Frame delineation behavior of H.264 in libwebrtc

We've encountered an issue where incomplete frames can sometimes be forwarded from the `PacketBuffer` to the `FrameBuffer` due to incorrect frame delineation. The issue (as far as we can tell) is in the `PacketBuffer::FindFrames` function in packet_buffer.cc.

From our understanding, the following lines in `PacketBuffer::FindFrames` (lines 271-277) will attempt to find continuous packets that start with a `frame_begin` (for H.264 this is a `S` bit set to true) and end with `frame_end`.

```
        for (size_t i = 0; i < size_ && PotentialNewFrame(seq_num); ++i) {
            size_t index = seq_num % size_;
            sequence_buffer_[index].continuous = true;

            // If all packets of the frame is continuous, find the first packet
of the
            // frame and create an RtpFrameObject.
            if (sequence_buffer_[index].frame_end) {
```

In the above loop, we check to see if a packet could be a part of a frame to be forwarded with a call to `PotentialNewFrame()`. This will, in general, return true if the packet has a `frame_begin` or if we have a continuous sequence of packets before this packet that starts with a `frame_begin`. Then, if the given packet also is the end of a frame (`frame_end`), we attempt to then use this frame.

One issue worth mentioning is that the `frame_begin` in H.264 is sometimes incorrect, particularly for an H.264 frame with multiple slices, as the `S` bit is set on the first packet of a slice, not necessarily the first packet of a given frame. For example, let's say there's a frame with 6 packets across 2 slices (slice 1: packets 1-3, slice 2: packets 4-6). Packet 1 and 4 would both have a `S` bit set to true, and both would have `frame_begin` set to true. With all the above lines of code, we could potentially attempt to use packets 4-6 as a frame, instead of all the packets 1-6 that actually make up the frame.

After libwebrtc finds a packet with `frame_end`, it will then search backwards and (for VP8) will collect packets until it hits a packet with `frame_begin`, and then forward that as a complete frame (lines 302-303).

```
        if (!is_h264 && sequence_buffer_[start_index].frame_begin)
          break;
```

On the other hand, for H.264, there's a special workaround (lines 338-342) of using timestamp to address the issue of `frame_begin` not always being the beginning of a frame.

```
if (is_h264 &&
    (!sequence_buffer_[start_index].used ||
     data_buffer_[start_index].timestamp != frame_timestamp)) {
  break;
}
```

Issue of incorrect H.264 frame delineation under lossy network conditions

This above workaround is designed not to forward a frame starting with a `frame_begin` (since it can be the start of a slice), but rather to forward all packets with the same timestamp as a frame. This approach does work as long as there is no packet loss, and packets are coming in sequential order. However, as soon as we have missing packets, we can run into some serious issues.

For example, in the earlier example, if we're using packets 4-6 as a frame, but packets 1-3 are present in the `packet_buffer`, packets 1-6 will all have the same timestamp and will all be correctly forwarded as a frame. However, if we're missing packet 2, then when we use packets 4-6 as a frame, we will try to find packets with the same timestamp (going backwards in sequence number). We'll find packet 3, but not find packet 2 since it's missing, so we end up forwarding packets 3-6 as an incomplete frame (which doesn't contain packets 1 and 2).

Our proposal for H.264 frame delineation

We've tried to fix this workaround ourselves by trying to use `first_mb_in_slice` to more accurately determine the first packet of a given frame (following the guidance from https://bugs.chromium.org/p/webrtc/issues/detail?id=7106). From what we can see, a packet with `first_mb_in_slice == 0` and `S bit == true` will always be the first packet of a frame's slice packets. However, there might be `aud/sps/pps/sei/vui` packets in front, so we don't believe using `first_mb_in_slice == 0` and `S bit == true` is an effective approach to resolve the issue.

Instead we've implemented a simple (and perhaps a naive workaround) that utilizes the `frame_end` of the previous frame. As long as media packets' sequence numbers are continuous (which is not the case for ULPFEC, but true for FlexFEC), we can make the assumption that the `seq_num` of the first packet of the next frame is just the `seq_num` of the last packet of the current frame plus one. This actually seems to work well for us, and can handle packet loss well. The below code sample provides how we could modify `PotentialNewFrame()` (lines 242-266) to accomplish this.

```
bool PacketBuffer::PotentialNewFrame(uint16_t seq_num) const {
    size_t index = seq_num % size_;
    int prev_index = index > 0 ? index - 1 : size_ - 1;

    if (!sequence_buffer_[index].used)
        return false;
    if (sequence_buffer_[index].seq_num != seq_num)
        return false;
    if (sequence_buffer_[index].frame_created)
        return false;
    if (sequence_buffer_[index].frame_begin)
        return true;
+   if (sequence_buffer_[prev_index].seq_num !=
+       static_cast<uint16_t>(sequence_buffer_[index].seq_num - 1)) {
+       return false;
+   }
+   if (sequence_buffer_[prev_index].frame_end) {
+       return true;
+   }
    if (!sequence_buffer_[prev_index].used)
        return false;
    if (sequence_buffer_[prev_index].frame_created)
        return false;
-   if (sequence_buffer_[prev_index].seq_num !=
-       static_cast<uint16_t>(sequence_buffer_[index].seq_num - 1)) {
-       return false;
-   }
    if (sequence_buffer_[prev_index].continuous)
        return true;
    return false;
}
```

This is something we would like to provide a patch soon for, and we'd love to get the discussion started.