PROPOSAL

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

IBM Research - Zurich, Switzerland– September 3-4, 2015

Andras Varga

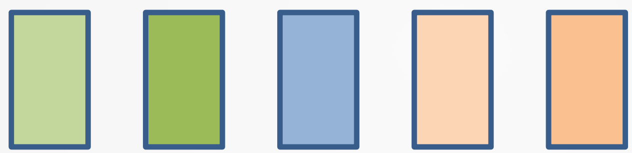
IEEE 802.11 Model Goals

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!) <u>without putting validity at risk</u>

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

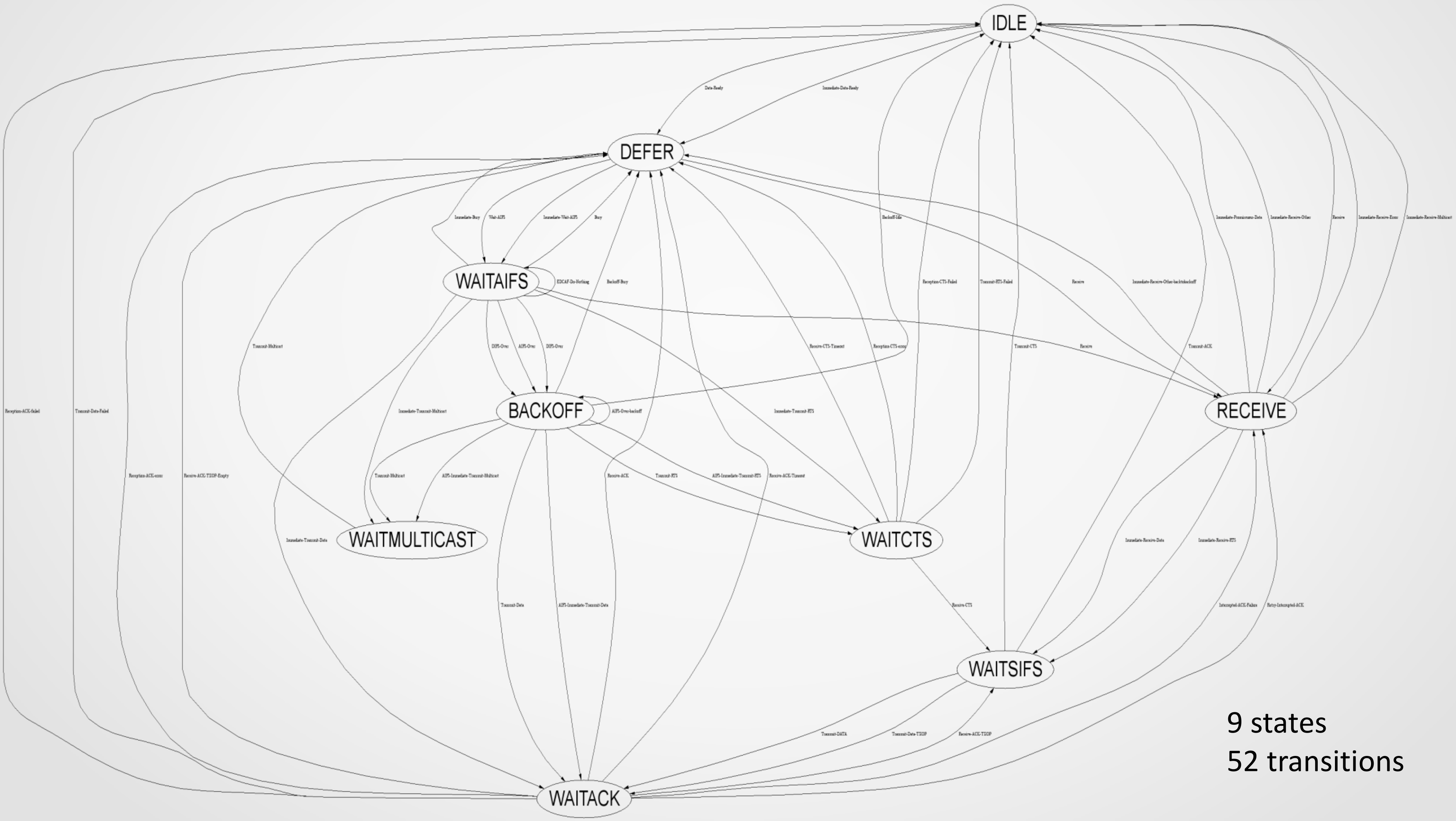
Problems With the Current Implementation

- 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

State Machine – Why?

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

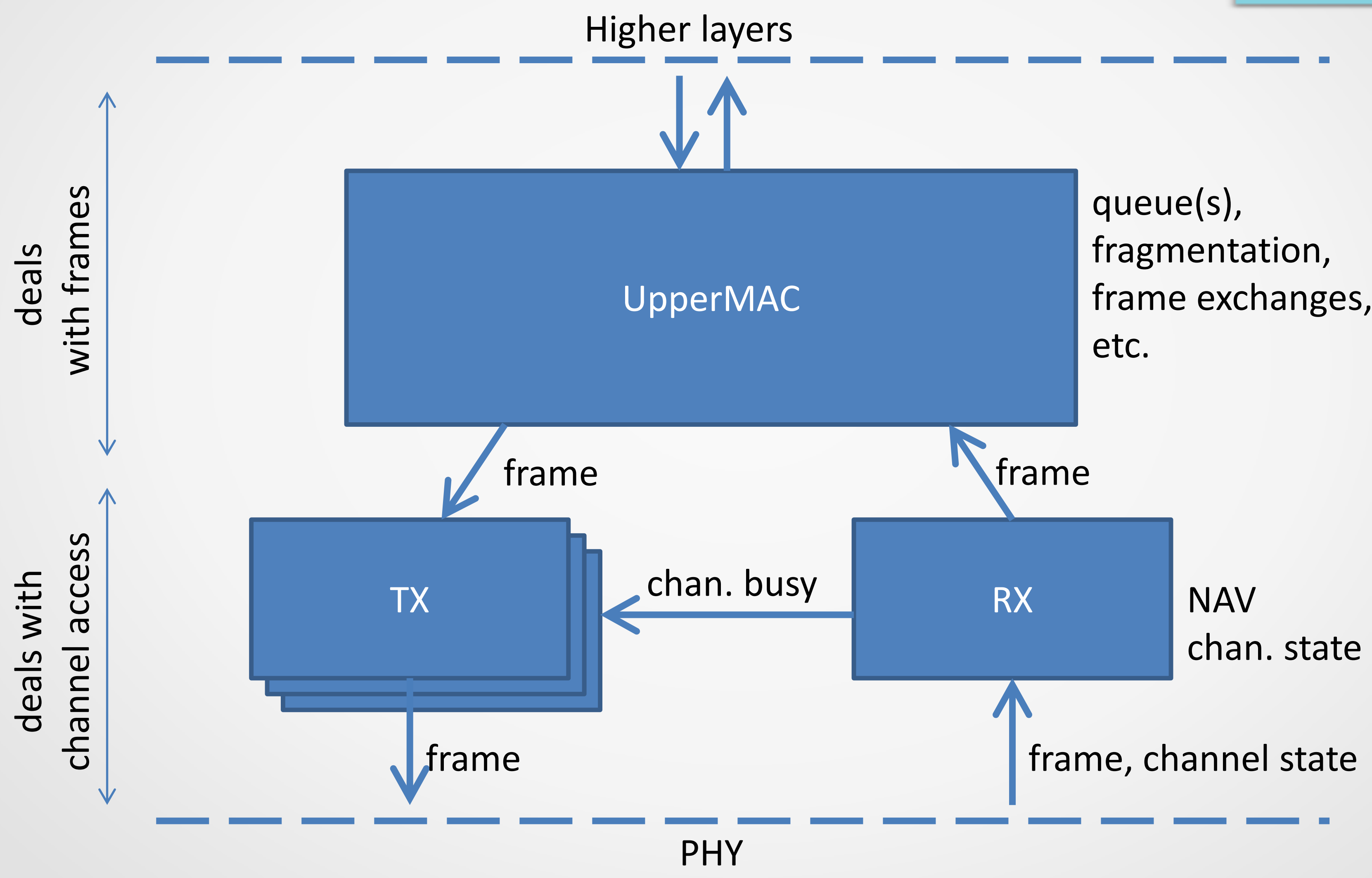
We tried to refactor, really tried...

But it's time for a reboot

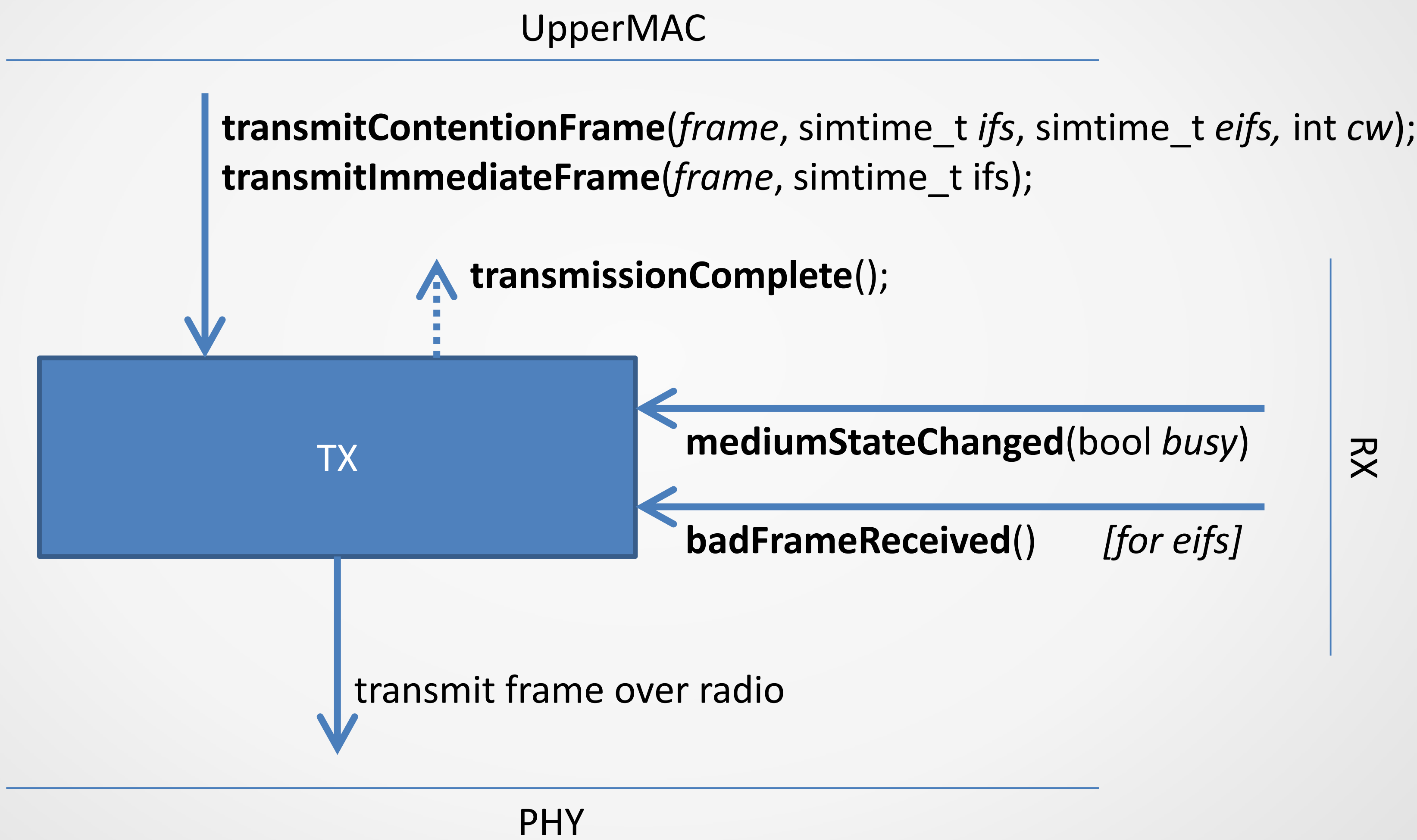
New MAC – Key Ideas

- *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 Process: Interface

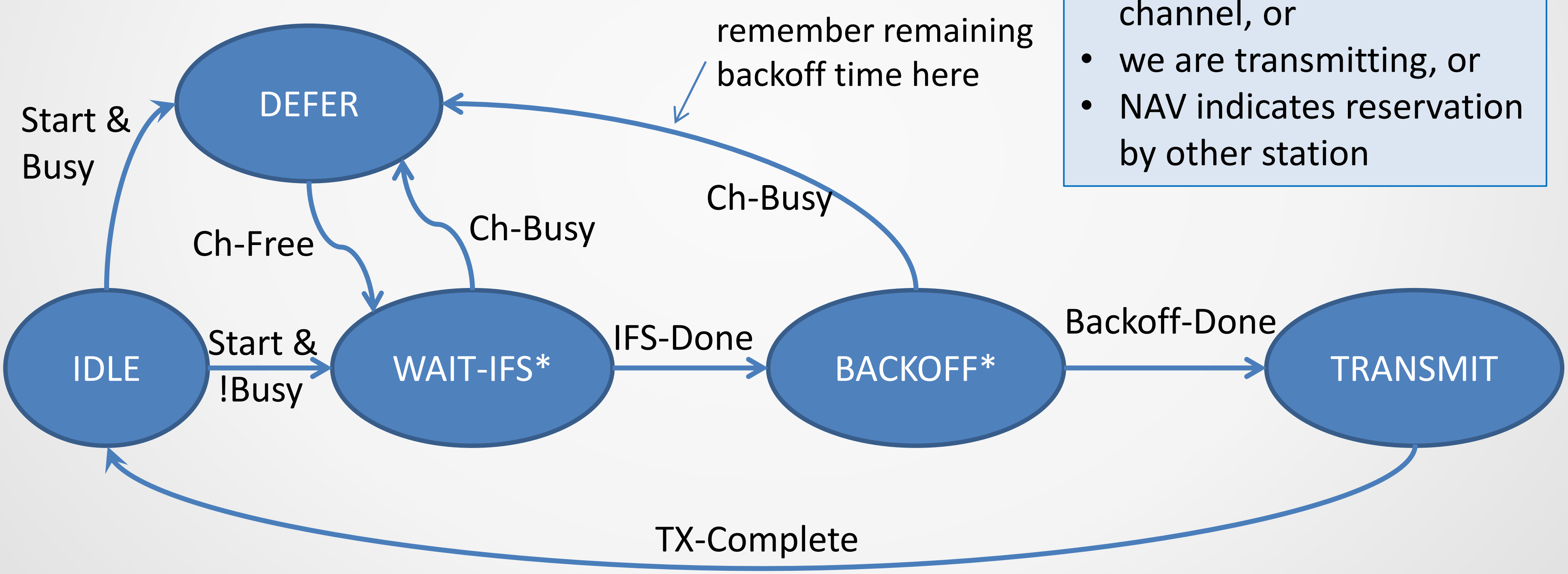


TX Process State Machine

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

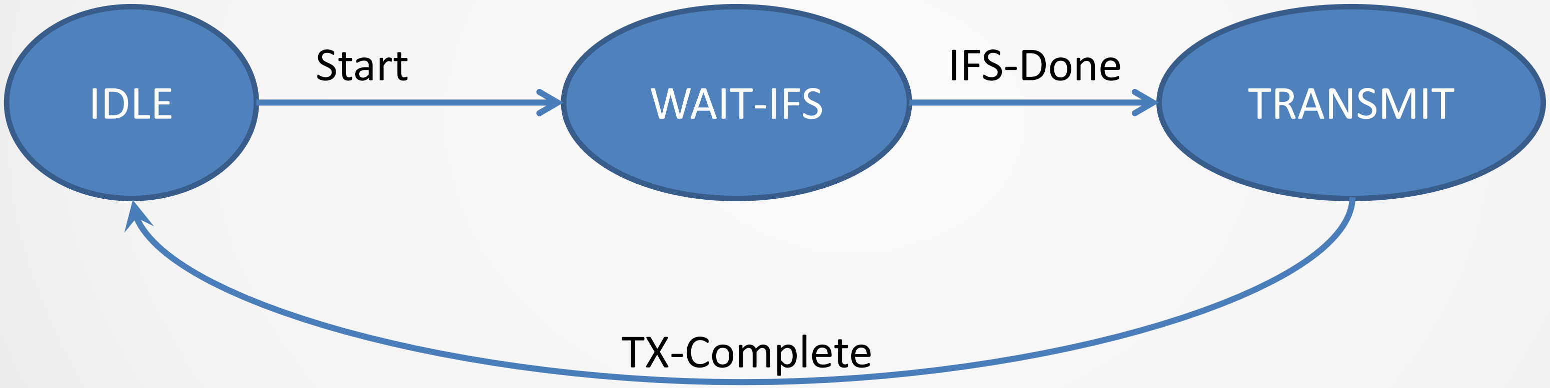


* omitted detail: switch to EIFS on reception of frame with bad checksum, and back on correct frame

TX Process: Immediate Frames

transmitImmediateFrame(*frame, ifs*)

- used e.g. for ACK, CTS, immediate BA, back-to-back data frames, etc.
- no contention



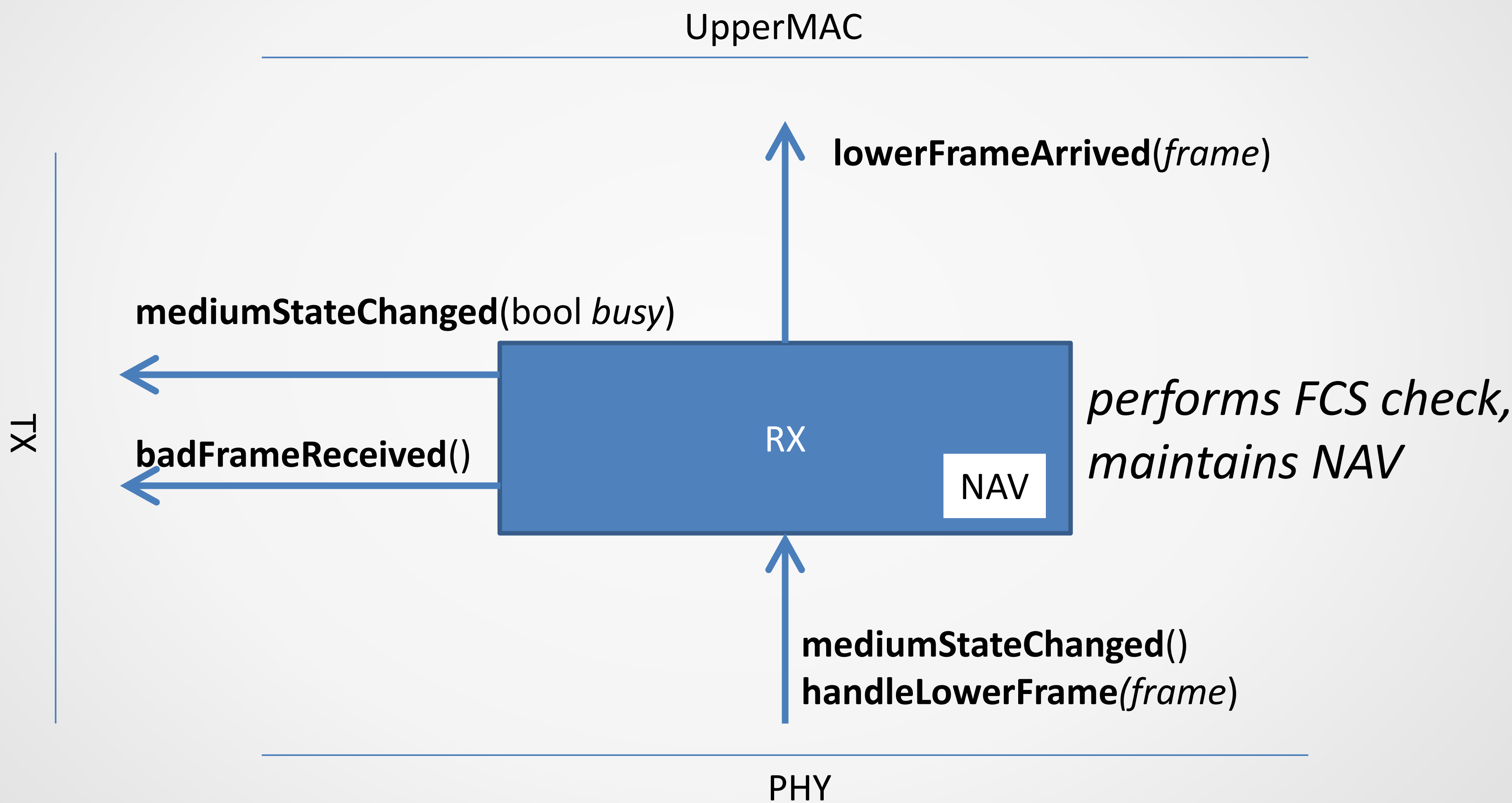
TX Process State Machine

- 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.

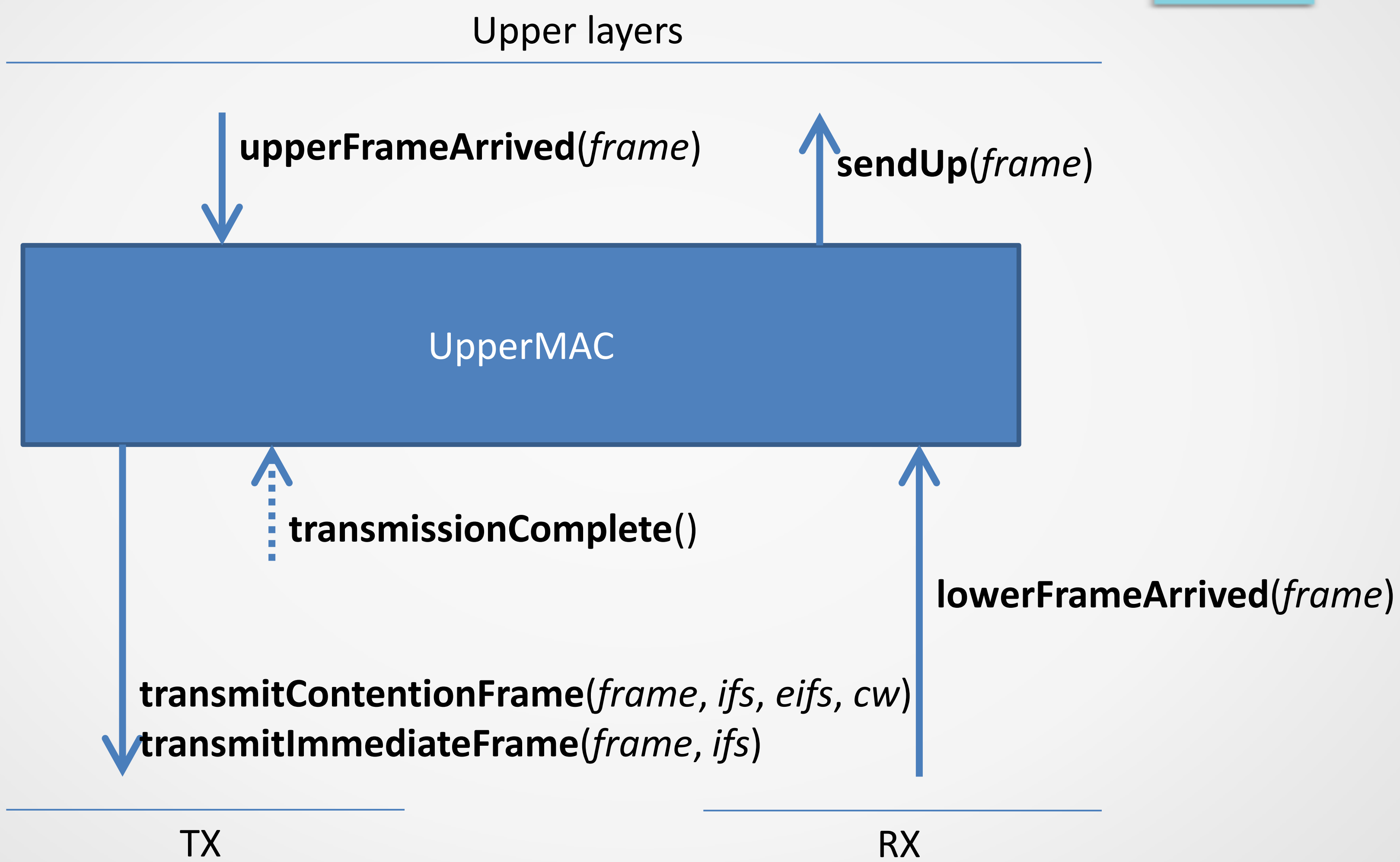
    <u>Today, no longer!</u> TXOP, Block ACK sequences, reverse direction frame exchange, etc...

    So: we want to take the complexity somewhere else
  - EDCA: just create 4 instances of *TX*

RX Process: Interface



UpperMAC: Interface



UpperMAC

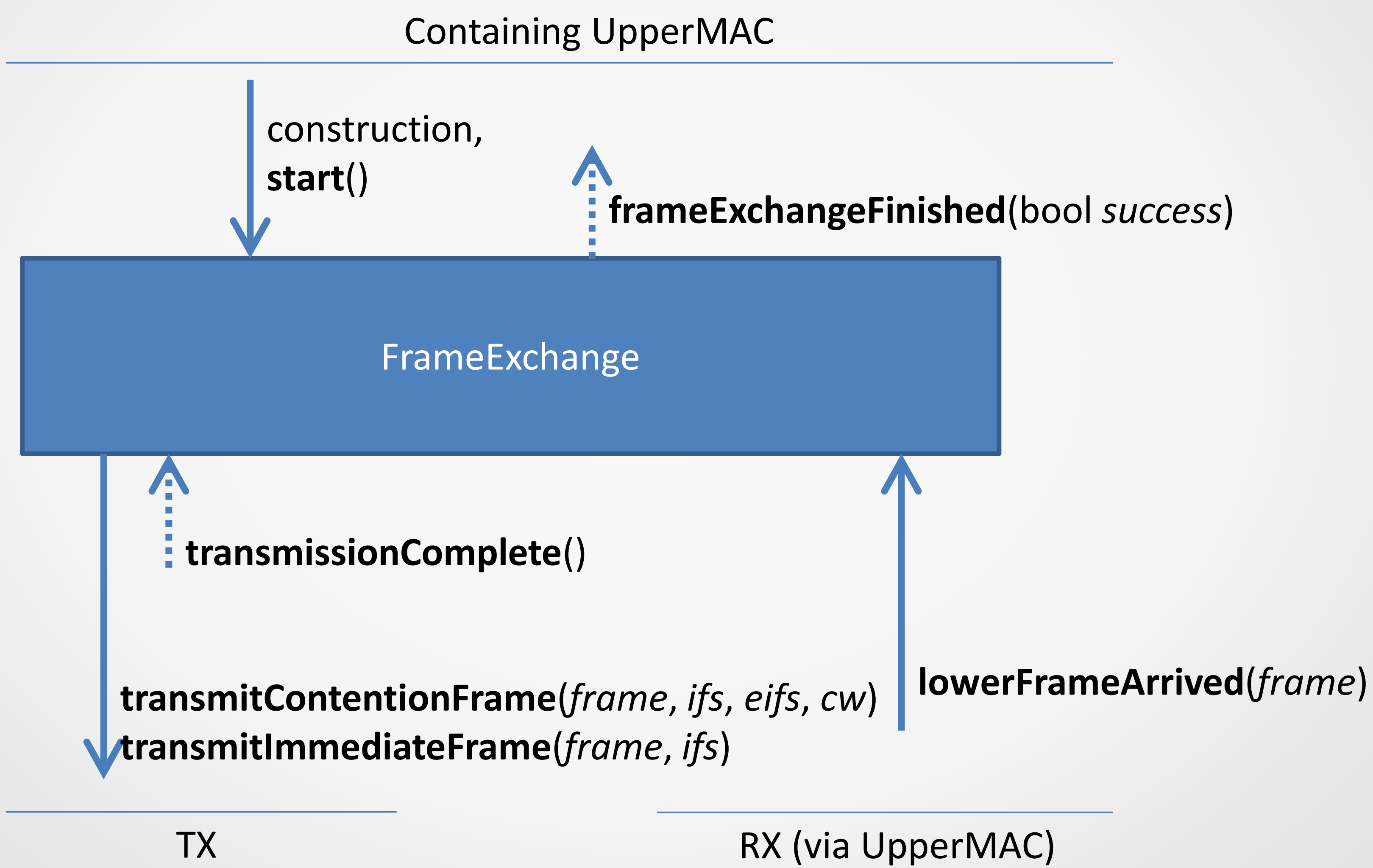
- Deals with exchanging frames
- Doesn’t need to care about channel access
  - reduces complexity!
- REPLACEABLE! May have simple, advanced and experimental variants
  - *80211b/g, 80211e, 80211n, experimental1, experimental2*, etc.
- May be modular in itself (see next slides)

Frame Exchanges

Frame exchanges are...

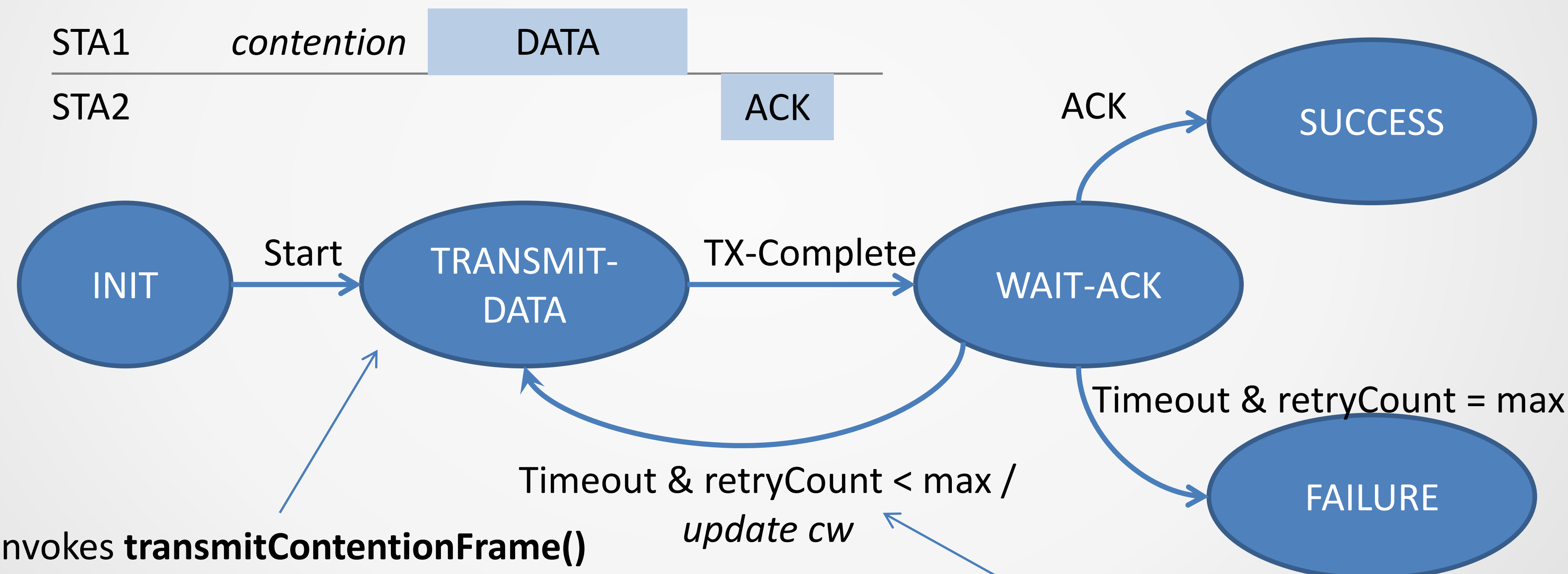
- C++ classes, used as building block for UpperMAC
- Created dynamically in UpperMAC as response to incoming frames or possibly other events
- Composable (?)
- Examples:
  - Data *ACK*
  - RTS *CTS* Data *Ack*
  - RTS *CTS* Data Data Data BAR *BA*
  - Reverse direction frame exchange
  - May map to one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

Frame Exchange: Interface



Implementation as State Machine

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



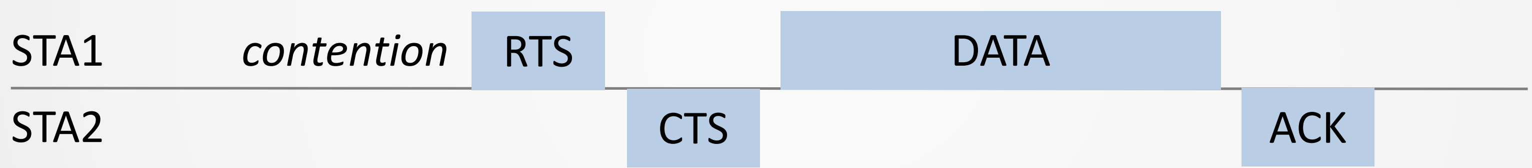
- invokes **transmitContentionFrame()**
- frames that arrive during IFS and backoff are processed separately by UpperMAC (ACKed, etc)

Exponential backoff procedure is here!

Step-Based Frame Exchanges

- Frame exchange classes allow for a concise and natural mapping of protocol to code

  Example: RTS+CTS+Data+ACK exchange:



  - 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

Step-Based Frame Exchange

```
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;
    }
}
```

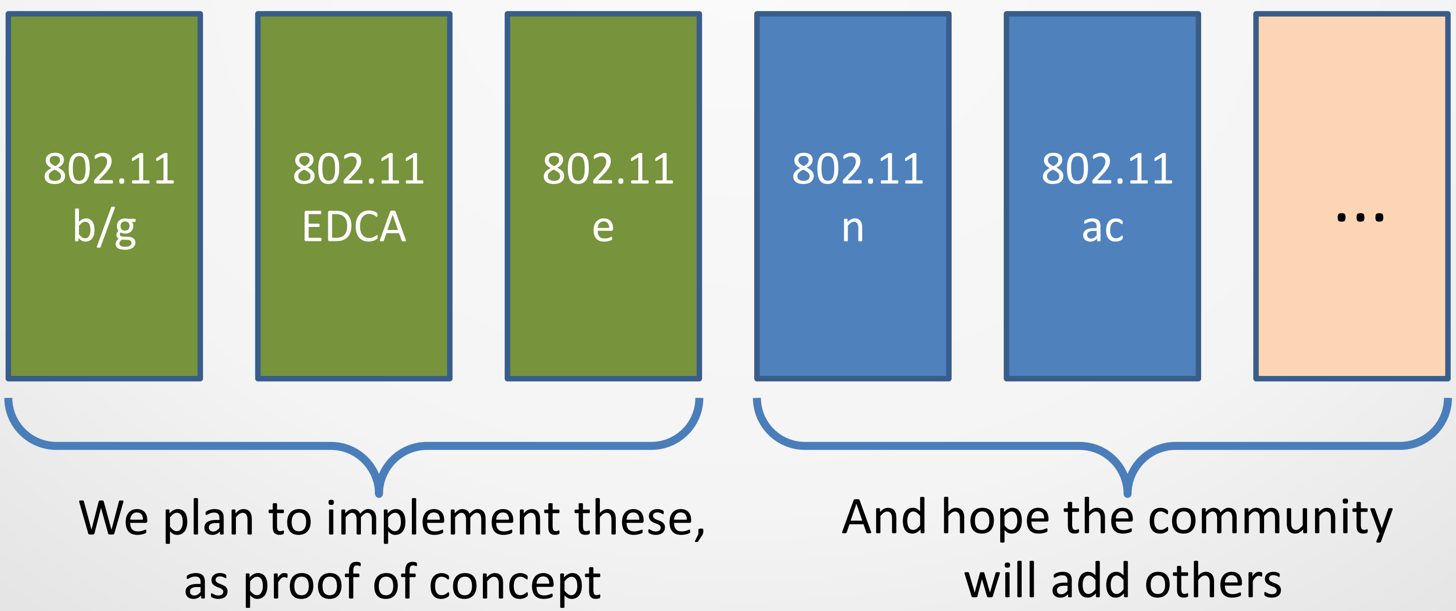
Further Componentization Possibilities

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 <u>multiple UpperMACs</u> of increasing complexity



What do you think?
Let’s discuss it!