Wallet Connect Messaging

Making Chunks

Let's assume we are sending message M from wallet A to wallet B. (A: sender, B: receiver).

Here M is in the form of a string. Let ‘x’ be the maximum message size that can be sent through Wallet Connect. x = 2000 bytes.

Our message M can have a maximum size of 12 Kb (12 * 1024 bytes).

So, we need to break M into smaller chunks and send the message chunk-wise. Later at the receiver’s end, the original message should be reconstructed by merging all chunks together. Each chunk should be associated with a unique identifier.

Total number of chunks $(n) = \lceil \frac{M.size()}{x-100} \rceil$

Here I am saving 100 bytes per chunk for its metadata, which would basically store its ID number.

A basic pseudo-code of how the original message will be divided into chunks. Here M is a string, and M[i : j] represents the sub-string of M from index i to j-1. (i: inclusive, j: exclusive).

```
chunk_size = x # x around 1800 - 1900 characters
message = "1e5c1fa7425e73043362938b9824" # generate from 2-party lib
total_chunks = ceil(len(message) / chunk_size)

to_send = {}

for ID from 1 to total_chunks:
  i = (ID - 1) * chunk_size
  j = min(i + chunk_size, len(message))
  data = M[i:j]
  to_send[ID] = {ID, data} # here first element of pair contains the unique id of the chunk
```

Before sending the chunks, we need to transfer total_chunks, so the receiver knows in advance, how many chunks it has to receive.

Ensuring Reliability

To ensure reliable communication over this unreliable channel, we will utilise Automatic Repeat Request (ARQ) mechanism. It provides re-transmission of lost packets to ensure reliable data transmission. Later we can optimise it more.

Algorithm:

- Assume that A is currently sending chunk `i`. After sending it to B, a timer will start on A's side. If A doesn't receive an acknowledgement from B before the timeout occurs, it will re-transmit the same chunk. Else it would move to chunk id `i+1`.
- B should continuously listen to messages from A. If it receives a chunk, it saves the message in its dictionary at index of chunk ID and send this chunk ID as ACK. Else, it does nothing.

SENDER SIDE

1. Initially it will send the receiver, the total number of chunks, it is planning to send. Let this be represented by variable `total`.

2. Rough sender side pseudo-code:

```
counter = 1
while (counter <= total):
  packet = to_send[counter]
  WC_send_api(packet)
  timer_start()

  while not timeout:
    if ACKNOWLEDGEMENT_received:  # basically receive a message from B and check if message == counter
      counter += 1
      break
    else if timeout:
      break
```

   This will make sure, that the sender will keep sending the same chunk, unless it receives an ACK from the receiver. In case it receives ACK from the other side, it will continue to send next chunk.

   Here ACK will be the chunk ID that receiver receives.

   Since, on average it was taking 500 ms for a message to reach the receiver from sender, it would be safe to take `timeout` as around 4*500 ms (negotiable). By incorporating the timeout mechanism, the sender can handle scenarios where an ACK is lost in transit, ensuring reliable delivery of packets.

RECEIVER SIDE

1. It already has the information about total number of packets it has to receive. `total` variable captures it.

2. Receiver side pseudo-code:

```
expected_ID = 1
received_chunks = {}

while (received_chunk.size() <= total):
  packet = WC_receive_api() # remember this packet was a pair of ID and message string
  received_chunks[packet.first] = packet.second
  WC_send_api(packet.first) # we are sending the packed ID as the acknowledgement to sender
```

3. Later receiver can reconstruct the complete message, by concatenating all the parts together.

> We can try to create some wrapper function around the above protocol, and then use that code for messaging.