Bug Fixes on the IEEE 802.11 DCF module of the Network Simulator ns-2.28

Felix Schmidt-Eisenlohr, Jon Letamendia-Murua,
Marc Torrent-Moreno, Hannes Hartenstein

Institute of Telematics, University of Karlsruhe, Germany
fschmidt@tm.uni-karlsruhe.de, jonle@web.de,
torrent@tm.uni-karlsruhe.de, hartenstein@rz.uni-karlsruhe.de

Technical Report 2006-1
ISSN 1432-7864
Dept. of Computer Science
Universität Karlsruhe (TH)

1 Introduction

The Network Simulator 2 (ns-2) [1] is largely the most used simulator in the Ad Hoc research community [2]. However, the 802.11 DCF module implemented in the default distribution of ns-2 presents some bugs, i.e., discordances with the IEEE 802.11 Standard specifications [3].

We present in this Technical Report the result of an extensive analysis of both the IEEE 802.11 DCF specification and the ns-2 module, realized with the support of [4]. We first describe the discordances found with respect to the different DCF's procedures. Second, we describe the different behavior corresponding to the physical layer capture model that current wireless interfaces present [5]. Finally we provide in the Appendix all source code modified in the different ns-2.28 files. All modified source code files can be found for download at `http://dsn.tm.uni-karlsruhe.de/ns-2.28-DCF-PHY-UKA.php`.

2 Bug fixes

Each of the following subsections describes one or several bugs concerning a certain issue of the MAC functionality. Within these sections, we describe the correspondent behavior according to the IEEE 802.11 specifications, the implementation provided by the default distribution of ns-2.28 and our proposition to fix the non-matching behavior.

In order to avoid confusion and improve readability in this document the term *transmission* will only refer to a transmission of the node being described. Transmissions from other stations will be referred as *receptions*.

2.1 Erroneous packet reception

The IEEE 802.11 specification contains several lines concerning the handling of erroneous packet reception and the usage of *Extended Inter Frame Space (EIFS)*. Section 9.2.3.4 states:

> "The EIFS shall be used by the DCF whenever the PHY has indicated to the MAC that a frame transmission was begun that did not result in the correct reception of a complete MAC frame with a correct FCS value. The duration of an EIFS is defined in 9.2.10. The EIFS interval shall begin following indication by the PHY that the medium is idle after detection of the erroneous frame, without regard to the virtual carrier-sense mechanism. The EIFS is defined to provide enough time for another STA to acknowledge what was, to this STA, an incorrectly received frame before this STA commences transmission. Reception of an error-free frame during the EIFS resynchronizes the STA to the actual busy/idle state of the medium,

so the EIFS is terminated and normal medium access (using DIFS and, if necessary, backoff) continues following reception of that frame.”

Sections 9.2.4, 9.2.5.1 and 9.2.5.2 state that EIFS will precede a packet transmission instead of DIFS when following the detection of a frame that was not correctly received. We remark the following points:

- EIFS must be started when the medium is detected as idle after every frame that is not received correctly.

- Reception of a correct frame must interrupt the EIFS period and the radio interface must resynchronize to the actual busy/idle state.

The default distribution of ns-2.28 manages erroneous packet reception and EIFS as follows: every time a station receives either a packet with errors or a packet that collides with another packet, the network allocation vector (NAV) is set with a duration of an EIFS period. The station virtually determines the medium as busy and in consequence is not allowed to send any frame during this period of time. Using the NAV for waiting the EIFS period mixes two different concepts, virtual carrier sensing and inter frame spaces. The ns-2.28 implementation does not fulfill the IEEE specifications because a mechanism to reset EIFS after receiving an error-free frame is missing. Also, in case that a backoff procedure has to be started, the time before a backoff restarts is too long, i.e., EIFS + DIFS, instead of EIFS. Our implementation solves the problems of the default distribution as follows:

- A flag variable `last_packet_correct_` indicates the result of the last packet reception. If the station receives a packet with errors the flag is set to *false*. The flag is set to *true* whenever the station receives an error-free packet, and at the latest, after waiting an interval of length EIFS even if no packet was received during this time span.

- In the case *i)* the MAC module gets, from a higher layer, a data packet to be transmitted during the reception of another packet, and *ii)* when having to resume a backoff period after the reception of a packet, the flag `last_packet_correct_` will be read. If the medium is sensed idle and `last_packet_correct_` is *true* at the end of the packet reception the DIFS period is used; in case `last_packet_correct_` is *false* EIFS is utilized.

The bugfix included changes in the following files: `mac/mac-802_11.cc`, `mac/mac-802_11.h`, `mac/mac-timers.cc` and `mac/mac-timers.h`. All changed code is enclosed in a block starting with “`// BUGFIX UKA: EIFS`” and ending with “`// BUGFIX UKA END: EIFS`” and is appropriately commented. The code of the modified methods is listed in Appendix A.

The affected methods in the file `mac/mac-802_11.cc` are:

- `checkBackoffTimer()`, `send()`: select the appropriate inter-frame space depending on the flag variable `last_packet_correct_`.

- `capture()`, `collision()`: start an EIFS period instead of the NAV timer.

- `deferHandler()`: finish an EIFS period after its expiration.

- `RetransmitRTS()`, `RetransmitDATA()`: usage of the new method `StartRetransmitBackoff()`, see below.

- `recv()`, `recv_timer()`: set the flag variable `last_packet_correct_` appropriately.

The following methods were added to the code:

- `setEIFS()`: method called at the start of an EIFS period to start the deferring time.

- `resetEIFS()`: method called at the moment a new packet is detected to manage the ongoing timers, if any.

- `startRetransmitBackoff()`: handles the different cases when scheduling a retransmission

In the file `mac/mac-timers.cc`:

- `BackoffTimer::start()`: method arguments were extended, causing adaption of the method calls at several places.

2.2 Packet arrival during a transmission

According to IEEE 802.11 specifications, wireless chipsets do not support the transmission and reception of packets at the same time, but have to switch between these two states. Consequently, a packet that arrives at a station during a transmission is not sensed by the station because the radio interface is in transmission state. In the ns-2 simulator, however, packets that arrive during a transmission are marked all erroneous at the moment they arrive, i.e., when the first symbol arrives at the interface, and discarded at the end, i.e., when the last symbol has arrived, resulting in the start of an EIFS period. In our implementation the packet is marked with a special label `TX_RX_ERROR`. After having received the last symbol of the packet, it is discarded and no EIFS period follows. Note that a packet arriving during a transmission can not be indicated by the physical layer as the beginning of a frame reception (Section 9.2.3.4 of the IEEE 802.11 specifications). Therefore, after the transmission ended, the reception of a packet can still not be indicated and consequently, no EIFS period follows.

The bugfix included changes in the files `mac/mac-802_11.cc` and `mac/mac-802_11.h`. All changed code is enclosed in a block starting with "`// BUGFIX UKA: TxRxError`" and ending with "`// BUGFIX UKA END: TxRxError`" and is appropriately commented. The code of the modified methods is listed in Appendix A.

The following methods are affected:

- `transmit()`, `recv()`: if there is a packet transmission and a packet reception at the same time, the received packet is marked with `TX_RX_ERROR = 1`.
- `recv_timer()`: discard packets that arrived during a transmission.

2.3 Packet transmission

Several bugs concern the correct handling of packets that should be sent by a station. Depending on the packet type there exist different issues explained in the following lines. The bugfix included changes in the file `mac/mac-802_11.cc`. All changed code is enclosed in a block starting with "`// BUGFIX UKA: transmission`" and ending with "`// BUGFIX UKA END: transmission`" and is appropriately commented. The code of the modified methods is listed in Appendix A.

Transmission of a DATA or an RTS packet: The transmission of DATA and RTS packets is controlled by the MAC layer using a set of timers and the physical and virtual indicators of the medium's busy/idle state. If the medium is sensed idle and neither backoff nor defer timer is running at the moment the MAC layer gets a packet from a higher layer, the RTS (or DATA) packet can be transmitted after an idle period of DIFS/EIFS (see section 9.2.5.1 of the IEEE 802.11 specification). However, if the medium is already busy or becomes busy during the DIFS/EIFS period, a backoff procedure has to follow (see section 9.2.5.2 of the IEEE 802.11 specification). The number of backoff intervals (slots) that the medium has to be free is determined with the help of the Contention Window (CW), that is increased on every retransmission attempt (see section 9.2.4 of the IEEE 802.11 specification).

However, the standard distribution of ns-2.28 does not follow these rules after the MAC layer gets a packet from a higher layer: *i)* in the method `send()`, a backoff period is started in case that the medium is free and none of the timers is running; *ii)* the Contention Window is increased whenever the medium becomes busy during DIFS/EIFS deferring period (see methods `check_pktRTS()` and `check_pktTx()`); *iii)* the backoff procedure is not initialized with the appropriate timer (`mhBackoff_`) in the method `tx_resume()`.

The bugs can be fixed by deferring for a period of length DIFS (without a backoff period) in the first case, not increasing the Contention Window in the second case and by using the appropriate timer in the last case.

Transmission of an acknowledgment packet (ACK): If a station receives a DATA packet it must acknowledge the reception by transmitting an acknowledgment packet (ACK) after waiting a time period of SIFS. The ACK packet should be transmitted in every case, according to section 9.2.8 of the specifications:

> "After a successful reception of a frame requiring acknowledgment, transmission of the ACK frame shall commence after a SIFS period, without regard to the busy/idle state of the medium."

In the default distribution, this did not happen in case that the station was waiting for another frame, i.e., CTS, DATA or ACK. The improved implementation changes this behavior; in every case, the ACK is sent after SIFS, see method `recvDATA()`. The pending timeout of the packet that the station is waiting for is handled by calling the method `sendHandler()`.

Transmission of a broadcast packet: According to section 9.2.4 of the 802.11 specification, both the short and the long retry counters should be reset if a broadcast packet was sent. This was not done in the default distribution and is now added in the method `RetransmitDATA()`.

Retransmission of a DATA or RTS packet: Retransmission must occur if the expected acknowledgment (or the CTS in case of an RTS packet) does not arrive before its expected time. This is described in section 9.2.5.7 of the 802.11 specification. Section 9.2.5.2 states:

> "The backoff procedure shall also be invoked when a transmitting STA infers a failed transmission as defined in 9.2.5.7 or 9.2.8."

However, the default distribution initiates the backoff procedure only in the case that the retry limit is not reached. This error is corrected in our implementation in the methods `RetransmitRTS()`, `RetransmitDATA()` and the new method `StartRetransmitBackoff()`.

2.4 Expiration of the network allocation vector (NAV)

`checkBackoffTimer()` is the method that is called when the state of the backoff process may need to be changed, i.e., it is responsible to stop, pause or resume it depending on the station's current status. However, in the default implementation it exists another method, `navHandler()`, that manages the backoff process in its own, i.e., without calling `checkBackoffTimer()`. Due to the changes in the EIFS timer (see Section 2.1) the code inside `navHandler()` was not consistent any longer. This incorrect and duplicate piece of code is fixed, now `navHandler()` handles the backoff timer calling `checkBackoffTimer()`.

The bugfix included changes in the file `mac/mac-802_11.cc`. All changed code is enclosed in a block starting with "`// BUGFIX UKA: NAV`" and ending with "`// BUGFIX UKA END: NAV`" and is appropriately commented. The code of the modified methods is listed in Appendix A.

2.5 Memory leak at the reception of a MAC control packet

DATA packets given to higher layers are removed from the memory when they are completely handled. MAC control packets (RTS, CTS and ACK packets), however, are not given to higher layers and the memory that they have allocated has to be freed by the MAC layer itself. This was not done in the default distribution, so it was added in the method `recv_timer()`.

The bugfix included changes in the file `mac/mac-802_11.cc`. All changed code is enclosed in a block starting with "`// BUGFIX UKA: Memory`" and ending with "`// BUGFIX UKA END: Memory`" and is appropriately commented. The code of the modified methods is listed in Appendix A.

2.6 Capture effect

The default distribution has implemented a physical layer capability referred to as *capture* effect: if a node is receiving a data packet (P_1) and during its reception another packet (P_2) reaches the station, the radio interface is able to continue decoding successfully the first packet if its reception power ($pow(P_1)$) is stronger than the reception power of the second packet ($pow(P_2)$) by at least a factor called *capture threshold* (C_{PThr}), i.e., $pow(P_1) \geq C_{PThr} \cdot pow(P_2)$.

However, newer wireless chipsets allow further capturing [5]: if a radio interface is synchronized to a packet (P_1) and during its duration a second packet (P_2) arrives, the station is able to resynchronize and decode this second packet under the following two conditions:

- The receiving power of the second packet is at least higher by the factor *capture threshold* than the receiving power of the first packet, i.e., $pow(P_2) \geq C_{PThr} \cdot pow(P_1)$.
- The second packet does not reach the receiver between $4\mu s$ and $10\mu s$ after the detection of the first packet in order to properly resynchronize.

Using the old chipset only the first of both packets could be 'captured'. The extended chipset capability can be activated and deactivated setting a variable in the tcl simulation script:

- `Mac/802_11 set newchipset_ false`: deactivation of new chipset feature (default).
- `Mac/802_11 set newchipset_ true`: activation of new chipset feature.

The changes in the code affect the methods `capture()` and `recv()` in the file `mac/mac-802_11.cc`. Also, the file `mac/mac-802_11.h` is modified and defines the variable to activate the new feature. The default value of the tcl variable `newchipset_` was added to the file `tcl/lib/ns-default.tcl`. Finally, we trace a packet discard as the result of the capture effect (to differentiate it from collisions) in the trace files, resulting in an additional state `CAP` defined in the file `trace/cmu-trace.h`. All related passages are enclosed between comments of the form "`// BUGFIX UKA: capture`" and "`// BUGFIX UKA END: capture`". The code of the modified methods is listed in Appendix A.

References

[1] "Network Simulator ns-2," http://www.isi.edu/nsnam/ns/.

[2] S. Kurkowski, T. Camp, and M. Colagrosso, "MANET Simulation Studies: the Incredibles," in *SIGMOBILE Mobile Computing and Communications Review (MC2R)*, vol. 9, no. 4. New York, NY, USA: ACM Press, 2005, pp. 50–61.

[3] ANSI/IEEE, "IEEE Std. 802.11, 1999 Edition, Part11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications," http://www.ieee802.org/11/, 1999.

[4] M. S. Gast, *802.11 Wireless Networks*, 2nd ed. O'Reilly, 2005.

[5] A. Kochut, A. Vasan, A. Shankar, and A. Agrawala, "Sniffing out the correct Physical Layer Capture model in 802.11b," in *Proceedings of 12th IEEE International Conference on Network Protocols (ICNP 2004)*, October 2004.

A Source code

In the following, all methods affected by the improvements are listed. The original (old) implementation and the improved (new) implementation are listed.

A.1 mac/mac-802_11.cc

```
inline void
Mac802_11::checkBackoffTimer()
{
    if(is_idle() && mhBackoff_.paused())
        // BUGFIX UKA: EIFS
        // When the channel becomes free again the station will have
        // to back off with DIFS or EIFS period depending on the
        // last received packet

        // old implementation
        // mhBackoff_.resume(phymib_.getDIFS());

        // new implementation
        if (last_packet_correct_ == true)
            mhBackoff_.resume(phymib_.getDIFS());
        else
            mhBackoff_.resume(phymib_.getEIFS());
        // BUGFIX UKA END: EIFS

    if(! is_idle() && mhBackoff_.busy() && ! mhBackoff_.paused())
        mhBackoff_.pause();
}

inline void
Mac802_11::transmit(Packet *p, double timeout)
{
    tx_active_ = 1;

    if (EOTtarget_) {
        assert (eotPacket_ == NULL);
        eotPacket_ = p->copy();
    }

    /*
     * If I'm transmitting without doing CS, such as when
     * sending an ACK, any incoming packet will be "missed"
     * and hence, must be discarded.
     */
    if(rx_state_ != MAC_IDLE) {
        struct hdr_mac802_11 *dh = HDR_MAC802_11(p);
        assert(dh->dh_fc.fc_type == MAC_Type_Control);
        assert(dh->dh_fc.fc_subtype == MAC_Subtype_ACK);
        assert(pktRx_);
        struct hdr_cmn *ch = HDR_CMN(pktRx_);

        // BUGFIX UKA: TxRxError
        // If a station transmits it cannot sense and thus not receive
        // packets at the same time physically. In the original simulator
        // however such a packet is "received", marked errornous, discarded
        // and an EIFS period follows.
        // This however should not happen, because the packet is not sensed.
        // Therefore the packet that is received is marked special and handled
        // correctly after complete "reception" (handle it as "never sensed")

        // old implementation
        //ch->error() = 1;        /* force packet discard */

        // new implementation
        ch->error() = TX_RX_ERROR;
        // BUGFIX UKA END: TxRxError
    }

    /*
     * pass the packet on the "interface" which will in turn
     * place the packet on the channel.
     *
     * NOTE: a handler is passed along so that the Network
     *       Interface can distinguish between incoming and
     *       outgoing packets.
     */
    downtarget_->recv(p->copy(), this);
    mhSend_.start(timeout);
    mhIF_.start(txtime(p));
}

/* ======================================================================
   Phy MIB Class Functions
   ====================================================================== */

PHY_MIB::PHY_MIB(Mac802_11 *parent)
{
    /*
     * Bind the phy mib objects.  Note that these will be bound
     * to Mac/802_11 variables
     */

    parent->bind("CWMin_", &CWMin);
    parent->bind("CWMax_", &CWMax);
    parent->bind("SlotTime_", &SlotTime);
    parent->bind("SIFS_", &SIFSTime);
    parent->bind("PreambleLength_", &PreambleLength);
    parent->bind("PLCPHeaderLength_", &PLCPHeaderLength);
    parent->bind_bw("PLCPDataRate_", &PLCPDataRate);

    // BUGFIX UKA: capture
    // bind variable
    // Set newchipset_ to false for classical chipset behavior
    // Set to true for improved capture support.
    parent->bind_bool("newchipset_", &newchipset);
    // BUGFIX UKA END: capture
}


/* ======================================================================
   Mac Class Functions
   ====================================================================== */
Mac802_11::Mac802_11() :
    Mac(), phymib_(this), macmib_(this), mhIF_(this), mhNav_(this),
    mhRecv_(this), mhSend_(this),
    mhDefer_(this), mhBackoff_(this)
{

    nav_ = 0.0;
    tx_state_ = rx_state_ = MAC_IDLE;
    tx_active_ = 0;
    eotPacket_ = NULL;
    pktRTS_ = 0;
    pktCTRL_ = 0;
    cw_ = phymib_.getCWMin();
    ssrc_ = slrc_ = 0;
    // Added by Sushmita
    et_ = new EventTrace();

    sta_seqno_ = 1;
    cache_ = 0;
    cache_node_count_ = 0;

    // BUGFIX UKA: capture
    // saves the point of time of the start of the last packet reception
    time_start_pktRx_= 0.0;
    // BUGFIX UKA END: capture

    // BUGFIX UKA: EIFS
    // On initialization the last received packet is assumed as correct.
    last_packet_correct_ = true;
    // BUGFIX UKA END: EIFS

    // chk if basic/data rates are set
    // otherwise use bandwidth_ as default;

    Tcl& tcl = Tcl::instance();
    tcl.evalf("Mac/802_11 set basicRate_");
    if (strcmp(tcl.result(), "0") != 0)
        bind_bw("basicRate_", &basicRate_);
    else
        basicRate_ = bandwidth_;

    tcl.evalf("Mac/802_11 set dataRate_");
    if (strcmp(tcl.result(), "0") != 0)
        bind_bw("dataRate_", &dataRate_);
    else
        dataRate_ = bandwidth_;

    EOTtarget_ = 0;
    bss_id_ = IBSS_ID;
    //printf("bssid in constructor %d\n",bss_id_);
}

// BUGFIX UKA: EIFS
// Two new functions for setting and resetting EIFS state

// Set_eifs: This method is called after an errornous packet
// reception, It sets last_packet_correct to false and
// starts the defer timer if backoff is not already running
// (in that case, the backoff timer cares about EIFS on
// resume (see checkbackofftimer()).

inline void
Mac802_11::set_eifs() {
    last_packet_correct_ = false;
    if (mhBackoff_.busy() == false)  {
        mhDefer_.start(phymib_.getEIFS());
    }

}

// reset_eifs: If the station starts receiving a packet, reset_eifs is
// called. It checks if the station is in the EIFS period and if this is
// done by defer timer. If this is the case the defer timer is stopped. If
// there are packets to send then the station will initialize a backoff
// period (if not already running), that is directly paused until the
// medium is idle again.

inline void
Mac802_11::reset_eifs()
{
    if (last_packet_correct_ == false && mhDefer_.busy() == true) {
        mhDefer_.stop();
        if (mhBackoff_.busy() == false && (pktRTS_ || pktTx_)) {
            mhBackoff_.start(cw_, is_idle(), phymib_.getDIFS());

        }
    }
}

// BUGFIX UKA END: EIFS

void
Mac802_11::capture(Packet *p)
{
    // BUGFIX UKA: capture, EIFS
    // Changes concerning capture effect:
    // - Packets that are discarded because of the capture effect are
    //   mentioned in  the trace
    // - A new version of the chipset implementation that handles an
    //   "extended capture effect" is implemented
    // Changes concerning EIFS handling:
    // - NAV is not used for EIFS handling anymore; replace by mechnism
    //   using last_packet_correct_ variable and defer/backoff timers

    // old implementation
    /*
     * Update the NAV so that this does not screw
     * up carrier sense.
     */
    // set_nav(usec(phymib_.getEIFS() + txtime(p)));
    // Packet::free(p);

    // new implementation

    last_packet_correct_ = false;

    if (phymib_.get_newchipset() == false) {
        // handle the classical capture effect (new chipset feature is not used)
        discard(p, DROP_MAC_CAPTURE);
    } else {
    // handle capture effect if (new chipset feature used)

        if (pktRx_->txinfo_.RxPr > p->txinfo_.RxPr){
            // RxPr first packet > RxPr second packet
            // (power difference a priori big enough, otherwise capture is not called)
            // => continue receive packet 1, discard packet 2
            discard(p,DROP_MAC_CAPTURE);
        } else {
            // RxPr first packet < RxPr second packet
            // (power difference a priori big enough, otherwise capture is not called)
            // => stop receive packet 1 and discard, receive packet 2 from now on
            mhRecv_.stop();                     // receive timer for packet 1 stopped
            mhRecv_.start(txtime(p));           // start receive timer for packet 2
            discard(pktRx_, DROP_MAC_CAPTURE);  // discard packet 1
            pktRx_ = p;                         // make packet 2 the one that is received now
        }
    }
    // BUGFIX UKA END: capture, EIFS
}

void
Mac802_11::collision(Packet *p)
{
    switch(rx_state_) {
    case MAC_RECV:
        setRxState(MAC_COLL);
        /* fall through */
    case MAC_COLL:
        assert(pktRx_);
        assert(mhRecv_.busy());
        /*
         *  Since a collision has occurred, figure out
         *  which packet that caused the collision will
         *  "last" the longest.  Make this packet,
         *  pktRx_ and reset the Recv Timer if necessary.
         */

        // BUGFIX UKA: EIFS
        // A collision implies reception of an errornous packet
        // set the last_packet_correct_ variable to false
         last_packet_correct_ = false;
        // BUGFIX UKA END: EIFS

        if(txtime(p) > mhRecv_.expire()) {
            mhRecv_.stop();
            discard(pktRx_, DROP_MAC_COLLISION);
            pktRx_ = p;
            mhRecv_.start(txtime(pktRx_));
        }
        else {
            discard(p, DROP_MAC_COLLISION);
        }
        break;
    default:
        assert(0);
    }
}

void
Mac802_11::tx_resume()
{
    double rTime;
    assert(mhSend_.busy() == 0);
    assert(mhDefer_.busy() == 0);

    if(pktCTRL_) {
        /*
         *  Need to send a CTS or ACK.
         */
        mhDefer_.start(phymib_.getSIFS());
    } else if(pktRTS_) {
        if (mhBackoff_.busy() == 0) {
            // BUGFIX UKA: transmission
            //
            // Do backoff period using the (interruptable) backoff timer, not
            // the (non-interruptable) defer timer

            // old implementation

            //rTime = (Random::random() % cw_) * phymib_.getSlotTime();
            //mhDefer_.start( phymib_.getDIFS() + rTime);

            // new implementation

            mhBackoff_.start(cw_, is_idle(), phymib_.getDIFS());
            //BUGFIX UKA END: transmission
        }
    } else if(pktTx_) {
        if (mhBackoff_.busy() == 0) {
            hdr_cmn *ch = HDR_CMN(pktTx_);
            struct hdr_mac802_11 *mh = HDR_MAC802_11(pktTx_);

            if ((u_int32_t) ch->size() < macmib_.getRTSThreshold()
                || (u_int32_t) ETHER_ADDR(mh->dh_ra) == MAC_BROADCAST) {
                // BUGFIX UKA: transmission
                //
                // Do backoff period using the (interruptable) backoff timer,
                // not the (non-interruptable) defer timer

                // old implementation

                // rTime = (Random::random() % cw_)
                //    * phymib_.getSlotTime();
                // mhDefer_.start(phymib_.getDIFS() + rTime);

                // new implementation

                mhBackoff_.start(cw_, is_idle(), phymib_.getDIFS());
                // BUGFIX UKA END: transmission
                       } else {
                mhDefer_.start(phymib_.getSIFS());
                        }
        }
    } else if(callback_) {
        Handler *h = callback_;
        callback_ = 0;
        h->handle((Event*) 0);
    }
    setTxState(MAC_IDLE);
}

void
Mac802_11::deferHandler()
{

    // BUGFIX UKA: EIFS

    // Defer timer is also used for EIFS handling now. This method is called at
    // the end of a complete EIFS. The assertion needs to be extended and the
    // last_packet_correct_ variable has to be reset.

    // old implementation

    // assert(pktCTRL_ || pktRTS_ || pktTx_);

    // new implementation

    assert(pktCTRL_ || pktRTS_ || pktTx_ || last_packet_correct_ == false);
    last_packet_correct_ = true;
    // BUGFIX UKA END: EIFS

     if(check_pktCTRL() == 0)
        return;
    assert(mhBackoff_.busy() == 0);
    if(check_pktRTS() == 0)
        return;
    if(check_pktTx() == 0)
        return;
}

void
Mac802_11::navHandler()
{
    // BUGFIX UKA: NAV
    // If NAV finishes, paused backoff timers have to be resumed.
    // Use the appropriate method instead of an individual solution here.

    // old implementation
    // if(is_idle() && mhBackoff_.paused())
    // mhBackoff_.resume(phymib_.getDIFS());

    // new implementation

    checkBackoffTimer();
    // BUGFIX UKA END: NAV
}

int
Mac802_11::check_pktRTS()
{
    struct hdr_mac802_11 *mh;
    double timeout;

    assert(mhBackoff_.busy() == 0);

    if(pktRTS_ == 0)
         return -1;
    mh = HDR_MAC802_11(pktRTS_);

     switch(mh->dh_fc.fc_subtype) {
    case MAC_Subtype_RTS:
        if(! is_idle()) {
            // BUGFIX UKA: transmission
            // The contention window should only be increased before retransmit
            // (see Standard spec. section 9.2.4)

            // old implementation

            // inc_cw();
            // BUGFIX UKA END: transmission

            // BUGFIX UKA: EIFS
            // changed method definition for backoff start causes changed call

            // old implementation

            // mhBackoff_.start(cw_, is_idle());

            // new implementation

            mhBackoff_.start(cw_, is_idle(), phymib_.getDIFS());
            // BUGFIX UKA END: EIFS
            return 0;
        }
        setTxState(MAC_RTS);
        timeout = txtime(phymib_.getRTSlen(), basicRate_)
            + DSSS_MaxPropagationDelay                      // XXX
            + phymib_.getSIFS()
            + txtime(phymib_.getCTSlen(), basicRate_)
            + DSSS_MaxPropagationDelay;
        break;
    default:
        fprintf(stderr, "check_pktRTS:Invalid MAC Control subtype\n");
        exit(1);
    }
    transmit(pktRTS_, timeout);


    return 0;
}

int
Mac802_11::check_pktTx()
{
    struct hdr_mac802_11 *mh;
    double timeout;

    assert(mhBackoff_.busy() == 0);

    if(pktTx_ == 0)
        return -1;

    mh = HDR_MAC802_11(pktTx_);

    switch(mh->dh_fc.fc_subtype) {
    case MAC_Subtype_Data:
        if(! is_idle()) {
            sendRTS(ETHER_ADDR(mh->dh_ra));
            // BUGFIX UKA: transmission
            // The contention window should only be increased before retransmit
            // (see Standard spec. section 9.2.4)

            // old implementation

            // inc_cw();
            // BUGFIX UKA END: transmission

            // BUGFIX UKA: EIFS
            // changed method definition for backoff start causes changed call

            // old implementation

            // mhBackoff_.start(cw_, is_idle());

            // new implementation

            mhBackoff_.start(cw_, is_idle(), phymib_.getDIFS());
            // BUGFIX UKA END: EIFS

            return 0;
        }
        setTxState(MAC_SEND);
        if((u_int32_t)ETHER_ADDR(mh->dh_ra) != MAC_BROADCAST)
                        timeout = txtime(pktTx_)
                                + DSSS_MaxPropagationDelay              // XXX
                               + phymib_.getSIFS()
                               + txtime(phymib_.getACKlen(), basicRate_)
                               + DSSS_MaxPropagationDelay;             // XXX
        else
            timeout = txtime(pktTx_);
        break;
    default:
        fprintf(stderr, "check_pktTx:Invalid MAC Control subtype\n");
        exit(1);
    }
    transmit(pktTx_, timeout);
    return 0;
}

// BUGFIX UKA: EIFS
//
// The new implementation of EIFS causes a more complicated retransmission
// handling. Depending on the expiration time of EIFS backoff timers have to
// be initialized with different waiting times. Since this functionality
// is needed at severeal places in the RetransmitRTS and RetransmitDATA
// methods, it is expoerted to an own method, StartRetransmitBackoff.

void
Mac802_11::StartRetransmitBackoff()
{
    // Set tx state to idle first so that the correct waiting time is used.
    // This is VERY dirty, but if the medium is not idle here, we would never start
    // e.g. with expire time of defer handler, but it would always be paused directly
    // and on resume, DIFS or EIFS is chosen.
    //
    // This call does not change anything, because after leaving this method, we
    // leave RestransmitRTS/DATA, and then, this call comes in send_timer in every
    // case, where tx state would be set to idle.
    setTxState(MAC_IDLE);

    if (last_packet_correct_ == false && mhDefer_.busy()) {
        // defer is running because of EIFS => stop defering, start backoff
        if (mhDefer_.expire() < phymib_.getDIFS()) {
            // time until defer expires is shorter than DIFS => start BO with DIFS
            mhDefer_.stop();
            mhBackoff_.start(cw_, is_idle(), phymib_.getDIFS());
        } else {
            // time until defer expires is greater than DIFS => wait rest of defer
            // (complete EIFS) and do backoff slots then
            mhBackoff_.start(cw_, is_idle(), mhDefer_.expire());
            mhDefer_.stop();
        }
     } else {
        // no defer running; just start backoff
        if (mhBackoff_.busy() == false) {
            mhBackoff_.start(cw_, is_idle(), phymib_.getDIFS());
        }
    }
}

// BUGFIX UKA END:EIFS


void
Mac802_11::RetransmitRTS()
{
    assert(pktTx_);
    assert(pktRTS_);
    assert(mhBackoff_.busy() == 0);
    macmib_.RTSFailureCount++;
    ssrc_ += 1;             // STA Short Retry Count

    if(ssrc_ >= macmib_.getShortRetryLimit()) {
        discard(pktRTS_, DROP_MAC_RETRY_COUNT_EXCEEDED); pktRTS_ = 0;
        /* tell the callback the send operation failed
           before discarding the packet */
        hdr_cmn *ch = HDR_CMN(pktTx_);
        if (ch->xmit_failure_) {
                        /*
                         *  Need to remove the MAC header so that
                         *  re-cycled packets don't keep getting
                         *  bigger.
                         */
            ch->size() -= phymib_.getHdrLen11();
                        ch->xmit_reason_ = XMIT_REASON_RTS;
                        ch->xmit_failure_(pktTx_->copy(),
                                          ch->xmit_failure_data_);
                }
        discard(pktTx_, DROP_MAC_RETRY_COUNT_EXCEEDED);
        pktTx_ = 0;
        ssrc_ = 0;
        rst_cw();
    } else {
        struct rts_frame *rf;
        rf = (struct rts_frame*)pktRTS_->access(hdr_mac::offset_);
        rf->rf_fc.fc_retry = 1;

        inc_cw();
        // BUGFIX UKA: EIFS
        // call StartRetransmitBackoff instead of directly starting backoff
        // This is now done at the end of this method.

        // old implementation

        // mhBackoff_.start(cw_, is_idle());
        // BUGFIX UKA END: EIFS
    }

    // BUGFIX UKA: transmission, EIFS
    // transmission: backoff is started here, because it has to be done in every
    // case and not only in case of not reaching the limit.
    // EIFS: backoff is not started directly but uses StartRetransmitBackoff
    // method.

    // new implementation

    StartRetransmitBackoff();
    // BUGFIX UKA END: transmission, EIFS

}

void
Mac802_11::RetransmitDATA()
{
    struct hdr_cmn *ch;
    struct hdr_mac802_11 *mh;
    u_int32_t *rcount, thresh;

    // BUGFIX UKA: EIFS
    // This assertion is not always valid, the BO timer might already be
    // running after an EIFS period

    // old implementation

    // assert(mhBackoff_.busy() == 0);
    // BUGFIX UKA END: EIFS

    assert(pktTx_);
    assert(pktRTS_ == 0);

    ch = HDR_CMN(pktTx_);
    mh = HDR_MAC802_11(pktTx_);

    /*
     *  Broadcast packets don't get ACKed and therefore
     *  are never retransmitted.
     */
    if((u_int32_t)ETHER_ADDR(mh->dh_ra) == MAC_BROADCAST) {
        Packet::free(pktTx_);
        pktTx_ = 0;

        // BUGFIX UKA: transmission
        // After sending a packet with a group address, both the short and
        // long retry counter should be reset (see Standard 9.2.4)
        ssrc_ = 0;
        slrc_ = 0;
        // BUGFIX UKA END: transmission
        /*
         * Backoff at end of TX.
         */
        rst_cw();

        // BUGFIX UKA: EIFS
        // Call StartRetransmitBackoff instead of directly starting backoff

        // old implementation

        // mhBackoff_.start(cw_, is_idle());

        // new implementation

        StartRetransmitBackoff();
        // BUGFIX UKA END: EIFS

        return;
    }

    macmib_.ACKFailureCount++;

    if((u_int32_t) ch->size() <= macmib_.getRTSThreshold()) {
                rcount = &ssrc_;
               thresh = macmib_.getShortRetryLimit();
        } else {
                rcount = &slrc_;
               thresh = macmib_.getLongRetryLimit();
        }

    (*rcount)++;

    if(*rcount >= thresh) {
        /* IEEE Spec section 9.2.3.5 says this should be greater than
           or equal */
        macmib_.FailedCount++;
        /* tell the callback the send operation failed
           before discarding the packet */
        hdr_cmn *ch = HDR_CMN(pktTx_);
        if (ch->xmit_failure_) {
                        ch->size() -= phymib_.getHdrLen11();
            ch->xmit_reason_ = XMIT_REASON_ACK;
                        ch->xmit_failure_(pktTx_->copy(),
                                          ch->xmit_failure_data_);
                }

        discard(pktTx_, DROP_MAC_RETRY_COUNT_EXCEEDED);
        pktTx_ = 0;
        *rcount = 0;
        rst_cw();
    }
    else {
        struct hdr_mac802_11 *dh;
        dh = HDR_MAC802_11(pktTx_);
        dh->dh_fc.fc_retry = 1;


        sendRTS(ETHER_ADDR(mh->dh_ra));
        inc_cw();

        // BUGFIX UKA: EIFS
        // Call StartRetransmitBackoff at the ned of the method instead of
        // directly starting backoff

        // old implementation

        // mhBackoff_.start(cw_, is_idle());
        // BUGFIX UKA END: EIFS
    }

    // BUGFIX UKA: transmission, EIFS
    // transmission: backoff is started here, because it has to be done in every
    // case and not only in case of not reaching the limit.
    // EIFS: backoff is not started directly but uses StartRetransmitBackoff
    // method.

    // new implementation

    StartRetransmitBackoff();
    // BUGFIX UKA END: transmission, EIFS
}

void
Mac802_11::send(Packet *p, Handler *h)
{
    double rTime;
    struct hdr_mac802_11* dh = HDR_MAC802_11(p);

    EnergyModel *em = netif_->node()->energy_model();
    if (em && em->sleep()) {
        em->set_node_sleep(0);
        em->set_node_state(EnergyModel::INROUTE);
    }

    callback_ = h;
    sendDATA(p);
    sendRTS(ETHER_ADDR(dh->dh_ra));

    /*
     * Assign the data packet a sequence number.
     */
    dh->dh_scontrol = sta_seqno_++;

    /*
     *  If the medium is IDLE, we must wait for a DIFS
     *  Space before transmitting.
     */
    if(mhBackoff_.busy() == 0) {
        if(is_idle()) {
            if (mhDefer_.busy() == 0) {
                /*
                 * If we are already deferring, there is no
                 * need to reset the Defer timer.
                 */
                // BUGFIX UKA: transmission
                // The station must defer only for DIFS if the medium is free
                // and the is no backoff or defer runnning

                // old implementation

                // rTime = (Random::random() % cw_)
                //     * (phymib_.getSlotTime());
                // mhDefer_.start(phymib_.getDIFS() + rTime);

                // new implementation

                mhDefer_.start(phymib_.getDIFS() );
                // BUGFIX UKA END: transmission

            }
            // BUGFIX UKA: EIFS
            // support of EIFS causes new sitauations at packet sending as well
            // If the last packet was not correct and defer is running and the
            // medium is free, defer runs in an EIFS period. If now the time
            // until expiration is smaller than DIFS, it is necessary to wait
            // at least for an additional period of DIFS before sending.

            // new implementation

            else {
                if (last_packet_correct_ == false && mhDefer_.expire() < phymib_.getDIFS()) {
                    // Defer runs because of EIFS and expiration
                    // time is shorter than DIFS
                    // => defer for DIFS again
                    mhDefer_.stop();
                    mhDefer_.start(phymib_.getDIFS());
                }
            }
            // BUGFIX UKA END: EIFS
        } else {
            /*
             * If the medium is NOT IDLE, then we start
             * the backoff timer.
             */

            // BUGFIX UKA: EIFS
            // method call for backoff start changed

            // old implementation

            // mhBackoff_.start(cw_, is_idle());

            // new implementation

            mhBackoff_.start(cw_, is_idle(), phymib_.getDIFS());
            // BUGFIX UKA END: EIFS
        }
    }
}

void
Mac802_11::recv(Packet *p, Handler *h)
{
```

```
struct hdr_cmn *hdr = HDR_CMN(p);
/*
 * Sanity Check
 */
assert(initialized());

/*
 *  Handle outgoing packets.
 */
if(hdr->direction() == hdr_cmn::DOWN) {
    send(p, h);
    return;
}
/*
 *  Handle incoming packets.
 *
 *  We just received the 1st bit of a packet on the network
 *  interface.
 *
 */

/*
 *  If the interface is currently in transmit mode, then
 *  it probably won’t even see this packet.  However, the
 *  "air" around me is BUSY so I need to let the packet
 *  proceed.  Just set the error flag in the common header
 *  to that the packet gets thrown away.
 */

// BUGFIX UKA: TxRxError
// packets at the same time physically. In the original simulator
// however such a packet is "received", marked errornous, discarded
// and an EIFS period follows.
// This however should not happen, because the packet is not sensed.
// Therefore the packet that is received is marked special and handled
// correctly after complete "reception" (handle it as "never sensed")

// old implementation

//if(tx_active_ && hdr->error() == 0) {
//    hdr->error() = 1;
// }

// new implementation

if (tx_active_) {
     hdr->error() = TX_RX_ERROR;
}
// BUGFIX UKA END: TxRxError

// BUGFIX UKA: EIFS
// on receiving a packet, a running EIFS period is stopped

// new implementation

reset_eifs();
// BUGFIX UKA END: EIFS


if(rx_state_ == MAC_IDLE) {
    setRxState(MAC_RECV);
    pktRx_ = p;
    /*
     * Schedule the reception of this packet, in
     * txtime seconds.
     */
    mhRecv_.start(txtime(p));

    // BUGFIX UKA: capture
    // The starting time of packet reception has to be stored to make sure
    // the capture mechanism is simulated correctly. The new capture
    // effect does not work if the starting time of the second packet
    // reception is in the interval between 4 and 10ms after the start of
    // the first reception.

    // new implementation

    time_start_pktRx_ = Scheduler::instance().clock();
    // BUGFIX UKA END: capture

} else {
    /*
     *  If the power of the incoming packet is smaller than the
     *  power of the packet currently being received by at least
             *  the capture threshold, then we ignore the new packet.
     */
```

```
        // BUGFIX UKA: capture
        // Also support the new capture effect (can be activated and
        // deactivated using the tcl variable newchipset_). In case of
        // activation a capture is also possible if the packet arriving
        // later has a higher power of reception and does not reach the
        // receiver within an interval of 4 to 10ms after start of reception
        // of the first packet due to synchronization constraints.

        // old implementation

        /*
            if(pktRx_->txinfo_.RxPr / p->txinfo_.RxPr >= p->txinfo_.CPThresh) {
                capture(p);
            } else {
                collision(p);
            }
        */

        // new implementation

        // store current time and capture threshold
        // (given in dB; transform to a factor)
        double now = Scheduler::instance().clock();
        double Threshold = pow(10,p->txinfo_.CPThresh/10);
        if (phymib_.get_newchipset() == false) {
            // classic chipset, capture only possible if first packet is stronger
            if(pktRx_->txinfo_.RxPr / p->txinfo_.RxPr >= Threshold) {
                capture(p);
            } else {
                collision(p);
            }
        } else {
            // improved chipset
            if (pktRx_->txinfo_.RxPr / p->txinfo_.RxPr >= Threshold){
                capture(p);
            } else if ((p->txinfo_.RxPr / pktRx_->txinfo_.RxPr >= Threshold )&&
                   ((time_start_pktRx_ + 4e-6 > now)||(time_start_pktRx_ + 10e-6 < now))) {
                   // in case that second packet is stronger, check if
                   // sychronization is possible
                capture(p);
                time_start_pktRx_ = now;
            } else {
                collision(p);
            }
        }
        // BUGFIX UKA END: capture
    }
}

void
Mac802_11::recv_timer()
{
    u_int32_t src;
    hdr_cmn *ch = HDR_CMN(pktRx_);
    hdr_mac802_11 *mh = HDR_MAC802_11(pktRx_);
    u_int32_t dst = ETHER_ADDR(mh->dh_ra);

    u_int8_t  type = mh->dh_fc.fc_type;
    u_int8_t  subtype = mh->dh_fc.fc_subtype;

    assert(pktRx_);
    assert(rx_state_ == MAC_RECV || rx_state_ == MAC_COLL);

        /*
         *  If the interface is in TRANSMIT mode when this packet
         *  "arrives", then I would never have seen it and should
         *  do a silent discard without adjusting the NAV.
         */

    // BUGFIX UKA: TxRxError
    // Discard a packet that reaches the station during a transmission.
    // The packet could technically not be received, however it is
    // possible in simulation. Therefore, such packets are marked
    // special and just ignored after complete "reception".

    // old implementation

    //if(tx_active_) {
    //        Packet::free(pktRx_);
    //        goto done;
    //}

    // new implementation

    if(tx_active_ || ch->error() == TX_RX_ERROR) {
        Packet::free(pktRx_);
        goto done;
```

```
     }
// BUGFIX UKA END: TxRxError

/*
 * Handle collisions.
 */
if(rx_state_ == MAC_COLL) {
    discard(pktRx_, DROP_MAC_COLLISION);
    // BUGFIX UKA: EIFS
    // Start an EIFS period instead of the NAV timer by calling set_eifs

    // old implementation

    // set_nav(usec(phymib_.getEIFS()));

    // new implementation

    set_eifs();
    // BUGFIX UKA END: EIFS

    goto done;
}

/*
 * Check to see if this packet was received with enough
 * bit errors that the current level of FEC still could not
 * fix all of the problems - ie; after FEC, the checksum still
 * failed.
 */
if( ch->error() ) {
    Packet::free(pktRx_);

    // BUGFIX UKA: EIFS
    // Start an EIFS period instead of the NAV timer by calling set_eifs

    // old implementation

    // set_nav(usec(phymib_.getEIFS()));

    // new implementation

    set_eifs();
    // BUGFIX UKA END: EIFS

    goto done;
}

// BUGFIX UKA: EIFS
// At this point it is sure that an error-free packet is received.
// Remember this fact.

// new implementation

last_packet_correct_ = true;
// BUGFIX UKA END: EIFS

/*
 * IEEE 802.11 specs, section 9.2.5.6
 *    - update the NAV (Network Allocation Vector)
 */
if(dst != (u_int32_t)index_) {
    set_nav(mh->dh_duration);
}

    /* tap out - */
if (tap_ && type == MAC_Type_Data &&
    MAC_Subtype_Data == subtype )
    tap_->tap(pktRx_);
/*
 * Adaptive Fidelity Algorithm Support - neighborhood infomation
 * collection
 *
 * Hacking: Before filter the packet, log the neighbor node
 * I can hear the packet, the src is my neighbor
 */
if (netif_->node()->energy_model() &&
    netif_->node()->energy_model()->adaptivefidelity()) {
    src = ETHER_ADDR(mh->dh_ta);
    netif_->node()->energy_model()->add_neighbor(src);
}
/*
 * Address Filtering
 */
if(dst != (u_int32_t)index_ && dst != MAC_BROADCAST) {
    /*
     *  We don’t want to log this event, so we just free
     *  the packet instead of calling the drop routine.
```

```
         */
        discard(pktRx_, "---");
        goto done;
    }

    switch(type) {

    case MAC_Type_Management:
        discard(pktRx_, DROP_MAC_PACKET_ERROR);
        goto done;
    case MAC_Type_Control:
        switch(subtype) {
        case MAC_Subtype_RTS:
            recvRTS(pktRx_);
            break;
        case MAC_Subtype_CTS:
            recvCTS(pktRx_);
            break;
        case MAC_Subtype_ACK:
            recvACK(pktRx_);
            break;
        default:
            fprintf(stderr,"recvTimer1:Invalid MAC Control Subtype %x\n",
                subtype);
            exit(1);
        }

        // BUGFIX UKA: Memory
        // The memory used by MAC Control packets should be freed if packets
        // are received and not used anymore This step was missing in the
        // original implementation.

        // new implementation

        Packet::free(pktRx_);
        // BUGFIX UKA END: Memory

        break;
    case MAC_Type_Data:
        switch(subtype) {
        case MAC_Subtype_Data:
            recvDATA(pktRx_);
            break;
        default:
            fprintf(stderr, "recv_timer2:Invalid MAC Data Subtype %x\n",
                subtype);
            exit(1);
        }
        break;
    default:
        fprintf(stderr, "recv_timer3:Invalid MAC Type %x\n", subtype);
        exit(1);
    }
 done:
    pktRx_ = 0;
    rx_resume();
}



void
Mac802_11::recvDATA(Packet *p)
{
    struct hdr_mac802_11 *dh = HDR_MAC802_11(p);
    u_int32_t dst, src, size;
    struct hdr_cmn *ch = HDR_CMN(p);

    dst = ETHER_ADDR(dh->dh_ra);
    src = ETHER_ADDR(dh->dh_ta);
    size = ch->size();
    /*
     * Adjust the MAC packet size - ie; strip
     * off the mac header
     */
    ch->size() -= phymib_.getHdrLen11();
    ch->num_forwards() += 1;

    /*
     *  If we sent a CTS, clean up...
     */
    if(dst != MAC_BROADCAST) {
        if(size >= macmib_.getRTSThreshold()) {
            if (tx_state_ == MAC_CTS) {
                assert(pktCTRL_);
                Packet::free(pktCTRL_); pktCTRL_ = 0;
                mhSend_.stop();
                /*
```

```
             * Our CTS got through.
             */
        } else {
            discard(p, DROP_MAC_BUSY);
            return;
        }
        sendACK(src);
        tx_resume();
    } else {
        /*
         *  We did not send a CTS and there’s no
         *  room to buffer an ACK.
         */
        if(pktCTRL_) {
            discard(p, DROP_MAC_BUSY);
            return;
        }
        sendACK(src);
        // BUGFIX UKA: transmission
        // An ACK packet should be sent in every case directly after a SIFS
        // period, even if there is a timeout pending
        // (See 9.2.8 in the Standard)

        // old implementation

        // if(mhSend_.busy() == 0)
        //  tx_resume();

        // new implementation

        if (mhSend_.busy() == 0) {
            tx_resume();
        } else {
            // the station is waiting for a timeout. Stop waiting and
            // schedule a retransmit (done by sendHandler). tx_resume
            // to send the ACK is called from within sendHandler!
            mhSend_.stop();
            sendHandler();
        }
        // BUGFIX UKA END: transmission
    }
}

/* ============================================================
   Make/update an entry in our sequence number cache.
   ============================================================ */

/* Changed by Debojyoti Dutta. This upper loop of if{}else was
   suggested by Joerg Diederich <dieder@ibr.cs.tu-bs.de>.
   Changed on 19th Oct’2000 */

if(dst != MAC_BROADCAST) {
    if (src < (u_int32_t) cache_node_count_) {
        Host *h = &cache_[src];

        if(h->seqno && h->seqno == dh->dh_scontrol) {
            discard(p, DROP_MAC_DUPLICATE);
            return;
        }
            h->seqno = dh->dh_scontrol;
        } else {
        static int count = 0;
        if (++count <= 10) {
            printf ("MAC_802_11: accessing MAC cache_ array out of range
                    (src %u, dst %u, size %d)!\n", src, dst, cache_node_count_);
            if (count == 10)
                printf ("[suppressing additional MAC cache_ warnings]\n");
        };
    };
}

/*
 *  Pass the packet up to the link-layer.
 *  XXX - we could schedule an event to account
 *  for this processing delay.
 */

/* in BSS mode, if a station receives a packet via
 * the AP, and higher layers are interested in looking
 * at the src address, we might need to put it at
 * the right place - lest the higher layers end up
 * believing the AP address to be the src addr! a quick
 * grep didn’t turn up any higher layers interested in
 * the src addr though!
 * anyway, here if I’m the AP and the destination
 * address (in dh_3a) isn’t me, then we have to fwd
 * the packet; we pick the real destination and set
```

```
         * set it up for the LL; we save the real src into
         * the dh_3a field for the 'interested in the info'
         * receiver; we finally push the packet towards the
         * LL to be added back to my queue - accomplish this
         * by reversing the direction!*/

        if ((bss_id() == addr()) && ((u_int32_t)ETHER_ADDR(dh->dh_ra)!= MAC_BROADCAST)&&
           ((u_int32_t)ETHER_ADDR(dh->dh_3a) != addr())) {
            struct hdr_cmn *ch = HDR_CMN(p);
            u_int32_t dst = ETHER_ADDR(dh->dh_3a);
            u_int32_t src = ETHER_ADDR(dh->dh_ta);
            /* if it is a broadcast pkt then send a copy up
             * my stack also
             */
            if (dst == MAC_BROADCAST) {
                uptarget_->recv(p->copy(), (Handler*) 0);
            }

            ch->next_hop() = dst;
            STORE4BYTE(&src, (dh->dh_3a));
            ch->addr_type() = NS_AF_ILINK;
            ch->direction() = hdr_cmn::DOWN;
        }

        uptarget_->recv(p, (Handler*) 0);
}


void
Mac802_11::recvACK(Packet *p)
{
    struct hdr_cmn *ch = HDR_CMN(p);

    if(tx_state_ != MAC_SEND) {
        discard(p, DROP_MAC_INVALID_STATE);
        return;
    }
    assert(pktTx_);

    mhSend_.stop();

    /*
     * The successful reception of this ACK packet implies
     * that our DATA transmission was successful.  Hence,
     * we can reset the Short/Long Retry Count and the CW.
     *
     * need to check the size of the packet we sent that's being
     * ACK'd, not the size of the ACK packet.
     */
    if((u_int32_t) HDR_CMN(pktTx_)->size() <= macmib_.getRTSThreshold())
        ssrc_ = 0;
    else
        slrc_ = 0;
    rst_cw();
    Packet::free(pktTx_);
    pktTx_ = 0;

    /*
     * Backoff before sending again.
     */
    assert(mhBackoff_.busy() == 0);

    // BUGFIX UKA: EIFS
    // changed method definition for backoff start causes changed call

    // old implementation

    // Backoff_.start(cw_, is_idle());

    // new implementation

    mhBackoff_.start(cw_, is_idle(), phymib_.getDIFS());
    // BUGFIX UKA END: EIFS

    tx_resume();

    mac_log(p);
}
```

A.2 mac/mac-802_11.h

```
class PHY_MIB {
public:
    PHY_MIB(Mac802_11 *parent);

    inline u_int32_t getCWMin() { return(CWMin); }
    inline u_int32_t getCWMax() { return(CWMax); }
    inline double getSlotTime() { return(SlotTime); }
    inline double getSIFS() { return(SIFSTime); }
    inline double getPIFS() { return(SIFSTime + SlotTime); }

    inline double getDIFS() { return(SIFSTime + 2 * SlotTime); }

    inline double getEIFS() {
        // see (802.11-1999, 9.2.10)
        return(SIFSTime + getDIFS()
                       + (8 *  getACKlen())/PLCPDataRate);
    }
    inline u_int32_t getPreambleLength() { return(PreambleLength); }
    inline double getPLCPDataRate() { return(PLCPDataRate); }

    inline u_int32_t getPLCPhdrLen() {
        return((PreambleLength + PLCPHeaderLength) >> 3);
    }

    inline u_int32_t getHdrLen11() {
        return(getPLCPhdrLen() + sizeof(struct hdr_mac802_11)
                       + ETHER_FCS_LEN);
    }

    inline u_int32_t getRTSlen() {
        return(getPLCPhdrLen() + sizeof(struct rts_frame));
    }

    inline u_int32_t getCTSlen() {
        return(getPLCPhdrLen() + sizeof(struct cts_frame));
    }

    inline u_int32_t getACKlen() {
        return(getPLCPhdrLen() + sizeof(struct ack_frame));
    }

    // BUGFIX UKA: capture
    // returns true if the new implementation of the chipset is supported,
    // otherwise false

    // new implementation

    inline bool get_newchipset() {
        return newchipset;
    }
    // BUGFIX UKA END: capture

 private:

    u_int32_t    CWMin;
    u_int32_t    CWMax;
    double        SlotTime;
    double        SIFSTime;
    u_int32_t    PreambleLength;
    u_int32_t    PLCPHeaderLength;
    double        PLCPDataRate;

    // BUGFIX UKA: capture
    // true if new chipset implementation is used, false otherwise

    // new implementation

    int newchipset;
    // BUGFIX UKA END: capture
};

// BUGFIX UKA: TxRxError
// Define a specific value to mark packets that are not really received
// because of the RxTxError

// new implementation

#define TX_RX_ERROR 5
// BUGFIX UKA END: TxRxError


class Mac802_11 : public Mac {
    friend class DeferTimer;


    friend class BackoffTimer;
    friend class IFTimer;
    friend class NavTimer;
    friend class RxTimer;
    friend class TxTimer;
public:
    Mac802_11();
    void            recv(Packet *p, Handler *h);
    inline int        hdr_dst(char* hdr, int dst = -2);
    inline int        hdr_src(char* hdr, int src = -2);
    inline int        hdr_type(char* hdr, u_int16_t type = 0);

    inline int bss_id() { return bss_id_; }

    // Added by Sushmita to support event tracing
        void trace_event(char *, Packet *);
        EventTrace *et_;

protected:
    void    backoffHandler(void);
    void    deferHandler(void);
    void    navHandler(void);
    void    recvHandler(void);
    void    sendHandler(void);
    void    txHandler(void);

private:
    int            command(int argc, const char*const* argv);

    /*
     * Called by the timers.
     */
    void            recv_timer(void);
    void            send_timer(void);
    int            check_pktCTRL();
    int            check_pktRTS();
    int            check_pktTx();

    /*
     * Packet Transmission Functions.
     */
    void    send(Packet *p, Handler *h);
    void     sendRTS(int dst);
    void    sendCTS(int dst, double duration);
    void    sendACK(int dst);
    void    sendDATA(Packet *p);

    // BUGFIX UKA: EIFS
    // definition of new method

    // new implementation

    void    StartRetransmitBackoff();
    // BUGFIX UKA END: EIFS

    void    RetransmitRTS();
    void    RetransmitDATA();

    /*
     * Packet Reception Functions.
     */
    void    recvRTS(Packet *p);
    void    recvCTS(Packet *p);
    void    recvACK(Packet *p);
    void    recvDATA(Packet *p);

    void            capture(Packet *p);
    void            collision(Packet *p);
    void            discard(Packet *p, const char* why);
    void            rx_resume(void);
    void            tx_resume(void);

    inline int        is_idle(void);

    /*
     * Debugging Functions.
     */
    void            trace_pkt(Packet *p);
    void            dump(char* fname);

    inline int initialized() {
        return (cache_ && logtarget_
                        && Mac::initialized());
    }

    inline void mac_log(Packet *p) {
                logtarget_->recv(p, (Handler*) 0);
        }

    double txtime(Packet *p);
    double txtime(double psz, double drt);
    double txtime(int bytes) { /* clobber inherited txtime() */ abort(); return 0;}

    inline void transmit(Packet *p, double timeout);
    inline void checkBackoffTimer(void);
    inline void postBackoff(int pri);
    inline void setRxState(MacState newState);
    inline void setTxState(MacState newState);

    // BUGFIX UKA: EIFS
    // declaration of methods to set and reset EIFS

    // new implementation

    inline void set_eifs();
    inline void reset_eifs();
    // BUGFIX UKA END: EIFS


    inline void inc_cw() {
        cw_ = (cw_ << 1) + 1;
        if(cw_ > phymib_.getCWMax())
            cw_ = phymib_.getCWMax();
    }
    inline void rst_cw() { cw_ = phymib_.getCWMin(); }

    inline double sec(double t) { return(t *= 1.0e-6); }
    inline u_int16_t usec(double t) {
        u_int16_t us = (u_int16_t)floor((t *= 1e6) + 0.5);
        return us;
    }
    inline void set_nav(u_int16_t us) {
        double now = Scheduler::instance().clock();
        double t = us * 1e-6;
        if((now + t) > nav_) {
            nav_ = now + t;
            if(mhNav_.busy())
                mhNav_.stop();
            mhNav_.start(t);
        }
    }

protected:
    PHY_MIB         phymib_;
        MAC_MIB         macmib_;

       /* the macaddr of my AP in BSS mode; for IBSS mode
        * this is set to a reserved value IBSS_ID - the
        * MAC_BROADCAST reserved value can be used for this
        * purpose
        */
       int     bss_id_;
       enum    {IBSS_ID=MAC_BROADCAST};


private:
    double        basicRate_;
     double        dataRate_;

    /*
     * Mac Timers
     */
    IFTimer        mhIF_;        // interface timer
    NavTimer    mhNav_;        // NAV timer
    RxTimer        mhRecv_;        // incoming packets
    TxTimer        mhSend_;        // outgoing packets

    DeferTimer    mhDefer_;    // defer timer
    BackoffTimer    mhBackoff_;    // backoff timer

    /* ============================================================
       Internal MAC State
       ============================================================ */
    double        nav_;        // Network Allocation Vector

    // BUGFIX UKA: EIFS
    // variable that remembers if the last received packet was error-free

    // new implementation

    int        last_packet_correct_;
    // BUGFIX UKA END: EIFS

    // BUGFIX UKA: CAPTURE
    // Variable saving the time of reception start

    // new implementation

    double time_start_pktRx_;
    // BUGFIX UKA END: CAPTURE
    MacState    rx_state_;    // incoming state (MAC_RECV or MAC_IDLE)
    MacState    tx_state_;    // outgoint state
    int        tx_active_;    // transmitter is ACTIVE

    Packet          *eotPacket_;    // copy for eot callback

    Packet        *pktRTS_;    // outgoing RTS packet
    Packet        *pktCTRL_;    // outgoing non-RTS packet

    u_int32_t    cw_;        // Contention Window
    u_int32_t    ssrc_;        // STA Short Retry Count
    u_int32_t    slrc_;        // STA Long Retry Count

    int        min_frame_len_;

    NsObject*    logtarget_;
    NsObject*       EOTtarget_;     // given a copy of packet at TX end



    /* ============================================================
       Duplicate Detection state
       ============================================================ */
    u_int16_t    sta_seqno_;    // next seqno that I'll use
    int        cache_node_count_;
    Host        *cache_;
};
```

A.3 mac/mac-timers.cc

```
// BUGFIX UKA: EIFS
// New additional parameter for this method, specifing the time the backoff
// process should wait before counting down the slots.
// The value is of interest ONLY if medium is NOT busy at the moment
// of calling this method!

// old implementation

//void
//BackoffTimer::start(int cw, int idle)

// new implementation

void
BackoffTimer::start(int cw, int idle, double time)
// BUGFIX UKA END: EIFS

{
    Scheduler &s = Scheduler::instance();

    assert(busy_ == 0);

    busy_ = 1;
    paused_ = 0;
    stime = s.clock();

    rtime = (Random::random() % cw) * mac->phymib_.getSlotTime();



#ifdef USE_SLOT_TIME
    ROUND_TIME();
#endif

    // BUGFIX UKA: EIFS
    // use the time parameter if necessary

    // old implementation

    /*
    difs_wait = 0.0;
    if(idle == 0)
        paused_ = 1;
    else {
        assert(rtime >= 0.0);
        s.schedule(this, &intr, rtime);
    }
    */

    // new implementation

    if(idle == 0) {
        difs_wait = 0.0;
        paused_ = 1;
    } else {
        difs_wait = time;
        assert(rtime >= 0.0);
        s.schedule(this, &intr, rtime + difs_wait);
    }
    // BUGFIX UKA END: EIFS
}
```

A.4 mac/mac-timers.h

```
class BackoffTimer : public MacTimer {
public:
    BackoffTimer(Mac802_11 *m) : MacTimer(m), difs_wait(0.0) {}
    // BUGFIX UKA: EIFS
    // new declaration of start method with new parameter

    // old implementation

    // void    start(int cw, int idle);

    // new implementation

    void    start(int cw, int idle, double time);
    // BUGFIX UKA END: EIFS

    void    handle(Event *e);
    void    pause(void);
    void    resume(double difs);
private:
    double    difs_wait;
};
```

A.5 trace/cmu-trace.h

```
// BUGFIX UKA: capture
// packet drop in case of a packet capture
#define DROP_MAC_CAPTURE        "CAP"
// BUGFIX UKA END: capture
```

A.6 tcl/lib/ns-default.tcl

```
# BUGFIX UKA: capture
# By default, new implementation of the card chipset is deactivated.
 Mac/802_11 set newchipset_ false
# BUGFIX UKA END: capture
```