BCP Message Set

August 1, 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 ad

A message that tells other clients that it has an operation ready to read. This is more time- and bandwidth-efficient than broadcasting an op to machines that likely already have it. For this reason, clients *should* straight-broadcast ops that originate from them, and advertise ops from other sources. [1]

{"type":"ad", "hash":"7344"}

[1] Be advised that this, like all references to hashing in this document, refers to the checksum() function described in the general BCP documentation.

4 getop

The reaction message to “ad”. As you might guess, it requests an op message by a hash.

{”type”:”getop”, “hash”:”7344”}

Part II
Syncronization/Merging

5 check

Request for hashes of eras of the currently selected tree. The “eras” property may be an int or a range (2-element array). -1 is a special value that means “the highest era you know is in the document”.

{”type”:”check”, “eras”:5}

{”type”:”check”, “eras”:[1,2]}

6 thash

“Tree hash,” the response to “check”. “eras” will always be a dictionary with era numbers for keys and int checksums for values. Any ranges will be expanded by the server into individual era numbers. The checksums are of Era objects, as described in Part IV of this document.

{”type”:”thash”, “eras”:{”1”:44263...}}

7 get

A request for one or more “era” responses. This can either be a “flat” or a “tree”, denoting whether to send the flattened result of all previous eras, or send the actual eras themselves. If the property “flat” is used, it must be an int. If the property “tree” is used, it’s in the same format as the “eras” property for the “check” message. Both can be used in the same message.

{”type”:”get”, “flat”:5}

{”type”:”get”, “tree”:[9,-1]}

// or, as a single message

{”type”:”get”, “flat”:5,“tree”:[9,-1]}

8 era

A concisely-constructed representation of an era. While this *can* be expressed as an operation, this would be fairly bulky compared to a purely structural view. It's a right-tool-for-the-right-job situation. Ops modify, eras declare.

While a "get" can request a lot of stuff at once, each "era" response can only send one thing at a time.

{"type":"era","subtype":"flat", "layer":5,"flat":[...]}

{"type":"era","subtype":"tree", "layer":9,"era":{...}}

Part III
Notifications

There is a bit of functional overlap between this and Part I (live transfer). Just remember that you still have to subscribe to a document for other machines to send live updates of any kind to you.

9 subscribe

There are different levels of subscription: *oponly, live*, and *notify*. A document can only be subscribed as one at a time, so subscribing to something twice will put it in whatever level was used the second time, overwriting the level of the previous subscription.

oponly is exactly what it sounds like. Don't bother sending ads, always send the operation directly and save a round trip by assuming the op has not been recieved from another source.

live means that the remote end will send operations (or ads) as soon as they happen, and ads when the operation is secondhand. This is the default mode, and will be assumed if no "subtype" argument is present.

notify means that only ads will be sent.

{"type":"subscribe", "subtype":"live", "docnames":["willynelson"]}

10 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!*

{”type”:”unsubscribe”, “docnames”:[“The Event”, “Smallville”]}

{”type”:”unsubscribe”, “docnames”:[]} // unsubscribes you from everything

Part IV
Authorization

11 login

The BCP authorization system uses, for the most part, the private/public key model. The login message is provided as an alternative for more traditional username/password scenarios.

The response to a correct login is an authok with no “key” property. Incorrect logins should trigger an error.

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

12 auth

A request for an MEL of a random string. A live BCP node can only authorize for documents it’s a part of/knows the participants of.

{”type”:”auth”,”docname”:”willynelson”}

13 authmel

The response to an auth request. It’s a random string encoded into an MEL (multiply-encoded list, where each element is the plaintext data encoded in a different participant’s public key).

{”type”:”authmel”, ”docname”:”willynelson”, “mel”:[...]}

14 authmd

Response to authmel. It’s a list of decrypted strings, one of which hopefully will match the random string. It stands for Authorization MEL Decrypted.

{”type”:”authmd”, ”docname”:”willynelson”, “mel”:[...]}

15 authok

The final part of the auth dance. The server sends the client the symmetric key of the document.

{"type":"authok", "docname":"willynelson", "key":"..."}

16 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 a public key for finding people. Example subtypes are *person* and *document.*

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

17 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", "id":"philip.horger", "name":{

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

}, "avatar":"http://example.com/avatar.gif",
"hobby":"Collecting decapitated teddy bears"
}

Part V
Meta

Messages that contain messages. Well, except for "key", but it still seemed like the right place to put that one, so hush.

18 part

In order to keep large messages from hanging a connection interminably, BCP enforces a very strict character limit: 2048 characters per message. This is more than enough for most messages, but for larger messages, it's the grim reaper.

Multipart messages are a simple way to send large messages without clogging the pipe or hitting the character limit. The example below is a segment of a large segmented message. The contained message is treated like a big string and will be evaluated as soon as a piece is sent with the property done set to true (this can be ommitted in any non-final piece).

While it supports the hashalgorithm property used for the "encrypted" message type, such precautions are much less likely to be necessary for the "part" message type.

{"type":"part","hash":"44385","contents":"..."}

19 encrypted

Ideally, you should run a BCP stream inside an encrypted connection (preferably point-to-point), instead of running encrypted messages over an unencrypted channel. Sometimes, though, that's not an option, and we want to make a standard mechanism for encryption in those use cases.

When there is a key for each end, the sender always encrypts with their private key first, then with the recipient's public key (the recipient decrypts this in reverse order). If the docname selected by the sending end refers to an encrypted document, they must add a layer of encryption with the document's symettric key beneath the endpoint encryption layers.

If you're using cryptography on your connection, you probably have support for advanced hashing algorithms on both ends, too. Not only that, but this is one of the few cases where the hashes actually need to be cryptographically secure. Therefore we give you a "hashalgorithm" property to indicate what you're encoding in, which the remote end can accept, or respond with an error that contains a list of all supported algorithms. "default" is the BCP sum() function and should not be used for this. However, it is what will be assumed if you do not specify the algorithm (hashalgorithm is an optional property).

{"type":"encrypted",

"hash":"0b00411f",

"hashalgorithm":"md5",

"contents":"..."}

20 key

You probably noticed that there was no key exchange in the "encrypted" message. That's because we do that with "key" messages, which work a bit like "select" messages. Key messages exchange public keys between ends of a connection, and are also used to acknowledge that a key has been recieved. A key can be set to a string or to null.

Encryption should not be used on a key until it has been acknowledged.

{”type”:”key”, “mine”:”...”, “yours”:null}

Part VI
Objects

21 Address

Addresses are *the* way to specify which tree in a document you're talking about. Addresses are arrays, with an optional shortcut as the first element.

```
[
    “2#5235”,
    [4,"carbide ma46456"]
]
```

A 2-element address starting at shortcut “2#5235” and progressing to a subtree in 5235 (position 4, key “carbide ma46456”, which happens to have a value of “carbide manufacturing”.)

When no shortcut address is given, special address “0#root” is assumed. This refers to the single root element (which has value “”) every BCP document has. So in these cases, the very first element progresses down the tree from the root node. Regular elements are in the form of 2-element arrays of [pos, key].

22 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, [[0,”Have you e51685”]], 3, [5,7] ]
[1, [[0,”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.

23 Tree

Used in "era" messages, tree structures describe 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).

24 Era

An era is a container for trees. It's nothing more than a dict keyed on shortcut address, with Tree arrays as values (see above).

{"1288":[*...tree data...*]}

Era messages use these to relate trees that are on the same layer, but with no other guarantees of being related.

25 Flat

A flat gives an era its context, usually for rendering more than correctness. An era is a set of trees on the same layer, but without a basic tree describing all previous layers and how they connect to the era, it's kinda useless information.

A flat is a compressed description of zero or more eras that you can plug an era into, without having to actually know what's in those eras.

["A", 45899," example of a ", 112,"flat."]

All the tree data in era 0 has been flattened into a compact representation that includes shortcut references in the correct order and positions.

The Flat is very similar to the tree, except instead of child trees, it has integer references to shortcuts. The main part of the shortcut is always an int, so we just express shortcut refs as ints, and use the type to imply that they're not part of the Flat's value.

Part VII
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:

26 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.

27 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.

28 3XX : Authentication Errors

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

29 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).

400 Bad message

401 Unknown message type

402 Document moved temporarily[2]

403 Document moved permanently

404 Document not found by that name[3]

451 Could not parse JSON

452 Missing required argument[4]

453 Wrong argument type

454 Wrong JSON root type[5]

[2] 402 and 403 supply the alternate location as the data property.
[3] Returns failed docname as data property
[4] 452 and 453 return argument name and type as two element string array.
[5] Only JSON objects ({}) are allowed to be messages.

30 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[6]

502 Resource not found[7]

503 Service recognized but unavailable

504 Request timeout

505 BCP version not supported

[6] 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.

[7] Like 404, but more general, for things like eras and lookup requests.