BCP Message Set

July 20, 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.

{"type":"ad", "hash":"89a7344"}

4 getop

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

{”type”:”getop”, “hash”:”89a7344”}

Part II
Syncronization/Merging

5 check

Request for hashes of eras of the currently selected tree. The “eras” property should be an array of ints and/or strings. Strings denote ranges, possibly including X, which represents the current top era. Strings that do not follow the /[0-9xX]?-[0-9xX]?/ should be rejected and invalidate the message and trigger an error.

{”type”:”check”,”eras”:[1,3,5,”9-11”, “13-X”]}

6 thash

“Tree hash,” the response to “check”. “eras” will always be a dictionary with era numbers (integers) for keys and string hashes for values. Any ranges will be expanded by the server into individual era numbers.

{”type”:”thash”, “eras”:{1:”7a844263”...}}

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-X”]}

// or, as a single message

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

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,"objects":{...}}

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

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).

{"type":"part","hash":"ac44f385","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).

{"type":"encrypted", "hash":"0b00411f", "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 Instruction

Instructions are one of the integral pieces of ConcurrenTree technology. *TODO: fill this section in later using Wave as a reference.*

22 Tree

Used in "era" messages, tree structures use both direct containment in the data structure and references to objects outside them (simply use an array of child hashes rather than a dictionary matching hashes to values). It's used to describe flats as well, which pack rather succinctly in this structure.

{
"2/39e2125b":{

"text":"labama",
"deletions":[0,5,6,7],
"children":{4:{

```
            “5821ce93”:{ ... }
            “93bef739”:{ ... }
        }},
        “markers”:{1:{
            “style/italic”:true;
        }}
        }
}
```

Part VII
Errors

TODO: fill this section in later using Wave as a reference.