PROPOSAL
For a Modular, Pluggable 802.11 MAC Model
To Facilitate Experimentation and Contributions
IEEE 802.11 Model Goals

1. Full-featured, validated model
   • support for fragmentation, EDCA, block acknowledgement, frame aggregation, HT extensions, HT/legacy mixed mode... you name it

2. Allow experimentation
   • configurable and hackable

3. Allow experimentation (and experimental features!) without putting validity at risk
IEEE 802.11 Model Goals

• “Allow experimentation and experimental features without putting validity at risk” – HOW?
• Answer: modular, plug-in architecture
  If part X has multiple (pluggable) implementations, then...

  ▼ ▼ ▼ ▼ ▼ ▼ ...

  – users of one implementation are shielded from changes (incl. possible bugs!) in other implementations
  – you may use the simplest implementation of part X that suits your project (less room for bugs, better performance)
  – it helps accepting contributions: when a patch affects only an “experimental” implementation of part X, code review can be more relaxed
Problems With the Current Implementation

• Missing features
  – No fragmentation, aggregation, block ack, etc.

• Monolithic
  – It’s a single class, so any change will affect ALL users
    • This mandates careful review and testing for each and every patch on behalf of INET maintainers!

• Difficult to maintain and extend
  Complicated logic – difficult to comprehend and contribute to
  Symptoms:
  – ~70 data members -- difficult to comprehend and reason about
  – state machine with >50 transitions (plus some extra code on at the top of handleWithFSM()) -- difficult to comprehend or extend
Existing State Machine

9 states
52 transitions
How it grew so big?

- Part of the problem is that the state machine mixes two different aspects: **channel access** (interframe space, backoff period, retries with exponential backoff, etc) with **frame exchanges** (Data+ACK, RTS+CTS+Data+ACK, TXOPs, etc.), and also scrams them into a small number of states → hence the large amount of state variables and FSM transitions

We tried to refactor, really tried...
But it’s time for a reboot
New MAC – Key Ideas

- *Transmit* process(es) decoupled from *Receive* process
- frame exchanges decoupled from channel access
- frame exchanges as building blocks
- many protocol features can be encapsulated in their own C++ classes
  - fragmentation, aggregation, automatic rate control, etc.
Basic Architecture - Concept

- **TX**: deals with channel access, frame, frame exchange, etc.
- **RX**: queue(s), fragmentation, frame exchanges, etc.
- **UpperMAC**: deals with frames, channel state, frame, channel state, etc.
- **PHY**: channel busy, frame, channel state
TX Process: Interface

UpperMAC

transmitContentionFrame(frame, simtime_t ifs, simtime_t eifs, int cw);
transmitImmediateFrame(frame, simtime_t ifs);

transmissionComplete();

mediumStateChanged(bool busy)

badFrameReceived() [for eifs]

transmit frame over radio

PHY

TX

RX
transmitContentionFrame(frame, ifs, eifs, cw)
• used e.g. for data frames
• Note: doesn’t contain retransmission! (it’s done elsewhere)

Busy if:
• receiver senses busy channel, or
• we are transmitting, or
• NAV indicates reservation by other station

* omitted detail: switch to EIFS on reception of frame with bad checksum, and back on correct frame
tx: TX Process: Immediate Frames

transmitImmediateFrame(frame, ifs)
- used e.g. for ACK, CTS, immediate BA, back-to-back data frames, etc.
- no contention

Diagram:
- IDLE
- WAIT-IFS
- TRANSMIT
- Start
- IFS-Done
- TX-Complete

Transition graph:
- IDLE to WAIT-IFS
- WAIT-IFS to TRANSMIT
- TRANSMIT to IDLE (circular transition)
TX Process State Machine

• Why so simple...?
  – Where is ACK, RTS/CTS, etc?
  – Also, where is retransmission handling?
  – EDCA?

• Reason:
  – In early 802.11, frame exchanges were simple: just Data+ACK, RTS+CTS – it could be encoded into the state machine.

Today, no longer! TXOP, Block ACK sequences, reverse direction frame exchange, etc...

So: we want to take the complexity somewhere else

– EDCA: just create 4 instances of TX
RX Process: Interface

- **lowerFrameArrived**(frame)
- **mediumStateChanged**(bool busy)
- **handleLowerFrame**(frame)
- **badFrameReceived**()
- **mediumStateChanged**()
- **handleLowerFrame**(frame)

performs FCS check, maintains NAV
UpperMAC: Interface

Upper layers

- `upperFrameArrived(frame)`
- `sendUp(frame)`
- `transmissionComplete()`

UpperMAC

- `transmitContentionFrame(frame, ifs, eifs, cw)`
- `transmitImmediateFrame(frame, ifs)`

RX

TX
UpperMAC

• Deals with exchanging frames
• Doesn’t need to care about channel access
  – reduces complexity!
• REPLACEABLE! May have simple, advanced and experimental variants
  – 80211b/g, 80211e, 80211n, experimental1, experimental2, etc.
• May be modular in itself (see next slides)
Frame Exchanges

Frame exchanges are...

- C++ classes, used as building block for UpperMAC
- Created dynamically in UpperMAC as response to incoming frames or possibly other events
- Composable (?)
- Examples:
  - Data ACK
  - RTS CTS Data Ack
  - RTS CTS Data Data Data BAR BA
  - Reverse direction frame exchange
  - May map to one TXOP or multiple TXOPs
Frame Exchange: Interface

FrameExchange

- transmitContentionFrame(frame, ifs, eifs, cw)
- transmitImmediateFrame(frame, ifs)
- lowerFrameArrived(frame)
- transmissionComplete()
- frameExchangeFinished(bool success)

TX

RX (via UpperMAC)

Containing UpperMAC
Implementation as State Machine

- Frame Exchange classes may be implemented in terms of state machines. Example: Data + ACK

  STA1  contention  DATA
  STA2  ACK

- invokes `transmitContentionFrame()`
- frames that arrive during IFS and backoff are processed separately by UpperMAC (ACKed, etc)

Exponential backoff procedure is here!

Timeout & retryCount < max / update cw
Step-Based Frame Exchanges

- Frame exchange classes allow for a concise and natural mapping of protocol to code
  
  Example: RTS+CTS+Data+ACK exchange:

<table>
<thead>
<tr>
<th>STA1</th>
<th>contention</th>
<th>RTS</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>STA2</td>
<td></td>
<td>CTS</td>
<td>ACK</td>
</tr>
</tbody>
</table>

  - Can be described in terms of send and expect steps!
  - So: why not define a StepBasedFrameExhange base class that defines send and expect as primitives?
  - Note one difficulty: RTS needs to be retransmitted if there’s no CTS
class SendDataWithRtsCtsFrameExchange : public StepBasedFrameExchange {
    ...;
}

bool SendDataWithRtsCtsFrameExchange::doStep(int step) {
    switch (step) {
        case 0: transmitContentionFrame(buildRtsFrame(dataFrame, difs,...)); return true;
        case 1: expectReply(ctsTimeout); return true; // true=more steps to follow
        case 2: transmitImmediateFrame(dataFrame, sifs); return true;
        case 3: expectReply(ackTimeout); return false; // false=no more steps
    }
}

bool SendDataWithRtsCtsFrameExchange::processReply(int step, Ieee80211Frame *frame) {
    switch (step) {
        case 1: return isCtsFrom(frame, destAddress); // true=accepted
        case 3: return isAckFrom(frame, destAddress);
    }
}

void SendDataWithRtsCtsFrameExchange::processTimeout(int step) {
    switch (step) {
        case 1: if (retryCount < max) {incRetryVariables(); gotoStep(0);} else fail(); break
        case 3: fail(); break;
    }
}
Further Componentization Possibilities

Candidates for wrapping into self-contained classes:

- Fragmentation
- MSDU aggregation
- MPDU aggregation
- Rate control
- Frame exchange selection policy
- ...
Status

- Early implementation draft exists
- Looking for contributors once the design is getting stable
- The plan is to implement multiple UpperMACs of increasing complexity

We plan to implement these, as proof of concept
And hope the community will add others
What do you think?
Let’s discuss it!