HACKERMONTHLY Issue 46 March 2014

On Hacking

by Richard Stallman

Curator
Lim Cheng Soon

Contributors
First Round Capital
Richard Stallman
Nathan Wong
Mike Ash
Seth Brown
Rich Adams
Evan Miller
Jonathan E. Chen
Chad Fowler

Proofreaders
Emily Griffin
Sigmarie Soto

Illustrators
Jaime G. Wong
Joel Benjamin

Ebook Conversion
Ashish Kumar Jha

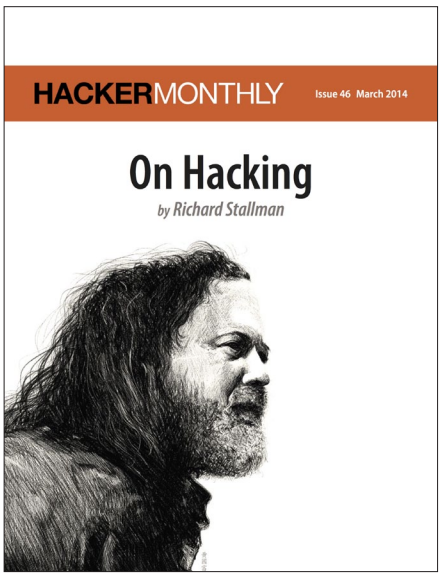
Printer
MagCloud

HACKER MONTHLY is the print magazine version of Hacker News — *news.ycombinator.com*, a social news website wildly popular among programmers and startup founders. The submission guidelines state that content can be "anything that gratifies one's intellectual curiosity." Every month, we select from the top voted articles on Hacker News and print them in magazine format.
For more, visit *hackermonthly.com*

Advertising
ads@hackermonthly.com

Contact
contact@hackermonthly.com

Published by
Netizens Media
46, Taylor Road,
11600 Penang,
Malaysia.



Cover Illustration: Jaime G. Wong

Contents

Drew Houston, illustrated by Joel Benjamin.

For links to Hacker News dicussions, visit *hackermonthly.com/issue-46*

How to Win as a First-Time Founder

A Drew Houston Manifesto

By FIRST ROUND CAPITAL

Illustration by Joel Benjamin.

In 2007, Drew Houston flew to San Francisco determined to find a co-founder for Dropbox. At the time, it was just him. No backers. No team. On a friend's advice, he walked into Y Combinator's offices unsolicited to talk to Paul Graham about finding the right person. It didn't go well.

"It wasn't a great experience, coming in unannounced," Houston recently told students in an exclusive Dorm Room Fund interview at MIT. "Getting into Y Combinator is like getting into a great school. So imagine having your two minutes with the dean of admissions and them coming away thinking you're an asshole. That plane ride back was the worst. No co-founder. Lower chance of getting into YC. I was panicked."

The good news is, early founders can turn things around. Soon after he thought it was all over, Houston teamed with fellow-MIT alum Arash Ferdowsi and made it into YC. Today, he's led Dropbox to nearly 200 million users — and the company's growing faster than ever before. This hasn't been a piece of cake, but Houston's rocky start did teach him to forge ahead and throw out assumptions that discourage many would-be founders. Looking back, he recommends six strategies that helped him cut through the fear, drown out the noise, and make it happen.

1 Start with a worthy problem.

Prospective entrepreneurs are primed to find problems. While he was still in college, Houston signed up to beta test an online game as it was being built. When he ran out of things to do, he started poking around under the hood, and he discovered a bunch of security vulnerabilities.

"So I started hacking around on the game, and ended up telling the developers, 'Hey guys, you have to do this and this…' They responded, 'Okay great, want to just do that for us?'" That's how Houston landed his first engineering gig. Dropbox was born out of a similar moment, when he simply got fed up with the lack of seamless storage solutions for his files.

But not every idea is bound to be a good one, or worth your time. After coming up with a cohort of aspiring founders (some successful, some not) and observing their various fates, Houston has devised a list to help new entrants choose their projects wisely:

- **It just pulls you.** This is the least scientific of his recommendations, but that gut feeling that a problem is critical and needs an answer shouldn't be overlooked. "Sometimes you just get this feeling — it's a compulsion or an obsession. You can't stop thinking about it. You just have to work on this thing," he says. "You need that hunger no matter what, because eventually the honeymoon period wears off. Somewhere between printing your business cards that say 'founder' on them and everything else you have to do, you realize, 'Oh, actually this is a ton of work.'"
- **You think it can go far.** "With something like Dropbox, it was immediately like, "Wow, this is literally something that anyone with an internet connection could use." Everyone needs something like this, they just don't realize it yet." Now, with the app approaching 200 million users, Houston already has his eyes fixed on a billion. "It's crazy that we live in a world where that's a totally reasonable thing to go after. But I look at all the things we can do, and the magnitude of the opportunity in front of us is so clear."

- **It optimizes for learning.** It's always smart to go where you'll have the ability to learn the most. Go where people are smart and fierce, because wherever you go, you're bound to learn through osmosis. "If you join a company, work with world-class people because that's the fastest way to learn how to do things. If you start your own thing, you can learn a lot really fast from doing things wrong. Ask yourself, 'Where can I find an environment where I can work really hard and learn the most?'"

2 Own Being a Beginner.

In his book Outliers, Malcolm Gladwell suggests that it takes 10,000 hours before you can truly become an expert at anything. Given the immense challenge of starting a company, one might think that founders need to be vastly experienced. But Houston disagrees. He's got some powerful evidence, too: Google, Apple, Dell and Facebook — all unicorns, all started by first-timers or people who failed on the first try.

"A lot of times it's an asset to not know everything about everything," Houston says. "As you advance in your career, you feel like you know so much about the world and what's possible. Then you have this mental model about how things work that gets less and less flexible. You can get stuck."

His favorite example came early on when the first articles were being written about his company. He remembers one quote precisely:

> *"Fortunately, the Dropbox founders are too stupid to know everyone's already tried this."*

"A lot of really great, innovative things have happened when people just didn't know it wasn't supposed to be possible," Houston says.

It's important to not underestimate your ability to learn on the fly. "Everything can seem so mystifying before you start," he says. "But when you look behind the curtain at how some of these huge companies were built, it wasn't a lot of magic. It's people iteratively trying to make reasonable decisions and surround themselves with the smartest people they can."

3 Assume Nothing & the First Mover Disadvantage

At the time Houston got the idea for Dropbox, people thought the problem was already solved. They had email attachments and thumb drives — and for the power users, external hard drives. What more did they need? Even the forward thinkers would have guessed a solution would come from Google or Microsoft.

"People make basic assumptions based on what they have now. But you have to ask yourself, is this really what people are going to be doing in five years?" he says. "Very few people ask themselves what they would actually want instead if they could wave a magic wand. What if there could be this magic folder that you could access from anywhere and never need to back up?"

Something a lot of entrepreneurs assume is that they have to be first to market in order to win in a category. But when you look at the breakout success stories, this is almost never the case. Google was preceded by Yahoo, Alta Vista, Ask Jeeves, and 100 other little search engines. Facebook entered stage left and slaughtered both MySpace and Friendster.

"The fact is that there's a problem with being first," Houston says. "When you do that, you create a market, and if you're too early, you essentially leave the door open behind you for someone to do it better. I actually don't think it matters how early or late you are as long as you hit critical mass."

When Dropbox was getting off the ground in 2007, there were hundreds of small storage companies. It was almost a cliché, the way that many people believe mobile photo sharing is a cliché now, he says. "The important thing was, I would keep asking people if they used any one of these hundred options, and they all said no. These are my favorite problems to solve. You can't focus on what everyone else is doing — it has to be about what's really broken and what you can do to fix it."

Even today, Houston's reminded all the time that he has 400 people against Google's 40,000. It's daunting, but he has to shrug it off. In the end, tech is about disruption, and there's plenty of proof that numbers of users, or employees or dollars doesn't always make the difference.

> *"Small teams can take on bigger companies because of their focus and speed. That's also what makes it fun."*

This kind of challenge can seem like too big a gamble for many people who might otherwise start companies. With odds so heavily in favor of the Goliaths, chances for success seem slim, but Houston does his best to de-risk the idea for aspiring entrepreneurs.

"People assume — and misunderstand — that it's risky to join a startup or start their own company, but you have to know this is ridiculous," he says. "Even if it doesn't work out, the experience is so valuable to so many employers that your worst case scenario is, "Ok, so that was a bust, I"ll get a six-figure job at whatever company.' Risk is this outmoded idea — your parents might not understand that, but taking these types of risks doesn't have a downside."

4 Build a knowledge machine. For Houston, learning new things became an addiction — one he actually systematized.

"I was living in Boston, working for a startup during the summer, living in my fraternity house. But every weekend, I would take this folding chair up to the roof with all these books I got on Amazon. I would just sit there and read all of them. I would spend the whole weekend just reading, reading, reading."

His process wasn't complicated, but he did keep a list of target topics in his head. "I'd be like, alright, I don't know anything about sales. So I would search for sales on Amazon, get the three top-rated books and just go at it. I did that for marketing, finance, product, and engineering. If there was one thing that was really important for me, that was it."

If you've never started a company, or worked at a smaller company, you'll run into a vertical learning curve, Houston says. There's no way to know everything you need to from the start, so you need to a) gain as much knowledge as you can as fast as you can, and b) plan ahead to learn what you'll need months down the line. You have to be prepared for a never-ending conveyor belt of challenges.

"You have to adopt a mindset that says, "Okay, in three months, I"ll need to know all this stuff, and then in six months there's going to be a whole other set of things to know — again in a year, in five years.' The tools will change, the knowledge will change, the worries will change."

> *"You have to get good at preparing yourself to understand what's on the horizon."*

This is especially important for skills and habits that you can't internalize overnight. "You're not going to become a great manager overnight. You're not going to become a great public speaker or figure out how to raise money," he says. "These are the things you want to start the clock on as early as possible."

As a founder, this goes for both you and your employees. This can be a huge advantage when it comes to recruiting the best talent, too. One young engineer comes to mind for Houston, who was swayed by the opportunity to be thrown into the deep end right away.

"We had this enormous infrastructure project where we were spending millions of dollars and he was in charge of it — and he was like 20 at the time. He just wouldn't have gotten that opportunity if he had been employee 20,000 at Google or something," he says. "This engineer even said to him at some point, "Dropbox let me do things that I wasn"t ready for.'"

This chance, to work on real things and move the needle at a company serving millions, is rare and extremely valuable. "I look at the interns we have at Dropbox, and they're shipping real stuff every day," Houston says. "In contrast, I had a friend who worked at Microsoft for a summer, and he spent the entire time working on the back button on Internet Explorer."

The upshot: Making learning central to your company's culture pays serious dividends.

5 Be resourceful. Fast.

Houston may have gotten off on the wrong foot with Y Combinator, but he was able to turn it around just as fast with limited tools.

"It was one of those things where it was a couple weeks before the deadline, and I just realized I had no choice. I had to write this application," he says. "I was already at a disadvantage because I was a single founder and YC really wants co-founders. But I said to hell with it, I'll just do it anyway. So I made a video."

This demo video is now part of Dropbox mythology. Not only did it catch fire on Hacker News and Reddit, it also convinced YC partner Trevor. The key was Houston knew his audience. "I was part of that audience, so I made the video that would get me excited about Dropbox. The production value wasn't great. It was just me sitting in my bedroom at 3 a.m., but I knew what to say." It worked — he got an email from Paul Graham saying there was interest, but to go any further, he'd need to find a co-founder.

He approached this task with the same attitude as his YC application. He knew what he needed. He went after it, and he moved quicker than he felt comfortable with. That's the pace you have to get used to when you're involved with a startup, he says. Finding a co-founder on this timetable can be one of the most daunting things an entrepreneur can do.

> *"It was sort of like them telling me I needed to find someone to marry in two weeks."*

Luckily, the video came in handy here, too. By the time he met with Arash Ferdowsi, a friend of a friend at MIT, his future CTO had already seen the demo and was interested.

"We went to the coffee house at the student center because that's the only thing we could do," Houston recalls. "At the time, I was just like, this kid seems pretty smart. I can't say it was this careful process where I had 19 things I was looking for, but he seemed intelligent and cool, and we spent a good two hours together talking. At the end, he said "Okay, yeah, I'll drop out next week.'"

Now that he's had time to reflect, he realizes how lucky he got with Ferdowsi, and he has some advice for young entrepreneurs looking for their other halves. "The most important thing is whether you respect this person, whether you trust them. Are they someone that you can see yourself being in the trenches with for a long time, because you're going to see them more than your spouse or your significant other."

6 Don't lose your North Star.

Inevitably companies evolve as they grow, but Houston knows the value of keeping a higher purpose front and center. This is especially critical for Dropbox right now as it adds hundreds of new employees and expands more and more into enterprise software.

Many of even the most successful startups in tech will say their culture evolved organically — that they're only just now starting to be intentional about it at 100 to 300 employees. Dropbox falls into this category. But Houston advocates for an earlier attack.

"When you're studying and getting your engineering degree, things like mission or values sound totally unnecessary," he says. "But then it turns out that you have to evolve from building this system of code to building a system of people. It's like updating your operating system. You have to adapt very quickly."

To keep this top of mind, you have to make the company's mission about something more than money or building great products. It has to be about the value created for users.

"Whole businesses are living out of Dropbox right now, big and small," he says. "That's something that's really valuable — the fact that we're helping employees be more productive, even at giant companies. IT departments and administrators have become an important audience for us, but at the same time we have to remember why we do what we do: We do it to make people happy."

"We get these emails from people that just blow us away," Houston says. "They say things like "I just used Dropbox to start a music festival" or "I made a movie" or "I started the company I"ve been dreaming about my whole life.' People tell us that Dropbox has completely changed how they work. And I think that's what's really exciting — being able to redefine how people collaborate. It's not just the why of what we do, it's also a huge market. ■

Drew Houston is the founder of Dropbox.

First Round Capital is a San Francisco-based venture capital firm focusing on seed funding for technology startups and creating a vibrant community of entrepreneurs working to change the world.

Reprinted with permission of First Round Review [firstround.com/review], a publication of First Round Capital.

On Hacking

By RICHARD STALLMAN

In June 2000, while visiting Korea, I did a fun hack that clearly illustrates the original and true meaning of the word "hacker".

I went to lunch with some GNU fans, and was sitting down to eat some tteokpaekki [1], when a waitress set down six chopsticks right in front of me. It occurred to me that perhaps these were meant for three people, but it was more amusing to imagine that I was supposed to use all six. I did not know any way to do that, so I realized that if I could come up with a way, it would be a hack. I started thinking. After a few seconds I had an idea.

First I used my left hand to put three chopsticks into my right hand. That was not so hard, though I had to figure out where to put them so that I could control them individually. Then I used my right hand to put the other three

chopsticks into my left hand. That was hard, since I had to keep the three chopsticks already in my right hand from falling out. After a couple of tries I got it done.

Then I had to figure out how to use the six chopsticks. That was harder. I did not manage well with the left hand, but I succeeded in manipulating all three in the right hand. After a couple of minutes of practice and adjustment, I managed to pick up a piece of food using three sticks converging on it from three different directions, and put it in my mouth.

It didn't become easy — for practical purposes, using two chopsticks is completely superior. But precisely because using three in one hand is hard and ordinarily never thought of, it has "hack value", as my lunch companions immediately recognized. Playfully doing something difficult, whether useful or not, that is hacking.

I later told the Korea story to a friend in Boston, who proceded to put four chopsticks in one hand and use them as two pairs — picking up two different pieces of food at once, one with each pair. He had topped my hack. Was his action, too, a hack? I think so. Is he therefore a hacker? That depends on how much he likes to hack.

The hacking community developed at MIT and some other universities in the 1960s and 1970s. Hacking included a wide range of activities, from writing software, to practical jokes, to exploring the roofs and tunnels of the MIT campus. Other activities, performed far from MIT and far from computers, also fit hackers' idea of what hacking means: for instance, I think the controversial 1950s "musical piece" by John Cage, 4'33" [2], is more of a hack than a musical composition. The palindromic three-part piece written by Guillaume de Machaut in the 1300s, "Ma Fin Est Mon Commencement", was also a good hack, even better because it also sounds good as music. Puck appreciated hack value.

It is hard to write a simple definition of something as varied as hacking, but I think what these activities have in common is playfulness, cleverness, and exploration. Thus, hacking means exploring the limits of what is possible, in a spirit of playful cleverness. Activities that display playful cleverness have "hack value".

Hackers typically had little respect for the silly rules that administrators like to impose, so they looked for ways around. For instance, when computers at MIT started to have "security" (that is, restrictions on what users could do), some hackers found clever ways to bypass the security, partly so they could use the computers freely, and partly just for the sake of cleverness (hacking does not need to be useful). However, only some hackers did this — many were occupied with other kinds

of cleverness, such as placing some amusing object on top of MIT's great dome [3], finding a way to do a certain computation with only 5 instructions when the shortest known program required 6, writing a program to print numbers in roman numerals, or writing a program to understand questions in English.

Meanwhile, another group of hackers at MIT found a different solution to the problem of computer security: they designed the Incompatible Timesharing System without security "features". In the hacker's paradise, the glory days of the Artificial Intelligence Lab, there was no security breaking, because there was no security to break. It was there, in that environment, that I learned to be a hacker, though I had shown the inclination previously. We had plenty of other domains in which to be playfully clever, without building artificial security obstacles which then had to be overcome.

Yet when I say I am a hacker, people often think I am making a naughty admission, presenting myself specifically as a security breaker. How did this confusion develop?

Around 1980, when the news media took notice of hackers, they fixated on one narrow aspect of real hacking: the security breaking which some hackers occasionally did. They ignored all the rest of hacking, and took the term to mean breaking security, no more and no less. The media have since spread that definition, disregarding our attempts to correct them. As a result, most people have a mistaken idea of what we hackers actually do and what we think.

You can help correct the misunderstanding simply by making a distinction between security breaking and hacking — by using the term "cracking" for security breaking. The people who do it are "crackers" [4]. Some of them may also be hackers, just as some of them may be chess players or golfers; most of them are not.

****

1. Pronounced like stuckpeckee minus the s (with an unaspirated t), if I recall right.
2. The piece 4'33" is a trivial piece. For each "movement", the pianist opens the keyboard cover, waits the appropriate amount of time, then closes it; that's all. It is a musical counterpart of the empty set.
3. Going on the great dome is "forbidden", so in a sense it constitutes "breaking security". Nonetheless, the MIT Museum proudly exhibited photos of some of the best dome hacks, as well as some of the objects that hackers placed on the dome in their hacks. The MIT administration thus implicitly recognizes that "breaking security" is not necessarily evil and need not be invariably

condemned. Whether security breaking is wrong depends on what the security breaker proceeds to do with the "forbidden" access thus obtained. Hurting people is bad, amusing the community is good.

4. I coined the term "cracker" in the early 80s when I saw journalists were equating "hacker" with "security breaker".

****

Here are some examples of fun hacks. If they make you smile, you're a hacker at heart.

First, some of mine.

- I learned to use two pairs of chopsticks too. Here I demonstrate this. [twitpic.com/558zg]
- Speaking of chopsticks, some kinds of Italian grissini work fine as chopsticks — then, after the meal, you can eat them. I brought a bag of them to Taiwan once just to show them that Italy has chopsticks too.
- Customer Training College changed to Customer Draining College [hn.my/drain]. Sassy, not computer-related.
- Photos of some other hacks I've done are here. [hn.my/stallhacks]
- In India there is a chain of fine Bengali restaurants called "Oh! Calcutta". The staff, and the clients, have no idea why that expression is notorious. During my 2014 visit to India I decided to inform them by bringing to the restaurant some printouts of the painting, and a publicity photo from the play. I left a copy of each with the staff.
- Pre-Zen studies (an April fool). [hn.my/prezen]
- Many years ago I had a root canal operation in a molar in the back of my mouth. It was difficult for me to keep my mouth open far enough, and the dentist said this was because I had a rather small mouth.

  When it was done, I had him sign a testimonial affirming this fact. I gave it to my mother to show she was wrong about me, all those years when she said I had a big mouth.
- My puns are also playful cleverness. [hn.my/puns]

Other people's hacks.

- Everyone's first hack: walking in the wrong direction on an escalator. That's not the way it's designed to be used, but can you make it work?
- I think this award-winning art project was actually a hack. [hn.my/voina]
- A robot that climbs windows to deploy a sun shade. Pure, sweet, & computer-based. [hn.my/shady]

- Hoisting Nigerian scammers on their own petard. [hn.my/eater]Cunning, mischievous, and not using computers except for email and phone calls.
- Lady Gaga's approach to clothing seems like hacking to me.
- A charming hack in the London Underground. [hn.my/subway]
- The hacker who made this poster was arrested for it. [hn.my/drone]
- TicBot is a conversation hack. [touretteshero.com]
- Just for the hack of it, the ultimate series of hacks with ordinary everyday objects appears in the 1987 film, Der Lauf der Dinge, by Fischli and Weiss. (This should not be confused with the unrelated 2006 film by the same name.)
- Although hacking and cracking are conceptually unrelated, occasionally they are found together. This is hacking that involves some cracking. [hn.my/cat]

  This hack has pointed out the injustice of the laws against "child" pornography, but doing that by causing other people to be jailed seems wrong to me. (Hacks can raise ethical issues just as other activities do; cleverness and playfulness do not guarantee that one can do no wrong.) It is also foolhardy to taunt a dangerous monster.
- A fun hack implemented via cracking: making TV emergency alert system give warnings about dead bodies emerging from graves. [hn.my/zombie]

  The security holes that made this possible might be used humorlessly to do real harm, but this hack didn't do harm. ■

Richard Stallman is the founder of the Free Software Foundation and the GNU project.

HACK ON YOUR SEARCH ENGINE

and help change the future of search

duckduckhack.com

Make the Type System Do the Work

By NATHAN WONG

Declaring types and being restricted by the type system is often cited as a negative aspect of C++. I think this is an unfair assessment: a type system can make a programmer's life considerably easier if it's embraced instead of fought, as we're seeing with the rise in popularity of Haskell. But C++, despite all its warts, has a pretty formidable type system of its own.

The object-oriented paradigm is commonly taught with the "Dog is-a Mammal" architectural mentality where your classes are supposed to mirror real life objects and act accordingly. Make no mistake, this approach is an over-simplification of software architecture and should be treated as such, but the principles behind it are actually fairly sound. Classes should aim to be a self-contained representation of some concept or thing that has state and actions. Here, we're going to focus on how to make the type system work for you instead of against you.

Specifically, we're going to focus on the conversion of data from one form to another. Many seem to think of conversions as being functions, taking one piece of data and returning another. But in doing so, we callously throw away dimensional analysis, a skill that appears to have been lost in translation from the natural sciences to computing.

A simple example that demonstrates the importance of dimensional consistency is temperature conversions. All too often we see functions converting equivalent units look something like this:

Function-Based Conversion

```
double celsiusToFahrenheit(double
deg_celsius)
{
  return deg_celsius * 9 / 5 + 32;
}

double temperature_fahrenheit =
celsiusToFahrenheit(20);
```

OK, it works. It compiles, runs, gives the right answer, and passes all tests. The only problem is that you end up with a variable that fails to describe itself better than "I'm a number". We end up using Hungarian-like system (apps Hungarian, specifically) to indicate the true units of the variable (Fahrenheit or Celsius). We recognize the importance of maintaining unit analysis, but we fail to enforce this convention; as with all Hungarian systems, the onus falls on the developer (and future developers) to maintain the accuracy of the system.

Instead, we should rely on the type system of the language to enforce this.

Type-Enforced Conversion

```
struct Degrees
{
    double val;
    Degrees(double _val) : val(_
val) {}
};
struct DegCelsius : public Degrees
{
    DegCelsius(double deg) :
Degrees(deg) {}
    DegCelsius(const DegFahrenheit
&deg)
        : Degrees((deg.val - 32)
* 5 / 9) {}
};
struct DegFahrenheit : public
Degrees
{
    DegFahrenheit(double deg) :
Degrees(deg) {}
    DegFahrenheit(const DegCelsius
&deg)
        : Degrees(deg.val * 9 / 5
+ 32) {}
};

DegFahrenheit input(68);
DegCelsius temperature = input;
```

Now it's obvious to any developer what type of degrees the `temperature` variable is holding, and the units are carried and enforced by the compiler; you're physically unable to assign a Celsius degree to a Fahrenheit degree without it converting it properly for you.

The overhead of setting up a coherent type system may seem burdensome, but in an application or library that handles many conversions in ways that should be transparent to the developer, this time investment will pay for itself. All units coming from math and science would benefit from being setup this way: just think how much easier it would be if `sin` took `Radians` instead

of a `double`, and `Radians` had a constructor that took Degrees: you could write `sin(Degrees(180))` and get the correct result.

Coordinates

Let's say you're plotting points on a graph (one of the many widgets in your application). You want the user to be able to click on a point in the graph and have it draw the point and log the graph coordinates.

Since we're just dealing with `x` and `y`, we could get away with just passing `aint32_t`'s around. But often this gets confusing because the graph widget's mouse click event gives you the coordinates relative to itself, whereas the graph coordinates have the origin at the center of the graph widget, and `y` grows as you go up instead of down. (And to make things more confusing, sometimes you have absolute coordinates relative to your graph widget's parent, too.)

As with before, we may have a function with the signature `Point pointCoordToGraphCoord(const Point &coord);`, but this requires the programmer to remember what type of coordinates they have when handling the data, and creating a developer-enforced naming convention to help convey this meaning is error-prone and tedious. Instead, the type system will not only enforce this convention, it will convert between the coordinate systems as well.

Type-Enforced Coordinates

```
// just holds an (x,y), oblivious
// to its purpose in life
struct Point
{
    int32_t x, y;
    Point(int32_t _x, int32_t _y)
: x(_x), y(_y) {}
    Point() : x(0), y(0) {}
};
// represents a point where (0,0)
// is the top-left of the widget
struct RealPoint : public Point
{
    RealPoint(int32_t x, int32_t
y) : Point(x, y) {}
    RealPoint() : Point() {}
};
// represents a point where (0,0)
// is in the center, & y grows up
struct GraphPoint : public Point
{
    GraphPoint(int32_t x, int32_t
y) : Point(x, y) {}
    GraphPoint() : Point() {}
};
```

Our mouse handler event, being a system call, probably still gives us a raw `x` and `y`, with which we can immediately construct a `RealPoint` for further use. Now our conversion function can be called `GraphPoint realToGraphCoords(const RealPoint &point);`, and it's clear what type of coordinate system any given variable is using.

Naturally, this conversion function should be part of `GraphPoint`, such as `static GraphPoint GraphPoint::FromRealCoords(const RealPoint &coords);`. Once the problem has been reduced to just converting real coordinates to graph coordinates, though, it makes the most sense to just create a constructor in the `GraphPoint` to handle the conversion for us.

Implicit Unit Conversion

```
// represents a point where (0,0) is in the center, and y grows up
struct GraphPoint : public Point
{
    GraphPoint(int32_t x, int32_t y) : Point(x, y) {}
    GraphPoint() : Point() {}
    GraphPoint(const RealPoint &coords) {
        x = coords.x - GraphWidget::width / 2;
        y = GraphWidget::height - coords.y - GraphWidget::height / 2;
    }
};
```

Now, as a developer, we don't even have to think about which coordinates we have on-hand.

Example Usage

```
bool GraphWidget::clickHandler(int32_t x, int32_t y)
{
    RealPoint coords(x, y);

    drawPoint(coords);
    logPoint(coords, "user click");

    return true;
}

void GraphWidget::drawPoint(const RealPoint &coords)
{
    DrawingLibrary::Circle(coords, 2); // etc.
}

void GraphWidget::logPoint(const GraphPoint &coords,
    const string &action)
{
    logfile << action << " at (" << coords.x << ", " << coords.y << ")"
        << endl;
}
```

The type system does all the work for us. The click handler (i.e., the user of our system) does not need to know that drawing and logging require different coordinates systems, and perhaps even better, the `drawPoint` and `logPoint` functions don't need to worry about what's being passed in. Nobody needs to make assumptions, which means less human errors and more reliable code.

Further Reading

The type system affords developers an opportunity to save time and reduce bugs. Writing maintainable code should be a first priority, and embracing the power of static typing can make code easier to work with down the road. Wrong code should look wrong, and failing to compile is even better. There are numerous everyday examples of how types can help. One such example is handling safe and unsafe strings to prevent XSS attacks by having the type-system enforce unsafe-by-default output: `print(NoEscapeString("<b>Note:</b>)); print(usermsg);` is easy to reason with.

Since first writing this article in January, I've been exposed to Bjarne Stroustrup's C++11 Style talk [hn.my/cpp11] which inspired me to finally edit and post it. Stroustrup's talk includes a great demonstration of how to implement a unit system using C++11's new user-defined literals, and makes a great argument for type-rich programming.

It's time to start embracing type systems instead of using non-descript number types and to ask ourselves: how else can I take advantage of the type system to make my life easier? ■

Nathan Wong is the Co-Founder and CTO of BuySellAds, an ad-tech startup focused on making advertising more accessible. You can read his blog about the intersection of business and technology at *nathan.ca*, or follow him on Twitter at *@nathandev*

Reprinted with permission of the original author.
First appeared in *hn.my/typesystem* (nathan.ca)

Why Registers Are Fast and RAM Is Slow

By MIKE ASH

Distance

Let's start with distance. It's not necessarily a big factor, but it's the most fun to analyze. RAM is farther away from the CPU than registers are, which can make it take longer to fetch data from it.

Take a 3GHz processor as an extreme example. The speed of light is roughly one foot per nanosecond, or about 30cm per nanosecond for you metric folk. Light can only travel about four inches in time of a single clock cycle of this processor. That means a roundtrip signal can only get to a component that's two inches away or less, and that assumes that the hardware is perfect and able to transmit information at the speed of light in vacuum. For a desktop PC, that's pretty significant. However, it's much less important for an iPhone, where the clock speed is much lower (the 5S runs at 1.3GHz) and the RAM is right next to the CPU.

Cost

As much as we might wish it wasn't, cost is always a factor. In software, when trying to make a program run fast, we don't go through the entire program and give it equal attention. Instead, we identify the hotspots that are most critical to performance, and give them the most attention. This makes the best use of our limited resources. Hardware is similar. Faster hardware is more expensive, and that expense is best spent where it'll make the most difference.

Registers get used extremely frequently, and there aren't a lot of them. There are only about 6,000 bits of register data in an A7 (32 64-bit general-purpose registers plus 32 128-bit floating-point registers, and some miscellaneous ones). There are about 8 billion bits (1GB) of RAM in an iPhone 5S. It's worthwhile to spend a bunch of money making each register

bit faster. There are literally a million times more RAM bits, and those 8 billion bits pretty much have to be as cheap as possible if you want a $650 phone instead of a $6,500 phone.

Registers use an expensive design that can be read quickly. Reading a register bit is a matter of activating the right transistor and then waiting a short time for the register hardware to push the read line to the appropriate state.

Reading a RAM bit, on the other hand, is more involved. A bit in the DRAM found in any smartphone or PC consists of a single capacitor and a single transistor. The capacitors are extremely small, as you'd expect given that you can fit 8 billion of them in your pocket. This means they carry a very small amount of charge, which makes it hard to measure. We like to think of digital circuits as dealing in ones and zeroes, but the analog world comes into play here. The read line is pre-charged to a level that's halfway between a one and a zero. Then the capacitor is connected to it, which either adds or drains a tiny amount of charge. An amplifier is used to push the charge towards zero or one. Once the charge in the line is sufficiently amplified, the result can be returned.

The fact that a RAM bit is only one transistor and one tiny capacitor makes it extremely cheap to manufacture. Register bits contain more parts and thereby cost much more.

There's also a lot more complexity involved just in figuring out what hardware to talk to with RAM because there's so much more of it. Reading from a register looks like:

1. Extract the relevant bits from the instruction.
2. Put those bits onto the register file's read lines.
3. Read the result.

Reading from RAM looks like:

1. Get the pointer to the data being loaded. (Said pointer is probably in a register. This already encompasses all of the work done above!)
2. Send that pointer off to the MMU.
3. The MMU translates the virtual address in the pointer to a physical address.
4. Send the physical address to the memory controller.
5. Memory controller figures out what bank of RAM the data is in and asks the RAM.
6. The RAM figures out particular chunk the data is in, and asks that chunk.

7. Step 6 may repeat a couple of more times before narrowing it down to a single array of cells.
8. Load the data from the array.
9. Send it back to the memory controller.
10. Send it back to the CPU.
11. Use it!

Whew.

Dealing With Slow RAM

That sums up why RAM is so much slower. But how does the CPU deal with such slowness? A RAM load is a single CPU instruction, but it can take potentially hundreds of CPU cycles to complete. How does the CPU deal with this?

First, just how long does a CPU take to execute a single instruction? It can be tempting to just assume that a single instruction executes in a single cycle, but reality is, of course, much more complicated.

Back in the good old days, when men wore their sheep proudly and the nation was undefeated in war, this was not a difficult question to answer. It wasn't one-instruction-one-cycle, but there was at least some clear correspondence. The Intel 4004, for example, took either 8 or 16 clock cycles to execute one instruction, depending on what that instruction was. Nice and understandable. Things gradually got more complex, with a wide variety of timings for different instructions. Older CPU manuals will give a list of how long each instruction takes to execute.

Now? Not so simple.

Along with increasing clock rates, there's also been a long drive to increase the number of instructions that can be executed per clock cycle. Back in the day, that number was something like 0.1 of an instruction per clock cycle. These days, it's up around 3-4 on a good day. How does it perform this wizardry? When you have a billion or more transistors per chip, you can add in a lot of smarts. Although the CPU might be executing 3-4 instructions per clock cycle, that doesn't mean each instruction takes 1/4th of a clock cycle to execute. They still take at least one cycle, often more. What happens is that the CPU is able to maintain multiple instructions in flight at any given time. Each instruction can be broken up into pieces: load the instruction, decode it to see what it means, gather the input data, perform the computation, and store the output data. Those can all happen on separate cycles.

On any given CPU cycle, the CPU is doing a bunch of stuff simultaneously:

1. Fetching potentially several instructions at once.
2. Decoding potentially a completely different set of instructions.
3. Fetching the data for potentially yet another different set of instructions.
4. Performing computations for yet more instructions.
5. Storing data for yet more instructions.

But, you say, how could this possibly work? For example:

```
add x1, x1, x2
add x1, x1, x3
```

These can't possibly execute in parallel like that! You need to be finished with the first instruction before you start the second!

It's true, that can't possibly work. That's where the smarts come in. The CPU is able to analyze the instruction stream and figure out which instructions depend on other instructions and shuffle things around. For example, if an instruction after those two adds doesn't depend on them, the CPU could end up executing that instruction before the second add, even though it comes later in the instruction stream. The ideal of 3-4 instructions per clock cycle can only be achieved in code that has a lot of independent instructions.

What happens when you hit a memory load instruction? First of all, it is definitely going to take forever, relatively speaking. If you're really lucky and the value is in L1 cache, it'll only take a few cycles. If you're unlucky and it has to go all the way out to main RAM to find the data, it could take literally hundreds of cycles. There may be a lot of thumb-twiddling to be done.

The CPU will try not to twiddle its thumbs, because that's inefficient. First, it will try to anticipate. It may be able to spot that load instruction in advance, figure out what it's going to load, and initiate the load before it really starts executing the instruction. Second, it will keep executing other instructions while it waits, as long as it can. If there are instructions after the load instruction that don't depend on the data being loaded, they can still be executed. Finally, once it's executed everything it can and it absolutely cannot proceed any further without that data it's waiting on, it has little choice but to stall and wait for the data to come back from RAM.

Conclusion

- RAM is slow because there's a ton of it.
- That means you have to use designs that are cheaper, and cheaper means slower.
- Modern CPUs do crazy things internally and will happily execute your instruction stream in an order that's wildly different from how it appears in the code.
- That means that the first thing a CPU does while waiting for a RAM load is run other code.
- If all else fails, it'll just stop and wait, and wait, and wait, and wait. ■

Mike Ash has been programming for Apple platforms for over two decades and for Mac OS X since the Public Beta. He is the author of the bi-weekly Friday Q&A [mikeash.com/pyblog] blog series on deep technical topics related to Mac and iOS programming, as well as the compilation book The Complete Friday Q&A: Volume I. In between abusing the Objective-C runtime, he flies his glider over the beautiful Shenandoah Valley. When not flying, he holds down a day job at Plausible Labs.

Reprinted with permission of the original author.
First appeared in *hn.my/registers* (mikeash.com)

Vim Croquet

By SETH BROWN

I RECENTLY DISCOVERED AN interesting game called VimGolf [vimgolf.com]. The objective of the game is to transform a snippet of text from one form to another in as few keystrokes as possible. As I was playing around with different puzzles on the site, I started to get curious about my text editing habits. I wanted to better understand how I manipulated text with vim and to see if I could identify any inefficiencies in my workflow. I spend a huge amount of time inside my text editor, so correcting even slight areas of friction can result in worthwhile productivity gains. This post explains my analysis and how I reduced the number of keystrokes I use in vim. I call this game Vim Croquet.

Data Acquisition

I started my analysis by collecting data. All my text editing on a computer is done with vim, so for 45 days I logged every keystroke I used in vim with the `scriptout` flag. For convenience, I aliased vim in my shell to record all my keystrokes into a log file:

```
alias vim='vim -w ~/.vimlog "$@"'
```

Next, I needed to parse the resulting data. Parsing vim is complicated. vim is a modal editor where a single command can have different meanings in different modes. Commands can also have contextual effects where the behavior of certain actions can be different depending on where they are executed within a buffer. For example, typing `cib` in normal mode moves the user into insert mode if the command is executed between parentheses, but leaves the user in normal mode if executed outside of parentheses. If `cib` is executed in insert mode it has an altogether different behavior; it writes the characters `cib` into the current buffer.

I looked at several candidate tools for parsing vim commands including industrial parser libraries like antler [antlr.org] and parsec [hn.my/parsec] as well as a vim-specific project called

vimprint [hn.my/vimprint]. After some deliberation, I decided to write my own tool. I don't do a lot of language processing, so investing the time to learn a sophisticated parser seemed unwarranted.

I wrote a crude lexer in Haskell to tokenize the keystrokes I collected into individual vim commands. My lexer uses monoids to extract normal mode commands from my log for further analysis. Here's the source code for the lexer:

```
import qualified Data.ByteString.
Lazy.Char8 as LC
import qualified Data.List as DL
import qualified Data.List.Split
as LS
import Data.Monoid
import System.IO

main = hSetEncoding stdout utf8
>>
       LC.getContents >>= mapM_
putStrLn . process

process =   affixStrip
          . startsWith
          . splitOnMode
          . modeSub
          . capStrings
          . split mark
          . preprocess

subs = appEndo . mconcat . map
(Endo . sub)

sub (s,r) lst@(x:xs)
    | s `DL.isPrefixOf` lst = sub'
    | otherwise = x:sub (s,r) xs
    where
     sub' = r ++ sub (s,r) (drop
(length s) lst)
sub (_,_) [] = []

preprocess =   subs meta
             . DL.intercalate " "
             . DL.words
             . DL.unwords
             . DL.lines
             . LC.unpack

splitOnMode = DL.concat $ map (\
el -> split mode el)

startsWith = filter (\el -> mark
`DL.isPrefixOf` el && el /= mark)

modeSub = map (subs mtsl)

split s r = filter (/= "") $ s
`LS.splitOn` r

affixStrip =    clean
             . concat
             . map (\el -> split
mark el)

capStrings = map (\el -> mark ++
el ++ mark)

clean = filter (not . DL.isInfixOf
"[M")
```

```
(mark, mode, n) = ("-(*)-","-(!)-
", "")
meta = [("\"",n),("\\",n),
        ("\195\130\194\128\195\13
1\194\189`",n),
        ("\194\128\195\189`",n),
        ("\194\128kb\ESC",n),
        ("\194\128kb",n),("[>0;95
;c",n),
        ("[>0;95;0c",n),
        ("\ESC",mark),("\
ETX",mark),("\r",mark)]
mtsl = [(":",mode),("A",mode),
("a",mode),
        ("I",mode), ("i",mode),
        ("O",mode),("o",mode),
        ("v", mode),("/",mode),("\
ENQ","^e"),
       ("\DLE","^p"),("\NAK","^u"),
       ("\EOT","^d"),("\ACK","^f"),
       ("\STX","^f"),("\EM","^y"),
       ("\SI","^o"),("\SYN","^v"),
        ("\DC2","^r")]
```

Here's a sample of the data in its unprocessed form and its structure after lexing:

```
cut -c 1-42 ~/.vimlog | tee >(cat
-v;echo)
| ./lexer
`Mihere's some text^Cyyp$bimore
^C0~A.^C:w^M:q
`M
yyp$b
0~
```

My lexer reads from stdin and sends processed normal mode commands to stdout. In the above example pipe, I use a process substitution to print a representation of the unprocessed data on the second line and the resulting output of the lexer on subsequent lines. Each line in the output of the lexer represents a grouping of normal mode commands executed in sequence. The lexer correctly determined that I started in normal mode by navigating to a specific buffer using the `M mark; then typing `here's some text` in insert mode; then copying and pasting the line and moving to the start of the last word on the line using `yyp$b`; then entering additional text; and finally navigating to the start of the line and capitalizing the first character using `0~`.

Key Heat Map

After lexing my log data, I forked Patrick Wied's awesome heatmap-keyboard project [hn.my/heatmap] and added my own custom layout to read the output of my lexer. Patrick's project does not detect most meta-characters like escape, control, and command, so it was necessary for me to write a data loader in JavaScript and make some other modifications so the heatmap would accurately depict key usage in vim. I translated metacharacters used in vim to unicode representations and mapped these onto the keyboard. Here's what my key usage

looked like based on ≈500,000 normal mode keystrokes processed by my lexer. Increasing wavelengths denotes more prevalent key usage:

A prominent feature of the heatmap is the prevalent usage of the control key. I use control for numerous movement commands in vim. For example, I use `^p` for Control P [hn.my/ctrlp] and I cycle forward and backward through open buffers with `^j` and `^k`, respectively. Control is an efficient movement on my Kinesis Advantage because I remap it to left thumb delete.

Another pattern in the heatmap that jumped out at me was my heavy use of `^E` and `^Y`. I routinely use these commands to navigate up and down through source code, but moving vertically with these commands is inefficient. Each time one of these commands is executed, the cursor only moves a few lines at a time. A more efficient pattern would be to use larger vertical movements with `^U` and `^D`. These commands move the cursor up or down a half screen at a time, respectively.

Command Frequency

The heatmap gives a good overview of how I use individual keys, but I also wanted to learn more about how I used different key sequences. I sorted the lines in the output of my lexer by frequency to uncover my most used normal commands using a simple one-liner:

```
$ sort normal_cmds.txt | uniq -c
| sort -nr | head -10 | \
    awk '{print NR,$0}' | column
-t

1   2542    j
2   2188    k
3   1927    jj
4   1610    p
5   1602    ^j
6   1118    Y
7   987     ^e
8   977     zR
9   812     P
10  799     ^y
```

Seeing `zR` rank as my 8th most used sequence was unexpected. After pondering this, I realized a huge inefficiency in my text editing. My `.vimrc` is setup to automatically fold text. The problem with this configuration is that I almost immediately unfold all folded text, so it makes no sense for my vim configuration to use automatically fold text by default. Therefore, I removed this setting so that I would no longer need to repeatedly use the `zR` command.

Command Complexity

Another optimization I wanted to look at was normal mode command complexity. I was curious to see if I could find any commands that I routinely used which also required an excessive number of keystrokes to execute. I wanted to find these commands so that I could create shortcuts to speed up their execution. I used entropy as a proxy to measure command complexity using a short script in Python:

```
#!/usr/bin/env python
import sys
from codecs import getreader, get-
writer
from collections import Counter
from operator import itemgetter
from math import log, log1p

sys.stdin = getreader('utf-8')
(sys.stdin)
sys.stdout = getwriter('utf-8')
(sys.stdout)

def H(vec, correct=True):
    """Calculate the Shannon
Entropy of a vector
    """
    n = float(len(vec))
    c = Counter(vec)
    h = sum(((-freq / n) *
log(freq / n, 2)) for freq in
c.values())

    # impose a penality to correct
for size
    if all([correct is True, n >
0]):
        h = h / log1p(n)

    return h

def main():
    k = 1
    lines = (_.strip() for _ in
sys.stdin)
    hs = ((st, H(list(st))) for st
in lines)
    srt_hs = sorted(hs,
key=itemgetter(1), reverse=True)
    for n, i in enumerate(srt_
hs[:k], 1):
        fmt_st = u'{r}\t{s}\
t{h:.4f}'.format(r=n, s=i[0],
h=i[1])
        print fmt_st

if __name__ == '__main__':
    main()
```

The entropy script reads from stdin and finds the normal mode command with the highest entropy. I used the output of my lexer as input for my entropy calculation:

```
$ sort normal_cmds.txt | uniq -c
| sort -nr | sed "s/^[ \t]*//" | \
    awk 'BEGIN{OFS="\t";}{if
($1>100) print $1,$2}' | \
    cut -f2 | ./entropy.py

1 ggvG$"zy 1.2516
```

In the command above, I first filtered all the normal mode commands that I executed more than 100 times. Then, among this subset, I found the command with the highest entropy. This analysis precipitated the command `ggvG$"zy,` which I executed 246 times in 45 days. The command takes an unwieldy 11 keystrokes and yanks the entire current buffer into the z register. I typically use this command to move the contents of one buffer into another buffer. Since I use this sequence so frequently, I added a short cut to my `.vimrc` to reduce the number of keystrokes I need to execute:

```
nnoremap <leader>ya ggvG$"zy
```

Conclusions

My Vim Croquet match revealed three optimizations to decrease the number of keystrokes I use in vim:

- Use coarser navigation commands like `^U` and `^D` instead of `^E` and `^Y`
- Prevent buffers from automatically folding text to obviate using `zR`
- Create shortcuts for verbose commands that are frequently used like `ggvG$"zy`

These 3 simple changes have saved me thousands of superfluous keystrokes each month.

The code snippets above are presented in isolation and may be difficult to follow. To help clarify the steps in my analysis, here's my Makefile, which shows how the code presented in this post fits together:

```
SHELL           := /bin/bash
LOG             := ~/.vimlog
CMDS            := normal_cmds.txt
FRQS            := frequencies.txt
ENTS            := entropy.txt
LEXER_SRC       := lexer.hs
LEXER_OBJS      := lexer.{o,hi}
LEXER_BIN       := lexer
H               := entropy.py
UTF             := iconv -f iso-
8859-1 -t utf-8

.PRECIOUS: $(LOG)
.PHONY: all entropy clean dist-
clean

all: $(LEXER_BIN) $(CMDS) $(FRQS)
entropy

$(LEXER_BIN): $(LEXER_SRC)
    ghc --make $^

$(CMDS): $(LEXER_BIN)
    cat $(LOG) | $(UTF) | ./$^ >
$@
```

```
$(FRQS): $(H) $(LOG) $(CMDS)
    sort $(CMDS) | uniq -c | sort
-nr | sed "s/^[ \t]*//" | \
      awk 'BEGIN{OFS="\t";}{if
($$1>100) print NR,$$1,$$2}' > $@
entropy: $(H) $(FRQS)
    cut -f3 $(FRQS) | ./$(H)
clean:
    @- $(RM) $(LEXER_OBJS)
$(LEXER_BIN) $(CMDS) $(FRQS)
$(ENTS)

distclean: clean
```

■

Seth Brown is a Data Scientist in the telecommunications industry. His research focuses on understanding the topology of the global Internet using large-scale computing, statistical modeling, and data visualization techniques. Prior to computer networking, he was a research scientist in bioinformatics where he studied the structure and function of gene regulatory networks. Seth writes about topics in data analysis and data visualization on his website, *drbunsen.org*. He can be found on Twitter *@drbunsen*

Reprinted with permission of the original author.
First appeared in *hn.my/vimcroquet* (drbunsen.org)

AWS Tips I Wish I'd Known Before I Started

A collection of random tips for Amazon Web Services (AWS) that I wish I'd been told a few years ago.

By RICH ADAMS

Moving from physical servers to the "cloud" involves a paradigm shift in thinking. Generally in a physical environment you care about each individual host; they each have their own static IP, you probably monitor them individually, and if one goes down you have to get it back up ASAP. You might think you can just move this infrastructure to AWS and start getting the benefits of the "cloud" straight away. Unfortunately, it's not quite that easy (believe me, I tried). You need to think differently when it comes to AWS, and it's not always obvious what needs to be done.

So, inspired by Sehrope Sarkuni's recent post [hn.my/sarkuni], here's a collection of AWS tips I wish someone had told me when I was starting out. These are based on things I've learned deploying various applications on AWS both personally and for my day job. Some are just "gotcha"'s to watch out for (and that I fell victim to), some are things I've heard from other people that I ended up implementing and finding useful, but mostly they're just things I've learned the hard way.

Application Development

Store no application state on your servers.

The reason for this is so that if your server gets killed, you won't lose any application state. To that end, sessions should be stored in a database, not on the local filesystem. Logs should be handled via syslog (or similar) and sent

to a remote store. Uploads should go direct to S3 (don't store on local filesystem and have another process move to S3 for example). And any post-processing or long running tasks should be done via an asynchronous queue (SQS is great for this).

Store extra information in your logs. Log lines normally have information like timestamp, pid, etc. You'll also probably want to add instance-id, region, availability-zone and environment (staging, production, etc.), as these will help debugging considerably. You can get this information from the instance metadata service. The method I use is to grab this information as part of my bootstrap scripts, and store it in files on the filesystem (/env/az, /env/region, etc). This way I'm not constantly querying the metadata service for the information. You should make sure this information gets updated properly when your instances reboot, as you don't want to save an AMI and have the same data persist, as it will then be incorrect.

If you need to interact with AWS, use the SDK for your language. Don't try to roll your own; I did this at first as I only needed a simple upload to S3, but then you add more services and it's just an all-around bad idea. The AWS SDKs are well written, handle authentication automatically, handle retry logic, and they're maintained and iterated on by Amazon. Also, if you use EC2 IAM roles (which you absolutely should, more on this later) then the SDK will automatically grab the correct credentials for you.

Have tools to view application logs. You should have an admin tool, syslog viewer, or something that allows you to view current real-time log info without needing to SSH into a running instance. If you have centralized logging (which you really should), then you just want to be sure you can read the logs there without needing to use SSH. Needing to SSH into a running application instance to view logs is going to become problematic.

Operations

Disable SSH access to all servers. This sounds crazy, I know, but port 22 should be disallowed for everyone in your security group. If there's one thing you take away from this post, this should be it: If you have to SSH into your servers, then your automation has failed. Disabling it at the firewall level (rather than on the servers themselves) will help the transition to this frame of thinking, as it will highlight any areas you need to automate, while still letting you easily re-instate access to solve immediate issues. It's incredibly freeing to know that you never need to SSH into an instance. This is both the most frightening and yet most useful thing I've learned.

Servers are ephemeral; you don't care about them. You only care about the service as a whole.

If a single server dies, it should be of no big concern to you. This is where the real benefit of AWS comes in compared to using physical servers yourself. Normally if a physical server dies, there's panic. With AWS, you don't care, because auto-scaling will give you a fresh new instance soon anyway. Netflix has taken this several steps further with their simian army, where they have things like Chaos Monkey, which will kill random instances in production (they also have Chaos Gorilla to kill AZs and I've heard rumor of a Chaos Kong to kill regions...). The point is that servers will fail, but this shouldn't matter in your application.

Don't give servers static/elastic IPs.

For a typical web application, you should put things behind a load balancer, and balance them between AZs. There are a few cases where Elastic IPs will probably need to be used, but in order to make best use of auto-scaling you'll want to use a load balancer instead of giving every instance their own unique IP.

Automate everything.

This is more of general operations advice than AWS specific, but everything needs to be automated. Recovery, deployment, failover, etc. Package and OS updates should be managed by something, whether it's just a bash script, or Chef/Puppet, etc. You shouldn't have to care about this stuff. As mentioned earlier, you should also make sure to disable SSH access, as this will pretty quickly highlight any part of your process that isn't automated. Remember the key phrase from earlier, if you have to SSH into your servers, then your automation has failed.

Everyone gets an IAM account. Never login to the master.

Usually you'll have an "operations account" for a service, and your entire ops team will have the password. With AWS, you definitely don't want to do that. Everyone gets an IAM user with just the permissions they need (least privilege). An IAM user can control everything in the infrastructure. At the time of writing, the only thing an IAM user can't access are some parts of the billing pages.

If you want to protect your account even more, make sure to enable multi-factor authentication for everyone (you can use Google Authenticator). I've heard of some users who give the MFA token to two people, and the password to two others, so to perform any action on the master account, two of the users need to agree. This is overkill for my case, but worth mentioning in case someone else wants to do it.

Get your alerts to become notifications.

If you've set everything up correctly, your health checks should automatically destroy bad instances and spawn new ones. There's usually no action to take when getting a CloudWatch alert, as everything should be automated. If you're getting alerts where manual intervention is required, do a post-mortem and figure out if there's a way you can automate the action in the future. The last time I had an actionable alert from CloudWatch was about a year ago, and it's extremely awesome not to be woken up at 4am for ops alerts any more.

Billing

Set up granular billing alerts.

You should always have at least one billing alert set up, but that will only tell you on a monthly basis once you've exceeded your allowance. If you want to catch runaway billing early, you need a more fine grained approach. The way I do it is to set up an alert for my expected usage each week. So the first week's alert for say $1,000, the second for $2,000, third for $3,000, etc. If the week-2 alarm goes off before the 14th/15th of the month, then I know something is probably going wrong. For even more fine-grained control, you can set this up for each individual service, that way you instantly know which service is causing the problem. This could be useful if your usage on one service is quite steady month-to-month, but another is more erratic. Have the individual weekly alerts for the steady one, but just an overall one for the more erratic one. If everything is steady, then this is probably overkill, as looking at CloudWatch will quickly tell you which service is the one causing the problem.

Security

Use EC2 roles, do not give applications an IAM account.

If your application has AWS credentials baked into it, you're "doing it wrong." One of the reasons it's important to use the AWS SDK for your language is that you can really easily use EC2 IAM roles. The idea of a role is that you specify the permissions a certain role should get, then assign that role to an EC2 instance. Whenever you use the AWS SDK on that instance, you don't specify any credentials. Instead, the SDK will retrieve temporary credentials which have the permissions of the role you set up. This is all handled transparently as far as you're concerned. It's secure, and extremely useful.

Assign permissions to groups, not users.

Managing users can be a pain, if you're using Active Directory, or some other external authentication mechanism which you've integrated with IAM,

then this probably won't matter as much (or maybe it matters more). But I've found it much easier to manage permissions by assigning them only to groups, rather than to individual users. It's much easier to rein in permissions and get an overall view of the system than going through each individual user to see what permissions have been assigned.

Set up automated security auditing.
It's important to keep track of changes in your infrastructure's security settings. One way to do this is to first set up a security auditor role [hn.my/secaudit], which will give anyone assigned that role read-only access to any security-related settings on your account. You can then use this rather fantastic Python script [hn.my/seccofig], which will go over all the items in your account and produce a canonical output showing your configuration. You set up a cronjob somewhere to run this script, and compare its output to the output from the previous run. Any differences will show you exactly what has been changed in your security configuration. It's useful to set this up and just have it email you the diff of any changes.

Use CloudTrail to keep an audit log.
CloudTrail will log any action performed via the APIs or web console into an S3 bucket. Set up the bucket with versioning to be sure no one can modify your logs, and you then have a complete audit trail of all changes in your account. You hope that you will never need to use this, but it's well worth having for when you do.

S3

Use "-" instead of "." in bucket names for SSL.
If you ever want to use your bucket over SSL, using a "." will cause you to get certificate mismatch errors. You can't change bucket names once you've created them, so you'd have to copy everything to a new bucket.

Avoid filesystem mounts (FUSE, etc.).
I've found them to be about as reliable as a large government department when used in critical applications. Use the SDK instead.

You don't have to use CloudFront in front of S3 (but it can help).
If all you care about is scalability, you can link people directly to the S3 URL instead of using CloudFront. S3 can scale to any capacity (although some users have reported that it doesn't scale instantly), so it is great if that's all your care about. Additionally, updates are available quickly in S3, yet you have to wait for the TTL when using a CDN to see the change (although I believe you can set a 0s TTL in CloudFront now, so this point is probably moot).

If you need speed, or are handling very high bandwidth (10TB+), then

you might want to use a CDN like CloudFront in front of S3. CloudFront can dramatically speed up access for users around the globe, as it copies your content to edge locations. Depending on your use case, this can also work out slightly cheaper if you deal with very high bandwidth (10TB+) with lower request numbers, as it's about $0.010/GB cheaper for CloudFront bandwidth than S3 bandwidth once you get above 10TB, but the cost per request is slightly higher than if you were to access the files from S3 directly. Depending on your usage pattern, the savings from bandwidth could outweigh the extra cost per request. Since content is only fetched from S3 infrequently (and at a much lower rate than normal), your S3 cost would be much smaller than if you were serving content directly from S3. The AWS documentation on CloudFront explains how you can use it with S3.

Use random strings at the start of your keys.

This seems like a strange idea, but one of the implementation details of S3 is that Amazon uses the object key to determine where a file is physically placed in S3. So files with the same prefix might end up on the same hard disk for example. By randomizing your key prefixes, you end up with a better distribution of your object files.

EC2/VPC

Use tags!

Pretty much everything can be given tags, use them! They're great for organizing things, make it easier to search and group things up. You can also use them to trigger certain behaviors on your instances, for example a tag of env=debug could put your application into debug mode when it deploys, etc.

Use termination protection for non-auto-scaling instances. Thank me later.

If you have any instances which are one-off things that aren't under auto-scaling, then you should probably enable termination protection, to stop anyone from accidentally deleting the instance. I've had it happen, it sucks, learn from my mistake!

Use a VPC.

VPC either wasn't around, or I didn't notice it when I got started with AWS. It seems like a pain at first, but once you get stuck in and play with it, it's surprising easy to set up and get going. It provides all sorts of extra features over EC2 that are well worth the extra time it takes to set up a VPC. First, you can control traffic at the network level using ACLs, you can modify instance size, security groups, etc. without needing to terminate an instance. You can specify egress firewall rules (you cannot control outbound traffic from normal EC2). But the biggest thing is that you

have your own private subnet where your instances are completely cut off from everyone else, so it adds an extra layer of protection. Don't wait like I did, use VPC straight away to make things easy on yourself.

Use reserved instances to save big $$$.
Reserving an instance is just putting some money upfront in order to get a lower hourly rate. It ends up being a lot cheaper than an on-demand instance would cost. So if you know you're going to be keeping an instance around for 1 or 3 years, it's well worth reserving them. Reserved instances are a purely logical concept in AWS, you don't assign a specific instance to be reserved, but rather just specify the type and size, and any instances that match the criteria will get the lower price.

Lock down your security groups.
Don't use 0.0.0.0/0 if you can help it; make sure to use specific rules to restrict access to your instances. For example, if your instances are behind an ELB, you should set your security groups to only allow traffic from the ELBs, rather than from 0.0.0.0/0. You can do that by entering "amazon-elb/amazon-elb-sg" as the CIDR (it should auto-complete for you). If you need to allow some of your other instances access to certain ports, don't use their IP, but specify their security group identifier instead (just start typing "sg-" and it should auto-complete for you).

Don't keep unassociated Elastic IPs.
You get charged for any Elastic IPs you have created but not associated with an instance, so make sure you don't keep them around once you're done with them.

ELB

Terminate SSL on the load balancer.
You'll need to add your SSL certificate information to the ELB, but this will take the overhead of SSL termination away from your servers which can speed things up. Additionally, if you upload your SSL certificate, you can pass through the HTTPS traffic and the load balancer will add some extra headers to your request (x-forwarded-for, etc.), which are useful if you want to know who the end user is. If you just forward TCP, then those headers aren't added and you lose the information.

Pre-warm your ELBs if you're expecting heavy traffic.
It takes time for your ELB to scale up capacity. If you know you're going to have a large traffic spike (selling tickets, big event, etc.), you need to "warm up" your ELB in advance. You can inject a load of traffic, and it will cause ELB to scale up and not choke when you actually get the traffic; however, AWS suggests you contact them instead to pre-warm your load balancer. Alternatively you can install your own load balancer software on an EC2 instance and use that instead (HAProxy, etc).

ElastiCache

Use the configuration endpoints, instead of individual node endpoints.
Normally you would have to make your application aware of every Memcached node available. If you want to dynamically scale up your capacity, then this becomes an issue as you will need to have some way to make your application aware of the changes. An easier way is to use the configuration endpoint, which means using an AWS version of a Memcached library that abstracts away the auto-discovery of new nodes. The AWS guide to cache node auto-discovery has more information.

RDS

Set up event subscriptions for failover.
If you're using a Multi-AZ setup, this is one of those things you might not think about which ends up being incredibly useful when you do need it.

CloudWatch

Use the CLI tools.
It can become extremely tedious to create alarms using the web console, especially if you're setting up a lot of similar alarms, as there's no ability to "clone" an existing alarm while making a minor change elsewhere. Scripting this using the CLI tools can save you lots of time.

Use the free metrics.
CloudWatch monitors all sorts of things for free (bandwidth, CPU usage, etc.), and you get up to 2 weeks of historical data. This saves you having to use your own tools to monitor you systems. If you need longer than 2 weeks, unfortunately you'll need to use a third-party or custom built monitoring solution.

Use custom metrics.
If you want to monitor things not covered by the free metrics, you can send your own metric information to CloudWatch and make use of the alarms and graphing features. This can not only be used for things like tracking disk space usage, but also for custom application metrics too. The AWS page on publishing custom metrics has more information.

Use detailed monitoring.
It's ~$3.50 per instance/month, and well worth the extra cost for the extra detail. 1 minute granularity is much better than 5 minutes. You can have cases where a problem is hidden in the 5 minute breakdown but shows itself quite clearly in the 1 minute graphs. This may not be useful for everyone, but it's made investigating some issues much easier for me.

Auto-Scaling

Scale down on INSUFFICIENT_DATA as well as ALARM.

For your scale-down action, make sure to trigger a scale-down event when there's no metric data, as well as when your trigger goes off. For example, if you have an app which usually has very low traffic, but experiences occasional spikes, you want to be sure that it scales down once the spike is over and the traffic stops. If there's no traffic, you'll get INSUFFIFIENT_DATA instead of ALARM for your low traffic threshold and it won't trigger a scale-down action.

Use ELB health check instead of EC2 health checks.

This is a configuration option when creating your scaling group, you can specify whether to use the standard EC2 checks (is the instance connected to the network), or to use your ELB health check. The ELB health check offers way more flexibility. If your health check fails and the instance gets taken out of the load balancing pool, you're pretty much always going to want to have that instance killed by auto-scaling and a fresh one take its place. If you don't set up your scaling group to use the ELB checks, then that won't necessarily happen. The AWS documentation on adding the health check has all the information you need to set this up.

Only use the availability zones (AZs) your ELB is configured for.

If you add your scaling group to multiple AZs, make sure your ELB is configured to use all of those AZs, otherwise your capacity will scale up, and the load balancer won't be able to see them.

Don't use multiple scaling triggers on the same group.

If you have multiple CloudWatch alarms which trigger scaling actions for the same auto-scaling group, it might not work as you initially expect it to. For example, let's say you add a trigger to scale up when CPU usage gets too high, or when the inbound network traffic gets high, and your scale down actions are the opposite. You might get an increase in CPU usage, but your inbound network is fine. So the high CPU trigger causes a scale-up action, but the low inbound traffic alarm immediately triggers a scale-down action. Depending on how you've set your cool down period, this can cause quite a problem as they'll just fight against each other. If you want multiple triggers, you can use multiple auto-scaling groups.

IAM

Use IAM roles.

Don't create users for application, always use IAM roles if you can. They simplify everything, and keeps things secure. Having application users just creates a point of failure (what if someone accidentally deletes the API key?) and it becomes a pain to manage.

Users can have multiple API keys.

This can be useful if someone is working on multiple projects, or if you want a one-time key just to test something out, without wanting to worry about accidentally revealing your normal key.

IAM users can have multi-factor authentication, use it!

Enable MFA for your IAM users to add an extra layer of security. Your master account should most definitely have this, but it's also worth enabling it for normal IAM users too.

Route53

Use ALIAS records.

An ALIAS record will link your record set to a particular AWS resource directly (i.e., you can map a domain to an S3 bucket), but the key is that you don't get charged for any ALIAS lookups. So whereas a CNAME entry would cost you money, an ALIAS record won't. Also, unlike a CNAME, you can use an ALIAS on your zone apex. You can read more about this on the AWS page for creating alias resource record sets.

Elastic MapReduce

Specify a directory on S3 for Hive results.

If you use Hive to output results to S3, you must specify a directory in the bucket, not the root of the bucket, otherwise you'll get a rather unhelpful NullPointerException with no real explanation as to why.

Miscellaneous Tips

Scale horizontally.

I've found that using lots of smaller machines is generally more reliable than using a smaller number of larger machines. You need to balance this though, as trying to run your application from 100 t1.micro instances probably isn't going to work very well. Breaking your application into lots of smaller instances means you'll be more resilient to failure in one of the

machines. If you're just running from two massive compute cluster machines, and one goes down, things are going to get bad.

Your application may require changes to work on AWS.

While a lot of applications can probably just be deployed to an EC2 instance and work well, if you're coming from a physical environment, you may need to re-architect your application in order to accommodate changes. Don't just think you can copy the files over and be done with it.

Decide on a naming convention early, and stick to it.

There's a lot of resources on AWS where you can change the name later, but there's equally a lot where you cannot (security group names, etc.). Having a consistent naming convention will help to self-document your infrastructure. Don't forget to make use of tags too. ■

Rich Adams is a systems engineer at Gracenote who used to work on departure control systems for the airline industry. He now splits his time between playing with Amazon Web Services and making sure there's enough Mountain Dew flowing through him. Say hi to him on Twitter at *@r_adams*

Reprinted with permission of the original author.
First appeared in *hn.my/awstips* (wblinks.com)

Why I'm Betting on Julia

By EVAN MILLER

THE PROBLEM WITH most programming languages is they're designed by language geeks, who tend to worry about things that I don't much care for. Safety, type systems, homoiconicity, and so forth. I'm sure these things are great, but when I'm messing around with a new project for fun, my two concerns are 1) making it work and 2) making it fast. For me, code is like a car. It's a means to an end. The "expressiveness" of a piece of code is about as important to me as the "expressiveness" of a catalytic converter.

This approach to programming is often (derisively) called cowboy coding. I don't think a cowboy is quite the right image, because a cowboy must take frequent breaks due to the physical limitations of his horse. A better aspirational image is an obsessed scientist who spends weeks in the laboratory and emerges, bleary-eyed, exhausted, and wan, with an ingenious new contraption that possibly causes a fire on first use.

Enough about me. Normally I use one language to make something work, and a second language to make it fast, and a third language to make it scream. This pattern is fairly common. For many programmers, the prototyping language is often Python, Ruby, or R. Once the code works, you rewrite the slow parts in C or C++. If you are truly insane, you then rewrite the inner C loops using assembler, CUDA, or OpenCL.

Unfortunately, there's a big wall between the prototyping language and C, and another big wall between C and assembler. Besides having to learn three different languages to get the job done, you have to mentally switch between the layers of abstraction. At a more quotidian level, you have to write a significant amount of glue code, and often find yourself switching between different source files, different code editors, and disparate debuggers.

I read about Julia [julialang.org] a while back, and thought it sounded cool, but not like something I urgently needed. Julia is a dynamic language with great performance. That's nice, I thought, but I've already invested a lot of time putting a Ferrari engine into my VW Beetle — why would I buy a new car? Besides, nowadays a number of platforms — Java HotSpot, PyPy, and asm.js, to name a few — claim to offer "C performance" from a language other than C.

Only later did I realize what makes Julia different from all the others. Julia breaks down the second wall — the wall between your high-level code and native assembly. Not only can you write code with the performance of C in Julia, you can take a peek behind the curtain of any function into its LLVM Intermediate Representation as well as its generated assembly code — all within the REPL. Check it out.

```
emiller ~/Code/julia (master) ./
julia

|  A fresh approach to technical
|  computing Documentation:
|  http://docs.julialang.org
|  Type "help()" to list help
|  topics
|  Version 0.3.0-prerelease+261
|  (2013-11-30)
|  Commit 97b5983 (0 days old
|  master)
|  x86_64-apple-darwin12.5.0

julia> f(x) = x * x
f (generic function with 1 method)

julia> f(2.0)
4.0

julia> code_llvm(f, (Float64,))

define double @julia_f662(double) {
top:
  %1 = fmul double %0, %0, !dbg
!3553
  ret double %1, !dbg !3553
}

julia> code_native(f, (Float64,))
     .section        __TEXT,__
text,regular,pure_instructions
Filename: none
Source line: 1
        push    RBP
        mov     RBP, RSP
Source line: 1
        vmulsd  XMM0, XMM0, XMM0
        pop     RBP
        ret
```

Bam — you can go from writing a one-line function to inspecting its LLVM-optimized X86 assembler code in about 20 seconds.

So forget the stuff you may have read about Julia's type system, multiple dispatch and homoiconi-whatever. That stuff is cool (I guess), but if you're like me, the real benefit is being able to go from the first prototype all the way to balls-to-the-wall multi-core SIMD

performance optimizations without ever leaving the Julia environment.

That, in a nutshell, is why I'm betting on Julia. I hesitate to make the comparison, but it's poised to do for technical computing what Node.js is doing for web development — *getting disparate groups of programmers to code in the same language*. With Node.js, it was the front-end designers and the back-end developers. With Julia, it's the domain experts and the speed freaks. That is a major accomplishment.

Julia's only drawback at this point is the relative dearth of libraries — but the language makes it unusually easy to interface with existing C libraries. Unlike with native interfaces in other languages, you can call C code without writing a single line of C, and so I anticipate that Julia's libraries will catch up quickly. From personal experience, I was able to access 5K lines of C code using about 150 lines of Julia — and no extra glue code in C.

If you work in a technical group that's in charge of a dizzying mix of Python, C, C++, Fortran, and R code — or if you're just a performance-obsessed gun-slinging cowboy shoot-from-the-hip Lone Ranger like me — I encourage you to download Julia and take it for a spin. If you're hesitant to complicate your professional life with Yet Another Programming Language, think of Julia as a tool that will eventually help you reduce the number of languages that your project depends on.

I almost neglected to mention: Julia is actually quite a nice language, even ignoring its excellent performance characteristics. I'm no language aesthete, but learning it entailed remarkably few head-scratching moments. At present Julia is in my top 3 favorite programming languages.

Finally, you'll find an active and supportive Julia community. My favorite part about the community is that it is full of math-and-science types who tend to be very smart and very friendly. That's because Julia was not designed by language geeks — it came from math, science, and engineering MIT students who wanted a fast, practical language to replace C and Fortran. So it's not designed to be beautiful (though it is); it's designed to give you answers quickly. That, for me, is what computing is all about. ■

Evan Miller is the creator of Wizard [wizardmac.com], a next-generation statistics package for Mac.

Reprinted with permission of the original author.
First appeared in *hn.my/julia* (evanmiller.org)

mailjet

EMAIL FOR YOUR APPS

SEND. TRACK. DELIVER.

Your one stop shop for **ALL your email needs.**
Manage lists as well. No extra fees for **Newsletters.**
Priority headers to deliver notifications in **real time.**

Go for mailjet.com

Forever Alone

Why Loneliness Matters In The Social Age

By JONATHAN E. CHEN

Photo credit: *flickr.com/photos/vinothchandar/6646251667*

Loneliness was a problem I experienced most poignantly in college. In the three years I spent at Carnegie Mellon, the crippling effects of loneliness slowly pecked away at my enthusiasm for learning and for life, until I was drowning in an endless depressive haze that never completely cleared until I left Pittsburgh.

It wasn't for lack of trying either. At the warm behest of the orientation counselors, I joined just the right number of clubs, participated in most of the dorm activities, and tried to expand my social portfolio as much as possible.

None of it worked.

I got up and went over and looked out the window. I felt so lonesome, all of a sudden. I almost wished I was dead. Boy, did I feel rotten. I felt so damn lonesome. I just didn't want to hang around anymore. It made me too sad and lonesome.
— J.D. Salinger in Catcher in the Rye

To the extent that I sought out CAPS (our student psych and counseling service) for help, the platitudes they offered as advice ("Just put yourself out there!") only served to confirm my suspicion that loneliness isn't a very visible problem. (After all, the cure for loneliness isn't exactly something that could be prescribed. "Have you considered transferring?" they finally

suggested, after exhausting their list of thought-terminating clichés. I graduated early instead.)

As prolonged loneliness took its toll, I became very unhappy — to put it lightly — and even in retrospect I have difficulty pinpointing a specific cause. It wasn't that I didn't know anyone or failed to make any friends, and it wasn't that I was *alone* more than I liked.

Sure, I could point my finger at the abysmally fickle weather patterns of Pittsburgh, or the pseudo-suburban bubble that envelops the campus. There might even be a correlation between my academic dissonance with computer science and my feelings of loneliness. I might also just be an extremely unlikable person.

For whatever the reason (or a confluence thereof) the reality remained that I struggled with loneliness throughout my time in college.

****

I RECALL A CONVERSATION with my friend Dev one particular evening on the patio of our dormitory. It was the beginning of my junior and last year at CMU, and I had just finished throwing an ice cream party for the residents I oversaw as an RA.

"Glad to be back?" he asked as he plopped down on a lawn chair beside me.

"No, not really."

The sun was setting, and any good feelings about the upcoming semester with it. We made small talk about the school in general, as he had recently transferred, but eventually Dev asked me if I was happy there.

"No, not really."

"Why do you think you're so miserable here?"

"I don't know. A lot of things, I guess. But mostly because I feel lonely. Like I don't belong, like I can't relate to or connect with anyone on an emotional level. I haven't made any quality relationships here that I would look back on with any fond memories. Fuck… I don't know what to do."

College, at least for me, was a harrowing exercise in how helplessly debilitating, hopelessly soul-crushing, and at times life-threatening loneliness could be. It's a problem nobody talks about, and it's been a subject of much personal relevance and interest.

Loneliness as a Health Problem

A recent article published on Slate outlines the hidden dangers of social isolation. Chronic loneliness, as Jessica Olien discovered, poses serious health risks that not only impact mental health but physiological well-being as well.

The lack of quality social relationships in a person's life has been linked to an increased mortality risk comparable to smoking and alcohol consumption and exceeds the influence of other risk factors like physical inactivity and obesity. It's hard to brush off loneliness as a character flaw or an ephemeral feeling when you realize it kills more people than obesity.

Research also shows that loneliness diminishes sleep quality and impairs physiological function, in some cases reducing immune function and boosting inflammation, which increases risk for diabetes and heart disease.

Why hasn't loneliness gotten much attention as a medical problem? Olien shares the following observation:

> *As a culture we obsess over strategies to prevent obesity. We provide resources to help people quit smoking. But I have never had a doctor ask me how much meaningful social interaction I am getting. Even if a doctor did ask, it is not as though there is a prescription for meaningful social interaction.*

As a society we look down upon those who admit to being lonely, we cast and ostracize them with labels like "loners" insofar as they prefer to hide behind shame and doubt rather than speak up. This dynamic only makes it harder to devise solutions to what is clearly a larger societal issue, and it certainly brings to question the effects of culture on our perception of loneliness as a problem.

Loneliness as a Culture Problem

Stephen Fry, in a blog post titled Only the Lonely which explains his suicide attempt last year, describes in detail his struggle with depression. His account offers a rare and candid glimpse into the reality of loneliness with which those afflicted often hide from the public:

> *"Lonely? I get invitation cards through the post almost every day. I shall be in the Royal Box at Wimbledon and I have serious and generous offers from friends asking me to join them in the South of France, Italy, Sicily, South Africa, British Columbia and America this summer. I have two months to start a book before I go off to Broadway for a run of Twelfth Night there.*
>
> *"I can read back that last sentence and see that, bipolar or not, if I'm under treatment and not actually depressed, what the fuck right do I have to be lonely, unhappy or forlorn? I don't*

have the right. But there again I don't have the right not to have those feelings. Feelings are not something to which one does or does not have rights.

"In the end loneliness is the most terrible and contradictory of my problems."

In the United States, approximately 60 million people, or 20% of the population, feel lonely. According to the General Social Survey, between 1985 and 2004, the number of people with whom the average American discusses important matters decreased from three to two, and the number with no one to discuss important matters with tripled.

Modernization has been cited as a reason for the intensification of loneliness in every society around the world, attributed to greater migration, smaller household sizes, and a larger degree of media consumption.

In Japan, loneliness is an even more pervasive, layered problem mired in cultural parochialisms. Gideon Lewis-Kraus pens a beautiful narrative on Harper's in which he describes his foray into the world of Japanese co-sleeping cafés:

"Why do you think he came here, to the sleeping café?"

"He wanted five-second hug maybe because he had no one to hug. Japan is haji culture. Shame. Is shame culture. Or maybe also is shyness. I don't know why. Tokyo people…very alone. And he does not have…" She thought for a second, shrugged, reached for her phone. "Please hold moment."

She held it close to her face, multi-touched the screen not with thumb and forefinger but with tiny forefinger and middle finger. I could hear another customer whispering in Japanese in the silk-walled cubicle at our feet. His co-sleeper laughed loudly, then laughed softly. Yukiko tapped a button and shone the phone at my face. The screen said COURAGE.

It took an enormous effort for me to come to terms with my losing battle with loneliness and the ensuing depression at CMU, and an even greater leap of faith to reach out for help. (That it was to no avail is another story altogether.) But what is even more disconcerting to me is that the general stigma against loneliness and mental health issues, hinging on an unhealthy stress culture, makes it hard for afflicted students to seek assistance at all.

As Olien puts it, "In a society that judges you based on how expansive your social networks appear, loneliness is difficult to fess up to. It feels shameful."

To truly combat loneliness from a cultural angle, we need to start by examining our own fears about being alone and to recognize that as humans, loneliness is often symptomatic of our unfulfilled social needs. Most importantly, we need to accept that it's okay to feel lonely. Fry, signing off on his heartfelt post, offers this insight:

> *"Loneliness is not much written about (my spell-check wanted me to say that loveliness is not much written about — how wrong that is) but humankind is a social species and maybe it's something we should think about more than we do."*

Loneliness as a Technology Problem

Technology, and by extension media consumption in the Internet age, adds the most perplexing (and perhaps the most interesting) dimension to the loneliness problem. As it turns out, technology isn't necessarily helping us feel more connected; in some cases, it makes loneliness worse.

The amount of time you spend on Facebook, as a recent study found, is inversely related to how happy you feel throughout the day.

Take a moment to watch this video: *http://vimeo.com/70534716*

It's a powerful, sobering reminder that our growing dependence on technology to communicate has serious social repercussions, to which Cohen presents his central thesis:

> *We are lonely, but we're afraid of intimacy, while the social networks offer us three gratifying fantasies: 1) That we can put our attention wherever we want it to be. 2) That we will always be heard. 3) That we will never have to be alone.*
>
> *And that third idea, that we will never have to be alone, is central to changing our psyches. It's shaping a new way of being. The best way to describe it is:*
>
> *I share, therefore I am.*

Public discourse on the cultural ramifications of technology is certainly not a recent development, and the general sentiment that our perverse obsession with sharing will be humanity's downfall continues to echo in various forms around the web: articles proclaiming that Instagram is ruining people's lives, the existence of a section on Reddit called cringepics where people congregate to ridicule things others post on the Internet, the increasing number of self-proclaimed "social media gurus" on Twitter, to name a few.

The signs seem to suggest we have reached a tipping point for "social" media that's not very social on a personal level, but whether it means a catastrophic implosion or a gradual return to more authentic forms of interpersonal communications remains to be seen.

While technology has been a source of social isolation for many, it has the capacity to alleviate loneliness as well. A study funded by the online dating site eHarmony shows that couples who met online are less likely to divorce and achieve more marital satisfaction than those who met in real life.

The same model could potentially be applied to friendships, and it's frustrating to see that there aren't more startups leveraging this opportunity when the problem is so immediate and in need of solutions. It's a matter of exposure and education on the truths of loneliness, and unfortunately we're just not there yet.

****

The perils of loneliness shouldn't be overlooked in an increasingly hyper-connected world that often tells another story through rose-tinted lenses. Rather, the gravity of loneliness should be addressed and brought to light as a multifaceted problem, one often muted and stigmatized in our society. I learned firsthand how painfully real of a problem loneliness could be, and more should be done to spread its awareness and to help those affected.

"What do you think I should do?" I looked at Dev as the last traces of sunlight teetered over the top of Morewood Gardens. It was a rhetorical question — things weren't about to get better.

"Find better people," he replied.

I offered him a weak smile in return, but little did I know then how prescient those words were.

In the year that followed, I started a fraternity with some of the best kids I'd come to know (Dev included), graduated college and moved to San Francisco, made some of the best friends I've ever had, and never looked back, if only to remember, and remember well, that it's never easy being lonely. ■

Jonathan E. Chen (*@wikichen*) is a designer based in California. He received his B.S. in computer science from Carnegie Mellon University. In the past he's worked as a front-end developer and interaction designer at various startups. He is currently taking some time off to explore his interests in food and photography and is looking for new opportunities.

Reprinted with permission of the original author.
First appeared in *hn.my/foreveralone* (wikichen.is)

Killing the Crunch Mode Anti-pattern

By CHAD FOWLER

In the software industry, especially the startup world, Crunch Mode is a ubiquitous, unhealthy anti-pattern. Crunch Mode refers to periods of overtime work brought on by the need to meet a project deadline. Developers stereotypically glorify the ability and propensity to stay up all night grinding through a difficult problem. It's part of our folklore. It's part of how we're measured. It's something companies and leaders take advantage of in order to accomplish more with less.

And it's stupid.

If you want a "knowledge worker" to be as ineffective and produce the lowest level of quality possible, deprive them of their sleep and hold them to an unrealistic deadline. In other words, activate Crunch Mode.

Why Not Crunch?

- **It makes us stupid.** The more I work, the less relevant my years of experience become. I constantly make rookie mistakes. I break things in production. I leave messes behind. I waste hours going down the wrong train of thought.
- **It burns people out, sometimes permanently.** They burn up their passion that takes down time to replenish. Unless the non-Crunch work is sufficiently energizing (and frequent), enough crunching can cause your best people to leave.
- **It makes people lazy and less productive.** This may seem ironic, but when someone puts in heroic levels of effort, they start to place less value on each minute. I know that if I work all night, then an hour brain-break mid-day sounds very reasonable. The problem is that these breaks become a habit that can persist between Crunch times.

- **It's a risky way to make your commitments.** Crunch Mode means you are using your team beyond capacity. That's like trying to drive 50km on 40km of gas. It might be OK, but if you do it all the time you're going to end up broken down on the side of the road waiting for help at some point. Maybe more often than not.
- **Accountability is lost.** When someone is working all hours, they can't be blamed for mistakes. They can't be blamed for coming in late, forgetting an email, introducing bugs, not writing tests, cutting technical corners, and doing all sorts of things that don't describe how you want people on your team behaving.
- **It puts the credibility of management in question every time.** Because, managers, believe it or not, every single time it happens, the entire team asks themselves, "But why?"
- **It shows a team that the leader cares about meeting a business goal more than he or she cares about their health.** This may sound harsh, but it is literally true.

The more you have to use your brain, the less effective and healthy Crunch Mode is. In fields that require less creativity and thought, it might even really work as a (ruthless) management technique. In software development, it just doesn't.

Why do we do it?

The number one reason teams go into Crunch Mode is that their leaders have failed to understand and/or set realistic expectations for the time it takes to complete a project. In worst cases, the deadlines are arbitrarily set by management and not tied to any specific business need. In other cases, the deadlines are inflexible, but the scope can and should be adjusted to a realistic level. Sure, it may be that the team committed to those incorrect deadlines, but it's up to the ones deciding on the deadlines to verify that they're realistic before making a commitment.

Fear and the resulting breakdown of communication also drive us into Crunch Mode. "Can you get this done by ?" "Uh…yes?" Developers fear saying "no." Managers fear looking bad by committing to what seem like far off dates. Managers fear setting far off deadlines, because developers miss dates more often than not. "If we pad the estimates are we going to miss those by 20% too?"

Another reason we go into Crunch Mode is that we are perpetuating a culture of cowboy heroism which many of us unwittingly get caught up in. The feeling of finishing tons of work in a short period and depriving oneself of quality personal time can be addicting, especially when it results in "saving the day" for a project. Rolling up your sleeves and cranking to the end of a

deadline makes you feel valuable in a very concrete way. Without your overtime, the project doesn't get done on time. With it, the project is saved. It's hard to find such black and white ways to add value in daily "normal" work.

Maybe the most addictive feature of Crunch Mode is it's the easiest way to see a team really click. At the beginning of Crunch Mode, people get intensely focused. Communication is streamlined. The big important stuff gets tackled quickly and finished. A team can initially raise its skill level a notch with the focus alone. It feels great as both a manager and a team member to work that efficiently and effectively. Unfortunately it's difficult (not impossible) to work this way all the time, so we're tempted to activate Crunch Mode on occasion just to feel this way again.

Alternatives to Crunch-Mode

- **Miss the deadline.** Ya, that's right. Let your customers down this time. Make less money. Incur opportunity cost. Just fail. You already failed to manage your team and your time. Maybe you should let that have more visible consequences?
- **Set smaller goals.** When you set a massive goal, way off in the future, it's impossible to estimate whether it's actually realistic. However, if you set a goal for this afternoon, you're probably going to be pretty accurate with your estimates.
- **Measure progress concretely and in small steps.** Never trust a status report, even from yourself. In software, the only deliverable that matters is one that you can execute.
- **Set more realistic goals for the team and the problems you face.** If you're continually having to slip into Crunch Mode, you clearly don't understand your capabilities. Admit that you're going to go slower than you expected and adjust for it.

As unhealthy, counterproductive, and just plain stupid as Crunch Mode is, sometimes you just have to do it. We all accept that. Crunch Mode is the nuclear option. A leader needs to have it available as a tool, but each time he or she wields this tool, he or she pays in long-term credibility and trust.

Can we stop it?

It's time to finally stop this insanity. Think of the time, money, energy, and potential happiness wasted on poor planning, communication, and leadership.

Managers, hold yourself accountable for Crunch Mode when it happens. See it as a personal failure.

Everyone else, hold yourself accountable for every non-crunch minute you work. Make them count. Over-communicate. Focus. ■

Chad Fowler is an internationally known software developer, trainer, manager, speaker, and musician. Over the past decade he has worked with some of the world's largest companies and most admired software developers. Chad is the author or co-author of a number of popular software books, including "The Passionate Programmer: Creating a Remarkable Career in Software Development".

Reprinted with permission of the original author.
First appeared in *hn.my/crunchmode* (chadfowler.com)