mechanism as opposed to *go back n* to prevent redundant retransmission. This helps to conserve the expensive cellular bandwidth.

The protocol described in [APL+95, PLA+95] is based on similar ideas. However, the focus of this paper is different from those in [APL+95] and [PLA+95]. In this paper, we show the systematic development of the above protocol starting from a very simple protocol[1]. After formally specifying the simple protocol, we prove it to be correct and as we keep refining the protocol, we prove that the correctness of the protocol is maintained at each step. The remaining part of the paper is organized as follows. The next section describes the first attempt in formally specifying a simple protocol, followed by the verification in section 3. Observations leading to the modification of the simple protocol are presented in section 4, followed by the formal specification of the final protocol in one direction. Section 5 presents the final protocol in both directions and the conclusion is given in section 6.

2 First Attempt

The protocol we consider in this paper consists of two processes called receiver and sender. The receiver process, q, has an infinite buffer where it stores the data mesages received from the sender process, p, until the messages are removed by the host of the receiver.

[1]The performance figures and comparison with similar protocols are presented in [APL+95].

The receiver q sends the current state of its infinite buffer to the sender p at random (non-deterministically)[2]. On receiving a status message, p can decide which packets have been lost during transmission and so need to be retransmitted. It can also decide how many new packets it can send to q. The state of the infinite buffer consists of three components:

(rcvd, qr, qt)

where *rcvd* is an infinite boolean array whose ith. entry is 1 or 0 depending on whether the ith. packet has been received or not; qr and qt are two pointers which divide the buffer into three regions:

(1) An *already-received region* from the beginning of the buffer to just before qr.

(2) A *current-window region* from qr to just before qt.

(3) A *future-window region* from qt to the end of buffer.

p can send and q can receive packets in the second region while they cannot do so either in the first region or the third region. Old packets still reside waiting to be removed by the host of the receiver in the first region, while there are empty spaces in the third region reserved for future transmissions by p.

Transmitted packets may be lost in the channel. In order to detect which packets are lost, the packets are given a sequence number, which is an integer in our simple protocol. The sender also has an infinite buffer to store packets delivered to it by the host of the sending process and an infinite boolean array *ackd* to keep track of which packets have been acknowledged by the receiver. The ith. entry of array *ackd* is 1 or 0 depending on whether the receiver has acknowledged the receipt of the ith. packet or not. The array has three pointers pr, pt and ps. pr indicates the next packet to be acknowledged (that is, all packets until pr-1 have been acknowledged by the receiver), pt indicates the next packet that cannot be sent and ps indicates the next packet to be transmitted (that is, ps-1 is the highest packet number transmitted so far).

When the sender p receives a status message (*ackd, pr, pt*) from q, for all packets between pr and ps which have not been acknowledged by the receiver (i.e., $ackd[i] = 0$), it decides non-deterministically[3] to retransmit the corresponding packet.

If the window is open (that is, $ps \neq pt$), the sender can send new packets.

[2]This condition will be refined later on.

[3]We become more specific about the retransmission mechanism in later versions.

The receiver process q, on receiving a data packet $data(j)$ from the sender process p, updates the $rcvd(j)$ entry if j is within the *current-window region* and moves qr until there is a hole (entry corresponding to a packet not received) in the $rcvd$ array. If j is outside the *current-window region*, the receiver does nothing.

Before formally specifying the protocol, let us introduce the notation next.

2.1 Notation

Each process of the protocol is defined by a set of local variables and a set of actions.

process ⟨ process name ⟩

const ⟨ Pascal-like declarations of constants ⟩

var ⟨ Pascal-like declarations of local variables ⟩

begin

⟨ action name ⟩

□ ⟨ action name ⟩

...

□ ⟨ action name ⟩

end

Each action is of the form:

⟨ label ⟩ : ⟨ guard ⟩ → ⟨ statement ⟩

The label is a number that uniquely identifies the action. The guard is either a boolean expression or a receive statement of the form: **rcv** g, for some message g. The statement is defined recursively as one of the following:

a **skip** statement

an assignment statement

a send statement of the form: **send** g

a sequence of statements of the form:

⟨ statement ⟩; ⟨ statement ⟩

a conditional statement of the form:

if ⟨ condition ⟩ **then** ⟨ statement ⟩ **fi** or

an iterative statement of the form:

for ⟨ condition ⟩ **do** ⟨ statement ⟩ **od**

Terminology

There are two *one-directional channels* between processes p and q. At each instant during protcol execution, each channel has a sequence of messages. The message sequence in the channel from q to p contains all status messages that have been sent by q but not yet received by p. The message sequence in the channel from p to q contains all the data messages that have been sent by p but not yet received by q. The sending of a message from p to q (or from q to p) consists of adding the message at the tail of the message sequence in the channel from p to q (or from q to p). The receiving of a message from p to q (or from q to p) consists of removing the message at the head of the message sequence in the channel from p to q (or from q to p). The message sequence in the channel is updated by executing send or receive statements.

A *protocol state* is defined by a value for each local variable in processes p and q, a sequence of status messages in the channel from q to p and a sequence of data messages in the channel from p to q.

An *action* in process p (or q) is enabled for execution at a protocol state s if the guard of the action is a boolean expression that evaluates to true at s, or if the guard is a receive statement **rcv** g, and g is the head message in the message sequence in the channel from p to q (or from q to p) at state s.

Execution of an enabled action in process p (or q) depends on whether the guard of the action is a boolean expression or a receive statement. If the guard is a boolean expression, the action is executed by executing the statement of the action. If the guard is a receive statement, the action is executed by first receiving the head message from the message sequence in the channel from p to q (or from q to p), then executing the statement of the action.

A *protocol computation* is a maximal sequence of the form:

State.0; action.0; state.1; action.1; state.2; $\cdots$

where each state $state.i$ is a protocol state, and each action $action.i$ is an action in p or q that is enabled for execution at $state.i$. Moreover, executing $action.i$ starting at $state.i$ yields $state.(i+1)$. The maximality of the sequence means that the sequence is infinite, or it is finite but no action is enabled for execution at its last state.

As mentioned earlier, we make several assumptions for the simple protocol. In particular, we make the following assumptions:

1. Variables used for the pointers of the window (variables pr, pt, ps, i, qr, qt, j in sections 2.2 and 2.3) are of type integer.
2. Large Arrays are used for maintaining the status of packets at the sender and the receiver (array variables $ackd$ and $rcvd$ in sections 2.2 and 2.3)
3. Sender retransmits non-deterministically (action 1 in process p of section 2.2)

Formal specifications of the transmitting process (which we call p) and the receiving process (which we call q) follow:

2.2 Sender

process p:

const w $\{0 < w\}$

var ackd : **array [integer] of boolean**

pr, pt, ps, i : **integer**

begin

1: **rcv** st(ackd, pr, pt) **from** q $\rightarrow$
i := pr;
do i $\neq$ ps $\rightarrow$
if true $\rightarrow$ **skip**
$\square \neg$ ackd[i] $\rightarrow$ **send** data(i) **to** q
fi;
i := i+1
od

$\square$ 2: ps $\neq$ pt $\rightarrow$ **send** data(ps) **to** q; ps := ps+1

end

2.3 Receiver

process q:

const w

var rcvd : **array [integer] of boolean**

qr, qt, j : **integer**

begin

3: **true** $\rightarrow$ **send** st(rcvd,qr,qt) **to** p

$\square$ 4: **rcv** data(j) **from** p $\rightarrow$

if (qr $\leq$ j < qt) $\wedge \neg$ rcvd[j] $\rightarrow$ rcvd[j] = **true**
$\square \neg$ (qr $\leq$ j < qt) $\vee$ rcvd[j] $\rightarrow$ **skip**

fi;

do rcvd[qr] $\rightarrow$ rcvd[qr], qr := **false**, qr := qr+1 **od**

$\square$ 5: qt $\neq$ qr + w $\rightarrow$ qt := qt+1

end

3 Protocol Verification

Correctness of any protocol can be established by showing that the protocol satisfies two logical properties: safety and progress. These two properties can be expressed in terms of some predicate C that assigns a value, true or false, to every state of the protocol.

1. **Safety:** This property states that if the protocol is at any state where C holds, then executing any action in p or q yield a state where C holds; that is, C is closed under protocol execution.
2. **Progress:** This property states that for every protocol computation that starts with a state where C holds, each of the three pointers ps, qr and qt is incremented by one infinitely often along the computation.

A predicate C that satisfies these two properties defines a closed domain for protocol execution. In this closed domain, safety is always guaranteed and progress is guaranteed infinitely often. Such a predicate is called a protocol closure [G93] or a protocol invariant.

For our protocol, we adopt the following predicate as a candidate for a protocol closure.

C = C.0 $\wedge$ C.1 $\wedge$ C.2 $\wedge$ C.3 $\wedge$ C.4 $\wedge$ C.5 $\wedge$ C.6 $\wedge$ C.7 $\wedge$ C.8 $\wedge$ C.9 $\wedge$ C.10

where

C.0 = (pr $\leq$ ps $\leq$ pt $\leq$ pr+w)

C.1 = ($\forall$ st(b,r,t) in ch.q.p: r $\leq$ ps $\leq$ t $\leq$ r+w)

C.2 = (qr $\leq$ ps $\leq$ qt $\leq$ qr+w)

C.3 = (st(b,r,t) is head in ch.q.p $\Rightarrow$ pr $\leq$ r $\wedge$ pt $\leq$ t)

C.4 = (st(b,r,t) precedes st(b$'$,r$'$,t$'$)in ch.q.p $\Rightarrow$ r $\leq$ r$'$ $\wedge$ t $\leq$ t$'$)

C.5 = (st(b,r,t) is tail in ch.q.p $\Rightarrow$ r $\leq$ qr $\wedge$ t $\leq$ qt)

C.6 = (ch.q.p = ϕ $\Rightarrow$ pr $\leq$ qr $\wedge$ pt $\leq$ qt)

C.7 = ($\forall$ data(i) in ch.p.q: qr-w $\leq$ i < ps

C.8 = (data(i) precedes data(j) in ch.p.q $\wedge$ i > j $\Rightarrow$ i-j $\leq$ w-1

C.9 = (data(i) and data(j) in ch.p.q $\wedge$ i > j $\Rightarrow$ i-j $\leq$ 2w-1

C.10 = ($\forall i$: rcvd[i] $\Rightarrow$ i < ps $\wedge$ ($\forall$ data(j) in ch.p.q where i > j, i-j $\leq$ w-1))

In order to show that C is indeed a protocol closure, we need to show that it satisfies the above safety and progress properties. This is done in the next two subsections. Before that, we consider an initial state of the protocol which satisfies

D = D.0 $\wedge$ D.1 $\wedge$ D.2 $\wedge$ D.3 where

D.0 = (pr = ps = qr = 0 $\wedge$ pt = qt = 1)

D.1 = ($\forall k$: $\neg$ rcvd[k])

D.2 = Channel from p to q is empty

D.3 = Channel from q to p is empty

Since D.0 satisfies C.0 and C.2; D.1 satisfies C.10; D.2 satisfies C.7, C.8 and C.9; D.3 satisfies C.1, C.3, C.4 and C.5; D.3 and D.0 together satisfy C.6, any protocol state that satisfies D.0, D.1, D.2 and D.3 also satisfies C. Therefore, the protocol can start executing at any state that satisfies D.

3.1 Proving Safety

To prove safety, it is sufficient to prove that if every action in p or q is executed at a state where C holds, then C also holds at the resulting state. We give here the proof for only one action, action 4 in process q. (The proofs for all other actions are similar.)

Recall that action 4 is as follows:

4: **rcv** data(j) **from** p $\rightarrow$

if (qr $\leq$ j < qt) $\wedge$ $\neg$ rcvd[j] $\rightarrow$ rcvd[j] = **true**

$\square$ $\neg$ (qr $\leq$ j < qt) $\vee$ rcvd[j] $\rightarrow$ **skip**

fi;

do rcvd[qr] $\rightarrow$ rcvd[qr], qr := **false**, qr := qr+1 **od**

Since this action does not update any of the variables pr, ps, pt, it cannot invalidate C.0. Also this action does not add (remove) any message to (from) channel ch.q.p, it cannot invalidate C.1, C.3, C.4. It remains to

be shown that action 4 cannot invalidate C.2, C.5, C.6, C.7, C.8, C.9 and C.10. If C.5 and C.6 are true before action 4 is executed, they will be true after action 4 is executed because qr is either incremented or it remains the same and pr, pt and qt remain unchanged. Also, if C.8 and C.9 are true before the execution of action 4, they will remain true after the execution of action 4 because no message is added to $ch.p.q$.

Now, we need to show that C.2, C.7 and C.10 hold after the execution of action 4 in order to support our claim that C is the protocol closure. First we show that C.2 is preserved after execution of action 4. Regarding the array $rcvd$ in q, there are two possible scenarios:

(1) $rcvd[qr-1]$ is the last element of the array $rcvd$ which is 1 (true) or

(2) There is an $i > qr$ such that $rcvd[i]$ is 1 (true).

If $rcvd$ array is in state (1) before action 4 and a data message with number qr is received, qr will be incremented by one after action 4. Since C.7 is true before action 4, the incremented qr after action 4 can be at most equal to ps.

If $rcvd$ array is in state (2) before action 4 and a data message with number qr is received, qr may be incremented more than once but it can at most reach ps because C.10 is true before action 4 is executed and every i for which $rcvd[i]$ is true is less than ps.

Thus qr can at most be equal to ps after action 4 is executed. This implies that C.2 will hold after the execution of action 4.

C.7 can be violated since qr is incremented in action 4. But we will show that such a thing does not occur. In action 4, $rcvd[i]$ is set to 1 provided i is at least equal to qr and qr is incremented provided $i = qr$. Since C.8 is true before the execution of action 4 and $i = qr$ when qr is incremented in action 4, C.7 will hold after the execution of action 4.

Since C.7 is true before action 4, there is no data message numbered ps in $ch.p.q$. In action 4, the data message at the head of the queue in $ch.p.q$ is removed from the queue and the corresponding entry in $rcvd$ array is set to true. These two statements together imply that after action 4 is executed, $i < ps$ clause of C.10 holds. Also, since C.8 was true before the execution of action 4, the second clause of C.10: ($\forall$ data(j) in ch.p.q where i > j, i-j $\leq$ w-1) will also hold after action 4 is executed. Thus action 4 does not invalidate C.10.

In conclusion, the execution of action 4 does not invalidate the invariant C. This completes our proof that if action 4 is executed at a state where C holds, then C also holds at the resulting state.

3.2 Proving Progress

We need to prove that each of the variables qr, ps and qt is incremented by one infinitely often along any protocol computation that starts with a state where C holds. First, we show that every protocol computation is infinite. Second, we show that along every infinite computation that starts with a state where C holds, each of the variables qr, ps and qt is incremented by one infinitely often.

To prove the first part, note that action 3 in process q is enabled for execution at each state of the protocol. Therefore, each protocol computation is infinite.

To prove the second part, let c be an arbitrary computation of the protocol, and assume that c starts with a state where C holds. From the safety property, C holds at every state in computation c. Therefore, the relation C.2 holds at every state in c. Hence, it is enough to show that at least one of the variables qr, ps or qt is incremented by one infinitely often along c. Because qr is incremented in action 4, qt is incremented in action 5 and ps is incremented in action 2, we need to show that at least one of the actions 2, 4 or 5 is executed infinitely often along computation c.

We adopt the following fairness assumption:

If any action is continuously enabled along some computation, then this action is eventually executed along that computation.

Along computation c, if any of the actions 2, 4 or 5 is enabled for execution, then this action will continue to be enabled for execution until it is executed by the fairness assumption. Therefore, all we need to show is that if computation c has a state where none of the actions 2, 4 and 5 is enabled for execution, then c has a subsequent state where one of these actions is enabled for execution.

Let S be a state in computation c where none of the actions 2, 4 and 5 is enabled for execution. The following condition holds at state S.

$(ps = pt) \wedge (qt = qr + \mathrm{w}) \wedge \neg\, rcvd[qr]$

Since action 3 is always enabled, q will send $st(rcvd, qr, qt)$ to p and will be received by p as $st(ackd, pr, pt)$. Say the new state is S'. Now, in state S', $pr = qr$ and $pt = qt$. Since $qt = qr$ + w in state S, $pt = pr$ + w after reception of $st(ackd, pr, pt)$. Note that ps remains the same in state S' as it was in S. This enables action 1 in p which eventually leads p to send $data(pr)$ where $pr = qr$. On receiving $data(pr)$

in action 4, $rcvd[qr]$ will be true and qr will be incremented.

Incrementation of qr will enable action 5 and will lead to the incrementation of qt. Action 3, which is always enabled, will lead to sending of $st(rcvd, qr, qt)$, reception of which will enable action 1 and also action 2. Thus ps will be eventually incremented.

This completes our proof of progress for the simple protocol.

4 Next Attempts

In this section, we make several key observations, which help us to get rid of the assumptions in the simple protocol one by one, while preserving the correctness of the protocol. For brevity, the revised protocol specification will not be shown after each refinement. However, the protocol after the final modification will be presented.

Observation 1: It is clear from condition C.9 of the previous section that at any instant of time, sequence numbers of the data messages in the channel $ch.p.q$ can differ by at most $2w$ - 1.

Implication 1: The sequence number space needs to range from 0 to $2w$ - 1 and all arithmetic (in particular, incrementation) can be done *modulo* n where n is at least $2w$. Therefore, variables pr, pt, ps, i, qr, qt and j are no longer simple integers. Also the arrays $ackd$ and $rcvd$ are at most of size $2w$.

With modular arithmetic, we need to define a predicate:

bet (qr, j, qt) $\equiv$ (qr $\neq$ qt) $\wedge$ (j = qr $\vee$ j = qr $+_n$ 1 $\vee$ $\cdots$ $\vee$ j = qt $-_n$ 1)

Observation 2: In the wireless point-to-point link, data messages can never be reordered because of the serial nature of transmission. Also, the sender does not send a window of packets unless the old window is acknowledged. This is reflected in the invariant C.8.

Implication 2: $ackd$ array in process p and $rcvd$ array in process q are at most of size w.

Observation 3: Introduction of a clock (which can be thought of as the regular clock in the protocol processor) and a new array called clk, whose ith. element $clk[i]$ keeps track of the time when data message with sequence number i is (re)transmitted, can reduce the number of redundant retransmissions and thereby, conserve expensive cellular bandwidth. The explanation follows:

Assume that the sender has some estimate of the round-trip delay d between itself and the receiver. When a retransmission request for a data message i arrives at the sender (say at time t), the sender compares the difference between t and the time when the data message i was (re)transmitted and if the difference is greater than the round-trip delay, the sender retransmits data message i (action 6 in process p). Note that if the receiver sends multiple retransmission requests for the same data message, only one of them will be responded to by the sender. This prevents redundant retransmissions and improves protocol efficiency.

Implication 3: The sender process p no longer retransmits non-deterministically (action 1 in the simple protocol is modified).

Here is the new formal specification of the protocol:

process p:

const n, w, d {0 < w < n/2}
  {d is any estimate of round-trip delay}

var ackd : **array** [0 $\cdots$ w-1] **of boolean** pr, pt,
  ps, i : 0 $\cdots$ n-1
  x : 0 $\cdots$ w
  clk : **array** [0 $\cdots$ w-1] **of integer** clock : integer

begin

  6: **rcv** st(ackd, pr, pt) **from** q $\rightarrow$
    i, x := pr, 0;
    do i $\neq$ ps $\rightarrow$
      if ackd[x] $\vee$ (clock-clk[i]) < d $\rightarrow$
        skip
      $\square$ $\neg$ ackd[i] $\wedge$ (clock-clk[i]) $\geq$ d $\rightarrow$
        send data(i) **to** q
      fi;
      i, x := i $+_n$ 1, x+1
    od

$\square$ ps $\neq$ pt $\rightarrow$ **send** data(ps) **to** q;
  ps, clk[ps] := ps $+_n$ 1, clock

$\square$ **true** $\rightarrow$ clock := clock+1

end

process q:

const n, w {0 < w < n/2}

var rcvd : **array** [0 $\cdots$ w-1] **of boolean** qr, qt, j : 0 $\cdots$ n-1
  y : 0 $\cdots$ w-1

begin

true → **send** st(rcvd,qr,qt) **to** p

□ **rcv** data(j) **from** p →

y := (n+j-qr) **mod** n;

if bet(qr, j, qt) ∧ ¬ rcvd[y] → rcvd[y] = **true**

□ ¬ bet(qr, j, qt) ∨ rcvd[y] → **skip**

fi;

do rcvd[0] →

qr, y := qr +$_n$ 1, 0;

do y ≠ w -$_n$ 1 →

rcvd[y], y := rcvd[y+1], y+1 **od**;

rcvd[w-1] := **false**

od

□ qt ≠ qr +$_n$ w → qt := qt +$_n$ 1

end

4.1 Proof of Correctness

The proof given in section 3 holds after modifications to the simple protocol (given in section 2) are made according to observations 2 and 3, because the invariants are not altered. In addition, since the protocol has been proven to be correct with non-deterministic retransmissions, it must be correct if the retransmissions are done at specific instants of time. This proves the new protocol to be correct.

5 Final Protocol in Two Directions

The final protocol for the wireless channel is obtained by combining the processes p and q in each of the two endpoints (the wireless unit and the base station) of the wireless link as shown in Fig.2

The same process is used for wireless unit wr and for base station bs except for the guard G and for the four statements S, S′, S″ and S‴ which are defined differently.

In fact, the statements S, S′, S″ and S‴ and the guard G introduce the *asymmetry* in the design of the protocol. As mentioned earlier, the asymmetry is needed because the wireless unit has limited battery power and lower processing capability as compared to the base station.

Note that the base station, when receiving, sends its status periodically to the wireless transmitter (action 10, guard G). The periodicity is maintained using a

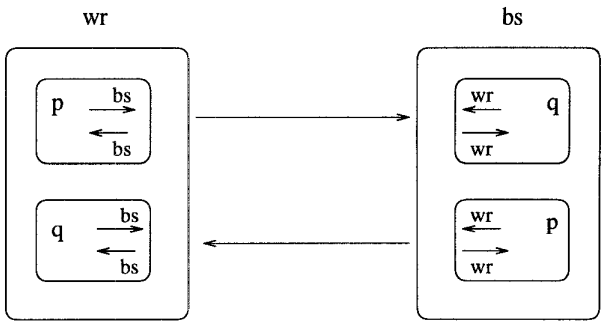

Figure 2: Final Protocol for Wireless Link in both directions

Timer. The wireless unit has no Timer, regardless of whether it is transmitting or receiving. This helps to keep the protocol on the wireless unit simple. Therefore, the wireless receiver sends its status either when it is polled (action 10, guard G (sndt = **true**)) or when it has received a complete block (action 10, guard G (c = BlockSize)).

Each time the wireless unit transmits its status to the base transmitter, it consumes some battery power. Therefore by sending its status on receiving a complete block (action 10, guard G (c = BlockSize)) as opposed to sending its status on receiving every packet, the wireless unit conserves battery power.

The base transmitter, on setting the variable $rcvt$ in S′ (actions 8 and 10) and sending it as part of the status message in action 10, is able to *poll* the wireless receiver, if it does not hear from the wireless unit for some time. This is necessary because the wireless receiver does not have a timer to detect the loss of a status packet that it sent to the base station and it needs to be informed by the base (about the loss of status packets) by sending *poll* messages. This is not necessary in the other direction.

Formal specification of the final asymmetric protocol in both directions is given next.

process wr:

const n, w, d {0 < w < n/2},

{d is any estimate of round-trip delay}

BlockSize

var ackd : **array** [0 · · · w-1] **of boolean**

pr, pt, ps, i : 0 · · · n-1

x : 0 · · · w

clk : **array** [0 · · · w-1] **of integer**

clock, c : integer

```
    rcvd :      array [0 ··· w-1] of boolean
    qr, qt, j :     0 ··· n-1
    y :     0 ··· w-1
    rcvt, sndt : boolean {* new variables *}
begin
    7: rcv st(ackd, pr, pt, rcvt) from bs →
        S;
        i, x := pr, 0;
        do i ≠ ps →
            if ackd[x] ∨ (clock-clk[i]) < d →
               skip
                □ ¬ ackd[i] ∧ (clock-clk[i]) ≥
                d → S';send data(i, rcvt) to bs
            fi;
            i, x := i +n 1, x+1
        od
□   8: ps ≠ pt → S' send data(ps, rcvt) to bs; ps,
    clk[ps] := ps +n 1, clock
□   9: true → clock := clock+1
□   10: G → S'; send st(rcvd, qr, qt, rcvt) to bs; S''
□   11: rcv data(j, rcvt) from bs →
        S;
        y := (n+j-qr) mod n;
        if bet(qr, j, qt) ∧ ¬ rcvd[y] → rcvd[y] = true
        □ ¬ bet(qr, j, qt) ∨ rcvd[y] → skip
        fi;
        do rcvd[0] →
            S'''; qr, y := qr +n 1, 0;
            do y ≠ w -n 1 →
            rcvd[y], y := rcvd[y+1], y+1 od;
            rcvd[w-1] := false
        od
□   12: qt ≠ qr +n w → qt := qt +n 1
end
```

6 Conclusion

In this paper, we have done a systematic design of an asymmetric protocol in a step-by-step manner maintaining the correctness properties of the protocol at each step of refinement. In addition to that, we have taken into account the specific properties of a wireless channel to make the final protocol not just correct but also efficient.

	wr	bs
S	**if** rcvt → sndt := **true** □ ¬ rcvt → **skip** **fi**	**skip**
S'	**skip**	rcvt := **any**
S''	sndt := **false**	**skip**
S'''	c := c+1	**skip**
G	sndt = **true** ∨ c = BlockSize	timeout

Table 1: Additional Statements and Guards

References

[APL+95] E. Ayanoglu, S. Paul, T.F. La Porta, K.K. Sabnani and R.D. Gitlin, "AIRMAIL: A Link-Layer Protocol for Wireless Networks," *Wireless Networks*, Vol.1,No.1, pp. 47-60.

[CI93] R. Caceres and L. Iftode, "The Effects of Mobility on Reliable Transport Protocols," *Proceedings 14th. ICDCS*, June 1994.

[CI94] R. Caceres and L. Iftode, "The Effects of Mobility on Reliable Transport Protocols," *JSAC, special issue on Mobile Computing Networks*, 1994.

[NRS90] A. N. Netravali, W. D. Roome and K. K. Sabnani, "Design and Implementation of a High Speed Transport Protocol," *IEEE Transactions on Communications*, Vol.38, No.11, November 1990.

[GNS94] M.G. Gouda, A. N. Netravali and K. K. Sabnani, "A Periodic State Exchange Protocol and its Verification," *submitted for publication.*

[PLA+95] S. Paul, T.F. La Porta, E. Ayanoglu, K-W.H. Chen, K.K. Sabnani and R.D. Gitlin, "An Asymmetric Link-Layer Protocol for Digital Cellular Communications," *Proceedings of IEEE INFOCOM*, pp. 1053-1062, Boston, April 4-6, 1995.