PubCrawl Project Proposal

COMS W4115

Matt Dean, Sireesh Gururaja, Kevin Mangan, Alden Quimby
mtd2121, skg2123, kmm2256, adq2101

September 25, 2013

1 Description

PubCrawl is a distributed systems programming language with a focus on list operations. Given some number of networked slave computers, it will automatically split work up among them, requiring the developer only to specify whether a function should distribute work or not. To create a robust platform, we will also implement redundant processing, so that processed data is not lost in the case of an unstable network.

To distribute work, the developer will specify the IP addresses of the slave computers, which will be running a slave server utility. When the output program is run, it sends compiled and serialized functions to the remote servers. It then passes data to each slave computer as the functions are called. This is blocking - the program does not continue until all the results have been received.

2 Proposed Uses

Pubcrawl helps you distribute your computationally extensive or network related problems across multiple machines in a cluster. Those machines can chip away at their portion of the problem and then send results back to the master machine. A simple analogy for this type of problem would be counting the number of books in the library. It would be much faster to have two people counting (one person counts the even numbered shelves, and the other counting the odd shelves). When finished, they just combine their numbers to produce the result.

One concrete use case is counting the appearance of a word in a large set of HTML pages. With PubCrawl, you would easily be able to have each node fetch certain URL's, count the number of word appearences, and return them to the master for further computation.

3 Syntax

3.1 Comments

Comments will be the same as in java. It's simple and widely understood.

```
// single line comments start
/*
        multi-line comments
*/
/*
        /* this breaks */
        // this, however, works
*/
/* this will break too /* */
```

3.2 Variable declaration

Variables are defined via classic imperative programming standards. Whatever name is on the left hand side is given the value of the result of the right hand side. And equals sign splits the two. There's no keyword "var" or type declaration. Types are inferred and variables can be overwritten, much like javascript.

3.3 Types

Strings will use single quotes, and single quotes only!

```
name = 'Alden';
hiThere = 'Hi, my name is {{name}}'; // built in formatting
hereisasinglequote = '\''; // backslash escape
```

Numbers will be... well numbers.

```
n1 = 4;
n2 = 5.0; //no special "floats", everything's just a number
```

Unlike in a language such as javascript, only three things make a correctly formatted boolean expression.

```
b1 = true;
b2 = false;
b3 = null; //will resolve to false
```

pubCrawl is object oriented and objects will strictly follow JSON syntax. We find JSON syntax elegant plus it will allow for seamless network integration.

```
o1 = {};
o2 = {
        p1: 'hey',
        'p2': {
                p3: 7
        },
        myFunc: (x,y) -> x + y
};
```

Object access is similar to javascript, if not identical.

```
thisIsHey = o2.p2;
alsoHey = o2['p2']; // must be a string, o2[7] is invalid

// allows for cases where you must access via "[]"
weirdObj = {
        '72': 'hey',
    'c.o.o.l': 'dude'
};
weirdObj.72; // fails
weirdObj['72']; // 'hey'
weirdObj.c.o.o.l; // fails
weirdObj['c.o.o.l.']; // 'dude'

// keys is also available to all objects, returns strings
k = o2.keys; // ['p1', 'p2', 'myFunc'];
```

Lists? Arrays? It's all just a collection to pubCrawl. They're a simple, easy to use structure with random access, add and removal, and subsets. In addition, strings are handled like a list of single string characters, this allows list functions to work for strings. "ABC" = ["A","B","C"]

```
// collection creation
l1 = [];
l2 = [1,2,3,4,5,6];

// access
five = l2[4];
oneTwoThree = l2[0:2];
copyOfL2 = l2[:];

// modification
oneTwoThree.add(4); // now [1,2,3,4]
oneTwoThree.remove(2); // now [1,3,4]

mystring = 'ABCDEFG';
DEF = mystring[3:5];
```

3.4 Control Flow

pubCrawl includes basic control flow. Notably we use "elif" instead of "else if" but otherwise it's just like java (but without those annoying do-whiles).

```
// conditionals (else is not required)
if (bool) {
// do stuff here
}
elif (otherBool) {
// more stuff here
}
else {
// other stuff
}

// foreach loops
for (item in list) {

}

// for loops
for (i = 0; i < 10; i++) {

}

// while loops
while (bool) {

}
```

3.5 Functions

Functions are a big part of what makes pubCrawl great. We have super simple function declaration as well as some syntactic sugar to keep your code clean and easy to read. Taking a chapter out of the functional programming book, all functions must return a result since everything is pass by value. Additionally, functions are first class objects and can be passed as arguments to other functions. This will come in handy later on.

```
// declaration
add(x,y) -> x+y; //notice no brackets = no return statement
square(num) -> {
        if (true) {
                return num*num;
        }
        return 7; // return statement REQUIRED if braces
} // no semi-colon needed to end function if braces

// calling
seven = add(5,2);

// all functions can also be called as extensions on the first argument
alsoSeven = 5.add(2);
```

3.6 IO

pubCrawl provides some simple IO. Standard in and out, file read and write, as well as simple web downloading is all supported.

```
// read to EOF
input = read(); // stdin
input = read('path/to/file.txt'); //returns collection of lines

// write
print('Hello World'); // stdout
print('Hello World', 'path/to/file.txt');

//download
download('http://www.google.com'); //returns a string
download('http://images.mywebsite.com/myimage.png');//returns bytes
```

3.7 Distribution

The true magic of pubCrawl is in one, special function called "distribute". This function is the equivalent of a "map" function, however it splits the work across any slaves that are available. Distribute returns a special object that holds some information about the node, if there was an error, as well as the input and output objects. This allows the programmer to handle the results however (s)he wants. Distribute is most elegantly called as an extension function on any list as seen in the example below.

```
urls = ['google.com', 'apple.com', 'amazon.com'];

// getPageTitle is defined somewhere else
// it downloads the url and finds the title on the page
results = urls.distribute(x -> getPageTitle(x));

// results is a collection of special objects:
/*
{
  'ip': '129.324.1.1', // IP address of slave where task ran
  'error': '', // error message if something failed
  'input': 'amazon.com', // input object for task
  'output': 'Amazon.com: Online Shopping for...' // result of task
}
*/
```

3.8 Utilites

pubCrawl will come preloaded with some basic utilites to help programmers get off the ground running. Since pubCrawl is all about working with collections and data, we're including map, filter, and any functions. To aid with string manipulation, we'll also provide a find function.

```
// part of a standard library to make list processing easier

// filtering
filtered = results.where(x -> x % 2 == 0);
alsoFiltered = results.where(x -> {
        return x % 2 == 0;
});

// mapping
mapped = results.map(x -> x.name);
alsoMapped = results.map(x -> {
        if (x == 5) {
                return 9;
        } else {
                return 8;
        }
});

//find
longString = 'Today was a great day!';
locationWas = longString.find('was'); //returns 6
locationHey = longString.find('hey'); //returns -1, aka not found
```

4 Example Programs

4.1 Bubble Sort

Just a simple algorithm example.

```
bubble(values) -> {
        swapped = true;
        while(swapped) {
                swapped = false;
                for(var i = 0; i < values.length-1; i++) {
                        if (values[i] > values[i+1]) {
                                var temp = values[i];
                                values[i] = values[i+1];
                                values[i+1] = temp;
                                swapped = true;
                          }
                }
        }
        return values;
}

mylist = [7, 4, 5, 2, 9, 1];
mylist = mylist.bubble();
print(mylist); // 1, 2, 4, 5, 7, 9
```

4.2 Downloading Xkcd Comics

This time we're using the cluster to help us out! Let's download the first 100 comics from xkcd.com.

```
getComic (id) -> {
        wholeSite = download('http://xkcd.com/' + id);
        imgChunk = wholeSite.split('<')
                .where(x->x.contains('imgs.xkcd.com/comics'));
        if(imgChunk.length!=1){
                return null; //couldn't find a picture
        }
        else{
                httpStart = imgChunk.find('http');
                jpgEnd = imgChunk.find('jpg') + 3;
                return download(imgChunk[httpStart:jpgEnd]);
        }
}

// The real magic happens here, each machin in the cluster splits the
// load of the getcomic function across numbers 1-100
pics = range(1, 100).distribute(x -> getComic(x));

// save all images to disk
for(result in pics) {
        print(result.output, 'xkcdimg' + result.input + '.jpg');
}
```

4.3 Distribute some tough math

What if we had a super cluster at our disposal? We could do tricky and time consuming tasks such as finding the prime factorization of many numbers. We're using a very basic algorithm in this case, but prime factorization is a resource consuming process nonetheless.

```
//a simple prime sieve
getPrimes (max) -> {
        myPrimes = [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59
,61,67,71,73,79,83,89,97,101,103,107,109,113,127,131,137,139
,149,151,157,163,167,173,179,181,191,193,197,199,211,223,227
,229,233,239,241,251,257,263,269,271,277,281,283,293,307,311
,313,317,331,337,347,349,353,359,367,373,379,383,389,397,401
,409,419,421,431,433,439,443,449,457,461,463,467,479,487,491,499];
        return myPrimes.where(x->x < max); //perfect filter example
}

//basic prime factoring algorithm
trial_division(n) -> {
    if(n == 1) {
            return [1];
    }
    primes = getPrimes(n/2 + 1);
    prime_factors = [];
    for(p in primes) {
        while (n % p == 0) {
            prime_factors.add(p);
            n = n/p;
        }
    }
    if (n > 1) {
            prime_factors.add(n);
    }
    return prime_factors;
}

// prime factorize every number up to 1000
range(0, 1000).distribute(trial_division)
              .map(x -> print(x.input + ': ' + x.output));
```