iconet documentation

Iconet Foundation

Mar 30, 2023

CONTENTS

This document outlines the proposed Iconet (interconnected networks) specification, which describes a minimal fallback mechanism to display and interact with content from natively unsupported social networks and clients.

This documentation is an early structural draft of a standard for interconnectivity. The standard recommendation is yet to be created through peer reviews and an open community process. Different parts can potentially be subject to radical change in the future.

If you have any questions, feedback, or ideas, don't hesitate to contact or open an issue / pull request on codeberg.

CHAPTER

ONE

WHAT'S HERE?

- *The Specification of..*
  - data required to provide a presentation fallback or a packet translation (*Required Iconet Data for a Packet*)
  - manifests that contain the required data to create a fallback presentation or translation (*Interpreter Manifests*)
  - the usage of iframes to provide a packet presentation fallback or to translate a packet, as well as the communication flow between iframe and embedding application (*Fallback-iframes*)
  - methods and discussions on isolating iframes from the embedding application to prevent data leakage (*Iframe Permissions and Sandboxing*)
- *Challenges & Discussions* outlines discussions we faced during development and possible alternatives for the spec.
- *Glossary* explains the basic terms and components

CHAPTER

TWO

PROTOTYPES

To demonstrate and experiment with the functionality of the spec, we developed a prototype network and developed an extension to mastodon:

- Prototype Network A and
- Mastodon Fork

You can find all of our repositories on Codeberg.

CHAPTER

THREE

WEBSITE

See the Iconet Foundation Website for current blog updates, explanatory materials and ways to contribute.

SPONSORED BY THE

From September 2022 to February 2023 this project receives funding from the German Ministry of Education and Research.

3.1 Use Case

This page aims to give a better understanding off the general purpose of the project, aswell as by whom and how it should be adapted.

3.1.1 Summary

Interconnectivity for social networks aims to ensure that users of different networks based on different technologies are able to interact.

This is achieved by establishing a common fallback mechanism, both within different decentral networks, and across.

Pretext: Interoperability

There is a variety of open social networks based on protocols which are open source and therefore freely accessible. Many of those networks based on the same or a similar protocol, have chosen to federate, this means allowing a cross communication for users. This is called interoperability. A prominent example for such a protocol is ActivityPub which is implemented by 50+ Softwares and connects millions of people.

There are plenty of technologies and applications that wish to federate

Problem: Incompatibilities

The different applications for such protocols are developed by independent teams - each of them with varying intentions and goals for the functionality of their application. This creates a great diversity and ensures freedom of choice across the network. But also it often results in technical incompatibility, because different aspects of a protocol are used, different extensions are created, or entirely different protocols or tools are implemented in the first place. Due to those issues the general promise of open federated networks is not fully given across ends. For the uninitiated user those limits are not comprehensible.

While decentral systems are fully capable, developers of users applications often only implement subsets of possible formats and interaction methods. Therefore content by users off different applications often can not be presented.

Solution: Interconnectivity

In contrast to interoperability, which ensures broad functionality for networks, with interconnectivity we focus on the communication capability between explicitly incompatible systems. For many use cases, it is important that deeper functionality is fully mapped across all networks involved. Wherever this does not work interconnectivity ensures that communication is transmitted securely and presented properly. It serves as a minimal fallback mechanism, applications can use in addition to their core functionality.

Having a fallback in place, ensures communication.

3.1.2 Adoption

By whom and how is interconnectivity to be adopted? Here we have an overview:

Inner Use case of interconnectivity

Projects like the Fediverse, Matrix.org, SoLiD, BlueSky, XMPP each enable complex decentral datastructures. The inner use case is for those systems, to include the fallback mechanism into their internal packet exchanges.

If they adopt the fallback to their protocol, they gain the following advantages:

- End to end communication within the protocol is ensured.
- Applications can become more diverse, because innovative formats can be adopted by anyone right away.
- The eco system may evolve, because older versions stay compatible with newer ones.

To adopt the fallback internally, the following measures need to be taken:

- for any format, a interpreter of the native package needs to be provided. (HTML)
- for any client, the displaying of an fallback interpreter needs to be provided. (Java Script)
- for any packet, the information on where to find the corresponding interpreter needs to be added. (JSON)

The inner use case is necessary for decentral data exchanges, to avoid internal incompatibilities.

Outer Use case

Networks which adopted the fallback mechanism, can ensure communication across their network. The fallback mechanism then can be especially effective, if also fallback means of transportation are adopted. If e.g. from an endpoint in the fediverse a packet including fallback info can be delivered to an end point in Matrix, communication here can also be assured. This is the outer use case, to use the fallback to acchieve interconnectivity across technical barriers.

If networks adopt the fallback transport, they gain the following advantages:

- massivly extended connectivity for their users (in and out)
- organical distribution of communication technologies, because they know can peacefully coexist.
- a stable global communication network, available to anyone independent of technology.

To adopt the fallback externally, the following measures need to be taken (additional to the inner use case):

- adopt the fallback means of addressing, packeting, encryption as a second layer of transport.

The outer use case is the chance to bridge the gap between different communication technologies, for maximum independence of the users.

Closing statement

With interconnectivity, the former limitations of interoperability turn into strengths. While before different standards and implementations where competing in an incompatible manner, they now can coexist in a user friendly way. New implementations can arise and spread organically, users can openly decide, which technology to use.

Note: The restraint of interconnectivity to the bare technical minimum is not meant to span a meaningful social network of its own. Of course deeper functionality and security mechanisms are to be adapted, to create a meaningful and save user experience. Each actor who initiates communication, by choosing the software and the technology, also chooses the more detailed conditions of communication. If actors on the receiving end of such communication do not agree with any of these conditions, it should be blocked. Interconnectivity aims to remove arbitrary technical barriers, so that developers and users have the chance to enforce barriers by design and by choice. When actors have contradicting communication desires, that's not a technical issue and couldn't be solved by any protocol.

---

interoperability > interconnectivity > no connectivity.

3.2 Specification

3.2.1 1. Overview

Iconet is about presenting content from foreign sources in situation where a client is not able to present this content with native means.

This document and subdocuments linked here will guide you through the process of supporting interconnectivity. The walkthrough comprises the following steps:

1. To begin with, a packet is sent to a client that does not have the capabilities to display / present it to the user.
2. The packet however has iconet-(meta)data attached which the client can use as a fallback. The data required is described in section *2. Required Iconet Data for a Packet*. The iconet data may also provide the packet in additional formats.

3. The iconet data includes information that allows the client to render a presentation from the packet (using *fallback-iframes*) and optionally *translate* the packet to different formats. Note, that packet content (dynamic & private) is intentionally separated from the fallback iframe data (static & public). See section *3. Interpreter Manifests*.
4. The presentation is rendered (or translated) with a fallback-iframe. Fallback-iframes and the communication flow between the fallback-iframe and the embedding application, as well as sandboxing, is described in section *4. Fallback-iframes*.
5. Last but not least - some packet types (e.g. polls) need support for interactions, to communicate back to sender of a packet (or even to third parties). Receiving updates to certain packets (e.g. read receipts or comments) are valid use cases as well. How this can be done is discussed in sections *4.2.4. Sending Interactions* and 4.2.5. Receiving Updates / Interactions.

1.1 Introductory Notes

The format and schema of packets varies among protocols. To support an iconet fallback presentation (fallback-iframe), embedding applications and fallback iframe developers will however have to support a common set of standardized procedures.

What is standardized?

- Communication between fallback-iframe and embedding application
- Iconet-specific metadata
- *Interpreter Manifests* (documents about the fallback iframes (i.e. interpreters) used to display or convert a packet)
- Communicating interactions between embedding application with fallback-iframes

What is not standardized?

- What packets look like that are to be presented to the user
- *What* is communicated
- How packets are transported from network A to B / What transport protocol is used.
- *How* the iconet metadata is formatted in a given protocol packet

The data objects described here are JSON-LD formatted. If you are not familiar with JSON-LD, think of it as plain JSON with some fancy `@context` and `@type` fields. They allow the JSON keys to be globally uniquely identifiable. Iconet's JSON-LD `@context` namespace is: `https://ns.iconet-foundation.org#`

3.2.2 2. Required Iconet Data for a Packet

Say, a client received a foreign packet and cannot provide a presentation natively. However, the packet contains iconet data that the client knows how to deal with. This section describes the required iconet data. Note, that iconet metadata that describes how to interpret a packet is static. This makes it cacheable and reduces the privacy and security attack surface for the actual user-content.

The following example shows a representation of how the iconet metadata *could* be contained in a JSON-LD object. It holds minimal amount of data, since only the mandatory interpreter from native to html is given.

```
{
  "@context": "https://ns.iconet-foundation.org#",
  "@type": "Packet",
  "actor":"iconet:alice@alicenet.net",
  "to": "iconet:bob@bobnet.net",
  "interpreterManifests": [
```

(continues on next page)

(continued from previous page)

```
    {
      "@id": "<e.g. https://app.example/interpreter/manifest>",
      "sourceTypes": ["<accepting packet source type, e.g. application/mastodon+json>"],
      "targetTypes":["<available output types, e.g. application/iconet+html or␣
↪application/matrix+json>"],
      "sha512": "<sha512 hash of the manifest document linked>"
    }
  ],
  "content": [
    {
      "packetType": "<the type of the packet, e.g. application/mastodon+json>",
      "payload": "<data of the native packet>"
    },
    {
      "packetType": "<e.g. text/plain>",
      "payload": "<e.g. This message is a poll which your client does not support>"
    }
  ]
}
```

Note: It is up to a given protocol, how this metadata is *actually* contained in a packet. XML-based protocol designers may wish to use an XML-based representation over a JSON-based one, for example. When the packet crosses protocol borders however, it needs to be ensured to be formatted in JSON-LD.

2.1 Field Descriptions

Field Name	Type	Description
`@context`	`string|object|array`	The JSON-LD context namespace. This should be set to `https://ns.iconet-foundation.org#`. You can find more details here.
`@type`	`string`	The type of data, the packet contains. For regular payloads, this would be `Packet`. Depending on the context, the types `Interaction`, `TranslatedPacket`, (`Update`, `UpdateInquiry`) may be applicable.|br| The semantics for those packets are discussed in later sections.
`interpreterManifests`	`Array` of interpreter manifest descriptions	An interpreter manifest contains a list of interpreter descriptions. Interpreters take a foreign protocol's packet and either show a presentation or translate the packet to a different format.
`interpreterManifests[i].@id`	`string`	The location of the manifest or a `data:` URI containing the interpreter manifest.
`interpreterManifests[i].sourceTypes`	`Array` of mime type `string`	A list of mime types or custom, application-specific types. For each input type in the list, the manifest must provide an interpreter accepting the given input type.
`interpreterManifests[i].targetTypes`	`Array` of mime type `string`	A list of mime types or custom, application-specific types. For each target type in the list, the manifest must provide an interpreter producing the given target type. Every packet must be able to find an interpreter with target `application/iconet+html` (mime type of fallback presentation-iframe).
`interpreterManifests[i].sha512`	`string`	A sha512 hex signature of the interpreter manifest document. Tip: can be computed with `crypto.subtle.digest('sha512', data)` in javascript.
`content`	`array` of content records.	Each content record consists of fields that describe and hold the same data but in a different format. If a client does not support the first listed content record, it can go down the list. It is advisable to provide a plain text fallback as last item.
`content[i].packetType`	an extended mime type `string`	This field describes the type of the `payload` field content. This may be a general type like `text/plain` or `image/jpeg`, or it can be a non-standard application-specific mime type (e.g. application/matrix).
`content[i].payload`	`string`	This field contains the data of the packet sent. The type and format is not specified and needs to be interpreted by the application or an interpreter that is linked in the interpreter manifest. An implementing protocol will likely want to allow this field to be be undefined, if the wrapping packet (i.e. the packet that contains this iconet packet) is to be used as payload.

3.2.3 3. Interpreter Manifests

WIP-Level: 2

The interpreter manifests are JSON-documents that contain the metadata for fallback-iframes and translators. The manifests are linked by iconet-supporting packets and should be cached by the clients. When a client does not know how to present a packet to a user, it will fetch the manifest for a given source packet type and use a fallback presenter- or translator-iframe referenced in the interpreter manifest to present or translate the foreign packet.

The manifest format is standardized, in comparison to the metadata described in the *section above (Required Iconet Data for a Packet)*. You can see an example below:

```
{
  "@context": "https://ns.iconet-foundation.org#",
  "@type": "InterpreterManifest",
  "@id": "<URI of this document>",
  "interpreters": [
    {
      "@id": "<URI to fallback-iframe>",
      "sourceType": "<(custom) mime type, e.g. application/activity+json>",
      "targetType": "<(custom) mime type application/matrix+json or application/
↪iconet+html>",
      "sha512": "<sha512 hash of the linked document>",
      "permissions": {
        "<some permission>": "<value>"
      }
    }
  ]
}
```

If the `targetType` of a here given interpreter is `application/iconet+html`, the `@id` field must reference a HTML document that is a *presenter iframe*. If the `targetType` of a here given interpreter is anything else, the `@id` field must reference a HTML document that is a *translator iframe* for that target type. The `@id` field may have a `data:` URI value containing the HTML payload. Permissions are described in *4.1.1 Fallback-iframe Permissions*.

3.2.4 4. Fallback-iframes

Fallback-iframes build the foundation of presenting packets to the user or translating them. The *interpreter manifest's* `targetType` must reference a fallback iframe HTML document. The HTML document must be embedded and sandboxed using the HTML iframe tag. There are multiple methods described for embedding and sandboxing in section *4.1.3 Sandboxing Iframes*.

This section describes the communication and encapsulation between *embedding application* and *fallback-iframe*. The embedding application passes a packet to the embedded fallback-iframe to display it, see *4.2 Communication between Embedding Application and Fallback-iframe* for the walkthrough.

There are two types of fallback iframes: Translator Iframes and Presenter Iframes. Translator iframes simply return a packet translated to a different format when they are given a packet, as described in *4.2.3. Packet Translation Response*. Presenter iframes instead render a HTML presentation. Users may interact with them and the iframes may trigger interaction packets back to the sender of the packet, as described in *4.2.4. Sending Interactions*.

See also:

You can find an example document, an embedding application that embeds an iframe, enforces restrictions, and initiates communication here.

Since by default, fallback-iframes are not allowed to connect to endpoints on the web, all data needs to be embedded within the iframe's HTML. You can see an example of an embedded image here.

To maintain security and safety for users, it is important to sandbox the iframes and restrict permissions. For permissions and sandboxing, see the section linked here:

4.1 Iframe Permissions and Sandboxing

Fallback-iframes are used to display and encapsulate iconet messages in HTML format. The following sections describe how communication and encapsulation between parent and the fallback-iframe can be accomplished, different levels of isolation, and the permissions used to describe them in the *interpreter manifest*.

See also:

You can find an example document, an embedding application that embeds an iframe, enforces restrictions, and initiates communication here.

Since by default, fallback-iframes are not allowed to connect to endpoints on the web, all data needs to be embedded within the iframe's HTML. You can see an example of an embedded image here.

4.1.1 Fallback-iframe Permissions

By default, an iframe is not allowed to establish connections to the outside world. By requesting permissions, the iframe communication restrictions may be alleviated.

Name	Range	Description
`allowedSources`	array of CSP source values	Sources to URIs the iframe may connect to. Defaults to empty.
`allowInteractions`	boolean	If the fallback-iframe is allowed to send interactions via the embedding application. If true, must be set along with `interactionCooldown`. Default is `false`.
`interactionCooldown`	number	The number of seconds the fallback-iframe needs to wait until it can send another interaction. Requires `allowInteractions` to be set to `true`.
`allowContentRequest`	boolean	Experimental: Allow the iframe to make requests through the embedding application, to check if there are updates to the packet using the *interaction pull method*.

4.1.2 Levels of Isolation

WIP-Level: 3 (figuring out which permissions to offer by spec and when to apply which ones is still in debate)

Depending on the trust established between receiver and sender, the capabilities of the iframe may be restricted to varying extent. The following subsections discuss different levels of isolation, beginning with the strictest.

Disabled Remote Resource Access

For this level, the fallback-iframe will receive the iconet packet but is not allowed to send information away or request additional content (e.g. images). All resources have to reside in the HTML of the fallback-iframe, e.g. images must be encoded in the HTML. Encoded binary data should reside at the bottom of the document to enable pre-rendering the rest of the document.

Controlled Remote Communication

Sending packets from the fallback-iframe is allowed but only by using a channel controlled by the embedding application.

This way, the parent can restrict the frequency of messages sent and restrict the addressees to the sender audience. See the section on *sending interactions*

TODO: Sort out specification for allowed addressee restriction.

Enabled Remote Resource Fetching

TODO: enhance CSP configuration description.

The fallback-iframe is allowed to fetch resources from certain sources that are whitelisted in the iconet packet. The parent applies a corresponding `default-src` Content Security Policy.

Additional CSP directives do not seem to provide additional security (e.g. using a more liberal `img-src` compared to `default-src` cannot prevent the script from communicating through image resources)

The degree of trust between sender and receiver must be fairly high, since a lot of metadata (e.g. IP, browser fingerprint, time of rendering the message, user behavior) can be exposed this way.

Trust

There are at least three imaginable sources for interpreters and multiple authorities for determining the level of trust that a fallback-iframe should be granted.

Sources for interpreters:

- sending client (linked in iconet packet)
- receiving client (either from cache or configuration)
- community repository

Authorities for establishing trust:

- A designated third party authority or community repository
- The interpreter domain / organization
- The sending user
- The receiving user by review

Most times, there is to some degree a trade-off between usability and privacy. We want to make this as small as possible while preserving a high degree of privacy. Therefore, a fallback-iframe should limit its level of isolation to the minimum required. This will improve usability for users, since they do not have to perform a review to elevate the fallback-iframe's permission.

Establishing trust is a critical component and needs additional elaboration and agreement among participating parties. This is a big TODO and detailed specification seems out of scope for now.

4.1.3 Sandboxing Iframes

WIP-Level: 1

Fallback-iframes must be sandboxed to enforce isolation and prevent execution of arbitrary code in the parent container and communication to the outside world.

Content Security Policy

To prevent iframes from unmonitored communication in browser-based contexts, a Content Security Policy (CSP) can be employed. Below, you can find three approaches to enforce CSPs on iframes.

Option 0: Using the Iframe's `csp` Attribute

As of 2022-11-23, there exists a draft spec to enforce the CSP on an iframe:

```
<iframe csp="default-src 'none'">
```

It is however not supported by Firefox and Safari.

Option 1: Add a CSP to the Iframe's HTML

(1) Fetch the fallback-iframe's HTML by script

(2) Modify the fetched HTML appending a `meta` tag at the top of the HTML head:

```
<html>
  <head>
    <meta http-equiv="Content-Security-Policy" content="child-src 'none';" />
    ...
```

(3) Set the modified HTML to the iframe's `srcdoc` attribute.

Warning: You *must* make sure that no scripts or remote content precedes the meta tag. The CSP only takes effect, once the tag is parsed. See the spec for more details.

Option 2: Use a proxy that sets the appropriate Content Security Policy headers

Instead of using the URI to the fallback-iframe directly, use a trusted proxy that fetches the iframe HTML and sets the desired CSP headers on response.

4.2 Communication between Embedding Application and Fallback-iframe

The communication flow to set up the communication looks as follows

1. The fallback-iframe initiates the communication, once loaded. It transfers a message port to the parent.
2. The parent receives the initiation request and responds with the iconet packet to render to the user.
3. For Translators: The iconet translator iframe responds with the translated packet. Communication has finished.
4. Optional: The iconet presenter iframe requests to send away an interaction packet.
5. Optional & at Discussion: The parent receives an additional packet that is passed on to the fallback-iframe.

4.2.1 Fallback-Iframe Ready

When the fallback-iframe is ready to receive packets, it calls `parent.postMessage()` (see reference) with `targetOrigin` of *.

The `message` parameter must look as follows:

```
{
  "@context": "https://ns.iconet-foundation.org#",
  "@type": "IframeReady"
}
```

The fallback-iframe MUST create a message channel and pass its message port using the `transfer` parameter of `parent.postMessage()`. The message channel must be used for all future communication.

The initiation in the fallback-iframe may look something like:

```
document.addEventListener("DOMContentLoaded", async() => {
  // Create a message channel for the future communication with the parent.
  const messageChannel = new MessageChannel();
  messageChannel.port1.onmessage = (messageEvent) => {
    console.info("Message received from parent!", messageEvent);
    // Handle incoming data and render a presentation.
  };

  // Send initial message to parent, transferring the message port.
  parent.postMessage({
      "@context": "https://ns.iconet-foundation.org#",
      "@type": "IframeReady"
    },
    "*",
    [messageChannel.port2]
  );
});
```

4.2.2 Parent Response

The parent, listening to the iframe's `message` events, receives the message event and the transferred message port. Using the message port, the parent responds with the packet payload (`event.ports[0].postMessage(packetPayload)`).

The listener must validate that the iframe's initial message comes from the expected source iframe.

See the code example for an embedding application listening to an iframe:

```
window.addEventListener("message", (event) => {
  // Validate event source.
  if (event.source !== targetIframe.contentWindow) {
    // Handle illegal message.
    return;
  }

  // The iframe passed a message port for further and secure communication.
  iframeMessagePort = event.ports[0];
  iframeMessagePort.onMessage =  (message) => {
    // Handle message from iframe here.
  };

  // Pass the received foreign iconet packet payload to the iframe.
  iframeMessagePort.postMessage(payloadForIframesMimeType);
});
```

The fallback iframe can now use the received payload (which was formatted in the `sourceType` specified by the interpreter manifest), to either render a presentation (if the `targetType` was `application/iconet+html`) or translate the packet to the according `targetType` otherwise.

4.2.3. Packet Translation Response

WIP-Level: 2

If the fallback-iframe is a translator iframe and has received the foreign packet, it responds with the translated packet as following:

```
{
  "@context": "https://ns.iconet-foundation.org#",
  "@type": "TranslatedPacket",
  "originalPacket": "<id of original packet>",
  "mimetype": "<translated packet's mimetype",
  "payload": "<translated packet>"
}
```

4.2.4. Receiving Updates / Sending Interactions

It is necessary to receive updates to a packet, e.g. to load the newest version of a post. Since creation of the post and the actual viewing by the recipient interactions and updates may have happened.

The updates are requested and interactions are sent by the fallback iframe through native APIs provided by the sending server.

In the manifest such native API's need to be listed in allowedSources according to the *Fallback-iframe Permissions*.

Example:

```
"allowedSources": [
  "http://localhost:8001/api/getPost.php",
  "http://localhost:8001/api/addComment.php"
]
```

This leverages flexible capabilities for the fallback-iframe developer but may lead to a larger amount of (meta) data leakage to the whitelisted sources. In addition, the whitelisted sources should be approved by the embedding application itself and/or the user which might turn out to be challenging from a UX perspective.

3.2.5 5. Transport

So how do the packets reach their supposed destination? There are three ways:

5.1 Native Transport

The core use-case of iconet is, to provide a fallback for within networks, where connectivity is already established. In this case decentral applications with already common understandings of actors, relationships do share means of communication and security.

It is trivial for them, to also include the required fallback fields into their exchanged packets.

Here you see an example of a activitystream packet including the iconet information:

```
{
  "type":"Create",
  "@context":"https://www.w3.org/ns/activitystreams",
  "id":"https://bridge.localhost/b3f764ba-6343-419a-be42-a91580b7454b",
  "actor":"https://bridge.localhost/user/alice__neta.localhost",
  "object":{
    "type":"Note",
    "https://ns.iconet-foundation.org#iconet":{
      "@context":"https://ns.iconet-foundation.org#",
      "@type":"Packet",
      "@id":"https://bridge.localhost/9c94ca30-60e0-4951-aef2-e406309b0390",
      "actor":"alice__neta.localhost@bridge.localhost",
      "to":[
        "admin@localhost:3000"
      ],
      "interpreterManifests":[
        {
          "manifestUri":"http://neta.localhost/iconet/formats/markdown/manifest.json",
          "sourceTypes":[
```

(continues on next page)

(continued from previous page)


```
            "text/markdown"
          ],
          "targetTypes":[
            "application/iconet+html"
          ],
          "sha-512":"empty"
        }
      ],
      "content":[
        {
          "packetType":"text/markdown",
          "payload":"## Title\nText `code`"
        }
      ],
      "XDEBUG_SESSION_START":"13228"
    },
    "@context":"https://www.w3.org/ns/activitystreams",
    "id":"https://bridge.localhost/4fb30562-6f2e-48f2-9e98-bb59b7f9e049",
    "published":"2023-02-26T14:21:36+00:00",
    "content":"This status contains iconet data.",
    "to":[
      "http://localhost:3000/users/admin",
      "https://www.w3.org/ns/activitystreams#Public"
    ]
  }
}
```


5.2 Fallback Transport

For communication between parties on different protocols, the information format, schema, authentication methods, API endpoints, transmission protocol, etc. might not align.

Different networks which have the same fallback mechanism established and manage to exchange the fallback information, can provide interconnectivity to their users.

That's why it makes also sense, to have a fallback way for transport established.

5.2.1 Adressing & Routing

To send a message, the addressed receiver needs to be uniquely identifiable and locatable.

Users need to exchange addresses in URI format. If they do not have a common addressing protocol, they shall use the iconet schema:

iconet:<domain-specific identifier>@<dereferenceable address>

- The dereferenceable address is the endpoint the message is sent to.
- The domain-specific identifier is only required to be processed by the receiving server.

The `iconet:` URI scheme specifies the protocol to be used, i.e. makes clear that this is an iconet address.

As transport protocol HTTPS is used. As endpoint address we define `<dereferenceable address>/iconet/`, while in the future a port on the main address should be reserved for fallback transportation.

5.2.2 Encryption & Security

If the sending and receiving party do not share means for encryption and authentication, we recommend a fallback encryption method.

We use a hybrid cryptosystem similiar to PGP for our encryption porpuses.

The payload of each packet gets encrypted symmetrically with an AES-Secret. The secret get encrypted with each contacts public key. How each contacts public key is obtained and trusted, needs to be done according to the personal trust model.

Here is our example for an encrypted Packet:

```
{
  "@context": "https://ns.iconet-foundation.org#",
  "@type": "EncryptedPacket",
  "actor": "bob@netb.localhost",
  "to": "alice@neta.localhost",
  "EncryptedSecret": <AES-Secret, Encrypted with recipients public key>
  "EncryptedContent": <Manifestinfo & Packetcontent, Encrypted with AES-Secret>
}
```

5.3 Bridged Transport

Bridges are the hybrid solution between each native transport and a universal fallback transport.

While there are a couple of downside with bridges, they can be useful in the early adoption phase, when not many servers have adopted the fallback transport yet. This way more users can be reached with interconnectivity.

1. **Where are bridges hosted?**
    - **Case 1: Every individual user hosts its own bridge.** Bridges are a risk for privacy, since traffic has to be decrypted before it can be bridged (and encrypted again). This issue can be avoided, when users host their own bridges on a trusted device for example on the client. Also, bridging remains easier since the bridge only needs to act as the user on the remote network (puppet bridge). **Downside**: Every user will have to host an own bridge.
    - **Case 2: Hosting a bridge per room or server.** In this case, users don't need individual bridges. However, all users will have to trust the bridge not to abuse its power to read and manipulate users messages. Additionally, many platforms do not support bridges that operate this way. In the worst-case a bridge bot that joined a room will post a message that describes the sender by appending its name to the message body.
2. **(How) can we discover and reach out to users of different protocols?** Ideally, a user is able to find contacts on different platforms (e.g. by phone number or email address). Therefore, not only a cross-platform user index needs to be present but also an endpoint must be clear which the protocol can target to transfer a message across to a different protocol. The bridging endpoint needs to be added to each users address, so it can be forwarded from here. E.g.: *alice@netA.org@bridge.net*

There are different types of bridges to transfer packets across platforms. This matrix post discusses different types of bridges for the interested reader.

3.3 Challenges & Approach Discussion

Connecting (partially) incompatible networks opens up questions. For example:

- How to address a user / audience on a remote platform?
- How to present their content?
- How do systems communicate compatibility support?
- How can interactions (i.e. a poll or reactions) be supported?
- How / in what schema is content formatted?
- How does the API work?
- How to enable secure communication?
- **Finding consensus & acceptance in the community**

This page is a very incomplete list of challenges and discussions on possible solution pathways for solving those issues.

3.3.1 Presentation

- When to fall back to non-native presentation of content?
- What format does the fallback presentation have?
- How is fallback presentation generated - on the receiver side by using the sender's platform-native message and transforming it in a receiver-understood target format?
- What, if the client understands parts of the message (e.g. an attached vCard) but does not know how to present the whole information?

For presentation on the client side, we identify three approaches which are not mutually exclusive but combinable.

1. Using format interpreters that support logic and which take any kind of message to create a supported presentation (flexible but complex). This is the most flexible approach and where Iconet is heading towards.
2. Using a very basic common default format supported by all clients, e.g. plain text (which has to be sent along the original packet or on request). This could be a useful fallback for clients that do not supported HTML+Javascript.
3. Using content negotiation to find out about the intersection of supported formats between sender and receiver application. Disadvantage: what happens if there are no common formats? A more detailed elaboration of the approaches is given in the subsections.

A possible example procedure combining the approaches could look as follows:

1. The receiving client has received a packet that it cannot present to the user. The packet has a plain text presentation attached as well (as last resort).
2. The receiving client supports HTML and decides to fetch a format interpreter HTML document linked in the packet. Alternatively, the document could reside in the packet. (Optionally: When fetching the document, content negotiation could also result in a different kind of interpreter that, for certain protocols, can translate the packet into a format understood by the client.)
3. The format interpreter HTML document can be embedded as an iframe by the receiving client. The receiving client passes the packet to the format interpreter HTML document which renders a HTML representation.

Format Interpretation

The idea of providing an interpreter is the following: If the receiver application is not able to present the received packet in the sender application's format, the interpreter can be used to translate the information that was not understood by the receiver application.

The interpreter can be thought of as a converter that takes the sender's native packet and generates a markup that the client can render. To support interactive content, the interpreter could be an iframe taking data passed to it to display a widget.

Ideally, no sensitive data should be exposed to anyone but the receiver in that process. Therefore, the question arises as to how the interpreter information is transmitted to the receiver application. By request (and to whom?) if the packet is not understood or in advance by the sender application?

The advantage of using an interpreter is that the sender does not need to provide (another) target format. The transformer can be developed separately. The message does not need to be re-requested if the receiver application does not understand the format but only a transformer.

In some cases, certain parts of the foreign packet might be understood by the receiving client. In that case, the receiving client is in charge of deciding if to render the fallback and how handle the packet.

Potential Examples

An isolated iframe (with no access to external (internet) resources) receives the packet that's not able to be presented by the receiver application's native methods.

In the iframe, the interpreter can execute JavaScript to process the data and render a presentation for the user. Outgoing communication could be channeled through an interface controlled by the client, that acts as a proxy and filters requests to external resources.

Default Formats

Alternatively, instead of providing an interpreter, the message sent could be formatted in a very generic way understood by any client (e.g. plain text) or multiple formats. See this matrix MSC as a source of inspiration for example. The MSC proposes to append a markup for a message in multiple formats, each with a mimetype attached. The client is supposed to present the first format supported.

```
"m.markup": [
  {"body": "<h1>Hello there!</pre>", "mimetype": "text/html"},
  {"body": "Hello there!",           "mimetype": "text/plain"}
]
```

Content Negotiation

In the previous section, there were multiple formats of the same message were sent in one packet. In some cases, it might be useful to perform content negotiation.

The client or the open inbox could ask to receive the message in a preferred format that both support. Downside: The inbox might not know in advance which formats the client fetching the messages supports.

A different form of content negotiation could be used when the receiving client requests a format interpreter. The endpoint that provides the format interpreters could support content negotiation for different target formats (that would not even need to convert to a markup but a client-native packet format).

tl;dr

Options:

1. The Iconet packet contains the sender-native packet.
    - The iconet packet contains a link to an endpoint providing interpreters (e.g. iframes) that the client can use to convert and render the packet. Using content negotiation, different target formats could be requested. Caching and security considerations should be considered.
    - Instead of providing an endpoint for the interpreter, the needed data could be transmitted in the iconet packet as well but reducing caching options.
    - The interpreters's code should be provided with a checksum to reduce attack vectors.
2. The iconet packet contains multiple content formats of which at least one should be supported by the receiving application (probably plaintext).
3. Both of the above

3.3.2 Content Interaction

- How are interactions (e.g. a like or a comment) made available / communicated back?
- Should iframes be allowed to communicate back themselves (via their own requests) or should they request their parent window to communicate?
- Is an iconet (interaction) packet processed by the client or the inbox server?
- Can interactions be authorized and how?
- Use cases:
    - Use Case: Bob can like Alice's post, but Claire (to whom the post was forwarded), can't Can authorization to access formats be delegated to receivers?
    - (How) can messages be forwarded? Can formats handle 'forwarding' internally (not part of the standard)?
    - Should we aim for standard auth mechanisms (e.g. OAuth)?
    - Use Case?: A shared document is sent to Bob and a preview is rendered but Claire, to whom the post was forwarded, should not see the preview. (How) can access be granted or requested?
    - A public post that is sent to Bob in an iconet packet is supposed be be commentable by Bob only.

Option 1: Iframe communicates with native Apis

The main approach, as documented in *4.2.4. Receiving Updates / Sending Interactions*

Option 2: Tunneled communication between iframe & embedding Application

The fallback-iframe can communicate over a controlled channel provided by its embedding application. Packets sent over the controlled channel are sent back to the sender of the original packet. Since transport is handled by the embedding application, this reduces overhead for the fallback-iframe creators and moves questions of signing and authenticating as well as encrypting packets there. Therefore, for end-to-end encrypted communication, this has the advantage of minimizing possible data-leakage to third-parties. On the other hand, it is less flexible.

The interaction packet is a JSON-LD-formatted object, the **format is standardized**:

```
{
  "@context": "https://ns.iconet-foundation.org#",
  "@type": "Interaction",
  "@id": "<an IRI identifying the packet>",
  "timestamp": "<ISO 8601 timestamp>",
  "originalPacket": "<the IRI identifying the original / target packet>",
  "payload": "<custom data>"
}
```

The interaction packet's payload is then forwarded back to the sender of the original packet.

Modes of Communication & Security

A container in the receivers application is used to wrap the interpreter and its packet presentation.

Possible proposal: Multiple trust levels

1. container is completely isolated
2. container may communicate via its parent host in a limited fashion
3. container may communicate by itself

The user or the client is in charge of allowing to elevate a message's trust level.

Embedded (non-native) content MUST NOT be allowed to alter their parents' state directly. Communication between the parent application and the embedded content must be supervised.

Here, we illustrate three concepts on how interactions or updates can reach the client.

1. *Option 1)* would support neither method but only allow trusted fallback-iframes to connect to sources using the available javascript interfaces, e.g. `fetch`. Thus, no further specification on the iconet side would be necessary. See *Sending Interactions* for the related discussion.
2. *Option 2)* would allow actors to send updates (e.g. a message was edited) for packets they previously sent to the inbox. This method can be thought of as push-like.
3. *Option 3)* would allow clients to poll for updates at the source, if needed. This method can be though of as pull-like.

5.1. Option 1: Fallback-iframe Self-Governing

In analogy to the previous section, the iframe connects to a whitelisted source by itself and polls for updates. The process is thus not iconet-specific.

5.2. Option 2: Push-Based

If updates to a packet (i.e. someone commented on a post) arise, the embedding application receives an interaction packet / update to a packet. The embedding application passes the payload over to the fallback-iframe.

Note that if transport is not handled by a common iconet protocol, the schema and format may vary. The payload however must not be altered during transport This is in analogy to the regular packets.

```
{
    "@context": "https://ns.iconet-foundation.org#",
    "@type": ["Packet", "Update"],
    "@id": "<id of packet>",
    "refersTo": "<id of original packet>",
    "actor": "<sender>",
    "to": ["<addressee>"],
    "content": ["<... content data array, same as in the regular packets>"]
}
```

(5.3. Option 3: Pull-Based)

WIP-Level: 4

A request to ask for updates could look as follows. Note that if transport is not handled by a common iconet protocol, the the schema and format may vary.

```
{
    "@context": "https://ns.iconet-foundation.org#",
    "@type": "UpdateInquiry",
    "originalId": "<id of original message>",
    "sender": "<sender address of this packet>",
    "to": ["<address of the original packets sender>"]
}
```

The response would be a regular iconet packet with an additional `updateTo` field that contains the original packet's identifier. Alternatively, the packet could maintain the same id and the packet could have a timestamp and an update count.

One option would be that the client invalidates the old message from thereon. Alternatively, the client could pass the payload of the response to the iconet iframe, using the established message channel between client and fallback-iframe.

A response to an update inquiry could look as follows. If status `NoChange` is set, the packet does not need to have type Packet and the corresponding payload field.

```
{
    "@context": "https://ns.iconet-foundation.org#",
    "@type": ["Packet", "Update"],
    "@id": "<id of packet, if packet has update>",
    "predecessor": "<id of original message>",
    "status": "<either Update or NoChange.>",
    "sender": "<sender address of this packet>",
    "to": ["<address of the original packets sender>"],
    "payload": "<native data>"
}
```

Authentication

The authenticity of an actor could be established if communication is channeled through the parent host and the parent host's authenticity to the original sender is given. In the end, that leads to a situation where trust needs to be established between platforms and users, where the respective platform ensures the authenticity of senders and receivers.

A separate, dedicated mechanism seems out of scope for a basic protocol. Even the whole web community hasn't agreed on a single working interoperable spec that fulfills everyone's use case.

3.4 Glossary

3.4.1 Fallback Iframe (Interpreter)

The heart of fallback data interpretation. Iframes are HTML elements that can embed external html content in an isolated sandbox. The fallback iframe is used by the *embedding application* which is not capable of interpreting a *foreign packet*. The fallback iframe communicates with the embedding application in a standardized manner, to pass the packet to the iframe and enable further interaction.

There are two types of *fallback iframes*: *Presenter Iframes* and *Translator Iframes*

Presenter Iframes render a HTML presentation of the packet passed to it and may allow user interaction with the iframe.

Translator Iframes function in the same way, however, they do not render a presentation but return the given packet data in a different format back to the embedding application, hence "translate" the packet. For example, a client received a packet of type `text/markdown`. The packet has translator iframes referenced to convert `text/markdown` to `image/jpeg` and `text/html`. Since the application doesn't know how to interpret markdown, it uses the translator to convert it to a HTML document, sanitizes the HTML, and shows it to the user.

3.4.2 Embedding Application

An embedding application is an application that wants to display content that it cannot interpret itself but that has iconet metadata attached. It therefore passes the content (a packet) to a fallback iframe which renders a presentation.

3.4.3 Foreign Protocol

A protocol that is not (natively) supported by a given *embedding application*.

3.4.4 Foreign Packet

A packet that is in the format of a *foreign protocol*.

3.4.5 Interpreter Manifest

A JSON-LD object that provides information on fallback iframes, their supported input packet types, their location, and their security-relevant permission requirements.

3.4.6 Packet

A collection of data used to communicate from one application to another.

3.4.7 WIP-Level

Some sections begin with a `WIP-Level:` indicator. The Work-In-Progress-Levels is a guide as to how much mature the given section is. Where WIP-Level 5 means "absolutely just some notes or thoughts" and WIP-Level 0 means "we elaborated extensively and came up with something we feel confident enough to put in our prototype implementation".

3.4.8 Status

TODO: Somethink like a value of `implemented`, `idea`, `in discussion`.