BCP Message Set

October 21, 2011

This document is a continuous process of flux and being fleshed out. Direct any questions or complaints to Philip Horger at campadrenalin@gmail.com.

Part I
Transfer of live data

1 select

Until next “select” message, all document-specific messages will apply to the given docname.

{”type”:”select”, “docname”:”willynelson”}

2 op

An operation, consisting of a list of instructions.

{”type”:”op”, “instructions”:[...]}

3 subscribe

Subscribing to a document indicates to the remote to forward any ops on that docname to you. The “docnames” parameter is an optional list of docnames, if it is not present, the currently selected docname is used. You will not recieve op messages for a docname if you do not subscribe to it.

{”type”:”subscribe”, “docnames”:[“willynelson”]}

{”type”:”subscribe”}

4 unsubscribe

Disable any subscription on the given docnames. *Warning: sending an empty list will disable all subscriptions from the remote end to you, so never send this if you don't mean it! This is very different from omitting the parameter, which unsubscribes you from the selected docname.*

> {"type":"unsubscribe", "docnames":["The Event", "Smallville"]}
>
> {"type":"unsubscribe", "docnames":[]} // unsubscribes you from everything
>
> {"type":"unsubscribe"} // unsubscribes you from currently selected docname

Part II
Syncronization/Merging

5 handshake

The very first message you send before anything else. Both ends must "shake" before a connection is valid: the initiating end first, then the responding end. After the handshake the protocol is initiation-agnostic, meaning it doesn't treat the initiator different from the responder.

A handshake message tells the remote end what message types you are capable of *accepting*, and in which versions (in order of preference). A version can be a string or int, and a set of versions can be given as a single version if there's only one (though it should be treated as an array with a single element). When sending a versioned message, the type is the base message type concatenated with the string representation of the version, except version 0, which is treated as "".

The handshake itself is unversioned, and will never change throughout the history of the protocol. You don't need to bring it up in the handshake message itself.

> {"type":"handshake",
>
> > "select":0,
> > "op":0,
> > "tree":[0,1],
> > "rtoken":["_facebook", "-webkit0"],
> > "metadata":"-webkit0"
>
> }

6 check

Request for the checksum of a node of the currently selected tree. Allows you to quickly check syncronicity.

{"type":"check", "address": [["You're a "]] }

7 tsum

"Tree sum," the response to "check". Returns the treesum for an address.

{"type":"tsum", "address":[["You're a"]], "value":656172}

8 get

A request for one or more "tree" responses. The optional "depth" property specifies how many layers deep the response should be in real nodes (anything deeper than that in the response is implied to be a flat node). For example, if depth=1, the response should be just the requested node, and flat versions of its children.

{"type":"get", "address":[["Barney th"], [1, "l"]]}

9 tree

A concisely-constructed representation of a tree. See the object reference in Part V for more information.

{"type":"tree", "address"[["Barney th"], [1, "l"]], value": ... }

Part III
Authorization

10 token

The thing you actually authorize with. There are two types so far, but OpenID support is expected to expand this at some point in the future.

local tokens are session tokens specific to the site you're logging into. Further details are arbitrary, although a simple unsigned int session identifier is a practical option for most implementations. Should either include an expiration date or imply one from the date it is issued. Servers must support multiple logins with a single token simultaneously.

identity tokens have the properties “address”, “key”, and “sigs”. Address is a valid MCP address, “key” is the corresponding public key, and “sigs” is a list of address/signature pairs expressed as a mapping ({}).

{”type”:”token”, “subtype”:”local”, “something”:”somethingelse”}

{”type”:”token”, “subtype”:”identity”,

“address”:”philip.horger@orchard.com”,
“key”:”<some pgp public key>”,
“sigs”:{
“nat.abbots@orchard.com”:”<philip’s public key signed with nat’s private key>”
}

}

11 login

A local token request. The response should be an rtoken message. Username and password are standard arguments but this can be as application-specific as you need it to be, although if you’re going really off the wall you can always make up a proprietary token type, expand the message set, whatever you need.

{”type”:”login”, ”username”:”philip”, ”password”:”bukket”}

12 logout

Ends a local token session. Returns an rtoken with property “valid” set to false.

{”type”:”logout”, ”token”: ... }

13 rtoken

A server-provided local token in response to a login request (response token -> rtoken). May include “valid” property when it would be true, must include it when it is false.

{”type”:”rtoken”, “token”: ... , “valid”:true}

Part IV
Meta

These messages are not part of core BCP, they’re simply a standard starting point for extensions.

14 lookup

Look up metadata about something, found by an identifier. The identifier will vary in type and format based on how you're using BCP, it can be whatever you want as long as you're consistent. In ECP, for example, we use an email address for finding people. Example subtypes are *person* and *document.*

{"type":"lookup", "subtype":"person" "id":"philip.horger"}

15 metadata

The response to "lookup". The only defined properties are "id" and "name", the latter of which can be either a string or a dictionary (for things like last name, first name, and username). It's really open-ended for a reason - you can use it for anything you want. Obviously you have to be self-consistent, though!

{

"type" : "metadata",

"subtype" : "person",

"id" : "philip.horger",

"name":{

"first":"Philip",
"last":"Horger",
"nick":"Philip"

}, "avatar":"http://example.com/avatar.gif",

"hobby":"Collecting decapitated teddy bears"

}

Part V
Objects

16 Address

Addresses are *the* way to specify which tree in a document you're talking about. Addresses are arrays of shortcut chains and steps.

[

["re", "quest", " some "],
[4,"carbide ma46456"]

]

A 2-element address starting with a shortcut chain, and then progressing to a subtree (position 4, key "carbide ma46456", which happens to have a value of "carbide manufacturing".)

A step is an index/key pair that indicates moving from a parent to a child. A shortcut chain is a list of strings only - specifically, keys. The position for each key is assumed to be the highest/last possible position in the node in question.

17 Instruction

Instructions are one of the integral pieces of ConcurrenTree technology. Their JSON serialization is very compact and much less semantically friendly than most of BCP, but bandwidth is a big concern when you're sending (potentially) thousands of instructions.

[0, [["Have you e51685"]], 3, [5,7]]

[1, [["Have you e51685"]], 4, "querque, by W"]

Two examples: a deletion of positions 3 and range 5-7, and an insertion at position 4 of a node with value "querque, by W"

An instruction always starts off with an integer typecode and an address. The typecode specifies what kind of instruction it is.

0 Deletion. Arguments 3-n are ints and ranges. A range is a 2 element array, start and end inclusive. You must have at least one range or int.

1 Insertion. Third argument is the position in the parent node, fourth argument is the immutable child value.

18 Tree

Used in "tree" messages, tree structures serialize a tree and its descendents in a structurally minimalist way.

[

"I feel dis",

["tres", []],

"sed.",

[[0, 6]]

]

> *This example describes a node that has immutable value* "I feel dissed.*", and then through a child node and a deletion over the range [0,6], is transformed into* "distressed.*"*

The tree is described as an array. The tree array can contain strings and children, always ending with a deletionset made of ints, and ranges. The ints and ranges represent exactly what they do in deletion instructions.

The strings represent pieces of the tree's immutable string. Each tree array string is a contiguous piece of the tree uninterrupted by children. At positions where there are children, the strings must be interrupted with those children, in any order (order is inferred by the child's value).

19 Flat

A Flat is included in a tree structure, and represents a collapsed version of a node. It's a three element list: key, value, and treesum. It's distinguishable from trees by its ending with an int (versus an array).

> ["sterisk", "asterisks can hide a whole lot of", 56123]

20 Summable

A single node representation used by the treesum algorithm. It's like the tree representation, but where there are children, they're expressed as a dict keyed on child key, with that node's treesum as the value. The treesum of a node is the sum of the strict of its summable.

You might ask why we have two ways to represent nodes. The answer is efficiency. The Tree is good for serializing all information about a tree in a way that can be unpacked later, while the Summable is intended to have as few and easy knowledge dependencies as possible, conveying the minimum amount of information necessary to check equivalence.

> ["Tokyo D", {"ri":5460}, "irft", [7,8]]

Part VI
Errors

All errors have roughly the same format:

```
{
    "type":"error",
    "code":451,
    "detail":"Bad message"
```

}

An integer code representing the BCP error that has occurred, a more detailed textual message explaining the error, and the type is always "error". Some errors also return an optional "data" property, the type and meaning depends on the error code. However, even on these error codes, the data property is still *optional,* therefore you must not depend on it being supplied.

The code prefixes are very roughly based on HTTP error codes to allow the more well-known ones to coincide (for example, BCP 404 is very closely comparable to HTTP 404).

A full, more up-to-date list of error types will be available in the root of this repository under the name of "ErrorCodes.txt", but here are the headlines:

21 1XX : Connection Status

These messages are the only ones that are rarely actually sent over the wire. They are optionally generated by the backend code that maintains and manages a connection, to signal the BCP-based code when the state of the connection changes. It's largely undeveloped right now and totally optional.

22 2XX : Unused

Official BCP does not use the 2XX series because it simply doesn't need it. Therefore we're reserving it for proprietary extensions to the BCP message set. That's right - you can make up your own errors for the 2XX series based on your custom BCP extensions, and rest comfortably knowing that there will never be an official BCP version that conflicts with them. 6XX and above are not reserved in this way.

23 3XX : Authentication Errors

Authentication is such a big and complicated thing, it really needed its own error series. So we made one.

24 4XX : Foreign Error

When the remote end messes up, send them a 4XX error message to let them know. In some cases this will match the client/server error codes of HTTP, and certainly matches up with the intention of the series, but we use the Foreign/Local terminology instead to prevent confusion in P2P contexts (which ConcurrenTree was originally invented for).

400 is a general error for some unspecified problem with the foreign message, 401-449 are reserved for semantic and advanced errors, and 450-499 are for parsing errors. The distinction is a bit like the difference between a pumpkin

pie that's clearly been run over by a truck and inedible (unparsable), and a nice, friendly-looking pie with a bunch of razor blades in it (parsable, but with bad data inside). Well-formed messages will not trigger parsing errors, and valid data will not trigger semantic errors.

400 Bad message

401 Unknown message type

402 Document moved temporarily[1]

403 Document moved permanently

404 Document not found by that name[2]

451 Could not parse JSON

452 Missing required argument[3]

453 Wrong argument type

454 Wrong JSON root type[4]

25 5XX : Local Error

This series is for when responsibility for the error rests on the local side. It's a lot nicer to send a vague 500 error to the remote end when something goes wrong than to just drop the connection, or try to ignore it. Of course, the more specific you can be with your error, the better equipped the remote end is to handle it.

500 Unspecified local error

501 Service not supported[5]

502 Resource not found[6]

503 Service recognized but unavailable

504 Request timeout

505 BCP version not supported

[1] 402 and 403 supply the alternate location as the data property.
[2] Returns failed docname as data property
[3] 452 and 453 return argument name and type as two element string array.
[4] Only JSON objects ({}) are allowed to be messages.
[5] BCP's message set is divided into services. This error tells you that the service named in the data property is not supported at this location, as opposed to 503, where it is intended to be available but is not, due to such things as high traffic load.
[6] Like 404, but more general, for things like eras and lookup requests.