DDP Specification

DDP is a protocol between a client and a server that supports two operations:

- Remote procedure calls by the client to the server.
- The client subscribing to a set of documents, and the server keeping the client informed about the contents of those documents as they change over time.

This document specifies the version "1" of DDP. It's a rough description of the protocol and not intended to be entirely definitive.

General Message Structure:

DDP may use either SockJS or WebSockets as a lower-level message transport. (For now, you connect via SockJS at the URL `/sockjs` and via WebSockets at the URL `/websocket`. The latter is likely to change to be the main app URL specifying a WebSocket subprotocol.)

DDP messages are JSON objects, with some fields specified to be EJSON. Each one has a `msg` field that specifies the message type, as well as other fields depending on message type.

The client and the server must ignore any unknown fields in messages. Future minor revisions of DDP might add extra fields without changing the DDP version; the client must therefore silently ignore unknown fields. However, the client must not send extra fields other than those documented in the DDP protocol, in case these extra fields have meaning to future servers. On the server, all field changes must be optional/ignorable for compatibility with older clients; otherwise a new protocol version would be required.

Establishing a DDP Connection:

Messages:

- `connect` (client -> server)
  - `session` : string (if trying to reconnect to an existing DDP session)
  - `version` : string (the proposed protocol version)
  - `support` : array of strings (protocol versions supported by the client, in order of preference)
- `connected` (server->client)
  - `session` : string (an identifier for the DDP session)
- `failed` (server->client)
  - `version` : string (a suggested protocol version to connect with)

Procedure:

The server may send an initial message which is a JSON object lacking a `msg` key. If so, the client should ignore it. The client does not have to wait for this message. (The message was once used to help implement Meteor's hot code reload feature; it is now only included to force old clients to update).

- The client sends a `connect` message.
- If the server is willing to speak the `version` of the protocol specified in the `connect` message, it sends back a `connected` message.
- Otherwise the server sends back a `failed` message with a version of DDP it would rather speak, informed by the `connect` message's `support` field, and closes the underlying transport.

- The client is then free to attempt to connect again speaking a different version of DDP. It can do that by sending another `connect` message on a new connection. The client may optimistically send more messages after the `connect` message, assuming that the server will support the proposed protocol version. If the server does not support that version, it must ignore those additional messages.

The versions in the `support` field of the client's `connect` message are ordered according to the client's preference, most preferred first. If, according to this ordering, the `version` proposed by the client is not the best version that the server supports, the server must force the client to switch to the better version by sending a `failed` message.

When a client is connecting to a server for the first time it will typically set `version` equal to its most preferred version. If desired, the client can then remember the version that is ultimately negotiated with the server and begin with that version in future connections. The client can rely on the server sending a `failed` message if a better version is possible as a result of the client or the server having been upgraded.

Heartbeats

Messages:

- `ping`
  - `id` : optional string (identifier used to correlate with response)
- `pong`
  - `id` : optional string (same as received in the `ping` message)

Procedure:

At any time after the connection is established either side may send a `ping` message. The sender may chose to include an `id` field in the `ping` message. When the other side receives a `ping` it must immediately respond with a `pong` message. If the received `ping` message includes an `id` field, the `pong` message must include the same `id` field.

Managing Data:

Messages:

- `sub` (client -> server):
  - `id` : string (an arbitrary client-determined identifier for this subscription)
  - `name` : string (the name of the subscription)
  - `params` : optional array of EJSON items (parameters to the subscription)
- `unsub` (client -> server):
  - `id` : string (the id passed to 'sub')
- `nosub` (server -> client):
  - `id` : string (the id passed to 'sub')
  - `error` : optional Error (an error raised by the subscription as it concludes, or sub-not-found)
- `added` (server -> client):
  - `collection` : string (collection name)
  - `id` : string (document ID)
  - `fields` : optional object with EJSON values
- `changed` (server -> client):

    - `collection` : string (collection name)
    - `id` : string (document ID)
    - `fields` : optional object with EJSON values
    - `cleared` : optional array of strings (field names to delete)
- `removed` (server -> client):
    - `collection` : string (collection name)
    - `id` : string (document ID)
- `ready` (server -> client):
    - `subs` : array of strings (ids passed to 'sub' which have sent their initial batch of data)
- `addedBefore` (server -> client):
    - `collection` : string (collection name)
    - `id` : string (document ID)
    - `fields` : optional object with EJSON values
    - `before` : string or null (the document ID to add the document before, or null to add at the end)
- `movedBefore` (server -> client):
    - `collection` : string
    - `id` : string (the document ID)
    - `before` : string or null (the document ID to move the document before, or null to move to the end)

Procedure:

- The client specifies sets of information it is interested in by sending `sub` messages to the server.
- At any time, but generally informed by the `sub` messages, the server can send data messages to the client. Data consist of `added` , `changed` , and `removed` messages. These messages model a local set of data the client should keep track of.
    - An `added` message indicates a document was added to the local set. The ID of the document is specified in the `id` field, and the fields of the document are specified in the `fields` field. Minimongo interperets the string id field in a special way that transforms it to the _id field of Mongo documents.
    - A `changed` message indicates a document in the local set has new values for some fields or has had some fields removed. The `id` field is the ID of the document that has changed. The `fields` object, if present, indicates fields in the document that should be replaced with new values. The `cleared` field contains an array of fields that are no longer in the document.
    - A `removed` message indicates a document was removed from the local set. The `id` field is the ID of the document.
- A collection is either ordered, or not. If a collection is ordered, the `added` message is replaced by `addedBefore` , which additionally contains the ID of the document after the one being added in the `before` field. If the document is being added at the end, `before` is set to null. For a given collection, the server should only send `added` messages or `addedBefore` messages, not a mixture of both, and should only send `movedBefore` messages for a collection with `addedBefore` messages.

  NOTE: The ordered collection DDP messages are not currently used by Meteor. They will likely be used by Meteor in the future.

- The client maintains one set of data per collection. Each subscription does not get its own datastore, but rather overlapping subscriptions cause the server to send the union of facts about the one collection's data. For example, if subscription A says document `x` has fields `{foo: 1, bar: 2}` and subscription B says document `x` has fields `{foo: 1, baz:3}`, then the client will be informed that document `x` has fields `{foo: 1, bar: 2, baz: 3}`. If field values from different subscriptions conflict with each other, the server should send one of the possible field values.
- When one or more subscriptions have finished sending their initial batch of data, the server will send a `ready` message with their IDs.

Remote Procedure Calls:

Messages:

- `method` (client -> server):
  - `method` : string (method name)
  - `params` : optional array of EJSON items (parameters to the method)
  - `id` : string (an arbitrary client-determined identifier for this method call)
  - `randomSeed` : optional JSON value (an arbitrary client-determined seed for pseudo-random generators)
- `result` (server -> client):
  - `id` : string (the id passed to 'method')
  - `error` : optional Error (an error thrown by the method (or method-not-found)
  - `result` : optional EJSON item (the return value of the method, if any)
- `updated` (server -> client):
  - `methods` : array of strings (ids passed to 'method', all of whose writes have been reflected in data messages)

Procedure:

- The client sends a `method` message to the server
- The server responds with a `result` message to the client, carrying either the result of the method call, or an appropriate error.
- Method calls can affect data that the client is subscribed to. Once the server has finished sending the client all the relevant data messages based on this procedure call, the server should send an `updated` message to the client with this method's ID.
- There is no particular required ordering between `result` and `updated` messages for a method call.
- The client may provide a randomSeed JSON value. If provided, this value is used to seed pseudo-random number generation. By using the same seed with the same algorithm, the same pseudo-random values can be generated on the client and the server. In particular, this is used for generating ids for newly created documents. If randomSeed is not provided, then values generated on the server and the client will not be identical.
- Currently randomSeed is expected to be a string, and the algorithm by which values are produced from this is not yet documented. It will likely be formally specified in future when we are confident that the complete requirements are known, or when a compatible implementation requires this to be specified.

Errors:

Errors appear in `result` and `nosub` messages in an optional error field. An error is an Object with the following fields:

- `error` : string (previously a number. See appendix 3)
- `reason` : optional string
- `message` : optional string
- `errorType` : pre-defined string with a value of `Meteor.Error`

Such an Error is used to represent errors raised by the method or subscription, as well as an attempt to subscribe to an unknown subscription or call an unknown method.

Other erroneous messages sent from the client to the server can result in receiving a top-level `msg: 'error'` message in response. These conditions include:

- sending messages which are not valid JSON objects
- unknown `msg` type
- other malformed client requests (not including required fields)
- sending anything other than `connect` as the first message, or sending `connect` as a non-initial message

The error message contains the following fields:

- `reason` : string describing the error
- `offendingMessage` : if the original message parsed properly, it is included here

Appendix: EJSON

EJSON is a way of embedding more than the built-in JSON types in JSON. It supports all types built into JSON as plain JSON, plus the following:

Dates:

```
{"$date": MILLISECONDS_SINCE_EPOCH}
```

Binary data:

```
{"$binary": BASE_64_STRING}
```

(The base 64 string has `+` and `/` as characters 62 and 63, and has no maximum line length)

Escaped things that might otherwise look like EJSON types:

```
{"$escape": THING}
```

For example, here is the JSON value `{$date: 10000}` stored in EJSON:

```
{"$escape": {"$date": 10000}}
```

Note that escaping only causes keys to be literal for one level down; you can have further EJSON inside. For example, the following is the key `$date` mapped to a Date object:

```
{"$escape": {"$date": {"$date": 32491}}}
```

User-specified types:

```
{"$type": TYPENAME, "$value": VALUE}
```

Implementations of EJSON should try to preserve key order where they can. Users of EJSON should not rely on key order, if possible.

> *MongoDB relies on key order. When using EJSON with MongoDB, the implementation of EJSON must preserve key order.*

Appendix 2: randomSeed backward/forward compatibility

randomSeed was added into DDP pre2, but it does not break backward or forward compatibility.

If the client stub and the server produce documents that are different in any way, Meteor will reconcile this difference. This may cause 'flicker' in the UI as the values change on the client to reflect what happened on the server, but the final result will be correct: the server and the client will agree.

Consistent id generation / randomSeed does not alter the syncing process, and thus will (at worst) be the same:

- If neither the server nor the client support randomSeed, we will get the classical/flicker behavior.
- If the client supports randomSeed, but the server does not, the server will ignore randomSeed, as it ignores any unknown properties in a DDP method call. Different ids will be generated, but this will be fixed by syncing.
- If the server supports randomSeed, but the client does not, the server will generate unseeded random values (providing a randomSeed is optional); different ids will be generated; and again this will be fixed by syncing.
- If both client and server support randomSeed, but different ids are generated, either because the generation procedure is buggy, or the stub behaves differently to the server, then syncing will fix this.
- If both client and server support randomSeed, in the normal case the ids generated will be the same, and syncing will be a no-op.

Appendix 3: Error message `error` field can be a string or a number

In the `pre1` and `pre2` specifications for DDP, the `error` field on error messages was documented as a number, and in practice often matched the closest HTTP error code.

It is typically better to be specific about the type of error being returned, so new code should use strings like `'wrong-password'` instead of `401`. To support both the new string error codes and the old number codes, a DDP client should accept a string or a number in this field. Many meteor packages, including some core packages, still use the old numeric codes.

Version History

`pre1` was the first version of DDP

`pre2` added keep-alive (ping & pong messages), and randomSeed.

`1` should be considered the first official version of DDP. The type of the error code field has been clarified to reflect the behavior of existing clients and servers.