APPENDIX 1

The section lists the source code for TCP-WLAware. The modified TCP-NewReno to bring forth TCP-WLAware is specifically listed

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include "packet.h"
#include "ip.h"
#include "tcp.h"
#include "flags.h"

static class NewRenoTcpClass : public TelClass {
public:
    NewRenoTcpClass() : TelClass("Agent/TCP/Newreno") { }
    TclObject* create(int, const char*const*) {
        return (new NewRenoTcpAgent( ));
    }
} class_newreno;

NewRenoTcpAgent::NewRenoTcpAgent() : newreno_changes_(0),
    newreno_changes1_(0), acked_(0), firstpartial_(0),
```
partial_window_deflation_0, exit_recovery_fix_0,
arq_recovery_enable_0, arq_recovery_0, arq_recovery_seqno_0,
}
torecover = new PacQueue;
bind("newreno_changes_", &newreno_changes_);
bind("newreno_changes1_", &newreno_changes1_);
bind("exit_recovery_fix_", &exit_recovery_fix_);
bind("partial_window_deflation_", &partial_window_deflation_);
bind("arq_recovery_enable_", &arq_recovery_enable_);
}

/*
* Process a packet that acknowledges previously unacknowledged data, but
* does not take us out of Fast Retransmit
*/

void NewRenoTcpAgent::partialnewack(Packet* pkt)
{
    hdr_tep *tcph = hdr_tep::access(pkt);
    if (partial_window_deflation_)
        // Do partial window deflation before resetting last_ack_
        unsigned int deflate = 0;
        if (tcph->seqno() > last_ack_ // assertion
            deflate = tcph->seqno() - last_ack_;
    else
        printf("False call to partialnewack: deflate %u \n"
            last_ack_ %d\n", deflate, last_ack_);
        if((arq_recovery_ && is_available(last_ack_ +1) == 1) ||
            !arq_recovery_enable_)
            if (dupwnd_ > deflate)
                dupwnd_ = (deflate - 1);
            else
cwnd_ -= (deflate - dupwnd_);
    // Leave dupwnd_ > 0 to flag "fast recovery" phase
    dupwnd_ = 1;
    }
  }
}
if (cwnd_ < 1) {cwnd_ = 1;}
}
last_ack_ = tcph_->seqno();
highest_ack_ = last_ack_;
if (t_seqno_ < last_ack_ + 1)
    t_seqno_ = last_ack_ + 1;
if (rtt_active_ && tcph_->seqno() >= rtt_seq_)
    {
    rtt_active_ = 0;
    t_backoff_ = 1;
    }
} /* This function handles */
void NewRenoTcpAgent::partialnewack_helper(Packet* pkt)
{
if (!newreno_changes1_ || firstpartial_ == 0) {
    firstpartial_ = 1;
    /* For newreno_changes1_,
    * only reset the retransmit timer for the first
    * partial ACK, so that, in the worst case, we
    * don't have to wait for one packet retransmitted
    * per RTT.
    */
    newtimer(pkt);
}
partialnewack(pkt);

/* Retransmit the lost segment when the 3rd duplicate arrives
 * If the sender in arq recovery phase retransmit the lost segment
 * only if the
 * segment is not in the arq_recover_list
 * */

if(arq_recovery_ && arq_recovery_enable_)
{
    if(is_available(last_ack_ +1) == 1) output(last_ack_ + 1, 0);
}
else output(last_ack_ + 1, 0);

int NewRenoTcpAgent::allow_fast_retransmit(int /* last_cwnd_action */) {
    return 0;
}

void NewRenoTcpAgent::dupack_action()
{
    int recovered = (highest_ack_ > recover_);
    int recoveredl = (highest_ack_ == recover_);
    int allowFastRetransmit = allow_fast_retransmit(last_cwnd_action_);
    if (recovered || (!bugfix_ && !ecn_) || allowFastRetransmit
        || (bugfix_ss_ && highest_ack_ == 0)) {
        // (highest_ack_ == 0) added to allow Fast Retransmit
        // when the first data packet is dropped
        // Bug report from Mark Allman
        goto reno_action;
    }
if (bug_fix_ && less_careful_ && recovered1) {
  /*
   * For the Less Careful variant, allow a Fast Retransmit
   * if highest_ack_ == recover.
   * RFC 2582 recommends the Careful variant, not the
   * Less Careful one.
   */
  goto reno_action;
}

if (ecn_ && last_cwnd_action_ == CWND_ACTION_ECN) {
  last_cwnd_action_ = CWND_ACTION_DUPACK;
  /*
   * What if there is a DUPACK action followed closely by ECN
   * followed closely by a DUPACK action?
   * The optimal thing to do would be to remember all
   * congestion actions from the most recent window
   * of data. Otherwise "bugfix" might not prevent
   * all unnecessary Fast Retransmits
   */
  reset_rtx_timer(1,0);
  output(last_ack_ + 1, TCP_REASON_DUPACK);
  dupwnd_ = numdupacks_;
  return;
}

if (bug_fix_ ) {
  if (bugfix_ts_ && tss[highest_ack_ % tss_size_] == ts_echo_)
    goto reno_action;
}
else if (bugfix_ack_ && cwnd_ > 1 && highest_ack_ -
prev_highest_ack_ <= numdupacks_)
    goto reno_action;

else
    /*
    * The line below, for "bug_fix" true, avoids
    * problems with multiple fast retransmits in one
    * window of data.
    */
    return;

}

reno_action:
    recover_ = maxseq_;
    reset_rtx_timer(1,0);
    if (!lossQuickStart()) {
        trace_event("NEWRENO_FAST_RETX");
        last_cwnd_action_ = CWND_ACTION_DUPACK;
        slowdown(CLOSE_SSTHRESH_HALF|CLOSE_CWND_HALF);

        output(last_ack_ + 1, TCP_REASON_DUPACK);

        dupwnd_ = numdupacks_;
    }
    return;
}

// This function is invoked when TCP-sender receives an ACK

void NewRenoTcpAgent::recv(Packet *pkt, Handler*)
{
    hdr_tcp *tcph = hdr_tcp::access(pkt);
    hdr_ip *hip = HDR_IP(pkt);
    hdr_flags* tepf = hdr_flags::access(pkt);
int valid_ack = 0;

double now;
//static double now1;

/* Use first packet to calculate the RTT --contributed by Allman */

// this module will not be executed if the segment is ALN feedback
if(tcpf->getrq_retransmit( ) == 0){
    printf("inside if\n");
    if(qs_approved_ == 1 && tcpf->seqno() > last_ack_)
        endQuickStart( );
    if(qs_requested_ == 1)    processQuickStart(pkt);
    if(acked_ == 1)
        basertt_ = Scheduler::instance( ).clock( ) - firstsent_;

    /* Estimate ssthresh based on the calculated RTT and the estimated
    bandwidth (using ACKs 2 and 3). */

    else if(acked_ == 2)
        ack2_ = Scheduler::instance( ).clock( );
    else if(acked_ == 3){
        ack3_ = Scheduler::instance( ).clock( );
        //printf("before calculation\n");
        new_ssthresh_ = int(basertt_ * (size_ / (ack3_ - ack2_)))/
        size_);

        if(newreno_changes_ > 0 && new_ssthresh_ < ssthresh_)
            ssthresh_ = new_ssthresh_;
    }
#endif

    ifdef notdef
    if(pkt->type_ != PT_ACK) {
    

fprintf(stderr,
    "ns: configuration error: tcp received non-ack\n");
exit(1);

#endif

if (teph->ts() < lastreset_)
    // Remove packet and do nothing
    Packet::free(pkt);
    return;

++nackpack_;  
ts_peer_ = teph->ts();
if (hdr_flags::access(pkt)->ecn() && ecn_)
    ecn(teph->seqno());
recv_helper(pkt);
recv_frto_helper(pkt);

/*If an ACK is received by the sender, this module checks
 * if the sequence number acknowledged is greater than the largest
 * consecutive ACK, and if the sender is already in ARQ recovery phase
 * it checks if the ACK acknowledges all the segments present in the
 * recovery queue the variables are cleared
 * The segments are acknowledged are
 * removed from the
 * recovery queue and initializes the parameters from head
 */
if (arq_recovery_enable_ && teph->getarq_retransmit() == 0){
    printf("%d arq_retransmit2\n", teph->getarq_retransmit());
    if (teph->seqno() > last_ack_){
        if (arq_recovery_){
            if (is_covered(teph->seqno())){
                arq_recovery_ = 0;
            }
        }
    }
}
now = Scheduler::instance()->clock();
fprintf(stderr,"arq recovered2 %f for %d", now,tcp->seqno());
arq_recovery_seqno_ = -1;
arq_retrans_time_ = -1; dupacks_ = 0;
}
else {fprintf(stderr," dupack for %d\n", arq_recovery_seqno_);
struct info* p2;

   p2 = torecover->head();

   arq_recovery_seqno_ = torecover->getseqno(p2);
arq_retrans_time_ = torecover->gettime(p2);
dupacks_ = 0;
}
/*If the sender is ARQ recovery phase, it retransmits the lost segment */
* reschedules the retransmission timer and if RTT measurement is active for
* segment, clear it as the retransmission is happening
* it enqueues the sequence number present in the ACK
* and the current time in the recovery queue.
*
*/
if(tcpf->getarq_retransmit() == 1 && tcpf->seqno() > last_ack_) {

   double x = recent_rtt_
   printf("rtt %f\n",x);
   now = Scheduler::instance()->clock();
printf("ARQ Recovery transmitting \%d, at \%f due to wireless loss of \%d\n", tcph->seqno(),now,hip->daddr());
tcpf->arq_recover_time() = now;
    // retransmit the lost segment
output(tcph->seqno(),4);
    /* inserts the current time and sequence number present in the
    * feedback in to the queues
    * arq_recovery_list_
    * arq_recovery_time_list_
    */
torecover->enqueue(tcph->seqno(),now);
set_rtx_timer();
    // If RTT is active for this segment, deactivate it which allows
    // the subsequent measurement for any other segment
if(rtt_active_ && rtt_seq_ == tcph->seqno()) rtt_active_ = 0;
struct info *p2;

    arq_recovery_ = 1;
    if (last_ack_ == tcph->seqno() - 1 ) {
        dupacks_ = 0;
arq_recovery_seqno_ = tcph->seqno();
arq_retrans_time_ = tcpf->arq_recover_time();
    }

else {
    p2 = torecover->head();
arq_recovery_seqno_ = torecover->getseqno(p2);

    arq_retrans_time_ = torecover->gettime(p2);
    }
    int d = dupacks_;
if (tcph->seqno() > last_ack_)
{
    if (tcph->seqno() >= recover_
        || (last_cwnd_action_ != CWND_ACTION_DUPACK))
    {
        if (dupwnd_ > 0)
        {
            dupwnd_ = 0;
            if (last_cwnd_action_ == CWND_ACTION_DUPACK)
                last_cwnd_action_ = CWND_ACTION_EXITED;
        }
        if (exit_recovery_fix_)
            int outstanding = maxseq_ - tcph->seqno() + 1;
        if (ssthresh_ < outstanding)
            cwnd_ = ssthresh_;
    }
    else
        cwnd_ = outstanding;
}

firstpartial_ = 0;
printf("%d arq_retransmit2---1\n");
recv_newack_helper(pkt);
printf("%d arq_retransmit2---5\n");
if (last_ack_ == 0 && delay_growth_)
{
    cwnd_ = initial_window();
}
} else {
/* received new ack for a packet sent during Fast
 * Recovery, but sender stays in Fast Recovery
 */
    if (partial_window_deflation_ == 0)
dupwnd_ = 0;
printf("%d arq_retransmit2-2\n")
partialnewack_helper(pkt);
}

} else if(tcph->seqno() == last_ack_)
{
if(hdr_flags: access(pkt)->eln_ && eln_)

tcp_eln(pkt);

return;
}

/* This part checks for duplicate ACKs
* After an ARQ recovery, the segment is considered to be lost only if
* the duplicates arrive after RTT from the retransmission time.
* *
if (arq_recovery_ & & last_ack_ == arq_recovery_seqno_ - 1 ){

now = Scheduler: :instance().clock();
printf("now %f--- previous now2 %f\n", now, now1);

double a2 = arq_retrans_time_;
printf("retransmitted time %f\n", a2);
if (now - arq_retrans_time_ > recent_rtt_ )
{
++dupacks_;

int d = dupacks_;

printf("duplicates %d at %f counting %d\n ",
tcph->seqno(), now, d);
}

now1 = now;
//printf("time %f dupack %d", now, d);

int a = arq_recovery_seqno_;
double a1 = arq_retrans_time_;
//double rtt = t_rtxcur_;
double rtt = recent_rtt_;
else {++dupacks_; int a1 = dupacks_; int a = last_ack_;  
    printf("duplicate identified %d and count = %d", a,a1);}

if (dupacks_ == numdupacks_) {  
    //printf("inside upacks = numdupacks\n");
    if (arq_recovery_ && last_ack_ == arq_recovery_seqno_-1){  
        //printf("inside upacks = numdupacks4\n");
torecover->remove(arq_recovery_seqno_);  
        //printf("inside upacks = numdupacks2\n");
struct info* p2;
p2 = torecover->head();
if(p2 != NULL){  
arq_recovery_seqno_ = torecover->getseqno(p2);
arq_retrans_time_ = torecover->gettime(p2);
}
else {arq_retrans_time_ =-1;
arq_recovery_seqno_ =-1;}
}

//printf("inside upacks = numdupacks3\n");
dupack_action();

if (!exitFastRetrans_)  
dupwnd_ = numdupacks_;  
}
else if (dupacks_ > numdupacks_ & & (!exitFastRetrans_  
    || last_cwnd_action_ == CWND_ACTION_DUPACK)) {  
    trace_event("NEWRENO_FAST_RECOVERY");  
    ++dupwnd_; // fast recovery

    /* For every two duplicate ACKs we receive (in the  
    * "fast retransmit phase"), send one entirely new  
    * data packet "to keep the flywheel going". --Allman  
    */
if (newreno_changes_ > 0 && (dupacks_ % 2) == 1)
    output (t_seqno_++,0);
else if (dupacks_ < numdupacks_ && singledup_ ) {
    send_one();
}

if (tcph->seqno() >= last_ack_)
    // Check if ACK is valid. Suggestion by Mark Allman
    valid_ack = 1;
Packet::free(pkt);
#endif

if (trace_)
    plot();
#endif

/*

  * Try to send more data

  */

if (valid_ack || aggressive_maxburst_)
    if (dupacks_ == 0)
        /*
         * Maxburst is really only needed for the first
         * window of data on exiting Fast Recovery.
         */
        send_much(0, 0, maxburst_);
    else if (dupacks_ > numdupacks_ - 1 && newreno_changes_ == 0)
        send_much(0, 0, 2);
        printf("%d arq_retransmit2--6\n");
}

/* This function removes the sequence numbers that are within
* 'seq'
*/

int NewRenoTcpAgent::is_covered(int seq)
{
    torecover->resetIterator();
    struct info* p1 = torecover->getNext();

    while (p1 != (struct info*)0 && seq >= p1->a_seq_no)
    {  //fprintf(stderr,"inside is covered\n");
        torecover->remove(p1->a_seq_no);
        p1 = torecover->getNext();
        //fprintf(stderr,"coming out of covered\n");
    }
    //fprintf(stderr,"out of while\n");
    if (p1 == (struct info*)0) {
        return 1;
    } else return 0;
}

/* This function checks if a sequence number is present in the 
* arq_recover_list. If not present the function returns 1. 
* */
int NewRenoTcpAgent::is_available(int seq)
{
    torecover->resetIterator();
    struct info * p1 = torecover->getNext();
    while (p1 != NULL && seq != p1->a_seq_no)
    {
        p1 = torecover->getNext();
    }
    if (!p1) return 1;
    else return 0;
}