BCP/ConcurrenTree Basics

July 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
ConcurrenTree Model

ConcurrenTree is a concurrent text technology that puts live text editing in everybody's hands with its simplicity, especially compared to competitors like Operational Transform. CTree code is easy to write, easy to port, and doesn't have algorithmic ambiguity.

1 Trees

Trees are a fairly simple data structure at the core of ConcurrenTree. Just about every operation you can perform on a CTree document will traverse and/or modify trees, so it's good to get a nice solid understanding of them before rushing into the rest of this documentation. I mean, you can if you want, but you should learn to walk before you run, grasshoppa.

A tree consists (most importantly) of an immutable string and a boolean array called the "deletions array." There's a boolean in the array for every character marking whether it's deleted or not.[1]

The tree has a few more properties we have to worry about, though. The next biggest one (importance-wise, I mean) is the tree's children. The most conceptually accurate way to do this is as an array of string-key dictionaries, but there's nothing stopping you from finding more space- or memory-efficient ways to do it. What you're trying to model here is the fact that any arbitrary

[1]One might ask why we don't just, you know, actually delete characters instead of having an immutable string. Good question, and there's a lot of interesting theory behind that - but it's mostly beyond the scope of this document. The headlines: we need to maintain our index positions in the face of deletions, storing a char is basically as efficient as marking it deleted anyways, and it allows for consistent hashing throughout the lifetime of the tree.

number of child trees can be attached to any index position[2] in the parent tree, and it can be found again given the hash of its original value and the index it's attached to.

Using these properties, we have a working plaintext model for which we can write algorithms that insert text (by adding children), delete text (by applying deletions), "export" the tree to a single resultant string, and convert a position in the resultant string to a position in the tree or its children. Expect a lot of recursion when we get to the algorithm section. If you don't like recursion, then tough, it's a fundamental programming concept and the right tool for this kind of job.

Finally, we have markers. Markers handle formatting in a way that gets out of the way of regular text (important for concurrently edited documents) and makes a very simple kind of sense. Things get a bit nastier and dirtier under the hood to maintain eventual consistency though, which is one of the key properties of ConcurrenTree. So we'll get into the inner workings a bit later, but for now, here's markers in a nutshell: a marker has a type, a value, and a position, and it applies to everything to the right of it in the resultant text, at least until another marker of the same type shows up.

2 Addresses

There are two types of addresses, *absolute* and *shorthand*. Absolute addresses are conceptually simpler, so you can learn those now, build on that knowledge, and then learn the shorthand variation later on when you learn about eras.

The basic idea of an address is not as complicated as it sounds. We want to be able to reference any node in a complex tree with a string. And we can! The best part is we borrow a little syntax from filesystem syntax to make it intuitive and familiar.

0:a98321/4:393bf8/1:de73c0

As you might have guessed from our example, each node is referenced with an index:hash combination, the / denotes a parent-child relationship, and we can see that a tree with a hash of 393bf8 is a child of root node 0:a98321 at index 4.

Here's how it works. Any node can find any of its children given an index and a hash. A recursive algorithm can pass a request down a tree and bring the answer to the top by having each node pop a child off the front of it, so each node gets an address to resolve that's a little shorter, until finally the correct node is found and passed back up with return statements.

3 Instructions and Operations

One thing that CTree borrows from its *very* distant ancestor OT is the notion of instructions and operations. However, some neat properties of CTree make

[2]An index position represents a space between characters or on either end of the full string. So it can be any integer between (and including) 0 and n for a string of length n.

them a lot simpler. Operations never need to be “tranformed”, they can be applied on any machine in any state or queued until their dependencies are satisfied. It’s even okay to do stuff more than once, since everything is designed to be idempotent - multiple applications of the same operation have the same effect as one.

But I’m getting ahead of myself. If you’re reading this, you’re probably new to the party. So don’t panic, this stuff is simpler than it sounds.

Instructions are individual actions you can on a tree given an address and a few other parameters. Basically, the kind of stuff that takes one function call. Insert a child to a node, delete characters from a node, marker stuff, etc. So there’s not too many types of instructions to worry about, and we’ll get into them individually with sample algorithms soon enough.

An operation is a sequence of instructions. You can write functions that analyze the instructions to figure out the dependencies of an operation (what nodes have to be in a tree before the operation makes sense/can be applied), you can operation merging functions, but ultimately, an operation is a vector. Nothing more, nothing less.

So, you might be wondering why we even bothered to use terminology for this stuff. The reason is serialization. Instructions and operations can be serialized to JSON in a consistent fashion that allows them to be transferred between machines and applications, and quite importantly, can be hash-checked to see if it’s already been applied.

But no worries if it’s got overlap with stuff you’ve already applied to your tree! A few CPU cycles wasted, but no harm done. Everything will merge together correctly and not increase memory usage with the overlap.

There’s also a built-in transactional benefit to operations, letting you bundle together instructions that should apply more or less at once and have semantic relatedness, allowing you a little control in a world of unpredictable application order, personal view, and eventual consistency. There’s no real safety benefit to transactions in CTree though, so they are not nearly as useful here as in other situations.

4 Eras

ConcurrenTree’s magic “secret” formula is that on a conceptual level, information cannot be deleted. It can only increase or overwrite in fairly non-destructive ways.

Yeah, that look of horror and the memory usage panics running through your head are normal. We had ’em too during the development process. It sounds like the data storage requirements of a single document would eventually spiral out of control, especially in live-typing situations where you very quickly get deeply-structured trees of small nodes with a scary overhead-to-data ratio.

Well, the good news is that we fixed it. The bad news is that the fix is one of the more difficult concepts in ConcurrenTree. It’s called an “era.”

Every tree heirarchy can be thought of as being in layers. Let's use directories as an example. / is at layer 0, */dev* and */bin* are on layer 2, and */var/lib/alsa* is on the same layer as */home/philip/documents* even though they're distantly diverged in the tree.

In the same way, you can think of CTree documents as having layers (although unlike our directory example, layer zero can have as many nodes as you want, there is no top-level node). For every 16 layers[3] in a document, there's an era. Layer 15 is the last layer in era 0, layer 16 is the first in era 1. Or, if you like hexadecimal, layer 0F is the last layer in era 0, and layer 10 is the first in era 1.

So, we've organized our layers into bigger segments, but we haven't solved any memory crises quite yet. But we're getting there. You see, at each era border, we can collapse all previous eras into a single "virtual node" called a flat, allowing us to treat unlimited structural data like a single node.

This voodoo is accomplished with more or less the same algorithms used to convert a tree into a resultant string. This lets us scale *down* our memory usage. We can also scale it back *up* as needed if we get an operation that operates on nodes above our copy's flatline, by requesting that information from the hard drive cache or the web. Era algorithms let us plug in higher eras to our stack and move the flatline in either direction at will.

5 Shorthand Addresses

So now that you understand eras (or at least enough to nod your head and pretend you do), you can now understand shorthand addresses.

On deep trees (let's say, 80 layers deep), absolute addresses are just gonna be unmanageably large. With about 12 characters per layer minimum... that's 96 characters. Not so bad on its own, but addresses are used everywhere, and that's a lot of overhead, and it's assuming fairly short hashes.

The answer to this is a special type of address that lets us shortcut to certain nodes - say, any node on the first layer of any era. It's not too complicated to do this.

The shortcut format is "#738bbe95", where the number is the hash of it's shortcut address from the next shortcut up the tree. For era 1 shortcuts, this is just their absolute address, for era 2 it's the shortcut from their era 1 ancestor, etc. From there you can add on normal "/index:hash" segments to move further down the tree.

This is a little more complicated than always using the absolute address, but it allows for gradual "forgetting" of ancient eras.

[3] Why 16? Well, it was an arbitrary decision. We decided it was a nice power of two about the right size for it. Theoretically, we could use any era size, but then you wouldn't have the nice symmetry with hexadecimal numbers (for example, era 0 starts at 00, era 1 starts at 10, era 2 at 20, and so on).

Part II
Algorithms

The ConcurrenTree model is designed so that you can figure out algorithms from scratch correctly without a reference, but it's still useful to have some examples, especially for stuff like markers. These algorithms to not do any sanity or validity checking, for the sake of simplicity, so while they may be easier to read than the source code in the repository, they're also a bit brittle, and for educational purposes only.

6 find

This resolves an address to a node object in memory using recursion. It's the foundation of other document-level algorithms (which are really just shortcuts to tree-level actions in most cases).

```
tree::resolve(string address) {
    // isolate top-level piece
    string top = everything to the left of the first "/";
    address = everything after it;
    if (address =="") return this;
    tree next;
    if top[0] == "#" {
        // shortcut address
        next = this.document.shortcuts[top];
    } else {
        int index = top.split(":")[0];
        string child = top.split(":")[1];
        next = this.children[index][child];
    }
    return next.resolve(address);
}
```

7 trace

The trace algorithm takes a position in a resultant string and converts it into an (address, index) pair representing an index position in a specific node. This allows you to do all your other operations in result-space, for example, inserting text to a position in the displayed resultant text.

While it's one of the most fundamentally useful algorithms, it can also be one of the most complicated in strongly typed languages. In JavaScript and

Python, we like to rely on the return type to imply results as well as the value - did we find the end result, or should we just shave off some of the total we're trying to reach - which handles things very efficiently in terms of computation. This is a big deal for functions called as much as trace, especially when the language is interpreted.

If you pass the target total as an int pointer, and return a null pointer when the target is not among the current node's descendants, you can get around this in an even more efficient way, but it requires a pointer-aware language (or at least one where you can hack it to pass an int by reference). This is what we give an example of below. Most of this carries over quite well to a return-type-based approach like we talked about earlier, most notably the loop design.

```
tree::trace(int* target) {

    for (int i in this.children) {

        for (string x in this.children[i]) {
            dict result = this.children[i][x].trace(target);
            if (result != null) {
                result["address"] = string(i) + ":" +
                x + "/" + result[address];
                return result;
            }
        }
        if (i<this.children.length) {
            /* remember, there's more indexes than
            characters, we have to be careful not to
            access characters that don't exist! */
            if (this.deleted[i]) &target–;
            if (&target ==0) return {"tree":this,"pos":i,
            "address":""};
        }
    }

}
```

8 insert

By now you can probably figure this out from the context of the previous algorithms, but here we go in case you're skipping around or in over your head. The tree-level insert function is brilliantly simple, although it is omitting the constructor for the subtree from a string.

```
tree::insert(int pos, tree* subtree) {
        // how you want to do pointers and
        // references is up to you, here.
        this.children[pos][subtree->hash] = subtree;
        // optional step if you want nodes to
        // be parent-aware for any reason
        subtree.parent = &this;
    }
```

The document-level insert algorithm on the other hand is fairly simple as well, but in a different way. Instead of directly affecting the tree, document-level functions return operations, which allows you to easily translate from display actions *and* keep track of local and foreign operations.

```
    document::insert(int pos, string contents) {
        dict trace_results = this.root.trace(pos);
        return new operation([
            new insertion(
                trace_results["pos"],
                trace_results["address"]
                );
        ]);
    }
```

9 delete

Again, we're gonna show off multiple algorithms, like we did for insert, starting with our beloved tree.

```
    tree::delete(int pos) {
        self.deletions[pos] = true;
    }
```

Yep. That was easy. Now the document-level algorithm:

```
    document::delete(int pos) {
        dict trace_results = this.root.trace(pos);
        return new operation([
            new deletion(
                trace_results["pos"],
                trace_results["address"]
                );
        ]);
    }
```

Basically document::insert copied, pasted, and very slightly modified. In fact, if you want to generalize code like this into another function and be a better code reuser, you're highly welcome to, as it'll make your life a bit easier should you ever need to maintain these parts of the codebase.

Part III
BCP Basics

Basic ConcurrenTree Protocol is a protocol specification for transferring and syncronizing CTree documents. The syntax is not too hard to learn - it's just a bidirectional stream of JSON objects called "messages." Each message always has a required property called "type", and messages are separated by null bytes.

BCP can be used over any stream where packet order can be guaranteed, for example TCP sockets. For browser-based CTree systems, one might find a lot of usefulness with Socket.IO, Node.js, and the Javascript library provided in the ConcurrenTree/Orchard repository.

If you want to write your own BCP interpreter in your language of choice (and good for you, you awesome person), you're gonna find the BCP Message Set reference document in the repository *very* handy. It details every message type with examples and everything, and even if you're just going to use an existing BCP library, you might find it interesting reading to see how things work under the hood.

10 Documents

A document is a tree 0 characters long and with a name. Names are arbitrary strings, so any name scheme is possible, though anything more specific is yet to be determined.

Because each document is, alone, identical in its simplicity, a remote end can safely assume a blank document when you select an unknown docname from them, allowing you to add children to the document. If you give it any ops with dependencies it doesn't have, it will start trying to synchronize tree state with you, and find other peers to sync with.

Care should be taken with docnames. The network is your namespace, and unless you've added some sort of collision prevention in your name scheme (like another layer of namespacing or two), you will probably eventually find the problems involved with two separate, unrelated documents by the same name. When the machines with these two documents realize they can sync with each other, they will, resulting in an amalgamated result document which will usually be the contents of one document followed by the contents of the other, with the order determined by the hashes of the low-level trees.

So, you might be wondering why a 404 error would even still be relevant to this system? Well, BCP is designed to be flexible, universally usable for any CTree model concurrency software. And in some scenarios, a 404 error might be preferable to our idea of "standard behavior." So it stays in for 3rd party use, but we don't use it.

11 Peer State

Maintaining a connection in BCP does require tracking a minimalist set of state for each end of the connection. For each end of the connection, you need to store the selected docname, subscriptions, any operations you advertise to the remote, a little authorization state, and optionally, the op hashes you know have been applied at that end.

The auth state is surprisingly simple to store. You simply need a dictionary keyed on docname, containing random strings, generated as needed during auth dances. Store entries when you need to remember the random source of an MEL, free the memory after you check the appropriate *authmd* response for your original source (whether it validates or not).