T. Oberstein
A. Goedde
Tavendo GmbH

October 11, 2015

# The Web Application Messaging Protocol
## draft-oberstet-hybi-tavendo-wamp-02

Abstract

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.

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 http://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 April 13, 2016.

Table of Contents

# 1. Introduction

## 1.1. Background

_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.

## 1.2. Protocol Overview

_This section is non-normative._

The PubSub messaging pattern defines three roles: _Subscribers_ and _Publishers_, which communicate via a _Broker_.

The routed RPC messaging pattern also defines three roles: _Callers_ and _Callees_, which communicate via a _Dealer_.

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.

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 which is message-oriented, 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. Design Philosophy

_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.

### 1.3.1. Basic and Advanced Profiles

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.

### 1.3.2. 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.

### 1.3.3. 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.

### 1.3.4. Router Implementation Specifics

This specification only deals with the protcol 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. Relationship to WebSocket

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

## 2. 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.

### 2.1. 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.

## 3. 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 [1] subprotocol.

# 4. 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.

## 4.1. 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.

## 4.2. Remote Procedure Call Roles

The Remote Procedure Call 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.

### 4.3. Publish & Subscribe Roles

The Publish & Subscribe messaging pattern involves peers of three different roles:

- Subscriber (Client)

- Publisher (Client)

- Broker (Router)

A Publishers 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.

### 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.

# 5. Building Blocks

WAMP is defined with respect to the following building blocks

1. Identifiers
2. Serializations
3. Transports

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

## 5.1. Identifiers

### 5.1.1. 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 - 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.

#### 5.1.1.1. 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 above rules:

```
<CODE BEGINS>
    ## loose URI check disallowing empty URI components
    pattern = re.compile(r"^([^\s\.#]+\.)*([^\s\.#]+)$")
<CODE ENDS>
```

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):

```
<CODE BEGINS>
    ## loose URI check allowing empty URI components
    pattern = re.compile(r"^(([^\s\.#]+\.)|\.)*([^\s\.#]+)?$")
<CODE ENDS>
```

#### 5.1.1.2. Strict URIs

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

As an example, the following regular expression could be used in Python to check URIs according to the above rules:

```
<CODE BEGINS>
    ## strict URI check disallowing empty URI components
    pattern = re.compile(r"^([0-9a-z_]+\.)*([0-9a-z_]+)$")
<CODE ENDS>
```

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):

```
<CODE BEGINS>
    ## strict URI check allowing empty URI components
    pattern = re.compile(r"^(([0-9a-z_]+\.)|\.)*([0-9a-z_]+)?$")
<CODE ENDS>
```

Following the suggested regular expression 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.

#### 5.1.1.3. 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"

### 5.1.2. 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) *0* and *2^53* (9007199254740992):

- IDs in the _global scope_ MUST be drawn _randomly_ from a _uniform distribution_ over the complete range [0, 2^53]

- IDs in the _router scope_ can be chosen freely by the specific router implementation

- IDs in the _session scope_ SHOULD be incremented by 1 beginning with 1 (for each direction - _Client-to-Router_ and _Router-to-Client_)

  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 Section 6.

#### 5.1.2.1. Global Scope IDs

- "WELCOME.Session"
- "PUBLISHED.Publication"
- "EVENT.Publication"

#### 5.1.2.2. Router Scope IDs

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

#### 5.1.2.3. 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"

## 5.2. Serializations

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.

WAMP defines two bindings for message _serialization_:

1. JSON
2. MsgPack

Other bindings for _serialization_ may be defined in future WAMP versions.

#### 5.2.1. JSON

With JSON serialization, each WAMP message is serialized according to the JSON specification as described in RFC4627.

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

#### 5.2.2. MsgPack

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

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

### 5.3. 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.

#### 5.3.1. WebSocket Transport

The default transport binding for WAMP is WebSocket.

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.

WAMP uses the following WebSocket subprotocol identifiers for unbatched modes:

- "wamp.2.json"
- "wamp.2.msgpack"

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

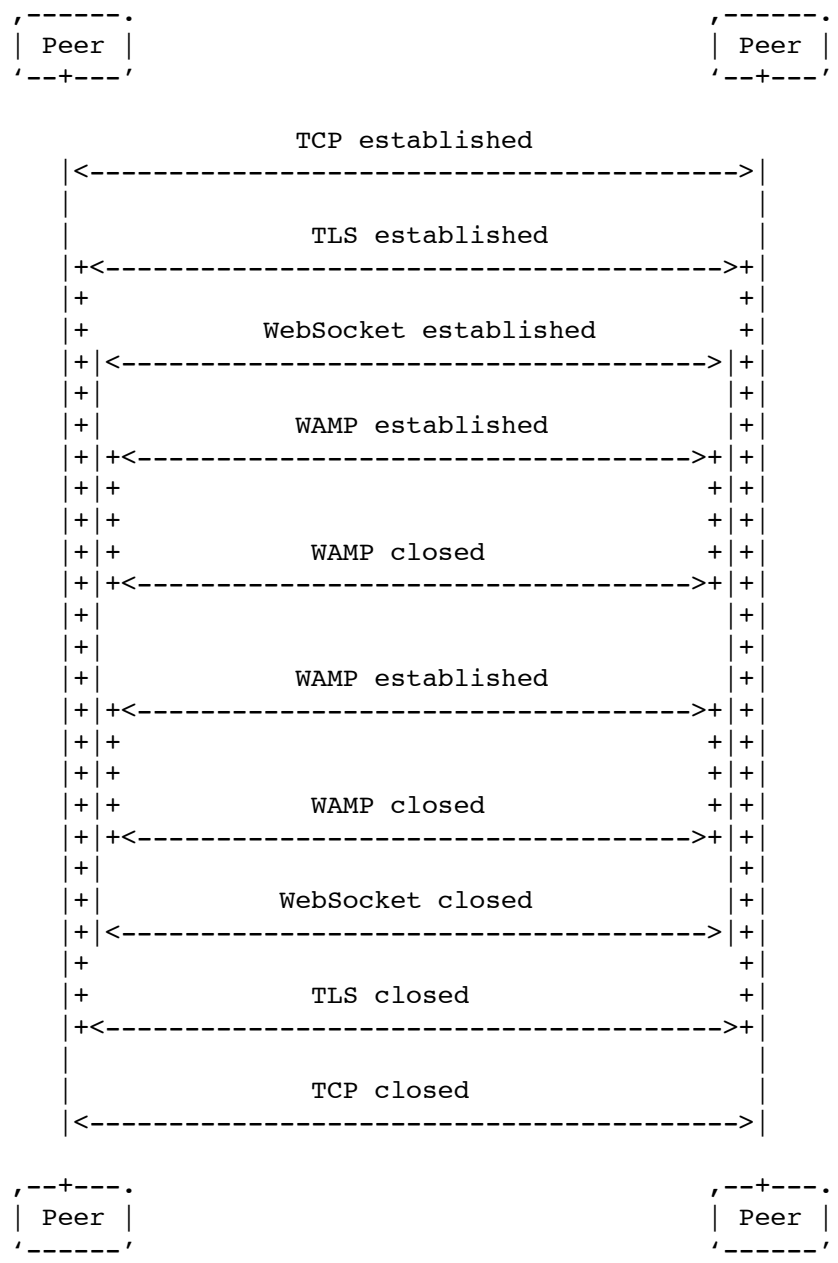
With "wamp.2.msgpack", _all_ WebSocket messages MUST BE of type *binary* and use the MsgPack message serialization.

> 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.

### 5.3.2. 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.




## 6. 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 Section 5.1.1
- "id": an integer ID as defined in Section 5.1.2
- "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"]
```

## 6.1. 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.

## 6.2. 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.

### 6.3. 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.

### 6.4. 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.

#### 6.4.1. Session Lifecycle

##### 6.4.1.1. HELLO

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

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

##### 6.4.1.2. WELCOME

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

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

##### 6.4.1.3. ABORT

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

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

#### 6.4.1.4. 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]
```

#### 6.4.1.5. 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]
```

### 6.4.2. Publish & Subscribe

#### 6.4.2.1. 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]
```

#### 6.4.2.2. PUBLISHED

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

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

#### 6.4.2.3. SUBSCRIBE

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

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

#### 6.4.2.4. SUBSCRIBED

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

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

#### 6.4.2.5. UNSUBSCRIBE

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

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

#### 6.4.2.6. UNSUBSCRIBED

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

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

#### 6.4.2.7. 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.

### 6.4.3. Routed Remote Procedure Calls

#### 6.4.3.1. 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]
```

#### 6.4.3.2. 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]
```

#### 6.4.3.3. REGISTER

A _Callees_ request to register an endpoint at a _Dealer_.

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

#### 6.4.3.4. REGISTERED

Acknowledge sent by a _Dealer_ to a _Callee_ for successful registration.

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

#### 6.4.3.5. UNREGISTER

A _Callees_ request to unregister a previously established registration.

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

#### 6.4.3.6. UNREGISTERED

Acknowledge sent by a _Dealer_ to a _Callee_ for successful unregistration.

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

#### 6.4.3.7. 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, C* Arguments|list]

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

#### 6.4.3.8. 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]
```

### 6.5. Message Codes and Direction

The following table lists the message type code for *all 25 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 |

## 6.6. 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.

## 6.7. 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.

# 7. 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"

## 7.1. Session Establishment

### 7.1.1. 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]
```

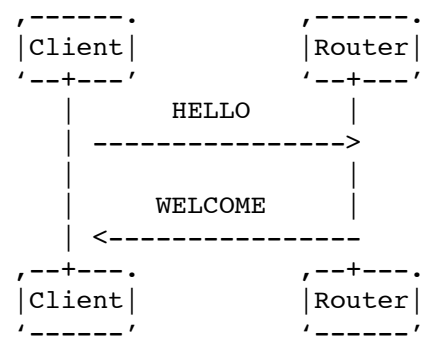
where

- "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 fail the session if that happens.

#### 7.1.1.1. 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": {}
  }
}]
```

### 7.1.2. 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.

#### 7.1.2.1. 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": {}
   }
}]
```

### 7.1.3. ABORT

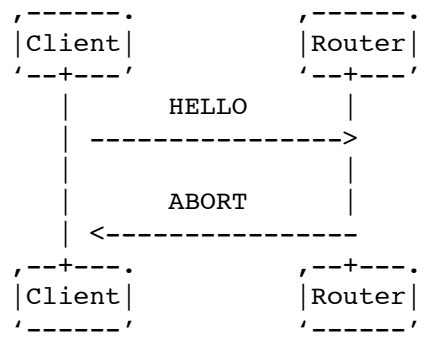
Both the _Router_ and the _Client_ may abort the opening of a WAMP session by sending an "ABORT" message.

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

where

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

No response to an "ABORT" message is expected.




_Example_

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

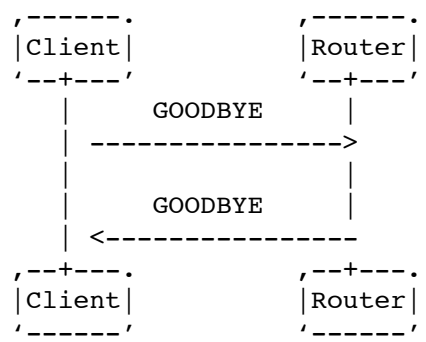
## 7.2. Session Closing

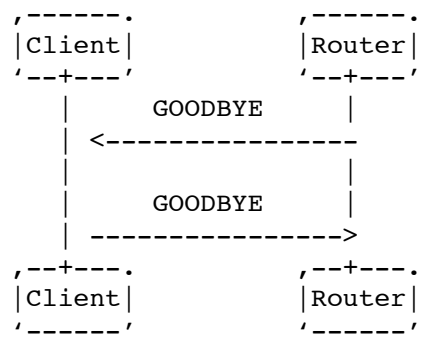
A WAMP session starts its lifetime with the _Router_ sending a "WELCOME" message to the _Client_ and ends when the underlying transport disappears or when the WAMP session is closed explicitly by a "GOODBYE" message sent by one _Peer_ and a "GOODBYE" message sent from the other _Peer_ in response.

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

where

- "Reason" MUST be an 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.error.system_shutdown"]
```

and the other peer replies

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

_Example_. One _Peer_ initiates closing

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

and the other peer replies

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

### 7.2.1. Difference between ABORT and GOODBYE

The differences between "ABORT" and "GOODBYE" messages are:

1. "ABORT" gets sent only _before_ a _Session_ is established, while "GOODBYE" is sent only _after_ a _Session_ is already established.

2. "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.

### 7.3. 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": {}
    }
}]
```

## 8. 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 _Dealer_.

### 8.1. 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:

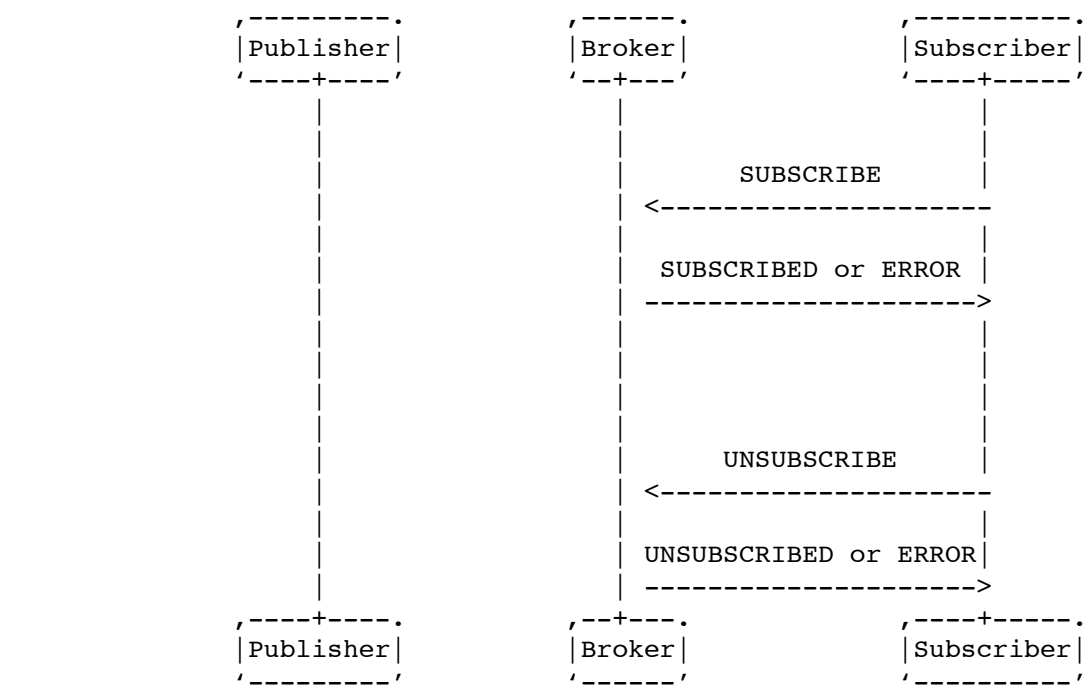
1. "SUBSCRIBE"

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

#### 8.1.1. 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 an 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.

#### 8.1.2. 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".

#### 8.1.3. 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 an URI that gives the error of why the request could not be fulfilled.

_Example_

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

#### 8.1.4. 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]
```

#### 8.1.5. 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]
```

#### 8.1.6. 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 an URI that gives the error of why the request could not be fulfilled.

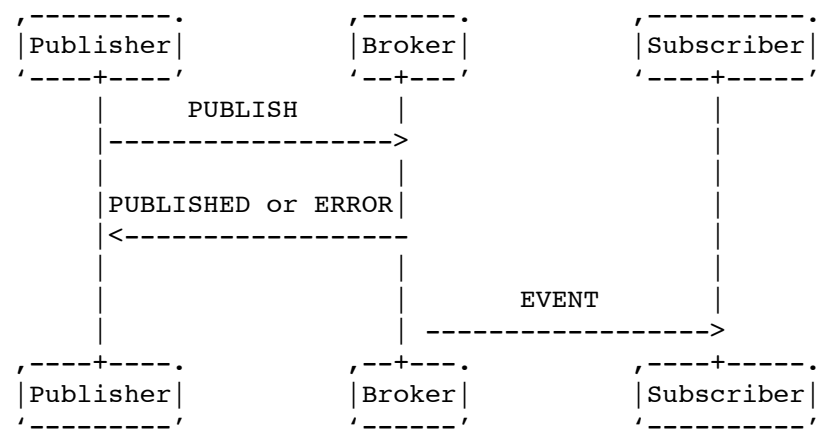
_Example_

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

## 8.2. 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"




### 8.2.1. 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 random, ephemeral ID chosen 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_ is able to fulfill and allowing 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]}]
```

### 8.2.2. 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]
```

#### 8.2.3. 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 an URI that gives the error of why the request could not be fulfilled.

_Example_

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

#### 8.2.4. 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 _Subscriber_.
- "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]}]
```

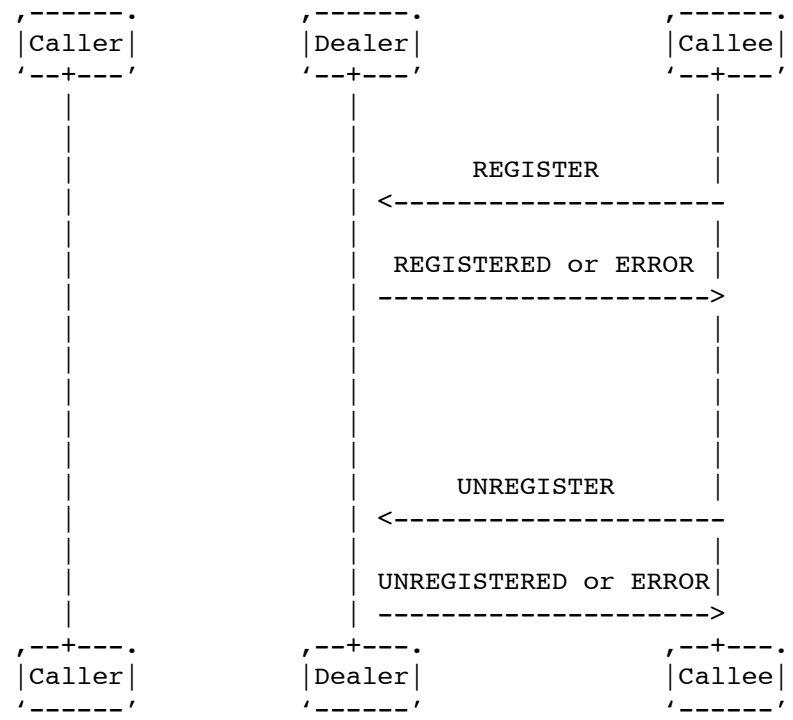
## 9. Remote Procedure Calls

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

### 9.1. 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"




#### 9.1.1. 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 random, ephemeral ID chosen 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"]
```

#### 9.1.2. 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]
```

#### 9.1.3. 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 an URI that gives the error of why the request could not be fulfilled.

_Example_

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

#### 9.1.4. 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 random, ephemeral ID chosen 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]
```

#### 9.1.5. 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]
```

#### 9.1.6. 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 an URI that gives the error of why the request could not be fulfilled.

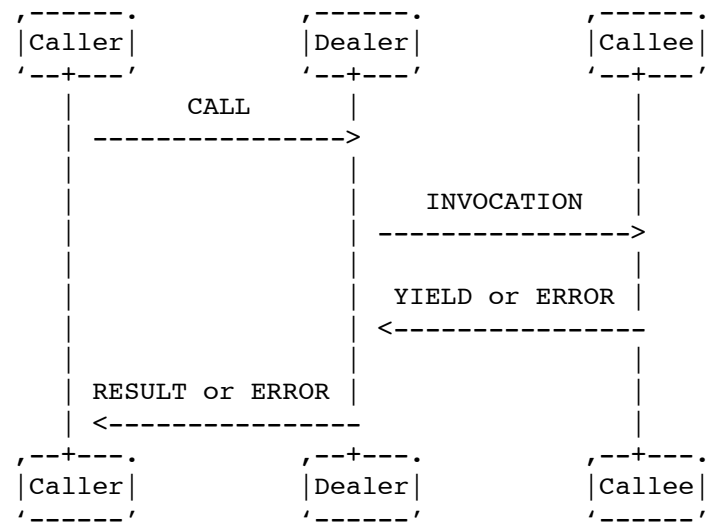
_Example_

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

## 9.2. 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_.

#### 9.2.1. 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 random, ephemeral ID chosen 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"}]
```

#### 9.2.2. 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 random, ephemeral ID chosen 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"}]
```

### 9.2.3. 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}]
```

#### 9.2.4. 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}]
```

#### 9.2.5. 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 an 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}]
```

#### 9.2.6. 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 an 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"]
```

# 10. Predefined URIs

WAMP pre-defines the following error URIs for the basic and for the advanced profile. WAMP peers MUST use only the defined error messages.

## 10.1. Basic Profile

### 10.1.1. 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

### 10.1.2. 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

#### 10.1.3. Session Close

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

wamp.error.system_shutdown

The _Peer_ want to leave the realm - used as a "GOODBYE" reason.

wamp.error.close_realm

A _Peer_ acknowledges ending of a session - used as a "GOODBYE" reply reason.

wamp.error.goodbye_and_out

#### 10.1.4. 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

### 10.2. Advanced Profile

uri_Dealer_ or _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 _Dealer_ could not perform a call, since a procedure with the given URI is registered, but _Callee Black- and Whitelisting_ and/or _Caller Exclusion_ lead to the exclusion of (any) _Callee_ providing the procedure.

wamp.error.no_eligible_callee

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

## 11. Ordering Guarantees

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.

### 11.1. 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.

### 11.2. 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.

## 12. 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.

### 12.1. 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).

### 12.2. 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.

### 12.3. 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_.

#### 12.3.1. 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.

## 13. Advanced Profile

While 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 is 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.

### 13.1. Messages

The Advanced Profile defines the following additional messages which are explained in detail in separate sections.

#### 13.1.1. Message Definitions

The following 4 additional message types MAY be used in the Advanced Profile.

##### 13.1.1.1. 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]
```

##### 13.1.1.2. 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]
```

##### 13.1.1.3. 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]
```

##### 13.1.1.4. 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]
```

#### 13.1.2. Message Codes and Direction

The following table list the message type code for *the OPTIONAL messages* defined in this part of the document and their direction between peer roles.

| Cod | Message | Pub | Brk | Subs | Calr | Dealr | 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 |

"Tx" ("Rx") means the message is sent (received) by a peer of the respective role.

## 13.2. Features

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.

| 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, interoperatble implementations. |

### 13.2.1. 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 |
| procedure_reflection | sketch | | | | | X | |

### 13.2.2. 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 | | | |
| session_meta_api | beta | | X | | | | |
| subscription_meta_api | beta | | X | | | | |
| pattern_based_subscription | beta | | X | X | | | |
| sharded_subscription | alpha | | X | X | | | |
| event_history | alpha | | X | X | | | |
| topic_reflection | sketch | | X | | | | |

### 13.2.3. Other Advanced Features

| Feature | Status |
|---|---|
| challenge-response authentication | beta |
| cookie authentication | beta |
| ticket authentication | beta |
| rawsocket transport | stable |
| batched WS transport | sketch |
| longpoll transport | beta |
| session meta api | beta |

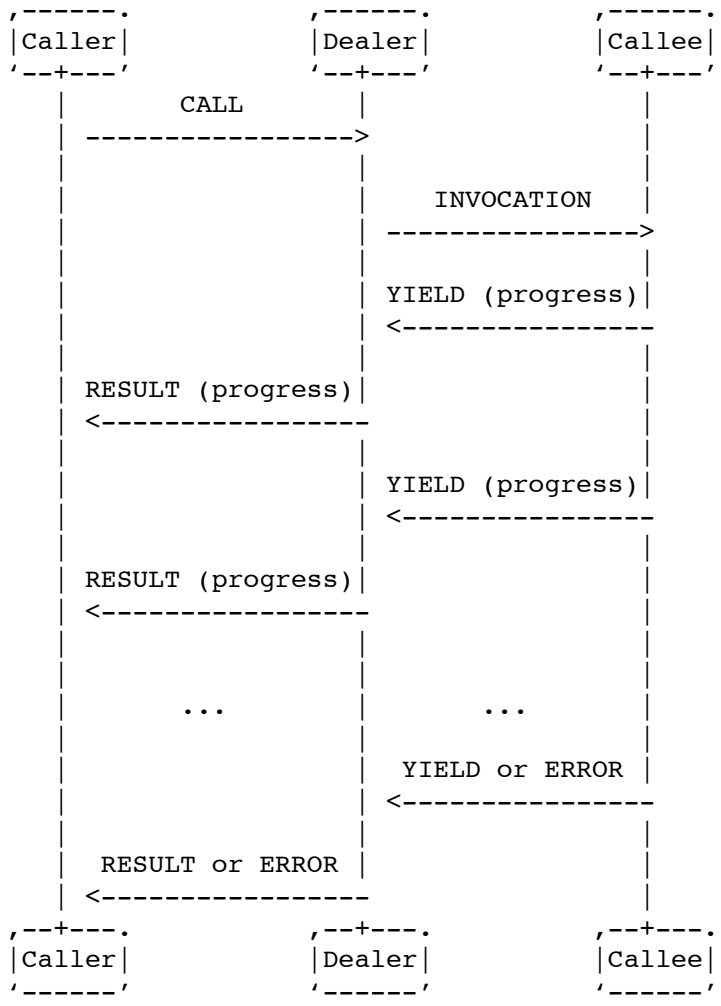
## 13.3. Advanced RPC Features

### 13.3.1. Progressive Call Results

#### 13.3.1.1. Feature Definition

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.

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.Options.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 "YIELDs"

```
    [
        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 "RESULTs"

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

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

...

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 "YIELDs"

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

_Example._ Dealer-to-Caller "RESULTs"

```
[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 "YIELDs"

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

_Example._ Dealer-to-Caller "RESULTs"

```
[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 "YIELDs"

```
[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 "RESULTs"

```
[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 it's 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.Options.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.

#### 13.3.1.2. 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
```

### 13.3.2. Progressive Calls

#### 13.3.2.1. Feature Definition

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>

### 13.3.3. Call Timeouts

#### 13.3.3.1. Feature Definition

A _Caller_ might want to issue a call providing a _timeout_ for the call to finish.

A _timeout_ allows to *automatically* cancel 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_

#### 13.3.3.2. 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

### 13.3.4. Call Canceling

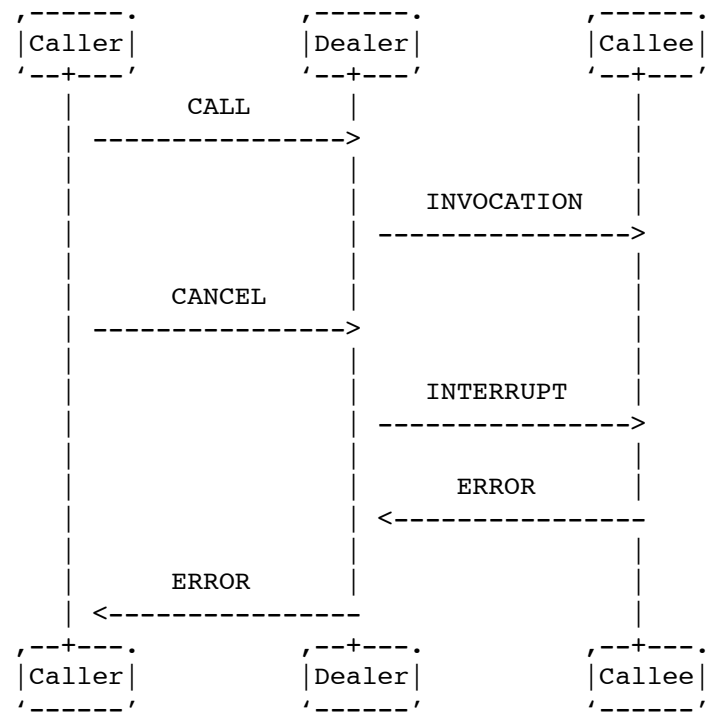
#### 13.3.4.1. Feature Definition

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.

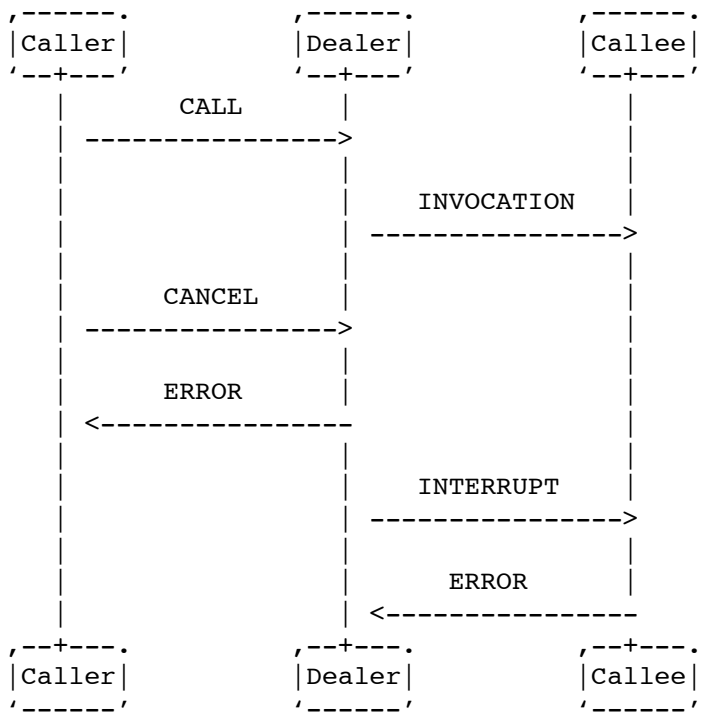
The message flow between _Callers_, a _Dealer_ and _Callees_ for canceling remote procedure calls involves the following messages:

- "CANCEL"
- "INTERRUPT"

A call may be cancelled at the _Callee_




A call may be cancelled at the _Dealer_




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"
```

#### 13.3.4.2. 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
```

### 13.3.5. Caller Identification

#### 13.3.5.1. Feature Definition

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.

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.

#### 13.3.5.2. 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
```

### 13.3.6. Call Trust Levels

#### 13.3.6.1. Feature Defintion

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".

#### 13.3.6.2. 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
```

### 13.3.7. Registration Meta API

#### 13.3.7.1. Feature Definition

#### 13.3.7.1.1. Introduction

*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.

#### 13.3.7.1.2. Registration Meta 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 an 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.

#### 13.3.7.1.2.1. wamp.registration.on_create

Fired when a registration is created through a registration request for an URI which was previously without a registration.

*Event 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 [2] for a description of "match_policy".

NOTE: invocation_policy IS NOT YET DESCRIBED IN THE ADVANCED SPEC

#### 13.3.7.1.2.2. wamp.registration.on_register

Fired when a session is added to a registration.

*Event 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.

#### 13.3.7.1.2.3. wamp.registration.on_unregister

Fired when a session is removed from a subscription.

*Event 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.

#### 13.3.7.1.2.4. wamp.registration.on_delete

Fired when a registration is deleted after the last session attached to it has been removed.

*Event Arguments*

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

- "registration|id": The ID of the registration being deleted.

### 13.3.7.1.3. Registration Meta-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.

#### 13.3.7.1.3.1. 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 [3] for a description of match policies.

#### 13.3.7.1.3.2. 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.

#### 13.3.7.1.3.3. 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.

#### 13.3.7.1.3.4. 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 [4] for a description of match policies.

_NOTE: invocation_policy IS NOT YET DESCRIBED IN THE ADVANCED SPEC_

#### 13.3.7.1.3.5. 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 calles 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.

#### 13.3.7.1.3.6. 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.

### 13.3.7.2. Feature Announcement

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

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

*Example*

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
                }
            }
        }
    }
]
```

#### 13.3.8. Pattern-based Registrations

##### 13.3.8.1. Feature Definition

###### 13.3.8.1.1. Introduction

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*

#### 13.3.8.1.2. 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.

#### 13.3.8.1.3. 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.

#### 13.3.8.1.4. General

##### 13.3.8.1.4.1. No set semantics

Since each _Callee_'s' registrations "stands on it's 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.

#### 13.3.8.1.4.2. Calls matching multiple registrations

The behavior when a single call matches more than one of a _Callee's_ registrations or more than one registration in general is still being discussed - see <https://github.com/tavendo/WAMP/issues/182>.

#### 13.3.8.1.4.3. 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!"]
]
```

### 13.3.8.2. 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
```

## 13.3.9. Shared Registration

Feature status: *alpha*

### 13.3.9.1. Feature Definition

As a default, only a single *Callee* may register a procedure for an 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 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 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 attempst to register a procedure for the URI where the value for this option does not match that of the initial registration.

#### 13.3.9.1.1. 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.

#### 13.3.9.1.2. 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.

### 13.3.9.2. 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
```

## 13.3.10. Sharded Registration

Feature status: *sketch*

### 13.3.10.1. Feature Definition

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

### 13.3.10.2. "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.

### 13.3.10.3. 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

#### 13.3.11. Registration Revocation

##### 13.3.11.1. Feature Definition

Actively and forcefully revoke a previously granted registration from a session.

##### 13.3.11.2. Feature Announcement

#### 13.3.12. Procedure 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*

A topic or procedure is defined for reflection:

wamp.reflect.define

A topic or procedure was asked to be described (reflected upon):

wamp.reflect.describe

*Reflection*

A topic or procedure has been defined for reflection:

wamp.reflect.on_define

A topic or procedure has been unfined from reflection:

wamp.reflect.on_undefine

## 13.4. Advanced PubSub Features

### 13.4.1. Subscriber Black- and Whitelisting

#### 13.4.1.1. Introduction

*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*.

#### 13.4.1.2. Use Cases

##### 13.4.1.2.1. 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.

##### 13.4.1.2.2. 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.

#### 13.4.1.3. 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 integeres 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".

#### 13.4.1.4. 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
```

### 13.4.2. Publisher Exclusion

## 13.5. Feature Definition

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 it's 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".

## 13.6. 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
```

#### 13.6.1. Publisher Identification

##### 13.6.1.1. Feature Definition

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.

##### 13.6.1.2. 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
```

### 13.6.2. Publication Trust Levels

#### 13.6.2.1. Feature Definition

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".

#### 13.6.2.2. 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
```

### 13.6.3. Subscription 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.

#### 13.6.3.1. Subscription Meta-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.

##### 13.6.3.1.1. Meta-Event Specifications

###### 13.6.3.1.1.1. wamp.subscription.on_create

Fired when a subscription is created through a subscription request for a topic which was previously without subscribers.

*Event 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 Section 13.6.4 for a description of "match_policy".

#### 13.6.3.1.1.2. wamp.subscription.on_subscribe

Fired when a session is added to a subscription.

*Event 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.

#### 13.6.3.1.1.3. wamp.subscription.on_unsubscribe

Fired when a session is removed from a subscription.

*Event 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.

#### 13.6.3.1.1.4. wamp.subscription.on_delete

Fired when a subscription is deleted after the last session attached to it has been removed.

*Arguments*

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

#### 13.6.3.2. Subscription Meta-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.

##### 13.6.3.2.1. Meta-Procedure Specifications

###### 13.6.3.2.1.1. wamp.subscription.list

Retrieves subscription IDs listed according to match policies.

*Arguments*

- None

*Results*

- "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 Section 13.6.4 for information on match policies.

#### 13.6.3.2.1.2. 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*

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

#### 13.6.3.2.1.3. 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*

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

#### 13.6.3.2.1.4. wamp.subscription.get

Retrieves information on a particular subscription.

*Arguments*

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

*Results*

- "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 Section 13.6.4 for information on match policies.

#### 13.6.3.2.1.5. 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*

- "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.

#### 13.6.3.2.1.6. 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*

- "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.

#### 13.6.4. Pattern-based Subscriptions

##### 13.6.4.1. Introdution

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

##### 13.6.4.2. 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.

#### 13.6.4.3. 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.

#### 13.6.4.4. General

##### 13.6.4.4.1. 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.

##### 13.6.4.4.2. 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.

##### 13.6.4.4.3. 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!"]
]
```

#### 13.6.4.5. 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
```

### 13.6.5. Sharded Subscriptions

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"]
```

### 13.6.6. Event History

#### 13.6.6.1. Feature Definition

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.

#### 13.6.6.2. 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

### 13.6.7. Registration Revocation

#### 13.6.7.1. Feature Definition

Actively and forcefully revoke a previously granted subscription from a session.

#### 13.6.7.2. Feature Announcement

### 13.6.8. Topic Reflection

o see *Procedure Reflection* for now

## 13.7. Other Advanced Features

### 13.7.1. Session Meta API

#### 13.7.1.1. Introduction

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.

#### 13.7.1.2. Session Meta 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.

##### 13.7.1.2.1. 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.

#### 13.7.1.2.2. wamp.session.on_leave

Fired when a session leaves a realm on the router or is disconnected. The event payload consists of a single positional argument "session|id" with the session ID of the session that left.

### 13.7.1.3. Session Meta Procedures

A client can actively retrieve information about 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.

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

#### 13.7.1.3.1. 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.

#### 13.7.1.3.2. 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).

#### 13.7.1.3.3. 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.

#### 13.7.1.4. 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
```

*Example*

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.

### 13.7.2. Authentication

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.

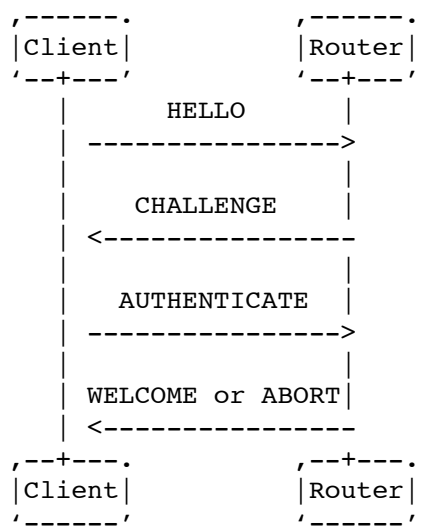
And some applications might want to use a transport independent scheme, nevertheless predefined by WAMP.

#### 13.7.2.1. 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 Section 13.7.2.3 or Section 13.7.2.4 for the use in these authentication methods.

If two-factor authentication is desired, then two subsequent rounds of "CHALLENGE" and "RESPONSE" may be employed.

#### 13.7.2.1.1. 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]
```

#### 13.7.2.1.2. AUTHENTICATE

In response to a "CHALLENGE" message, the Client MUST send an "AUTHENTICATION" message.

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

If the authentication succeeds, the Router MUST send a "WELCOME" message, else it MUST send an "ABORT" message.

### 13.7.2.2. Transport-level Authentication

#### 13.7.2.2.1. 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 forward to the WAMP level within "HELLO.Details.transport.auth|any" in the client-to-server direction.

#### 13.7.2.2.2. 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 forward to the WAMP level within "HELLO.Details.transport.auth|any" in both directions (if available).

#### 13.7.2.3. 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:

```
[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.

#### 13.7.2.3.1. 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"

#### 13.7.2.3.2. 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 thrid party to check.

#### 13.7.2.3.3. 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 [5] 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.

#### 13.7.2.4. 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.

### 13.7.3. Alternative Transports

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:

- Section 13.7.3.1
- Section 13.7.3.2
- Section 13.7.3.3
- Section 13.7.3.4

Other transports such as HTTP 2.0 ("SPDY") or UDP might be defined in the future.

### 13.7.3.1. 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).

#### 13.7.3.1.1. 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:

```
<CODE BEGINS>
import struct

octets_received = b"\x11\x22\x33\x44"
i = struct.unpack(">L", octets_received)[0]
<CODE ENDS>
```

The integer received has the value "287454020".

And here is how you would send out an integer to the wire in Python:

```
<CODE BEGINS>
octets_to_be_send = struct.pack(">L", i)
<CODE ENDS>
```

The octets to be sent are "b"\x11\x22\x33\x44"".

#### 13.7.3.1.2. 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 [6] and here [7]). 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: MsgPack
3 - 15: reserved for future serializers
```

Here is a Python program that prints all (currently) permissible values for the _second octet_:

```
<CODE BEGINS>
SERMAP = {
   1: 'json',
   2: 'msgpack'
}

## 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))
<CODE ENDS>
```

The _third and forth octet_ are *reserved* and MUST be all zeros for now.

*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:

```
<CODE BEGINS>
## 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")
<CODE ENDS>
```

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 it's 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:

```
<CODE BEGINS>
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)
<CODE ENDS>
```

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:

```
<CODE BEGINS>
## 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')))
<CODE ENDS>
```

#### 13.7.3.1.3. Serialization

To send a WAMP message, the message is serialized according to the WAMP serializer agreed in the handshake (e.g. JSON or MsgPack).

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")

#### 13.7.3.1.4. 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.

#### 13.7.3.2. Batched WebSocket Transport for WAMP

_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"

Batching with JSON works by serializing each WAMP message to JSON as normally, appending the single ASCII control character "\30" (record separator [8]) 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 MsgPack works by serializing each WAMP message to MsgPack as normally, prepending a 32 bit unsigned integer (4 octets in big-endian byte order) with the length of the serialized MsgPack 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 MsgPack 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.

#### 13.7.3.3. A HTTP Longpoll Transport for WAMP

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"

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").

#### 13.7.3.4. Multiplexed Transport

A Transport may support the multiplexing of multiple logical transports over a single "physical" transport.

By using such a Transport, multiple WAMP sessions can be transported over a single underlying transport at the same time.

As an example, the proposed WebSocket extension "permessage-priority" [9] would allow creating multiple logical Transports for WAMP over a single underlying WebSocket connection.

Sessions running over a multiplexed Transport are completely independent: they get assigned different session IDs, may join different realms and each session needs to authenticate itself.

Because of above, Multiplexed Transports for WAMP are actually not detailed in the WAMP spec, but a feature of the transport being used.

> Note: Currently no WAMP transport supports multiplexing. The work on the MUX extension with WebSocket has stalled, and the "permessage-priority" proposal above is still just a proposal. However, with RawSocket, we should be able to add multiplexing in the the future (with downward compatibility).

#### 13.7.3.5. 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.

#### 13.7.3.6. 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.

## 14. Binary conversion of JSON Strings

Binary data follows a convention for conversion to JSON strings.

A *byte array* is converted to a *JSON string* as follows:

1. convert the byte array to a Base64 encoded (host language) string

2. prepend the string with a "\0" character

3. serialize the string to a JSON string

_Example_

Consider the byte array (hex representation):

```
10e3ff9053075c526f5fc06d4fe37cdb
```

This will get converted to Base64

```
EOP/kFMHXFJvX8BtT+N82w==
```

prepended with "\0"

```
\x00EOP/kFMHXFJvX8BtT+N82w==
```

and serialized to a JSON string

```
"\\u0000EOP/kFMHXFJvX8BtT+N82w=="
```

A *JSON string* is unserialized to either a *string* or a *byte array* using the following procedure:

1. Unserialize a JSON string to a host language (Unicode) string
2. If the string starts with a "\0" character, interpret the rest (after the first character) as Base64 and decode to a byte array
3. Otherwise, return the Unicode string

Below are complete Python and JavaScript code examples for conversion between byte arrays and JSON strings.

### 14.1. Python

Here is a complete example in Python showing how byte arrays are converted to and from JSON:

```
<CODE BEGINS>

import os, base64, json, sys, binascii
PY3 = sys.version_info >= (3,)
if PY3:
   unicode = str

data_in = os.urandom(16)
print("In:   {}".format(binascii.hexlify(data_in)))

## encoding
encoded = json.dumps('\0' + base64.b64encode(data_in).
                                        decode('ascii'))

print("JSON: {}".format(encoded))

## decoding
decoded = json.loads(encoded)
if type(decoded) == unicode:
   if decoded[0] == '\0':
      data_out = base64.b64decode(decoded[1:])
   else:
      data_out = decoded

print("Out:  {}".format(binascii.hexlify(data_out)))

assert(data_out == data_in)

<CODE ENDS>
```

## 14.2. JavaScript

Here is a complete example in JavaScript showing how byte arrays are converted to and from JSON:

```
<CODE BEGINS>

var data_in = new Uint8Array(new ArrayBuffer(16));

// initialize test data
for (var i = 0; i < data_in.length; ++i) {
   data_in[i] = i;
}
console.log(data_in);

// convert byte array to raw string
var raw_out = '';
for (var i = 0; i < data_in.length; ++i) {
   raw_out += String.fromCharCode(data_in[i]);
}

// base64 encode raw string, prepend with \0
// and serialize to JSON
var encoded = JSON.stringify("\0" + window.btoa(raw_out));
console.log(encoded); // "\u0000AAECAwQFBgcICQoLDA0ODw=="

// unserialize from JSON
var decoded = JSON.parse(encoded);

var data_out;
if (decoded.charCodeAt(0) === 0) {
   // strip first character and decode base64 to raw string
   var raw = window.atob(decoded.substring(1));

   // convert raw string to byte array
   var data_out = new Uint8Array(new ArrayBuffer(raw.length));
   for (var i = 0; i < raw.length; ++i) {
      data_out[i] = raw.charCodeAt(i);
   }
} else {
   data_out = decoded;
}

console.log(data_out);

<CODE ENDS>
```

## 15. Security Considerations

-- write me --

## 16. IANA Considerations

TBD

## 17. Contributors

## 18. Acknowledgements

## 19. References

### 19.1. Normative References

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <http://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, <http://www.rfc-editor.org/info/rfc3986>.

[RFC6455] Fette, I. and A. Melnikov, "The WebSocket Protocol", RFC 6455, DOI 10.17487/RFC6455, December 2011, <http://www.rfc-editor.org/info/rfc6455>.

### 19.2. 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, <http://www.rfc-editor.org/info/rfc2119>.

### 19.3. URIs

[1] http://www.iana.org/assignments/websocket/websocket.xml

[2] pattern-based-registration.md

[3] pattern-based-registration.md

[4] pattern-based-registration.md

[5] http://en.wikipedia.org/wiki/PBKDF2

[6] http://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html#sec5.1

[7] http://www.w3.org/Protocols/rfc2616/rfc2616-sec2.html#sec2.2

[8] http://en.wikipedia.org/wiki/Record_separator#Field_separators

[9] https://github.com/oberstet/permessage-priority/blob/master/draft-oberstein-hybi-permessage-priority.txt

## Authors' Addresses

Tobias G. Oberstein
Tavendo GmbH

Email: tobias.oberstein@tavendo.de

Alexander Goedde
Tavendo GmbH

Email: alexander.goedde@tavendo.de