Workgroup:	BiDirectional or Server-Initiated HTTP
Internet-Draft:	WAMP
Published:	28 June 2022
Intended Status:	Experimental
Expires:	30 December 2022
Author:	T. Oberstein *typedef int GmbH*

The Web Application Messaging Protocol

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 30 December 2022.

Copyright Notice

Table of Contents

1. WAMP Basic Profile

This document defines the Web Application Messaging Protocol (WAMP). WAMP is a routed protocol that provides two messaging patterns: Publish & Subscribe and routed Remote Procedure Calls. It is intended to connect application components in distributed applications. WAMP uses WebSocket as its default transport, but can be transmitted via any other protocol that allows for ordered, reliable, bi-directional, and message-oriented communications.

1.1. Basic vs Advanced Profile

This document first describes a Basic Profile for WAMP in its entirety, before describing an Advanced Profile which extends the basic functionality of WAMP.

The separation into Basic and Advanced Profiles is intended to extend the reach of the protocol. It allows implementations to start out with a minimal, yet operable and useful set of features, and to expand that set from there. It also allows implementations that are tailored for resource-constrained environments, where larger feature sets would not be possible. Here implementers can weigh between resource constraints and functionality requirements, then implement an optimal feature set for the circumstances.

Advanced Profile features are announced during session establishment, so that different implementations can adjust their interactions to fit the commonly supported feature set.

1.2. Introduction

This section is non-normative.

The WebSocket protocol brings bi-directional real-time connections to the browser. It defines an API at the message level, requiring users who want to use WebSocket connections in their applications to define their own semantics on top of it.

The Web Application Messaging Protocol (WAMP) is intended to provide application developers with the semantics they need to handle messaging between components in distributed applications.

WAMP was initially defined as a WebSocket sub-protocol, which provided Publish & Subscribe (PubSub) functionality as well as Remote Procedure Calls (RPC) for procedures implemented in a WAMP router. Feedback from implementers and users of this was included in a second version of the protocol which this document defines. Among the changes was that WAMP can now run over any transport which is message-oriented, ordered, reliable, and bi-directional.

WAMP is a routed protocol, with all components connecting to a *WAMP Router*, where the WAMP Router performs message routing between the components.

WAMP provides two messaging patterns: *Publish & Subscribe* and *routed Remote Procedure Calls*.

Publish & Subscribe (PubSub) is an established messaging pattern where a component, the *Subscriber*, informs the router that it wants to receive information on a topic (i.e., it subscribes to a topic). Another component, a *Publisher*, can then publish to this topic, and the router distributes events to all Subscribers.

Routed Remote Procedure Calls (RPCs) rely on the same sort of decoupling that is used by the Publish & Subscribe pattern. A component, the *Callee*, announces to the router that it provides a certain procedure, identified by a procedure name. Other components, *Callers*, can then call the procedure, with the router invoking the procedure on the Callee, receiving the procedure's result, and then forwarding this result back to the Caller. Routed RPCs differ from traditional client-server RPCs in that the router serves as an intermediary between the Caller and the Callee.

The decoupling in routed RPCs arises from the fact that the Caller is no longer required to have knowledge of the Callee; it merely needs to know the identifier of the procedure it wants to call. There is also no longer a need for a direct connection between the caller and the callee, since all traffic is routed. This enables the calling of procedures in components which are not reachable externally (e.g. on a NATted connection) but which can establish an outgoing connection to the WAMP router.

Combining these two patterns into a single protocol allows it to be used for the entire messaging requirements of an application, thus reducing technology stack complexity, as well as networking overheads.

1.3. Protocol Overview

This section is non-normative.

1.3.1. Realms, Sessions and Transports

A Realm is a WAMP routing and administrative domain, optionally protected by authentication and authorization. WAMP messages are only routed within a Realm.

A Session is a transient conversation between two Peers attached to a Realm and running over a Transport.

A Transport connects two WAMP Peers and provides a channel over which WAMP messages for a WAMP Session can flow in both directions.

WAMP can run over any Transport which is message-based, bidirectional, reliable and ordered.

The default transport for WAMP is WebSocket [RFC6455], where WAMP is an officially registered subprotocol.

1.3.2. Peers and Roles

A WAMP Session connects two Peers, a Client and a Router. Each WAMP Peer MUST implement one role, and MAY implement more roles.

A Client MAY implement any combination of the Roles:

- Callee
- Caller
- Publisher
- Subscriber

and a Router MAY implement either or both of the Roles:

- Dealer
- Broker

> This document describes WAMP as in client-to-router communication. Direct client-to-client communication is not supported by WAMP. Router-to-router communication MAY be defined by a specific router implementation.

A *Router* is a component which implements one or both of the Broker and Dealer roles. A *Client* is a component which implements any or all of the Subscriber, Publisher, Caller, or Callee roles.

WAMP *Connections* are established by Clients to a Router. Connections can use any *transport* that is message-based, ordered, reliable and bi-directional, with WebSocket as the default transport.

WAMP *Sessions* are established over a WAMP Connection. A WAMP Session is joined to a *Realm* on a Router. Routing occurs only between WAMP Sessions that have joined the same Realm.

The *WAMP Basic Profile* defines the parts of the protocol that are required to establish a WAMP connection, as well as for basic interactions between the four client and two router roles. WAMP implementations are required to implement the Basic Profile, at minimum.

The *WAMP Advanced Profile* defines additions to the Basic Profile which greatly extend the utility of WAMP in real-world applications. WAMP implementations may support any subset of the Advanced Profile features. They are required to announce those supported features during session establishment.

1.3.3. Publish & Subscribe

The Publish & Subscribe ("PubSub") messaging pattern involves peers of three different roles:

- Subscriber (Client)
- Publisher (Client)
- Broker (Router)

A Publisher publishes events to topics by providing the topic URI and any payload for the event. Subscribers of the topic will receive the event together with the event payload.

Subscribers subscribe to topics they are interested in with Brokers. Publishers initiate publication first at Brokers. Brokers route events incoming from Publishers to Subscribers that are subscribed to respective topics.

The Publisher and Subscriber will usually run application code, while the Broker works as a generic router for events decoupling Publishers from Subscribers.

1.3.4. Remote Procedure Calls

The (routed) Remote Procedure Call ("RPC") messaging pattern involves peers of three different roles:

- Callee (Client)
- Caller (Client)
- Dealer (Router)

A Caller issues calls to remote procedures by providing the procedure URI and any arguments for the call. The Callee will execute the procedure using the supplied arguments to the call and return the result of the call to the Caller.

Callees register procedures they provide with Dealers. Callers initiate procedure calls first to Dealers. Dealers route calls incoming from Callers to Callees implementing the procedure called, and route call results back from Callees to Callers.

The Caller and Callee will usually run application code, while the Dealer works as a generic router for remote procedure calls decoupling Callers and Callees.

1.4. Design Aspects

This section is non-normative.

WAMP was designed to be performant, safe and easy to implement. Its entire design was driven by a implement, get feedback, adjust cycle.

An initial version of the protocol was publicly released in March 2012. The intent was to gain insight through implementation and use, and integrate these into a second version of the protocol, where there would be no regard for compatibility between the two versions. Several interoperable, independent implementations were released, and feedback from the implementers and users was collected.

The second version of the protocol, which this RFC covers, integrates this feedback. Routed Remote Procedure Calls are one outcome of this, where the initial version of the protocol only allowed the calling of procedures provided by the router. Another, related outcome was the strict separation of routing and application logic.

While WAMP was originally developed to use WebSocket as a transport, with JSON for serialization, experience in the field revealed that other transports and serialization formats were better suited to some use cases. For instance, with the use of WAMP in the Internet of Things sphere, resource constraints play a much larger role than in the browser, so any reduction of resource usage in WAMP implementations counts. This lead to the decoupling of WAMP from any particular transport or serialization, with the establishment of minimum requirements for both.

1.4.1. Application Code

WAMP is designed for application code to run within Clients, i.e. *Peers* having the roles Callee, Caller, Publisher, and Subscriber.

Routers, i.e. Peers of the roles Brokers and Dealers are responsible for **generic call and event routing** and do not run application code.

This allows the transparent exchange of Broker and Dealer implementations without affecting the application and to distribute and deploy application components flexibly.

> Note that a **program** that implements, for instance, the Dealer role might at the same time implement, say, a built-in Callee. It is the Dealer and Broker that are generic, not the program.

1.4.2. Language Agnostic

WAMP is language agnostic, i.e. can be implemented in any programming language. At the level of arguments that may be part of a WAMP message, WAMP takes a 'superset of all' approach. WAMP implementations may support features of the implementing language for use in arguments, e.g. keyword arguments.

1.4.3. Symmetric Messaging

It is important to note that though the establishment of a Transport might have a inherent asymmetry (like a TCP client establishing a WebSocket connection to a server), and Clients establish WAMP sessions by attaching to Realms on Routers, WAMP itself is designed to be fully symmetric for application components.

After the transport and a session have been established, any application component may act as Caller, Callee, Publisher and Subscriber at the same time. And Routers provide the fabric on top of which WAMP runs a symmetric application messaging service.

1.4.4. Peers with multiple Roles

Note that Peers might implement more than one role: e.g. a Peer might act as Caller, Publisher and Subscriber at the same time. Another Peer might act as both a Broker and a Dealer.

1.4.5. Router Implementation Specifics

This specification only deals with the protocol level. Specific WAMP Broker and Dealer implementations may differ in aspects such as support for:

- router networks (clustering and federation),
- authentication and authorization schemes,
- message persistence, and,
- management and monitoring.

The definition and documentation of such Router features is outside the scope of this document.

1.4.6. Relationship to WebSocket

WAMP uses WebSocket as its default transport binding, and is a registered WebSocket subprotocol.

2. Building Blocks

WAMP is defined with respect to the following building blocks

1. Identifiers
2. Serializers
3. Transports

For each building block, WAMP only assumes a defined set of requirements, which allows to run WAMP variants with different concrete bindings.

2.1. Identifiers

2.1.1. URIs

WAMP needs to identify the following persistent resources:

1. Topics

2. Procedures
3. Errors

These are identified in WAMP using Uniform Resource Identifiers (URIs) [RFC3986] that MUST be Unicode strings.

When using JSON as WAMP serialization format, URIs (as other strings) are transmitted in UTF-8 [RFC3629] encoding.

Examples

- `com.myapp.mytopic1`
- `com.myapp.myprocedure1`
- `com.myapp.myerror1`

The URIs are understood to form a single, global, hierarchical namespace for WAMP. The namespace is unified for topics, procedures and errors, that is these different resource types do NOT have separate namespaces.

To avoid resource naming conflicts, the package naming convention from Java is used, where URIs SHOULD begin with (reversed) domain names owned by the organization defining the URI.

Relaxed/Loose URIs

URI components (the parts between two `.`s, the head part up to the first `.`, the tail part after the last `.`) MUST NOT contain a `.`, `#` or whitespace characters and MUST NOT be empty (zero-length strings).

The restriction not to allow `.` in component strings is due to the fact that `.` is used to separate components, and WAMP associates semantics with resource hierarchies, such as in pattern-based subscriptions that are part of the Advanced Profile. The restriction not to allow empty (zero-length) strings as components is due to the fact that this may be used to denote wildcard components with pattern-based subscriptions and registrations in the Advanced Profile. The character # is not allowed since this is reserved for internal use by Dealers and Brokers.

As an example, the following regular expression could be used in Python to check URIs according to the above rules, when **NO empty URI components are allowed**:

```
pattern = re.compile(r"^([^\s\.#]+\.)*([^\s\.#]+)$")
```

When **empty URI components are allowed** (which is the case for specific messages that are part of the Advanced Profile), this following regular expression can be used (shown used in Python):

```
pattern = re.compile(r"^(([^\s\.#]+\.)|\.)*([^\s\.#]+)?$")
```

Strict URIs

While the above rules MUST be followed, following a stricter URI rule is recommended: URI components SHOULD only contain lower-case letters, digits and `_`.

As an example, the following regular expression could be used in Python to check URIs according to the above rules, when **NO empty URI components are allowed**:

```
pattern = re.compile(r"^([0-9a-z_]+\.)*([0-9a-z_]+)$")
```

When **empty URI components are allowed**, which is the case for specific messages that are part of the Advanced Profile, the following regular expression can be used (shown in Python):

```
pattern = re.compile(r"^(([0-9a-z_]+\.)|\.)*([0-9a-z_]+)?$")
```

Following the suggested regular expression for **strict URIs** will make URI components valid identifiers in most languages (modulo URIs starting with a digit and language keywords) and the use of lower-case only will make those identifiers unique in languages that have case-insensitive identifiers. Following this suggestion can allow implementations to map topics, procedures and errors to the language environment in a completely transparent way.

Reserved URIs

Further, application URIs MUST NOT use `wamp` as a first URI component, since this is reserved for URIs predefined with the WAMP protocol itself.

Examples

- `wamp.error.not_authorized`
- `wamp.error.procedure_already_exists`

2.1.2. IDs

WAMP needs to identify the following ephemeral entities each in the scope noted:

1. Sessions (*global scope*)
2. Publications (*global scope*)
3. Subscriptions (*router scope*)
4. Registrations (*router scope*)
5. Requests (*session scope*)

These are identified in WAMP using IDs that are integers between (inclusive) **1** and $\mathbf{2^{53}}$ (9007199254740992):

- IDs in the *global scope* MUST be drawn *randomly* from a *uniform distribution* over the complete range [1, 2^53]
- IDs in the *router scope* CAN be chosen freely by the specific router implementation
- IDs in the *session scope* MUST be incremented by 1 beginning with 1 (for each direction - *Client-to-Router* and *Router-to-Client*)

> The reason to choose the specific lower bound as 1 rather than 0 is that 0 is the null-like (falsy) value for many programming languages. The reason to choose the specific upper bound is that 2^53 is the largest integer such that this integer and *all* (positive) smaller integers can be represented exactly in IEEE-754 doubles. Some languages (e.g. JavaScript) use doubles as their sole number type. Most languages do have signed and unsigned 64-bit integer types that both can hold any value from the specified range.

The following is a complete list of usage of IDs in the three categories for all WAMP messages. For a full definition of these see messages section.

Global Scope IDs

- `WELCOME.Session`
- `PUBLISHED.Publication`
- `EVENT.Publication`

Router Scope IDs

- `EVENT.Subscription`
- `SUBSCRIBED.Subscription`
- `REGISTERED.Registration`
- `UNSUBSCRIBE.Subscription`
- `UNREGISTER.Registration`
- `INVOCATION.Registration`

Session Scope IDs

- `ERROR.Request`
- `PUBLISH.Request`
- `PUBLISHED.Request`
- `SUBSCRIBE.Request`
- `SUBSCRIBED.Request`
- `UNSUBSCRIBE.Request`
- `UNSUBSCRIBED.Request`
- `CALL.Request`
- `CANCEL.Request`
- `RESULT.Request`
- `REGISTER.Request`
- `REGISTERED.Request`
- `UNREGISTER.Request`
- `UNREGISTERED.Request`
- `INVOCATION.Request`
- `INTERRUPT.Request`
- `YIELD.Request`

2.2. Serializers

WAMP is a message based protocol that requires serialization of messages to octet sequences to be sent out on the wire.

A message serialization format is assumed that (at least) provides the following types:

- `integer` (non-negative)
- `string` (UTF-8 encoded Unicode)

- `bool`
- `list`
- `dict` (with string keys)

> WAMP *itself* only uses the above types, e.g. it does not use the JSON data types `number` (non-integer) and `null`. The *application payloads* transmitted by WAMP (e.g. in call arguments or event payloads) may use other types a concrete serialization format supports.

There is no required serialization or set of serializations for WAMP implementations (but each implementation MUST, of course, implement at least one serialization format). Routers SHOULD implement more than one serialization format, enabling components using different kinds of serializations to connect to each other.

The WAMP Basic Profile defines the following bindings for message serialization:

1. JSON
2. MessagePack
3. CBOR

Other bindings for serialization may be defined in the WAMP Advanced Profile.

With JSON serialization, each WAMP message is serialized according to the JSON specification as described in [RFC7159].

Further, binary data follows a convention for conversion to JSON strings. For details see the Appendix.

With MessagePack serialization, each WAMP message is serialized according to the MessagePack specification.

Version 5 or later of MessagePack MUST BE used, since this version is able to differentiate between strings and binary values.

With CBOR serialization, each WAMP message is serialized according to the CBOR specification as described in [RFC8949].

2.3. Transports

WAMP assumes a transport with the following characteristics:

1. message-based
2. reliable
3. ordered
4. bidirectional (full-duplex)

There is no required transport or set of transports for WAMP implementations (but each implementation MUST, of course, implement at least one transport). Routers SHOULD implement more than one transport, enabling components using different kinds of transports to connect in an application.

2.3.1. WebSocket Transport

The default transport binding for WAMP is WebSocket ([RFC6455]).

In the Basic Profile, WAMP messages are transmitted as WebSocket messages: each WAMP message is transmitted as a separate WebSocket message (not WebSocket frame). The Advanced Profile may define other modes, e.g. a **batched mode** where multiple WAMP messages are transmitted via single WebSocket message.

The WAMP protocol MUST BE negotiated during the WebSocket opening handshake between Peers using the WebSocket subprotocol negotiation mechanism ([RFC6455] section 4).

WAMP uses the following WebSocket subprotocol identifiers (for unbatched modes):

- `wamp.2.json`
- `wamp.2.msgpack`
- `wamp.2.cbor`

With `wamp.2.json`, *all* WebSocket messages MUST BE of type **text** (UTF8 encoded payload) and use the JSON message serialization.

With `wamp.2.msgpack`, *all* WebSocket messages MUST BE of type **binary** and use the MessagePack message serialization.

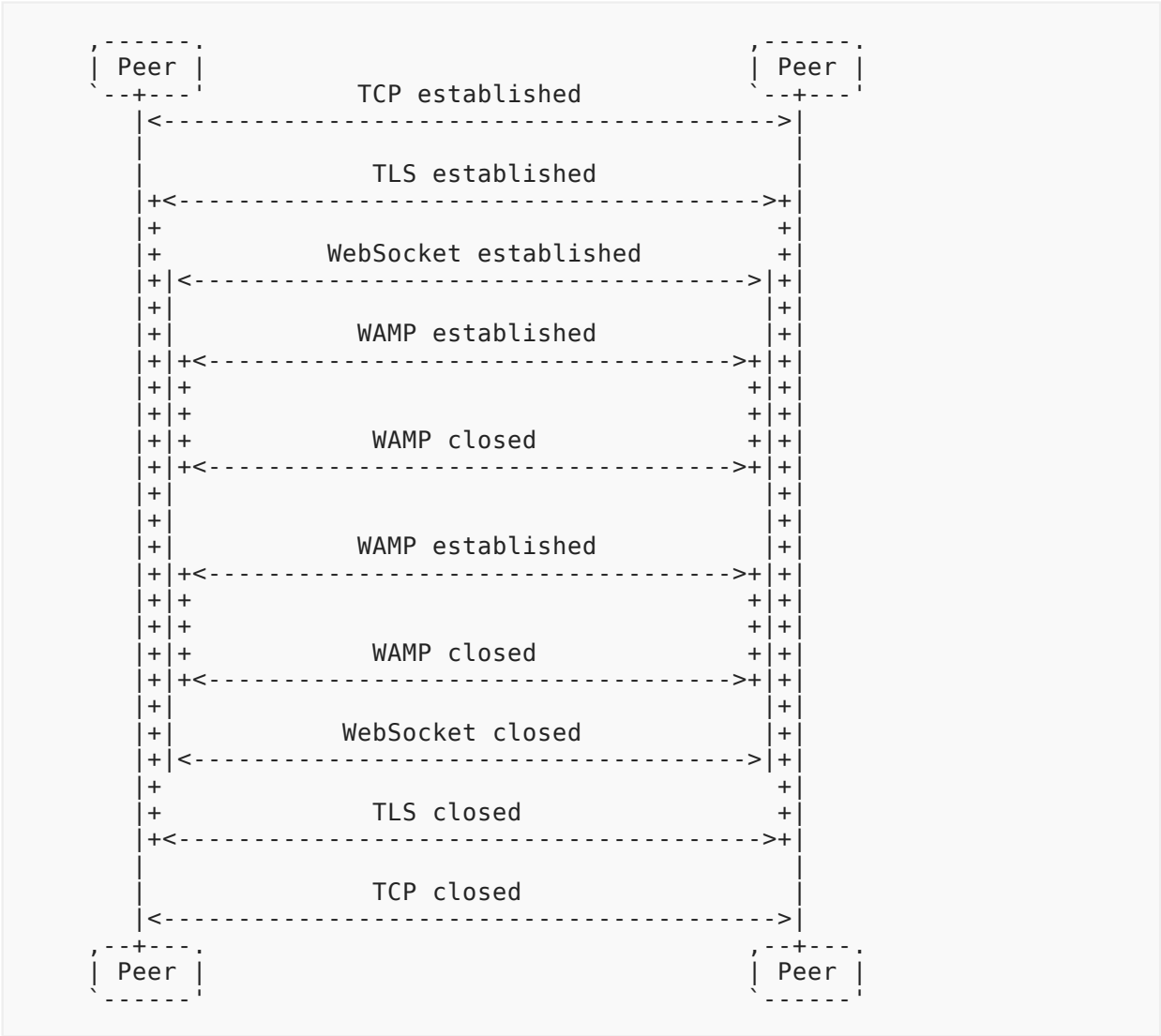
With `wamp.2.cbor`, *all* WebSocket messages MUST BE of type **binary** and use the CBOR message serialization.

> To avoid incompatibilities merely due to naming conflicts with WebSocket subprotocol identifiers, implementers SHOULD register identifiers for additional serialization formats with the official WebSocket subprotocol registry.

2.3.2. Transport and Session Lifetime

WAMP implementations MAY choose to tie the lifetime of the underlying transport connection for a WAMP connection to that of a WAMP session, i.e. establish a new transport-layer connection as part of each new session establishment. They MAY equally choose to allow re-use of a transport connection, allowing subsequent WAMP sessions to be established using the same transport connection.

The diagram below illustrates the full transport connection and session lifecycle for an implementation which uses WebSocket over TCP as the transport and allows the re-use of a transport connection.



2.3.3. Protocol errors

WAMP implementations MUST close sessions (disposing all of their resources such as subscriptions and registrations) on protocol errors caused by offending peers.

Following scenarios have to be considered protocol errors:

- Receiving `WELCOME` message, after session was established.
- Receiving `HELLO` message, after session was established.
- Receiving `CHALLENGE` message, after session was established.
- Receiving `GOODBYE` message, before session was established.
- Receiving `ERROR` message, before session was established.

- Receiving `ERROR` message with invalid REQUEST.Type.
- Receiving `SUBSCRIBED` message, before session was established.
- Receiving `UNSUBSCRIBED` message, before session was established.
- Receiving `PUBLISHED` message, before session was established.
- Receiving `RESULT` message, before session was established.
- Receiving `REGISTERED` message, before session was established.
- Receiving `UNREGISTERED` message, before session was established.
- Receiving `INVOCATION` message, before session was established.
- Receiving protocol incompatible message, such as empty array, invalid WAMP message type id, etc.
- Catching error during message encoding/decoding.
- Any other exceptional scenario explicitly defined in any relevant section of this specification below (such as receiving a second `HELLO` within the lifetime of a session).

In all such cases WAMP implementations:

1. MUST send an `ABORT` message to the offending peer, having reason `wamp.error.protocol_violation` and optional attributes in ABORT.Details such as a human readable error message.
2. MUST close the WAMP session by disposing any allocated subscriptions/registrations for that particular client and without waiting for or processing any messages subsequently received from the peer,
3. SHOULD also drop the WAMP connection at transport level (recommended to prevent denial of service attacks)

3. Messages

All WAMP messages are a `list` with a first element `MessageType` followed by one or more message type specific elements:

```
[MessageType|integer, ... one or more message type specific
    elements ...]
```

The notation `Element|type` denotes a message element named `Element` of type `type`, where `type` is one of

- `uri`: a string URI as defined in URIs
- `id`: an integer ID as defined in IDs
- `integer`: a non-negative integer
- `string`: a Unicode string, including the empty string
- `bool`: a boolean value (`true` or `false`) - integers MUST NOT be used instead of boolean value
- `dict`: a dictionary (map) where keys MUST be strings, keys MUST be unique and serialization order is undefined (left to the serializer being used)

- `list`: a list (array) where items can be again any of this enumeration

Example

A `SUBSCRIBE` message has the following format

```
[SUBSCRIBE, Request|id, Options|dict, Topic|uri]
```

Here is an example message conforming to the above format

```
[32, 713845233, {}, "com.myapp.mytopic1"]
```

3.1. Extensibility

Some WAMP messages contain `Options|dict` or `Details|dict` elements. This allows for future extensibility and implementations that only provide subsets of functionality by ignoring unimplemented attributes. Keys in `Options` and `Details` MUST be of type `string` and MUST match the regular expression `[a-z][a-z0-9_]{2,}` for WAMP predefined keys. Implementations MAY use implementation-specific keys that MUST match the regular expression `_[a-z0-9_]{3,}`. Attributes unknown to an implementation MUST be ignored.

3.2. No Polymorphism

For a given `MessageType` and number of message elements the expected types are uniquely defined. Hence there are no polymorphic messages in WAMP. This leads to a message parsing and validation control flow that is efficient, simple to implement and simple to code for rigorous message format checking.

3.3. Structure

The application payload (that is call arguments, call results, event payload etc) is always at the end of the message element list. The rationale is: Brokers and Dealers have no need to inspect (parse) the application payload. Their business is call/event routing. Having the application payload at the end of the list allows Brokers and Dealers to skip parsing it altogether. This can improve efficiency and performance.

3.4. Message Definitions

WAMP defines the following messages that are explained in detail in the following sections.

The messages concerning the WAMP session itself are mandatory for all Peers, i.e. a Client MUST implement `HELLO`, `ABORT` and `GOODBYE`, while a Router MUST implement `WELCOME`, `ABORT` and `GOODBYE`.

All other messages are mandatory per role, i.e. in an implementation that only provides a Client with the role of Publisher MUST additionally implement sending `PUBLISH` and receiving `PUBLISHED` and `ERROR` messages.

3.4.1. Session Lifecycle

3.4.1.1. HELLO

Sent by a Client to initiate opening of a WAMP session to a Router attaching to a Realm.

```
    [HELLO, Realm|uri, Details|dict]
```

3.4.1.2. WELCOME

Sent by a Router to accept a Client. The WAMP session is now open.

```
    [WELCOME, Session|id, Details|dict]
```

3.4.1.3. ABORT

Sent by a Peer*to abort the opening of a WAMP session. No response is expected.

```
    [ABORT, Details|dict, Reason|uri]
```

3.4.1.4. GOODBYE

Sent by a Peer to close a previously opened WAMP session. Must be echo'ed by the receiving Peer.

```
    [GOODBYE, Details|dict, Reason|uri]
```

3.4.1.5. ERROR

Error reply sent by a Peer as an error response to different kinds of requests.

```
    [ERROR, REQUEST.Type|int, REQUEST.Request|id, Details|dict, Error|
uri]

    [ERROR, REQUEST.Type|int, REQUEST.Request|id, Details|dict, Error|
uri,
        Arguments|list]

    [ERROR, REQUEST.Type|int, REQUEST.Request|id, Details|dict, Error|
uri,
        Arguments|list, ArgumentsKw|dict]
```

3.4.2. Publish & Subscribe

3.4.2.1. PUBLISH

Sent by a Publisher to a Broker to publish an event.

```
[PUBLISH, Request|id, Options|dict, Topic|uri]

[PUBLISH, Request|id, Options|dict, Topic|uri, Arguments|list]

[PUBLISH, Request|id, Options|dict, Topic|uri, Arguments|list,
    ArgumentsKw|dict]
```

3.4.2.2. PUBLISHED

Acknowledge sent by a Broker to a Publisher for acknowledged publications.

```
[PUBLISHED, PUBLISH.Request|id, Publication|id]
```

3.4.2.3. SUBSCRIBE

Subscribe request sent by a Subscriber to a Broker to subscribe to a topic.

```
[SUBSCRIBE, Request|id, Options|dict, Topic|uri]
```

3.4.2.4. SUBSCRIBED

Acknowledge sent by a Broker to a Subscriber to acknowledge a subscription.

```
[SUBSCRIBED, SUBSCRIBE.Request|id, Subscription|id]
```

3.4.2.5. UNSUBSCRIBE

Unsubscribe request sent by a Subscriber to a Broker to unsubscribe a subscription.

```
[UNSUBSCRIBE, Request|id, SUBSCRIBED.Subscription|id]
```

3.4.2.6. UNSUBSCRIBED

Acknowledge sent by a Broker to a Subscriber to acknowledge unsubscription.

```
[UNSUBSCRIBED, UNSUBSCRIBE.Request|id]
```

3.4.2.7. EVENT

Event dispatched by Broker to Subscribers for subscriptions the event was matching.

```
    [EVENT, SUBSCRIBED.Subscription|id, PUBLISHED.Publication|id,
Details|dict]

    [EVENT, SUBSCRIBED.Subscription|id, PUBLISHED.Publication|id,
Details|dict,
        PUBLISH.Arguments|list]

    [EVENT, SUBSCRIBED.Subscription|id, PUBLISHED.Publication|id,
Details|dict,
        PUBLISH.Arguments|list, PUBLISH.ArgumentsKw|dict]
```

An event is dispatched to a Subscriber for a given `Subscription|id` only once. On the other hand, a Subscriber that holds subscriptions with different `Subscription|id`s that all match a given event will receive the event on each matching subscription.

3.4.3. Routed Remote Procedure Calls

3.4.3.1. CALL

Call as originally issued by the Caller to the Dealer.

```
   [CALL, Request|id, Options|dict, Procedure|uri]

   [CALL, Request|id, Options|dict, Procedure|uri, Arguments|list]

   [CALL, Request|id, Options|dict, Procedure|uri, Arguments|list,
       ArgumentsKw|dict]
```

3.4.3.2. RESULT

Result of a call as returned by Dealer to Caller.

```
    [RESULT, CALL.Request|id, Details|dict]

    [RESULT, CALL.Request|id, Details|dict, YIELD.Arguments|list]

    [RESULT, CALL.Request|id, Details|dict, YIELD.Arguments|list,
        YIELD.ArgumentsKw|dict]
```

3.4.3.3. REGISTER

A Callees request to register an endpoint at a Dealer.

```
    [REGISTER, Request|id, Options|dict, Procedure|uri]
```

3.4.3.4. REGISTERED

Acknowledge sent by a Dealer to a Callee for successful registration.

```
    [REGISTERED, REGISTER.Request|id, Registration|id]
```

3.4.3.5. UNREGISTER

A Callees request to unregister a previously established registration.

```
    [UNREGISTER, Request|id, REGISTERED.Registration|id]
```

3.4.3.6. UNREGISTERED

Acknowledge sent by a Dealer to a Callee for successful unregistration.

```
    [UNREGISTERED, UNREGISTER.Request|id]
```

3.4.3.7. INVOCATION

Actual invocation of an endpoint sent by Dealer to a Callee.

```
    [INVOCATION, Request|id, REGISTERED.Registration|id, Details|dict]

    [INVOCATION, Request|id, REGISTERED.Registration|id, Details|dict,
        CALL.Arguments|list]

    [INVOCATION, Request|id, REGISTERED.Registration|id, Details|dict,
        CALL.Arguments|list, CALL.ArgumentsKw|dict]
```

3.4.3.8. YIELD

Actual yield from an endpoint sent by a Callee to Dealer.

```
    [YIELD, INVOCATION.Request|id, Options|dict]

    [YIELD, INVOCATION.Request|id, Options|dict, Arguments|list]

    [YIELD, INVOCATION.Request|id, Options|dict, Arguments|list,
ArgumentsKw|dict]
```

3.5. Message Codes and Direction

The following table lists the message type code for all messages defined in the WAMP basic profile and their direction between peer roles.

Reserved codes may be used to identify additional message types in future standards documents.

"Tx" indicates the message is sent by the respective role, and "Rx" indicates the message is received by the respective role.

Cod	Message	Pub	Brk	Subs	Calr	Dealr	Callee
1	HELLO	Tx	Rx	Tx	Tx	Rx	Tx
2	WELCOME	Rx	Tx	Rx	Rx	Tx	Rx
3	ABORT	Rx	TxRx	Rx	Rx	TxRx	Rx
6	GOODBYE	TxRx	TxRx	TxRx	TxRx	TxRx	TxRx
8	ERROR	Rx	Tx	Rx	Rx	TxRx	TxRx
16	PUBLISH	Tx	Rx				
17	PUBLISHED	Rx	Tx				
32	SUBSCRIBE		Rx	Tx			
33	SUBSCRIBED		Tx	Rx			
34	UNSUBSCRIBE		Rx	Tx			
35	UNSUBSCRIBED		Tx	Rx			
36	EVENT		Tx	Rx			
48	CALL				Tx	Rx	
50	RESULT				Rx	Tx	
64	REGISTER					Rx	Tx
65	REGISTERED					Tx	Rx
66	UNREGISTER					Rx	Tx
67	UNREGISTERED					Tx	Rx
68	INVOCATION					Tx	Rx
70	YIELD					Rx	Tx

Table 1

3.6. Extension Messages

WAMP uses type codes from the core range [0, 255]. Implementations MAY define and use implementation specific messages with message type codes from the extension message range [256, 1023]. For example, a router MAY implement router-to-router communication by using extension messages.

3.7. Empty Arguments and Keyword Arguments

Implementations SHOULD avoid sending empty `Arguments` lists.

E.g. a `CALL` message

```
    [CALL, Request|id, Options|dict, Procedure|uri, Arguments|list]
```

where `Arguments == []` SHOULD be avoided, and instead

```
    [CALL, Request|id, Options|dict, Procedure|uri]
```

SHOULD be sent.

Implementations SHOULD avoid sending empty `ArgumentsKw` dictionaries.

E.g. a `CALL` message

```
    [CALL, Request|id, Options|dict, Procedure|uri, Arguments|list,
ArgumentsKw|dict]
```

where `ArgumentsKw == {}` SHOULD be avoided, and instead

```
    [CALL, Request|id, Options|dict, Procedure|uri, Arguments|list]
```

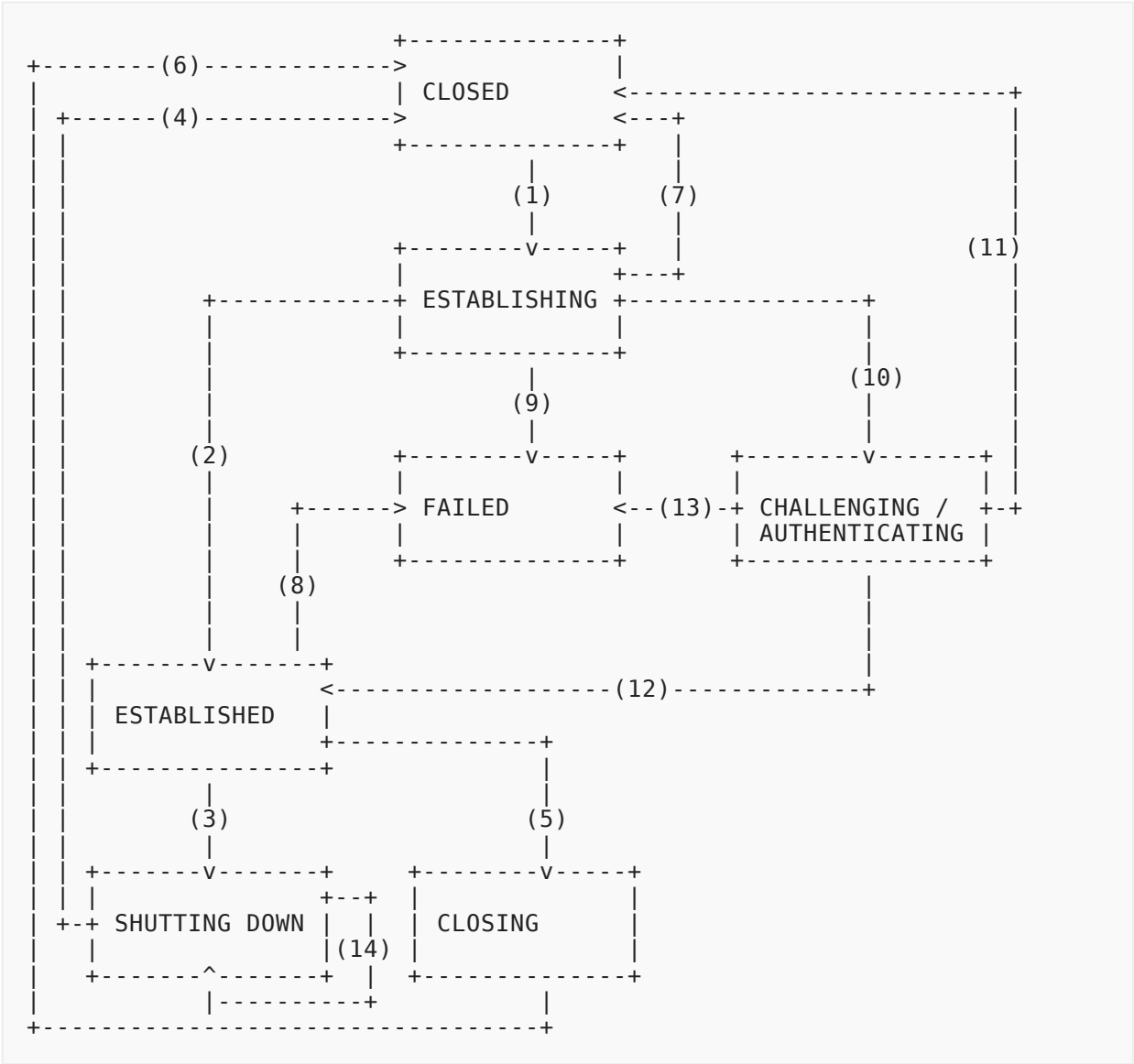
SHOULD be sent when `Arguments` is non-empty.

4. Sessions

The message flow between Clients and Routers for opening and closing WAMP sessions involves the following messages:

1. `HELLO`
2. `WELCOME`
3. `ABORT`
4. `GOODBYE`

The following state chart gives the states that a WAMP peer can be in during the session lifetime cycle.



The state transitions are listed in this table:

#	State
1	Sent HELLO
2	Received WELCOME
3	Sent GOODBYE
4	Received GOODBYE
5	Received GOODBYE

#	State
6	Sent GOODBYE
7	Received invalid HELLO / Send ABORT
8	Received HELLO or AUTHENTICATE
9	Received other
10	Received valid HELLO [needs authentication] / Send CHALLENGE
11	Received invalid AUTHENTICATE / Send ABORT
12	Received valid AUTHENTICATE / Send WELCOME
13	Received other
14	Received other / ignore

Table 2

4.1. Session Establishment

4.1.1. HELLO

After the underlying transport has been established, the opening of a WAMP session is initiated by the Client sending a `HELLO` message to the Router

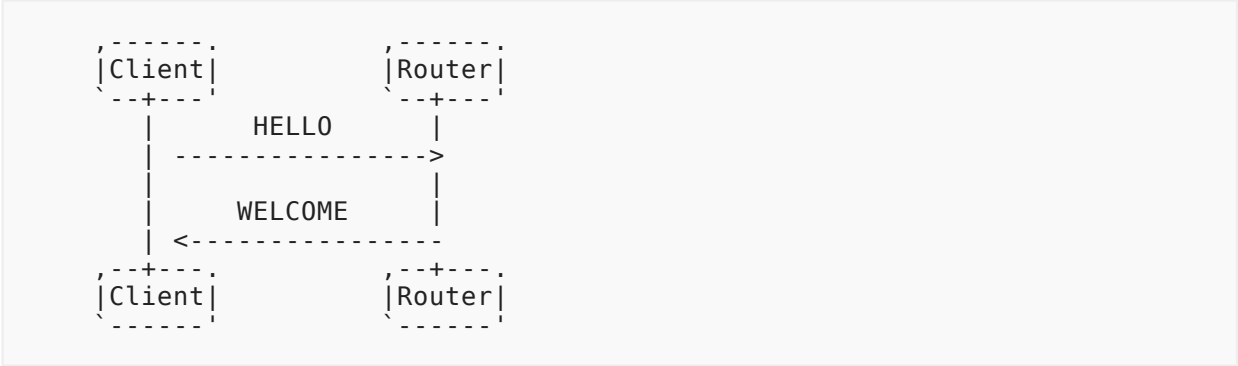
```
[HELLO, Realm|uri, Details|dict]
```

where

- `Realm` is a string identifying the realm this session should attach to
- `Details` is a dictionary that allows to provide additional opening information (see below).

The `HELLO` message MUST be the very first message sent by the Client after the transport has been established.

In the WAMP Basic Profile without session authentication the Router will reply with a `WELCOME` or `ABORT` message.



A WAMP session starts its lifetime when the Router has sent a `WELCOME` message to the Client, and ends when the underlying transport closes or when the session is closed explicitly by either peer sending the `GOODBYE` message (see below).

It is a protocol error to receive a second `HELLO` message during the lifetime of the session and the Peer MUST close the session if that happens.

Client: Role and Feature Announcement

WAMP uses *Role & Feature announcement* instead of *protocol versioning* to allow

- implementations only supporting subsets of functionality
- future extensibility

A Client must announce the roles it supports via `Hello.Details.roles|dict`, with a key mapping to a `Hello.Details.roles.<role>|dict` where `<role>` can be:

- `publisher`
- `subscriber`
- `caller`
- `callee`

A Client can support any combination of the above roles but must support at least one role.

The `<role>|dict` is a dictionary describing features supported by the peer for that role.

This MUST be empty for WAMP Basic Profile implementations, and MUST be used by implementations implementing parts of the Advanced Profile to list the specific set of features they support.

Example: A Client that implements the Publisher and Subscriber roles of the WAMP Basic Profile.

```
[1, "somerealm", {
  "roles": {
      "publisher": {},
      "subscriber": {}
  }
}]
```

Client: Agent Identification

When a software agent operates in a network protocol, it often identifies itself, its application type, operating system, software vendor, or software revision, by submitting a characteristic identification string to its operating peer.

Similar to what browsers do with the `User-Agent` HTTP header, both the `HELLO` and the `WELCOME` message MAY disclose the WAMP implementation in use to its peer:

```
HELLO.Details.agent|string
```

and

```
WELCOME.Details.agent|string
```

Example: A Client "HELLO" message.

```
[1, "somerealm", {
     "agent": "AutobahnJS-0.9.14",
     "roles": {
        "subscriber": {},
        "publisher": {}
     }
}]
```

Example: A Router "WELCOME" message.

```
[2, 9129137332, {
    "agent": "Crossbar.io-0.10.11",
    "roles": {
      "broker": {}
    }
}]
```

4.1.2. WELCOME

A Router completes the opening of a WAMP session by sending a `WELCOME` reply message to the Client.

```
[WELCOME, Session|id, Details|dict]
```

where

- `Session` MUST be a randomly generated ID specific to the WAMP session. This applies for the lifetime of the session.
- `Details` is a dictionary that allows to provide additional information regarding the open session (see below).

In the WAMP Basic Profile without session authentication, a `WELCOME` message MUST be the first message sent by the Router, directly in response to a `HELLO` message received from the Client. Extensions in the Advanced Profile MAY include intermediate steps and messages for authentication.

> Note. The behavior if a requested `Realm` does not presently exist is router-specific. A router may e.g. automatically create the realm, or deny the establishment of the session with a `ABORT` reply message.

Router: Role and Feature Announcement

Similar to a Client announcing Roles and Features supported in the `HELLO message, a Router announces its supported Roles and Features in the `WELCOME` message.

A Router MUST announce the roles it supports via `Welcome.Details.roles|dict`, with a key mapping to a `Welcome.Details.roles.<role>|dict` where `<role>` can be:

- `broker`
- `dealer`

A Router must support at least one role, and MAY support both roles.

The `<role>|dict` is a dictionary describing features supported by the peer for that role. With WAMP Basic Profile implementations, this MUST be empty, but MUST be used by implementations implementing parts of the Advanced Profile to list the specific set of features they support

Example: A Router implementing the Broker role of the WAMP Basic Profile.

```
[2, 9129137332, {
   "roles": {
      "broker": {}
   }
}]
```

4.1.3. ABORT

Both the Router and the Client may abort a WAMP session by sending an `ABORT` message.

```
[ABORT, Details|dict, Reason|uri]
```

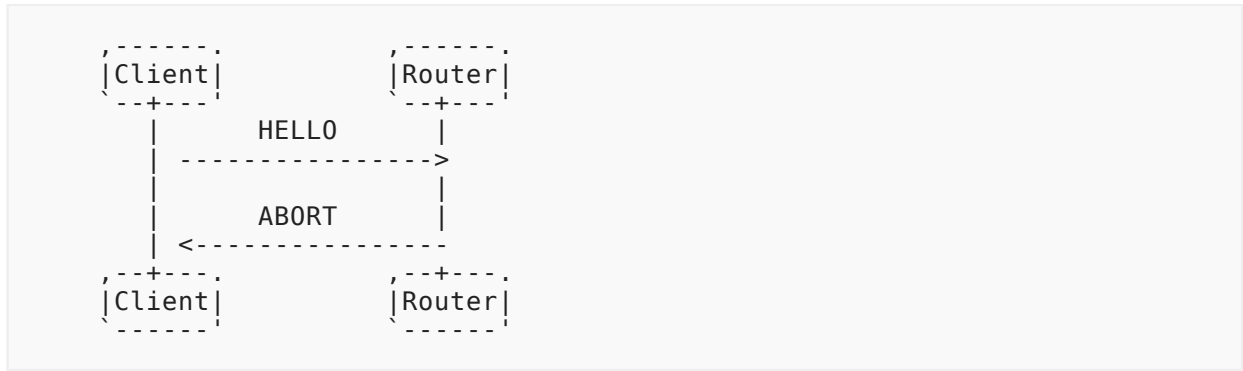
where

- `Reason` MUST be a URI.
- `Details` MUST be a dictionary that allows to provide additional, optional closing information (see below).

No response to an `ABORT` message is expected.

There are few scenarios, when (U+00A0)`ABORT` is used:

- During session opening, if peer decided to abort connect.



Example

```
[3, {"message": "The realm does not exist."},
    "wamp.error.no_such_realm"]
```

- After session is opened, when (U+00A0)protocol violation happens (see "Protocol errors" section).

Examples

- Router received second HELLO message.

```
[3, {"message":
        "Received HELLO message after session was established."},
    "wamp.error.protocol_violation"]
```

- Client peer received second WELCOME message

```
[3, {"message":
        "Received WELCOME message after session was established."},
    "wamp.error.protocol_violation"]
```

4.2. Session Closing

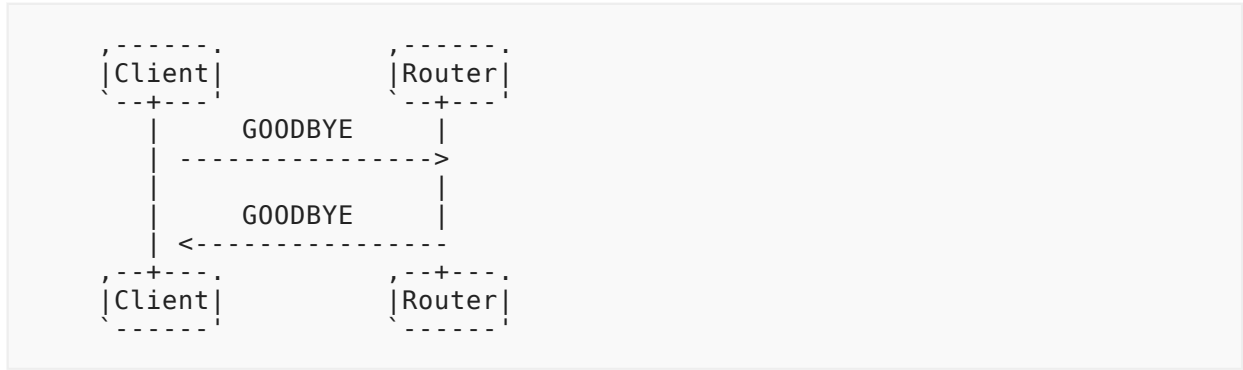
4.2.1. GOODBYE

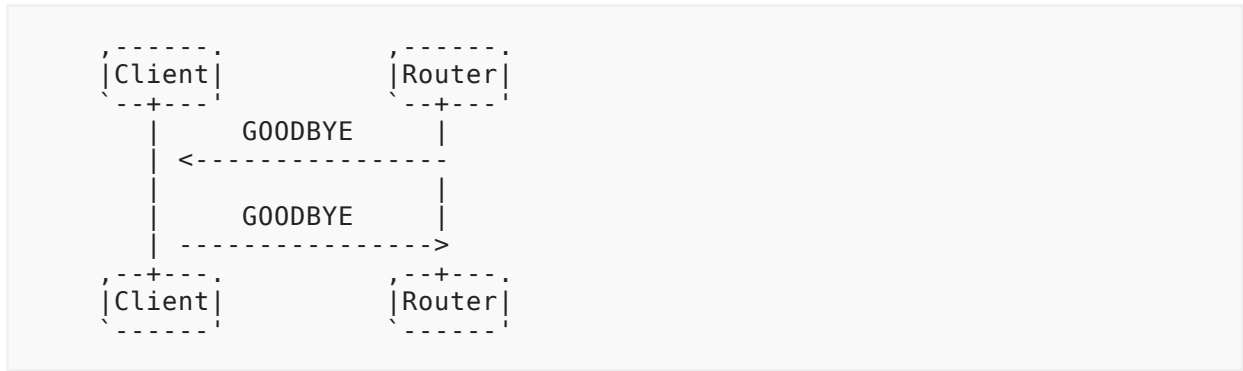
A WAMP session starts its lifetime with the Router sending a `WELCOME` message to the Client and ends when the underlying transport disappears or when the WAMP session is closed explicitly by a `GOODBYE` message sent by one Peer and a `GOODBYE` message sent from the other Peer in response.

```
[GOODBYE, Details|dict, Reason|uri]
```

where

- `Reason` MUST be a URI.
- `Details` MUST be a dictionary that allows to provide additional, optional closing information (see below).





Example. One Peer initiates closing

```
[6, {"message": "The host is shutting down now."},
    "wamp.close.system_shutdown"]
```

and the other peer replies

```
[6, {}, "wamp.close.goodbye_and_out"]
```

Example. One Peer initiates closing

```
[6, {}, "wamp.close.close_realm"]
```

and the other peer replies

```
[6, {}, "wamp.close.goodbye_and_out"]
```

Difference between ABORT and GOODBYE

The differences between `ABORT` and `GOODBYE` messages is that (U+00A0)`ABORT` is never replied to by a Peer, whereas `GOODBYE` must be replied to by the receiving Peer.

> Though `ABORT` and `GOODBYE` are structurally identical, using different message types serves to reduce overloaded meaning of messages and simplify message handling code.

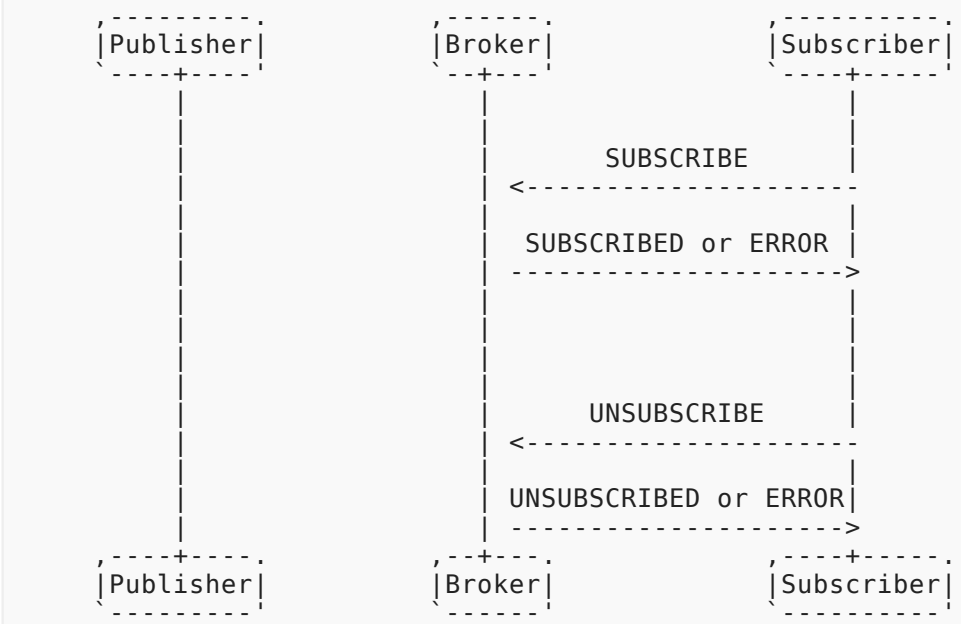
5. Publish and Subscribe

All of the following features for Publish & Subscribe are mandatory for WAMP Basic Profile implementations supporting the respective roles, i.e. *Publisher*, *Subscriber* and *Broker*.

5.1. Subscribing and Unsubscribing

The message flow between Clients implementing the role of Subscriber and Routers implementing the role of Broker for subscribing and unsubscribing involves the following messages:

1. `SUBSCRIBE`
2. `SUBSCRIBED`
3. `UNSUBSCRIBE`
4. `UNSUBSCRIBED`
5. `ERROR`



A Subscriber may subscribe to zero, one or more topics, and a Publisher publishes to topics without knowledge of subscribers.

Upon subscribing to a topic via the `SUBSCRIBE` message, a Subscriber will receive any future events published to the respective topic by Publishers, and will receive those events asynchronously.

A subscription lasts for the duration of a session, unless a Subscriber opts out from a previously established subscription via the `UNSUBSCRIBE` message.

> A Subscriber may have more than one event handler attached to the same subscription. This can be implemented in different ways: a) a Subscriber can recognize itself that it is already subscribed and just attach another handler to the subscription for incoming events, b) or it can send a new `SUBSCRIBE` message to broker (as it would be first) and upon receiving a `SUBSCRIBED.Subscription|id` it already knows about, attach the handler to the existing subscription

5.1.1. SUBSCRIBE

A Subscriber communicates its interest in a topic to a Broker by sending a `SUBSCRIBE` message:

```
[SUBSCRIBE, Request|id, Options|dict, Topic|uri]
```

where

- `Request` MUST be a random, ephemeral ID chosen by the Subscriber and used to correlate the Broker's response with the request.
- `Options` MUST be a dictionary that allows to provide additional subscription request details in a extensible way. This is described further below.
- `Topic` is the topic the Subscriber wants to subscribe to and MUST be a URI.

Example

```
    [32, 713845233, {}, "com.myapp.mytopic1"]
```

A Broker, receiving a `SUBSCRIBE` message, can fullfill or reject the subscription, so it answers with `SUBSCRIBED` or `ERROR` messages.

5.1.2. SUBSCRIBED

If the Broker is able to fulfill and allow the subscription, it answers by sending a `SUBSCRIBED` message to the Subscriber

```
    [SUBSCRIBED, SUBSCRIBE.Request|id, Subscription|id]
```

where

- `SUBSCRIBE.Request` MUST be the ID from the original request.
- `Subscription` MUST be an ID chosen by the Broker for the subscription.

Example

```
    [33, 713845233, 5512315355]
```

> Note. The `Subscription` ID chosen by the broker need not be unique to the subscription of a single Subscriber, but may be assigned to the `Topic`, or the combination of the `Topic` and some or all `Options`, such as the topic pattern matching method to be used. Then this ID may be sent to all Subscribers for the `Topic` or `Topic` / `Options` combination. This allows the Broker to serialize an event to be delivered only once for all actual receivers of the event.
>
> In case of receiving a `SUBSCRIBE` message from the same Subscriber and to already subscribed topic, Broker should answer with `SUBSCRIBED` message, containing the existing `Subscription|id`.

5.1.3. Subscribe ERROR

When the request for subscription cannot be fulfilled by the Broker, the Broker sends back an `ERROR` message to the Subscriber

```
[ERROR, SUBSCRIBE, SUBSCRIBE.Request|id, Details|dict, Error|uri]
```

where

- `SUBSCRIBE.Request` MUST be the ID from the original request.
- `Error` MUST be a URI that gives the error of why the request could not be fulfilled.

Example

```
[8, 32, 713845233, {}, "wamp.error.not_authorized"]
```

5.1.4. UNSUBSCRIBE

When a Subscriber is no longer interested in receiving events for a subscription it sends an `UNSUBSCRIBE` message

```
[UNSUBSCRIBE, Request|id, SUBSCRIBED.Subscription|id]
```

where

- `Request` MUST be a random, ephemeral ID chosen by the Subscriber and used to correlate the Broker's response with the request.
- `SUBSCRIBED.Subscription` MUST be the ID for the subscription to unsubscribe from, originally handed out by the Broker to the Subscriber.

Example

```
[34, 85346237, 5512315355]
```

5.1.5. UNSUBSCRIBED

Upon successful unsubscription, the Broker sends an `UNSUBSCRIBED` message to the Subscriber

```
[UNSUBSCRIBED, UNSUBSCRIBE.Request|id]
```

where

- `UNSUBSCRIBE.Request` MUST be the ID from the original request.

Example

```
[35, 85346237]
```

5.1.6. Unsubscribe ERROR

When the request fails, the Broker sends an `ERROR`

```
[ERROR, UNSUBSCRIBE, UNSUBSCRIBE.Request|id, Details|dict, Error|uri]
```

where

- `UNSUBSCRIBE.Request` MUST be the ID from the original request.
- `Error` MUST be a URI that gives the error of why the request could not be fulfilled.

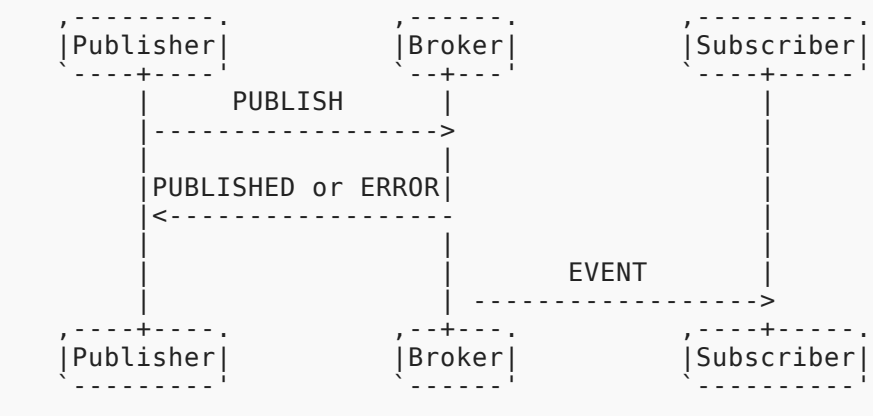
Example

```
[8, 34, 85346237, {}, "wamp.error.no_such_subscription"]
```

5.2. Publishing and Events

The message flow between Publishers, a Broker and Subscribers for publishing to topics and dispatching events involves the following messages:

1. `PUBLISH`
2. `PUBLISHED`
3. `EVENT`
4. `ERROR`



5.2.1. PUBLISH

When a Publisher requests to publish an event to some topic, it sends a `PUBLISH` message to a Broker:

```
[PUBLISH, Request|id, Options|dict, Topic|uri]
```

or

```
[PUBLISH, Request|id, Options|dict, Topic|uri, Arguments|list]
```

or

```
[PUBLISH, Request|id, Options|dict, Topic|uri, Arguments|list,
    ArgumentsKw|dict]
```

where

- `Request` is a sequential ID in the *session scope*, incremented by the Publisher and used to correlate the Broker's response with the request.
- `Options` is a dictionary that allows to provide additional publication request details in an extensible way. This is described further below.
- `Topic` is the topic published to.
- `Arguments` is a list of application-level event payload elements. The list may be of zero length.
- `ArgumentsKw` is an optional dictionary containing application-level event payload, provided as keyword arguments. The dictionary may be empty.

If the Broker allows and is able to fulfill the publication, the Broker will send the event to all current Subscribers of the topic of the published event.

By default, publications are unacknowledged, and the Broker will not respond, whether the publication was successful indeed or not. This behavior can be changed with the option `PUBLISH.Options.acknowledge|bool` (see below).

Example

```
[16, 239714735, {}, "com.myapp.mytopic1"]
```

Example

```
[16, 239714735, {}, "com.myapp.mytopic1", ["Hello, world!"]]
```

Example

```
[16, 239714735, {}, "com.myapp.mytopic1", [], {"color": "orange",
    "sizes": [23, 42, 7]}]
```

5.2.2. PUBLISHED

If the Broker is able to fulfill and allowing the publication, and `PUBLISH.Options.acknowledge == true`, the Broker replies by sending a `PUBLISHED` message to the Publisher:

```
[PUBLISHED, PUBLISH.Request|id, Publication|id]
```

where

- `PUBLISH.Request` is the ID from the original publication request.
- `Publication` is a ID chosen by the Broker for the publication.

Example

```
    [17, 239714735, 4429313566]
```

5.2.3. Publish ERROR

When the request for publication cannot be fulfilled by the Broker, and `PUBLISH.Options.acknowledge == true`, the Broker sends back an `ERROR` message to the Publisher

```
    [ERROR, PUBLISH, PUBLISH.Request|id, Details|dict, Error|uri]
```

where

- `PUBLISH.Request` is the ID from the original publication request.
- `Error` is a URI that gives the error of why the request could not be fulfilled.

Example

```
    [8, 16, 239714735, {}, "wamp.error.not_authorized"]
```

5.2.4. EVENT

When a publication is successful and a Broker dispatches the event, it determines a list of receivers for the event based on Subscribers for the topic published to and, possibly, other information in the event.

Note that the Publisher of an event will never receive the published event even if the Publisher is also a Subscriber of the topic published to.

> The Advanced Profile provides options for more detailed control over publication.

When a Subscriber is deemed to be a receiver, the Broker sends the Subscriber an `EVENT` message:

```
    [EVENT, SUBSCRIBED.Subscription|id, PUBLISHED.Publication|id,
 Details|dict]
```

or

```
    [EVENT, SUBSCRIBED.Subscription|id, PUBLISHED.Publication|id,
Details|dict,
        PUBLISH.Arguments|list]
```

or

```
    [EVENT, SUBSCRIBED.Subscription|id, PUBLISHED.Publication|id,
Details|dict,
        PUBLISH.Arguments|list, PUBLISH.ArgumentKw|dict]
```

where

- `SUBSCRIBED.Subscription` is the ID for the subscription under which the Subscriber receives the event - the ID for the subscription originally handed out by the Broker to the Subscribe*.
- `PUBLISHED.Publication` is the ID of the publication of the published event.
- `Details` is a dictionary that allows the Broker to provide additional event details in a extensible way. This is described further below.
- `PUBLISH.Arguments` is the application-level event payload that was provided with the original publication request.
- `PUBLISH.ArgumentKw` is the application-level event payload that was provided with the original publication request.

Example

```
    [36, 5512315355, 4429313566, {}]
```

Example

```
    [36, 5512315355, 4429313566, {}, ["Hello, world!"]]
```

Example

```
    [36, 5512315355, 4429313566, {}, [], {"color": "orange", "sizes":
[23, 42, 7]}]
```

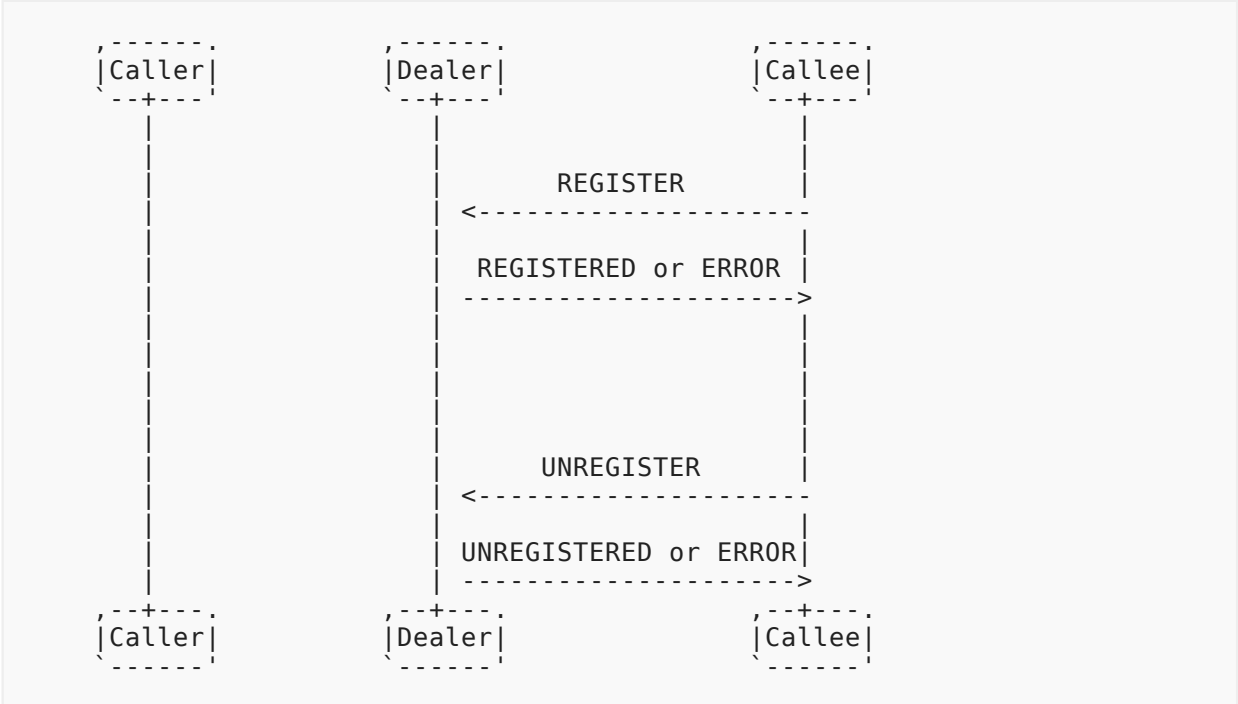
6. Remote Procedure Calls

All of the following features for Remote Procedure Calls are mandatory for WAMP Basic Profile implementations supporting the respective roles.

6.1. Registering and Unregistering

The message flow between Callees and a Dealer for registering and unregistering endpoints to be called over RPC involves the following messages:

1. `REGISTER`
2. `REGISTERED`
3. `UNREGISTER`
4. `UNREGISTERED`
5. `ERROR`



6.1.1. REGISTER

A Callee announces the availability of an endpoint implementing a procedure with a Dealer by sending a `REGISTER` message:

```
[REGISTER, Request|id, Options|dict, Procedure|uri]
```

where

- `Request` is a sequential ID in the *session scope*, incremented by the Callee and used to correlate the Dealer's response with the request.
- `Options` is a dictionary that allows to provide additional registration request details in a extensible way. This is described further below.
- `Procedure`is the procedure the Callee wants to register

Example

```
[64, 25349185, {}, "com.myapp.myprocedure1"]
```

6.1.2. REGISTERED

If the Dealer is able to fulfill and allowing the registration, it answers by sending a `REGISTERED` message to the `Callee`:

```
[REGISTERED, REGISTER.Request|id, Registration|id]
```

where

- `REGISTER.Request` is the ID from the original request.
- `Registration` is an ID chosen by the Dealer for the registration.

Example

```
[65, 25349185, 2103333224]
```

6.1.3. Register ERROR

When the request for registration cannot be fulfilled by the Dealer, the Dealer sends back an `ERROR` message to the Callee:

```
[ERROR, REGISTER, REGISTER.Request|id, Details|dict, Error|uri]
```

where

- `REGISTER.Request` is the ID from the original request.
- `Error` is a URI that gives the error of why the request could not be fulfilled.

Example

```
[8, 64, 25349185, {}, "wamp.error.procedure_already_exists"]
```

6.1.4. UNREGISTER

When a Callee is no longer willing to provide an implementation of the registered procedure, it sends an `UNREGISTER` message to the Dealer:

```
[UNREGISTER, Request|id, REGISTERED.Registration|id]
```

where

- `Request` is a sequential ID in the *session scope*, incremented by the Callee and used to correlate the Dealer's response with the request.
- `REGISTERED.Registration` is the ID for the registration to revoke, originally handed out by the Dealer to the Callee.

Example

```
[66, 788923562, 2103333224]
```

6.1.5. UNREGISTERED

Upon successful unregistration, the Dealer sends an `UNREGISTERED` message to the Callee:

```
[UNREGISTERED, UNREGISTER.Request|id]
```

where

- `UNREGISTER.Request` is the ID from the original request.

Example

```
[67, 788923562]
```

6.1.6. Unregister ERROR

When the unregistration request fails, the Dealer sends an `ERROR` message:

```
[ERROR, UNREGISTER, UNREGISTER.Request|id, Details|dict, Error|uri]
```

where

- `UNREGISTER.Request` is the ID from the original request.
- `Error` is a URI that gives the error of why the request could not be fulfilled.

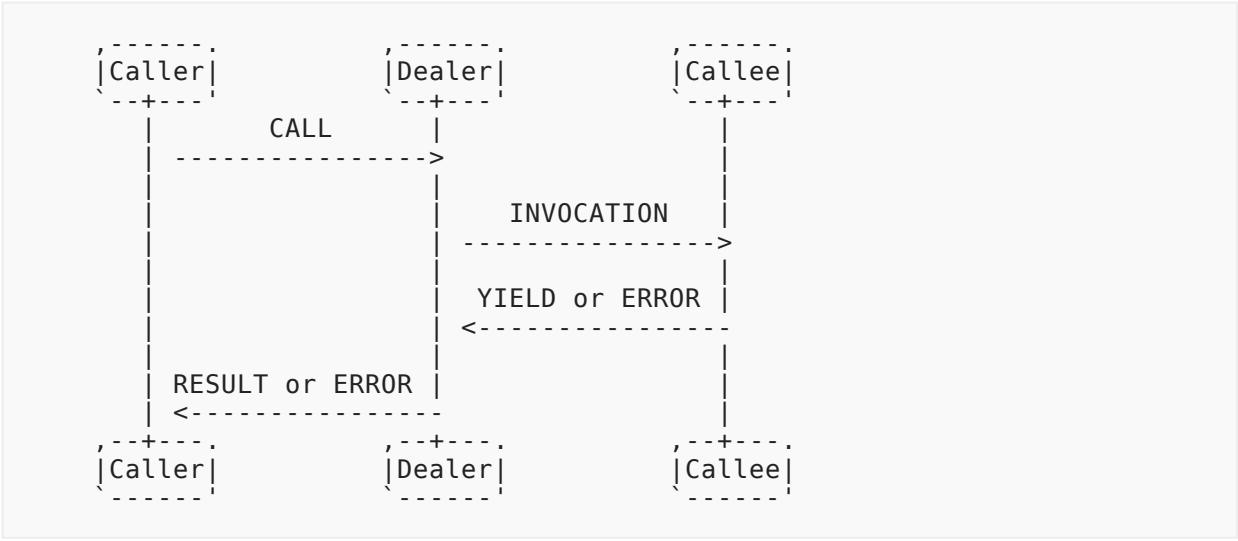
Example

```
[8, 66, 788923562, {}, "wamp.error.no_such_registration"]
```

6.2. Calling and Invocations

The message flow between Callers, a Dealer and Callees for calling procedures and invoking endpoints involves the following messages:

1. `CALL`
2. `RESULT`
3. `INVOCATION`
4. `YIELD`
5. `ERROR`



The execution of remote procedure calls is asynchronous, and there may be more than one call outstanding. A call is called outstanding (from the point of view of the Caller), when a (final) result or error has not yet been received by the Caller.

6.2.1. CALL

When a Caller wishes to call a remote procedure, it sends a `CALL` message to a Dealer:

```
[CALL, Request|id, Options|dict, Procedure|uri]
```

or

```
[CALL, Request|id, Options|dict, Procedure|uri, Arguments|list]
```

or

```
[CALL, Request|id, Options|dict, Procedure|uri, Arguments|list,
    ArgumentsKw|dict]
```

where

- `Request` is a sequential ID in the *session scope*, incremented by the Caller and used to correlate the Dealer's response with the request.
- `Options` is a dictionary that allows to provide additional call request details in an extensible way. This is described further below.
- `Procedure` is the URI of the procedure to be called.
- `Arguments` is a list of positional call arguments (each of arbitrary type). The list may be of zero length.
- `ArgumentsKw` is a dictionary of keyword call arguments (each of arbitrary type). The dictionary may be empty.

Example

```
[48, 7814135, {}, "com.myapp.ping"]
```

Example

```
[48, 7814135, {}, "com.myapp.echo", ["Hello, world!"]]
```

Example

```
[48, 7814135, {}, "com.myapp.add2", [23, 7]]
```

Example

```
[48, 7814135, {}, "com.myapp.user.new", ["johnny"],
    {"firstname": "John", "surname": "Doe"}]
```

6.2.2. INVOCATION

If the Dealer is able to fulfill (mediate) the call and it allows the call, it sends a `INVOCATION` message to the respective Callee implementing the procedure:

```
[INVOCATION, Request|id, REGISTERED.Registration|id, Details|dict]
```

or

```
[INVOCATION, Request|id, REGISTERED.Registration|id, Details|dict,
    CALL.Arguments|list]
```

or

```
[INVOCATION, Request|id, REGISTERED.Registration|id, Details|dict,
    CALL.Arguments|list, CALL.ArgumentsKw|dict]
```

where

- `Request` is a sequential ID in the *session scope*, incremented by the Dealer and used to correlate the *Callee's* response with the request.
- `REGISTERED.Registration` is the registration ID under which the procedure was registered at the Dealer.
- `Details` is a dictionary that allows to provide additional invocation request details in an extensible way. This is described further below.
- `CALL.Arguments` is the original list of positional call arguments as provided by the Caller.
- `CALL.ArgumentsKw` is the original dictionary of keyword call arguments as provided by the Caller.

Example

```
    [68, 6131533, 9823526, {}]
```

Example

```
    [68, 6131533, 9823527, {}, ["Hello, world!"]]
```

Example

```
    [68, 6131533, 9823528, {}, [23, 7]]
```

Example

```
    [68, 6131533, 9823529, {}, ["johnny"], {"firstname": "John",
"surname": "Doe"}]
```

6.2.3. YIELD

If the Callee is able to successfully process and finish the execution of the call, it answers by sending a `YIELD` message to the Dealer:

```
    [YIELD, INVOCATION.Request|id, Options|dict]
```

or

```
    [YIELD, INVOCATION.Request|id, Options|dict, Arguments|list]
```

or

```
    [YIELD, INVOCATION.Request|id, Options|dict, Arguments|list,
ArgumentsKw|dict]
```

where

- `INVOCATION.Request` is the ID from the original invocation request.
- `Options`is a dictionary that allows to provide additional options.
- `Arguments` is a list of positional result elements (each of arbitrary type). The list may be of zero length.
- `ArgumentsKw` is a dictionary of keyword result elements (each of arbitrary type). The dictionary may be empty.

Example

```
    [70, 6131533, {}]
```

Example

```
[70, 6131533, {}, ["Hello, world!"]]
```

Example

```
[70, 6131533, {}, [30]]
```

Example

```
[70, 6131533, {}, [], {"userid": 123, "karma": 10}]
```

6.2.4. RESULT

The Dealer will then send a `RESULT` message to the original Caller:

```
[RESULT, CALL.Request|id, Details|dict]
```

or

```
[RESULT, CALL.Request|id, Details|dict, YIELD.Arguments|list]
```

or

```
[RESULT, CALL.Request|id, Details|dict, YIELD.Arguments|list,
    YIELD.ArgumentsKw|dict]
```

where

- `CALL.Request` is the ID from the original call request.
- `Details` is a dictionary of additional details.
- `YIELD.Arguments` is the original list of positional result elements as returned by the Callee.
- `YIELD.ArgumentsKw` is the original dictionary of keyword result elements as returned by the Callee.

Example

```
[50, 7814135, {}]
```

Example

```
[50, 7814135, {}, ["Hello, world!"]]
```

Example

```
[50, 7814135, {}, [30]]
```

Example

```
    [50, 7814135, {}, [], {"userid": 123, "karma": 10}]
```

6.2.5. Invocation ERROR

If the Callee is unable to process or finish the execution of the call, or the application code implementing the procedure raises an exception or otherwise runs into an error, the Callee sends an `ERROR` message to the Dealer:

```
    [ERROR, INVOCATION, INVOCATION.Request|id, Details|dict, Error|uri]
```

or

```
    [ERROR, INVOCATION, INVOCATION.Request|id, Details|dict, Error|uri,
Arguments|list]
```

or

```
    [ERROR, INVOCATION, INVOCATION.Request|id, Details|dict, Error|uri,
Arguments|list,
        ArgumentsKw|dict]
```

where

- `INVOCATION.Request` is the ID from the original `INVOCATION` request previously sent by the Dealer to the Callee.
- `Details` is a dictionary with additional error details.
- `Error` is a URI that identifies the error of why the request could not be fulfilled.
- `Arguments` is a list containing arbitrary, application defined, positional error information. This will be forwarded by the Dealer to the Caller that initiated the call.
- `ArgumentsKw` is a dictionary containing arbitrary, application defined, keyword-based error information. This will be forwarded by the Dealer to the Caller that initiated the call.

Example

```
    [8, 68, 6131533, {}, "com.myapp.error.object_write_protected",
        ["Object is write protected."], {"severity": 3}]
```

6.2.6. Call ERROR

The Dealer will then send a `ERROR` message to the original Caller:

```
    [ERROR, CALL, CALL.Request|id, Details|dict, Error|uri]
```

or

```
    [ERROR, CALL, CALL.Request|id, Details|dict, Error|uri, Arguments|
list]
```

or

```
    [ERROR, CALL, CALL.Request|id, Details|dict, Error|uri, Arguments|
list,
        ArgumentsKw|dict]
```

where

- `CALL.Request` is the ID from the original `CALL` request sent by the Caller to the Dealer.
- `Details` is a dictionary with additional error details.
- `Error` is a URI identifying the type of error as returned by the Callee to the Dealer.
- `Arguments` is a list containing the original error payload list as returned by the Callee to the Dealer.
- `ArgumentsKw` is a dictionary containing the original error payload dictionary as returned by the Callee to the Dealer

Example

```
    [8, 48, 7814135, {}, "com.myapp.error.object_write_protected",
        ["Object is write protected."], {"severity": 3}]
```

If the original call already failed at the Dealer **before** the call would have been forwarded to any Callee, the Dealer will send an `ERROR` message to the Caller:

```
    [ERROR, CALL, CALL.Request|id, Details|dict, Error|uri]
```

Example

```
    [8, 48, 7814135, {}, "wamp.error.no_such_procedure"]
```

7. Security Model

The following discusses the security model for the Basic Profile. Any changes or extensions to this for the Advanced Profile are discussed further on as part of the Advanced Profile definition.

All WAMP implementations, in particular Routers MUST support the following ordering guarantees.

A WAMP Advanced Profile may provide applications options to relax ordering guarantees, in particular with distributed calls.

7.1. Ordering Guarantees

Publish & Subscribe Ordering

Regarding **Publish & Subscribe**, the ordering guarantees are as follows:

If *Subscriber A* is subscribed to both **Topic 1** and **Topic 2**, and *Publisher B* first publishes an **Event 1** to **Topic 1** and then an **Event 2** to **Topic 2**, then *Subscriber A* will first receive **Event 1** and then **Event 2**. This also holds if **Topic 1** and **Topic 2** are identical.

In other words, WAMP guarantees ordering of events between any given *pair* of Publisher and Subscriber.

Further, if *Subscriber A* subscribes to **Topic 1**, the `SUBSCRIBED` message will be sent by the *Broker* to *Subscriber A* before any `EVENT` message for **Topic 1**.

There is no guarantee regarding the order of return for multiple subsequent subscribe requests. A subscribe request might require the *Broker* to do a time-consuming lookup in some database, whereas another subscribe request second might be permissible immediately.

Remote Procedure Call Ordering

Regarding **Remote Procedure Calls**, the ordering guarantees are as follows:

If *Callee A* has registered endpoints for both **Procedure 1** and **Procedure 2**, and *Caller B* first issues a **Call 1** to **Procedure 1** and then a **Call 2** to **Procedure 2**, and both calls are routed to *Callee A*, then *Callee A* will first receive an invocation corresponding to **Call 1** and then **Call 2**. This also holds if **Procedure 1** and **Procedure 2** are identical.

In other words, WAMP guarantees ordering of invocations between any given *pair* of Caller and Callee.

There are no guarantees on the order of call results and errors in relation to *different* calls, since the execution of calls upon different invocations of endpoints in Callees are running independently. A first call might require an expensive, long-running computation, whereas a second, subsequent call might finish immediately.

Further, if *Callee A* registers for **Procedure 1**, the `REGISTERED` message will be sent by *Dealer* to *Callee A* before any `INVOCATION` message for **Procedure 1**.

There is no guarantee regarding the order of return for multiple subsequent register requests. A register request might require the *Broker* to do a time-consuming lookup in some database, whereas another register request second might be permissible immediately.

7.2. Transport Encryption and Integrity

WAMP transports may provide (optional) transport-level encryption and integrity verification. If so, encryption and integrity is point-to-point: between a Client and the Router it is connected to.

Transport-level encryption and integrity is solely at the transport-level and transparent to WAMP. WAMP itself deliberately does not specify any kind of transport-level encryption.

Implementations that offer TCP based transport such as WAMP-over-WebSocket or WAMP-over-RawSocket SHOULD implement Transport Layer Security (TLS).

WAMP deployments are encouraged to stick to a TLS-only policy with the TLS code and setup being hardened.

Further, when a Client connects to a Router over a local-only transport such as Unix domain sockets, the integrity of the data transmitted is implicit (the OS kernel is trusted), and the privacy of the data transmitted can be assured using file system permissions (no one can tap a Unix domain socket without appropriate permissions or being root).

7.3. Router Authentication

To authenticate Routers to Clients, deployments MUST run TLS and Clients MUST verify the Router server certificate presented. WAMP itself does not provide mechanisms to authenticate a Router (only a Client).

The verification of the Router server certificate can happen

1. against a certificate trust database that comes with the Clients operating system
2. against an issuing certificate/key hard-wired into the Client
3. by using new mechanisms like DNS-based Authentication of Named Enitities (DNSSEC)/TLSA

Further, when a Client connects to a Router over a local-only transport such as Unix domain sockets, the file system permissions can be used to create implicit trust. E.g. if only the OS user under which the Router runs has the permission to create a Unix domain socket under a specific path, Clients connecting to that path can trust in the router authenticity.

7.4. Client Authentication

Authentication of a Client to a Router at the WAMP level is not part of the basic profile.

When running over TLS, a Router MAY authenticate a Client at the transport level by doing a *client certificate based authentication*.

7.5. Routers are trusted

Routers are *trusted* by Clients. In particular, Routers can read (and modify) any application payload transmitted in events, calls, call results and call errors (the `Arguments` or `ArgumentsKw` message fields).

Hence, Routers do not provide confidentiality with respect to application payload, and also do not provide authenticity or integrity of application payloads that could be verified by a receiving Client.

Routers need to read the application payloads in cases of automatic conversion between different serialization formats.

Further, Routers are trusted to **actually perform** routing as specified. E.g. a Client that publishes an event has to trust a Router that the event is actually dispatched to all (eligible) Subscribers by the Router.

A rogue Router might deny normal routing operation without a Client taking notice.

8. Basic Profile URIs

WAMP pre-defines the following error URIs for the **Basic Profile**. WAMP peers SHOULD only use the defined error messages.

Incorrect URIs

When a Peer provides an incorrect URI for any URI-based attribute of a WAMP message (e.g. realm, topic), then the other Peer MUST respond with an `ERROR` message and give the following *Error URI*:

```
wamp.error.invalid_uri
```

Interaction

Peer provided an incorrect URI for any URI-based attribute of WAMP message, such as realm, topic or procedure

```
wamp.error.invalid_uri
```

A Dealer could not perform a call, since no procedure is currently registered under the given URI.

```
wamp.error.no_such_procedure
```

A procedure could not be registered, since a procedure with the given URI is already registered.

```
wamp.error.procedure_already_exists
```

A Dealer could not perform an unregister, since the given registration is not active.

```
wamp.error.no_such_registration
```

A Broker could not perform an unsubscribe, since the given subscription is not active.

```
wamp.error.no_such_subscription
```

A call failed since the given argument types or values are not acceptable to the called procedure. In this case the Callee may throw this error. Alternatively a Router may throw this error if it performed *payload validation* of a call, call result, call error or publish, and the payload did not conform to the requirements.

```
wamp.error.invalid_argument
```

Session Close

The Peer is shutting down completely - used as a `GOODBYE` (or `ABORT`) reason.

```
wamp.close.system_shutdown
```

The Peer want to leave the realm - used as a `GOODBYE` reason.

```
wamp.close.close_realm
```

A Peer acknowledges ending of a session - used as a `GOODBYE` reply reason.

```
wamp.close.goodbye_and_out
```

A Peer received invalid WAMP protocol message (e.g. `HELLO` message after session was already established) - used as a `ABORT` reply reason.

```
wamp.error.protocol_violation
```

Authorization

A join, call, register, publish or subscribe failed, since the Peer is not authorized to perform the operation.

```
wamp.error.not_authorized
```

A Dealer or Broker could not determine if the Peer is authorized to perform a join, call, register, publish or subscribe, since the authorization operation *itself* failed. E.g. a custom authorizer did run into an error.

```
wamp.error.authorization_failed
```

Peer wanted to join a non-existing realm (and the Router did not allow to auto-create the realm).

```
wamp.error.no_such_realm
```

A Peer was to be authenticated under a Role that does not (or no longer) exists on the Router. For example, the Peer was successfully authenticated, but the Role configured does not exists - hence there is some misconfiguration in the Router.

```
wamp.error.no_such_role
```

9. WAMP Advanced Profile

While all implementations MUST implement the subset of the Basic Profile necessary for the particular set of WAMP roles they provide, they MAY implement any subset of features from the Advanced Profile. Implementers SHOULD implement the maximum of features possible considering the aims of an implementation.

> Note: Features listed here may be experimental or underspecced and yet unimplemented in any implementation. This part of the specification is very much a work in progress. An approximate status of each feature is given at the beginning of the feature section.

9.1. Feature Announcement

Support for advanced features must be announced by the peers which implement them. The following is a complete list of advanced features currently defined or proposed.

Advanced RPC Features

Feature	Status	P	B	S	Cr	D	Ce
Progressive Call Results	beta				X	X	X
Progressive Calls	sketch				X	X	X
Call Timeout	alpha				X	X	X
Call Canceling	alpha				X	X	X
Caller Identification	alpha				X	X	X
Call Trustlevels	alpha					X	X
Registration Meta API	beta					X	
Pattern-based Registration	beta					X	X
Shared Registration	beta					X	X
Sharded Registration	alpha					X	X
Registration Revocation	alpha					X	X

Feature	Status	P	B	S	Cr	D	Ce
(Interface) Procedure Reflection	sketch					X	

Table 3

Advanced PubSub Features

Feature	Status	P	B	S	Cr	D	Ce
Subscriber Blackwhite Listing	stable	X	X				
Publisher Exclusion	stable	X	X				
Publisher Identification	alpha	X	X	X			
Publication Trustlevels	alpha		X	X			
Subscription Meta API	beta		X				
Pattern-based Subscription	beta		X	X			
Sharded Subscription	alpha		X	X			
Event History	alpha		X	X			
(Interface) Topic Reflection	sketch		X				

Table 4

Other Advanced Features

Feature	Status
Challenge-response Authentication	beta
Ticket authentication	beta
Cryptosign authentication	beta
RawSocket transport	stable
Batched WebSocket transport	sketch
HTTP Longpoll transport	beta
Session Meta API	beta
Call Rerouting	sketch

Table 5

The status of the respective AP feature is marked as follow:

Status	Description
sketch	There is a rough description of an itch to scratch, but the feature use case isn't clear, and there is no protocol proposal at all.
alpha	The feature use case is still fuzzy and/or the feature definition is unclear, but there is at least a protocol level proposal.
beta	The feature use case is clearly defined and the feature definition in the spec is sufficient to write a prototype implementation. The feature definition and details may still be incomplete and change.
stable	The feature definition in the spec is complete and stable and the feature use case is field proven in real applications. There are multiple, interoperable implementations.

Table 6

9.2. Additional Messages

The Advanced Profile defines additional WAMP-level messages which are explained in detail in separate sections. The following 4 additional message types MAY be used in the Advanced Profile and their direction between peer roles. Here, "Tx" ("Rx") means the message is sent (received) by a peer of the respective role.

Code	Message	Publisher	Broker	Subscriber	Caller	Dealer	Callee
4	`CHALLENGE`	Rx	Tx	Rx	Rx	Tx	Rx
5	`AUTHENTICATE`	Tx	Rx	Tx	Tx	Rx	Tx
49	`CANCEL`				Tx	Rx	
69	`INTERRUPT`					Tx	Rx

Table 7

9.2.1. CHALLENGE

The `CHALLENGE` message is used with certain Authentication Methods. During authenticated session establishment, a **Router** sends a challenge message.

```
[CHALLENGE, AuthMethod|string, Extra|dict]
```

9.2.2. AUTHENTICATE

The `AUTHENTICATE` message is used with certain Authentication Methods. A **Client** having received a challenge is expected to respond by sending a signature or token.

```
[AUTHENTICATE, Signature|string, Extra|dict]
```

9.2.3. CANCEL

The `CANCEL` message is used with the Call Canceling advanced feature. A *Caller* can cancel and issued call actively by sending a cancel message to the *Dealer*.

```
[CANCEL, CALL.Request|id, Options|dict]
```

9.2.4. INTERRUPT

The `INTERRUPT` message is used with the Call Canceling advanced feature. Upon receiving a cancel for a pending call, a *Dealer* will issue an interrupt to the *Callee*.

```
[INTERRUPT, INVOCATION.Request|id, Options|dict]
```

10. Meta API

10.1. Session Meta API

WAMP enables the monitoring of when sessions join a realm on the router or when they leave it via **Session Meta Events**. It also allows retrieving information about currently connected sessions via **Session Meta Procedures**.

Meta events are created by the router itself. This means that the events, as well as the data received when calling a meta procedure, can be accorded the same trust level as the router.

> Note that an implementation that only supports a *Broker* or *Dealer* role, not both at the same time, essentially cannot offer the **Session Meta API**, as it requires both roles to support this feature.

The following sections contain an informal, easy to digest description of the WAMP procedures and topics available in (this part of) the WAMP Meta API. A formal definition of the WAMP Meta API in terms of available WAMP procedures and topics including precise and complete type definitions of the application payloads, that is procedure arguments and results or event payloads is contained in

- Compiled Binary Schema: `<WAMP API Catalog>/schema/wamp-meta.bfbs`
- FlatBuffers Schema Source: `<WAMP API Catalog>/src/wamp-meta.fbs`

which uses FlatBuffers IDL to describe the API. The method of using FlatBuffers IDL and type definitions to formally define WAMP procedures and topics is detailed in section WAMP IDL.

Feature Announcement

Support for this feature MUST be announced by **both** *Dealers* and *Brokers* via:

```
HELLO.Details.roles.<role>.features.
    session_meta_api|bool := true
```

Here is a `WELCOME` message from a *Router* with support for both the *Broker* and *Dealer* role, and with support for **Session Meta API**:

```
[
    2,
    4580268554656113,
    {
        "authid":"OL3AeppwDLXiAAPbqm9IVhnw",
        "authrole": "anonymous",
        "authmethod": "anonymous",
        "roles": {
            "broker": {
                "features": {
                    "session_meta_api": true
                }
            },
            "dealer": {
                "features": {
                    "session_meta_api": true
                }
            }
        }
    }
]
```

> Note in particular that the feature is announced on both the *Broker* and the *Dealer* roles.

10.1.1. Events

A client can subscribe to the following session meta-events, which cover the lifecycle of a session:

- `wamp.session.on_join`: Fired when a session joins a realm on the router.
- `wamp.session.on_leave`: Fired when a session leaves a realm on the router or is disconnected.

Session Meta Events MUST be dispatched by the *Router* to the same realm as the WAMP session which triggered the event.

10.1.1.1. wamp.session.on_join

Fired when a session joins a realm on the router. The event payload consists of a single positional argument `details|dict`:

- `session|id` - The session ID of the session that joined

- `authid|string` - The authentication ID of the session that joined
- `authrole|string` - The authentication role of the session that joined
- `authmethod|string` - The authentication method that was used for authentication the session that joined
- `authprovider|string`- The provider that performed the authentication of the session that joined
- `transport|dict` - Optional, implementation defined information about the WAMP transport the joined session is running over.

> See **Authentication** for a description of the `authid`, `authrole`, `authmethod` and `authprovider` properties.

10.1.1.2. wamp.session.on_leave

Fired when a session leaves a realm on the router or is disconnected. The event payload consists of three positional arguments:

- `session|id` - The session ID of the session that left
- `authid`|string` - The authentication ID of the session that left
- `authrole|string` - The authentication role of the session that left

10.1.2. Procedures

A client can actively retrieve information about sessions, or forcefully close sessions, via the following meta-procedures:

- `wamp.session.count`: Obtains the number of sessions currently attached to the realm.
- `wamp.session.list`: Retrieves a list of the session IDs for all sessions currently attached to the realm.
- `wamp.session.get`: Retrieves information on a specific session.
- `wamp.session.kill`: Kill a single session identified by session ID.
- `wamp.session.kill_by_authid`: Kill all currently connected sessions that have the specified authid.
- `wamp.session.kill_by_authrole`: Kill all currently connected sessions that have the specified authrole.
- `wamp.session.kill_all`: Kill all currently connected sessions in the caller's realm.

Session meta procedures MUST be registered by the *Router* on the same realm as the WAMP session about which information is retrieved.

10.1.2.1. wamp.session.count

Obtains the number of sessions currently attached to the realm.

Positional arguments

1. `filter_authroles|list[string]` - Optional filter: if provided, only count sessions with an `authrole` from this list.

Positional results

1. `count|int` - The number of sessions currently attached to the realm.

10.1.2.2. wamp.session.list

Retrieves a list of the session IDs for all sessions currently attached to the realm.

Positional arguments

1. `filter_authroles|list[string]` - Optional filter: if provided, only count sessions with an `authrole` from this list.

Positional results

1. `session_ids|list` - List of WAMP session IDs (order undefined).

10.1.2.3. wamp.session.get

Retrieves information on a specific session.

Positional arguments

1. `session|id` - The session ID of the session to retrieve details for.

Positional results

1. `details|dict` - Information on a particular session:
   - `session|id` - The session ID of the session that joined
   - `authid|string` - The authentication ID of the session that joined
   - `authrole|string` - The authentication role of the session that joined
   - `authmethod|string` - The authentication method that was used for authentication the session that joined
   - `authprovider|string`- The provider that performed the authentication of the session that joined
   - `transport|dict` - Optional, implementation defined information about the WAMP transport the joined session is running over.

> See **Authentication** for a description of the `authid`, `authrole`, `authmethod` and `authprovider` properties.

Errors

- `wamp.error.no_such_session` - No session with the given ID exists on the router.

10.1.2.4. wamp.session.kill

Kill a single session identified by session ID.

The caller of this meta procedure may only specify session IDs other than its own session. Specifying the caller's own session will result in a `wamp.error.no_such_session` since no *other* session with that ID exists.

The keyword arguments are optional, and if not provided the reason defaults to `wamp.close.normal` and the message is omitted from the `GOODBYE` sent to the closed session.

Positional arguments

1. `session|id` - The session ID of the session to close.

Keyword arguments

1. `reason|uri` - reason for closing session, sent to client in `GOODBYE.Reason`.
2. `message|string` - additional information sent to client in `GOODBYE.Details` under the key "message".

Errors

- `wamp.error.no_such_session` - No session with the given ID exists on the router.
- `wamp.error.invalid_uri` - A `reason` keyword argument has a value that is not a valid non-empty URI.

10.1.2.5. wamp.session.kill_by_authid

Kill all currently connected sessions that have the specified `authid`.

If the caller's own session has the specified `authid`, the caller's session is excluded from the closed sessions.

The keyword arguments are optional, and if not provided the reason defaults to `wamp.close.normal` and the message is omitted from the `GOODBYE` sent to the closed session.

Positional arguments

1. `authid|string` - The authentication ID identifying sessions to close.

Keyword arguments

1. `reason|uri` - reason for closing sessions, sent to clients in `GOODBYE.Reason`
2. `message|string` - additional information sent to clients in `GOODBYE.Details` under the key "message".

Positional results

1. `sessions|list` - The list of WAMP session IDs of session that were killed.

Errors

- `wamp.error.invalid_uri` - A `reason` keyword argument has a value that is not a valid non-empty URI.

10.1.2.6. wamp.session.kill_by_authrole

Kill all currently connected sessions that have the specified `authrole`.

If the caller's own session has the specified `authrole`, the caller's session is excluded from the closed sessions.

The keyword arguments are optional, and if not provided the reason defaults to `wamp.close.normal` and the message is omitted from the `GOODBYE` sent to the closed session.

Positional arguments

1. `authrole|string` - The authentication role identifying sessions to close.

Keyword arguments

1. `reason|uri` - reason for closing sessions, sent to clients in `GOODBYE.Reason`
2. `message|string` - additional information sent to clients in `GOODBYE.Details` under the key "message".

Positional results

1. `count|int` - The number of sessions closed by this meta procedure.

Errors

- `wamp.error.invalid_uri` - A `reason` keyword argument has a value that is not a valid non-empty URI.

10.1.2.7. wamp.session.kill_all

Kill all currently connected sessions in the caller's realm.

The caller's own session is excluded from the closed sessions. Closing all sessions in the realm will not generate session meta events or testament events, since no subscribers would remain to receive these events.

The keyword arguments are optional, and if not provided the reason defaults to `wamp.close.normal` and the message is omitted from the `GOODBYE` sent to the closed session.

Keyword arguments

1. `reason|uri` - reason for closing sessions, sent to clients in `GOODBYE.Reason`
2. `message|string` - additional information sent to clients in `GOODBYE.Details` under the key "message".

Positional results

1. `count|int` - The number of sessions closed by this meta procedure.

Errors

- `wamp.error.invalid_uri` - A `reason` keyword argument has a value that is not a valid non-empty URI.

10.2. Registration Meta API

Registration Meta Events are fired when registrations are first created, when *Callees* are attached (removed) to (from) a registration, and when registrations are finally destroyed.

Furthermore, WAMP allows actively retrieving information about registrations via **Registration Meta Procedures**.

Meta-events are created by the router itself. This means that the events as well as the data received when calling a meta-procedure can be accorded the same trust level as the router.

> Note that an implementation that only supports a *Broker* or *Dealer* role, not both at the same time, essentially cannot offer the **Registration Meta API**, as it requires both roles to support this feature.

The following sections contain an informal, easy to digest description of the WAMP procedures and topics available in (this part of) the WAMP Meta API. A formal definition of the WAMP Meta API in terms of available WAMP procedures and topics including precise and complete type definitions of the application payloads, that is procedure arguments and results or event payloads is contained in:

- Compiled Binary Schema: `<WAMP API Catalog>/schema/wamp-meta.bfbs`
- FlatBuffers Schema Source: `<WAMP API Catalog>/src/wamp-meta.fbs`

which uses FlatBuffers IDL to describe the API. The method of using FlatBuffers IDL and type definitions to formally define WAMP procedures and topics is detailed in section WAMP IDL.

Feature Announcement

Support for this feature MUST be announced by a *Dealers* (`role := "dealer"`) via:

```
    HELLO.Details.roles.<role>.features.
        registration_meta_api|bool := true
```

Here is a `WELCOME` message from a *Router* with support for both the *Broker* and *Dealer* role, and with support for **Registration Meta API**:

```
    [
        2,
        4580268554656113,
        {
            "authid":"OL3AeppwDLXiAAPbqm9IVhnw",
            "authrole": "anonymous",
            "authmethod": "anonymous",
            "roles": {
                "broker": {
                    "features": {
                    }
                },
                "dealer": {
                    "features": {
                        "registration_meta_api": true
                    }
                }
            }
        }
    ]
```

10.2.1. Events

A client can subscribe to the following registration meta-events, which cover the lifecycle of a registration:

- `wamp.registration.on_create`: Fired when a registration is created through a registration request for a URI which was previously without a registration.
- `wamp.registration.on_register`: Fired when a *Callee* session is added to a registration.
- `wamp.registration.on_unregister`: Fired when a *Callee* session is removed from a registration.
- `wamp.registration.on_delete`: Fired when a registration is deleted after the last *Callee* session attached to it has been removed.

A `wamp.registration.on_register` event MUST be fired subsequent to a `wamp.registration.on_create` event, since the first registration results in both the creation of the registration and the addition of a session.

Similarly, the `wamp.registration.on_delete` event MUST be preceded by a `wamp.registration.on_unregister` event.

Registration Meta Events MUST be dispatched by the router to the same realm as the WAMP session which triggered the event.

10.2.1.1. wamp.registration.on_create

Fired when a registration is created through a registration request for a URI which was previously without a registration. The event payload consists of positional arguments:

- `session|id`: The session ID performing the registration request.
- `RegistrationDetails|dict`: Information on the created registration.

Object Schemas

```
RegistrationDetails :=
{
    "id": registration|id,
    "created": time_created|iso_8601_string,
    "uri": procedure|uri,
    "match": match_policy|string,
    "invoke": invocation_policy|string
}
```

See Pattern-based Registrations for a description of `match_policy`.

> NOTE: invocation_policy IS NOT YET DESCRIBED IN THE ADVANCED SPEC

10.2.1.2. wamp.registration.on_register

Fired when a session is added to a registration. The event payload consists of positional arguments:

- `session|id`: The ID of the session being added to a registration.
- `registration|id`: The ID of the registration to which a session is being added.

10.2.1.3. wamp.registration.on_unregister

Fired when a session is removed from a subscription. The event payload consists of positional arguments:

- `session|id`: The ID of the session being removed from a registration.
- `registration|id`: The ID of the registration from which a session is being removed.

10.2.1.4. wamp.registration.on_delete

Fired when a registration is deleted after the last session attached to it has been removed. The event payload consists of positional arguments:

- `session|id`: The ID of the last session being removed from a registration.
- `registration|id`: The ID of the registration being deleted.

10.2.2. Procedures

A client can actively retrieve information about registrations via the following meta-procedures:

- `wamp.registration.list`: Retrieves registration IDs listed according to match policies.
- `wamp.registration.lookup`: Obtains the registration (if any) managing a procedure, according to some match policy.
- `wamp.registration.match`: Obtains the registration best matching a given procedure URI.
- `wamp.registration.get`: Retrieves information on a particular registration.
- `wamp.registration.list_callees`: Retrieves a list of session IDs for sessions currently attached to the registration.
- `wamp.registration.count_callees`: Obtains the number of sessions currently attached to the registration.

10.2.2.1. wamp.registration.list

Retrieves registration IDs listed according to match policies.

Arguments

- None

Results

- `RegistrationLists|dict`: A dictionary with a list of registration IDs for each match policy.

Object Schemas

```
RegistrationLists :=
{
    "exact": registration_ids|list,
    "prefix": registration_ids|list,
    "wildcard": registration_ids|list
}
```

See Pattern-based Registrations for a description of match policies.

10.2.2.2. wamp.registration.lookup

Obtains the registration (if any) managing a procedure, according to some match policy.

Arguments

- `procedure|uri`: The procedure to lookup the registration for.
- (Optional) `options|dict`: Same options as when registering a procedure.

Results

- (Nullable) `registration|id`: The ID of the registration managing the procedure, if found, or null.

10.2.2.3. wamp.registration.match

Obtains the registration best matching a given procedure URI.

Arguments

- `procedure|uri`: The procedure URI to match

Results

- (Nullable) `registration|id`: The ID of best matching registration, or null.

10.2.2.4. wamp.registration.get

Retrieves information on a particular registration.

Arguments

- `registration|id`: The ID of the registration to retrieve.

Results

- `RegistrationDetails|dict`: Details on the registration.

Error URIs

- `wamp.error.no_such_registration`: No registration with the given ID exists on the router.

Object Schemas

```
    RegistrationDetails :=
    {
        "id": registration|id,
        "created": time_created|iso_8601_string,
        "uri": procedure|uri,
        "match": match_policy|string,
        "invoke": invocation_policy|string
    }
```

See Pattern-based Registrations for a description of match policies.

NOTE: invocation_policy IS NOT YET DESCRIBED IN THE ADVANCED SPEC

10.2.2.5. wamp.registration.list_callees

Retrieves a list of session IDs for sessions currently attached to the registration.

Arguments

- `registration|id`: The ID of the registration to get callees for.

Results

- `callee_ids|list`: A list of WAMP session IDs of callees currently attached to the registration.

Error URIs

- `wamp.error.no_such_registration`: No registration with the given ID exists on the router.

10.2.2.6. wamp.registration.count_callees

Obtains the number of sessions currently attached to a registration.

Arguments

- `registration|id`: The ID of the registration to get the number of callees for.

Results

- `count|int`: The number of callees currently attached to a registration.

Error URIs

- `wamp.error.no_such_registration`: No registration with the given ID exists on the router.

10.3. Subscriptions Meta API

Within an application, it may be desirable for a publisher to know whether a publication to a specific topic currently makes sense, i.e. whether there are any subscribers who would receive an event based on the publication. It may also be desirable to keep a current count of subscribers to a topic to then be able to filter out any subscribers who are not supposed to receive an event.

Subscription *meta-events* are fired when topics are first created, when clients subscribe/unsubscribe to them, and when topics are deleted. WAMP allows retrieving information about subscriptions via subscription *meta-procedures*.

Support for this feature MUST be announced by Brokers via

```
    HELLO.Details.roles.broker.features.subscription_meta_api|
        bool := true
```

Meta-events are created by the router itself. This means that the events as well as the data received when calling a meta-procedure can be accorded the same trust level as the router.

The following sections contain an informal, easy to digest description of the WAMP procedures and topics available in (this part of) the WAMP Meta API. A formal definition of the WAMP Meta API in terms of available WAMP procedures and topics including precise and complete type definitions of the application payloads, that is procedure arguments and results or event payloads is contained in

- Compiled Binary Schema: `<WAMP API Catalog>/schema/wamp-meta.bfbs`
- FlatBuffers Schema Source: `<WAMP API Catalog>/src/wamp-meta.fbs`

which uses FlatBuffers IDL to describe the API. The method of using FlatBuffers IDL and type definitions to formally define WAMP procedures and topics is detailed in section WAMP IDL.

Feature Announcement

Support for this feature MUST be announced by a *Brokers* (`role := "nroker"`) via:

```
HELLO.Details.roles.<role>.features.
    subscription_meta_api|bool := true
```

Here is a `WELCOME` message from a *Router* with support for both the *Broker* and *Dealer* role, and with support for **Subscription Meta API**:

```
[
    2,
    4580268554656113,
    {
        "authid":"OL3AeppwDLXiAAPbqm9IVhnw",
        "authrole": "anonymous",
        "authmethod": "anonymous",
        "roles": {
            "broker": {
                "features": {
                    "subscription_meta_api": true
                }
            },
            "dealer": {
                "features": {
                }
            }
        }
    }
]
```

10.3.1. Events

A client can subscribe to the following session meta-events, which cover the lifecycle of a subscription:

- `wamp.subscription.on_create`: Fired when a subscription is created through a subscription request for a topic which was previously without subscribers.
- `wamp.subscription.on_subscribe`: Fired when a session is added to a subscription.

- `wamp.subscription.on_unsubscribe`: Fired when a session is removed from a subscription.
- `wamp.subscription.on_delete`: Fired when a subscription is deleted after the last session attached to it has been removed.

A `wamp.subscription.on_subscribe` event MUST always be fired subsequent to a `wamp.subscription.on_create` event, since the first subscribe results in both the creation of the subscription and the addition of a session. Similarly, the `wamp.subscription.on_delete` event MUST always be preceded by a `wamp.subscription.on_unsubscribe` event.

The WAMP subscription meta events shall be dispatched by the router to the same realm as the WAMP session which triggered the event.

10.3.1.1. wamp.subscription.on_create

Fired when a subscription is created through a subscription request for a topic which was previously without subscribers. The event payload consists of positional arguments:

- `session|id`: ID of the session performing the subscription request.
- `SubscriptionDetails|dict`: Information on the created subscription.

Object Schemas

```
SubscriptionDetails :=
{
    "id": subscription|id,
    "created": time_created|iso_8601_string,
    "uri": topic|uri,
    "match": match_policy|string
}
```

See Pattern-based Subscriptions for a description of `match_policy`.

10.3.1.2. wamp.subscription.on_subscribe

Fired when a session is added to a subscription. The event payload consists of positional arguments:

- `session|id`: ID of the session being added to a subscription.
- `subscription|id`: ID of the subscription to which the session is being added.

10.3.1.3. wamp.subscription.on_unsubscribe

Fired when a session is removed from a subscription. The event payload consists of positional arguments:

- `session|id`: ID of the session being removed from a subscription.
- `subscription|id`: ID of the subscription from which the session is being removed.

10.3.1.4. wamp.subscription.on_delete

Fired when a subscription is deleted after the last session attached to it has been removed. The event payload consists of positional arguments:

- `session|id`: ID of the last session being removed from a subscription.
- `subscription|id`: ID of the subscription being deleted.

10.3.2. Procedures

A client can actively retrieve information about subscriptions via the following meta-procedures:

- `wamp.subscription.list`: Retrieves subscription IDs listed according to match policies.
- `wamp.subscription.lookup`: Obtains the subscription (if any) managing a topic, according to some match policy.
- `wamp.subscription.match`: Retrieves a list of IDs of subscriptions matching a topic URI, irrespective of match policy.
- `wamp.subscription.get`: Retrieves information on a particular subscription.
- `wamp.subscription.list_subscribers`: Retrieves a list of session IDs for sessions currently attached to the subscription.
- `wamp.subscription.count_subscribers`: Obtains the number of sessions currently attached to the subscription.

10.3.2.1. wamp.subscription.list

Retrieves subscription IDs listed according to match policies.

Arguments - None

Results

The result consists of one positional argument:

- `SubscriptionLists|dict`: A dictionary with a list of subscription IDs for each match policy.

Object Schemas

```
SubscriptionLists :=
{
    "exact": subscription_ids|list,
    "prefix": subscription_ids|list,
    "wildcard": subscription_ids|list
}
```

See Pattern-based Subscriptions for information on match policies.

10.3.2.2. wamp.subscription.lookup

Obtains the subscription (if any) managing a topic, according to some match policy.

Arguments

- `topic|uri`: The URI of the topic.
- (Optional) `options|dict`: Same options as when subscribing to a topic.

Results

The result consists of one positional argument:

- (Nullable) `subscription|id`: The ID of the subscription managing the topic, if found, or null.

10.3.2.3. wamp.subscription.match

Retrieves a list of IDs of subscriptions matching a topic URI, irrespective of match policy.

Arguments

- `topic|uri`: The topic to match.

Results

The result consists of positional arguments:

- (Nullable) `subscription_ids|list`: A list of all matching subscription IDs, or null.

10.3.2.4. wamp.subscription.get

Retrieves information on a particular subscription.

Arguments

- `subscription|id`: The ID of the subscription to retrieve.

Results

The result consists of one positional argument:

- `SubscriptionDetails|dict`: Details on the subscription.

Error URIs

- `wamp.error.no_such_subscription`: No subscription with the given ID exists on the router.

Object Schemas

```
SubscriptionDetails :=
{
    "id": subscription|id,
    "created": time_created|iso_8601_string,
    "uri": topic|uri,
    "match": match_policy|string
}
```

See Pattern-based Subscriptions for information on match policies.

10.3.2.5. wamp.subscription.list_subscribers

Retrieves a list of session IDs for sessions currently attached to the subscription.

Arguments - `subscription|id`: The ID of the subscription to get subscribers for.

Results

The result consists of positional arguments:

- `subscribers_ids|list`: A list of WAMP session IDs of subscribers currently attached to the subscription.

Error URIs

- `wamp.error.no_such_subscription`: No subscription with the given ID exists on the router.

10.3.2.6. wamp.subscription.count_subscribers

Obtains the number of sessions currently attached to a subscription.

Arguments

- `subscription|id`: The ID of the subscription to get the number of subscribers for.

Results

The result consists of one positional argument:

- `count|int`: The number of sessions currently attached to a subscription.

Error URIs

- `wamp.error.no_such_subscription`: No subscription with the given ID exists on the router.

11. Advanced RPC

11.1. Progressive Call Results

A procedure implemented by a *Callee* and registered at a *Dealer* may produce progressive results. Progressive results can e.g. be used to return partial results for long-running operations, or to chunk the transmission of larger results sets.

Feature Announcement

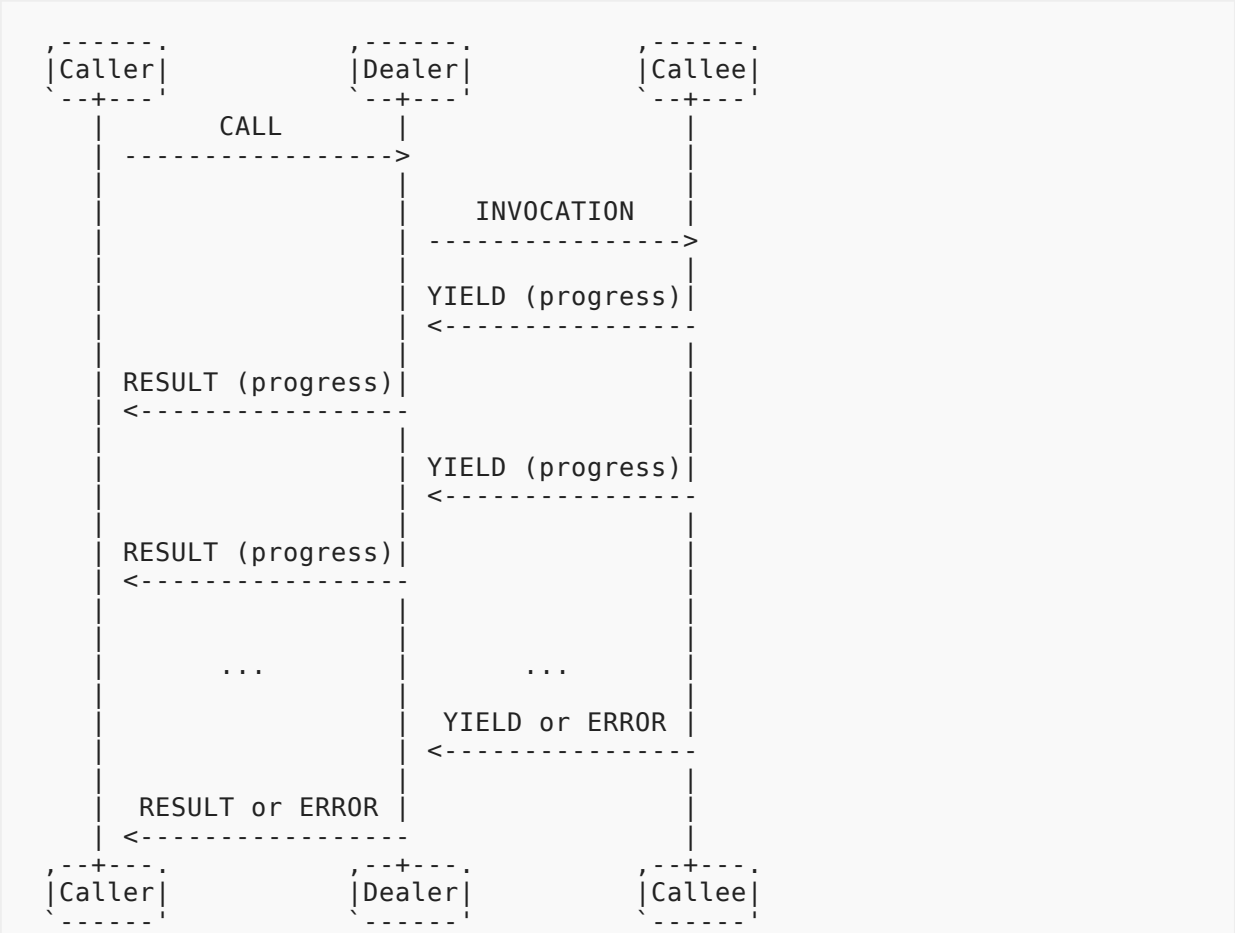
Support for this advanced feature MUST be announced by *Callers* (`role := "caller"`), *Callees* (`role := "callee"`) and *Dealers* (`role := "dealer"`) via

```
    HELLO.Details.roles.<role>.features.
         progressive_call_results|bool := true
```

Additionally, *Callees* and *Dealers* MUST support Call Canceling, which is required for canceling progressive results if the original *Caller* leaves the realm. If a *Callee* supports Progressive Call Results, but not Call Canceling, then the *Dealer* disregards the *Callees* Progressive Call Results feature.

Message Flow

The message flow for progressive results involves:



A *Caller* indicates its willingness to receive progressive results by setting

```
CALL.Options.receive_progress|bool := true
```

Example. Caller-to-Dealer `CALL`

```
[
    48,
    77133,
    {
        "receive_progress": true
    },
    "com.myapp.compute_revenue",
    [2010, 2011, 2012]
]
```

If the *Callee* supports progressive calls, the *Dealer* will forward the *Caller's* willingness to receive progressive results by setting

```
INVOCATION.Details.receive_progress|bool := true
```

Example. Dealer-to-Callee `INVOCATION`

```
[
    68,
    87683,
    324,
    {
        "receive_progress": true
    },
    [2010, 2011, 2012]
]
```

An endpoint implementing the procedure produces progressive results by sending `YIELD` messages to the *Dealer* with

```
YIELD.Options.progress|bool := true
```

Example. Callee-to-Dealer progressive `YIELD`s

```
[
    70,
    87683,
    {
        "progress": true
    },
    ["Y2010", 120]
]
```

```
[
    70,
    87683,
    {
        "progress": true
    },
    ["Y2011", 205]
]
```

Upon receiving an `YIELD` message from a *Callee* with `YIELD.Options.progress == true` (for a call that is still ongoing), the *Dealer* will **immediately** send a `RESULT` message to the original *Caller* with

```
RESULT.Details.progress|bool := true
```

Example. Dealer-to-Caller progressive `RESULT`s

```
[
    50,
    77133,
    {
        "progress": true
    },
    ["Y2010", 120]
]
```

```
[
    50,
    77133,
    {
        "progress": true
    },
    ["Y2011", 205]
]
```

and so on...

An invocation MUST *always* end in either a *normal* `RESULT` or `ERROR` message being sent by the *Callee* and received by the *Dealer*.

Example. Callee-to-Dealer final `YIELD`

```
[
    70,
    87683,
    {},
    ["Total", 490]
]
```

Example. Callee-to-Dealer final `ERROR`

```
[
    4,
    87683,
    {},
    "com.myapp.invalid_revenue_year",
    [1830]
]
```

A call MUST *always* end in either a *normal* `RESULT` or `ERROR` message being sent by the *Dealer* and received by the *Caller*.

Example. Dealer-to-Caller final `RESULT`

```
[
    50,
    77133,
    {},
    ["Total", 490]
]
```

Example. Dealer-to-Caller final `ERROR`

```
[
    4,
    77133,
    {},
    "com.myapp.invalid_revenue_year",
    [1830]
]
```

In other words: `YIELD` with `YIELD.Options.progress == true` and `RESULT` with `RESULT.Details.progress == true` messages may only be sent *during* a call or invocation is still ongoing.

The final `YIELD` and final `RESULT` may also be empty, e.g. when all actual results have already been transmitted in progressive result messages.

Example. Callee-to-Dealer `YIELD`s

```
[70, 87683, {"progress": true}, ["Y2010", 120]]
[70, 87683, {"progress": true}, ["Y2011", 205]]
 ...
[70, 87683, {"progress": true}, ["Total", 490]]
[70, 87683, {}]
```

Example. Dealer-to-Caller `RESULT`s

```
[50, 77133, {"progress": true}, ["Y2010", 120]]
[50, 77133, {"progress": true}, ["Y2011", 205]]
 ...
[50, 77133, {"progress": true}, ["Total", 490]]
[50, 77133, {}]
```

The progressive `YIELD` and progressive `RESULT` may also be empty, e.g. when those messages are only used to signal that the procedure is still running and working, and the actual result is completely delivered in the final `YIELD` and `RESULT`:

Example. Callee-to-Dealer `YIELD`s

```
[70, 87683, {"progress": true}]
[70, 87683, {"progress": true}]
...
[70, 87683, {}, [["Y2010", 120], ["Y2011", 205], ...,
    ["Total", 490]]]
```

Example. Dealer-to-Caller `RESULT`s

```
[50, 77133, {"progress": true}]
[50, 77133, {"progress": true}]
...
[50, 77133, {}, [["Y2010", 120], ["Y2011", 205], ...,
    ["Total", 490]]]
```

> Note that intermediate, progressive results and/or the final result MAY have different structure. The WAMP peer implementation is responsible for mapping everything into a form suitable for consumption in the host language.

Example. Callee-to-Dealer `YIELD`s

```
[70, 87683, {"progress": true}, ["partial 1", 10]]
[70, 87683, {"progress": true}, [], {"foo": 10,
    "bar": "partial 1"}]
 ...
[70, 87683, {}, [1, 2, 3], {"moo": "hello"}]
```

Example. Dealer-to-Caller `RESULT`s

```
[50, 77133, {"progress": true}, ["partial 1", 10]]
[50, 77133, {"progress": true}, [], {"foo": 10,
    "bar": "partial 1"}]
 ...
[50, 77133, {}, [1, 2, 3], {"moo": "hello"}]
```

Even if a *Caller* has indicated its expectation to receive progressive results by setting `CALL.Options.receive_progress|bool := true`, a *Callee* is **not required** to produce progressive results. `CALL.Options.receive_progress` and `INVOCATION.Details.receive_progress` are simply indications that the *Caller* is prepared to process progressive results, should there be any produced. In other words, *Callees* are free to ignore such `receive_progress` hints at any time.

Progressive Call Result Cancellation

Upon receiving a `YIELD` message from a *Callee* with `YIELD.Options.progress == true` (for a call that is still ongoing), if the original *Caller* is no longer available (has left the realm), then the *Dealer* will send an `INTERRUPT` to the *Callee*. The `INTERRUPT` will have `Options.mode` set to `"killnowait"` to indicate to the client that no response should be sent to the `INTERRUPT`. This `INTERRUPT` in only sent in response to a progressive `YIELD` (`Details.progress == true`), and is not sent in response to a normal or final `YIELD`.

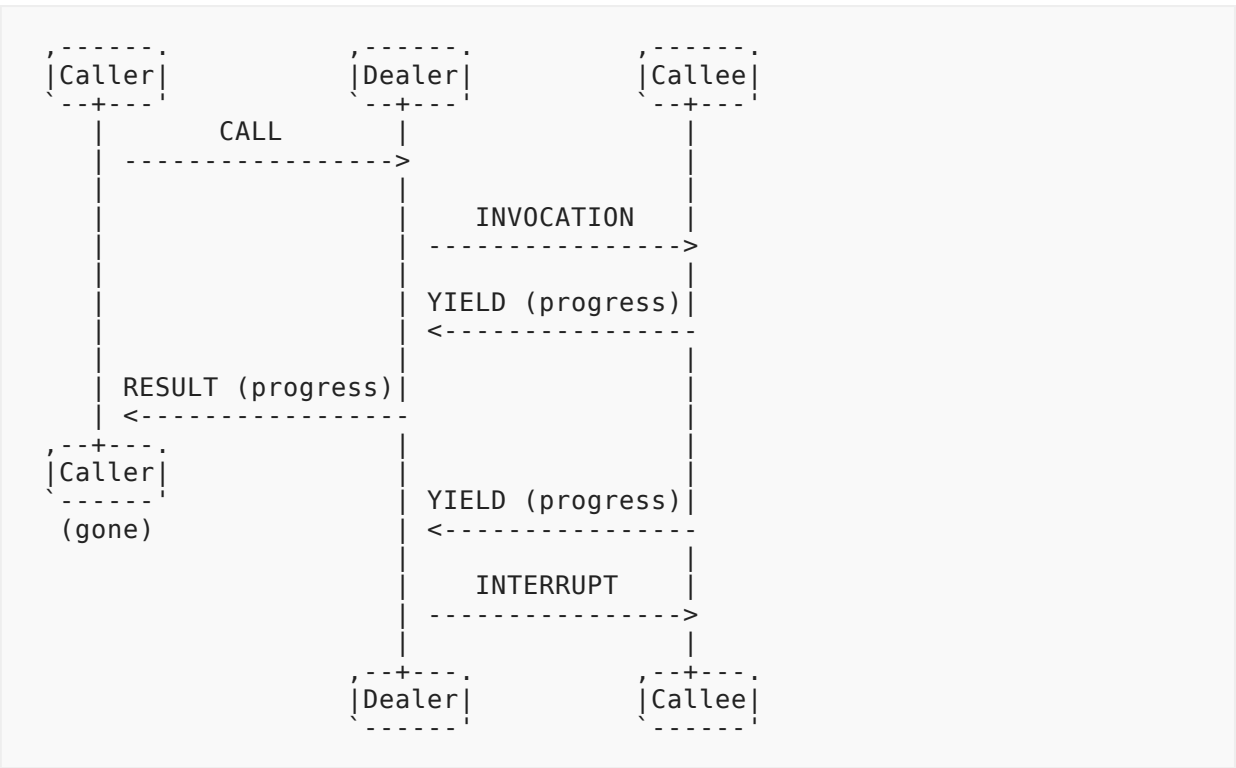
```
[INTERRUPT, INVOCATION.Request|id, Options|dict]
```

Options:

```
INTERRUPT.Options.mode|string == "killnowait"
```

Progressive call result cancellation closes an important safety gap: In cases where progressive results are used to stream data to *Callers*, and network connectivity is unreliable, *Callers* my often get disconnected in the middle of receiving progressive results. Recurring connect, call, disconnect cycles can quickly build up *Callees* streaming results to dead *Callers*. This can overload the router and further degrade network connectivity.

The message flow for progressive results cancellation involves:



Note: Any `ERROR` returned by the *Callee*, in response to the `INTERRUPT`, is ignored (same as in call canceling when mode="killnowait"). So, it is not necessary for the *Callee* to send an `ERROR` message.

Ignoring Progressive Call Requests

A *Callee* that does not support progressive results SHOULD ignore any `INVOCATION.Details.receive_progress` flag.

A *Callee* that supports progressive results, but does not support call canceling is considered by the *Dealer* to not support progressive results.

11.2. Progressive Calls

A procedure implemented by a *Callee* and registered at a *Dealer* may receive a progressive call. Progressive results can e.g. be used to start processing initial data where a larger data set may not yet have been generated or received by the *Caller*.

See this GitHub issue for more discussion: https://github.com/wamp-proto/wamp-proto/issues/167

11.3. Call Timeouts

A *Caller* might want to issue a call and provide a *timeout* after which the call will finish.

A *timeout* allows for **automatic** cancellation of a call after a specified time either at the *Callee* or at the *Dealer*.

A *Caller* specifies a timeout by providing

```
    CALL.Options.timeout|integer
```

in ms. A timeout value of `0` deactivates automatic call timeout. This is also the default value.

The timeout option is a companion to, but slightly different from the `CANCEL` and `INTERRUPT` messages that allow a *Caller* and *Dealer* to **actively** cancel a call or invocation.

In fact, a timeout timer might run at three places:

- *Caller*
- *Dealer*
- *Callee*

Feature Announcement

Support for this feature MUST be announced by *Callers* (`role := "caller"`), *Callees* (`role := "callee"`) and *Dealers* (`role := "dealer"`) via

```
    HELLO.Details.roles.<role>.features.call_timeout|bool := true
```

11.4. Call Canceling

A *Caller* might want to actively cancel a call that was issued, but not has yet returned. An example where this is useful could be a user triggering a long running operation and later changing his mind or no longer willing to wait.

Feature Announcement

Support for this feature MUST be announced by *Callers* (`role := "caller"`), *Callees* (`role := "callee"`) and *Dealers* (`role := "dealer"`) via

```
    HELLO.Details.roles.<role>.features.call_canceling|bool := true
```

Message Flow

The message flow between *Callers*, a *Dealer* and *Callees* for canceling remote procedure calls involves the following messages:

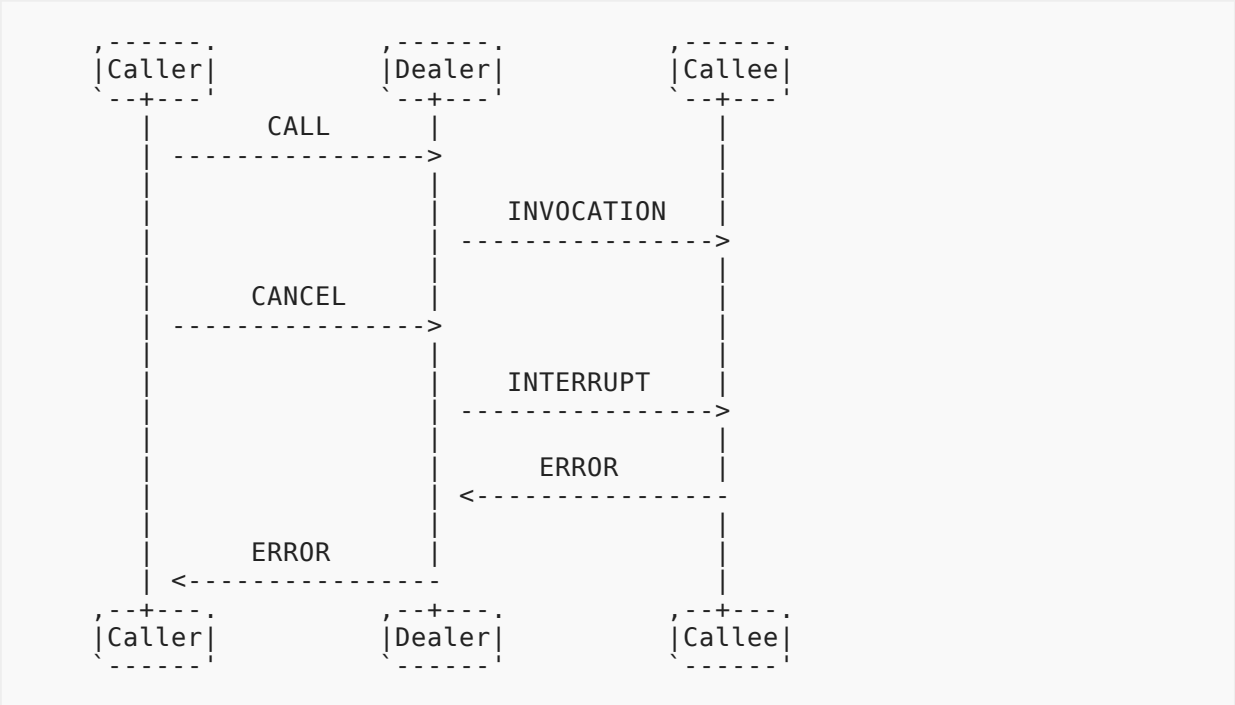
- `CANCEL`
- `INTERRUPT`
- `ERROR`

A call may be canceled at the *Callee* or (U+00A0)at (U+00A0)the (U+00A0)*Dealer* side. Cancellation behaves differently depending on the mode:

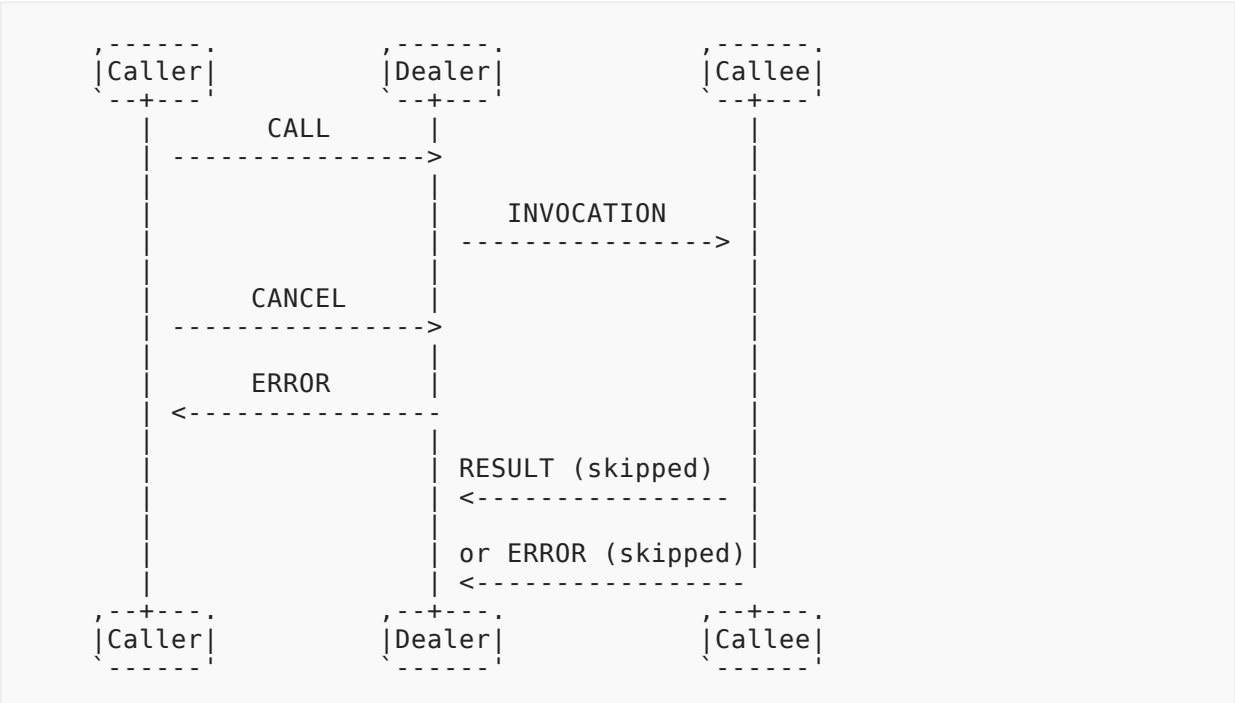
- **skip**: The pending call is canceled and `ERROR` is sent immediately back to the caller. No `INTERRUPT` is sent to the callee and the result is discarded when received.
- **kill**: `INTERRUPT` is sent to the callee, but `ERROR` is not returned to the caller until after the callee has responded to the canceled call. In this case the caller may receive `RESULT` or `ERROR` depending whether the callee finishes processing the invocation or the interrupt first.
- **killnowait**: The pending call is canceled and `ERROR` is sent immediately back to the caller. `INTERRUPT` is sent to the callee and any response to the invocation or interrupt from the callee is discarded when received.

If the callee does not support call canceling, then behavior is **skip**.

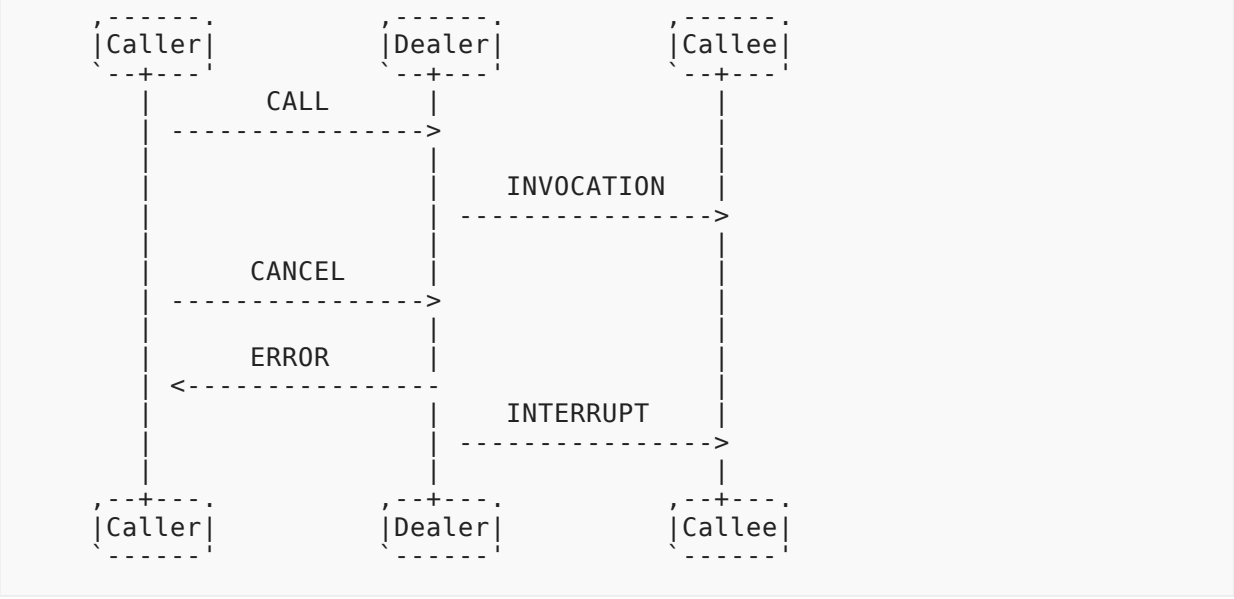
Message flow during call canceling when (U+00A0)*Callee* supports this feature and (U+00A0)mode is `kill`



Message flow during call canceling when (U+00A0)*Callee* does not support this feature or (U+00A0)mode is `skip`



Message flow during call canceling when (U+00A0)*Callee* supports this feature and (U+00A0)mode is `killnowait`



A *Caller* cancels a remote procedure call initiated (but not yet finished) by sending a `CANCEL` message to the *Dealer*:

```
[CANCEL, CALL.Request|id, Options|dict]
```

A *Dealer* cancels an invocation of an endpoint initiated (but not yet finished) by sending a `INTERRUPT` message to the *Callee*:

```
[INTERRUPT, INVOCATION.Request|id, Options|dict]
```

Options:

```
CANCEL.Options.mode|string == "skip" | "kill" | "killnowait"
```

Ignoring Results after Cancel

After the *Dealer* sends an `INTERRUPT` when mode="killnowait", any responses from the *Callee* are ignored. This means that it is not necessary for the *Callee* to respond with an `ERROR` message, when mode="killnowait", since the *Dealer* ignores it.

11.5. Call Re-Routing

A *CALLEE* might not be able to attend to a call. This may be due to a multitude of reasons including, but not limited to:

- *CALLEE* is busy handling other requests and is not able to attend
- *CALLEE* has dependency issues which prevent it from being able to fulfil the request

- In a HA environment, the *Callee* knows that it is scheduled to be taken off the HA cluster and as such should not handle the request.

A *unavailable* response allows for **automatic** reroute of a call by the *Dealer* without the *CALLER* ever having to know about it.

When such a situation occurs, the *Callee* responds to a `INVOCATION` message with the error uri:

wamp.error.unavailable

When the *Dealer* receives the `wamp.error.unavailable` message in response to an `INVOCATION`, it will reroute the `CALL` to another *registration* according to the rerouting rules of the `invocation_policy` of the `procedure`, as given below.

Feature Announcement

Support for this feature MUST be announced by *Callees* (`role := "callee"`) and *Dealers* (`role := "dealer"`) via

HELLO.Details.roles.<role>.features.call_reroute|bool := true

Rerouting Rules

The *Dealer* MUST adhere to the `invocation policy` of the `procedure` when rerouting the `CALL`, while assuming that the `unavailable` registration virtually does not exist.

For different `invocation policy` the *Dealer* MUST follow:

Invocation Policy	Operation
`single`	Responds with a `wamp.error.no_available_callee` error message to the *CALLER*
`roundrobin`	Picks the next `registration` from the *Registration Queue* of the *Procedure*
`random`	Picks another `registration` at random from the Registration Queue of the *Procedure*, as long as it is not the same `registration`
`first`	Picks the `registration` which was registered after the *called* registration was registered
`last`	Picks the `registration` which was registered right before the *called* registration was registered

Table 8

Failure Scenario

In case all available registrations of a *Procedure* responds with a `wamp.error.unavailable` for a *CALL*, the *Dealer* MUST respond with a `wamp.error.no_available_callee` to the *CALLER*

11.6. Caller Identification

A *Caller* MAY **request** the disclosure of its identity (its WAMP session ID) to endpoints of a routed call via

```
CALL.Options.disclose_me|bool := true
```

Example

```
[48, 7814135, {"disclose_me": true}, "com.myapp.echo",
    ["Hello, world!"]]
```

If above call is issued by a *Caller* with WAMP session ID `3335656`, the *Dealer* sends an `INVOCATION` message to *Callee* with the *Caller's* WAMP session ID in `INVOCATION.Details.caller`:

Example

```
[68, 6131533, 9823526, {"caller": 3335656}, ["Hello, world!"]]
```

Note that a *Dealer* MAY disclose the identity of a *Caller* even without the *Caller* having explicitly requested to do so when the *Dealer* configuration (for the called procedure) is setup to do so.

Feature Announcement

Support for this feature MUST be announced by *Callers* (`role := "caller"`), *Callees* (`role := "callee"`) and *Dealers* (`role := "dealer"`) via

```
HELLO.Details.roles.<role>.features.
     caller_identification|bool := true
```

Request Identification

A *Dealer* MAY deny a *Caller's* request to disclose its identity:

Example

```
[8, 7814135, "wamp.error.disclose_me.not_allowed"]
```

A *Callee* MAY **request** the disclosure of caller identity via

```
REGISTER.Options.disclose_caller|bool := true
```

Example

```
[64, 927639114088448, {"disclose_caller":true},
    "com.maypp.add2"]
```

With the above registration, the registered procedure is called with the caller's sessionID as part of the call details object.

11.7. Call Trust Levels

A *Dealer* may be configured to automatically assign *trust levels* to calls issued by *Callers* according to the *Dealer* configuration on a per-procedure basis and/or depending on the application defined role of the (authenticated) *Caller*.

A *Dealer* supporting trust level will provide

```
INVOCATION.Details.trustlevel|integer
```

in an `INVOCATION` message sent to a *Callee*. The trustlevel `0` means lowest trust, and higher integers represent (application-defined) higher levels of trust.

Example

```
[68, 6131533, 9823526, {"trustlevel": 2}, ["Hello, world!"]]
```

In above event, the *Dealer* has (by configuration and/or other information) deemed the call (and hence the invocation) to be of trustlevel 2.

Feature Announcement

Support for this feature MUST be announced by *Callees* (`role := "callee"`) and *Dealers* (`role := "dealer"`) via

```
HELLO.Details.roles.<role>.features.call_trustlevels|bool := true
```

11.8. Pattern-based Registrations

By default, *Callees* register procedures with **exact matching policy**. That is a call will only be routed to a *Callee* by the *Dealer* if the procedure called (`CALL.Procedure`) *exactly* matches the endpoint registered (`REGISTER.Procedure`).

A *Callee* might want to register procedures based on a *pattern*. This can be useful to reduce the number of individual registrations to be set up or to subscribe to a open set of topics, not known beforehand by the *Subscriber*.

If the *Dealer* and the *Callee* support **pattern-based registrations**, this matching can happen by

- **prefix-matching policy**
- **wildcard-matching policy**

Feature Announcement

Support for this feature MUST be announced by *Callees* (`role := "callee"`) and *Dealers* (`role := "dealer"`) via

```
HELLO.Details.roles.<role>.features.
    pattern_based_registration|bool := true
```

11.8.1. Prefix Matching

A *Callee* requests **prefix-matching policy** with a registration request by setting

```
REGISTER.Options.match|string := "prefix"
```

Example

```
[
    64,
    612352435,
    {
        "match": "prefix"
    },
    "com.myapp.myobject1"
]
```

When a **prefix-matching policy** is in place, any call with a procedure that has `REGISTER.Procedure` as a *prefix* will match the registration, and potentially be routed to *Callees* on that registration.

In above example, the following calls with `CALL.Procedure`

- `com.myapp.myobject1.myprocedure1`
- `com.myapp.myobject1-mysubobject1`
- `com.myapp.myobject1.mysubobject1.myprocedure1`
- `com.myapp.myobject1`

will all apply for call routing. A call with one of the following `CALL.Procedure`

- `com.myapp.myobject2`
- `com.myapp.myobject`

will not apply.

11.8.2. Wildcard Matching

A *Callee* requests **wildcard-matching policy** with a registration request by setting

```
REGISTER.Options.match|string := "wildcard"
```

Wildcard-matching allows to provide wildcards for **whole** URI components.

Example

```
[
    64,
    612352435,
    {
        "match": "wildcard"
    },
    "com.myapp..myprocedure1"
]
```

In the above registration request, the 3rd URI component is empty, which signals a wildcard in that URI component position. In this example, calls with `CALL.Procedure` e.g.

- `com.myapp.myobject1.myprocedure1`
- `com.myapp.myobject2.myprocedure1`

will all apply for call routing. Calls with `CALL.Procedure` e.g.

- `com.myapp.myobject1.myprocedure1.mysubprocedure1`
- `com.myapp.myobject1.myprocedure2`
- `com.myapp2.myobject1.myprocedure1`

will not apply for call routing.

When a single call matches more than one of a *Callees* registrations, the call MAY be routed for invocation on multiple registrations, depending on call settings.

11.8.3. Design Aspects

No set semantics

Since each *Callee*'s' registrations "stands on its own", there is no *set semantics* implied by pattern-based registrations.

E.g. a *Callee* cannot register to a broad pattern, and then unregister from a subset of that broad pattern to form a more complex registration. Each registration is separate.

Calls matching multiple registrations

There can be situations, when (U+00A0)some call URI matches more then one registration. In (U+00A0)this case a call is routed to one and (U+00A0)only one best matched RPC registration, or (U+00A0)fails with ERROR `wamp.error.no_such_procedure`.

The following algorithm MUST be applied to (U+00A0)find a single RPC registration to (U+00A0)which a call is routed:

1. Check for (U+00A0)exact matching registration. If this match exists — (U+00A0 U+2014) use it.

2. If there are prefix-based registrations, (U+00A0)find the registration with the longest prefix match. Longest means it has more URI components matched, e.g. for (U+00A0)call URI `a1.b2.c3.d4` registration `a1.b2.c3` has higher priority than registration `a1.b2`. If this match exists — (U+00A0 U+2014) use it.
3. If there are wildcard-based registrations, find the registration with the longest portion of (U+00A0)URI components matched before each wildcard. E.g. for (U+00A0)call URI `a1.b2.c3.d4` registration `a1.b2..d4` has higher priority than registration `a1...d4`, see below for more complex examples. If this match exists — (U+00A0 U+2014) use it.
4. If there is no exact match, no prefix match, and no wildcard match, then *Dealer* MUST return ERROR `wamp.error.no_such_procedure`.

Examples

```
Registered RPCs:
    1. 'a1.b2.c3.d4.e55' (exact),
    2. 'a1.b2.c3' (prefix),
    3. 'a1.b2.c3.d4' (prefix),
    4. 'a1.b2..d4.e5',
    5. 'a1.b2.c33..e5',
    6. 'a1.b2..d4.e5..g7',
    7. 'a1.b2..d4..f6.g7'

Call request RPC URI: 'a1.b2.c3.d4.e55' →
    exact match. Use RPC 1
Call request RPC URI: 'a1.b2.c3.d98.e74' →
    no exact match, single prefix match. Use RPC 2
Call request RPC URI: 'a1.b2.c3.d4.e325' →
    no exact match, 2 prefix matches (2,3), select longest one.
    Use RPC 3
Call request RPC URI: 'a1.b2.c55.d4.e5' →
    no exact match, no prefix match, single wildcard match.
    Use RPC 4
Call request RPC URI: 'a1.b2.c33.d4.e5' →
    no exact match, no prefix match, 2 wildcard matches (4,5),
    select longest one. Use RPC 5
Call request RPC URI: 'a1.b2.c88.d4.e5.f6.g7' →
    no exact match, no prefix match, 2 wildcard matches (6,7),
    both having equal first portions (a1.b2), but RPC 6 has longer
    second portion (d4.e5). Use RPC 6
Call request RPC URI: 'a2.b2.c2.d2.e2' →
    no exact match, no prefix match, no wildcard match.
    Return wamp.error.no_such_procedure
```

Concrete procedure called

If an endpoint was registered with a pattern-based matching policy, a *Dealer* MUST supply the original `CALL.Procedure` as provided by the *Caller* in

```
    INVOCATION.Details.procedure
```

to the *Callee*.

Example

```
    [
        68,
        6131533,
        9823527,
        {
            "procedure": "com.myapp.procedure.proc1"
        },
        ["Hello, world!"]
    ]
```

11.9. Shared Registration

Feature status: **alpha**

As a default, only a single **Callee** may register a procedure for a URI.

There are use cases where more flexibility is required. As an example, for an application component with a high computing load, several instances may run, and load balancing of calls across these may be desired. As another example, in an application a second or third component providing a procedure may run, which are only to be called in case the primary component is no longer reachable (hot standby).

When shared registrations are supported, then the first **Callee** to register a procedure for a particular URI MAY determine that additional registrations for this URI are allowed, and what **Invocation Rules** to apply in case such additional registrations are made.

This is done through setting

```
    REGISTER.Options.invoke|string := <invocation_policy>
```

where `<invocation_policy>` is one of

- 'single'
- 'roundrobin'
- 'random'
- 'first'
- 'last'

If the option is not set, 'single' is applied as a default.

With 'single', the **Dealer** MUST fail all subsequent attempts to register a procedure for the URI while the registration remains in existence.

With the other values, the **Dealer** MUST fail all subsequent attempts to register a procedure for the URI where the value for this option does not match that of the initial registration.

Feature Announcement

Support for this feature MUST be announced by *Callees* (`role := "callee"`) and *Dealers* (`role := "dealer"`) via

```
    HELLO.Details.roles.<role>.features.
        shared_registration|bool := true
```

11.9.1. Load Balancing

For sets of registrations registered using either 'roundrobin' or 'random', load balancing is performed across calls to the URI.

For 'roundrobin', callees are picked subsequently from the list of registrations (ordered by the order of registration), with the picking looping back to the beginning of the list once the end has been reached.

For 'random' a callee is picked randomly from the list of registrations for each call.

11.9.2. Hot Stand-By

For sets of registrations registered using either 'first' or 'last', the first respectively last callee on the current list of registrations (ordered by the order of registration) is called.

11.10. Sharded Registration

Feature status: **sketch**

Sharded Registrations are intended to allow calling a procedure which is offered by a sharded database, by routing the call to a single shard.

Feature Announcement

Support for this feature MUST be announced by *Callers* (`role := "caller"`), *Callees* (`role := "callee"`) and *Dealers* (`role := "dealer"`) via

```
HELLO.Details.roles.<role>.features.sharded_registration|bool := true
```

11.10.1. "All" Calls

Write me.

11.10.2. "Partitioned" Calls

If `CALL.Options.runmode == "partition"`, then `CALL.Options.rkey` MUST be present.

The call is then routed to all endpoints that were registered ..

The call is then processed as for "All" Calls.

11.11. Registration Revocation

Feature status: **alpha**

This feature allows a *Dealer* to actively revoke a previously granted registration. To achieve this, the existing UNREGISTERED message is extended as described below.

Feature Announcement

Support for this feature MUST be announced by *Callees* (role := "callee") and *Dealers* (role := "dealer") via

```
    HELLO.Details.roles.<role>.features.
         registration_revocation|bool := true
```

If the *Callee* does not support registration_revocation, the *Dealer* may still revoke a registration to support administrative functionality. In this case, the *Dealer* MUST NOT send an **UNREGISTERED** message to the *Callee*. The *Callee* MAY use the registration meta event `wamp.registration.on_unregister` to determine whether a session is removed from a registration.

Extending UNREGISTERED

When revoking a registration, the router has no request ID to reply to. So it's set to zero and another argument is appended to indicate which registration to revoke. Optionally, a reason why the registration was revoked is also appended.

```
    [UNREGISTERED, 0, Details|dict]
```

where

- `Details.registration|bool` MUST be a previously issued registration ID.
- `Details.reason|string` MAY provide a reason as to why the registration was revoked.

Example

```
    [67, 0, {"registration": 1293722, "reason": "moving endpoint to
other callee"}]
```

12. Advanced PubSub

12.1. Subscriber Black- and Whitelisting

Subscriber Black- and Whitelisting is an advanced *Broker* feature where a *Publisher* is able to restrict the set of receivers of a published event.

Under normal Publish & Subscriber event dispatching, a *Broker* will dispatch a published event to all (authorized) *Subscribers* other than the *Publisher* itself. This set of receivers can be further reduced on a per-publication basis by the *Publisher* using **Subscriber Black- and Whitelisting**.

The *Publisher* can explicitly **exclude** *Subscribers* based on WAMP `sessionid`, `authid` or `authrole`. This is referred to as **Blacklisting**.

A *Publisher* may also explicitly define a **eligible** list of *Subscribers** based on WAMP `sessionid`, `authid` or `authrole`. This is referred to as **Whitelisting**.

Use Cases include the following.

Avoiding Callers from being self-notified

Consider an application that exposes a procedure to update a product price. The procedure might not only actually update the product price (e.g. in a backend database), but additionally publish an event with the updated product price, so that **all** application components get notified actively of the new price.

However, the application might want to exclude the originator of the product price update (the **Caller** of the price update procedure) from receiving the update event - as the originator naturally already knows the new price, and might get confused when it receives an update the **Caller** has triggered himself.

The product price update procedure can use `PUBLISH.Options.exclude|list[int]` to exclude the **Caller** of the procedure.

> Note that the product price update procedure needs to know the session ID of the **Caller** to be able to exclude him. For this, please see **Caller Identification**.

A similar approach can be used for other CRUD-like procedures.

Restricting receivers of sensitive information

Consider an application with users that have different `authroles`, such as "manager" and "staff" that publishes events with updates to "customers". The topics being published to could be structured like

```
    com.example.myapp.customer.<customer ID>
```

The application might want to restrict the receivers of customer updates depending on the `authrole` of the user. E.g. a user authenticated under `authrole` "manager" might be allowed to receive any kind of customer update, including personal and business sensitive information. A user under `authrole` "staff" might only be allowed to receive a subset of events.

The application can publish **all** customer updates to the **same** topic `com.example.myapp.customer.<customer ID>` and use `PUBLISH.Options.eligible_authrole|list[string]` to safely restrict the set of actual receivers as desired.

Feature Definition

A *Publisher* may restrict the actual receivers of an event from the set of *Subscribers* through the use of

- Blacklisting Options
  - `PUBLISH.Options.exclude|list[int]`
  - `PUBLISH.Options.exclude_authid|list[string]`
  - `PUBLISH.Options.exclude_authrole|list[string]`
- Whitelisting Options
  - `PUBLISH.Options.eligible|list[int]`
  - `PUBLISH.Options.eligible_authid|list[string]`
  - `PUBLISH.Options.eligible_authrole|list[string]`

`PUBLISH.Options.exclude` is a list of integers with WAMP `sessionids` providing an explicit list of (potential) *Subscribers* that won't receive a published event, even though they may be subscribed. In other words, `PUBLISH.Options.exclude` is a **blacklist** of (potential) *Subscribers*.

`PUBLISH.Options.eligible` is a list of integers with WAMP WAMP `sessionids` providing an explicit list of (potential) *Subscribers* that are allowed to receive a published event. In other words, `PUBLISH.Options.eligible` is a **whitelist** of (potential) *Subscribers*.

The `exclude_authid`, `exclude_authrole`, `eligible_authid` and `eligible_authrole` options work similar, but not on the basis of WAMP `sessionid`, but `authid` and `authrole`.

An (authorized) *Subscriber* to topic T will receive an event published to T if and only if all of the following statements hold true:

1. if there is an `eligible` attribute present, the *Subscriber*'s `sessionid` is in this list
2. if there is an `eligible_authid` attribute present, the *Subscriber*'s `authid` is in this list
3. if there is an `eligible_authrole` attribute present, the *Subscriber*'s `authrole` is in this list
4. if there is an `exclude attribute` present, the *Subscriber*'s `sessionid` is NOT in this list
5. if there is an `exclude_authid` attribute present, the *Subscriber*'s `authid` is NOT in this list
6. if there is an `exclude_authrole` attribute present, the *Subscriber*'s `authrole` is NOT in this list

For example, if both `PUBLISH.Options.exclude` and `PUBLISH.Options.eligible` are present, the *Broker* will dispatch events published only to *Subscribers* that are not explicitly excluded in `PUBLISH.Options.exclude` **and** which are explicitly eligible via `PUBLISH.Options.eligible`.

Example

```
[
   16,
   239714735,
   {
      "exclude": [
         7891255,
         1245751
      ]
   },
   "com.myapp.mytopic1",
   [
      "Hello, world!"
   ]
]
```

The above event will get dispatched to all *Subscribers* of `com.myapp.mytopic1`, but not WAMP sessions with IDs `7891255` or `1245751` (and also not the publishing session).

Example

```
[
   16,
   239714735,
   {
      "eligible": [
         7891255,
         1245751
      ]
   },
   "com.myapp.mytopic1",
   [
      "Hello, world!"
   ]
]
```

The above event will get dispatched to WAMP sessions with IDs `7891255` or `1245751` only - but only if those are actually subscribed to the topic `com.myapp.mytopic1`.

Example

```
[
   16,
   239714735,
   {
      "eligible": [
         7891255,
         1245751,
         9912315
      ],
      "exclude": [
         7891255
      ]
   },
   "com.myapp.mytopic1",
   [
      "Hello, world!"
   ]
]
```

The above event will get dispatched to WAMP sessions with IDs `1245751` or `9912315` only, since `7891255` is excluded - but only if those are actually subscribed to the topic `com.myapp.mytopic1`.

Feature Announcement

Support for this feature MUST be announced by *Publishers* (`role := "publisher"`) and *Brokers* (`role := "broker"`) via

```
HELLO.Details.roles.<role>.features.
    subscriber_blackwhite_listing|bool := true
```

12.2. Publisher Exclusion

By default, a *Publisher* of an event will **not** itself receive an event published, even when subscribed to the `Topic` the *Publisher* is publishing to. This behavior can be overridden using this feature.

To override the exclusion of a publisher from its own publication, the `PUBLISH` message must include the following option:

```
PUBLISH.Options.exclude_me|bool
```

When publishing with `PUBLISH.Options.exclude_me := false`, the *Publisher* of the event will receive that event, if it is subscribed to the `Topic` published to.

Example

```
[
    16,
    239714735,
    {
        "exclude_me": false
    },
    "com.myapp.mytopic1",
    ["Hello, world!"]
]
```

In this example, the *Publisher* will receive the published event, if it is subscribed to `com.myapp.mytopic1`.

Feature Announcement

Support for this feature MUST be announced by *Publishers* (`role := "publisher"`) and *Brokers* (`role := "broker"`) via

```
HELLO.Details.roles.<role>.features.
    publisher_exclusion|bool := true
```

12.3. Publisher Identification

A *Publisher* may request the disclosure of its identity (its WAMP session ID) to receivers of a published event by setting

```
PUBLISH.Options.disclose_me|bool := true
```

Example

```
[16, 239714735, {"disclose_me": true}, "com.myapp.mytopic1",
    ["Hello, world!"]]
```

If above event is published by a *Publisher* with WAMP session ID `3335656`, the *Broker* would send an `EVENT` message to *Subscribers* with the *Publisher's* WAMP session ID in `EVENT.Details.publisher`:

Example

```
[36, 5512315355, 4429313566, {"publisher": 3335656},
    ["Hello, world!"]]
```

Note that a *Broker* may deny a *Publisher's* request to disclose its identity:

Example

```
[8, 239714735, {}, "wamp.error.option_disallowed.disclose_me"]
```

A *Broker* may also (automatically) disclose the identity of a *Publisher* even without the *Publisher* having explicitly requested to do so when the *Broker* configuration (for the publication topic) is set up to do so.

Feature Announcement

Support for this feature MUST be announced by *Publishers* (`role := "publisher"`), *Brokers* (`role := "broker"`) and *Subscribers* (`role := "subscriber"`) via

```
HELLO.Details.roles.<role>.features.
    publisher_identification|bool := true
```

12.4. Publication Trust Levels

A *Broker* may be configured to automatically assign *trust levels* to events published by *Publishers* according to the *Broker* configuration on a per-topic basis and/or depending on the application defined role of the (authenticated) *Publisher*.

A *Broker* supporting trust level will provide

```
EVENT.Details.trustlevel|integer
```

in an `EVENT` message sent to a *Subscriber*. The trustlevel `0` means lowest trust, and higher integers represent (application-defined) higher levels of trust.

Example

```
[36, 5512315355, 4429313566, {"trustlevel": 2},
    ["Hello, world!"]]
```

In above event, the *Broker* has (by configuration and/or other information) deemed the event publication to be of trustlevel `2`.

Feature Announcement

Support for this feature MUST be announced by *Subscribers* (`role := "subscriber"`) and *Brokers* (`role := "broker"`) via

```
HELLO.Details.roles.<role>.features.
    publication_trustlevels|bool := true
```

12.5. Pattern-based Subscription

By default, *Subscribers* subscribe to topics with **exact matching policy**. That is an event will only be dispatched to a *Subscriber* by the *Broker* if the topic published to (`PUBLISH.Topic`) *exactly* matches the topic subscribed to (`SUBSCRIBE.Topic`).

A *Subscriber* might want to subscribe to topics based on a *pattern*. This can be useful to reduce the number of individual subscriptions to be set up and to subscribe to topics the *Subscriber* is not aware of at the time of subscription, or which do not yet exist at this time.

If the *Broker* and the *Subscriber* support **pattern-based subscriptions**, this matching can happen by

- prefix-matching policy
- wildcard-matching policy

Feature Announcement

Support for this feature MUST be announced by *Subscribers* (`role := "subscriber"`) and *Brokers* (`role := "broker"`) via

```
    HELLO.Details.roles.<role>.features.
        pattern_based_subscription|bool := true
```

12.5.1. Prefix Matching

A *Subscriber* requests **prefix-matching policy** with a subscription request by setting

```
    SUBSCRIBE.Options.match|string := "prefix"
```

Example

```
    [
        32,
        912873614,
        {
            "match": "prefix"
        },
        "com.myapp.topic.emergency"
    ]
```

When a **prefix-matching policy** is in place, any event with a topic that has `SUBSCRIBE.Topic` as a *prefix* will match the subscription, and potentially be delivered to *Subscribers* on the subscription.

In the above example, events with `PUBLISH.Topic`

- `com.myapp.topic.emergency.11`
- `com.myapp.topic.emergency-low`
- `com.myapp.topic.emergency.category.severe`
- `com.myapp.topic.emergency`

will all apply for dispatching. An event with `PUBLISH.Topic` e.g. `com.myapp.topic.emerge` will not apply.

12.5.2. Wildcard Matching

A *Subscriber* requests **wildcard-matching policy** with a subscription request by setting

```
SUBSCRIBE.Options.match|string := "wildcard"
```

Wildcard-matching allows to provide wildcards for **whole** URI components.

Example

```
[
    32,
    912873614,
    {
        "match": "wildcard"
    },
    "com.myapp..userevent"
]
```

In above subscription request, the 3rd URI component is empty, which signals a wildcard in that URI component position. In this example, events with `PUBLISH.Topic`

- `com.myapp.foo.userevent`
- `com.myapp.bar.userevent`
- `com.myapp.a12.userevent`

will all apply for dispatching. Events with `PUBLISH.Topic`

- `com.myapp.foo.userevent.bar`
- `com.myapp.foo.user`
- `com.myapp2.foo.userevent`

will not apply for dispatching.

12.5.3. Design Aspects

No set semantics

Since each *Subscriber's* subscription "stands on its own", there is no *set semantics* implied by pattern-based subscriptions.

E.g. a *Subscriber* cannot subscribe to a broad pattern, and then unsubscribe from a subset of that broad pattern to form a more complex subscription. Each subscription is separate.

Events matching multiple subscriptions

When a single event matches more than one of a *Subscriber's* subscriptions, the event will be delivered for each subscription.

The *Subscriber* can detect the delivery of that same event on multiple subscriptions via `EVENT.PUBLISHED.Publication`, which will be identical.

Concrete topic published to

If a subscription was established with a pattern-based matching policy, a *Broker* MUST supply the original `PUBLISH.Topic` as provided by the *Publisher* in

```
    EVENT.Details.topic|uri
```

to the *Subscribers*.

Example

```
    [
        36,
        5512315355,
        4429313566,
        {
            "topic": "com.myapp.topic.emergency.category.severe"
        },
        ["Hello, world!"]
    ]
```

12.6. Sharded Subscription

Feature status: **alpha**

Support for this feature MUST be announced by *Publishers* (`role := "publisher"`), *Subscribers* (`role := "subscriber"`) and *Brokers* (`role := "broker"`) via

```
    HELLO.Details.roles.<role>.features.shareded_subscriptions|
        bool := true
```

Resource keys: `PUBLISH.Options.rkey|string` is a stable, technical **resource key**.

> E.g. if your sensor has a unique serial identifier, you can use that.

Example

```
    [16, 239714735, {"rkey": "sn239019"}, "com.myapp.sensor.sn239019.
        temperature", [33.9]]
```

Node keys: `SUBSCRIBE.Options.nkey|string` is a stable, technical **node key**.

> E.g. if your backend process runs on a dedicated host, you can use its hostname.

Example

```
[32, 912873614, {"match": "wildcard", "nkey": "node23"},
    "com.myapp.sensor..temperature"]
```

12.7. Event History

Instead of complex QoS for message delivery, a *Broker* may provide *message history*. A *Subscriber* is responsible to handle overlaps (duplicates) when it wants "exactly-once" message processing across restarts.

The *Broker* may allow for configuration on a per-topic basis.

The event history may be transient or persistent message history (surviving *Broker* restarts).

A *Broker* that implements *event history* must (also) announce role `HELLO.roles.callee`, indicate `HELLO.roles.broker.history == 1` and provide the following (builtin) procedures.

A *Caller* can request message history by calling the *Broker* procedure

```
wamp.topic.history.last
```

with `Arguments = [topic|uri, limit|integer]` where

- `topic` is the topic to retrieve event history for
- `limit` indicates the number of last N events to retrieve

or by calling

```
wamp.topic.history.since
```

with `Arguments = [topic|uri, timestamp|string]` where

- `topic` is the topic to retrieve event history for
- `timestamp` indicates the UTC timestamp since when to retrieve the events in the ISO-8601 format `yyyy-MM-ddThh:mm:ss:SSSZ` (e.g. `"2013-12-21T13:43:11:000Z"`)

or by calling

```
wamp.topic.history.after
```

with `Arguments = [topic|uri, publication|id]`

- `topic` is the topic to retrieve event history for
- `publication` is the id of an event which marks the start of the events to retrieve from history

FIXME

1. Should we use `topic|uri` or `subscription|id` in `Arguments`?
   - Since we need to be able to get history for pattern-based subscriptions as well, a subscription|id makes more sense: create pattern-based subscription, then get the event history for this.
   - The only restriction then is that we may not get event history without a current subscription covering the events. This is a minor inconvenience at worst.
2. Can `wamp.topic.history.after` be implemented (efficiently) at all?
3. How does that interact with pattern-based subscriptions?
4. The same question as with the subscriber lists applies where: to stay within our separation of roles, we need a broker + a separate peer which implements the callee role. Here we do not have a mechanism to get the history from the broker.
5. How are black/whitelisted sessionIDs treated? A client which requests event history will have a different sessionID than on previous connections, and may receive events for which it was excluded in the previous session, or not receive events for which it was whitelisted. - see https://github.com/wamp-proto/wamp-proto/issues/206

Feature Announcement

Support for this feature MUST be announced by *Subscribers* (`role := "subscriber"`) and *Brokers* (`role := "broker"`) via

```
HELLO.Details.roles.<role>.features.event_history|bool := true
```

12.8. Event Retention

Event Retention is where a particular topic has an event associated with it which is delivered upon an opting-in client subscribing to the topic.

It can be used for topics that generally have single or very few Publishers notifying Subscribers of a single updating piece of data -- for example, a topic where a sensor publishes changes of temperature & humidity in a data center. It may do this every time the data changes (making the time between updates potentially very far apart), which causes an issue for new Subscribers who may need the last-known value upon connection, rather than waiting an unknown period of time until it is updated. Event Retention covers this use case by allowing the Publisher to mark a event as 'retained', bound to the topic it was sent to, which can be delivered upon a new client subscription that asks for it. It is similar to Event History, but allows the publisher to decide what the most important recent event is on the topic, even if other events are being delivered.

A *Broker* that advertises support MAY provide *event retention* on topics it provides. This event retention SHOULD be provided on a best-effort basis, and MUST NOT be interpreted as permanent or reliable storage by clients. This event retention is limited to one event that all subscribers would receive, and MAY include other supplemental events that have limited distribution (for example, a event published with subscriber black/whitelisting).

A *Publisher* can request storage of a new Retained Event by setting `Publish.Options.retain|bool` to `true`. Lack of the key in `Publish.Options` MUST be interpreted as a `false` value. A Broker MAY decline to provide event retention on certain topics by ignoring the `Publish.Options.retain` value. Brokers that allow event retention on the given topic MUST set the topic Retained Event to this if it were eligible to be published on the topic.

Subscribers may request access to the Retained Event by setting `Subscribe.Options.get_retained|bool` to `true`. Lack of the key in `Subscribe.Options` MUST be interpreted as a `false` value. When they opt-in to receiving the Retained Event, the Broker MUST send the Subscriber the **most recent** Retained Event that they would have received if they were subscribing when it was published. The Broker MUST NOT send the Subscriber a Retained Event that they would not be eligible to receive if they were subscribing when it was published. The *Retained Event*, as sent to the subscribing client, MUST have `Event.Details.retained|bool` set to `true`, to inform subscribers that it is not an immediately new message.

Feature Announcement

Support for this feature MUST be announced by *Brokers* (`role := "broker"`) via

```
Welcome.Details.roles.broker.features.event_retention|bool := true
```

12.9. Subscription Revocation

Feature status: **alpha**

This feature allows a *Broker* to actively revoke a previously granted subscription. To achieve this, the existing UNSUBSCRIBED message is extended as described below.

Feature Announcement

Support for this feature MUST be announced by *Subscribers* (role := "subscriber") and *Brokers* (role := "broker") via

```
HELLO.Details.roles.<role>.features.
     subscription_revocation|bool := true
```

If the *Subscriber* does not support subscription_revocation, the *Broker* MAY still revoke a subscription to support administrative functionality. In this case, the *Broker* MUST NOT send an **UNSUBSCRIBED** message to the *Subscriber*. The *Subscriber* MAY use the subscription meta event `wamp.subscription.on_unsubscribe` to determine whether a session is removed from a subscription.

Extending UNSUBSCRIBED

When revoking a subscription, the router has no request ID to reply to. So it's set to zero and another argument is appended to indicate which subscription to revoke. Optionally, a reason why the subscription was revoked is also appended.

```
[UNSUBSCRIBED, 0, Details|dict]
```

where

- `Details.subscription|bool` MUST be a previously issued subscription ID.
- `Details.reason|string` MAY provide a reason as to why the subscription was revoked.

Example

```
[35, 0, {"subscription": 1293722, "reason": "no longer authorized"}]
```

12.10. Session Testament

When a WAMP client disconnects, or the WAMP session is destroyed, it may want to notify other subscribers or publish some fixed data. Since a client may disconnect uncleanly, this can't be done reliably by them. A *Testament*, however, set on the server, can be reliably sent by the *Broker* once either the WAMP session has detached or the client connection has been lost, and allows this functionality. It can be triggered when a Session is either detached (the client has disconnected from it, or frozen it, in the case of Session Resumption) or destroyed (when the WAMP session no longer exists on the server).

This allows clients that otherwise would not be able to know when other clients disconnect get a notification (for example, by using the WAMP Session Meta API) with a format the disconnected client chose.

Feature Announcement

Support for this feature MUST be announced by *Dealers* (`role := "dealer"`) via

```
HELLO.Details.roles.dealer.features.
    testament_meta_api|bool := true
```

Testament Meta Procedures

A *Client* can call the following procedures to set/flush Testaments:

- `wamp.session.add_testament` to add a Testament which will be published on a particular topic when the Session is detached or destroyed.
- `wamp.session.flush_testaments` to remove the Testaments for that Session, either for when it is detached or destroyed.

wamp.session.add_testament

Adds a new testament:

Positional arguments

1. `topic|uri` - the topic to publish the event on
2. `args|list` - positional arguments for the event
3. `kwargs|dict` - keyword arguments for the event

Keyword arguments

1. `publish_options|dict` - options for the event when it is published -- see `Publish.Options`. Not all options may be honoured (for example, `acknowledge`). By default, there are no options.
2. `scope|string` - When the testament should be published. Valid values are `detached` (when the WAMP session is detached, for example, when using Event Retention) or `destroyed` (when the WAMP session is finalized and destroyed on the Broker). Default MUST be `destroyed`.

`wamp.session.add_testament` does not return a value.

wamp.session.flush_testaments

Removes testaments for the given scope:

Keyword arguments

1. `scope|string` - Which set of testaments to be removed. Valid values are the same as `wamp.session.add_testament`, and the default MUST be `destroyed`.

`wamp.session.flush_testaments` does not return a value.

Testaments in Use

A *Client* that wishes to send some form of data when their *Session* ends unexpectedly or their *Transport* becomes lost can set a testament using the WAMP Testament Meta API, when a *Router* supports it. For example, a client may call `add_testament` (this example uses the implicit `scope` option of `destroyed`):

```
yield self.call('wamp.session.add_testament',
                'com.myapp.mytopic', ['Seeya!'], {'my_name': 'app1'})
```

The *Router* will then store this information on the WAMP Session, either in a `detached` or `destroyed` bucket, in the order they were added. A client MUST be able to set multiple testaments per-scope. If the *Router* does not support Session Resumption (therefore removing the distinction between a detached and destroyed session), it MUST still use these two separate buckets to allow `wamp.session.flush_testaments` to work.

When a *Session* is *detached*, the *Router* will inspect it for any Testaments in the `detached` scope, and publish them in the order that the Router received them, on the specified topic, with the specified arguments, keyword arguments, and publish options. The *Router* MAY ignore publish options that do not make sense for a Testament (for example, acknowledged publishes).

When a *Session* is going to be *destroyed*, the *Router* will inspect it for any Testaments in the `destroyed` scope, and publish them in the same way as it would for the `detached` scope, in the order that they were received.

A *Router* that does not allow Session Resumption MUST send `detached`-scope Testaments before `destroyed`-scope Testaments.

A *Client* can also clear testaments if the information is no longer relevant (for example, it is shutting down completely cleanly). For example, a client may call `wamp.session.flush_testaments`:

```
yield self.call('wamp.session.flush_testaments', scope='detached')
yield self.call('wamp.session.flush_testaments', scope='destroyed')
```

The *Router* will then flush all Testaments stored for the given scope.

13. Authentication Methods

Authentication is a complex area. Some applications might want to leverage authentication information coming from the transport underlying WAMP, e.g. HTTP cookies or TLS certificates.

Some transports might imply trust or implicit authentication by their very nature, e.g. Unix domain sockets with appropriate file system permissions in place.

Other application might want to perform their own authentication using external mechanisms (completely outside and independent of WAMP).

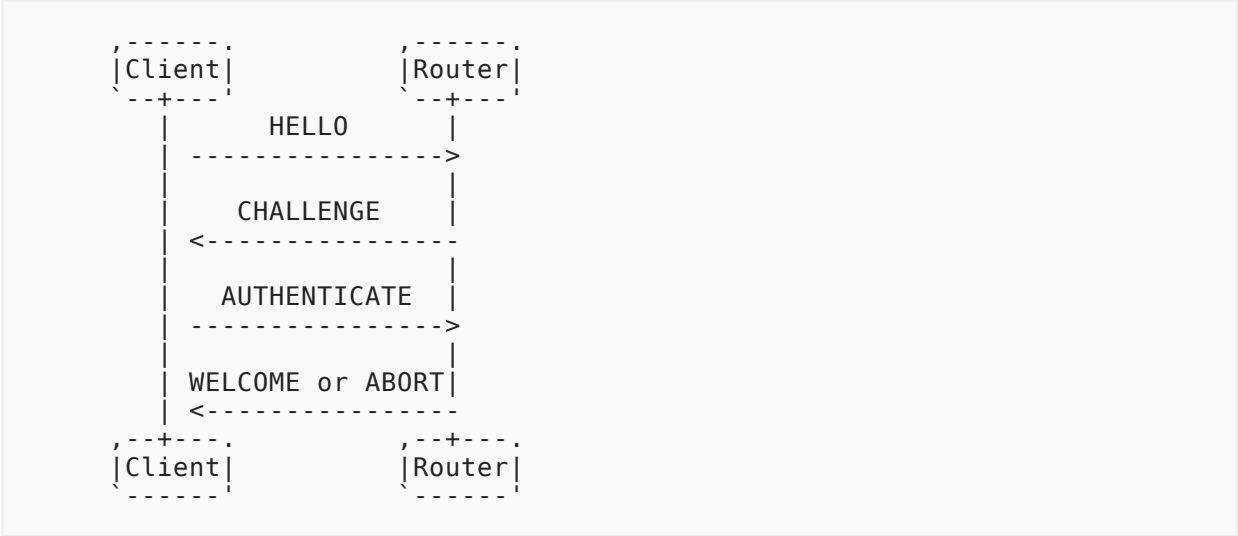
Some applications might want to perform their own authentication schemes by using basic WAMP mechanisms, e.g. by using application-defined remote procedure calls.

And some applications might want to use a transport independent scheme, nevertheless predefined by WAMP.

WAMP-level Authentication

The message flow between Clients and Routers for establishing and tearing down sessions MAY involve the following messages which authenticate a session:

1. `CHALLENGE`
2. `AUTHENTICATE`



Concrete use of `CHALLENGE` and `AUTHENTICATE` messages depends on the specific authentication method.

See WAMP Challenge-Response Authentication or ticket authentication for the use in these authentication methods.

If two-factor authentication is desired, then two subsequent rounds of `CHALLENGE` and `RESPONSE` may be employed.

CHALLENGE

An authentication MAY be required for the establishment of a session. Such requirement MAY be based on the `Realm` the connection is requested for.

To request authentication, the Router MUST send a `CHALLENGE` message to the *Endpoint*.

```
[CHALLENGE, AuthMethod|string, Extra|dict]
```

AUTHENTICATE

In response to a `CHALLENGE` message, the Client MUST send an `AUTHENTICATE` message.

```
[AUTHENTICATE, Signature|string, Extra|dict]
```

If the authentication succeeds, the Router MUST send a `WELCOME` message, else it MUST send an `ABORT` message.

Transport-level Authentication

Cookie-based Authentication

When running WAMP over WebSocket, the transport provides HTTP client cookies during the WebSocket opening handshake. The cookies can be used to authenticate one peer (the client) against the other (the server). The other authentication direction cannot be supported by cookies.

This transport-level authentication information may be forwarded to the WAMP level within `HELLO.Details.transport.auth|any` in the client-to-server direction.

TLS Certificate Authentication

When running WAMP over a TLS (either secure WebSocket or raw TCP) transport, a peer may authenticate to the other via the TLS certificate mechanism. A server might authenticate to the client, and a client may authenticate to the server (TLS client-certificate based authentication).

This transport-level authentication information may be forwarded to the WAMP level within `HELLO.Details.transport.auth|any` in both directions (if available).

13.1. Ticket-based Authentication

With *Ticket-based authentication*, the client needs to present the server an authentication "ticket" - some magic value to authenticate itself to the server.

This "ticket" could be a long-lived, pre-agreed secret (e.g. a user password) or a short-lived authentication token (like a Kerberos token). WAMP does not care or interpret the ticket presented by the client.

> Caution: This scheme is extremely simple and flexible, but the resulting security may be limited. E.g., the ticket value will be sent over the wire. If the transport WAMP is running over is not encrypted, a man-in-the-middle can sniff and possibly hijack the ticket. If the ticket value is reused, that might enable replay attacks.

A typical authentication begins with the client sending a `HELLO` message specifying the `ticket` method as (one of) the authentication methods:

```
[1, "realm1",
  {
    "roles": ...,
    "authmethods": ["ticket"],
    "authid": "joe"
  }
]
```

The `HELLO.Details.authmethods|list` is used by the client to announce the authentication methods it is prepared to perform. For Ticket-based, this MUST include `"ticket"`.

The `HELLO.Details.authid|string` is the authentication ID (e.g. username) the client wishes to authenticate as. For Ticket-based authentication, this MUST be provided.

If the server is unwilling or unable to perform Ticket-based authentication, it'll either skip forward trying other authentication methods (if the client announced any) or send an `ABORT` message.

If the server is willing to let the client authenticate using a ticket and the server recognizes the provided `authid`, it'll send a `CHALLENGE` message:

```
[4, "ticket", {}]
```

The client will send an `AUTHENTICATE` message containing a ticket:

```
[5, "secret!!!", {}]
```

The server will then check if the ticket provided is permissible (for the `authid` given).

If the authentication succeeds, the server will finally respond with a `WELCOME` message:

```
[2, 3251278072152162,
  {
    "authid": "joe",
    "authrole": "user",
    "authmethod": "ticket",
    "authprovider": "static",
    "roles": ...
  }
]
```

where

1. `authid|string`: The authentication ID the client was (actually) authenticated as.
2. `authrole|string`: The authentication role the client was authenticated for.
3. `authmethod|string`: The authentication method, here `"ticket"`
4. `authprovider|string`: The actual provider of authentication. For Ticket-based authentication, this can be freely chosen by the app, e.g. `static` or `dynamic`.

The `WELCOME.Details` again contain the actual authentication information active. If the authentication fails, the server will response with an `ABORT` message.

13.2. Challenge Response Authentication

WAMP Challenge-Response ("WAMP-CRA") authentication is a simple, secure authentication mechanism using a shared secret. The client and the server share a secret. The secret never travels the wire, hence WAMP-CRA can be used via non-TLS connections. The actual pre-sharing of the secret is outside the scope of the authentication mechanism.

A typical authentication begins with the client sending a `HELLO` message specifying the `wampcra` method as (one of) the authentication methods:

```
[1, "realm1",
    {
        "roles": ...,
        "authmethods": ["wampcra"],
        "authid": "peter"
    }
]
```

The `HELLO.Details.authmethods|list` is used by the client to announce the authentication methods it is prepared to perform. For WAMP-CRA, this MUST include `"wampcra"`.

The `HELLO.Details.authid|string` is the authentication ID (e.g. username) the client wishes to authenticate as. For WAMP-CRA, this MUST be provided.

If the server is unwilling or unable to perform WAMP-CRA authentication, it MAY either skip forward trying other authentication methods (if the client announced any) or send an `ABORT` message.

If the server is willing to let the client authenticate using WAMP-CRA, and the server recognizes the provided `authid`, it MUST send a `CHALLENGE` message:

```
[4, "wampcra",
    {
        "challenge": "{ \"nonce\": \"LHRTC9zeOIrt_9U3\",
            \"authprovider\": \"userdb\", \"authid\": \"peter\",
            \"timestamp\": \"2014-06-22T16:36:25.448Z\",
            \"authrole\": \"user\", \"authmethod\": \"wampcra\",
            \"session\": 3251278072152162}"
    }
]
```

The `CHALLENGE.Details.challenge|string` is a string the client needs to create a signature for. The string MUST BE a JSON serialized object which MUST contain:

1. `authid|string`: The authentication ID the client will be authenticated as when the authentication succeeds.
2. `authrole|string`: The authentication role the client will be authenticated as when the authentication succeeds.
3. `authmethod|string`: The authentication methods, here `"wampcra"`
4. `authprovider|string`: The actual provider of authentication. For WAMP-CRA, this can be freely chosen by the app, e.g. `userdb`.
5. `nonce|string`: A random value.
6. `timestamp|string`: The UTC timestamp (ISO8601 format) the authentication was started, e.g. `2014-06-22T16:51:41.643Z`.

7. `session|int`: The WAMP session ID that will be assigned to the session once it is authenticated successfully.

The client needs to compute the signature as follows:

```
signature := HMAC[SHA256]_{secret} (challenge)
```

That is, compute the HMAC-SHA256 using the shared `secret` over the `challenge`.

After computing the signature, the client will send an `AUTHENTICATE` message containing the signature, as a base64-encoded string:

```
[5, "gir1mSx+deCDUV7wRM5SGIn/+R/ClqLZuH4m7FJeBVI=", {}]
```

The server will then check if

- the signature matches the one expected
- the `AUTHENTICATE` message was sent in due time

If the authentication succeeds, the server will finally respond with a `WELCOME` message:

```
[2, 3251278072152162,
    {
        "authid": "peter",
        "authrole": "user",
        "authmethod": "wampcra",
        "authprovider": "userdb",
        "roles": ...
    }
]
```

The `WELCOME.Details` again contain the actual authentication information active.

If the authentication fails, the server will response with an `ABORT` message.

Server-side Verification

The challenge sent during WAMP-CRA contains

1. random information (the `nonce`) to make WAMP-CRA robust against replay attacks
2. timestamp information (the `timestamp`) to allow WAMP-CRA timeout on authentication requests that took too long
3. session information (the `session`) to bind the authentication to a WAMP session ID
4. all the authentication information that relates to authorization like `authid` and `authrole`

Three-legged Authentication

The signing of the challenge sent by the server usually is done directly on the client. However, this is no strict requirement.

E.g. a client might forward the challenge to another party (hence the "three-legged") for creating the signature. This can be used when the client was previously already authenticated to that third party, and WAMP-CRA should run piggy packed on that authentication.

The third party would, upon receiving a signing request, simply check if the client is already authenticated, and if so, create a signature for WAMP-CRA.

In this case, the secret is actually shared between the WAMP server who wants to authenticate clients using WAMP-CRA and the third party server, who shares a secret with the WAMP server.

This scenario is also the reason the challenge sent with WAMP-CRA is not simply a random value, but a JSON serialized object containing sufficient authentication information for the third party to check.

Password Salting

WAMP-CRA operates using a shared secret. While the secret is never sent over the wire, a shared secret often requires storage of that secret on the client and the server - and storing a password verbatim (unencrypted) is not recommended in general.

WAMP-CRA allows the use of salted passwords following the PBKDF2 key derivation scheme. With salted passwords, the password itself is never stored, but only a key derived from the password and a password salt. This derived key is then practically working as the new shared secret.

When the password is salted, the server will during WAMP-CRA send a `CHALLENGE` message containing additional information:

```
[4, "wampcra",
    {
        "challenge": "{ \"nonce\": \"LHRTC9zeOIrt_9U3\",
            \"authprovider\": \"userdb\", \"authid\": \"peter\",
            \"timestamp\": \"2014-06-22T16:36:25.448Z\",
            \"authrole\": \"user\", \"authmethod\": \"wampcra\",
            \"session\": 3251278072152162}",
        "salt": "salt123",
        "keylen": 32,
        "iterations": 1000
    }
]
```

The `CHALLENGE.Details.salt|string` is the password salt in use. The `CHALLENGE.Details.keylen|int` and `CHALLENGE.Details.iterations|int` are parameters for the PBKDF2 algorithm.

13.3. Salted Challenge Response Authentication

The WAMP Salted Challenge Response Authentication Mechanism ("WAMP-SCRAM"), is a password-based authentication method where the shared secret is neither transmitted nor stored as cleartext. WAMP-SCRAM is based on RFC5802 (*Salted Challenge Response Authentication Mechanism*) and RFC7677 (*SCRAM-SHA-256 and SCRAM-SHA-256-PLUS*).

WAMP-SCRAM supports the Argon2 (draft-irtf-cfrg-argon2) password-based key derivation function, a memory-hard algorithm intended to resist cracking on GPU hardware. PBKDF2 (RFC2898) is also supported for applications that are required to use primitives currently approved by cryptographic standards.

Security Considerations

With WAMP-SCRAM, if the authentication database is stolen, an attacker cannot impersonate a user unless they guess the password offline by brute force.

In the event that the server's authentication database is stolen, and the attacker either eavesdrops on one authentication exchange or impersonates a server, the attacker gains the ability to impersonate that particular user on that server. If the same salt is used on other servers, the attacker would gain the ability to impersonate that user on all servers using the same salt. That's why it's important to use a per-user random salt.

An eavesdropper that captures a user authentication exchange has enough information to mount an offline, brute-force dictionary attack for that particular user. If passwords are sufficiently strong, the cost/time needed to crack a password becomes prohibitive.

Note that when HTML/JavaScript assets are served to a web browser, WAMP-SCRAM does not safeguard against a man-in-the-middle tampering with those assets. Those assets could be tampered with in a way that captures the user's password and sends it to the attacker.

In light of the above security concerns, a secure TLS transport is therefore advised to prevent such attacks. The channel binding feature of SCRAM can be used to ensure that the TLS endpoints are the same between client and router.

Deviations from RFC5802

1. To simplify parsing, SCRAM attributes in the authentication exchange messages are encoded as members of the `Options`/`Details` objects without escaping the `,` and `=` characters. However, the `AuthMessage` used to compute the client and server signatures DOES use the exact syntax specified in RFC5802, section 7. This makes it possible to use existing test vectors to verify WAMP-SCRAM implementations.
2. Hashing based on the weaker SHA-1 specified in RFC5802 is intentionally not supported by WAMP-SCRAM, in favor of the stronger SHA-256 specified in RFC7677.
3. The Argon2 key derivation function MAY be used instead of PBKDF2.
4. Nonces are required to be base64-encoded, which is stricter than the `printable` syntax specification of RFC5802.

5. The `"y"` channel binding flag is not used as there is currently no standard way for WAMP routers to announce channel binding capabilities.
6. The use of `authzid` for user impersonation is not supported.

authmethod Type String

`"wamp-scram"` SHALL be used as the `authmethod` type string for WAMP-SCRAM authentication. Announcement by routers of WAMP-SCRAM support is outside the scope of this document.

Base64 encoding

Base64 encoding of octet strings is restricted to canonical form with no whitespace, as defined in RFC4648 (*The Base16, Base32, and Base64 Data Encodings*).

Nonces

In WAMP-SCRAM, a *nonce* (number used once) is a base64-encoded sequence of random octets. It SHOULD be of sufficient length to make a replay attack unfeasible. A length of 16 octets (128 bits) is recommended for each of the client and server-generated nonces.

See RFC4086 (*Randomness Requirements for Security*) for best practices concerning randomness.

Salts

A *salt* is a base64-encoded sequence of random octets.

To prevent rainbow table attacks in the event of database theft, the salt MUST be generated randomly by the server **for each user**. The random salt is stored with each user record in the authentication database.

Username/Password String Normalization

Username and password strings SHALL be normalized according to the *SASLprep* profile described in RFC4013, using the *stringprep* algorithm described in RFC3454.

While SASLprep preserves the case of usernames, the server MAY choose to perform case insensitive comparisons when searching for a username in the authentication database.

Channel Binding

Channel binding is a feature that allows a higher layer to establish that the other end of an underlying secure channel is bound to its higher layer counterpart. See RFC5056 (*On the Use of Channel Bindings*) for an in-depth discussion.

RFC5929 defines binding types for use with TLS transports, of which `tls-unique` and `tls-server-end-point` are applicable for WAMP-SCRAM. For each channel binding type, there is a corresponding definition of the *channel binding data* that must be sent in response to the authentication challenge.

Negotiation and announcement of channel binding is outside the scope of this document. RFC5929 section 6 recommends that application protocols use `tls-unique` exclusively, except perhaps where server-side proxies are commonly deployed.

Note that WAMP-SCRAM channel binding is not generally possible with web browser clients due to the lack of a suitable API for this purpose.

The tls-unique Channel Binding Type

The `tls-unique` channel binding type allows the WAMP layer to establish that the other peer is authenticating over the same, unique TLS connection. The channel binding data for this type corresponds to the bytes of the first TLS Finished message, as described in RFC5929, section 3. RFC5929 section 10.2 addresses the concern of disclosing this data over the TLS channel (in short, the TLS Finished message would already be visible to eavesdroppers).

To safeguard against the *triple handshake attack* described in RFC7627, this channel binding type MUST be used over a TLS channel that uses the *extended master secret* extension, or over a TLS channel where session resumption is not permitted.

The tls-server-end-point Channel Binding Type

The `tls-server-end-point` channel binding type allows the WAMP layer to establish that the other peer is authenticating over a TLS connection to a server having been issued a Public Key Infrastructure Certificate. The channel binding data for this type is a hash of the TLS server's certificate, computed as described in RFC5929, section 4.1. The certificate is hashed to accommodate memory-constrained implementations.

Authentication Exchange

WAMP-SCRAM uses a single round of challenge/response pairs after the client authentication request and before the authentication outcome.

The mapping of RFC5802 messages to WAMP messages is as follows:

SCRAM Message	WAMP Message
`client-first-message`	`HELLO`
`server-first-message`	`CHALLENGE`
`client-final-message`	`AUTHENTICATE`
`server-final-message` with `verifier`	`WELCOME`
`server-final-message` with `server-error`	`ABORT`

Table 9

Initial Client Authentication Message

WAMP-SCRAM authentication begins with the client sending a `HELLO` message specifying the `wamp-scram` method as (one of) the authentication methods:

```
    [1, "realm1",
        {
            "roles": ...,
            "authmethods": ["wamp-scram"],
            "authid": "user",
            "authextra":
                {
                    "nonce": "egVDf3DMJh0=",
                    "channel_binding": null
                }

        }
    ]
```

where:

1. `authid|string`: The identity of the user performing authentication. This corresponds to the `username` parameter in RFC5802.
2. `authextra.nonce|string`: A base64-encoded sequence of random octets, generated by the client. See Nonces.
3. `authextra.channel_binding|string`: Optional string containing the desired channel binding type. See Channel Bindings.

Upon receiving the `HELLO` message, the server MUST terminate the authentication process by sending an `ABORT` message under any of the following circumstances:

- The server does not support the WAMP-SCRAM `authmethods`, and there are no other methods left that the server supports for this `authid`.
- The the server does not support the requested `channel_binding` type.
- (Optional) The server does not recognize the given `authid`. In this case, the server MAY proceed with a mock `CHALLENGE` message to avoid leaking information on the existence of usernames. This mock `CHALLENGE` SHOULD contain a generated `salt` value that is always the same for a given `authid`, otherwise an attacker may discover that the user doesn't actually exist.

Initial Server Authentication Message

If none of the above failure conditions apply, and the server is ready to let the client authenticate using WAMP-SCRAM, then it SHALL send a `CHALLENGE` message:

```
[4, "wamp-scram",
    {
        "nonce": "egVDf3DMJh0=SBmkFIh7sSo=",
        "salt": "aBc+fx0NAVA=",
        "kdf": "pbkdf2",
        "iterations": 4096,
        "memory": null
    }
]
```

where:

1. `nonce|string`: A server-generated nonce that is appended to the client-generated nonce sent in the previous `HELLO` message. See Nonces.
2. `salt|string`: The base64-encoded salt for this user, to be passed to the key derivation function. This value is stored with each user record in the authentication database. See Salts.
3. `kdf`: The key derivation function (KDF) used to hash the password. This value is stored with each user record in the authentication database. See Key Derivation Functions.
4. `iterations|integer`: The execution time cost factor to use for generating the `SaltedPassword` hash. This value is stored with each user record in the authentication database.
5. `memory|integer`: The memory cost factor to use for generating the `SaltedPassword` hash. This is only used by the Argon2 key derivation function, where it is stored with each user record in the authentication database.

The client MUST respond with an ABORT message if `CHALLENGE.Details.nonce` does not begin with the client nonce sent in `HELLO.Details.nonce`.

The client SHOULD respond with an ABORT message if it detects that the cost parameters are unusually low. Such low-cost parameters could be the result of a rogue server attempting to obtain a weak password hash that can be easily cracked. What constitutes unusually low parameters is implementation-specific and is not covered by this document.

Final Client Authentication Message

Upon receiving a valid `CHALLENGE` message, the client SHALL respond with an `AUTHENTICATE` message:

```
[5, "dHzbZapWIk4jUhN+Ute9ytag9zjfMHgsqmmiz7AndVQ=",
    {
        "nonce": "egVDf3DMJh0=SBmkFIh7sSo=",
        "channel_binding": null,
        "cbind_data": null
    }
]
```

where:

1. `Signature|string` argument: The base64-encoded `ClientProof`, computed as described in the SCRAM-Algorithms section.
2. `nonce|string`: The concatenated client-server nonce from the previous `CHALLENGE` message.
3. `channel_binding|string`: Optional string containing the channel binding type that was sent in the original `HELLO` message.
4. `cbind_data|string`: Optional base64-encoded channel binding data. MUST be present if and only if `channel_binding` is not `null`. The format of the binding data is dependent on the binding type. See Channel Binding.

Upon receiving the `AUTHENTICATE` message, the server SHALL then check that:

- The `AUTHENTICATE` message was received in due time.
- The `ClientProof` passed via the `Signature|string` argument is validated against the `StoredKey` and `ServerKey` stored in the authentication database. See SCRAM Algorithms.
- `nonce` matches the one previously sent via `CHALLENGE`.
- The `channel_binding` matches the one sent in the `HELLO` message.
- The `cbind_data` sent by the client matches the channel binding data that the server sees on its side of the channel.

Final Server Authentication Message - Success

If the authentication succeeds, the server SHALL finally respond with a `WELCOME` message:

```
    [2, 3251278072152162,
        {
            "authid": "user",
            "authrole": "frontend",
            "authmethod": "wamp-scram",
            "authprovider": "static",
            "roles": ...,
            "authextra":
                {
                    "verifier":
                      "v=6rriTRBi23WpRR/wtup+mMhUZUn/dB5nLTJRsjl95G4="
                }
        }
    ]
```

where:

1. `authid|string`: The authentication ID the client was actually authenticated as.
2. `authrole|string`: The authentication role the client was authenticated for.
3. `authmethod|string`: The authentication method, here `"wamp-scram"`.

4. `authprovider|string`: The actual provider of authentication. For WAMP-SCRAM authentication, this can be freely chosen by the app, e.g. static or dynamic.
5. `authextra.verifier|string`: The base64-encoded `ServerSignature`, computed as described in the SCRAM Algorithms section.

The client SHOULD check the `verifier` for mutual authentication, terminating the session if invalid.

Final Server Authentication Message - Failure

If the authentication fails, the server SHALL respond with an `ABORT` message.

The server MAY include a SCRAM-specific error string in the `ABORT` message as a `Details.scram` attribute. SCRAM error strings are listed in RFC5802, section 7, under `server-error-value`.

SCRAM Algorithms

This section is non-normative.

RFC5802 specifies the algorithms used to compute the `ClientProof`, `ServerSignature`, `ServerKey`, and `StoredKey` values referenced by this document. Those algorithms are summarized here in pseudocode for reference.

Notation

- `"="`: The variable on the left-hand side is the result of the expression on the right-hand side.
- `"+"`: String concatenation.
- `IfNull(attribute, value, else)`: If the given `attribute` is absent or null, evaluates to the given `value`, otherwise evaluates to the given `else` value.
- `Decimal(integer)`: The decimal string representation of the given `integer`.
- `Base64(octets)`: Base64 encoding of the given octet sequence, restricted to canonical form with no whitespace, as defined in RFC4648.
- `UnBase64(str)`: Decode the given Base64 string into an octet sequence.
- `Normalize(str)`: Normalize the given string using the SASLprep profile RFC4013 of the "stringprep" algorithm RFC3454.
- `XOR`: The exclusive-or operation applied to each octet of the left and right-hand-side octet sequences.
- `SHA256(str)`: The SHA-256 cryptographic hash function.
- `HMAC(key, str)`: Keyed-hash message authentication code, as defined in RFC2104, with SHA-256 as the underlying hash function.
- `KDF(str, salt, params...)`: One of the supported key derivations function, with the output key length the same as the SHA-256 output length (32 octets). `params...` are the additional parameters that are applicable for the function: `iterations` and `memory`.
- `Escape(str)`: Replace every occurrence of "," and "=" in the given string with "=2C" and "=3D" respectively.

Data Stored on the Server

For each user, the server needs to store:

1. A random, per-user salt.
2. The type string corresponding to the key derivation function (KDF) used to hash the password (e.g. `"argon2id13"`). This is needed to handle future revisions of the KDF, as well as allowing migration to stronger KDFs that may be added to WAMP-SCRAM in the future. This may also be needed if the KDF used during user registration is configurable or selectable on a per-user basis.
3. Parameters that are applicable to the key derivation function : `iterations` and possibly `memory`.
4. The `StoredKey`.
5. The `ServerKey`.

where `StoredKey` and `ServerKey` are computed as follows:

```
SaltedPassword  = KDF(Normalize(password), salt, params...)
ClientKey       = HMAC(SaltedPassword, "Client Key")
StoredKey       = SHA256(ClientKey)
ServerKey       = HMAC(SaltedPassword, "Server Key")
```

Note that `"Client Key"` and `"Server Key"` are string literals.

The manner in which the `StoredKey` and `ServerKey` are shared with the server during user registration is outside the scope of SCRAM and this document.

AuthMessage

In SCRAM, `AuthMessage` is used for computing `ClientProof` and `ServerSignature`. `AuthMessage` is computed using attributes that were sent in the first three messages of the authentication exchange.

```
ClientFirstBare = "n=" + Escape(HELLO.Details.authid) + "," +
                  "r=" + HELLO.Details.authextra.nonce

ServerFirst = "r=" + CHALLENGE.Details.nonce + "," +
              "s=" + CHALLENGE.Details.salt + "," +
              "i=" + Decimal(CHALLENGE.Details.iterations)

CBindName = AUTHENTICATE.Extra.channel_binding

CBindData = AUTHENTICATE.Extra.cbind_data

CBindFlag = IfNull(CBindName, "n", "p=" + CBindName)

CBindInput = CBindFlag + ",," +
             IfNull(CBindData, "", UnBase64(CBindData))

ClientFinalNoProof = "c=" + Base64(CBindInput) + "," +
                     "r=" + AUTHENTICATE.Extra.nonce

AuthMessage = ClientFirstBare + "," + ServerFirst + "," +
              ClientFinalNoProof
```

ClientProof

`ClientProof` is computed by the client during the authentication exchange as follows:

```
SaltedPassword  = KDF(Normalize(password), salt, params...)
ClientKey       = HMAC(SaltedPassword, "Client Key")
StoredKey       = SHA256(ClientKey)
ClientSignature = HMAC(StoredKey, AuthMessage)
ClientProof     = ClientKey XOR ClientSignature
```

The `ClientProof` is then sent to the server, base64-encoded, via the `AUTHENTICATE.Signature` argument.

The server verifies the `ClientProof` by computing the `RecoveredStoredKey` and comparing it to the actual `StoredKey`:

```
ClientSignature    = HMAC(StoredKey, AuthMessage)
RecoveredClientKey = ClientSignature XOR ReceivedClientProof
RecoveredStoredKey = SHA256(RecoveredClientKey)
```

Note that the client MAY cache the `ClientKey` and `StoredKey` (or just `SaltedPassword`) to avoid having to perform the expensive KDF computation for every authentication exchange. Storing these values securely on the client is outside the scope of this document.

ServerSignature

`ServerSignature` is computed by the server during the authentication exchange as follows:

```
ServerSignature = HMAC(ServerKey, AuthMessage)
```

The `ServerSignature` is then sent to the client, base64-encoded, via the `WELCOME.Details.authextra.verifier` attribute.

The client verifies the `ServerSignature` by computing it and comparing it with the `ServerSignature` sent by the server:

```
ServerKey       = HMAC(SaltedPassword, "Server Key")
ServerSignature = HMAC(ServerKey, AuthMessage)
```

Key Derivation Functions

SCRAM uses a password-based key derivation function (KDF) to hash user passwords. WAMP-SCRAM supports both Argon2 and PBKDF2 as the KDF. Argon2 is recommended because of its memory hardness and resistance against GPU hardware. PBKDF2, which does not feature memory hardness, is also supported for applications that are required to use primitives currently approved by cryptographic standards.

The following table maps the `CHALLENGE.Details.kdf` type string to the corresponding KDF.

KDF type string	Function
`"argon2id13"`	Argon2id variant of Argon2, version 1.3
`"pbkdf2"`	PBKDF2

Table 10

To promote interoperability, WAMP-SCRAM client/server implementations SHOULD support both of the above KDFs. During authentication, there is no "negotiation" of the KDF, and the client MUST use the same KDF than the one used to create the keys stored in the authentication database.

Which KDF is used to hash the password during user registration is up to the application and/or server implementation, and is not covered by this document. Possibilities include:

- making the KDF selectable at runtime during registration,
- making the KDF statically configurable on the server, or,
- hard-coding the KDF selection on the server.

Argon2

The Argon2 key derivation function, proposed in draft-irtf-cfrg-argon2, is computed using the following parameters:

- `CHALLENGE.Details.salt` as the cryptographic salt,

- `CHALLENGE.Details.iterations` as the number of iterations,
- `CHALLENGE.Details.memory` as the memory size (in kibibytes),
- 1 as the parallelism parameter,
- Argon2id as the algorithm variant, and,
- 32 octets as the output key length.

For WAMP-SCRAM, the parallelism parameter is fixed to 1 due to the password being hashed on the client side, where it is not generally known how many cores/threads are available on the client's device.

Section 4 of the Argon2 internet draft recommends the general procedure for selecting parameters, of which the following guidelines are applicable to WAMP-SCRAM:

- A 128-bit salt is recommended, which can be reduced to 64-bit if space is limited.
- The `memory` parameter is to be configured to the maximum amount of memory usage that can be tolerated on client devices for computing the hash.
- The `iterations` parameter is to be determined experimentally so that execution time on the client reaches the maximum that can be tolerated by users during authentication. If the execution time is intolerable with `iterations` = 1, then reduce the `memory` parameter as needed.

PBKDF2

The PBKDF2 key derivation function, defined in RFC2898, is used with SHA-256 as the pseudorandom function (PRF).

The PDBKDF2 hash is computed using the following parameters:

- `CHALLENGE.Details.salt` as the cryptographic salt,
- `CHALLENGE.Details.iterations` as the iteration count, and,
- 32 octets as the output key length (`dkLen`), which matches the SHA-256 output length.

RFC2898 section 4.1 recommends at least 64 bits for the salt.

The `iterations` parameter SHOULD be determined experimentally so that execution time on the client reaches the maximum that can be tolerated by users during authentication. RFC7677 section 4 recommends an iteration count of at least 4096, with a significantly higher value on non-mobile clients.

13.4. Cryptosign-based Authentication

Write me.

13.5. Dynamic Authentication API

Write me.

14. Advanced Transports and Serializers

The only requirements that WAMP expects from a transport are: the transport must be message-based, bidirectional, reliable and ordered. This allows WAMP to run over different transports without any impact at the application layer.

Besides the WebSocket transport, the following WAMP transports are currently specified:

- RawSocket Transport
- Batched WebSocket Transport
- LongPoll Transport
- Multiplexed Transport

> Other transports such as HTTP 2.0 ("SPDY") or UDP might be defined in the future.

14.1. RawSocket Transport

WAMP-over-RawSocket is an (alternative) transport for WAMP that uses length-prefixed, binary messages - a message framing different from WebSocket.

Compared to WAMP-over-WebSocket, WAMP-over-RawSocket is simple to implement, since there is no need to implement the WebSocket protocol which has some features that make it non-trivial (like a full HTTP-based opening handshake, message fragmentation, masking and variable length integers).

WAMP-over-RawSocket has even lower overhead than WebSocket, which can be desirable in particular when running on local connections like loopback TCP or Unix domain sockets. It is also expected to allow implementations in microcontrollers in under 2KB RAM.

WAMP-over-RawSocket can run over TCP, TLS, Unix domain sockets or any reliable streaming underlying transport. When run over TLS on the standard port for secure HTTPS (443), it is also able to traverse most locked down networking environments such as enterprise or mobile networks (unless man-in-the-middle TLS intercepting proxies are in use).

However, WAMP-over-RawSocket cannot be used with Web browser clients, since browsers do not allow raw TCP connections. Browser extensions would do, but those need to be installed in a browser. WAMP-over-RawSocket also (currently) does not support transport-level compression as WebSocket does provide (`permessage-deflate` WebSocket extension).

Endianess

WAMP-over-RawSocket uses *network byte order* ("big-endian"). That means, given a unsigned 32 bit integer

```
    0x 11 22 33 44
```

the first octet sent out to (or received from) the wire is `0x11` and the last octet sent out (or received) is `0x44`.

Here is how you would convert octets received from the wire into an integer in Python:

```
import struct

octets_received = b"\x11\x22\x33\x44"
i = struct.unpack(">L", octets_received)[0]
```

The integer received has the value `287454020`.

And here is how you would send out an integer to the wire in Python:

```
octets_to_be_send = struct.pack(">L", i)
```

The octets to be sent are `b"\x11\x22\x33\x44"`.

Handshake: Client-to-Router Request

WAMP-over-RawSocket starts with a handshake where the client connecting to a router sends 4 octets:

```
    MSB                                 LSB
    31                                    0
    0111 1111 LLLL SSSS RRRR RRRR RRRR RRRR
```

The *first octet* is a magic octet with value `0x7F`. This value is chosen to avoid any possible collision with the first octet of a valid HTTP request (see here and here). No valid HTTP request can have `0x7F` as its first octet.

> By using a magic first octet that cannot appear in a regular HTTP request, WAMP-over-RawSocket can be run e.g. on the same TCP listening port as WAMP-over-WebSocket or WAMP-over-LongPoll.

The *second octet* consists of a 4 bit `LENGTH` field and a 4 bit `SERIALIZER` field.

The `LENGTH` value is used by the *Client* to signal the **maximum message length** of messages it is willing to **receive**. When the handshake completes successfully, a *Router* MUST NOT send messages larger than this size.

The possible values for `LENGTH` are:

```
     0: 2**9 octets
     1: 2**10 octets
    ...
    15: 2**24 octets
```

This means a *Client* can choose the maximum message length between **512** and **16M** octets.

The `SERIALIZER` value is used by the *Client* to request a specific serializer to be used. When the handshake completes successfully, the *Client* and *Router* will use the serializer requested by the *Client*.

The possible values for `SERIALIZER` are:

```
    0: illegal
    1: JSON
    2: MessagePack
    3 - 15: reserved for future serializers
```

Here is a Python program that prints all (currently) permissible values for the *second octet*:

```
SERMAP = {
    1: 'json',
    2: 'messagepack'
}

# map serializer / max. msg length to RawSocket handshake
# request or success reply (2nd octet)
for ser in SERMAP:
    for l in range(16):
        octet_2 = (l << 4) | ser
        print("serializer: {}, maxlen: {} => 0x{:
02x}".format(SERMAP[ser], 2 ** (l + 9), octet_2))
```

The *third and forth octet* are **reserved** and MUST be all zeros for now.

Handshake: Router-to-Client Reply

After a *Client* has connected to a *Router*, the *Router* will first receive the 4 octets handshake request from the *Client*.

If the *first octet* differs from `0x7F`, it is not a WAMP-over-RawSocket request. Unless the *Router* also supports other transports on the connecting port (such as WebSocket or LongPoll), the *Router* MUST **fail the connection**.

Here is an example of how a *Router* could parse the *second octet* in a *Clients* handshake request:

```
# map RawSocket handshake request (2nd octet) to
# serializer / max. msg length
for i in range(256):
    ser_id = i & 0x0f
    if ser_id != 0:
        ser = SERMAP.get(ser_id, 'currently undefined')
        maxlen = 2 ** ((i >> 4) + 9)
        print("{:02x} => serializer: {}, maxlen: {}".format(i, ser,
maxlen))
    else:
        print("fail the connection: illegal serializer value")
```

When the *Router* is willing to speak the serializer requested by the *Client*, it will answer with a 4 octets response of identical structure as the *Client* request:

```
    MSB                                 LSB
    31                                    0
    0111 1111 LLLL SSSS RRRR RRRR RRRR RRRR
```

Again, the *first octet* MUST be the value `0x7F`. The *third and forth octets* are reserved and MUST be all zeros for now.

In the *second octet*, the *Router* MUST echo the serializer value in `SERIALIZER` as requested by the *Client*.

Similar to the *Client*, the *Router* sets the `LENGTH` field to request a limit on the length of messages sent by the *Client*.

During the connection, *Router* MUST NOT send messages to the *Client* longer than the `LENGTH` requested by the *Client*, and the *Client* MUST NOT send messages larger than the maximum requested by the *Router* in its handshake reply.

If a message received during a connection exceeds the limit requested, a *Peer* MUST **fail the connection**.

When the *Router* is unable to speak the serializer requested by the *Client*, or it is denying the *Client* for other reasons, the *Router* replies with an error:

```
    MSB                                 LSB
    31                                    0
    0111 1111 EEEE 0000 RRRR RRRR RRRR RRRR
```

An error reply has 4 octets: the *first octet* is again the magic `0x7F`, and the *third and forth octet* are reserved and MUST all be zeros for now.

The *second octet* has its lower 4 bits zero'ed (which distinguishes the reply from an success/accepting reply) and the upper 4 bits encode the error:

```
    0: illegal (must not be used)
    1: serializer unsupported
    2: maximum message length unacceptable
    3: use of reserved bits (unsupported feature)
    4: maximum connection count reached
    5 - 15: reserved for future errors
```

> Note that the error code `0` MUST NOT be used. This is to allow storage of error state in a host language variable, while allowing `0` to signal the current state "no error"

Here is an example of how a *Router* might create the *second octet* in an error response:

```
ERRMAP = {
    0: "illegal (must not be used)",
    1: "serializer unsupported",
    2: "maximum message length unacceptable",
    3: "use of reserved bits (unsupported feature)",
    4: "maximum connection count reached"
}

# map error to RawSocket handshake error reply (2nd octet)
for err in ERRMAP:
    octet_2 = err << 4
    print("error: {} => 0x{:02x}").format(ERRMAP[err], err)
```

The *Client* - after having sent its handshake request - will wait for the 4 octets from *Router* handshake reply.

Here is an example of how a *Client* might parse the *second octet* in a *Router* handshake reply:

```
# map RawSocket handshake reply (2nd octet)
for i in range(256):
    ser_id = i & 0x0f
    if ser_id:
        # verify the serializer is the one we requested!
        # if not, fail the connection!
        ser = SERMAP.get(ser_id, 'currently undefined')
        maxlen = 2 ** ((i >> 4) + 9)
        print("{:02x} => serializer: {}, maxlen: {}".format(i, ser,
maxlen))
    else:
        err = i >> 4
        print("error: {}".format(ERRMAP.get(err, 'currently undefined')))
```

Serialization

To send a WAMP message, the message is serialized according to the WAMP serializer agreed in the handshake (e.g. JSON or MessagePack).

The length of the serialized messages in octets MUST NOT exceed the maximum requested by the *Peer*.

If the serialized length exceed the maximum requested, the WAMP message can not be sent to the *Peer*. Handling situations like the latter is left to the implementation.

E.g. a *Router* that is to forward a WAMP `EVENT` to a *Client* which exceeds the maximum length requested by the *Client* when serialized might:

- drop the event (not forwarding to that specific client) and track dropped events
- prohibit publishing to the topic already
- remove the event payload, and send an event with extra information (`payload_limit_exceeded = true`)

Framing

The serialized octets for a message to be sent are prefixed with exactly 4 octets.

```
MSB                                   LSB
31                                     0
RRRR RTTT LLLL LLLL LLLL LLLL LLLL LLLL
```

The *first octet* has the following structure

```
MSB   LSB
7       0
RRRR RTTT
```

The five bits `RRRRR` are reserved for future use and MUST be all zeros for now.

The three bits `TTT` encode the type of the transport message:

```
0: regular WAMP message
1: PING
2: PONG
3-7: reserved
```

The *three remaining octets* constitute an unsigned 24 bit integer that provides the length of transport message payload following, excluding the 4 octets that constitute the prefix.

For a regular WAMP message (`TTT == 0`), the length is the length of the serialized WAMP message: the number of octets after serialization (excluding the 4 octets of the prefix).

For a `PING` message (`TTT == 1`), the length is the length of the arbitrary payload that follows. A *Peer* MUST reply to each `PING` by sending exactly one `PONG` immediately, and the `PONG` MUST echo back the payload of the `PING` exactly.

For receiving messages with WAMP-over-RawSocket, a *Peer* will usually read exactly 4 octets from the incoming stream, decode the transport level message type and payload length, and then receive as many octets as the length was giving.

When the transport level message type indicates a regular WAMP message, the transport level message payload is unserialized according to the serializer agreed in the handshake and the processed at the WAMP level.

14.2. Message Batching

WAMP-over-Batched-WebSocket is a variant of WAMP-over-WebSocket where multiple WAMP messages are sent in one WebSocket message.

Using WAMP message batching can increase wire level efficiency further. In particular when using TLS and the WebSocket implementation is forcing every WebSocket message into a new TLS segment.

WAMP-over-Batched-WebSocket is negotiated between Peers in the WebSocket opening handshake by agreeing on one of the following WebSocket subprotocols:

- `wamp.2.json.batched`
- `wamp.2.msgpack.batched`
- `wamp.2.cbor.batched`

Batching with JSON works by serializing each WAMP message to JSON as normally, appending the single ASCII control character `\30` (record separator) octet `0x1e` to *each* serialized messages, and packing a sequence of such serialized messages into a single WebSocket message:

```
Serialized JSON WAMP Msg 1 | 0x1e |
    Serialized JSON WAMP Msg 2 | 0x1e | ...
```

Batching with MessagePack works by serializing each WAMP message to MessagePack as normally, prepending a 32 bit unsigned integer (4 octets in big-endian byte order) with the length of the serialized MessagePack message (excluding the 4 octets for the length prefix), and packing a sequence of such serialized (length-prefixed) messages into a single WebSocket message:

```
Length of Msg 1 serialization (uint32) |
    serialized MessagePack WAMP Msg 1 | ...
```

With batched transport, even if only a single WAMP message is to be sent in a WebSocket message, the (single) WAMP message needs to be framed as described above. In other words, a single WAMP message is sent as a batch of length **1**. Sending a batch of length **0** (no WAMP message) is illegal and a *Peer* MUST fail the transport upon receiving such a transport message.

14.3. HTTP Longpoll Transport

The *Long-Poll Transport* is able to transmit a WAMP session over plain old HTTP 1.0/1.1. This is realized by the Client issuing HTTP/POSTs requests, one for sending, and one for receiving. Those latter requests are kept open at the server when there are no messages currently pending to be received.

Opening a Session

With the Long-Poll Transport, a Client opens a new WAMP session by sending a HTTP/POST request to a well-known URL, e.g.

```
http://mypp.com/longpoll/open
```

Here, `http://mypp.com/longpoll` is the base URL for the Long-Poll Transport and `/open` is a path dedicated for opening new sessions.

The HTTP/POST request SHOULD have a `Content-Type` header set to `application/json` and MUST have a request body with a JSON document that is a dictionary:

```
{
   "protocols": ["wamp.2.json"]
}
```

The (mandatory) `protocols` attribute specifies the protocols the client is willing to speak. The server will chose one from this list when establishing the session or fail the request when no protocol overlap was found.

The valid protocols are:

- `wamp.2.json.batched`
- `wamp.2.json`
- `wamp.2.msgpack.batched`
- `wamp.2.msgpack`
- `wamp.2.cbor.batched`
- `wamp.2.cbor`

> The request path with this and subsequently described HTTP/POST requests MAY contain a query parameter x with some random or sequentially incremented value:
>
> http://mypp.com/longpoll/open?x=382913
>
> The value is ignored, but may help in certain situations to prevent intermediaries from caching the request.

Returned is a JSON document containing a transport ID and the protocol to speak:

```
{
   "protocol": "wamp.2.json",
   "transport": "kjmd3sBLOUnb3Fyr"
}
```

As an implied side-effect, two HTTP endpoints are created

```
http://mypp.com/longpoll/<transport_id>/receive
http://mypp.com/longpoll/<transport_id>/send
```

where `transport_id` is the transport ID returned from `open`, e.g.

```
http://mypp.com/longpoll/kjmd3sBLOUnb3Fyr/receive
http://mypp.com/longpoll/kjmd3sBLOUnb3Fyr/send
```

Receiving WAMP Messages

The Client will then issue HTTP/POST requests (with empty request body) to

```
http://mypp.com/longpoll/kjmd3sBLOUnb3Fyr/receive
```

When there are WAMP messages pending downstream, a request will return with a single WAMP message (unbatched modes) or a batch of serialized WAMP messages (batched mode).

The serialization format used is the one agreed during opening the session.

The batching uses the same scheme as with `wamp.2.json.batched` and `wamp.2.msgpack.batched` transport over WebSocket.

> Note: In unbatched mode, when there is more than one message pending, there will be at most one message returned for each request. The other pending messages must be retrieved by new requests. With batched mode, all messages pending at request time will be returned in one batch of messages.

Sending WAMP Messages

For sending WAMP messages, the *Client* will issue HTTP/POST requests to

```
http://mypp.com/longpoll/kjmd3sBLOUnb3Fyr/send
```

with request body being a single WAMP message (unbatched modes) or a batch of serialized WAMP messages (batched mode).

The serialization format used is the one agreed during opening the session.

The batching uses the same scheme as with `wamp.2.json.batched` and `wamp.2.msgpack.batched` transport over WebSocket.

Upon success, the request will return with HTTP status code 202 ("no content"). Upon error, the request will return with HTTP status code 400 ("bad request").

Closing a Session

To orderly close a session, a Client will issue a HTTP/POST to

```
http://mypp.com/longpoll/kjmd3sBLOUnb3Fyr/close
```

with an empty request body. Upon success, the request will return with HTTP status code 202 ("no content").

15. WAMP Interfaces

WAMP was designed with the goals of being easy to approach and use for application developers. Creating a procedure to expose some custom functionality should be possible in any supported programming language using that language's native elements, with the least amount of additional effort.

Following from that, WAMP uses *dynamic typing* for the application payloads of calls, call results and error, as well as event payloads.

A WAMP router will happily forward *any* application payload on *any* procedure or topic URI as long as the client is *authorized* (has permission) to execute the respective WAMP action (call, register, publish or subscribe) on the given URI.

This approach has served WAMP well, as application developers can get started immediately, and evolve and change payloads as they need without extra steps. These advantages in flexibility of course come at a price, as nothing is free, and knowing that price is important to be aware of the tradeoffs one is accepting when using dynamic typing:

- problematic coordination of *Interfaces* within larger developer teams or between different parties
- no easy way to stabilize, freeze, document or share *Interfaces*
- no way to programmatically describe *Interfaces* ("interface reflection") at run-time

Problems such above could be avoided when WAMP supported an *option* to formally define WAMP-based *Interfaces*. This needs to answer the following questions:

1. How to specify the `args|List` and `kwargs|Dict` application payloads that are used in WAMP calls, errors and events?

2. How to specify the type and URI (patterns) for WAMP RPCs *Procedures* and WAMP PubSub *Topics* that make up an *Interface*, and how to identify an *Interface* itself as a collection of *Procedures* and *Topics*?
3. How to package, publish and share *Catalogs* as a collection of *Interfaces* plus metadata

The following sections will describe the solution to each of above questions using WAMP IDL.

Using WAMP Interfaces finally allows to support the following application developer level features:

1. router-based application payload validation and enforcement
2. WAMP interface documentation generation and autodocs Web service
3. publication and sharing of WAMP Interfaces and Catalogs
4. client binding code generation from WAMP Interfaces
5. run-time WAMP type reflection as part of the WAMP meta API

15.1. WAMP IDL

15.1.1. Application Payload Typing

To define the application payload `Arguments|list` and `ArgumentsKw|dict`, WAMP IDL reuses the FlatBuffers IDL, specifically, we map a pair of `Arguments|list` and `ArgumentsKw|dict` to a FlatBuffers Table with WAMP defined FlatBuffers *Attributes*.

User defined WAMP application payloads are transmitted in `Arguments|list` and `ArgumentsKw|dict` elements of the following WAMP messages:

- `PUBLISH`
- `EVENT`
- `CALL`
- `INVOCATION`
- `YIELD`
- `RESULT`
- `ERROR`

A *Publisher* uses the

- `PUBLISH.Arguments|list` and `PUBLISH.ArgumentsKw|dict`

message elements to send the event payload to be published to the *Broker* in `PUBLISH` messages. When the event is accepted by the *Broker*, it will dispatch an `EVENT` message with

- `EVENT.Arguments|list` and `EVENT.ArgumentsKw|dict`

message elements to all (eligible, and not excluded) *Subscribers*.

A *Caller* uses the

- `CALL.Arguments|list` and `CALL.ArgumentsKw|dict`

message elements to send the call arguments to be used to the *Dealer* in `CALL` messages. When the call is accepted by the *Dealer*, it will forward

- `INVOCATION.Arguments|list` and `INVOCATION.ArgumentsKw|dict`

to the (or one of) *Callee*, and receive `YIELD` messages with

- `YIELD.Arguments|list` and `YIELD.ArgumentsKw|dict`

message elements, which it will return to the original *Caller* in `RESULT` messages with

- `RESULT.Arguments|list` and `RESULT.ArgumentsKw|dict`

In the error case, a *Callee* MAY return an `ERROR` message with

- `ERROR.Arguments|list` and `ERROR.ArgumentsKw|dict`

message elements, which again is returned to the original *Caller*.

> It is important to note that the above messages and message elements are the only ones free for use with application and user defined payloads. In particular, even though the following WAMP messages and message element carry payloads defined by the specific WAMP authentication method used, they do *not* carry arbitrary application payloads: `HELLO.Details["authextra"]|dict`, `WELCOME.Details["authextra"]|dict`, `CHALLENGE.Extra|dict`, `AUTHENTICATE.Extra|dict`.

For example, the Session Meta API includes a procedure to kill all sessions by authid with:

Positional arguments (`args|list`)

1. `authid|string` - The authentication ID identifying sessions to close.

Keyword arguments (`kwargs|dict`)

1. `reason|uri` - reason for closing sessions, sent to clients in `GOODBYE.Reason`
2. `message|string` - additional information sent to clients in `GOODBYE.Details` under the key "message".

as arguments. When successful, this procedure will return a call result with:

Positional results (`results|list`)

1. `sessions|list` - The list of WAMP session IDs of session that were killed.

Keyword results (`kwresults|dict`)

1. `None`

To specify the call arguments in FlatBuffers IDL, we can define a FlatBuffers table for both `args` and `kwargs`:

```
/// Call args/kwargs for "wamp.session.kill_by_authid"
table SessionKillByAuthid
{
    /// The WAMP authid of the sessions to kill.
    authid: string (wampuri);

    /// A reason URI provided to the killed session(s).
    reason: string (kwarg, wampuri);

    /// A message provided to the killed session(s).
    message: string (kwarg);
}
```

The table contains the list `args` as table elements (in order), unless the table element has an *Attribute* `kwarg`, in which case the element one in `kwarg`.

The attributes `wampid` and `wampuri` are special markers that denote values that follow the respective WAMP identifier rules for WAMP IDs and URIs.

When successful, the procedure will return a list of WAMP session IDs of session that were killed. Again, we can map this to FlatBuffers IDL:

```
table WampIds
{
    /// List of WAMP IDs.
    value: [uint64] (wampid);
}
```

15.1.2. WAMP IDL Attributes

WAMP IDL uses *custom FlatBuffer attributes* to

- mark `kwarg` fields which map to WAMP keyword argument vs `arg` (default)
- declare fields of a scalar base type to follow (stricter) WAMP rules (for IDs and URIs)
- specify the WAMP action type, that is *Procedure* vs *Topic*, on service declarations

> "Attributes may be attached to a declaration, behind a field, or after the name of a table/struct/enum/union. These may either have a value or not. Some attributes like deprecated are understood by the compiler; user defined ones need to be declared with the attribute declaration (like priority in the example above), and are available to query

> if you parse the schema at runtime. This is useful if you write your own code generators/ editors etc., and you wish to add additional information specific to your tool (such as a help text)." (from source).

The *Attributes* used in WAMP IDL are defined in `<WAMP API Catalog>/src/wamp.fbs`, and are described in the following sections:

- `arg`, `kwarg`
- `wampid`
- `wampname`, `wampname_s`
- `wampuri`, `wampuri_s`, `wampuri_p`, `wampuri_sp`, `wampuri_pp`, `wampuri_spp`
- `uuid`
- `ethadr`
- `type`

WAMP Positional and Keyword-based Payloads

Positional payloads `args|list` and keyword-based payloads `kwargs|dict` are table elements that have one of the following *Attributes*:

- `arg` (default)
- `kwarg`

One pair of `args` and `kwarg` types is declared by one FlatBuffer table with optional attributes on table fields, and the following rules apply or must be followed:

1. If neither `arg` nor `kwarg` attribute is provided, `arg` is assumed.
2. Only one of either `arg` or `kwarg` MUST be specified.
3. When a field has an attribute `kwarg`, all subsequent fields in the same table MUST also have attribute `kwarg`.

WAMP IDs and URIs

Integers which contain WAMP IDs use *Attribute*

1. `wampid`: WAMP ID, that is an integer `[1, 2^53]`

Strings which contain WAMP names ("URI components"), for e.g. WAMP roles or authids use *Attributes*

1. `wampname`: WAMP URI component (aka "name"), loose rules (minimum required to combine to dotted URIs), must match regular expression `^[^\s\.#]+$`.
2. `wampname_s`: WAMP URI component (aka "name"), strict rules (can be used as identifier in most languages), must match regular expression `^[\da-z_]+$`.

Strings which contain WAMP URIs or URI patterns use *Attribute*

1. `wampuri`: WAMP URI, loose rules, no empty URI components (aka "concrete or fully qualified URI"), must match regular expression `^([^\s\.#]+\.)*([^\s\.#]+)$`.
2. `wampuri_s`: WAMP URI, strict rules, no empty URI components, must match regular expression `^([\da-z_]+\.)*([\da-z_]+)$`.
3. `wampuri_p`: WAMP URI or URI (prefix or wildcard) pattern, loose rules (minimum required to combine to dotted URIs), must match regular expression `^(([^\s\.#]+\.)|\.)*([^\s\.#]+)?$`.
4. `wampuri_sp`: WAMP URI or URI (prefix or wildcard) pattern, strict rules (can be used as identifier in most languages), must match regular expression `^(([\da-z_]+\.)|\.)*([\da-z_]+)?$`.
5. `wampuri_pp`: WAMP URI or URI prefix pattern, loose rules (minimum required to combine to dotted URIs), must match regular expression `^([^\s\.#]+\.)*([^\s\.#]*)$`.
6. `wampuri_spp`: WAMP URI or URI prefix pattern, strict rules (can be used as identifier in most languages), must match regular expression `^([\da-z_]+\.)*([\da-z_]*)$`.

Type/Object UUIDs

Types and generally any objects can be globally identified using UUIDs [RFC4122]. UUIDs can be used in WAMP IDL using the `uuid` *Attribute*.

```
/// UUID (canonical textual representation).
my_field1: string (uuid);

/// UUID (128 bit binary).
my_field2: uint128_t (uuid);
```

The `uint128_t` is a struct type defined as

```
/// An unsigned integer with 128 bits.
struct uint128_t {
    /// Least significand 32 bits.
    w0: uint32;

    /// 2nd significand 32 bits.
    w1: uint32;

    /// 3rd significand 32 bits.
    w2: uint32;

    /// Most significand 32 bits.
    w3: uint32;
}
```

Ethereum Addresses

Ethereum addresses can be used to globally identify types or generally any object where the global ID also needs to be conflict free, consensually shared and owned by a respective Ethereum network user. Ethereum addresses can be used in WAMP IDL using the `ethadr` *Attribute*:

```
/// Ethereum address (checksummed HEX encoded address).
my_field1: string (ethadr);

/// Ethereum address (160 bit binary).
my_field2: uint160_t (ethadr);
```

The `uint160_t` is a struct type defined as

```
/// An unsigned integer with 160 bits.
struct uint160_t {
    /// Least significand 32 bits.
    w0: uint32;

    /// 2nd significand 32 bits.
    w1: uint32;

    /// 3rd significand 32 bits.
    w2: uint32;

    /// 4th significand 32 bits.
    w3: uint32;

    /// Most significand 32 bits.
    w4: uint32;
}
```

WAMP Actions or Service Elements

The type of WAMP service element **procedure**, **topic** or **interface** is designated using the *Attribute*

1. `type`: one of `"procedure"`, `"topic"` or `"interface"`

The `type` *Attribute* can be used to denote WAMP service interfaces, e.g. continuing with above WAMP Meta API procedure example, the `wamp.session.kill_by_authid` procedure can be declared like this:

```
rpc_service IWampMeta(type: "interface",
                      uuid: "88711231-3d95-44bc-9464-58d871dd7fd7",
                      wampuri: "wamp")
{
    session_kill_by_authid (SessionKillByAuthid): WampIds (
        type: "procedure",
        wampuri: "wamp.session.kill_by_authid"
    );
}
```

The value of attribute `type` specifies a WAMP *Procedure*, and the call arguments and result types of the procedure are given by:

- `SessionKillByAuthid`: procedure call arguments `args` (positional argument) and `kwargs` (keyword arguments) call argument follow this type
- `WampIds`: procedure call results `args` (positional results) and `kwargs` (keyword results)

The procedure will be registered under the WAMP URI `wamp.session.kill_by_authid` on the respective realm.

15.1.3. WAMP Service Declaration

WAMP services include

- *Procedures* registered by *Callees*, available for calling from *Callers*
- *Topics* published to by *Publishers*, available for subscribing by *Subscribers*

We map the two WAMP service types to FlatBuffers IDL using the *Attribute* `type == "procedure" | "topic"` as in this example:

```
rpc_service IWampMeta(type: "interface",
                      uuid: "88711231-3d95-44bc-9464-58d871dd7fd7",
                      wampuri: "wamp")
{
    session_kill_by_authid (SessionKillByAuthid): WampIds (
        type: "procedure",
        wampuri: "wamp.session.kill_by_authid"
    );

    session_on_leave (SessionInfo): Void (
        type: "topic",
        wampuri: "wamp.session.on_leave"
    );
}
```

When the procedure `wamp.session.kill_by_authid` is called to kill all sessions with a given `authid`, the procedure will return a list of WAMP session IDs of the killed sessions via `WampIds`. Independently, meta events on topic `wamp.session.on_leave` are published with detailed `SessionInfo` of the sessions left as event payload. This follows a common "do-something-and-notify-observers" pattern for a pair of a procedure and topic working together.

The *Interface* then collects a number of *Procedures* and *Topics* under one named unit of `type == "interface"` which includes a UUID in an `uuid` *Attribute*.

Declaring Services

Declaring services involves three element types:

- *Topics*
- *Procedures*

- *Interfaces*

The general form for declaring *Topics* is:

```
<TOPIC-METHOD> (<TOPIC-PAYLOAD-TABLE>): Void (
    type: "topic",
    wampuri: <TOPIC-URI>
);
```

The application payload transmitted in EVENTs is typed via `<TOPIC-PAYLOAD-TABLE>`. The return type MUST always be `Void`, which is a dummy marker type declared in `wamp.fbs`.

> Note: With *Acknowledge Event Delivery* (future), when a *Subscriber* receives an EVENT, the *Subscriber* will return an *Event-Acknowledgement* including `args`/`kwargs`. Once we do have this feature in WAMP PubSub, the type of the *Event-Acknowledgement* can be specified using a non-`Void` return type.

The general form for declaring *Procedures* is:

```
<PROCEDURE-METHOD> (<CALL-PAYLOAD-TABLE>): <CALLRESULT-PAYLOAD-TABLE> (
    type: "procedure",
    wampuri: <PROCEDURE-URI>
);
```

The application payload transmitted in CALLs is typed via `<CALL-PAYLOAD-TABLE>`. The return type of the CALL is typed via `<CALLRESULT-PAYLOAD-TABLE>`.

The general form for declaring *Interfaces*, which collect *Procedures* and *Topics* is:

```
rpc_service <INTERFACE> (
    type: "interface",
    uuid: <INTERFACE-UUID>,
    wampuri: <INTERFACE-URI-PREFIX>
) {
    /// Method declarations of WAMP Procedures and Topics
}
```

> Note: We are reusing FlatBuffers IDL here, specifically the `rpc_service` service definitions which were designed for gRPC. We reuse this element to declare both WAMP *Topics* and *Procedures* by using the `type` Attribute. Do not get confused with "rpc" in `rpc_service`.

Declaring Progressive Call Results

Write me.

Declaring Call Errors

Write me.

15.2. Interface Catalogs

Collections of types defined in FlatBuffers IDL are bundled in *Interface Catalogs* which are just ZIP files with

- one catalog.yaml file with catalog metadata
- one or more `*.bfbs` compiled FlatBuffer IDL schemas

and optionally

- schema source files
- image and documentation files

15.2.1. Catalog Archive File

The contents of an `example.zip` interface catalog:

```
unzip -l build/example.zip
Archive:  build/example.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
        0  1980-00-00 00:00   schema/
    14992  1980-00-00 00:00   schema/example2.bfbs
    15088  1980-00-00 00:00   schema/example4.bfbs
    13360  1980-00-00 00:00   schema/example3.bfbs
     8932  1980-00-00 00:00   schema/example1.bfbs
     6520  1980-00-00 00:00   schema/wamp.bfbs
     1564  1980-00-00 00:00   README.md
        0  1980-00-00 00:00   img/
    13895  1980-00-00 00:00   img/logo.png
     1070  1980-00-00 00:00   LICENSE.txt
     1288  1980-00-00 00:00   catalog.yaml
---------                     -------
    76709                     11 files
```

The bundled Catalog Interfaces in above are FlatBuffers binary schema files which are compiled using `flatc`

```
flatc -o ./schema --binary --schema --bfbs-comments --bfbs-builtins ./src
```

from FlatBuffers IDL sources, for example:

```
rpc_service IExample1 (
    type: "interface", uuid: "bf469db0-efea-425b-8de4-24b5770e6241"
) {
    my_procedure1 (TestRequest1): TestResponse1 (
        type: "procedure", wampuri: "com.example.my_procedure1"
    );

    on_something1 (TestEvent1): Void (
        type: "topic", wampuri: "com.example.on_something1"
    );
}
```

15.2.2. Catalog Metadata

The `catalog.yaml` file contains catalog metadata in YAML Format:

Field	Description
`name`	Catalog name, which must contain only lower-case letter, numbers, hyphen and underscore so the catalog name can be used in HTTP URLs
`version`	Catalog version (e.g. semver or calendarver version string)
`title`	Catalog title for display purposes
`description`	Catalog description, a short text describing the API catalog
`schemas`	FlatBuffers schemas compiled into binary schema reflection format
`author`	Catalog author
`publisher`	Ethereum Mainnet address of publisher
`license`	SPDX license identifier (see https://spdx.org/licenses/) for the catalog
`keywords`	Catalog keywords to hint at the contents, topic, usage or similar of the catalog
`homepage`	Catalog home page or project page
`git`	Git source repository location
`theme`	Catalog visual theme

Table 11

Here is a complete example:

```
name: example

version: 22.6.1

title: WAMP Example API Catalog

description: An example of a WAMP API catalog.

schemas:
  - schema/example1.bfbs
  - schema/example2.bfbs
  - schema/example3.bfbs
  - schema/example4.bfbs

author: typedef int GmbH

publisher: "0x60CC48BFC44b48A53e793FE4cB50e2d625BABB27"

license: MIT

keywords:
  - wamp
  - sample

homepage: https://wamp-proto.org/

git: https://github.com/wamp-proto/wamp-proto.git

theme:
  background: "#333333"
  text: "#e0e0e0"
  highlight: "#00ccff"
  logo: img/logo.png
```

15.2.3. Catalog Sharing and Publication

Archive File Preparation

The ZIP archive format and tools, by default, include filesystem and other metadata from the host producing the archive. That information usually changes, per-archive run, as e.g. the current datetime is included, which obviously progresses.

When sharing and publishing a WAMP Interface Catalog, it is crucial that the archive only depends on the actual contents of the compressed files.

Removing all unwanted ZIP archive metadata can be achieved using stripzip:

```
stripzip example.zip
```

The user build scripts for compiling and bundling an Interface Catalog ZIP file MUST be repeatable, and only depend on the input source files. A build process that fulfills this requirement is called Reproducible build.

The easiest way to check if your build scripts producing `example.zip` is reproducible is repeat the build and check that the file fingerprint of the resulting archive stays the same:

```
openssl sha256 example.zip
```

Catalog Publication on Ethereum and IPFS

Write me.

15.3. Interface Reflection

Feature status: **sketch**

Reflection denotes the ability of WAMP peers to examine the procedures, topics and errors provided or used by other peers.

I.e. a WAMP *Caller*, *Callee*, *Subscriber* or *Publisher* may be interested in retrieving a machine readable list and description of WAMP procedures and topics it is authorized to access or provide in the context of a WAMP session with a *Dealer* or *Broker*.

Reflection may be useful in the following cases:

- documentation
- discoverability
- generating stubs and proxies

WAMP predefines the following procedures for performing run-time reflection on WAMP peers which act as *Brokers* and/or *Dealers*.

Predefined WAMP reflection procedures to *list* resources by type:

```
    wamp.reflection.topic.list
    wamp.reflection.procedure.list
    wamp.reflection.error.list
```

Predefined WAMP reflection procedures to *describe* resources by type:

```
    wamp.reflection.topic.describe
    wamp.reflection.procedure.describe
    wamp.reflection.error.describe
```

A peer that acts as a *Broker* SHOULD announce support for the reflection API by sending

```
    HELLO.Details.roles.broker.reflection|bool := true
```

A peer that acts as a *Dealer* SHOULD announce support for the reflection API by sending

```
    HELLO.Details.roles.dealer.reflection|bool := true
```

> Since *Brokers* might provide (broker) procedures and *Dealers* might provide (dealer) topics, both SHOULD implement the complete API above (even if the peer only implements one of *Broker* or *Dealer* roles).

Reflection Events and Procedures

A topic or procedure is defined for reflection:

```
wamp.reflect.define
```

A topic or procedure is asked to be described (reflected upon):

```
wamp.reflect.describe
```

A topic or procedure has been defined for reflection:

```
wamp.reflect.on_define
```

A topic or procedure has been undefined from reflection:

```
wamp.reflect.on_undefine
```

16. Router-to-Router Links

Write me.

1. Resolve global realm name R_name via ENS to the on-chain address R_adr of the realm.
2. Retrieve list of Domains R_DR routing realm R_adr.
3. Retrieve the node's N1 own domain D_N1 given the node's address N1_adr.
4. Check D_N1 is in R_DR.
5. Select a domain D from R_DR and get endpoint E for D.
6. Connect to D and authenticate via WAMP-Cryptosign.
7. Verify connected node N2 by checking against D
8. Subscribe to `wamp.r2r.traffic_payable`
9. When receiving a traffic payable event, buy the respective key by calling `xbr.pool.buy_key`, and calling `wamp.r2r.submit_traffic_payment`, which returns a traffic usage report.

Data Spaces are end-to-end encrypted routing realms connecting data driven microservices.

The message routing between the microservice endpoints in

17. Advanced Profile URIs

WAMP pre-defines the following error URIs for the **Advanced Profile**. WAMP peers SHOULD only use the defined error messages.

A *Dealer* or (U+00A0)*Callee* canceled a call previously issued

```
wamp.error.canceled
```

A *Peer* requested an interaction with an option that was disallowed by the *Router*

```
wamp.error.option_not_allowed
```

A *Router* rejected client request to disclose its identity

```
wamp.error.option_disallowed.disclose_me
```

A *Router* encountered a network failure

```
wamp.error.network_failure
```

A *Callee* is not able to handle an invocation for a *call* and intends for the *Router* to re-route the *call* to another fitting *Callee*. For details, refer to RPC Call Rerouting

```
wamp.error.unavailable
```

A *Dealer* could not perform a call, since a procedure with the given URI is registered, but all available registrations have responded with `wamp.error.unavailable`

```
wamp.error.no_available_callee
```

18. IANA Considerations

WAMP uses the Subprotocol Identifier `wamp` registered with the WebSocket Subprotocol Name Registry, operated by the Internet Assigned Numbers Authority (IANA).

19. Conformance Requirements

All diagrams, examples, and notes in this specification are non-normative, as are all sections explicitly marked non-normative. Everything else in this specification is normative.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

Requirements phrased in the imperative as part of algorithms (such as "strip any leading space characters" or "return false and abort these steps") are to be interpreted with the meaning of the key word ("MUST", "SHOULD", "MAY", etc.) used in introducing the algorithm.

Conformance requirements phrased as algorithms or specific steps MAY be implemented in any manner, so long as the end result is equivalent.

19.1. Terminology and Other Conventions

Key terms such as named algorithms or definitions are indicated like *this* when they first occur, and are capitalized throughout the text.

20. Normative References

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <https://www.rfc-editor.org/info/rfc3629>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, DOI 10.17487/RFC4122, July 2005, <https://www.rfc-editor.org/info/rfc4122>.

[RFC6455] Fette, I. and A. Melnikov, "The WebSocket Protocol", RFC 6455, DOI 10.17487/RFC6455, December 2011, <https://www.rfc-editor.org/info/rfc6455>.

[RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <https://www.rfc-editor.org/info/rfc7159>.

[RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <https://www.rfc-editor.org/info/rfc8949>.

21. Informative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

Index

Author's Address

Tobias Oberstein
typedef int GmbH
Email: tobias.oberstein@typedefint.eu