Managing Distributed Workloads

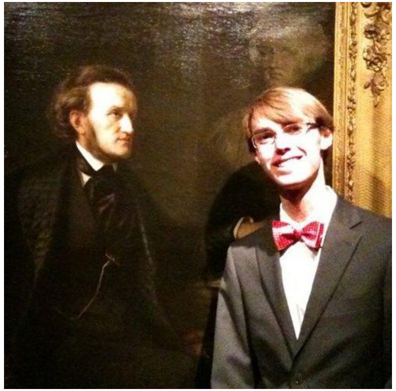
Benjamin Hanser
Miranda Li
Mengdi Lin

Language overview

M/s is language for implementing a distributed system

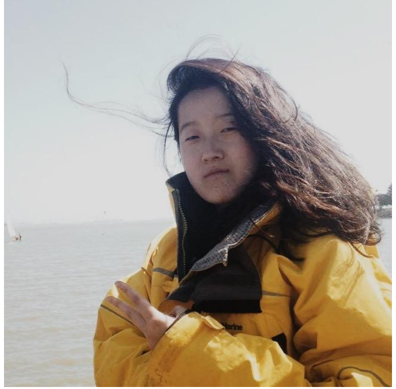
- A master server distributes work across slave nodes
- User defines a master (main) function, and jobs that can be run on slaves
- Hides messy socket handling, threading, and network packet serialization/deserialization for job inputs and outputs from the user!
- Also provides automatic garbage collection; vectors and structs; primitives; string; the typical binary and unary operators; control flow; printing

About the team



Benjamin Hanser
* System architect
* x86-man
* Bears resemblance to Wagner… (!?)
* Slave #1

Mengdi Lin
* Language guru
* Actual life guru
* Loves bubble tea
* regrets
* regrets
* Slave #2



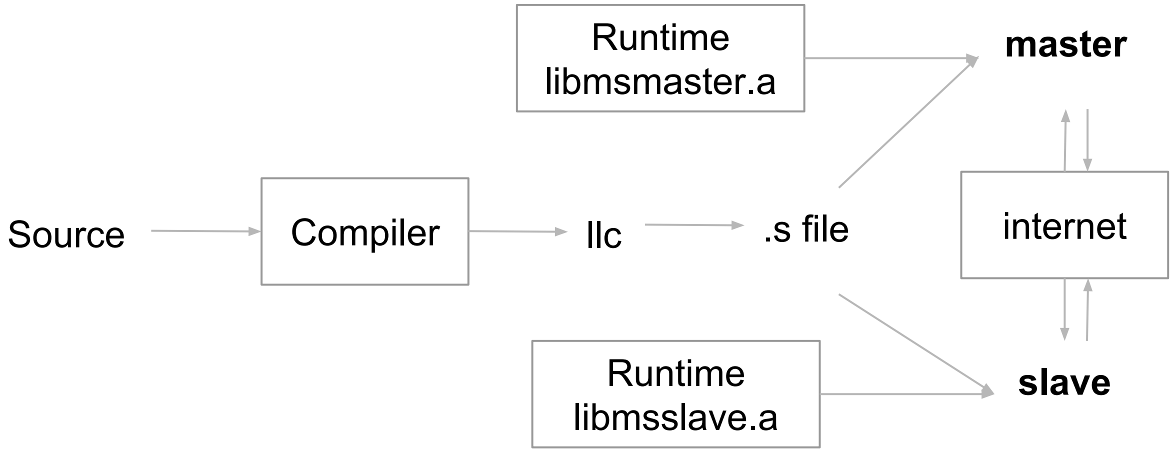
Miranda Li
* Team's faaavorite manager + tester
* Shift/reduce "guru"
* Slave #3

Stephen Edwards
* "TA Advisor"
* Talks about us in class
* Promised us an A+ at senior dinner, though perhaps doesn't remember…
* Our one true Master

Key features

- **`job`**
  - Define jobs as functions: **`job`** `int f(int a, int b) = { return 1 };`
  - Reference a running job: **`job`**`<int> j =` **`remote`** `gcd(2, 3);`
  - **`get`** result of job, **`cancel`** a job
  - Access job states: **`running`** (includes pending), **`finished`**, **`failed`**
- **`remote`**
  - Runs a job remotely, on a slave instance
- **`vector`**
  - C++-like vectors; **`vector<int>`** a; a::2; a[0] == 2
  - **`string`** = vector<char>
- **`struct`**
  - C-like structs; **`struct s`** `{int x; vector<int> v}; struct s a; a->x = 2;`

Compiler - Runtime interface



Runtime implementation

- Runtime manages running jobs and takes care of network operations
    - Written in C - compiles to two static libraries, libmsmaster.a and libmsslave.a
    - Link .s file from llc against each library to produce master and slave binaries
- Master runtime
    - Provides a main function that calls the compiled M/s code's "master" function
    - Exposes start_job and reap_job handles, which are called by compiled M/s code
    - One read thread and one write thread per socket
    - Shared job table belongs to all the sockets
        - Queue of jobs pending assignment
        - Stores return values of jobs before they are reaped
        - Restarts a job on a new slave if its current slave is disconnected
- Slave runtime
    - Listens to one socket, spins up a new thread for each job request received

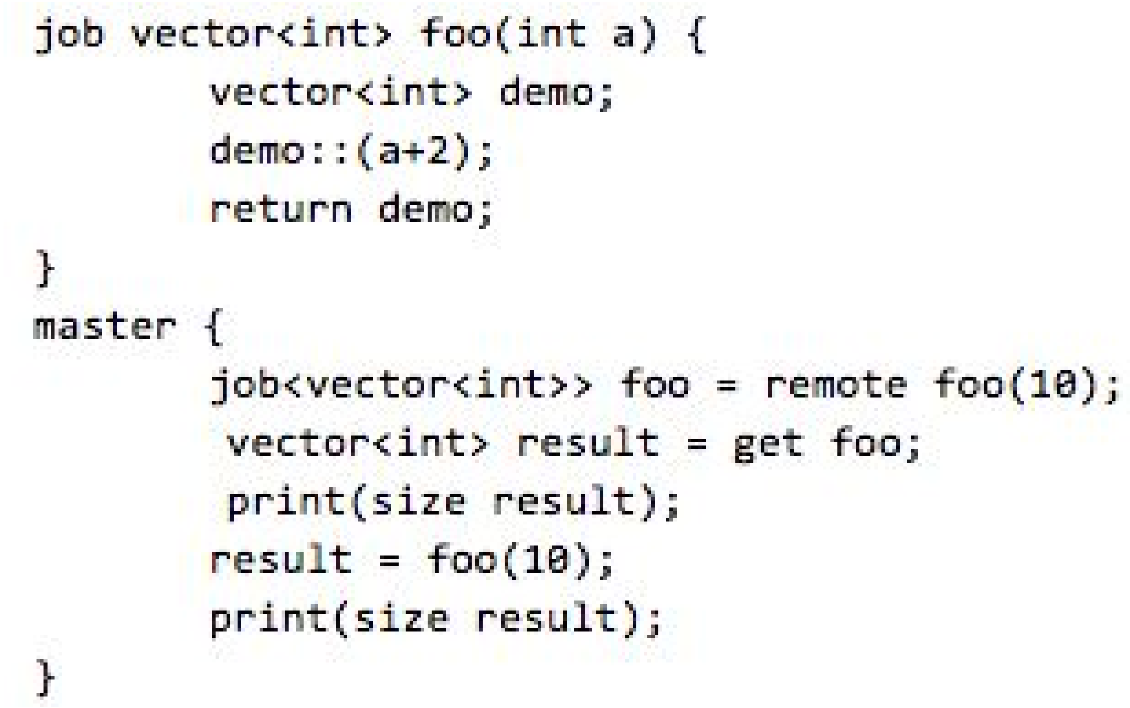
Protocol

- 12 byte header: [ordinal; jid; length]
  - ordinal is a positive integer representing the job function to be run
  - jid is a unique nonnegative integer created for each job - identifies the job’s return
- Data:
  - Each argument is serialized sequentially
  - Structs serialize each field sequentially
  - Vectors serialize the size (4 bytes) and then each element sequentially

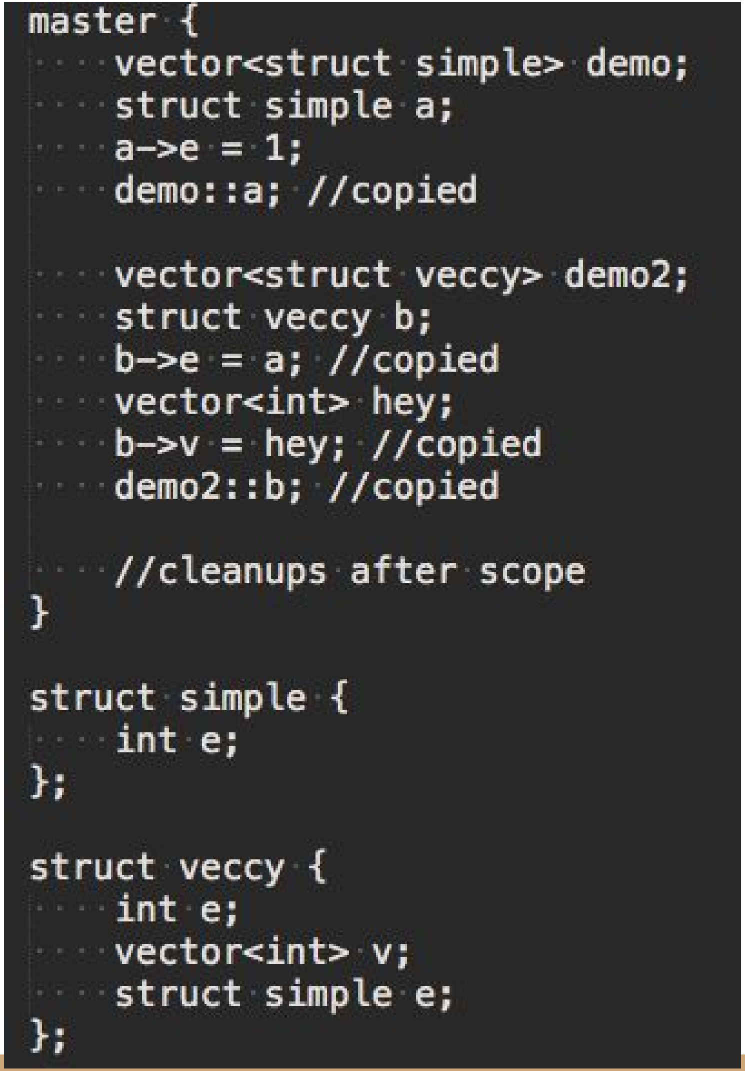
Program structure

```
master {
...
}
job int f(int a, int b) { ...}
struct s { int a; int b; }
```

Compiler implementation



```
job vector<int> foo(int a) {
        vector<int> demo;
        demo::(a+2);
        return demo;
}
master {
        job<vector<int>> foo = remote foo(10);
         vector<int> result = get foo;
         print(size result);
        result = foo(10);
        print(size result);
}
```



```
master {
    vector<struct simple> demo;
    struct simple a;
    a->e = 1;
    demo::a; //copied

    vector<struct veccy> demo2;
    struct veccy b;
    b->e = a; //copied
    vector<int> hey;
    b->v = hey; //copied
    demo2::b; //copied

    //cleanups after scope
}

struct simple {
    int e;
};

struct veccy {
    int e;
    vector<int> v;
    struct simple e;
};
```

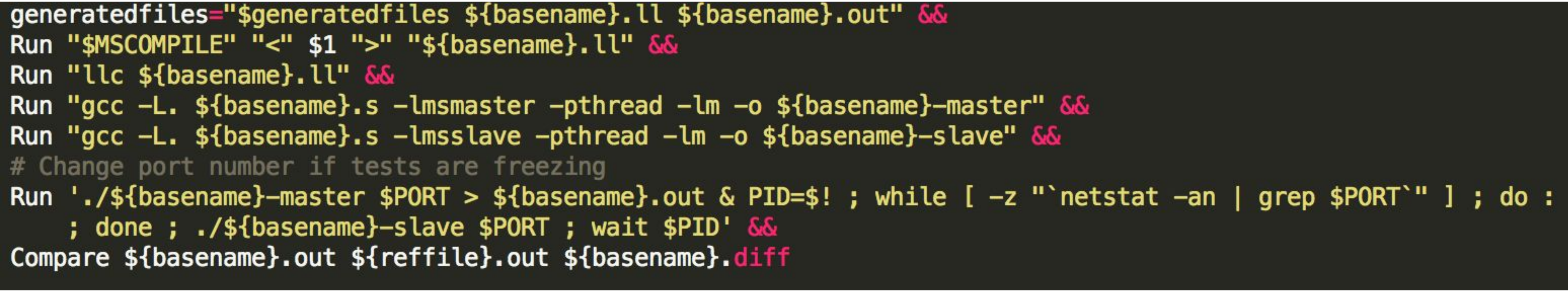
The rest of the compiler…

- …is probably exactly what you’d expect*!
- **Any questions?**

* Scan the input; parse it; make the AST; check semantics; generate code

Testing

Adapted testall.sh to automatically compile and run remote tests, starting master and slave processes:



```
generatedfiles="$generatedfiles ${basename}.ll ${basename}.out" &&
Run "$MSCOMPILE" "<" $1 ">" "${basename}.ll" &&
Run "llc ${basename}.ll" &&
Run "gcc -L. ${basename}.s -lmsmaster -pthread -lm -o ${basename}-master" &&
Run "gcc -L. ${basename}.s -lmsslave -pthread -lm -o ${basename}-slave" &&
# Change port number if tests are freezing
Run './${basename}-master $PORT > ${basename}.out & PID=$! ; while [ -z "`netstat -an | grep $PORT`" ] ; do :
    ; done ; ./${basename}-slave $PORT ; wait $PID' &&
Compare ${basename}.out ${reffile}.out ${basename}.diff
```

Testing

- Passing tests written as we created new features
- Fail tests written for every semant checking case
- Some examples:
  - Jobs: assignment, get, cancel, job states
  - Vector: creation, pushback, access, assignment
  - Structs: declaration, instantiation, field access, assignment
  - Vectors in structs and structs in vectors
  - Remote calls, memory freeing
  - Primitives, doubles, strings

```
[...]
-n test-remote-doubles...
OK
-n test-remote-int...
OK
-n test-remote-job-get...
OK
-n test-remote-job-states...
OK
-n test-remote-many-ints...
OK
-n test-remote-struct-serialize...
OK
-n test-remote-vector-serialize...
OK
-n test-string-concat...
OK
-n test-string1...
OK
-n test-string2...
OK
-n test-struct-field-copy...
OK
-n test-struct-in-vector...
OK
-n test-struct-nocopy...
OK
[...]
-n test-vector-args...
OK
-n test-vector-assign...
OK
-n test-vector-struct-copy-assign...
OK
-n test-vector-struct-copy-free...
OK
[...]
```

```
-n fail-assign-double...
OK
-n fail-assign-string...
OK
-n fail-assign-string1...
[...]
-n fail-func1...
OK
[...]
OK
-n fail-job-cancel...
OK
-n fail-job-get...
OK
-n fail-job-get2...
OK
-n fail-job-state1...
OK
-n fail-job-state2...
OK
OK
-n fail-remote1...
OK
-n fail-return1...
OK
-n fail-return2...
OK
-n fail-string-concat...
OK
-n fail-struct1...
OK
-n fail-struct2...
OK
[...]
OK
-n fail-vector...
OK
```

```
Example: test-struct-in-vector.ms
master
{
    vector<struct Books2> bookies;
    struct Books2 book;
    book->b->book_id = 99;
    bookies::book;
    struct Books2 outbook;
    outbook = bookies[0];
    print(outbook->b->book_id);
    print(bookies[0]->b->book_id);

    struct veccy vy;    vector<int> v;
    v::5;
    vy->v = v;
    v[0] = 6;
    vy->sz = 1;
    vector<int> vv;
    vv::778;
    vv = vy->v;
    print(vv[0]);
}
struct Books {
   int book_id;
   int d;
};
struct Books2 {
   int book_id;
   int d;
   struct Books b;
};
struct veccy {
    int sz;
    vector<int> v;
};
/* output:
99
99
5
*/
```

Lessens “lurnd”

Everythin’ was greaaat, and #noragrets*

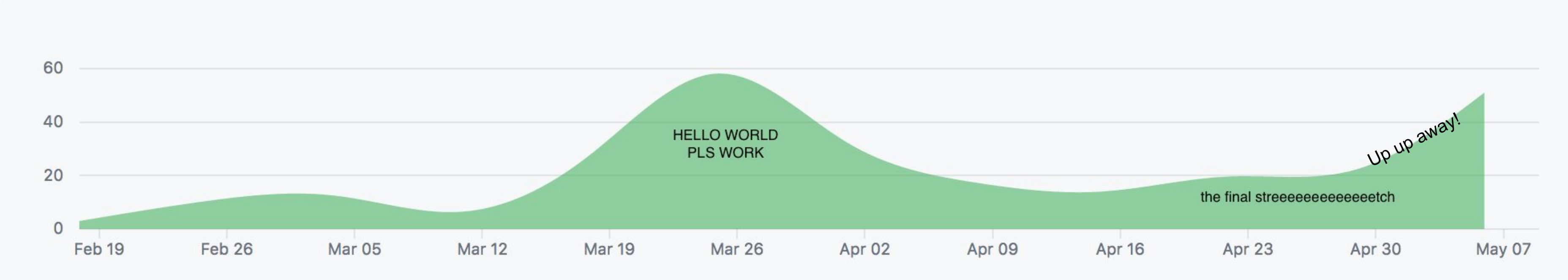
* Except...

GEP SEGFAULT ON ME WTF

WHY DID WE DECIDE TO IMPLEMENT MEM-SAFE VECTOR IN LLIR :(

Demo time!!

Project timeline


Feb 19
Feb 26
Mar 05
Mar 12
Mar 19
Mar 26
Apr 02
Apr 09
Apr 16
Apr 23
Apr 30
May 07
HELLO WORLD
PLS WORK
the final streeeeeeeeeeeeetch
Up up away!