moq
Internet-Draft
Intended status: Standards Track
Expires: 8 January 2026

C. Jennings
S. Nandakumar
Cisco

7 July 2025

P2P Chat with MoQ
draft-jennings-moq-p2p-00

Abstract

This protocol is designed for small groups of users sending relatively small messages. It is optimized for the assumption that a node will come online and collect all messages at least once every 15 days and that some nodes that act as mirrors will be online most of the time but may come and go over long periods of time.

Table of Contents

1. Overview

This protocol is heavily based on Media over QUIC Transport (MoQT) and assumes the reader is familiar with the terminology and basic operations of the MOQT Pub / Sub protocol.

Each client of this system is referred to as a node and publishes a few MOQT tracks. These include:

* Info: The Info track provides basic information about the node and its identity (NodeID). The node information carries a flag to indicate if it expects to be a stable mirror.

* Others: The Others track provides identification information of the other nodes that a given node knows about. This is used to help recently joined nodes discover the information about the other nodes in the network.

* Messages: The Messages track is used to publish Chat messages by this node.

* Mirror(s): The Mirror track is used to publish Chat messages that this node knows about but were received from other nodes within a chat session. There is one such track per chat session.

* Manifest: The Manifest track provides information for where this node publishes audio and video for a live chat.

When a node comes online, it needs to have a seed list of one or more other nodes in the network. It will retrieve the NodeIDs of other nodes from the Others MOQT track and recursively retrieve these lists until it has discovered all the nodes for a given chat session. It can use SUBSCRIBE_NAMESPACE to where Info tracks are published to discover new nodes that come online.

To receive new messages, the node will subscribe to all Messages tracks as well as fetch any older messages that the node does not have. Since some of these Nodes may be offline and may not be cached in relays, this might not get all of the prior messages in the chat.

Each message has a unique identifier called the MessageID that is formed by the tuple <ChatID, NodeID, and MessageCount>. The ChatID is a unique identifier for this chat that is typically formed by hashing the name of the chat. The NodeID uniquely identifies the node within this chat and the MessageCount is an incrementing counter kept by each NodeID that increments by one each time that node sends a new message.

Messages, in addition to the content of the messages, have some extra data that helps render the user interfaces that display the messages. This includes:

* PreviousMessages: Typically this is a single message that was the logical message that came before this message, but in some cases where the network was partitioned or in certain race conditions, there can be more than one previous message.

* Timestamp: The timestamp of message creation according to the clock on the node that created the message.

* Replaces: Indicates if this message replaces a previous message. Messages cannot be changed but new messages can replace old messages.

* Topic: indicates if this message is part of a thread in a chat.

A given node can look at all the messages and work backwards through the Previous Messages arrays in each message to find the MessageID of messages it does not have but that are part of the chat. If a node

sees another node come online that it is missing messages from, it should try to get those messages. The node can also "randomly" pick another node and try to retrieve the missing message from the other node's Mirror. There are notes later on how the random selection is done to maximize the odds of a cacheable hit in the relays when multiple nodes are doing this.

A node may periodically republish all of the recent messages in its message track to have them cached by the relays.

2. Details

2.1. Node ID

Each publisher participating in a given group is called a node. Each node picks its own Node ID. The Node ID is a random 62-bit integer that is stable over time for a given user/device combination. The node needs to store this in persistent storage.

2.2. Chat ID

Each chat has a unique 62-bit identifier. It is recommended that this is generated from the lower 62 bits of the SHA1 hash of a unique DNS name combined with a subdomain of the chat name. For example, for a chat called "water cooler" on a server called example.com, the chatID could be generated from the hash of "water_cooler.example.com".

2.3. Node Discovery

The goal of Node Discovery is for all the nodes to discover the Node IDs of all the other nodes including nodes that are currently offline.

When a node starts, it will have a seed list that consists of a set of Node IDs that it received from a configuration or obtained when running previously. This list may be empty.

Each time a node learns of a new node, it publishes an object "other node" on the Others track. This object has a Group ID one greater than the previous "other node" object and an Object ID of 0.

The Others track is published on a namespace formed by tuple ("moq://p2pchat.moq.arpa.org", "v1", {chat_id}, "nodes" ,{node_id}) and with a track name of "others".

The "other node" object contains the NodeID of the newly learnt node, encoded as a 64 bit integer.

Each node MUST announce its own namespace, that includes its Node ID, with the following tuples ("moq://p2pchat.moq.arpa.org", "v1", {chat_id}, "nodes", {node_id}).

Nodes intending to learn about other nodes do so by subscribing to namespace announcements, via sending SUBSCRIBE_ANNOUNCES to the namespace prefix containing the following tuples ("moq://p2pchat.moq.arpa.org", "v1", {chat_id}, "nodes").

When a node learns of a new node's namespace, it MAY proceed to subscribe to the "Others" track under that namespace. This results in the current node learning further new nodes as well as fetch any older objects on that track. Both of these can result in the node learning about the existence of other nodes. Each node SHOULD subscribe to the Others track of about 10 different nodes that are currently online. Nodes that have information indicating they are likely to stay online most of the time SHOULD be preferred. Nodes that are meant to be online all the time MAY subscribe to the Others track from many more or all of the nodes discovered.

Each node needs to keep track in persistent storage of the Node ID of any nodes it is aware of as well as the Group ID of the last group it has retrieved from that node's Others track. As part of subscribing to such nodes' Others track, any missing groups should be retrieved by issuing FETCH API.

3. Message Retrieval

Nodes interested in receiving messages subscribe to Messages tracks of all other nodes discovered via procedures in Section 2.3. SUBSCRIBE is done to the namespace ("moq://p2pchat.moq.arpa.org", "v1", {chat_id}, "nodes" , {node_id}) with a track name of "messages". Such a subscribe will fail for any nodes that are offline. Each node keeps track of what is the largest group in the track received from each other node. The largest group returned in the subscribe, can tell if it is missing any messages from this node. If messages are missing, it tries to FETCH them.

3.1. Message Graph

Each message has pointers to the messages that came immediately before it in the conversation. In most cases there is only one previous message, but in some cases where the network partitioned and later reconverged, there can be more than one previous message. It can also happen when two nodes sent a new message at roughly the same time. The pointer has the NodeID of the node that created the message along with GroupID and ObjectID. This creates a partial ordering of all messages.

A node can traverse this graph and find "last" messages. A last message is defined as having no messages that come "after" that message. "After" in this context means that there is some message B that has pointers to message A as messages before it.

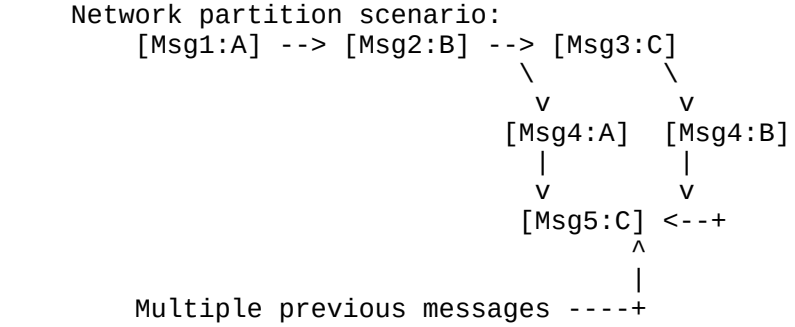
Traversing the graph also reveals any missing messages. The node SHOULD try to fetch these messages. Even if the node that produced the message is offline, a caching relay might have the information.

Message Graph Example:

Normal flow (linear):

```
[Msg1:A] --> [Msg2:B] --> [Msg3:C] --> [Msg4:A]
```

Network partition scenario:



Legend:

- [MsgN:X] = Message N from Node X
- --> = Points to previous message
- "Last" messages are those with no arrows pointing away from them

4. Message Container Format

Messages in the MoQ P2P chat system use a generic container format that provides message metadata and content structure while remaining transport-agnostic and encryption-scheme independent.

The message container consists of two main categories of information: * Message behavior and metadata fields * Content body parts and associated parameters

The proposed container format is adapted from the MIMI Content [MIMI] specification.

Message Container Structure (CDDL):

```
p2pMessage = [
  messageId: MessageId,                 ; {1}
  timestamp: uint .size 8,              ; {2}
  nodeId: NodeId,                       ; {3}
  chatId: ChatId,                       ; {4}
  previousMessages: [* MessageRef],     ; {5}
  replaces: null / MessageId,           ; {6}
  topicId: bstr,                        ; {7}
  expires: null / Expiration,           ; {8}
  extensions: {* ExtensionKey => any},  ; {9}
  contentBody: ContentBody              ; {10}
]

MessageId = bstr .size 32              ; Unique message identifier
NodeId = uint .size 8                  ; Node identifier
ChatId = uint .size 8                  ; Chat identifier
MessageRef = [NodeId, MessageId]       ; Reference to previous message

Expiration = [
  relative: bool,                      ; true=relative, false=absolute
  time: uint .size 4                   ; seconds (relative) or unix epoch
]

ExtensionKey = int / tstr .size (1..255)  ; Extension identifier
```

4.1. Message Metadata Fields

The messageId {1} is a unique 32-byte identifier for this message, formed by the tuple (chatId, nodeId, messageCount) as described in the Overview section.

The timestamp {2} field contains the message creation time according to the sending node’s clock, encoded as seconds since the Unix epoch.

The nodeId {3} and chatId {4} fields identify the sending node and target chat respectively.

The previousMessages {5} field contains an array of references to messages that logically precede this message in the conversation, enabling the message graph structure described in the Message Graph section.

The replaces {6} field indicates this message replaces a previous message with the specified messageId. Used for message editing and deletion.

The topicId {7} field groups related messages into conversation threads. A zero-length topicId indicates no specific thread grouping.

The expires {8} field provides expiration hint for message lifecycle management.

The extensions {9} field allows for protocol extensions and additional metadata.

4.2. Content Body Structure

ContentBody Structure (CDDL):

```
ContentBody = [
  disposition: Disposition,             ; {11}
  language: tstr,                       ; {12}
  ContentVariant
]

ContentVariant = NullContent / SingleContent / MultiContent

NullContent = (
  type: 0                               ; No content (for deletions, etc.)
)

SingleContent = (
  type: 1,
  contentType: tstr,                    ; MIME type
  content: bstr                         ; Content data
)

MultiContent = (
  type: 2,
  semantics: ContentSemantics,          ; How to process multiple parts
  parts: [2* ContentBody]               ; Nested content parts
)

ContentSemantics = uint .size 1 .within (0..2)
; 0 = pickOne (receiver picks one part)
; 1 = oneByOne (all parts form single entity)
; 2 = processAll (process parts independently)

Disposition = uint .size 1 .within (0..7)
; 0 = unspecified, 1 = render, 2 = reaction
; 3 = profile, 4 = inline, 5 = icon
; 6 = attachment, 7 = session
```

5. Sending Messages

Each message is sent as Object ID 0 on the next group on the node’s Message track. The format of messages is as defined in Section 4.

5.1. Mirroring

Some nodes may choose to mirror all the messages they are aware of. These nodes indicate they can mirror in the "info object" published on the node’s Info track. Nodes that mirror SHOULD republish any received messages into a track ("moq://p2pchat.moq.arpa.org", "v1", {chat_id}, "nodes" , {node_id}) with a track name of "mirror_{other_node_id}".

When mirroring, the message object’s Group and Object IDs are kept intact and forwarded as-is. This can result in Group IDs having gaps or decreasing, and subscribers MUST ensure to group the messages under the node corresponding to the mirrored Node ID. Mirrors SHOULD discard messages older than 30 days. If a node is missing messages from a particular Node ID, it MAY use subscribe and fetch to try and get those messages from any of the mirrors.

6. Rendering Messages

6.1. Replaces

6.2. Topics

7. Info Track

8. Read Receipts

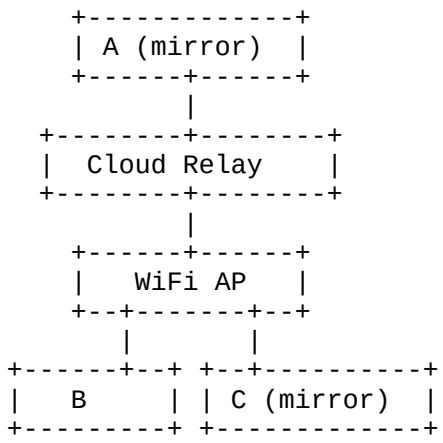
9. Example

This section describes a simple example of how the protocol works in practice.

A is connected to cloud relay. A and C are mirror nodes.

B and C connect to a common wifi AP that is connected to cloud relay.

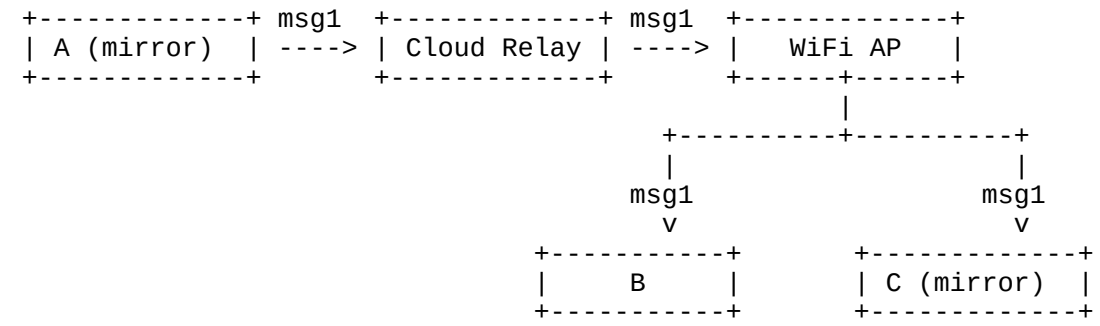
Initial Network Topology:



Legend: Nodes A & C are mirrors (cache messages from other nodes)

All are online. A sends 1. B & C get it as they are subscribed to A.

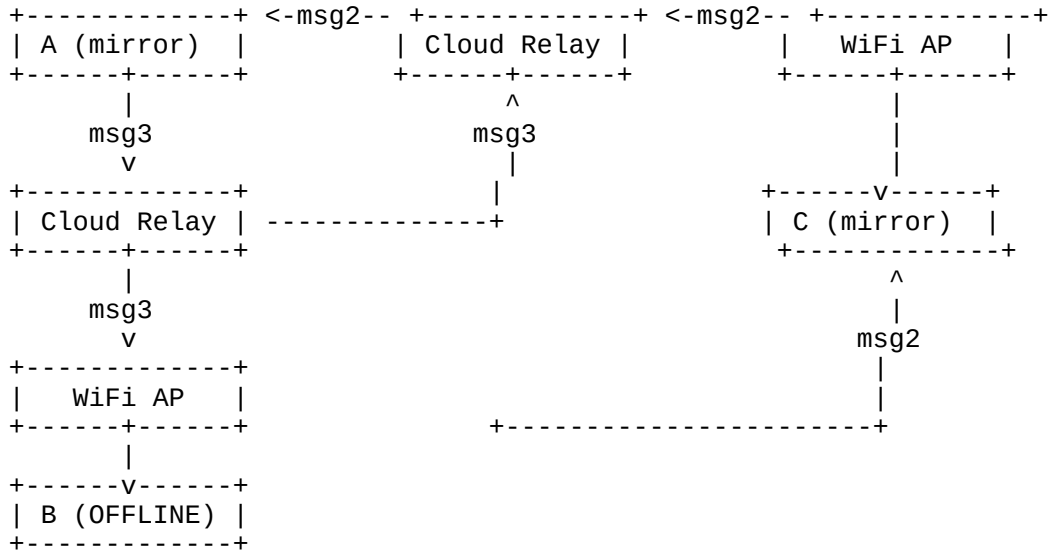
Message 1 Flow (All Online):



Result: A has [msg1], B has [msg1], C has [msg1]

B goes offline. C sends 2. A gets 2. A sends 3. C goes offline.

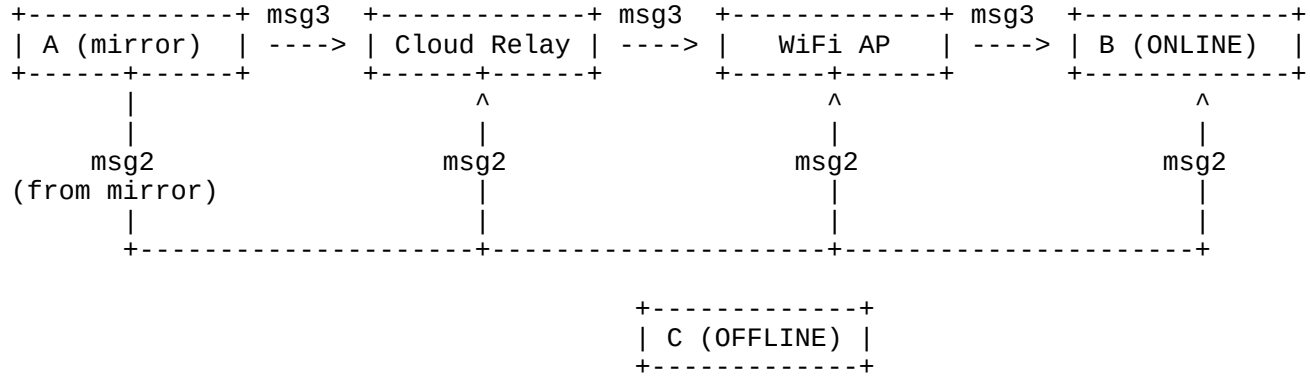
B Offline, Messages 2 & 3:



Result: A [msg1,2,3] | B [msg1] | C [msg1,2,3] -> offline

B comes online. It subscribes to A and gets 3. B realizes it is missing 2 and gets it from mirror track of C on A.

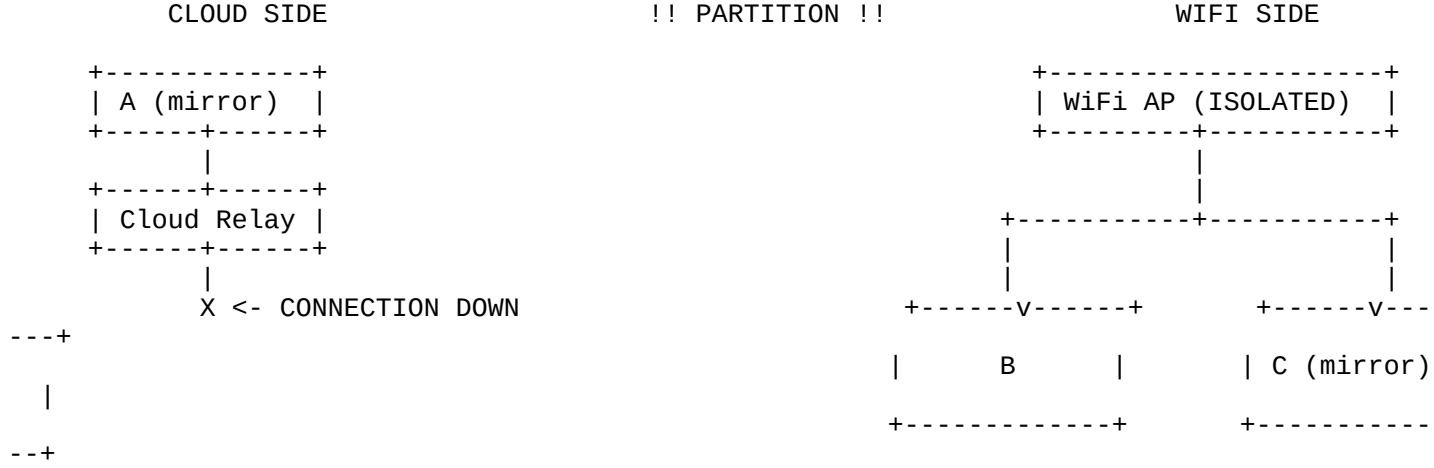
B Returns Online, Gets Missing Messages:



Result: A [msg1,2,3] | B [msg1,2,3] | C [msg1,2,3] offline

The connection between the wifi AP goes down.

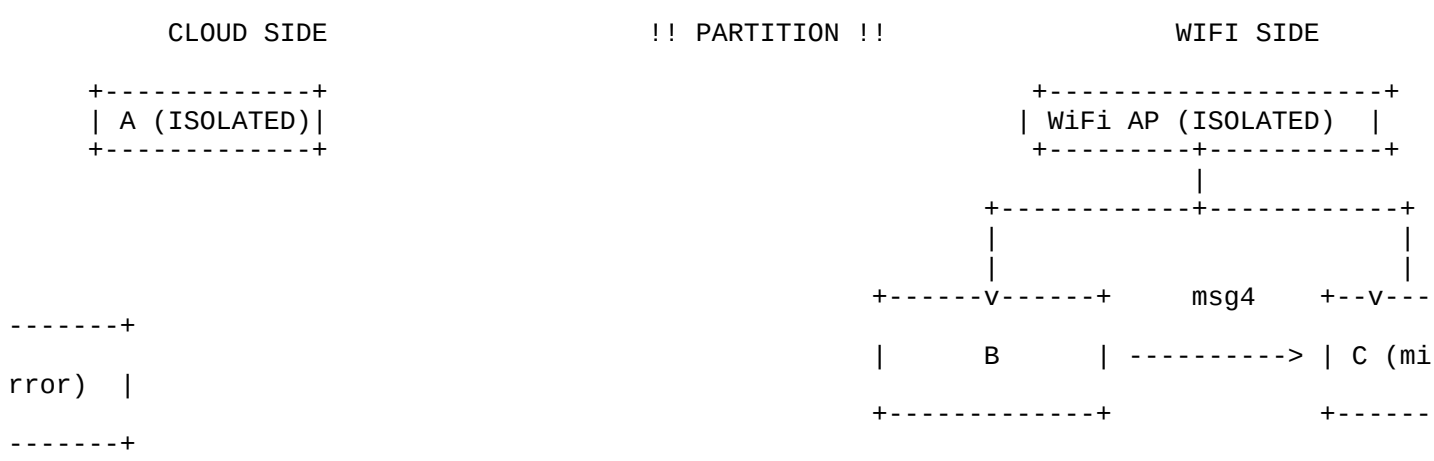
Network Partition (WiFi AP Connection Down):



Result: A isolated on cloud | B & C isolated on WiFi

B sends 4. C gets it but A does not as the network is partitioned.

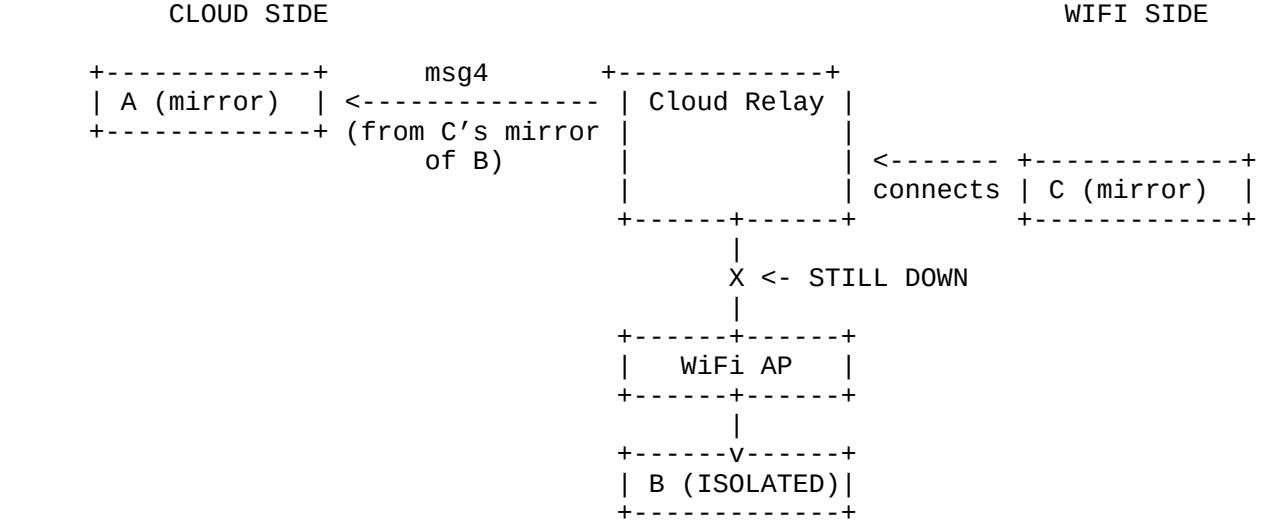
Message 4 During Partition:



Result: A [msg1,2,3] | B [msg1,2,3,4] | C [msg1,2,3,4]

C moves from being connected to the cloud relay instead of the AP. A sees C come online and gets 4 from C's mirror of B.

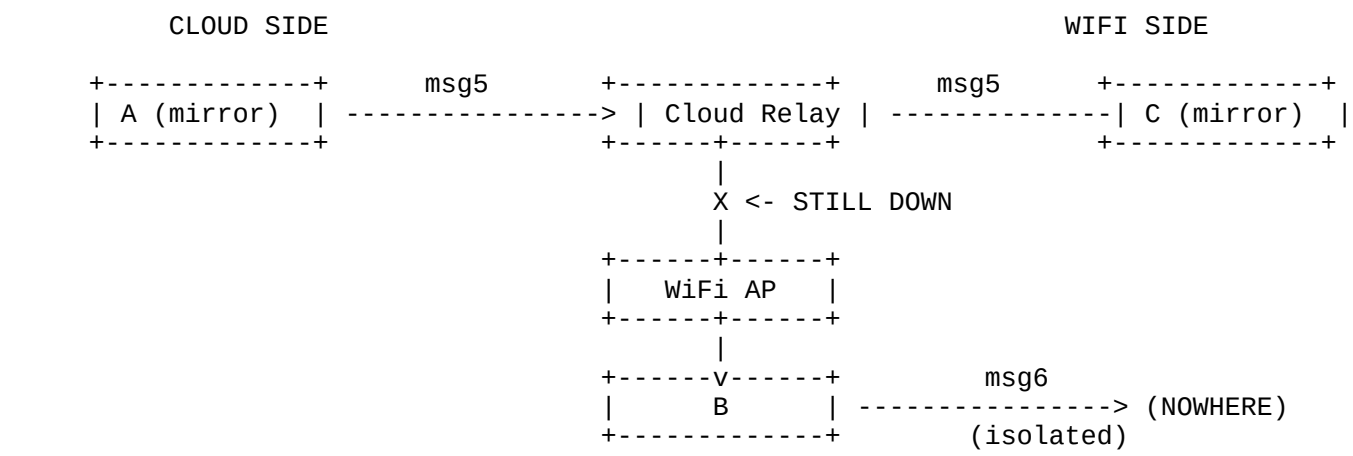
C Moves to Cloud Relay:



Result: A [msg1,2,3,4] | B [msg1,2,3,4] isolated | C [msg1,2,3,4]

A sends 5 and C gets it. B does not. B sends 6 and no one gets it.

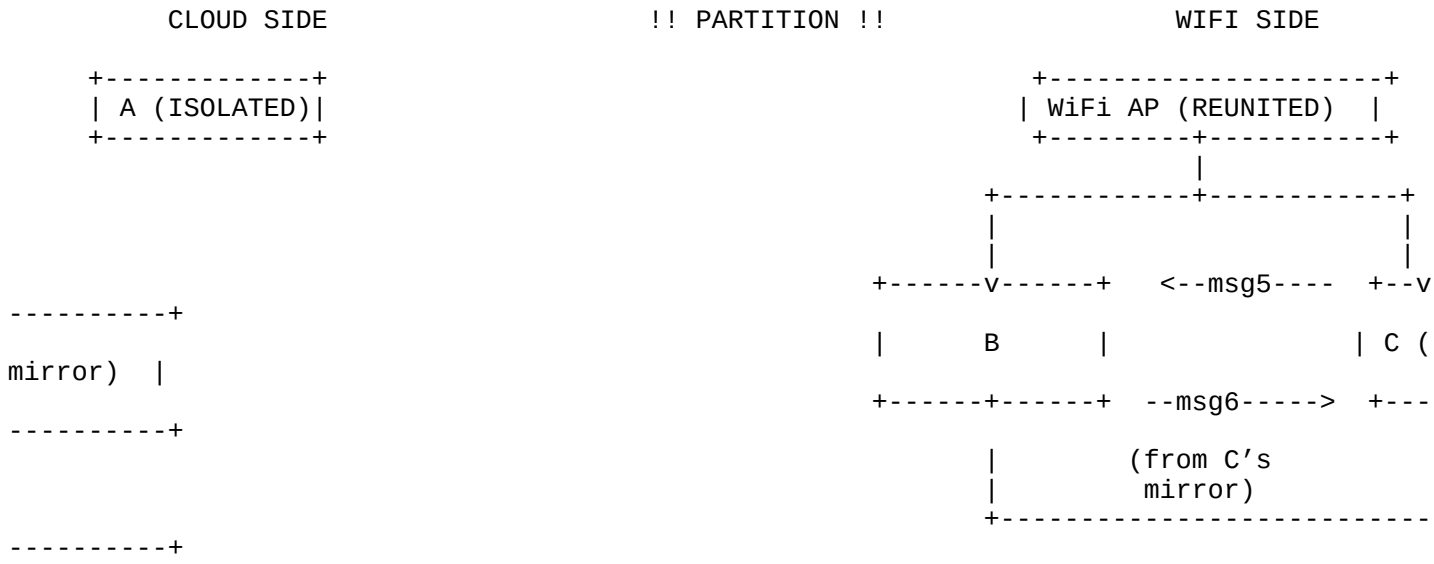
Messages 5 & 6 During Partition:



Result: A [msg1,2,3,4,5] | B [msg1,2,3,4,6] isolated | C [msg1,2,3,4,5]

C disconnects from cloud relay and moves back to wifi AP. C will get 6 from B and B will get 4 from C's mirror of A.

Final Synchronization (C Returns to WiFi):



Final Result: A [msg1,2,3,4,5] isolated | B [msg1,2,3,4,5,6] | C [msg1,2,3,4,5,6]

10. Reliability

Does this work in all cases ? No. Imagine A and B joined and exchanged some messages then went offline forever and the messages all expired on the caches, then C joined. C has no way to get the messages between A and B.

11. End to End Encryption

TBD - this is based on the MLS draft

12. References

12.1. Normative References

[MOQT] Nandakumar, S., Vasiliev, V., Swett, I., and A. Frindell, "Media over QUIC Transport", Work in Progress, Internet-Draft, draft-ietf-moq-transport-12, 23 June 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-moq-transport-12>.

12.2. Informative References

[MIMI] Mahy, R., "More Instant Messaging Interoperability (MIMI) message content", Work in Progress, Internet-Draft, draft-ietf-mimi-content-07, 7 July 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-mimi-content-07>.

Authors' Addresses

Cullen Jennings
Cisco
Email: fluffy@iii.ca

Suhas Nandakumar
Cisco
Email: snandaku@cisco.com