Workgroup:	BiDirectional or Server-Initiated HTTP
Internet-Draft:	WAMP
Published:	9 February 2023
Intended Status:	Experimental
Expires:	13 August 2023
Author:	T. Oberstein *typedef int GmbH*

WAMP Basic Profile

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 13 August 2023.

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*) {#session_scope_id}

> 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 {#session_scope_ids}

- SUBSCRIBE.Request
- SUBSCRIBED.Request (mirrored SUBSCRIBE.Request)
- UNSUBSCRIBE.Request
- UNSUBSCRIBED.Request (mirrored UNSUBSCRIBE.Request)
- PUBLISH.Request
- PUBLISHED.Request (mirrored PUBLISH.Request)
- REGISTER.Request
- REGISTERED.Request (mirrored REGISTER.Request)
- UNREGISTER.Request
- UNREGISTERED.Request (mirrored UNREGISTER.Request)
- CALL.Request
- RESULT.Request (mirrored CALL.Request)
- CANCEL.Request (mirrored CALL.Request)
- INVOCATION.Request
- YIELD.Request (mirrored INVOCATION.Request)
- INTERRUPT.Request (mirrored INVOCATION.Request)
- ERROR.Request (mirrored original request ID)

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.

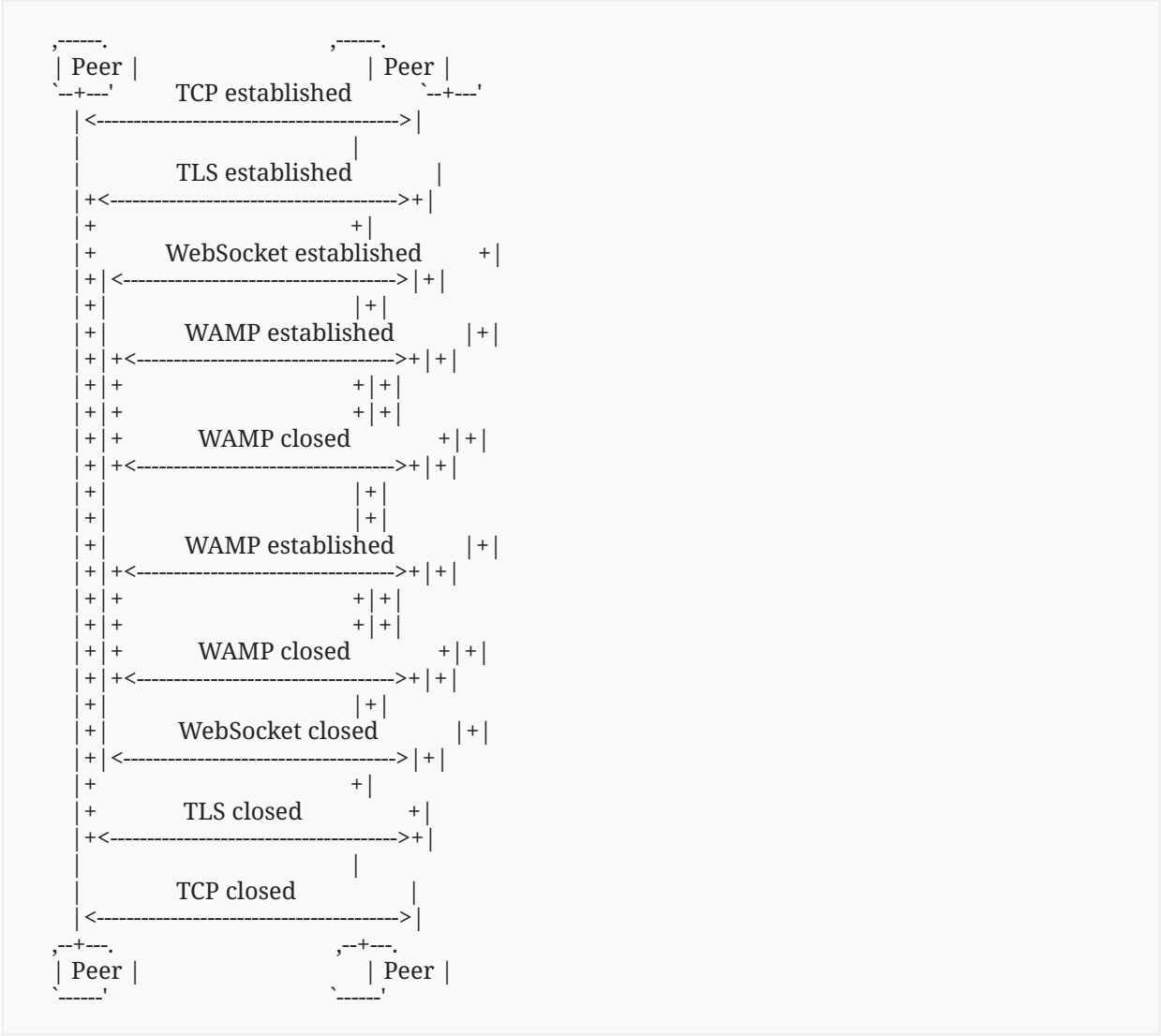
With `wamp.2.cbor`, *all* WebSocket messages MUST BE of type **binary** and use the CBOR message serialization.

> 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 abort 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 message with non-sequential session scope request ID, such as SUBSCRIBE, UNSUBSCRIBE, PUBLISH, REGISTER, UNREGISTER, CALL and YIELD.
- 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 abort 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|ids 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]
```

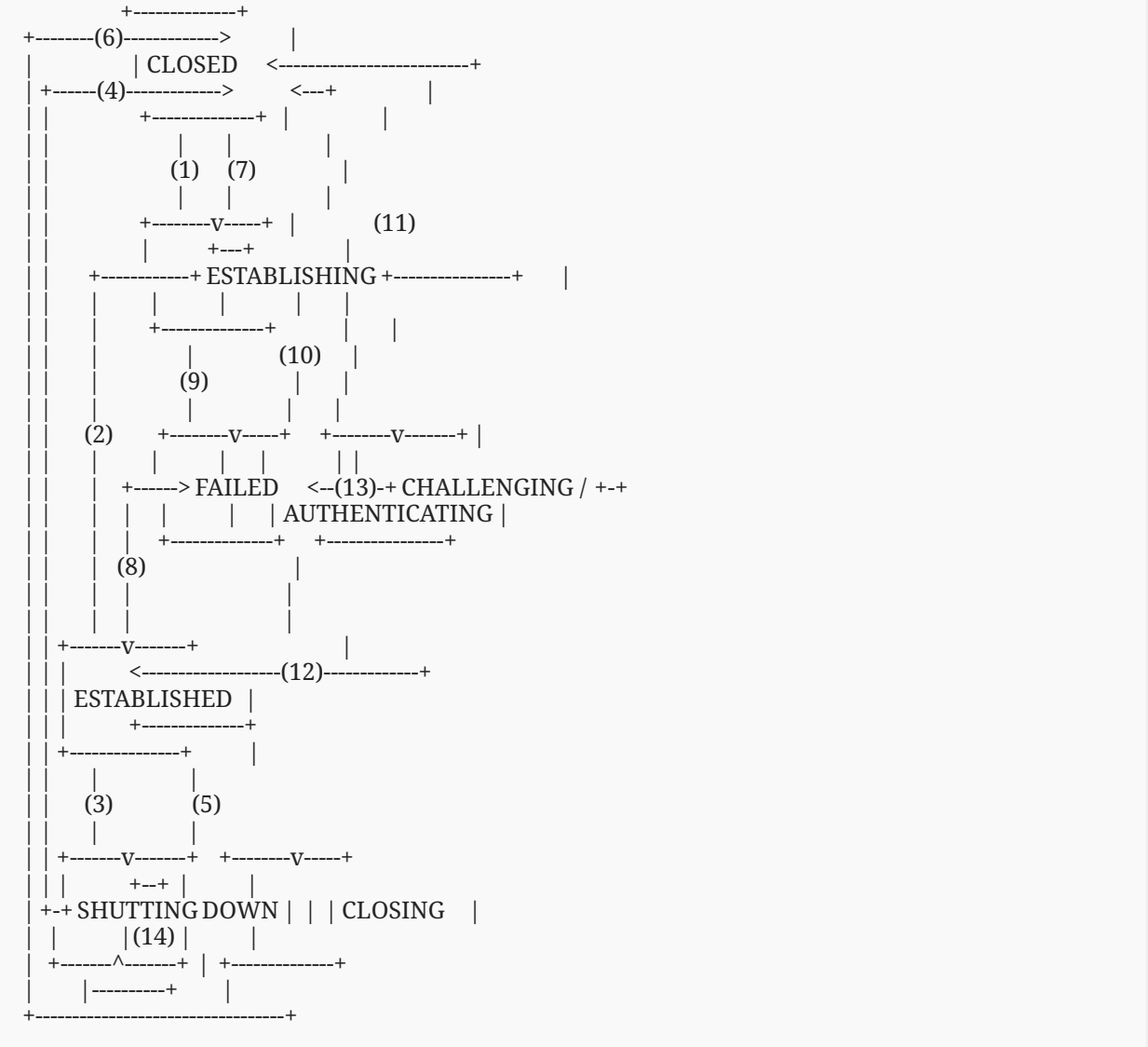
SHOULD be sent when `Arguments` is non-empty.

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

#	State
4	Received GOODBYE
5	Received GOODBYE
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

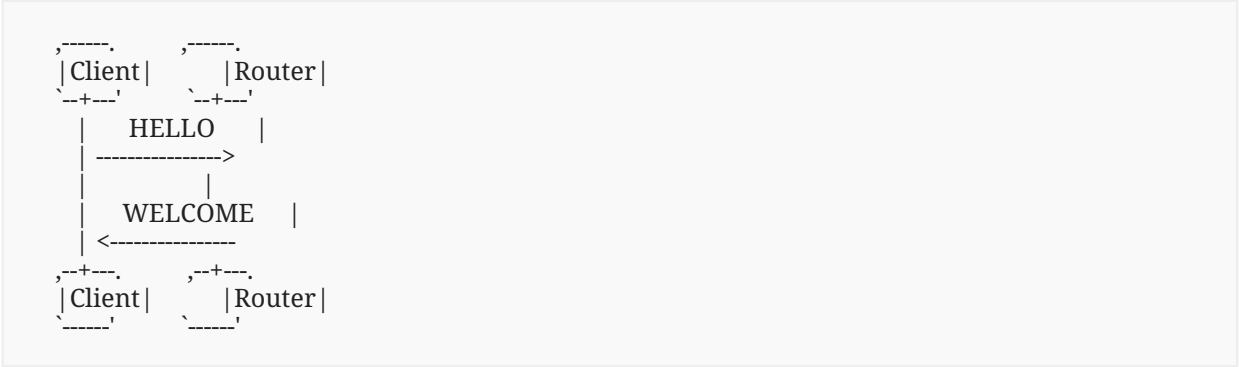
```
[HELLO, Realm|uri, Details|dict]
```

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]
```

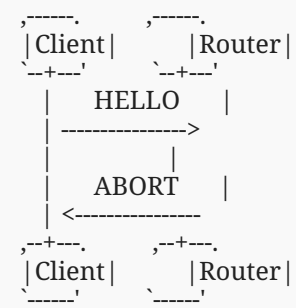
where

- 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

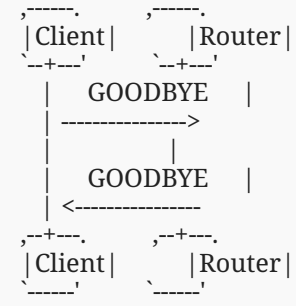
4.2.1. GOODBYE

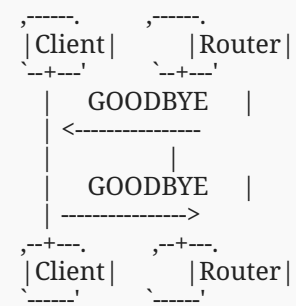
A WAMP session starts its lifetime with the Router sending a `WELCOME` message to the Client and ends when the underlying transport disappears or when the WAMP session is closed explicitly by a `GOODBYE` message sent by one Peer and a `GOODBYE` message sent from the other Peer in response.

```
[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.

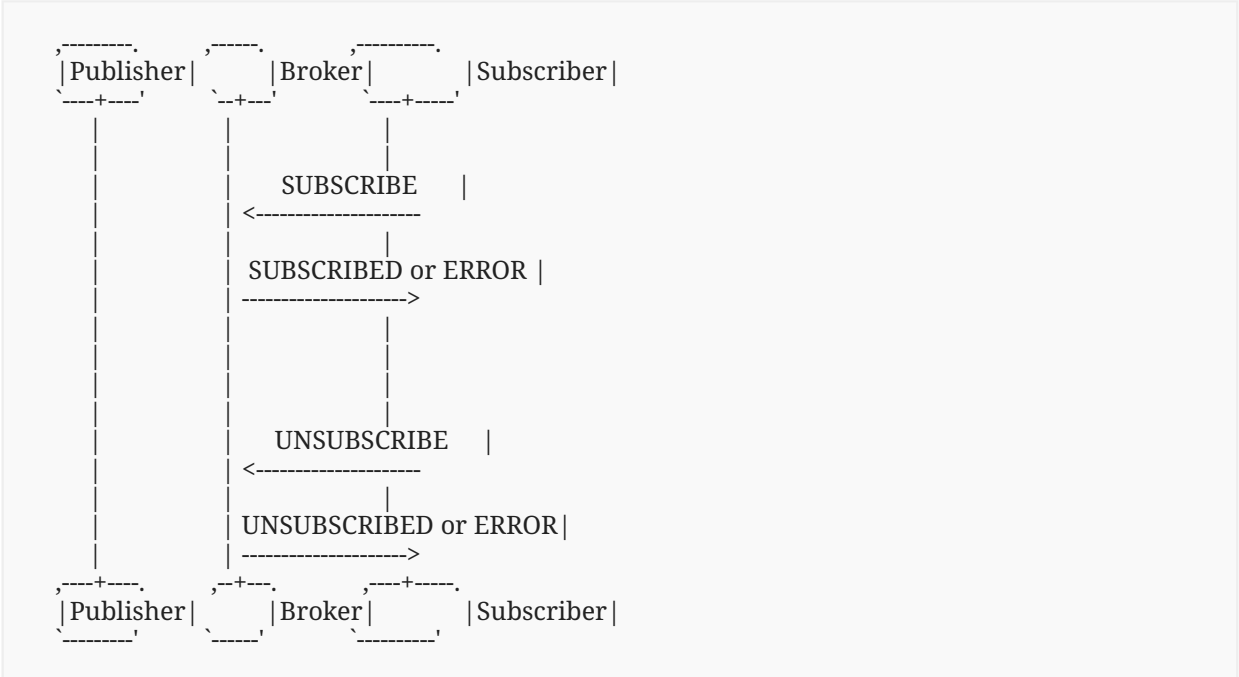
5. Publish and Subscribe

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 is a sequential ID in the *session scope*, incremented by the Subscriber and used to correlate the Broker's response with the request.
- Options is 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 is 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 is the ID from the original subscription request.
- Subscription is 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` 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, 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` is a sequential ID in the *session scope*, incremented by the Subscriber and used to correlate the Broker's response with the request.
- `SUBSCRIBED.Subscription` is 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 is 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 is the ID from the original request.
- Error is a URI that gives the error of why the request could not be fulfilled.

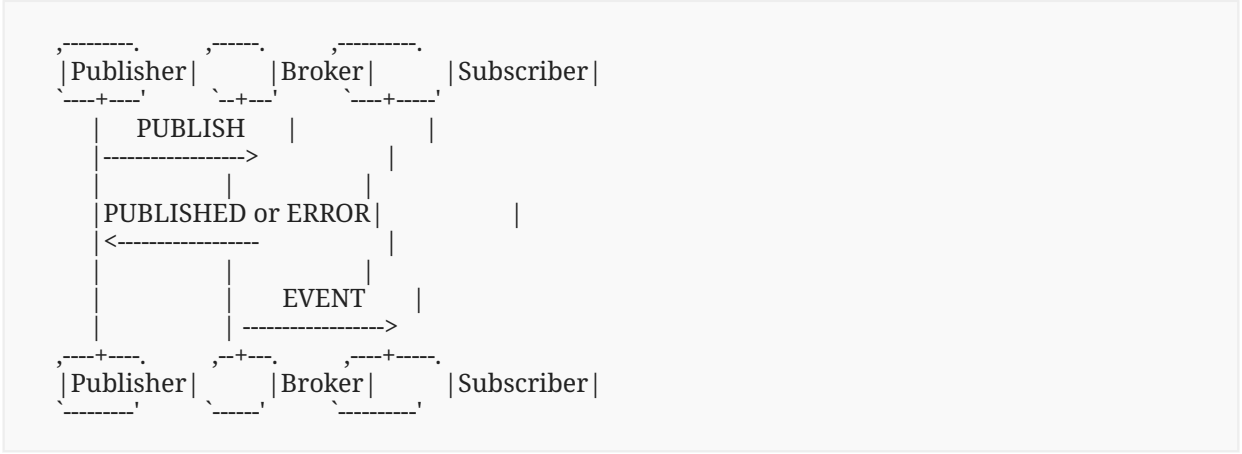
Example

```
[8, 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 an 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.ArgumentsKw|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.ArgumentsKw 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]}]
```

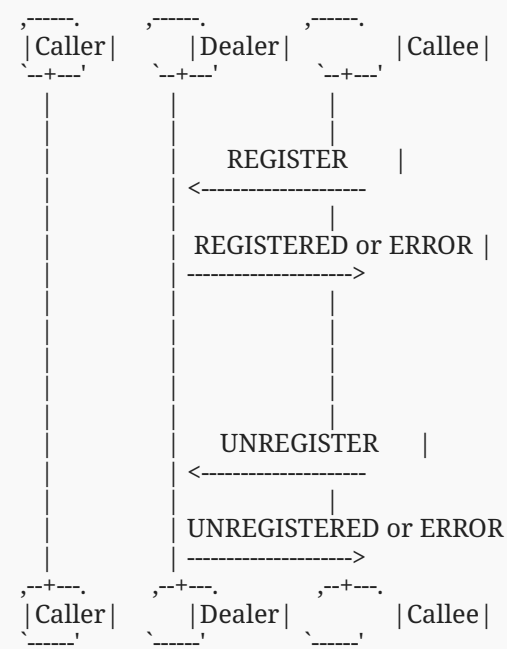
6. Remote Procedure Calls

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.
- Procedureis 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.

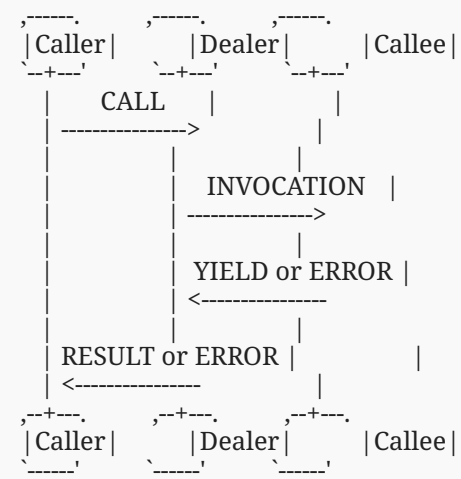
Example

```
[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.
- Optionsis 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. IANA Considerations

WAMP uses the Subprotocol Identifier `wamp` registered with the WebSocket Subprotocol Name Registry, operated by the Internet Assigned Numbers Authority (IANA).

10. 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.

10.1. 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.

11. Contributors

WAMP was developed in an open process from the beginning, and a lot of people have contributed ideas and other feedback. Here we are listing people who have opted in to being mentioned:

- Alexander Goedde
- Amber Brown
- Andrew Gillis
- David Chappelle
- Elvis Stansvik
- Emile Cormier
- Felipe Gasper
- Johan 't Hart
- Josh Soref
- Konstantin Burkalev
- Pahaz Blinov
- Paolo Angioletti
- Roberto Requena
- Roger Erens
- Christoph Herzog

- Tobias Oberstein
- Zhigang Wang

12. 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>.

[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>.

13. 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>.

Author's Address

Tobias Oberstein
typedef int GmbH
Email: tobias.oberstein@typedefint.eu