```
/*
 * CsmaMac.cc
 *
 */

#include "CsmaMac.h"
#include <cmath>
#include "AppMessage_m.h"
#include "AppResponse_m.h"
#include "CSRequest_m.h"
#include "CSResponse_m.h"
#include "MacPacket_m.h"
#include "TransmissionConfirmation_m.h"
#include "TransmissionIndication_m.h"
#include "TransmissionRequest_m.h"


Define_Module(CsmaMac);

using namespace omnetpp;


//
==================================================================================
//
==================================================================================
//
// Initialization and configuration
//
//
==================================================================================
//
==================================================================================

simsignal_t bufferLossSig = cComponent::registerSignal("bufferLossSig");
simsignal_t bufferEnteredSig = cComponent::registerSignal("bufferEnteredSig");
simsignal_t numberAttemptsSig = cComponent::registerSignal("numberAttemptsSig");
simsignal_t accessFailedSig = cComponent::registerSignal("accessFailedSig");
simsignal_t accessSuccessSig = cComponent::registerSignal("accessSuccessSig");


void CsmaMac::initialize(void)
{
    // read config values and check their validity
    ownAddress = par("ownAddress");
    bufferSize = par("bufferSize");
    maxBackoffs = par("maxBackoffs");
    maxAttempts = par("maxAttempts");
    macOverheadSizeData = par("macOverheadSizeData");
    macOverheadSizeAck = par("macOverheadSizeAck");
    macAckDelay = par("macAckDelay");
    ackTimeout = par("ackTimeout");

    assert(bufferSize >= 1);
    assert(maxBackoffs > 0);
    assert(maxAttempts > 0);
    assert(macOverheadSizeData > 0);
    assert(macOverheadSizeAck > 0);
    assert(macAckDelay > 0);
    assert(ackTimeout > 0);

    // gate id's
    gidFromApp = findGate("fromHigher");
    gidToApp = findGate("toHigher");
    gidFromXcvr = findGate("fromTransceiver");
    gidToXcvr = findGate("toTransceiver");

    // self messages
    currMsg = nullptr;
    currAckMsg = nullptr;
    csBackoffCompMsg = new cMessage("csBackoffComp");
    retransmitCompMsg = new cMessage("retransmitComp");
    ackTimeoutMsg = new cMessage("ackTimeout");
    macAckDelayExpired = new cMessage("macAckDelayExpired");
    succAttBackoffMsg = new cMessage("succAttBackoff");

    state = IDLE;

    attemptCnt = 0;
    backoffCnt = 0;
    ackSent = false;
}


void CsmaMac::handleMessage(cMessage* msg)
{
    if ((msg->arrivedOn(gidFromXcvr)) &&
dynamic_cast<TransmissionConfirmation*>(msg))
    {
        handleTransmissionConfirmation((TransmissionConfirmation*) msg);
        return;
    }
    if ((msg->arrivedOn(gidFromXcvr)) && dynamic_cast<TransmissionIndication*>(msg))
    {
        handleTransmissionIndication((TransmissionIndication*) msg);
        return;
    }
    if ((msg->arrivedOn(gidFromApp)) && dynamic_cast<AppMessage*>(msg))
    {
        handleAppMessage((AppMessage*) msg);
        return;
    }
    if (dynamic_cast<CSResponse*>(msg))
    {
        handleCSResponse((CSResponse*) msg);
        return;
    }
    if (msg == ackTimeoutMsg)
    {
        handleAckTimeout();
        return;
    }
    if (msg == retransmitCompMsg)
    {
        handleRetrasmitComp();
        return;
    }
    if (msg == csBackoffCompMsg)
    {
        handleCsBackoff();
        return;
    }
    if (msg == macAckDelayExpired)
    {
        sendAckMAC();
        return;
    }
    if (msg == succAttBackoffMsg)
    {
        handleSuccBackoff();
        return;
    }
    EV << msg->getArrivalGate() << endl;
    error("Unexpected message");
}


void CsmaMac::handleCSResponse(CSResponse* csResp)
{
    EV << ownAddress << ": Handling CS response message" << endl;
    assert(state == SENSE);

    if (csResp->getBusyChannel()) {
        EV << ownAddress << ": Channel is busy" << endl;
        csBackoff = par("csBackoffDistribution");
        scheduleAt(simTime() + csBackoff, csBackoffCompMsg);
        state = CSBACKOFF;
    } else {
        EV << ownAddress << ": Channel is free, sending transmission request" <<
endl;
```

```
        backoffCnt = 0;
        TransmissionRequest *transReq = new TransmissionRequest();
        transReq->encapsulate(currMsg->dup());
        send(transReq, gidToXcvr);
        emit(numberAttemptsSig, attemptCnt);
        state = TRANSMIT;
    }
    delete(csResp);
}

void CsmaMac::handleTransmissionConfirmation(TransmissionConfirmation* transConf)
{
    if (!ackSent) {
        EV << ownAddress << ": Handling transmission confirmation message" << endl;

        assert(state == TRANSMIT);
        scheduleAt(simTime() + ackTimeout, ackTimeoutMsg);
        state = ACK_LISTEN;
    } else {
        EV << ownAddress << ": Ack transmission confirmed" << endl;
        ackSent = false;
    }
    delete transConf;
}

void CsmaMac::handleTransmissionIndication(TransmissionIndication* transInd)
{
    EV << ownAddress << ": Handling transmission indication" << endl;
    MacPacket *decapMsg = dynamic_cast<MacPacket*>(transInd->decapsulate());
    assert(decapMsg->getReceiverAddress() == ownAddress);
    if (decapMsg->getMacPacketType() == MacDataPacket) {
        handleDataPkt(decapMsg);
    } else if (decapMsg->getMacPacketType() == MacAckPacket) {
        handleAckPkt(decapMsg);
    }
    delete transInd;
}

void CsmaMac::handleAppMessage(AppMessage* appMsg)
{
    EV << ownAddress << ": Handling app message" << endl;
    // Check if buffer is full
    if (macBuff.getLength() == bufferSize) {
        EV << ownAddress << ": Buffer full, dropping packet" << endl;
        AppResponse *resp;
        resp = new AppResponse();
        resp->setSequenceNumber(appMsg->getSequenceNumber());
        resp->setOutcome(BufferDrop);
        EV << ownAddress << ": Sending bufferDrop AppResponse message with seqNo="
                << resp->getSequenceNumber() << endl;
        send(resp, gidToApp);
        delete appMsg;
        emit(bufferLossSig, true);
        return;
    }
    // If not full, add to buffer and check state
    macBuff.insert(appMsg);
    emit(bufferEnteredSig, true);
    EV << ownAddress << ": Appending app message to MAC buffer" << endl;
    if (state == IDLE) {
        beginDataMAC();
    }
}

void CsmaMac::beginDataMAC()
{
    assert(state == IDLE);
    assert(macBuff.getLength() > 0);

    AppMessage *msg = dynamic_cast<AppMessage*>(macBuff.pop());
    MacPacket *encapMsg = new MacPacket();
    encapMsg->setByteLength(macOverheadSizeData);
    encapMsg->setTransmitterAddress(ownAddress);
    encapMsg->setReceiverAddress(msg->getReceiverAddress());
    encapMsg->setMacPacketType(MacDataPacket);
    encapMsg->encapsulate(msg);

    EV << ownAddress << ": Created data MacPacket with Rx address: "
            << msg->getReceiverAddress() << ", and Tx address: "
            << ownAddress << endl;

    currMsg = encapMsg;
    attemptCnt = 0;
    attemptTransmit();
}

void CsmaMac::attemptTransmit()
{
    EV << ownAddress << ": Attempting transmission" << endl;
    attemptCnt++;
    EV << ownAddress << ": Transmission attempt number: " << attemptCnt << endl;
    backoffCnt = 0;
    carrierSenseAttempt();
}

void CsmaMac::carrierSenseAttempt()
{
    backoffCnt++;
    EV << ownAddress << ": Carrying out carrier sensing attempt: " << backoffCnt <<
```

```
endl;

    CSRequest *csReq = new CSRequest();
    send(csReq, gidToXcvr);
    state = SENSE;
}

void CsmaMac::handleDataPkt(MacPacket* decapMsg)
{
    EV << ownAddress << ": Received data packet from Xcvr, forwarding to
Application" << endl;
    AppMessage *appMsg = dynamic_cast<AppMessage*>(decapMsg->decapsulate());
    send(appMsg, gidToApp);

    // This part may need to be optimised in future version. If currently generating
ACK, it won't acknowledge new data packets.
    if (currAckMsg == nullptr){
        AppMessage *ackMsg = new AppMessage();
        ackMsg->setSequenceNumber(appMsg->getSequenceNumber());
        MacPacket *encapMsg = new MacPacket();
        encapMsg->setByteLength(macOverheadSizeAck);
        encapMsg->setTransmitterAddress(ownAddress);
        encapMsg->setReceiverAddress(appMsg->getSenderAddress());
        encapMsg->setMacPacketType(MacAckPacket);
        currAckMsg = encapMsg;
        scheduleAt(simTime() + macAckDelay, macAckDelayExpired);
    }

    delete decapMsg;
}

void CsmaMac::sendAckMAC()
{
    EV << ownAddress << ": Generating and transmitting acknowledgment message. Tx: "
            << currAckMsg->getTransmitterAddress() << ", Rx: "
            << currAckMsg->getReceiverAddress()
            << endl;
    TransmissionRequest *transReq = new TransmissionRequest();
    transReq->encapsulate(currAckMsg);
    send(transReq, gidToXcvr);
    ackSent = true;
    currAckMsg = nullptr;
}

void CsmaMac::handleAckPkt(MacPacket* decapMsg)
{
    EV << ownAddress << ": Received acknowledgment message from Xcvr" << endl;
    assert(state == ACK_LISTEN);
    cancelEvent(ackTimeoutMsg);
    emit(accessSuccessSig, true);
    AppResponse *resp = new AppResponse();
    assert(currMsg);
    AppMessage *appMsg =
dynamic_cast<AppMessage*>(currMsg->getEncapsulatedPacket());
    resp->setSequenceNumber(appMsg->getSequenceNumber());
    resp->setOutcome(Success);
    send(resp, gidToApp);

    delete decapMsg;
    delete currMsg;
    currMsg = nullptr;
    EV << ownAddress << ": Backing off after successful attempt" << endl;
    succBackoff = par("succBackoffDistribution");
    scheduleAt(simTime()+succBackoff, succAttBackoffMsg);
}

void CsmaMac::handleSuccBackoff()
{
    state = IDLE;
    if (macBuff.getLength() > 0){
        beginDataMAC();
    }
}

void CsmaMac::handleAckTimeout() // TODO: Merge this with handleMaxBackoff()
{
    EV << ownAddress << ": ACK timed out" << endl;
    attBackoff = par("attBackoffDistribution");
    scheduleAt(simTime() + attBackoff, retransmitCompMsg);
    state = RETRANSMIT;
    emit(accessFailedSig, true);
}

void CsmaMac::handleMaxBackoff()
{
    EV << ownAddress << ": Max backoffs reached" << endl;
    attBackoff = par("attBackoffDistribution");
    scheduleAt(simTime() + attBackoff, retransmitCompMsg);
    state = RETRANSMIT;
}

void CsmaMac::handleRetrasmitComp()
{
    assert(state == RETRANSMIT);
    if (attemptCnt < maxAttempts) {
        attemptTransmit();
    } else {
        handleFailedTransmission();
    }
```

```
}

void CsmaMac::handleCsBackoff()
{
    assert(state == CSBACKOFF);
    if (backoffCnt < maxBackoffs) {
        carrierSenseAttempt();
    } else {
        handleMaxBackoff();
    }
}

void CsmaMac::handleFailedTransmission()
{
    EV << ownAddress << ": Exceeded max attempts, sending failed appResponse to
Application layer" << endl;
    AppResponse *resp = new AppResponse();
    assert(currMsg);
    AppMessage *appMsg =
dynamic_cast<AppMessage*>(currMsg->getEncapsulatedPacket());
    resp->setSequenceNumber(appMsg->getSequenceNumber());
    resp->setOutcome(ChannelFailure);
    send(resp, gidToApp);
    delete currMsg;
    currMsg = nullptr;

    state = IDLE;
    if (macBuff.getLength() > 0){
        beginDataMAC();
    }
}

CsmaMac::~CsmaMac()
{
    cancelAndDelete(currMsg);
    cancelAndDelete(csBackoffCompMsg);
    cancelAndDelete(ackTimeoutMsg);
    cancelAndDelete(retransmitCompMsg);
    cancelAndDelete(currAckMsg);
    cancelAndDelete(macAckDelayExpired);
//    cancelAndDelete(succAttBackoffMsg);
}
```